# ASL Reference
# DDI0626

Arm Architecture Technology Group

May 28, 2025

# Contents

# Chapter 1

# Non-Confidential Proprietary Notice

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

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

# Chapter 2

# Disclaimer

This document is part of the ASLRef material, 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, more precisely at Alpha quality as per Arm's quality standards. In particular, this means that it would be premature to base any production tool development on this material. A list of open items being worked on can be found in Appendix A and Appendix B.

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 ALP3.1

The following changes have been made.

### 3.1.1 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`.

### 3.1.2 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.

### 3.1.3 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.

### 3.1.4 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;
```

### 3.1.5 ASL-797: use of `elsif`

The keyword `elsif` is no longer valid in expressions.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

### 3.1.6 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)'
```

### 3.1.7 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]`.

### 3.1.8 ASL-808: standard library additions

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

### 3.1.9 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;
```

### 3.1.10 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}();
```

### 3.1.11 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`.

### 3.1.12 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.

## 3.2 ALP3

The following changes have been made.

### 3.2.1 ASL-625: `configs`

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

### 3.2.2 ASL-705: subprogram overriding

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

### 3.2.3 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;
```

### 3.2.4 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).

### 3.2.5 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'}
```

### 3.2.6 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.

### 3.2.7 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})
```

### 3.2.8 ASL-780: `collections`

Introduce a variant of `record`s 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.

### 3.2.9 ASL-781: name clashing

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

### 3.2.10 ASL-782: evaluation order

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

### 3.2.11 ASL-785: pending constrained global storage elements

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

### 3.2.12 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..2^N-1}`
  - now `UInt{N: integer{1..128}}`
- `SInt`:
  - previously `SInt{N} (x: bits(N)) => integer{-(2^(N-1)) .. 2^(N-1)-1}`
  - now `SInt{N: integer{1..128}}`
- `AlignDownSize`:
  - previously `AlignDownSize{N}(x: bits(N), size: integer {1..2^N}) => bits(N)`
  - now `AlignDownSize{N: integer{1..128}}(x: bits(N), size: integer {1..2^N}) => bits(N)`
- `AlignUpSize`:
  - previously `AlignUpSize{N}(x: bits(N), size: integer {1..2^N}) => bits(N)`
  - now `AlignUpSize{N: integer{1..128}}(x: bits(N), size: integer {1..2^N}) => bits(N) ...`

Removed the following functions:

- ASL-787: `Log2`

## 3.3 ALP2.1

The following changes have been made.

### 3.3.1 ASL-770: documented a taxonomy of ASL errors

See Chapter 37.

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

### 3.3.3 ASL-753: renamed "statically evaluable" to "symbolically evaluable"

### 3.3.4 Updated standard library

Implemented the following functions:

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

### 3.3.5 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;
```

## 3.4 ALP2

The following changes have been made.

### 3.4.1 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`.

### 3.4.2 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
```

### 3.4.3 ASL-677 and ASL-742: `integer{-}` syntax for inherited integer constraints

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

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

### 3.4.4 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.

### 3.4.5 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

### 3.4.6 ASL-629: Define side-effects

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

Summary

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

### 3.4.7 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;<br>print(abc); | 20 | |
| tuple | var a = 1;<br>var b = 2;<br>print((a, b)) | Static error | |

### 3.4.8 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
```

### 3.4.9 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'`

### 3.4.10 ASL-738: Rename `UNKNOWN` to `ARBITRARY`

`UNKNOWN` keyword is renamed to `ARBITRARY`.

### 3.4.11 ASL-637: Dynamic and static errors

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

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

### 3.4.12 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
```

### 3.4.13 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`).

### 3.4.14 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.

### 3.4.15 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"
       | ...
```

### 3.4.16 ASL-596: Remove `Int()` and `IsZeroBit()` in the standard library

### 3.4.17 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.

### 3.4.18 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 $A$:

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

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

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

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

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

**Definition 1 (Sign of a Rational Number)** *The function* $\texttt{sign} : \overbrace{\mathbb{Q}}^{q} \to \{-1, 0, 1\}$ *returns the sign of* $q$*:*

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

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

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

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

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

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

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

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

**Definition 7 (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* $\mathrm{dom}(f)$*, is the subset of* $A$ *for which it is defined. We write* $f(x) = \perp$ *to denote that* $x$ *is not in the domain of* $f$*, that is,* $x \notin \mathrm{dom}(f)$*.*

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

**Definition 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* $\emptyset_\lambda$.

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

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

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

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

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

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

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

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

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

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

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

### 5.1.1 Lists

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

A list of elements is either empty, denoted by $[\,]$, or non-empty. A non-empty list is either denoted by listing the elements in sequence, $v_1 \dots v_k$, 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 a the type of a possibly-empty list of elements of type $T$, and $T^+$ for a non-empty list of elements of type $T$.

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* $\mathit{list\_set} : \mathcal{P}(T) \times T^*$ *lists the elements of a set in an arbitrary order:*

$$\begin{gathered} \mathit{list\_set}(X) = x_{1..k} \\ |X| = k \\ \forall x \in X.\ \exists 1 \leq i \leq k.\ x = x_i \end{gathered}$$

**Definition 17 (List Concatenation)** *The parametric function* $+ : T^* \times T^* \to T^*$ *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* $\mathit{concat} : (T^*)^* \to T^*$ *concatenates a list of lists:*

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

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

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

*compares the length of two lists:*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{rcl} \mathit{unique}(l) & \triangleq & \mathit{unique'}(l, [\,]) \end{array}$$

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

### 5.1.2 Strings

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

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

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

**Definition 25 (Optional Data Type)** *The notation $\langle\cdot\rangle$ stands for either an empty set or a singleton set, where `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 $\{$`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:

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

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

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

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

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

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

variables. For example, the rule TypingRule.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

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

obtained by the following substitutions:

| free variable | value |
|---|---|
| `tenv` | $\emptyset_{\mathbb{SE}}$ |
| `e1` | $\mathsf{E\_Literal}(\mathsf{L\_Int}(2))$ |
| `e1′` | $\mathsf{E\_Literal}(\mathsf{L\_Int}(2))$ |
| `e2` | $\mathsf{E\_Literal}(\mathsf{L\_Int}(3))$ |
| `e2′` | $\mathsf{E\_Literal}(\mathsf{L\_Int}(3))$ |
| `t` | $\mathsf{T\_Int}$ |
| `t1` | $\mathsf{T\_Int}$ |
| `t2` | $\mathsf{T\_Int}$ |
| `op` | $\mathtt{MUL}$ |

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:

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

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

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

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

**Definition 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 $C = L(\ldots)$, denoted `config_dom`$(C)$, is the label $L$.

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

For example, the conclusion of the rule TypingRule.ELit has
*annotate_expr*(tenv, E_Literal(v)) as its input configuration and (t, E_Literal(v)) as its output configuration. Further,
`config_dom`(*annotate_expr*(tenv, E_Literal(v))) = *annotate_expr*, while the output configuration does not have a configuration domain, since it is an unlabelled pair.

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

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

**Definition 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:

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_cond}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_cond} \overset{\text{is}}{=} (\text{Bool}(\texttt{b}), \texttt{g1}) \qquad \texttt{e'} := \text{choice}(\texttt{b}, \texttt{e1}, \texttt{e2}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e'}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g2}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \xrightarrow{\texttt{asl\_ctrl}} \texttt{g2}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_Cond}(\texttt{e\_cond}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

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

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

is the list of premises

$$\begin{array}{l} a[1] > b[1] \\ \ldots \\ a[k] > b[k] \;\; . \end{array}$$

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

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

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

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_cond}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_cond} \overset{\text{is}}{=} (\text{Bool}(\texttt{b}), \texttt{g1}) \\ \mathit{eval\_expr}(\texttt{env1}, \text{choice}(\texttt{b}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g2}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_Cond}(\texttt{e\_cond}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g1} \xrightarrow{\texttt{asl\_ctrl}} \texttt{g2}), \texttt{new\_env})}$$

### 5.2.4 AST-related Notations

When deconstructing AST record nodes such as $\{f_1 : t_1, \ldots, f_k : t_k\}$, we sometimes only care about a subset of the fields $\{f_{i_1}, \ldots, f_{i_m}\} \subset \{f_{1..k}\}$. In such cases, we write $\{f_{i_1} : t_{i_1}, \ldots, f_{i_m} : t_{i_m}, \ldots\}$, where $\ldots$ 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 {body : `body`, args : `arg_decls`, ...} 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 $\texttt{l} \in \mathbb{L}$ 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

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

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

Now, scan the premises in order to see where `env`, `op`, `e1`, and `e2` are used and how premises assign values to `v`, `g`, and `new_env`. In this case, `v` is assigned as the result of the transition assertion $\mathit{binop}(\texttt{op}, \texttt{v1}, \texttt{v2}) \xrightarrow{\mathsf{eval}} \texttt{v}$, `g` is assigned the expression `g1` $\|$ `g2`, and `new_env` is assigned as the result of the transition assertion $\mathit{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{m2}, \texttt{new\_env})$. Notice that to assign values to the variables `v`, `g`, and `new_env`, intermediate values have to be assigned first. For example, $\mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{m1}, \texttt{env1})$ assigned values to `env1`, which is then used by the transition $\mathit{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{m2}, \texttt{new\_env})$. 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 $C \xrightarrow{R} C' \mathbin{/\!/} E$, which introduces alternative output configurations $C'$ and short-circuit $E$:

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

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

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

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

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

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

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

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

Stands for the set of rule macros

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

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

**Alternative Outcomes Expressed in English Prose:** In English prose, we use $/\!\!/_{x,y,\ldots}$ to mean "if the outcome is one of $x, y, \ldots$ then the result short-circuits the rule.

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

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

In this rule, #T and #DE are just shorthand notations for actual configurations, which are properly defined in 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 — Normal((v, g), new_env) — is achieved. Then, re-reading the rule would indicate where exceptions and dynamic errors may result in other output configurations. For example, if the first transition assertion results in a throwing configuration #T then the output configuration of the conclusion is also #T. This corresponds to the following rule in the expanded macro:

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

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

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

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

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

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

Expanding the last transition assertion, gives us the case:

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

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

### 5.2.7 Boolean Transition Assertions

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

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

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

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

### 5.2.8 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) $f : A \bigtimes B$ and applies it to an optional value $A?$:

$$\mathsf{optional}[\cdot] : \overbrace{A?}^{\texttt{v\_opt}} \bigtimes \overbrace{B?}^{\texttt{v\_opt\_new}}$$

#### 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

SOME

$$\frac{f(v) \longrightarrow v'}{\mathsf{optional}[f](\overbrace{\langle v \rangle}^{\texttt{v\_opt}}) \longrightarrow r\langle v' \rangle}$$

NONE

$$\mathsf{optional}[f](\overbrace{\mathsf{None}}^{\texttt{v\_opt}}) \longrightarrow \mathsf{None}$$

### 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:

T_BOOL

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Bool}}^{\texttt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{FALSE}))}^{\texttt{e\_init}}$$

T_REAL

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Real}}^{\texttt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Real}(0))}^{\texttt{e\_init}}$$

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 ↪ 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:

<ascii_char> ≜ ASCII{0-255}

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

<char> ≜ ASCII{10} | ASCII{13} | ASCII{32-126}

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

Table 6.1: Lexical Regular Expressions

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

## 6.3 Whitespace

**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> ≜ (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 FOO 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 FOO : integer = 1; // The declaration of FOO 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> ≜ "//" (<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
<int_lit> ≜ <digit> ( _ | <digit>)*
<hex_lit> ≜ "0x" (<digit> | abcdefABCDEF) ( _ | <digit> | abcdefABCDEF)*

## 6.6 Real Number Literals

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

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

This is formalized by the following lexical regular expression:

<real_lit> ≜ <digit> ( _␣ | <digit>)* "." <digit> ( _␣ | <digit>)*

## 6.7 Boolean Literals

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

## 6.8 Bitvector Literals

Constant 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'`.

## 6.9 Bitmasks

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

<bitmask_lit> ≜ '␣ ( <bit> | x␣ | (␣ <bit>+ )␣ )* '␣

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

Table 6.2: Escape Sequences in String Literals

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

## 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.

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

## 6.11 Identifiers

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

```
<letter>     ≙  abcdefghijklmnopqrstuvwxyz | ABCDEFGHIJKLMNOPQRSTUVWXYZ
<identifier> ≙  (<letter> | _) (<letter> | _ | <digit>)*
```

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

$$\mathit{scan} : \mathsf{LexSpec} \times \texttt{<ascii\_char>}^* \longrightarrow (\mathbb{TOKEN}^* \cup \{\texttt{\#BE\_LE}\})$$

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.

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

- Tokens of the form $\mathbf{INT\_LIT}(n)$ represent integer literals;
- Tokens of the form $\mathbf{REAL\_LIT}(q)$ represent real literals;
- Tokens of the form $\mathbf{STRING\_LIT}(s)$ represent string literals;
- Tokens of the form $\mathbf{STRING\_CHAR}(c)$ represent a single character in a string literal;
- The token **STRING_END** represents the closing quotes of a string literal;
- Tokens of the form $\mathbf{BITVECTOR\_LIT}(b)$ represent bitvector literals;
- Tokens of the form $\mathbf{MASK\_LIT}(m)$ represent constant bitmasks;
- Tokens of the form $\mathbf{BOOL\_LIT}(b)$ represent Boolean literals;
- Tokens of the form $\mathbf{ID}(i)$ 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* $[(r_1, a_1), \ldots, (r_k, a_k)] \in \mathit{LexSpec}$ *where each pair* $(r_i, a_i)$ *consists of a lexical regular expression* $r_i$ *and a* lexeme action $a_i : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{TOKEN}^*$.

The function

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

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

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

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

$$\textsc{no\_match}\quad \frac{\mathit{max\_matches}(R, s) = [\,]}{\mathit{scan}(R, s) \xrightarrow{\mathsf{scan}} \texttt{\#BE\_LE}}$$

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

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

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

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

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

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

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

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

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

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

- The lexeme action

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

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

- The lexeme action

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

  switches to scanning literal strings via *scan_string*.

- The lexeme action

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

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

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

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

- The function $\mathit{real\_to\_lit}(s)$ returns **REAL_LIT**$(q)$ where $q$ is the rational number for $s$, which is given via a floating point representation.

- The function $\mathit{str\_to\_lit}(s)$ returns **STRING_LIT**$(s')$ where $s'$ is the string value represented by $s$.

- The function $\mathit{bits\_to\_lit}(s)$ returns **BITVECTOR_LIT**$(b)$ where $b$ is the sequence of bits given by $s$.

- The function $\mathit{mask\_to\_lit}(s)$ returns **MASK_LIT**$(m)$ where $m$ is the bitmask given by $s$.

- The function $\mathit{false\_to\_lit}(s)$ returns **BOOL_LIT**(`FALSE`) ($s$ is ensured to be `FALSE`).

- The function $\mathit{true\_to\_lit}(s)$ returns **BOOL_LIT**(`TRUE`) ($s$ is ensured to be `TRUE`).

- The function $\mathit{token\_id}(s)$ returns **LEXEME**$(s)$.

- The function *lexical_error* returns **T_ERR**.

- The lexeme action

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

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

### LexicalRule.ReservedIdentifiers

The function $\mathit{to\_identifier}(s)$ 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**

$$\textit{to\_identifier}(s) = \begin{cases} \texttt{\#BE\_LE} & \text{if } s = \texttt{\_\_}\,s' \\ \textbf{ID}(s) & \text{else} \end{cases}$$

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

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

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

| Lexical Regular Expressions | Lexeme Action |
|---|---|
| "pure", "readonly" | *lexical_error* |

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

### 6.12.3 Scanning Strings

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

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

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

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

CHAR

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

END

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

We also employ the following lexeme actions:

- The lexeme action

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

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

- The lexeme action

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

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

- The lexeme action

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

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

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

| Lexical Regular Expression | Lexeme Action |
|---|---|
|  | *string_escape* |
|  | *string_escape* |
|  | *string_escape* |
|  | *string_escape* |
|  | *string_finish* |
| <char> | *string_char* |

### 6.12.4 Scanning Multi-line Comments

The lexeme action

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

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

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

| Lexical Regular Expression | Lexeme Action |
|---|---|
| "*/" | *discard* |
| <char> | *discard_comment_char* |

# Chapter 7

# Syntax

This chapter defines the grammar of ASL. The grammar is presented via two extensions to context-free grammars — *inlined derivations* and *parametric productions*, inspired by the Menhir Parser Generator [7] for the OCaml language. Those extensions can be viewed as macros over context-free grammars, which can be expanded to yield a standard context-free grammar.

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

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

The outline of this chapter is as follows:

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

## 7.1 Inlined Derivations

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

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

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

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

For example, consider the derivation

```
expr ⟶ expr binop expr
```

coupled with the derivation

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

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

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

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

```
binop ⟶(inline) "AND" | "&&" | "||" | "<->" | "DIV" | "DIVRM" | "XOR" | "==" | "!="
      | ">" | ">=" | "->" | "<" | "<=" | "+" | "-" | "MOD" | "*"
      | "OR" | "/" | "«" | "»" | "^" | "::"
```

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

## 7.2 Parametric Productions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 7.3 ASL Parametric Productions

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

### Optional Symbol

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

### Possibly-empty List

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

### Non-empty List

$\text{list1}(x) \longrightarrow x \mid x\ \text{list1}(x)$

### Non-empty Comma-separated List

$\text{clist1}(x) \longrightarrow x \mid x\ \texttt{","}\ \text{clist1}(x)$

### Possibly-empty Comma-separated List

$\text{clist0}(x) \longrightarrow \epsilon \mid \text{clist1}(x)$

### Comma-separated List With At Least Two Elements

$\text{clist2}(x) \longrightarrow x\ \texttt{","}\ \text{clist1}(x)$

### Possibly-empty Parenthesized, Comma-separated List

$\text{plist0}(x) \xrightarrow{\text{inline}} \texttt{"("}\ \text{clist0}(x)\ \texttt{")"}$

### Parenthesized Comma-separated List With At Least Two Elements

$\text{plist2}(x) \xrightarrow{\text{inline}} \texttt{"("}\ x\ \texttt{","}\ \text{clist1}(x)\ \texttt{")"}$

### Non-empty Comma-separated Trailing List

$$\begin{aligned}\text{tclist1}(x) \longrightarrow\ & x\ \text{option}(\texttt{","}) \\ \mid\ & x\ \texttt{","}\ \text{tclist1}(x)\end{aligned}$$

### 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 ⟶ list*(decl)

decl ⟶ override "func" ID params_opt func_args return_type recurse_limit
         ↪ func_body
       | override "func" ID params_opt func_args func_body
       | override "accessor" ID params_opt func_args "<=>" ID as_ty
         ↪ accessor_body
       | "type" ID "of" ty_decl subtype_opt ";"
       | "type" ID subtype ";"
       | global_let_or_constant ID option(":" ty)
         ↪ "=" expr ";"
       | "config" ID ":" ty "=" expr ";"
       | "var" ID option(":" ty) "=" expr ";"
       | "var" ID ":" ty ";"
       | "pragma" ID clist0(expr) ";"

recurse_limit ⟶ "recurselimit" expr
                | ϵ

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

```
subtype_opt ⟶ option(subtype)

typed_identifier ⟶ ID as_ty

opt_typed_identifier ⟶ ID option(as_ty)

as_ty ⟶ ":" ty

return_type ⟶ "=>" ty

params_opt ⟶ ϵ
             | "{" clist0(opt_typed_identifier) "}"

call ⟶ ID plist0(expr)
       | ID "{" clist1(expr) "}"
       | ID "{" clist1(expr) "}" plist0(expr)

elided_param_call ⟶ ID "{" "}"
                    | ID "{" "}" plist0(expr)
                    | ID "{" "," clist1(expr) "}"
                    | ID "{" "," clist1(expr) "}" plist0(expr)

func_args ⟶ plist0(typed_identifier)

maybe_empty_stmt_list ⟶ ϵ | stmt_list

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

ignored_or_identifier ⟶ "-" | **ID**

accessor_body ⟶ "begin" accessors "end" ";"

accessors ⟶ "getter" maybe_empty_stmt_list "end" ";"
↪"setter" maybe_empty_stmt_list "end" ";"
| "setter" maybe_empty_stmt_list "end" ";"
↪"getter" maybe_empty_stmt_list "end" ";"

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

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

local_decl_keyword_non_var ⟶ "let" | "constant"

global_let_or_constant ⟶ "let" | "constant"

direction ⟶ "to" | "downto"

case_alt_list ⟶ clist1(case_alt)

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

otherwise_opt ⟶ "otherwise" "=>" stmt_list
| $\epsilon$

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


loop_limit ⟶ "looplimit" expr
           | ϵ
```

```
stmt ⟶ "if" expr "then" stmt_list s_else "end" ";"
       | "case" expr "of" case_alt_list "end" ";"
       | "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_non_var decl_item option(as_ty) "=" expr ";"
       | lexpr "=" expr ";"
       | call setter_access "=" expr ";"
       | call setter_access slices "=" expr ";"
       | call "." "[" clist2(ID) "]" "=" expr ";"
       | local_decl_keyword_non_var decl_item as_ty "=" elided_param_call ";"
       | "var" decl_item option(as_ty) option("=" expr) ";"
       | "var" clist2(ID) as_ty ";"
       | "var" decl_item as_ty "=" elided_param_call ";"
       | "print" plist0(expr) ";"
       | "println" plist0(expr) ";"
       | "Unreachable" "(" ")" ";"
       | "repeat" stmt_list "until" expr loop_limit ";"
       | "throw" expr ";"
       | "throw" ";"
       | "pragma" ID clist0(expr) ";"

stmt_list ⟶ list1(stmt)

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

```
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

setter_access ⟶ ϵ
                | "." 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 ⟶ ID
            | plist2(ignored_or_identifier)

constraint_kind_opt ⟶ constraint_kind | ϵ

constraint_kind ⟶ "{" 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                                   precedence: UNOPS
               | "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 ")"

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) "}"

fields_opt ⟶ fields | ϵ

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_opt
        | "collection" fields_opt
        | "exception" fields_opt


field_assign ⟶ ID "=" expr
```

```
expr ⟶ 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 ⟶ INT_LIT
        | BOOL_LIT
        | REAL_LIT
        | BITVECTOR_LIT
        | STRING_LIT

unop ⟶(inline) "!" | "-" | "NOT"

binop ⟶(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 —* $\epsilon$*.*
- *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* $N \longrightarrow l_{1..k}$ *where for each* $i \in 1..k$ *either:*

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

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

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

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

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

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

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

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

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

## 7.6 Priority and Associativity

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

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

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

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

| **Terminals** | **Associativity** |
|---|---|
| `"else"` | nonassoc |
| `"\|\|"`, `"&&"`, `"->"`, `"as"` | left |
| `"=="`, `"!="` | left |
| `">"`, `">="`, `"<"`, `"<="` | nonassoc |
| `"+"`, `"-"`, `"OR"`, `"XOR"`, `"AND"`, `"::"` | left |
| `"*"`, `"DIV"`, `"DIVRM"`, `"/"`, `"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 $(n_1, \ldots, n_k)$;

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

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

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

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

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

## 8.2 Abstract Syntax Grammar

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

**Non-terminal.** A non-terminal variable;

**Terminal.** A lexical token $t$ or a label $L$;

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

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

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

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

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

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

## 8.3 Untyped Abstract Grammar

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

### 8.3.1 Identifiers

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

### 8.3.2 Literal Values

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

literal $\longrightarrow$ L_Int($\overset{\mathbb{Z}}{\square}$)

| L_Bool($\overset{\mathbb{B}}{\square}$)

| L_Real($\overset{\mathbb{Q}}{\square}$)

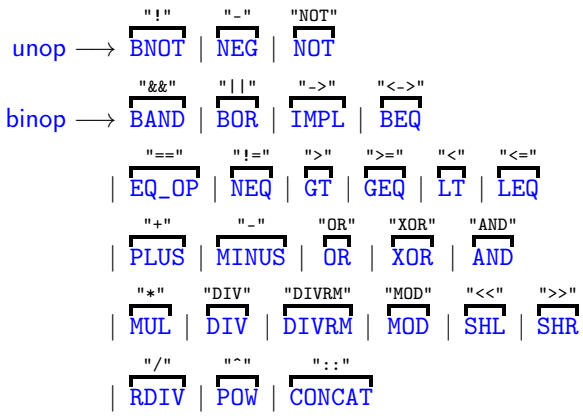
| L_Bitvector( $\overset{B \in \{0,1\}^*}{\square}$ )

| L_String( $\overset{S \in \{C \mid \text{"}C\text{"} \in \mathbb{S}\}}{\square}$ )

| L_Label( $\overset{\text{enumeration label}}{\square}$ )

### 8.3.3 Basic Operations

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




### 8.3.4 Expressions

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

matching expressions.

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

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 binay AND expression.
  // E_Concat 1: b0 :: b1 concatenates two bitvectors.
  // E_Arbitrary 1: ARBITRARY: bits(8) represents an arbitrary
  //                8-bits bitvector
  var b8 = b0 :: b1;
  b8 = (NOT b8) AND ARBITRARY: bits(8);
  return 0;
end;
```

Listing 8.2: More examples of expressions

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

  setter
    Unreachable();
  end;
end;

accessor g1_bits(p: integer) <=> value_in: bits(4)
begin
  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 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

$$
\begin{aligned}
\text{pattern} \longrightarrow\ & \text{Pattern\_All} \\
\mid\ & \text{Pattern\_Any}(\text{pattern}^*) \\
\mid\ & \text{Pattern\_Geq}(\text{expr}) \\
\mid\ & \text{Pattern\_Leq}(\text{expr}) \\
\mid\ & \text{Pattern\_Mask}(\overbrace{\{0,1,x\}^*}^{\text{mask constant}}) \\
\mid\ & \text{Pattern\_Not}(\text{pattern}) \\
\mid\ & \text{Pattern\_Range}(\overbrace{\text{expr}}^{\text{lower}}, \overbrace{\text{expr}}^{\text{upper}}) \\
\mid\ & \text{Pattern\_Single}(\text{expr}) \\
\mid\ & \text{Pattern\_Tuple}(\text{pattern}^*)
\end{aligned}
$$

### 8.3.6 Slices

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

### 8.3.7 Subprogram calls

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

### 8.3.8 Types

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

### 8.3.9 Constraints

$$
\begin{aligned}
\mathit{constraint\_kind} \longrightarrow\ & \mathit{Unconstrained} \\
\mid\ & \mathit{WellConstrained}(\mathit{int\_constraint}^+) \\
\mid\ & \mathit{PendingConstrained} \\
\mid\ & \mathit{Parameterized}(\overbrace{\mathit{identifier}}^{\text{parameter}})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{int\_constraint} \longrightarrow\ & \mathit{Constraint\_Exact}(\mathit{expr}) \\
\mid\ & \mathit{Constraint\_Range}(\overbrace{\mathit{expr}}^{\text{start}}, \overbrace{\mathit{expr}}^{\text{end}})
\end{aligned}
$$

### 8.3.10 Bit Fields

$$
\begin{aligned}
\mathit{bitfield} \longrightarrow\ & \mathit{BitField\_Simple}(\mathit{identifier}, \mathit{slice}^*) \\
\mid\ & \mathit{BitField\_Nested}(\mathit{identifier}, \mathit{slice}^*, \mathit{bitfield}^*) \\
\mid\ & \mathit{BitField\_Type}(\mathit{identifier}, \mathit{slice}^*, \mathit{ty})
\end{aligned}
$$

### 8.3.11 Array Indices

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

array_index ⟶ ArrayLength_Expr( array length: expr )

### 8.3.12 Fields and Typed Identifiers

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

field ⟶ (identifier, ty)

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

typed_identifier ⟶ (identifier, ty)

### 8.3.13 Left-hand Side Expressions

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

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

### 8.3.14 Local Declarations

local_decl_keyword ⟶ LDK_Var | LDK_Constant | LDK_Let

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

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

the local declaration item is

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

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

### 8.3.15 Statements

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

stmt $\longrightarrow$ S_Pass
| S_Seq(stmt, stmt)
| S_Decl(local_decl_keyword, local_decl_item, ty?, expr?)
| S_Assign(lexpr, expr)
| S_Call(call)
| S_Return(expr?)
| S_Cond(expr, stmt, stmt)
| S_Assert(expr)
| S_For $\left\{ \begin{array}{rcl} \text{index\_name} & : & \text{identifier}, \\ \text{start\_e} & : & \text{expr}, \\ \text{dir} & : & \text{for\_direction}, \\ \text{end\_e} & : & \text{expr}, \\ \text{body} & : & \text{stmt}, \\ \text{limit} & : & \text{expr}? \end{array} \right\}$
| S_While( $\overset{\text{condition}}{\text{expr}}$ , $\overset{\text{loop limit}}{\text{expr}?}$ , $\overset{\text{loop body}}{\text{stmt}}$ )
| S_Repeat( $\overset{\text{loop body}}{\text{stmt}}$ , $\overset{\text{condition}}{\text{expr}}$ , $\overset{\text{loop limit}}{\text{expr}?}$ )
// *The option represents an implicit throw:* `throw;`.
| S_Throw(expr?)

| S_Try(stmt, catcher*, $\overset{\text{otherwise}}{\text{stmt}?}$ )
| S_Print($\overset{\text{args}}{\text{expr}^*}$, $\overset{\text{newline}}{\mathbb{B}}$ )
| S_Pragma(**ID**, $\overset{\text{args}}{\text{expr}^*}$)
| S_Unreachable

### 8.3.16 Case Alternatives

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

### 8.3.17 Exception Catchers

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

### 8.3.18 Subprograms

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

$$\text{override\_info} \longrightarrow \text{Impdef} \mid \text{Implementation}$$

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

### 8.3.19 Global Declarations

Declaration keyword for global storage elements:

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

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

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

### 8.3.20 Specifications

spec $\longrightarrow$ decl$^*$

## 8.4 Typed Abstract Syntax Grammar

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

The rules for expressions have the following extra derivation rules:

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

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

lexpr $\longrightarrow$ LE_SetEnumArray(base: lexpr, index: expr)
| LE_SetCollectionFields(collection: identifier, field names: identifier$^*$)

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

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

$$\text{stmt} \longrightarrow \text{S\_Throw}((\text{expr}, \overset{\text{exception type}}{\boxed{\text{ty}}})?)$$

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:

$$\text{array\_index} \longrightarrow \text{ArrayLength\_Enum}(\overset{\text{name of enumeration}}{\boxed{\text{identifier}}}, \overset{\text{enumeration labels}}{\boxed{\text{identifier}^+}})$$

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).

$$\begin{aligned}\text{precision\_loss\_indicator} &\longrightarrow \text{Precision\_Full}\\ &\mid \text{Precision\_Lost}\\ \text{constraint\_kind} &\longrightarrow \text{WellConstrained}(\text{int\_constraint}^+, \text{precision\_loss\_indicator})\end{aligned}$$

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

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

## 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 $N \longrightarrow R_1 \mid \ldots R_k$, we define a builder function build_$N$ which takes a parse tree $\mathbb{PARSE}[N]$ and returns the corresponding AST or a *build error*

*configuration* `#BE` ∈ TBuildError. The builder function is defined in terms of one inference rule per alternative $R_i$. The input for the builder for an alternative $R = S_{1..m}$ is a parse node $N(S_{1..m})$. To allow the builder to refer to the children nodes of $N$, we use the notation $n_i : S_i$, which names the child node $S_i$ as $n_i$.

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

### 8.5.1 Example

Consider the derivation for while loops:

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

The parse node for a while statement has the form

`stmt`(`"while"`, `e` : `expr`, `"do"`, `stmt_list` : `stmt_list`, `"end"`, `";"`)

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

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

We also employ the following rule:

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

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

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

### 8.5.2 Abbreviated Rule Notation for AST Builders

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

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

In our example, this results in the abbreviated rule notation

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

## 8.6 Building Parameterized Productions

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

- ASTRule.List
- 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

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

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

EMPTY

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

NON_EMPTY

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

### ASTRule.CList

The meta relation

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

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

EMPTY

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

NON_EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast} \qquad \mathit{build\_clist}[b](\texttt{syms1}) \xrightarrow{\text{ast}} \texttt{sym\_asts1}}{\mathit{build\_clist}[b](\overbrace{\mathsf{clist0}(N)(\texttt{v} : E, \texttt{","}, \texttt{syms1} : \mathsf{clist1}(N))}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{v\_ast}] + \texttt{sym\_asts1}}^{\texttt{sym\_asts}}}$$

### ASTRule.PList

The meta relation

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

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

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

### ASTRule.NTCList

The meta relation

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

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

EMPTY

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

NON_EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast} \qquad \mathit{build\_tclist}[b](\texttt{syms1}) \xrightarrow{\text{ast}} \texttt{sym\_asts1}}{\mathit{build\_tclist}[b](\overbrace{\texttt{v} : E, \texttt{","}, \texttt{syms1} : \mathsf{tclist1}(N)}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{v\_ast}] + \texttt{sym\_asts1}}^{\texttt{sym\_asts}}}$$

### ASTRule.Option

The meta relation

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

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

NONE

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

SOME

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

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

LAST

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

### ASTRule.Identity

The meta function

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

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

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

## 8.7 Transforming Assignable Expressions to Expressions

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

| Left hand side expression | | Right hand side expression |
|---|---|---|
| $rexpr$(LE_Var(x)) | = | E_Var(x) |
| $rexpr$(LE_Slice(le, args)) | = | E_Slice($rexpr$(le), args) |
| $rexpr$(LE_SetArray(le, e)) | = | E_GetArray($rexpr$(le), e) |
| $rexpr$(LE_SetEnumArray(le, e)) | = | E_GetEnumArray($rexpr$(le), e) |
| $rexpr$(LE_SetField(le, x)) | = | E_GetField($rexpr$(le), x) |
| $rexpr$(LE_SetFields(le, x)) | = | E_GetFields($rexpr$(le), x) |
| $rexpr$(LE_Discard) | = | E_Var(-) |
| $rexpr$(LE_Destructuring([le$_{1..k}$])) | = | E_Tuple([$i = 1..k$ : $rexpr$(le$_i$)]) |

## 8.8 Abstract Syntax Abbreviations

We employ the following abbreviations for various AST nodes:

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

# 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* $\mathbb{SE}$*, are defined as follows (referring to symbols defined by the abstract syntax):*

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

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

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

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

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

The intuitive meaning of each component is as follows:

- declared_types assigns types to their declared names and time frame (the maximal time frame of any side effect descriptor inferred for the type definition);
- constant_values assigns literals to their declaring (constant) names;
- global_storage_types associates names of global storage elements to their inferred type and how they were declared — as constants, configuration variables, `let` variables, or mutable variables;
- local_storage_types associates names of local storage elements to their inferred type and how they were declared — as variables, constants, or as `let` variables;
- expr_equiv associates names of immutable storage elements to a simplified version of their initializing expression;
- subtypes associates type names to the names that their type subtypes;
- subprograms associates names of subprograms to the func AST node they were declared with and the set of side effect descriptors inferred for them;

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

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

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

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

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

## 9.2 Typing Rule Configurations

The output configurations of type system assertions have two flavors:

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

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

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

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

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

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

# Chapter 10

# Dynamic Semantics Definitions

The dynamic semantics of ASL define all valid behaviors of a given ASL specification. More precisely, an ASL specification is first parsed into an *abstract syntax tree*, or AST, for short. Second, a 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 `NV_Literal`(`v`) is a native value;
- TUPLE VALUES AND ARRAY VALUES: if `l` is a list of native values then `NV_Vector`(`l`) is a native value;
- RECORD VALUES: if `r` is a finite function from identifiers to native values then `NV_Record`(`r`) is a native value.

**Formally**

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

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

(Tuple Values and Array Values)

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

(Record Values)

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

We define the following shorthand notations for native value literals:

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

We define the following types of native values:

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

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

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

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

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

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

We assume that the static environment supports the following functions:

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

The partial function *find_func* returns the typed AST of the subprogram for a given identifier. (Recall that ASL allows subprogram overloading so a name does not uniquely identify a specific subprogram. However, the typechecker renames each function uniquely so that it can be accessed based on its name alone.) The function *type_satisfies*(`t`, `s`) returns true if the type `t` *type-satisfies* the type `s` (see 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 $\mathtt{env} \in \mathbb{E}$ to range over environments. From the perspective of the semantics, the static environment is immutable. That is, all environments share the same static environment.

### 10.5.1 Concurrent Execution Graphs

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

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

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

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

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

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

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

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

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

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

**Definition 39 (Read/Write Effects)**

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

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

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

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

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

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

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

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

### 10.5.2 Concurrent Values

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

### 10.5.3 Kinds of Semantic Configurations

Recall that the ASL dynamic semantics define a relation between input semantic configurations and output semantic configurations (Section 10.4). Input semantic configuration domains are unique to the semantic relation that employs them. For that reason, we name semantic relations by the name of the corresponding configuration domain of the input semantic configuration. For example, the semantic relation that employs input semantic configurations with the domain `eval_expr` is named `eval_expr`. We will often use the prefix `eval` for semantic relations with the intuition being that their input semantic configurations should be semantically evaluated, yielding an output semantic configuration.

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

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

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

**Exceptions.** Semantic configurations in

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

represent thrown exceptions.

There are two flavors of exceptions: exceptions without an exception value (as in `throw;`), and ones with an exception value, an identifier to which the Read Effect is attributed, and an associated type. The type $\mathsf{value\_read\_from}(\mathbb{V}, \mathbb{I})$ is a semantic configuration nested inside an exception semantic configuration. The ASL dynamic semantics propagate these *exceptional semantic configurations* to the nearest catch clause that matches them, and otherwise they are caught at the top-level and reported as errors (see dynamic errors below).

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

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

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

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

We define the following shorthand notations for types of output semantic configurations:

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

We will say that a semantic transition *terminates*:

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

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

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

### 10.5.4 Extracting and Substituting Elements of Semantic configurations

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

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

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

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

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

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

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

## 10.6 Semantic Evaluation

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

[1]The 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

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

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

We annotate subexpressions to allow referring to them.

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

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

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

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e1}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(1), \emptyset_{\mathbb{E}}) \;\; 15.1.4 \\ \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e2}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(2), \emptyset_{\mathbb{E}}) \;\; 15.1.4 \\ \mathit{binop}(\texttt{PLUS}, \texttt{Int}(1), \texttt{Int}(2)) \xrightarrow{\text{eval}} \texttt{Int}(3) \;\; 12.5 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e1\_2}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(3), \emptyset_{\mathbb{E}})} \; 15.3.4$$

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e4}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(4), \emptyset_{\mathbb{E}}) \;\; 15.1.4 \\ \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e5}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(5), \emptyset_{\mathbb{E}}) \;\; 15.1.4 \\ \mathit{binop}(\texttt{PLUS}, \texttt{Int}(4), \texttt{Int}(5)) \xrightarrow{\text{eval}} \texttt{Int}(9) \;\; 12.5 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e45}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(9), \emptyset_{\mathbb{E}})} \; 15.3.4$$

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e1\_2}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(3), \emptyset_{\mathbb{E}}) \\ \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e45}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(9), \emptyset_{\mathbb{E}}) \\ \mathit{binop}(\texttt{MUL}, \texttt{Int}(3), \texttt{Int}(9)) \xrightarrow{\text{eval}} \texttt{Int}(27) \;\; 12.5 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \mathsf{E\_Binop}(\texttt{MUL}, \texttt{e1\_2}, \texttt{e45})) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{Int}(27), \emptyset_{\mathbb{E}})} \; 15.3.4$$

### 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:

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \texttt{e\_cond}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{m\_cond}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_cond} \overset{\text{is}}{=} (\mathsf{Bool}(\texttt{b}), \texttt{g1}) \qquad \texttt{e'} := \mathsf{choice}(\texttt{b}, \texttt{e1}, \texttt{e2}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e'}) \xrightarrow{\text{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g2}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \xrightarrow{\text{asl\_ctrl}} \texttt{g2} \end{array}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\mathsf{E\_Cond}(\texttt{e\_cond}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

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

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

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

For these constructs, ASL defines an evaluation order. When there are multiple feasible choices for the next evaluation transition, ASL is defined to take the 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
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
  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 ⟶ **INT_LIT**
| **BOOL_LIT**
| **REAL_LIT**
| **BITVECTOR_LIT**
| **STRING_LIT**

## 11.2 Abstract Syntax

$$
\begin{aligned}
\mathsf{literal} \longrightarrow\ & \mathsf{L\_Int}(\overset{\mathbb{Z}}{\square}) \\
|\ & \mathsf{L\_Bool}(\overset{\mathbb{B}}{\square}) \\
|\ & \mathsf{L\_Real}(\overset{\mathbb{Q}}{\square}) \\
|\ & \mathsf{L\_Bitvector}(\overset{B \in \{0,1\}^*}{\square}) \\
|\ & \mathsf{L\_String}(\overset{S \in \mathbb{S}}{\square}) \\
|\ & \mathsf{L\_Label}(\overset{\text{enumeration label}}{\square})
\end{aligned}
$$

### 11.2.1 ASTRule.Value

The function

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

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

INTEGER

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

BOOLEAN

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

REAL

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

BITVECTOR

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

STRING

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

## 11.3 Typing

### 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___12345567890;
           // 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

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

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

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 `t` as the real type.

- All of the following apply (STRING):
  - `l` is a string literal;
  - define `t` as the string type.

- All of the following apply (BITS):
  - `l` is a bitvector literal of length `n`;
  - define `t` as the bitvector type of fixed width `n`.

- All of the following apply (LABEL):
  - `l` is an enumeration label for `label`;
  - define `t` as the type to which `label` is bound in the declared_types map of the global environment tenv.

**Formally**

INT

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

BOOL

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

REAL

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

STRING

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

BITS

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

LABEL

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

## 11.4 Semantics

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

**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

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

### ASTRule.Unop

The function

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

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

BNOT

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

NEG

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

NOT

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

### ASTRule.Binop

The function

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 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_BO`). 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

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

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:

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

### 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*(`l`)) is not in *unop_signatures*;
    * the result is a type error indicating that the combination of `op` and *ast_label*(`l`) is not legal.

- All of the following apply (NEGATE_INT):
    * `op` is `NEG` and `l` is an integer literal for `n`;
    * define `r` as the integer literal for $-$`n`.
- All of the following apply (NEGATE_REAL):
    * `op` is `NEG` and `l` is a real literal for `q`;
    * define `r` as the real literal for $-$`q`.
- All of the following apply (NOT_BOOL):
    * `op` is `BNOT` and `l` is a Boolean literal for `b`;
    * define `r` as the Boolean literal for $\neg$`b`.
- All of the following apply (NOT_BITS_EMPTY, NOT_BITS_NOT_EMPTY):
    * `op` is `NOT` and `l` is a bitvector literal for the sequence of bits `bits`;
    * `c` is the sequence of bits of the same length as `bits` where in each position the bit in `r` is defined as the negation of the bit of `bits` in the same position;
    * define `r` as the bitvector literal for `c`.

**Formally**

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

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

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

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

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

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

### TypingRule.BinopLiterals

The function

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

statically evaluates a binary operator `op` (a terminal derived from the AST non-terminal for binary operators) over a pair of literals `l1` and `l2` and returns the resulting literal `r`. The result is a type error, if it is illegal to apply the operator to the given values, or a different kind of type error is detected.

### 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("shiftleft_int: 1 << 10 = ", 1 << 10);
    println("shiftleft_int: 1 << 0 = ", 1 << 0);
    println("shiftleft_int: -1 << 10 = ", -1 << 10);
    // println("invalid", "shiftleft_int: 1 << -10 = ", 1 << -10);
    println("shiftright_int: 1 >> 10 = ", 1 >> 10);
    println("shiftright_int: 16 >> 2 = ", 16 >> 2);
    println("shiftright_int: -16 >> 2 = ", -16 >> 2);
    println("shiftright_int: 1 >> 0 = ", 1 >> 0);
    println("shiftright_int: -1 >> 10 = ", -1 >> 10);
    // println("invalid", "shiftright_int: 1 >> -10 = ", 1 >> -10);
    return 0;
end;
```

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

### Example: Integer Comparison

Listing 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:

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

**Prose**

One of the following applies:

- All of the following apply (ERROR):
  - (`op`, *ast_label*(`l1`), *ast_label*(`l2`)) is not included in *binop_signatures*;
  - the result is a type error indicating the `op` cannot be applied to the arguments with the types given by *ast_label*(`l1`) and *ast_label*(`l2`).
- All of the following apply (ADD_INT):
  - `op` is `PLUS`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - define `r` as the literal integer for $a + b$.
- All of the following apply (SUB_INT):
  - `op` is `MINUS`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - define `r` as the literal integer for $a - b$.
- All of the following apply (MUL_INT):
  - `op` is `MUL`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - define `r` as the literal integer for $a \times b$.
- All of the following apply (DIV_INT):
  - `op` is `DIV`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - checking that $b$ is positive yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - define $n$ as $a$ divided by $b$ (note that $n$ is potentially a fraction);
  - checking that $n$ is an integer yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - define `r` as the literal integer for $a \div b$.
- All of the following apply (FDIV_INT):
  - `op` is `DIVRM`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - checking that $b$ is positive yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - define $n$ as $a$ divided by $b$, rounded down (if $a$ is negative, $n$ is rounded down towards infinity);
  - define `r` as the literal integer for $n$.
- All of the following apply (FREM_INT):
  - `op` is `MOD`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - applying *binop_literals* to `DIVRM` with `l1` and `l2` yields $c^{/\!/\texttt{\#TE}}$;
  - define $n$ as $a - c$;

  - define `r` as the literal integer for $n$.

- All of the following apply (EXP_INT):
  - `op` is `POW`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - checking that $b$ is non-negative yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - define $n$ as $a^b$;
  - define `r` as the literal integer for $n$.

- All of the following apply (SHL):
  - `op` is `SHL`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - checking that $b$ is non-negative yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - applying *binop_literals* to `POW` with 2 and `l2` yields the literal integer for $e$;
  - applying *binop_literals* to `MUL` with 2 and the literal integer for $e$ yields `r`.

- All of the following apply (SHR):
  - `op` is `SHR`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - checking that $b$ is non-negative yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - applying *binop_literals* to `POW` with 2 and `l2` yields the literal integer for $e$;
  - applying *binop_literals* to `DIVRM` with 2 and the literal integer for $e$ yields `r`.

- All of the following apply (EQ_INT):
  - `op` is `EQ_OP`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is equal to $b$.

- All of the following apply (NE_INT):
  - `op` is `NEQ`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is different from $b$ holds.

- All of the following apply (LE_INT):
  - `op` is `LEQ`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is less than or equal to $b$s.

- All of the following apply (LT_INT):
  - `op` is `LT`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is less than $b$s.

- All of the following apply (GE_INT):

    - `op` is `GEQ`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
    - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is greater or equal than $b$s.

- All of the following apply (GT_INT):
    - `op` is `GT`, `l1` is the literal integer for $a$, and `l2` is the literal integer for $b$;
    - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is greater than $b$s.

- All of the following apply (AND_BOOL):
    - `op` is `BAND`, `l1` is the literal Boolean for $a$, and `l2` is the literal Boolean for $b$;
    - define `r` as the Boolean literal that is `TRUE` if and only if both $a$ and $b$ are `TRUE`.

- All of the following apply (OR_BOOL):
    - `op` is `BOR`, `l1` is the literal Boolean for $a$, and `l2` is the literal Boolean for $b$;
    - define `r` as the Boolean literal that is `TRUE` if and only if at least one of $a$ and $b$ is `TRUE`.

- All of the following apply (IMPLIES_BOOL):
    - `op` is `IMPL`, `l1` is the literal Boolean for $a$, and `l2` is the literal Boolean for $b$;
    - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is `FALSE` or $b$ is `TRUE`.

- All of the following apply (EQ_BOOL):
    - `op` is `EQ_OP`, `l1` is the literal Boolean for $a$, and `l2` is the literal Boolean for $b$;
    - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is equal to $b$.

- All of the following apply (NE_BOOL):
    - `op` is `NEQ`, `l1` is the literal Boolean for $a$, and `l2` is the literal Boolean for $b$;
    - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is different from $b$.

- All of the following apply (MUL_INT_REAL):
    - `op` is `MUL`, `l1` is the literal integer for $a$, and `l2` 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`, `l1` is the literal real for $a$, and `l2` 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`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the real literal for $a + b$.

- All of the following apply (SUB_REAL):
  - `op` is `MINUS`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the real literal for $a - b$.

- All of the following apply (MUL_REAL):
  - `op` is `MUL`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the real literal for $a \times b$.

- All of the following apply (DIV_REAL):
  - `op` is `RDIV`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - checking whether $b$ is different from 0 yields `TRUE`$^{\#\texttt{TE}}$;
  - define `r` as the real literal for $a \div b$.

- All of the following apply (EXP_REAL):
  - `op` is `POW`, `l1` is the literal real for $a$, and `l2` is the literal integer for $b$;
  - 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`$^{\#\texttt{TE}}$;
  - define `r` as the real literal for $a^b$.

- All of the following apply (EQ_REAL):
  - `op` is `EQ_OP`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is equal to $b$.

- All of the following apply (NE_REAL):
  - `op` is `NEQ`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is different from $b$.

- All of the following apply (LE_REAL):
  - `op` is `LEQ`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is less than or equal to $b$.

- All of the following apply (LT_REAL):
  - `op` is `LT`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is less than $b$.

- All of the following apply (GE_REAL):
  - `op` is `GEQ`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is greater than or equal to $b$.
- All of the following apply (GT_REAL):
  - `op` is `GT`, `l1` is the literal real for $a$, and `l2` is the literal real for $b$;
  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is greater than $b$.
- All of the following apply (BITWISE_DIFFERENT_BITWIDTHS):
  - `v1` is a bitvector literal for $a$;
  - `v2` is a bitvector literal for $b$;
  - the lengths of $a$ and $b$ are different;
  - the result is a type error indicating that the bitvectors must be of the same width.
- All of the following apply (BITWISE_EMPTY):
  - `v1` is the empty bitvector literal;
  - `v2` is the empty bitvector literal;
  - `op` is one of `OR`, `AND`, `XOR`, `PLUS`, or `MINUS`;
  - define `r` as the empty bitvector literal.
- All of the following apply (EQ_BITS_EMPTY):
  - `v1` is the empty bitvector literal;
  - `v2` is the empty bitvector literal;
  - `op` is `EQ_OP`;
  - define `r` as the Boolean literal for `TRUE`.
- All of the following apply (EQ_BITS_NOT_EMPTY):
  - `v1` is a bitvector literal for $a_{1..k}$;
  - `v2` is a bitvector literal for $b_{1..k}$;
  - `op` is `EQ_OP`;
  - define `b` as `TRUE` if and only if $a_i$ is equal to $b_i$, for $i = 1..k$;
  - define `r` as the Boolean literal for `b`.
- All of the following apply (NE_BITS):
  - `v1` is a bitvector literal for $a$;

    * `v2` is a bitvector literal for $b$;
    * `op` is `NEQ`;
    * applying *binop_literals* to `NEQ` for `v1` and `v2` yields the Boolean literal for $\mathtt{b}^{/\!/\mathtt{\#TE}}$;
    * 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`, `l1` is the literal string for $a$, and `l2` is the literal string for $b$;
    * define `r` as the Boolean literal that is `TRUE` if and only if $a$ is equal to $b$.

- All of the following apply (NE_STRING):
    * `op` is `NEQ`, `l1` is the literal string for $a$, and `l2` is the literal string for $b$;

  - define `r` as the Boolean literal that is `TRUE` if and only if $a$ is different from $b$.

- All of the following apply (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**

ERROR

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

**Arithmetic Operators Over Integer Values**

ADD_INT

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

SUB_INT

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

MUL_INT

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

DIV_INT

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

FDIV_INT

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

FREM_INT

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

EXP_INT

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

SHL

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

SHR

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

### Comparison Operators Over Integer Values

EQ_INT

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

NE_INT

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

LE_INT

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

LT_INT

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

GE_INT

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

GT_INT

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

### Boolean Operators Over Boolean Values

AND_BOOL

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

OR_BOOL

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

IMPLIES_BOOL

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

EQ_BOOL

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

NE_BOOL

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

### Arithmetic Operators Over Real Values

MUL_INT_REAL

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

MUL_REAL_INT

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

ADD_REAL

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

SUB_REAL

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

MUL_REAL

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

DIV_REAL

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

EXP_REAL

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

### Comparison Operators Over Real Values

EQ_REAL

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

NE_REAL

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

LE_REAL

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

LT_REAL

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

GE_REAL

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

GT_REAL

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

### Operators Over Bitvectors

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

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

and an empty sequence of bits into 0:

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

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

BITWISE_DIFFERENT_BITWIDTHS

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

BITWISE_EMPTY

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

EQ_BITS_EMPTY

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

EQ_BITS_NOT_EMPTY

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

NE_BITS

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

OR_BITS

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

AND_BITS

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

XOR_BITS

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

ADD_BITS

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

SUB_BITS

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

CONCAT_BITS

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

ADD_BITS_INT

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

SUB_BITS_INT

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

### Operators Over String Values

EQ_STRING

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

NE_STRING

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

CONCAT_STRINGS

$$\frac{\texttt{v1} \neq \mathsf{L\_Bitvector}(\_) \vee \texttt{v2} \neq \mathsf{L\_Bitvector}(\_) \qquad \texttt{s1} := \mathit{literal\_to\_string}(\texttt{v1}) \qquad \texttt{s2} := \mathit{literal\_to\_string}(\texttt{v2})}{\mathit{binop\_literals}(\overbrace{\texttt{CONCAT}}^{\texttt{op}}, \texttt{v1}, \texttt{v2}) \xrightarrow{\mathsf{type}} \overbrace{\texttt{s1} + \texttt{s2}}^{\texttt{r}}}$$

### Operators Over Label Values

EQ_LABEL

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

NE_LABEL

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

## 12.5 Semantics

### SemanticsRule.UnopValues

The function

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

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

### 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.

$$\frac{\mathit{unop\_literals}(\mathtt{NEG}, \mathsf{L\_Int}(1)) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(-1)}{\mathit{unop}(\mathtt{NEG}, \mathtt{Int}(1)) \xrightarrow{\mathsf{eval}} \mathtt{Int}(-1)}$$

### 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.

$$\frac{\mathit{unop\_literals}(\mathtt{NEG}, \mathsf{L\_Bool}(\mathtt{TRUE})) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_BO})}{\mathit{unop}(\mathtt{NEG}, \mathtt{Int}(\mathtt{TRUE})) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_BO})}$$

### Prose

One of the following applies:

- All of the following apply (OK):
  - `v` is a literal native value, that is, $\mathtt{NV\_Literal}(l)$;
  - statically evaluating `op` on the literal $l$ yields a literal $l'$;
  - `w` is the native literal value for $l'$.
- All of the following apply (STATIC_ERROR):
  - `v` is a literal native value, that is, $\mathtt{NV\_Literal}(l)$;
  - statically evaluating `op` on $l$ yields a type error;
  - 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**

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

$$\text{STATIC\_ERROR}\quad \frac{\mathit{unop\_literals}(\mathtt{op}, l) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\_)}{\mathit{unop}(\mathtt{op}, \overbrace{\mathtt{NV\_Literal}(l)}^{\mathtt{v}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_BO})}$$

$$\text{NON\_LITERAL}\quad \frac{\mathit{ast\_label}(\mathtt{v}) \neq \mathtt{NV\_Literal}}{\mathit{unop}(\mathtt{op}, \mathtt{v}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_BO})}$$

**SemanticsRule.BinopValues**

The function

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

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

**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.

$$\frac{\mathit{binop\_literals}(\mathtt{PLUS}, \mathsf{L\_Int}(2), \mathsf{L\_Int}(3)) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(5)}{\mathit{binop}(\mathtt{PLUS}, \mathtt{Int}(2), \mathtt{Int}(3)) \xrightarrow{\mathsf{eval}} \mathtt{Int}(5)}$$

**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.

$$\frac{\mathit{binop\_literals}(\mathtt{PLUS}, \mathsf{L\_Int}(2), \mathsf{L\_Real}(1/2)) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_BO})}{\mathit{binop}(\mathtt{PLUS}, \mathtt{Int}(2), \mathtt{Real}(1/2)) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_BO})}$$

### 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_BO`).
- 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_BO`).

### Formally

OK

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

STATIC_ERROR

$$\frac{\mathit{binop\_literals}(\mathtt{op}, l_1, l_2) \xrightarrow{\text{type}} \mathsf{TypeError}(\_)}{\mathit{binop}(\mathtt{op}, \overbrace{\mathtt{NV\_Literal}(l_1)}^{\mathtt{v1}}, \overbrace{\mathtt{NV\_Literal}(l_2)}^{\mathtt{v2}}) \xrightarrow{\text{eval}} \mathsf{DynError}(\mathtt{DE\_BO})}$$

NON_LITERAL

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

# 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

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

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

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

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

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

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

Formally:

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

### 13.1.2 Typing

The function

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

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*, *well-constrained*, *pending constrained*, and *parameterized*.

### 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{`$c_1, \ldots, c_n$`}` represents the union of sets of integers represented by the *integer constraints* $c_1, \ldots, c_n$. 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

Listing 13.2 shows examples of well-constrained integer types.

Listing 13.2: Well-typed well-constrained integer types

```
type MyType of integer{1..12}; // Name a well-constrained integer type.

func foo(x: integer{1..7}) => integer{1..12}
begin
  return x;
end;

func main () => integer
begin
  var x: integer{1..12};
  x = 4;
  x = foo(x as integer{1..7});

  let y: integer{1..12} = x;
  let x2 = x as integer{1..11};
  assert x2 == x;

  let z : integer{2..24} = x + y;
  // The type of 'w' is inferred to be integer{2..24}.
  var w = x + y;
  w = z;

  var c = 3; // The type of 'c' is inferred to be integer{3}.

  // 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;
```

### 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 13.6 shows an ill-typed specification.

**Example: Well-typed pending-constrained types**

Listing 13.3 shows examples of well-typed pending-constrained integer types.

Listing 13.3: 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 13.4 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 13.4: Well-typed parameterized integer types

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

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

$$
\begin{aligned}
\texttt{ty} &\longrightarrow \texttt{"integer"}\ \texttt{constraint\_kind\_opt} \\
\texttt{constraint\_kind\_opt} &\longrightarrow \texttt{constraint\_kind} \mid \epsilon \\
\texttt{constraint\_kind} &\longrightarrow \texttt{"\{"}\ \texttt{clist1}(\texttt{int\_constraint})\ \texttt{"\}"} \\
&\quad \mid \texttt{"\{"}\ \texttt{"-"}\ \texttt{"\}"} \\
\texttt{int\_constraint} &\longrightarrow \texttt{expr} \\
&\quad \mid \texttt{expr}\ \texttt{".."}\ \texttt{expr}
\end{aligned}
$$

### 13.2.6 Abstract Syntax

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

#### ASTRule.Ty.TInt

INTEGER

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

#### ASTRule.IntConstraintsOpt

The function

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

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

CONSTRAINED

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

UNCONSTRAINED

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

### 13.2.7 ASTRule.IntConstraints

The function

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

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

WELL_CONSTRAINED

$$\frac{\mathit{build\_clist}[\mathit{build\_int\_constraint}](\texttt{cs}) \xrightarrow{\text{ast}} \texttt{cs\_asts}}{\mathit{build\_constraint\_kind}(\texttt{constraint\_kind}(\texttt{"\{"}, \texttt{cs} : \mathsf{clist1}(\texttt{int\_constraint}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{WellConstrained}(\texttt{cs\_asts})}^{\texttt{ast\_node}}}$$

PENDING_CONSTRAINED

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

#### ASTRule.IntConstraint

The function

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

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

EXACT

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

RANGE

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

### 13.2.8 Typing Integer Types

We use the following helper predicates to classify integer types:

$$\mathsf{is\_unconstrained\_integer}(\overbrace{\mathsf{ty}}^{\texttt{t}}) \longrightarrow \mathbb{B}$$

$$\mathsf{is\_parameterized\_integer}(\overbrace{\mathsf{ty}}^{\texttt{t}}) \longrightarrow \mathbb{B}$$

$$\mathsf{is\_well\_constrained\_integer}(\overbrace{\mathsf{ty}}^{\texttt{t}}) \longrightarrow \mathbb{B}$$

Those are defined as follows:

$$\begin{aligned} \mathsf{is\_unconstrained\_integer}(\mathtt{t}) &\triangleq \mathtt{t} = \mathsf{T\_Int}(c) \wedge \mathit{ast\_label}(c) = \mathsf{Unconstrained} \\ \mathsf{is\_parameterized\_integer}(\mathtt{t}) &\triangleq \mathtt{t} = \mathsf{T\_Int}(c) \wedge \mathit{ast\_label}(c) = \mathsf{Parameterized} \\ \mathsf{is\_well\_constrained\_integer}(\mathtt{t}) &\triangleq \mathtt{t} = \mathsf{T\_Int}(c) \wedge \mathit{ast\_label}(c) = \mathsf{WellConstrained} \end{aligned}$$

We use the shorthand notation $\mathsf{unconstrained\_integer} \triangleq \mathsf{T\_Int}(\mathsf{Unconstrained})$ for unconstrained integers.

**Guide.ConstraintSymbolicallyConstrained** The expressions appearing in integer constraints must be both symbolically evaluable and constrained integer types. In Listing 13.5, the constraint `x..x+1` is ill-typed, since the type of `x` is not constrained.

Listing 13.5: Ill-typed constraint

```
func main() => integer
begin
  var x : integer = 1;
  let t: integer{x..x+1} = 2; // illegal as 'x' is not constrained.
  return 0;
end;
```

**TypingRule.TInt**

**Example: Ill-typed pending-constrained integer type**

Listing 13.6 and Listing 13.7 correspond to case PENDING_CONSTRAINED.

Listing 13.6: Ill-typed pending-constrained integer type

```
config x : integer{-} =  1;
```

Listing 13.7: 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 $\texttt{c}_i$, for $u = 1..k$;
  - annotating each constraint $\texttt{c}_i$, for $i = 1..k$, yields $(\texttt{new\_c}_i, \texttt{xs}_i)^{\text{// \#TE}}$;
  - `new_constraints` is the list of annotated constraints $\texttt{new\_c}_i$, for $i = 1..k$;
  - `new_ty` is the well-constrained integer type constrained by `new_constraints` with Precision_Full;
  - define `ses` as the union of all $\texttt{xs}_i$, for $i = 1..k$.
- All of the following apply (PARAMETERIZED):
  - `ty` is a parameterized integer type for `name`;
  - define `ses` as the singleton set for the singleton side effect descriptor, local read side effect descriptor for `name`, Constant, and `TRUE` for immutability.
  - `new_ty` is the unconstrained integer type.
- All of the following apply (UNCONSTRAINED):
  - `ty` is an unconstrained integer type;
  - `new_ty` is the unconstrained integer type;
  - define `ses` as the empty set.

**Formally**

PENDING_CONSTRAINED

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

WELL_CONSTRAINED

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

PARAMETERIZED

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

UNCONSTRAINED

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

### TypingRule.AnnotateConstraint

The function

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

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

Listing 13.8 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 13.8: 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 $(\texttt{e'}, \texttt{ses})^{\#\texttt{TE}}$;
  - define `new_c` as the exact integer constraint for `e'`, that is, Constraint_Exact(`e'`).
- All of the following apply (RANGE):
  - `c` is the range integer constraint for expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *annotate_symbolic_constrained_integer* to `e1` in `tenv` yields $(\texttt{e1'}, \texttt{ses1})^{\#\texttt{TE}}$;

* applying *annotate_symbolic_constrained_integer* to `e2` in `tenv` yields $(\texttt{e2'}, \texttt{ses2})^{/\!/ \texttt{\#TE}}$;
* define `new_c` as the range integer constraint for expressions `e1'` and `e2'`, that is, Constraint_Range(`e1'`, `e2'`);
* define `ses` as the union of `ses1` and `ses2`.

**Formally**

EXACT

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

RANGE

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

## 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 13.9, all the uses of the real type are well-typed.

Listing 13.9: 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 ⟶ "real"

### 13.3.2 Abstract Syntax

ty ⟶ T_Real

#### ASTRule.TReal

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

### 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

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

## 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 13.10, all the uses of the string type are well-typed.

Listing 13.10: 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 ⟶ `"string"`

### 13.4.2 Abstract Syntax

ty ⟶ T_String

**ASTRule.Ty.String**

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

### 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**

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

## 13.5 The Boolean Type

The *boolean type* represents Booleans.

### Example: Well-typed Boolean Types

In Listing 13.11, all the uses of the boolean type are well-typed.

Listing 13.11: 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 ⟶ "boolean"

### 13.5.2 Abstract Syntax

ty ⟶ T_Bool

**ASTRule.Ty.BoolType**

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

### 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**

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

## 13.6 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`).

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.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 13.12, 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 13.12: Large and small bitvectors

```
var large_bitvector: bits(2^20);
var zero_width_bitvector: bits(0);
```

**Guide.BitvectorWidthKind** The width of a bitvector type can be either statically evaluable or *constrained*. That is, a symbolically evaluable constrained integer.

### Example: Rotating a Bitvector

Listing 13.13 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 13.13: 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

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

**ASTRule.Ty.TBits**

BIT

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

BITS

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

### 13.6.3 Typing Bitvector Types

**TypingRule.TBits**

**Example: Well-typed Bitvector Types**

In Listing 13.14, all the uses of bitvector types are well-typed.

Listing 13.14: 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;
```

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, $\mathsf{T\_Bits}(\texttt{e\_width}, \texttt{bitfields})$;
- annotating the expression `e_width` yields $(\texttt{t\_width}, \texttt{e\_width'}, \texttt{ses\_width})^{/\!/\#\texttt{TE}}$;
- checking that `ses_width` is symbolically evaluable yields $\texttt{TRUE}^{/\!/\#\texttt{TE}}$;

- checking that the type `t_width` is a constrained integer in the static environment tenv yields TRUE$^{/\!/\texttt{\#TE}}$;
- One of the following applies:
  - All of the following apply (WITH_BITFIELDS):
    - `bitfields` is not empty;
    - checking whether all time frames in `ses_width` are less than or equal to Constant yields TRUE$^{/\!/\texttt{TE_SEV}}$;
    - annotating the bitfields `bitfields` yields (`bitfields'`, `ses_bitfields`)$^{/\!/\texttt{\#TE}}$;
    - statically evaluating the expression `e_width'` in the static environment tenv yields the literal L_Int(`width`);
    - checking that all pairs of bitfields in `bitfields'` that are in the same scope and share the same name correspond to the same slice of the containing bitvector type in the static environment tenv yields TRUE$^{/\!/\texttt{\#TE}}$;
    - 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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \texttt{e\_width}) \xrightarrow{\text{type}} (\texttt{t\_width}, \texttt{e\_width'}, \texttt{ses\_width}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_symbolically\_evaluable}(\texttt{ses\_width}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_constrained\_integer}(\mathsf{tenv}, \texttt{t\_width}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \overbrace{\texttt{bitfields} \neq [\,]}^{\text{******** common prefix ********}} \\ \mathit{check}(\mathit{ses\_is\_before}(\texttt{ses\_width}, \mathsf{Constant}), \texttt{TE\_SEV}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_bitfields}(\mathsf{tenv}, \texttt{e\_width'}, \texttt{bitfields}) \xrightarrow{\text{type}} \\ (\texttt{bitfields'}, \texttt{ses\_bitfields}) \;/\!/\; \texttt{\#TE} \\ \mathit{static\_eval}(\mathsf{tenv}, \texttt{e\_width'}) \xrightarrow{\text{type}} \mathsf{L\_Int}(\texttt{width}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_common\_bitfields\_align}(\mathsf{tenv}, \texttt{bitfields'}, \texttt{width}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_width} \cup \texttt{ses\_bitfields} \end{array}}{\mathit{annotate\_type}(\overbrace{\_}^{\text{decl}}, \mathsf{tenv}, \mathsf{T\_Bits}(\texttt{e\_width}, \texttt{bitfields})) \xrightarrow{\text{type}} (\overbrace{\mathsf{T\_Bits}(\texttt{e\_width'}, \texttt{bitfields'})}^{\texttt{new\_ty}}, \texttt{ses})}$$

NO_BITFIELDS

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e\_width}) \xrightarrow{\text{type}} (\texttt{t\_width}, \texttt{e\_width'}, \texttt{ses\_width}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_symbolically\_evaluable}(\texttt{ses\_width}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_constrained\_integer}(\texttt{tenv}, \texttt{t\_width}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \overset{\text{******** common prefix ********}}{\texttt{bitfields} = [\,]}\end{array}}{\mathit{annotate\_type}(\overbrace{\_}^{\texttt{decl}}, \texttt{tenv}, \texttt{T\_Bits}(\texttt{e\_width}, \texttt{bitfields})) \xrightarrow{\text{type}} (\overbrace{\texttt{T\_Bits}(\texttt{e\_width'}, \texttt{bitfields'})}^{\texttt{new\_ty}}, \overbrace{\texttt{ses\_width}}^{\texttt{ses\_width}})}$$

## 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 13.15, all the uses of tuple types are well-typed.

Listing 13.15: 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 13.16, both `x` and `y` have tuple types, whereas `w` and `z` are of the integer type, since `(5)` is considered a parenthesized expressions, not a tuple expression.

Listing 13.16: 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 13.17 demonstrates how variables of a tuple type may be assigned, but the tuple values they store may not be modified.

Listing 13.17: 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 13.18 shows examples of accessing the elements of the tuple stored in `x`.

Listing 13.18: Accessing tuple elements

```
func main() => integer
begin
    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;
    // The following statement in comment is illegal
    // as item2 is not an element of the tuple stored in 'x'.
    // x = (x.item1, x.item2);
    return 0;
end;
```

### 13.7.1 Syntax

ty $\longrightarrow$ plist0(ty)

### 13.7.2 Abstract Syntax

ty $\longrightarrow$ T_Tuple(ty$^*$)

**ASTRule.Ty.TTuple**

$$\frac{\mathit{build\_plist}[\mathit{build\_ty}](\texttt{types}) \xrightarrow{\text{ast}} \texttt{type\_asts}}{\mathit{build\_ty}(\texttt{ty}(\texttt{types} : \mathsf{plist0}(\texttt{ty}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Tuple}(\texttt{type\_asts})}^{\texttt{ast\_node}}}$$

### 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, $\mathsf{T\_Tuple}(\texttt{tys})$;
- `tys` is the list $\texttt{ty}_i$, for $i = 1..k$ and $k > 1$;
- annotating each type $\texttt{ty}_i$ in `tenv`, for $i = 1..k$, yields $(\texttt{ty'}_i, \texttt{xs}_i)^{/\!/ \texttt{\#TE}}$;
- `new_ty` is the tuple type with member types $\texttt{ty'}_i$, for $i = 1..k$;
- define `ses` as the union of all $\texttt{xs}_i$, for $i = 1..k$.

**Formally**

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

## 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 13.19 shows an example of a well-typed enumeration type declaration.

Listing 13.19: 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 13.20 shows an illegal specification where an enumeration is used outside of a type definition.

Listing 13.20: An illegal enumeration use

```
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 ⟶ "enumeration" "{" tclist1(**ID**) "}"

### 13.8.2 Abstract Syntax

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

**ASTRule.TyDecl.TEnum**

$$\frac{\mathit{build\_tclist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{id\_asts}}{\mathit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{"enumeration"}, \texttt{"\{"}, \texttt{ids} : \mathsf{tclist1}(\mathbf{ID}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Enum}(\texttt{id\_asts})}^{\text{ast\_node}}}$$

### 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 13.21 shows examples of ill-typed enumeration type declarations.

Listing 13.21: Ill-typed enumeration types

```
constant GREEN = 1;
type Color of enumeration { GREEN, ORANGE, RED }; // Illegal: GREEN already declared.
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, $\mathsf{T\_Enum}(\texttt{li})$;
- decl is `TRUE`, indicating that `ty` should be considered in the context of a declaration;
- determining that `li` does not contain duplicates yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- determining that none of the labels in `li` is declared in the global environment yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `new_ty` is the enumeration type `ty`;
- define `ses` as the empty set.

**Formally**

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

## 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 13.22, all the uses of array types are well-typed.

Listing 13.22: 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 ⟶ "array" "[[" expr "]]" "of" ty

### 13.9.2 Abstract Syntax

$$\text{ty} \longrightarrow \text{T\_Array}(\text{array\_index}, \text{ty})$$

$$\text{array\_index} \longrightarrow \text{ArrayLength\_Expr}(\overbrace{\text{expr}}^{\text{array length}})$$

#### ASTRule.Ty.TArray

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

### 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 13.23, the array type for `illegal_array` is ill-typed, since the expression `non_symbolically_evaluable` is not symbolically evaluable.

Listing 13.23: Ill-typed array types

```
func foo(symbolically_evaluable_var: integer)
begin
    var legal_array: array [[symbolically_evaluable_var]] of integer;
    let symbolically_evaluable_var2 = symbolically_evaluable_var * 2;
    var legal_array2: array [[symbolically_evaluable_var2]] of integer;

    // Illegal as non_symbolically_evaluable is mutable.
    var non_symbolically_evaluable = 5;
    var illegal_array: array [[non_symbolically_evaluable]] 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 $(\texttt{t'}, \texttt{ses\_t})^{/\!/\texttt{\#TE}}$;
- One of the following applies:
  - All of the following apply (EXPR_IS_ENUM):
    - the array index is `e` and determining whether `e` corresponds to an enumeration in `tenv` via *get_variable_enum* yields the enumeration variable name `s` of size `i`, that is, $\langle \texttt{s}, \texttt{i} \rangle^{/\!/\texttt{\#TE}}$;
    - `new_ty` is the array type indexed by an enumeration type named `s` of length `i` and of elements of type `t'`, that is, T_Array(ArrayLength_Enum(`s`, `i`), `t'`);
    - define `ses` as `ses_t`.
  - All of the following apply (EXPR_NOT_ENUM):
    - the array index is `e` and determining whether `e` corresponds to an enumeration in `tenv` via *get_variable_enum* yields `None` (meaning it does not correspond to an enumeration)$^{/\!/\texttt{\#TE}}$;
    - annotating the symbolically evaluable integer expression `e` yields $(\texttt{e'}, \texttt{ses\_index})^{/\!/\texttt{\#TE}}$;
    - `new_ty` the array type indexed by integer bounded by the expression `e'` and of elements of type `t'`, that is, T_Array(ArrayLength_Expr(`e'`), `t'`);
    - define `ses` as the union of `ses_t` and `ses_index`.

### Formally

EXPR_IS_ENUM

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

EXPR_NOT_ENUM

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

### TypingRule.GetVariableEnum

The function

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

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

### Example: Retrieving Enumeration Labels from Variable Expressions

Listing 13.24 shows examples of retrieving enumeration labels from variable expressions.

Listing 13.24: Retrieving enumeration labels from expressions

```
type Key of enumeration {One, Two, Three};
type SubKey subtypes Key;

func main() => integer
begin             // The right-hand-side expression is | Reason:
    var x = 5;    // Not an enumeration variable       | not a variable expression
    var y = x;    // Not an enumeration variable       | x is integer-typed
    var z = One; // An enumeration variable            | the underlying type is Key
    return 0;
end;
```

### Prose

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**

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

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

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

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

**Guide.SymbolicallyEvaluable** An expression is symbolically evaluable if its evaluation only involves the use of immutable values.

See Example: Annotating Symbolically Evaluable Expressions.

**TypingRule.AnnotateSymbolicallyEvaluableExpr**

The function

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

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 13.25 shows examples of expressions and classifies them as either symbolically evaluable or not.

Listing 13.25: Annotating symbolically evaluable Expressions

```
func pure_func(x: integer, y: integer) => integer
begin
    return x * y + y;
end;

type my_exception of exception {-};

func impure_func(x: integer, y: integer) => integer
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; // The underlying type of MyInteger is integer.

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
    // 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 `TRUE`$^{\#\texttt{TE}}_{/\!/}$.

### Formally

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

### TypingRule.AnnotateSymbolicInteger

The function

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

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

### Example: Annotating Symbolically Evaluable Integer Expressions

The expression `(pure_func(5, 6) + 9) + I` in Listing 13.25 is both symbolically evaluable and its underlying type is an integer type, and annotating and normalizing it yields the same expression. Annotating the expression `(3*I + 9) - 2*I`, which is also symbolically evaluable and has an underlying type is an integer type, yields the expression `I + 9` after normalization. Annotating the expression `i - 3`, which is also symbolically evaluable and has an underlying type is an integer type, yields the expression `6` after normalization.

### Prose

All of the following apply:

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

### Formally

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

### TypingRule.CheckUnderlyingInteger

The function

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

returns `TRUE` is `t` is 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 13.25 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'`$^{\text{// \#TE}}$;
- checking that `t'` is an integer type yields `TRUE`$^{\text{// TE\_UT}}$;
- the result is `TRUE`;

#### Formally

$$\frac{\mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t'} \;//\; \mathtt{\#TE} \qquad \mathit{check}(\mathit{ast\_label}(\mathtt{t'}) = \mathsf{T\_Int}, \mathtt{TE\_UT}) \xrightarrow{\text{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}}{\mathit{check\_underlying\_integer}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE}}$$

## 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 13.26, all the uses of record types are well-typed.

Listing 13.26: Well-typed structured types

```
type MyRecord of record { a: integer, b: boolean };
type RecordWithEmptyFieldList of record {-};
type RecordWithoutFields of record;

func main() => integer
begin
    - = MyRecord {a = 3, b = TRUE};
    - = RecordWithEmptyFieldList {-};
    - = RecordWithoutFields {-};
    return 0;
end;
```

In Listing 13.27, the record type `MyRecord` is ill-typed, since the field `v` repeats.

Listing 13.27: A record type with repeated fields

```
type MyRecord of record {v: integer, b: boolean, v: real};
```

### 13.10.1 Syntax

ty_decl ⟶ "record" fields_opt

### 13.10.2 Abstract Syntax

ty ⟶ T_Record(field*)

**ASTRule.TyDecl.TRecord**

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

### 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 — $(\texttt{x}_i, \texttt{t}_i)$, for $i = 1..k$;
- checking that the list of field identifiers $\texttt{x}_{1..k}$ does not contain duplicates yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- annotating each field type $\texttt{t}_i$, for $i = 1..k$, yields $(\texttt{t'}_i, \texttt{xs}_i)$ $^{/\!/\texttt{\#TE}}$;
- `fields'` is the list with $(\texttt{x}_i, \texttt{t'}_i)$, for $i = 1..k$;
- `new_ty` is the AST node with AST label $L$ (either record type or exception type, corresponding to the type `ty`) and fields `fields'`;
- define `ses` as the union of all $\texttt{xs}_i$, for $i = 1..k$.

**Formally**

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

## 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 13.28, all the uses of exception types are well-typed.

Listing 13.28: 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

ty_decl ⟶ "exception" fields_opt

### 13.11.2 Abstract Syntax

ty ⟶ T_Exception(field*)

**ASTRule.TyDecl.TException**

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

### 13.11.3 Typing Exception Types

The rule for typing exception types is TypingRule.TStructuredDecl.

## 13.12 Collection Types

Listing 13.29: Collection types

```
type MyCollection of collection { a: bits(8), b: bits(16) };
type CollectionWithEmptyFieldList of collection {-};
type CollectionWithoutFields of collection;

// The next type declaration in comment is illegal:
// only bitvector types are allowed as collection fields.
// type IllegalCollection of 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 - = MyCollection {a = Zeros{8}, b = Zeros{16}};
// var - : MyCollection = MyCollection {a = Zeros{8}, b = Zeros{16}};

var x : MyCollection;
var y : CollectionWithEmptyFieldList;
var z :CollectionWithoutFields;

func main() => integer
begin
    // The next declaration in comment is illegal:
    // local storage elements of collection types are forbidden.
    // var - : MyCollection;
    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 19.3 for an ill-typed specification.

### 13.12.1 Syntax

ty_decl ⟶ "collection" fields_opt

### 13.12.2 Abstract Syntax

ty ⟶ T_Collection(field*)

**ASTRule.TyDecl.TCollection**

$$build\_ty\_decl(\texttt{ty\_decl}(\texttt{"collection"}, \texttt{fields\_opt})) \xrightarrow{ast} \overbrace{\mathsf{T\_Collection}(\texttt{fields\_opt})}^{\texttt{ast\_node}}$$

## 13.13 Typing Collection Types

### Example: Typing Collection Types

Listing 13.29 shows examples of well-typed collection types and ill-typed collection types in comments. In addition, Listing 13.29 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 `TRUE`, indicating that `ty` should be considered in the context of a declaration;
- `fields` is a list of pairs where the first element is an identifier and the second is a type — $(\texttt{x}_i, \texttt{t}_i)$, for $i = 1..k$;
- checking that the list of field identifiers $\texttt{x}_{1..k}$ does not contain duplicates yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- annotating each field type $\texttt{t}_i$, for $i = 1..k$, yields $(\texttt{t'}_i, \texttt{xs}_i)$ $^{/\!/\texttt{\#TE}}$;
- `fields'` is the list with $(\texttt{x}_i, \texttt{t'}_i)$, for $i = 1..k$;
- checking that the structure of the type $\texttt{t'}_i$ in the static environment `tenv` is a bitvector type, for every $i$ in $1..k$, yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `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 $\texttt{xs}_i$, for $i = 1..k$.

### Formally

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

## 13.14 Named Types

### 13.14.1 Syntax

ty ⟶ **ID**

### 13.14.2 Abstract Syntax

ty ⟶ T_Named(identifier) (identifier: type name)

**ASTRule.Ty.TNamed**

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

### 13.14.3 Typing Named Types

**TypingRule.TNamed**

**Example: Well-typed Named Types**

In Listing 13.30, all the uses of `MyType` are well-typed.

Listing 13.30: 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;
```

**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`$^{\#\mathtt{TE}}_{/\!/}$;
- `x` is bound to a type with associated time frame `time_frame`;
- define `ses` as the singleton set for the global read side effect descriptor for `x`, `time_frame`, and `TRUE` for immutability;
- `new_ty` is `ty`.

**Formally**

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

# 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

`ty_decl` ⟶ `ty`

## 13.15.2 Abstract Syntax

**ASTRule.TyDecl**

TY

$$\mathit{build\_ty\_decl}(\mathtt{ty\_decl}(\mathtt{ty})) \xrightarrow{\mathsf{ast}} \overbrace{\mathtt{ty}}^{\mathtt{ast\_node}}$$

## 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 13.31, the use of a record type outside of a declaration is erroneous.

Listing 13.31: 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**

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

## 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

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

which assigns the set of values that a type `t` can hold in a given environment `env`. We say that $\mathsf{dyn\_dom}(\texttt{env}, \texttt{t})$ is the *dynamic domain* of `t` in the environment `env`. The

*static domain* of a type is the set of values which storage elements of that type may hold across all possible dynamic environments. The reason for this distinction is that the sets of values of integer types, bitvector types, and array types can depend on the dynamic values of variables.

Types that do not refer to variables whose values are only known dynamically have a static domain that is equal to any of their dynamic domains. In those cases, we simply refer to their *domain*.

Associating a set of values to a type is done by evaluating any expression appearing in the type definitions. Expressions appearing in types are guaranteed to be side-effect-free by the function *annotate_type*(). Evaluation is defined by the relation *eval_expr_sef*(). which evaluates side-effect-free expressions and either returns a configuration of the form Normal(v, g) or a dynamic error configuration #DE. In the first case, v is a native value and g is an *execution graph*. Execution graphs are related to the concurrent semantics and can be ignored in the context of defining dynamic domains. In the latter case (which can occur if, for example, an expression attempts to divide 8 by 0), a dynamic error configuration, for which we use the notation #DE, is returned. The dynamic domain is empty in cases where evaluating side-effect-free expressions results in a dynamic error. The dynamic domain is undefined if the type t is not well-typed in tenv. That is, if *annotate_type*(tenv, t) $\xrightarrow{\text{type}}$ #TE.

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

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

### 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, $\mathcal{B}$.
- All of the following apply (T_STRING):
  - t is the string type, T_String;
  - d is the set of all native string values, $\mathcal{STR}$.
- All of the following apply (T_REAL):
  - t is the real type, T_Real;
  - d is the set of all native real values, $\mathcal{R}$.
- All of the following apply (T_ENUMERATION):
  - t is the enumeration type with labels $\texttt{l}_{1..k}$, that is T_Enum($\texttt{l}_{1..k}$);

    * `d` is the set of all native labels `Label`($\texttt{l}_\texttt{i}$), for $\texttt{i} = 1..k$.

- 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, $\mathbb{Z}$.
- All of the following apply (T_INT_WELL_CONSTRAINED):
    * `t` is the well-constrained integer type T_Int(WellConstrained($\texttt{c}_{1..k}$));
    * `d` is the union of the dynamic domains of each of the constraints $\texttt{c}_{1..k}$ in `env`.
- All of the following apply (CONSTRAINT_EXACT_OKAY):
    * `c` is a constraint consisting of a single side-effect-free expression `e`, that is, Constraint_Exact(`e`);
    * evaluating `e` in `env` results in a configuration with the native integer for $n$;
    * `d` is the set containing the single native integer value for $n$.
- All of the following apply (CONSTRAINT_EXACT_DYNAMIC_ERROR):
    * `c` is a constraint consisting of a single side-effect-free expression `e`, that is, Constraint_Exact(`e`);
    * evaluating `e` in `env` results in a dynamic error configuration;
    * `d` is the empty set.
- All of the following apply (CONSTRAINT_RANGE_OKAY):
    * `c` is a range constraint consisting of a two side-effect-free expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
    * evaluating `e1` in `env` results in a configuration with the native integer for $a$;
    * evaluating `e2` in `env` results in a configuration with the native integer for $b$;
    * `d` is the set containing all native integer values for integers greater or equal to $a$ and less than or equal to $b$.
- All of the following apply (CONSTRAINT_RANGE_DYNAMIC_ERROR1):
    * `c` is a range constraint consisting of a two side-effect-free expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
    * evaluating `e1` in `env` results in a dynamic error configuration;
    * `d` is the empty set.
- All of the following apply (CONSTRAINT_RANGE_DYNAMIC_ERROR2):
    * `c` is a range constraint consisting of a two side-effect-free expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);

  - evaluating `e1` in `env` results in a configuration with the native integer for $a$;
  - evaluating `e2` in `env` results in a dynamic error configuration;
  - `d` is the empty set.

- All of the following apply (T_INT_PARAMETERIZED):
  - `t` is a parameterized integer type for parameter `id`, T_Int(Parameterized(`id`));
  - the native value associated with `id` in the local dynamic environment is the native integer value for $n$;
  - `d` is the set containing the single integer value for $n$.

- All of the following apply (T_BITS_DYNAMIC_ERROR):
  - `t` is a bitvector type with size expression `e`, T_Bits(`e`, _);
  - evaluating `e` in `env` results in a dynamic error configuration;
  - `d` is the empty set.

- All of the following apply (T_BITS_NEGATIVE_WIDTH_ERROR):
  - `t` is a bitvector type with size expression `e`, T_Bits(`e`, _);
  - evaluating `e` in `env` results in a configuration with the native integer for $k$;
  - $k$ is negative;
  - `d` is the empty set.

- All of the following apply (T_BITS_EMPTY):
  - `t` is a bitvector type with size expression `e`, T_Bits(`e`, _);
  - evaluating `e` in `env` results in a configuration with the native integer for 0;
  - `d` is the set containing the single native value for an empty bitvector.

- All of the following apply (T_BITS_NON_EMPTY):
  - `t` is a bitvector type with size expression `e`, T_Bits(`e`, _);
  - evaluating `e` in `env` results in a configuration with the native integer for $k$;
  - $k$ is greater than 0;
  - `d` is the set containing all native values for bitvectors of size exactly $k$.

- All of the following apply (T_TUPLE):
  - `t` is a tuple type over types $\texttt{t}_i$, for $i = 1..k$, T_Tuple($\texttt{t}_{1..k}$);
  - the domain of each element $\texttt{t}_i$ is $D_i$, for $i = 1..k$;
  - evaluating `e` in `env` results in a configuration with the native integer for $k$;

    * `d` is the set containing all native vectors of $k$ values, where the value at position $i$ is from $D_i$.

- All of the following apply:
    * `t` is an integer-indexed array type with length expression `e` and element type `t_elem`, T_Array(ArrayLength_Expr(`e`), `t_elem`);
    * One of the following applies:
        - All of the following apply (T_ARRAY_DYNAMIC_ERROR):
            ▷ evaluating `e` in `env` results in a dynamic error configuration;
            ▷ `d` is the empty set.
        - All of the following apply (T_ARRAY_NEGATIVE_LENGTH_ERROR):
            ▷ evaluating `e` in `env` results in a configuration with the native integer for $k$;
            ▷ $k$ is negative;
            ▷ `d` is the empty set.
        - All of the following apply (T_ARRAY_OKAY):
            ▷ evaluating `e` in `env` results in a configuration with the native integer for $k$;
            ▷ $k$ is greater than or equal to 0;
            ▷ the domain of `t1` is $D_{\texttt{t\_elem}}$;
            ▷ `d` is the set of all native vectors of $k$ values taken from $D_{\texttt{t\_elem}}$.

- All of the following apply (T_ENUM_ARRAY):
    * `t` is an enumeration-indexed array type with for the enumeration `id` with $k$ labels and element type `t_elem`, T_Array(ArrayLength_Enum(`id`, $k$), `t_elem`);
    * view `env` as the pair consisting of the static environment `tenv` and a dynamic environment;
    * the type bound to `id` in the declared_types map of the static environment of `tenv` is the enumeration type for the labels $\texttt{l}_{1..k}$, that is, T_Enum($\texttt{l}_{1..k}$);
    * the dynamic domain of `t_elem` in `env` is $D_{\texttt{t\_elem}}$;
    * `d` is the set of all native records where each $\texttt{l}_i$ is mapped to a value taken from $D_{\texttt{t\_elem}}$, for $i = 1..k$.

- All of the following apply (T_STRUCTURED):
    * `t` is a structured type with typed fields $(\texttt{id}_i, \texttt{t}_i)$, for $i = 1..k$, that is $L([i = 1..k : (\texttt{id}_i, \texttt{t}_i)])$ where $L \in \{\text{T\_Record}, \text{T\_Exception}\}$;
    * the domain of each type $\texttt{t}_i$ is $D_i$, for $i = 1..k$;
    * `d` is the set containing all native records where $\texttt{id}_i$ is mapped to a value taken from $D_i$, for $i = 1..k$.

- All of the following apply (T_NAMED):
  - `t` is a named type with name `id`, T_Named(`id`);
  - the type associated with `id` in `tenv` is `ty`;
  - `d` is the domain of `ty` in `env`.

**Formally**

T_BOOL

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

T_STRING

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

T_REAL

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

T_ENUMERATION

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

T_INT_UNCONSTRAINED

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

T_INT_WELL_CONSTRAINED

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

CONSTRAINT_EXACT_OKAY

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

CONSTRAINT_EXACT_DYNAMIC_ERROR

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

CONSTRAINT_RANGE_OKAY

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

CONSTRAINT_RANGE_DYNAMIC_ERROR1

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

CONSTRAINT_RANGE_DYNAMIC_ERROR2

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

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

T_INT_PARAMETERIZED

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

T_BITS_DYNAMIC_ERROR

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

T_BITS_NEGATIVE_WIDTH_ERROR

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

T_BITS_EMPTY

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

T_BITS_NON_EMPTY

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

T_TUPLE

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

T_ARRAY_DYNAMIC_ERROR

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

T_ARRAY_NEGATIVE_LENGTH_ERROR

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

T_ARRAY_OKAY

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

T_ENUM_ARRAY

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

STRUCTURED

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

T_NAMED

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

**Example: Type Domains**

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

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

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

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

The domain of `enumeration {GREEN, ORANGE, RED}` is the set $\{\texttt{Label}(\texttt{GREEN}), \texttt{Label}(\texttt{ORANGE}), \texttt{Label}(\texttt{RED})\}$ and so is the domain of `type TrafficLights of enumeration {GREEN, ORANGE, RED}`.

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

The domain of `(integer, integer)` is the set containing all pairs of native integer values.

The domain of `record {a: integer; b: boolean}` contains all native records that map `a` to a native integer value and `b` to a native Boolean value.

The dynamic domain of a subprogram parameter `N: integer` is the (singleton) set containing the native integer value $c$, which is assigned to `N` by a given dynamic environment. The static domain of that parameter is the infinite set of all native integer values.

## 13.17 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

$$is\_builtin\_singular(\overbrace{\texttt{ty}}^{\texttt{ty}}) \longrightarrow \mathbb{B}$$

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 13.32 defines variables of builtin singular types `integer`, `real`, `boolean`, `bits(4)`, and `bits(2)`

Listing 13.32: 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 13.33, the builtin singular type `Color` consists in two constants: `RED` and `BLACK`.

Listing 13.33: An enumeration type

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

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

### Formally

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

### TypingRule.BuiltinAggregateType

The predicate

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

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 13.34 provides examples of some builtin aggregate types.

Listing 13.34: 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 used of the integer-typed constant `3`) whereas `PointArray` is declared with the index of `Coord`, which is an enumeration type.

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

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

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

### Builtin Aggregate Exception Types

Listing 13.35 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 13.35: 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

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

### TypingRule.BuiltinSingularOrAggregate

The predicate

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

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

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

### TypingRule.NamedType

The predicate

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

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

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

### TypingRule.AnonymousType

The predicate

$$is\_anonymous(\overbrace{\texttt{ty}}^{\texttt{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is an anonymous type.

### Example: Anonymous Types

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

### 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

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

### TypingRule.SingularType

The predicate

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

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 $\texttt{t1}^{/\!/\#\mathsf{TE}}$;
- applying *is_builtin_singular* to `t1` yields `b`.

### Formally

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

### TypingRule.AggregateType

The predicate

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

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

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

### 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

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

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

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

### TypingRule.NonPrimitiveType

The predicate

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

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

### Formally

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

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

NAMED

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

TUPLE

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

ARRAY

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

STRUCTURED

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

### TypingRule.PrimitiveType

The predicate

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

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

### Example: Primitive Types

The following types are primitive:

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

### Prose

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

**Formally**

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

**TypingRule.Structure**

The function

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

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 terminates[1].

**Example: The Structure of a Type**

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

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

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

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

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

**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 $\texttt{t1}^{/\!/\#\mathsf{TE}}$;
  - obtaining the structure of `t1` static environment tenv yields $\texttt{t}^{/\!/\#\mathsf{TE}}$;
- All of the following apply (BUILTIN_SINGULAR):
  - `ty` is a builtin singular type;

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

    * `t` is `ty`.

- All of the following apply (TUPLE):
    * `ty` is a tuple type with list of types `tys`;
    * the types in `tys` are indexed as $\texttt{t}_i$, for $i = 1..k$;
    * obtaining the structure of each type $\texttt{t}_i$, for $i = 1..k$, in `tys` in the static environment `tenv`, yields $\texttt{t'}_i$ $/\!/_{\texttt{\#TE}}$;
    * `t` is a tuple type with the list of types $\texttt{t'}_i$, for $i = 1..k$.
- All of the following apply (ARRAY):
    * `ty` is an array type of length `e` with element type `t`;
    * obtaining the structure of `t` yields `t1` $/\!/_{\texttt{\#TE}}$;
    * `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 $\texttt{t}_{\texttt{id}}$ $/\!/_{\texttt{\#TE}}$;
    * `t` is a record, a collection or an exception, in correspondence to `ty`, with the list of pairs (`id`, `t_id`);

**Formally**

NAMED

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

BUILTIN_SINGULAR

$$\frac{\mathit{is\_builtin\_singular}(\texttt{ty}) \xrightarrow{\text{type}} \texttt{TRUE}}{\mathit{get\_structure}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{ty}}$$

TUPLE

$$\frac{\texttt{tys} \overset{\text{is}}{=} \texttt{t}_{1..k} \qquad i = 1..k : \mathit{get\_structure}(\texttt{tenv}, \texttt{t}_i) \xrightarrow{\text{type}} \texttt{t'}_i \;/\!/\; \texttt{\#TE}}{\mathit{get\_structure}(\texttt{tenv}, \texttt{T\_Tuple}(\texttt{tys})) \xrightarrow{\text{type}} \texttt{T\_Tuple}(i = 1..k : \texttt{t'}_i)}$$

ARRAY

$$\frac{\mathit{get\_structure}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{t1} \;/\!/\; \texttt{\#TE}}{\mathit{get\_structure}(\texttt{tenv}, \texttt{T\_Array}(\texttt{e}, \texttt{t})) \xrightarrow{\text{type}} \texttt{T\_Array}(\texttt{e}, \texttt{t1})}$$

$$\text{STRUCTURED}\ \frac{L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}, \mathsf{T\_Collection}\} \quad (\mathtt{id}, \mathtt{t}) \in \mathtt{fields} : \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t_{id}}\ /\!/\ \mathsf{\#TE}}{\mathit{get\_structure}(\mathsf{tenv}, L(\mathtt{fields})) \xrightarrow{\mathsf{type}} L([(\mathtt{id}, \mathtt{t}) \in \mathtt{fields} : (\mathtt{id}, \mathtt{t_{id}})])}$$

### TypingRule.MakeAnonymous

The function

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

returns the *underlying type* — `t` — of the type `ty` in the static environment `tenv` or a type error. Intuitively, `ty` is the first non-named type that is used to define `ty`. Unlike *get_structure*, *make_anonymous* replaces named types by their definition until the first non-named type is found but does not recurse further.

### Example: 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 $\mathtt{t1}^{/\!/\mathsf{\#TE}}$;
  - 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**

$$\text{NAMED}\quad \frac{\texttt{ty} \overset{\text{is}}{=} \mathsf{T\_Named}(\texttt{x}) \qquad \mathit{declared\_type}(\texttt{tenv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{t1} \;//\; \#\mathsf{TE} \qquad \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t1}) \xrightarrow{\text{type}} \texttt{t}}{\mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{t}}$$

$$\text{NON-NAMED}\quad \frac{\mathit{ast\_label}(\texttt{ty}) \neq \mathsf{T\_Named}}{\mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{ty}}$$

### TypingRule.CheckConstrainedInteger

A type is a *constrained integer* if it is either a well-constrained integer type or a parameterized integer type.

The function

$$\mathit{check\_constrained\_integer}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{ty}}^{\texttt{ty}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 13.36 shows examples of checking whether a type (used for the width of a bitvector type) is a constrained integer type.

Listing 13.36: 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

$$\mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{WellConstrained}(\_))) \xrightarrow{\text{type}} \mathtt{TRUE}$$

PARAMETERIZED

$$\mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{Parameterized}(\_))) \xrightarrow{\text{type}} \mathtt{TRUE}$$

UNCONSTRAINED

$$\frac{\mathit{ast\_label}(\mathsf{c}) = \mathsf{Unconstrained} \ \vee \ \mathit{ast\_label}(\mathsf{c}) = \mathsf{PendingConstrained}}{\mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{c})) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_UT})}$$

CONFLICTING_TYPE

$$\frac{\mathit{ast\_label}(\mathsf{t}) \neq \mathsf{T\_Int}}{\mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_UT})}$$

## 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 13.37:

| type A | type B | subtypes? | reason |
|---|---|---|---|
| `subInt` | `subInt` | yes | subtyping is reflexive for named types |
| `subInt` | `superInt` | yes | declared as a subtype |
| `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 | `integer` is not a named type |
| `integer` | `integer` | no | `integer` is not a named type |

Listing 13.37: 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

$$is\_subtype(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{ty}}^{\texttt{t1}}, \overbrace{\texttt{ty}}^{\texttt{t2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

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**

REFLEXIVE

$$is\_subtype(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id}), \mathsf{T\_Named}(\mathtt{id})) \xrightarrow{\mathsf{type}} \mathsf{TRUE}$$

TRANSITIVE

$$\frac{G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{id1}) = \mathtt{id3} \qquad \mathtt{id1} \neq \mathtt{id2} \qquad is\_subtype(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id3}), \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{b}}{is\_subtype(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id1}), \mathsf{T\_Named}(\mathtt{id2})) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

NO_SUPERTYPE

$$\frac{\mathtt{id1} \neq \mathtt{id2} \qquad G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{id1}) = \bot}{is\_subtype(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id1}), \mathsf{T\_Named}(\mathtt{id2})) \xrightarrow{\mathsf{type}} \mathsf{FALSE}}$$

NOT_NAMED

$$\frac{(ast\_label(\mathtt{t1}) \neq \mathsf{T\_Named} \vee ast\_label(\mathtt{t2}) \neq \mathsf{T\_Named})}{is\_subtype(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathsf{FALSE}}$$

**TypingRule.SubtypeSatisfaction**

The predicate

$$subtype\_satisfies(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 13.38 shows examples where the types of the right-hand-side expressions subtype-satisfy the types of the left-hand-side expressions.

Listing 13.38: 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[[i-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 13.39 and Listing 13.40 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 13.39: 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 13.40: 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 13.41 shows examples of legal and illegal assignments involving subtyping.

Listing 13.41: 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 13.42 shows more examples of legal and illegal assignments involving subtyping.

Listing 13.42: 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
    * 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` = `FALSE`;
    * either the AST labels of `length_s` and `length_t` are the same or the rule short-circuits with `b` = `FALSE`;
    * `length_s` is an array length expression with `length_expr_s`, that is ArrayLength_Expr(`length_expr_s`);
    * `length_t` is an array length expression with `length_expr_t`, that is ArrayLength_Expr(`length_expr_t`);
    * determining whether expressions `length_expr_s` and `length_expr_t` are equivalent gives `b`.

- All of the following apply (T_ARRAY_ENUM):
    * `s` has the underlying type of an array with index `length_s` and element type `ty_s`, that is T_Array(`length_s`, `ty_s`);
    * `t` has the underlying type of an array with index `length_t` and element type `ty_t`, that is T_Array(`length_t`, `ty_t`);
    * determining whether `ty_s` and `ty_t` are equivalent in `tenv` is either `TRUE` or `FALSE`, which short-circuits the entire rule with `b` = `FALSE`;
    * either the AST labels of `length_s` and `length_t` are the same or the rule short-circuits with `b` = `FALSE`;
    * `length_s` is an array with indices taken from the enumeration `name_s`, that is ArrayLength_Enum(`name_s`, _);

  - `length_t` is an array with indices taken from the enumeration `name_t`, that is ArrayLength_Enum(`name_t`, _);
  - `b` is TRUE if and only if `name_s` and `name_t` are the same.

- All of the following apply (T_TUPLE):
  - `s` has the underlying type of a tuple with type list `lis_s`, that is T_Tuple(`lis_s`);
  - `t` has the underlying type of a tuple with type list `lis_t`, that is T_Tuple(`lis_t`);
  - equating the lengths of `lis_s` and `lis_t` is either TRUE or FALSE, which short-circuits the entire rule with `b` = FALSE;
  - checking at each index `i` of the list `lis_s` whether the type `lis_t`[`i`] type-satisfies the type `lis_s`[`i`] yields $b_i$ // #TE;
  - `b` is TRUE if and only if all $b_i$ are TRUE;

- All of the following apply (STRUCTURED):
  - `s` has the underlying type $L$(`fields_s`), which is a structured type;
  - `t` has the underlying type $L$(`fields_t`), which is a structured type;
  - since both underlying types have the same AST label they are either both record types or both exception types 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**

ERROR1

$$\frac{\mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \#\mathtt{TE}}{\mathit{subtype\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \#\mathtt{TE}}$$

ERROR2

$$\frac{\mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t2} \qquad \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \#\mathtt{TE}}{\mathit{subtype\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \#\mathtt{TE}}$$

DIFFERENT_LABELS

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t2} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s2} \qquad \mathit{ast\_label}(\mathtt{t2}) \neq \mathit{ast\_label}(\mathtt{s2})\end{array}}{\mathit{subtype\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}$$

SIMPLE

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t2} \qquad \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s2} \\ \mathit{ast\_label}(\mathtt{t2}) \in \{\mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Bool}\} \qquad \mathtt{b} := \mathit{ast\_label}(\mathtt{s2}) = \mathit{ast\_label}(\mathtt{t2})\end{array}}{\mathit{subtype\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

T_INT

$$\frac{\mathit{make\_anonymous}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathtt{t2} \quad \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{s2} \quad \mathit{ast\_label}(\mathtt{t2}) = \mathit{ast\_label}(\mathtt{s2}) = \mathsf{T\_Int} \quad \mathit{symdom\_of\_type}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{ds} \quad \mathit{symdom\_of\_type}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathtt{dt} \quad \mathit{symdom\_subset\_unions}(\mathtt{tenv},\mathtt{ds},\mathtt{dt}) \xrightarrow{\text{type}} \mathtt{b}}{\mathit{subtype\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

T_ENUM

$$\frac{\mathit{make\_anonymous}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Enum}(\mathtt{lis\_t}) \quad \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Enum}(\mathtt{lis\_s}) \quad \mathtt{b} := \mathtt{lis\_t} = \mathtt{lis\_s}}{\mathit{subtype\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

T_BITS

$$\frac{\mathit{make\_anonymous}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\mathtt{w\_s},\mathtt{bfs\_s}) \quad \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\mathtt{w\_t},\mathtt{bfs\_t}) \quad \mathit{bitfields\_included}(\mathtt{tenv},\mathtt{bfs\_s},\mathtt{bfs\_t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \quad \mathit{symdom\_of\_width\_expr}(\mathtt{tenv},\mathtt{w\_s}) \xrightarrow{\text{type}} \mathtt{ds} \quad \mathit{symdom\_of\_width\_expr}(\mathtt{tenv},\mathtt{w\_t}) \xrightarrow{\text{type}} \mathtt{dt} \quad \mathit{symdom\_subset\_unions}(\mathtt{tenv},\mathtt{ds},\mathtt{dt}) \xrightarrow{\text{type}} \mathtt{b}}{\mathit{subtype\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

T_ARRAY_EXPR

$$\frac{\mathit{make\_anonymous}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\mathtt{length\_s},\mathtt{ty\_s}) \quad \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\mathtt{length\_t},\mathtt{ty\_t}) \quad \mathit{type\_equal}(\mathtt{tenv},\mathtt{ty\_s},\mathtt{ty\_t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \quad \mathit{bool\_transition}(\mathit{ast\_label}(\mathtt{length\_s}) = \mathit{ast\_label}(\mathtt{length\_t})) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \quad \mathtt{length\_s} \overset{\text{is}}{=} \mathsf{ArrayLength\_Expr}(\mathtt{length\_expr\_s}) \quad \mathtt{length\_t} \overset{\text{is}}{=} \mathsf{ArrayLength\_Expr}(\mathtt{length\_expr\_t}) \quad \mathit{expr\_equal}(\mathtt{tenv},\mathtt{length\_expr\_s},\mathtt{length\_expr\_t}) \xrightarrow{\text{type}} \mathtt{b}}{\mathit{subtype\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

T_ARRAY_ENUM

$$\frac{\mathit{make\_anonymous}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\mathtt{length\_s},\mathtt{ty\_s}) \quad \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\mathtt{length\_t},\mathtt{ty\_t}) \quad \mathit{type\_equal}(\mathtt{tenv},\mathtt{ty\_s},\mathtt{ty\_t}) \xrightarrow{\text{type}} \mathtt{TRUE} \quad \mathit{bool\_transition}(\mathit{ast\_label}(\mathtt{length\_s}) = \mathit{ast\_label}(\mathtt{length\_t})) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \quad \mathtt{length\_s} \overset{\text{is}}{=} \mathsf{ArrayLength\_Enum}(\mathtt{name\_s},\_) \quad \mathtt{length\_t} \overset{\text{is}}{=} \mathsf{ArrayLength\_Enum}(\mathtt{name\_t},\_) \quad \mathtt{b} := \mathtt{name\_s} = \mathtt{name\_t}}{\mathit{subtype\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

T_TUPLE

$$\frac{\begin{array}{c} \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Tuple}(\mathtt{lis\_s}) \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Tuple}(\mathtt{lis\_t}) \\ \mathit{equal\_length}(\mathtt{lis\_s}, \mathtt{lis\_t}) \xrightarrow{\text{type}} \mathtt{TRUE} \mathbin{/\!/} \mathtt{FALSE} \\ \mathtt{i} \in \mathit{indices}(\mathtt{lis\_s}) : \mathit{type\_satisfies}(\mathtt{tenv}, \mathtt{lis\_t}[\mathtt{i}], \mathtt{lis\_s}[\mathtt{i}]) \xrightarrow{\text{type}} \mathtt{b}_i \mathbin{/\!/} \mathsf{TTypeError} \\ \mathtt{b} := \bigwedge_{\mathtt{i}=1}^{k} \mathtt{b}_\mathtt{i} \end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

For a list of typed fields `fields`, we define the set of its field identifiers as:

$$\mathit{field\_names}(\mathtt{fields}) \triangleq \{\mathtt{id} \mid (\mathtt{id}, \mathtt{t}) \in \mathtt{fields}\}$$

We define the type associated with the field name `id` in a list of typed fields `fields`, if there is a unique one, as follows:

$$\mathit{field\_type}(\mathtt{fields}, \mathtt{id}) \triangleq \begin{cases} \mathtt{t} & \text{if } \{\mathtt{t'} \mid (\mathtt{id}, \mathtt{t'}) \in \mathtt{fields}\} = \{\mathtt{t}\} \\ \bot & \text{otherwise} \end{cases}$$

STRUCTURED

$$\frac{\begin{array}{c} L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}, \mathsf{T\_Collection}\} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\text{type}} L(\mathtt{fields\_s}) \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} L(\mathtt{fields\_t}) \\ \mathtt{names\_s} := \mathit{field\_names}(\mathtt{fields\_s}) \qquad \mathtt{names\_t} := \mathit{field\_names}(\mathtt{fields\_t}) \\ \mathit{bool\_transition}(\mathtt{names\_s} \subseteq \mathtt{names\_t}) \longrightarrow \mathtt{TRUE} \mathbin{/\!/} \mathtt{FALSE} \\ (\mathtt{id}, \mathtt{ty\_s}) \in \mathtt{fields\_s} : \mathit{type\_equal}(\mathtt{tenv}, \mathtt{ty\_s}, \mathit{field\_type}(\mathtt{fields\_t}, \mathtt{id})) \xrightarrow{\text{type}} \mathtt{b}_\mathtt{id} \\ \mathtt{b} := \bigwedge_{\mathtt{id} \in \mathtt{names\_s}} \mathtt{b}_\mathtt{id} \end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{b}}$$

### TypingRule.TypeSatisfaction

The predicate

$$\mathit{type\_satisfies}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{ty}}^{\text{t}}, \overbrace{\mathsf{ty}}^{\text{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\text{TE}}$$

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 13.43, `var pair: pairT = (1, dataT1)` is legal since the right-hand-side has anonymous, non-primitive type `(integer, T1)`.

Listing 13.43: 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)
return 0;
end;
```

### Example: More Type-satisfaction Examples

In Listing 13.44, `pair = (1, dataAsInt);` is legal since the right-hand-side has anonymous, primitive type `(integer, integer)`.

Listing 13.44: 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 13.45, `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 13.45: 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;
```

### 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 $\texttt{TRUE}^{/\!/\#\texttt{TE}}$;
  - `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`;
  - $(\texttt{b1} \vee \texttt{b2}) \wedge \texttt{b3}$ 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 $\texttt{width\_s}^{/\!/\#\texttt{TE}}$;
  - determining whether `width_t` and `width_s` are bitwidth-equivalent yields `b`.
- All of the following apply (OTHERWISE1):
  - `t` does not subtype `s` in `tenv`;

  - determining whether `t` is anonymous yields `b1`;
  - determining whether `s` is anonymous yields `b2`;
  - determining whether `t` subtype-satisfies `s` in `tenv` yields `b3`;
  - (`b1` ∨ `b2`) ∧ `b3` is `FALSE`;
  - obtaining the structure of `s` in `tenv` yields a `s_struct`$^{\#TE}$;
  - at least one of `t` and `s_struct` is not a bitvector type;

- All of the following apply (OTHERWISE2):
  - `t` does not subtype `s` in `tenv`;
  - determining whether `t` is anonymous yields `b1`;
  - determining whether `s` is anonymous yields `b2`;
  - determining whether `t` subtype-satisfies `s` in `tenv` yields `b3`;
  - (`b1` ∨ `b2`) ∧ `b3` is `FALSE`;
  - obtaining the structure of `s` in `tenv` yields a `s_struct`$^{\#TE}$;
  - both `t` and `s_struct` are bitvector types;
  - the bitvector type `t` has a non-empty list of bitfields;
  - `b` is `FALSE`;

**Formally**

SUBTYPES

$$\frac{\mathit{is\_subtype}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}{\mathit{type\_satisfies}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

ANONYMOUS

$$\frac{\mathit{is\_subtype}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_anonymous}(\mathsf{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b1} \qquad \mathit{is\_anonymous}(\mathsf{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b2} \qquad \mathtt{b1} \vee \mathtt{b2} \qquad \mathit{subtype\_satisfies}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}{\mathit{type\_satisfies}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

T_BITS

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{is\_anonymous}(\mathsf{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b1} \qquad \mathit{is\_anonymous}(\mathsf{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b2} \\ \mathit{subtype\_satisfies}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b3} \qquad \neg((\mathtt{b1} \vee \mathtt{b2}) \wedge \mathtt{b3}) \\ \mathtt{t} = \mathsf{T\_Bits}(\mathtt{width\_t},[\,]) \qquad \mathit{get\_structure}(\mathsf{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\mathtt{width\_s},\_) \;//\; \mathtt{\#TE} \\ \mathit{bitwidth\_equal}(\mathsf{tenv},\mathtt{width\_t},\mathtt{width\_s}) \xrightarrow{\mathsf{type}} \mathtt{b}\end{array}}{\mathit{type\_satisfies}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

OTHERWISE1

$$\frac{\begin{array}{c} \mathit{is\_subtype}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b1} \\ \mathit{is\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b2} \qquad \mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b3} \\ \neg((\mathtt{b1} \vee \mathtt{b2}) \wedge \mathtt{b3}) \qquad \mathit{get\_structure}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s\_struct} \\ \mathit{ast\_label}(\mathtt{t}) \neq \mathsf{T\_Bits} \vee \mathit{ast\_label}(\mathtt{s\_struct}) \neq \mathsf{T\_Bits} \end{array}}{\mathit{type\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

OTHERWISE2

$$\frac{\begin{array}{c} \mathit{is\_subtype}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b1} \\ \mathit{is\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b2} \qquad \mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b3} \\ \neg((\mathtt{b1} \vee \mathtt{b2}) \wedge \mathtt{b3}) \qquad \mathit{get\_structure}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s\_struct} \\ \mathit{ast\_label}(\mathtt{t}) = \mathsf{T\_Bits} \wedge \mathit{ast\_label}(\mathtt{s\_struct}) = \mathsf{T\_Bits} \\ \mathtt{t} = \mathsf{T\_Bits}(\mathtt{width\_t}, \mathtt{bitfields}) \qquad \mathtt{bitfields} \neq [\,] \end{array}}{\mathit{type\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

### TypingRule.CheckTypeSatisfaction

We also define

$$\mathit{checked\_typesat}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \{\mathtt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

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 13.43, checking whether `(integer, T1)` type-satisfies `pairT` for the assignment `var pair: pairT = (1, dataT1)` yields `TRUE`.

In Listing 13.45, checking whether `(intege, 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**

$$\text{OKAY}\quad \frac{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}{\mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

$$\text{ERROR}\quad \frac{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}{\mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_TSF})}$$

### 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

$$\mathit{lowest\_common\_ancestor}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{ty}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

returns the lowest common ancestor of types `t` and `s` in `tenv` — `ty`. The result is a type error if a lowest common ancestor does not exist or a type error is detected.

### Example: Lowest Common Ancestor

Listing 13.46 shows examples of conditional expressions and the resulting *lowest common ancestor*.

Listing 13.46: 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                          Type 1         Type 2
    var x : integer{N, M} = if nondet() then N         else M;
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                                  Type 1                Type 2
    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                          Type 1             Type 2
    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;
```

**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 $\texttt{ty}^{/\!/\texttt{\#TE}}$.
- 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 $\texttt{TRUE}^{/\!/\texttt{TE\_LCA}}$;
  - ▷ `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 $\texttt{b\_equal\_length}^{/\!/\texttt{\#TE}}$;
  - ▷ checking that `b_equal_length` is `TRUE` yields $\texttt{TRUE}^{/\!/\texttt{TE\_LCA}}$;
  - ▷ the lowest common ancestor of `ty_t` and `ty_s` is $\texttt{t1}^{/\!/\texttt{\#TE}}$;
  - ▷ `ty` is an array type with width expression `width_s` and element type `t1`.
- All of the following apply (T_TUPLE):
  - ▷ `t` is a tuple type with type list `lis_t`;
  - ▷ `s` is a tuple type with type list `lis_s`;

- ▷ checking whether `lis_t` and `lis_s` have the same number of elements yields `TRUE` or a type error, which short-circuits the entire rule (indicating that the number of elements in both tuples is expected to be the same and thus there is no lowest common ancestor);
- ▷ applying *lowest_common_ancestor* to `lis_t`[`i`] and `lis_s`[`i`] in `tenv`, for every position of `lis_t`, yields $\texttt{t}_\texttt{i}$ // `#TE`;
- ▷ define `li` to be the list of types $\texttt{t}_\texttt{i}$, for every position of `lis_t`;
- ▷ define `ty` as the tuple type with list of types `li`, that is, T_Tuple(`li`).

– All of the following apply (ERROR):

- ▷ either the AST labels of `t` and `s` are different, or one of them is T_Enum, T_Record, 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`.

TYPE_EQUAL

$$\frac{\mathit{type\_equal}(\text{tenv}, \texttt{t}, \texttt{s}) \xrightarrow{\text{type}} \texttt{TRUE}}{\mathit{lowest\_common\_ancestor}(\text{tenv}, \texttt{t}, \texttt{s}) \xrightarrow{\text{type}} \overbrace{\texttt{s}}^{\texttt{ty}}}$$

NAMED_SUBTYPE1

$$\frac{\begin{array}{c}\texttt{t} = \text{T\_Named}(\texttt{name\_s}) \\ \texttt{s} = \text{T\_Named}(\texttt{name\_t}) \qquad \mathit{type\_equal}(\text{tenv}, \texttt{t}, \texttt{s}) \xrightarrow{\text{type}} \texttt{FALSE} \\ \mathit{named\_lowest\_common\_ancestor}(\text{tenv}, \texttt{name\_s}, \texttt{name\_t}) \xrightarrow{\text{type}} \texttt{None} \;//\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\text{tenv}, \texttt{s}) \xrightarrow{\text{type}} \texttt{anon\_s} \;//\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{anon\_t} \;//\; \texttt{\#TE} \\ \mathit{lowest\_common\_ancestor}(\text{tenv}, \texttt{anon\_t}, \texttt{anon\_s}) \xrightarrow{\text{type}} \texttt{ty} \;//\; \texttt{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\text{tenv}, \texttt{t}, \texttt{s}) \xrightarrow{\text{type}} \texttt{ty}}$$

NAMED_SUBTYPE2

$$\frac{\begin{array}{c}\texttt{t} = \text{T\_Named}(\texttt{name\_s}) \\ \texttt{s} = \text{T\_Named}(\texttt{name\_t}) \qquad \mathit{type\_equal}(\text{tenv}, \texttt{t}, \texttt{s}) \xrightarrow{\text{type}} \texttt{FALSE} \\ \mathit{named\_lowest\_common\_ancestor}(\text{tenv}, \texttt{name\_s}, \texttt{name\_t}) \xrightarrow{\text{type}} \langle\texttt{name}\rangle \;//\; \texttt{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\text{tenv}, \texttt{t}, \texttt{s}) \xrightarrow{\text{type}} \overbrace{\text{T\_Named}(\texttt{name})}^{\texttt{ty}}}$$

ONE_NAMED1

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ (\mathit{ast\_label}(\mathtt{t}) = \mathsf{T\_Named} \vee \mathit{ast\_label}(\mathtt{s}) = \mathsf{T\_Named}) \\ \mathit{ast\_label}(\mathtt{t}) \neq \mathit{ast\_label}(\mathtt{s}) \qquad \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{anon\_s} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{anon\_t} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{type\_equal}(\mathtt{tenv},\mathtt{anon\_t},\mathtt{anon\_s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \\ \mathtt{ty} := \mathsf{choice}(\mathit{ast\_label}(\mathtt{t}) = \mathsf{T\_Named},\mathtt{t},\mathtt{s})\end{array}}{\mathit{lowest\_common\_ancestor}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}$$

ONE_NAMED2

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ (\mathit{ast\_label}(\mathtt{t}) = \mathsf{T\_Named} \vee \mathit{ast\_label}(\mathtt{s}) = \mathsf{T\_Named}) \\ \mathit{ast\_label}(\mathtt{t}) \neq \mathit{ast\_label}(\mathtt{s}) \qquad \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{anon\_s} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{anon\_t} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{type\_equal}(\mathtt{tenv},\mathtt{anon\_t},\mathtt{anon\_s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{lowest\_common\_ancestor}(\mathtt{tenv},\mathtt{anon\_t},\mathtt{anon\_s}) \xrightarrow{\mathsf{type}} \mathtt{ty} \;\; /\!/ \;\; \mathtt{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}$$

T_INT_UNCONSTRAINED

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{ast\_label}(\mathtt{t}) = \mathit{ast\_label}(\mathtt{s}) = \mathsf{T\_Int} \\ \mathsf{is\_unconstrained\_integer}(\mathtt{t}) \vee \mathsf{is\_unconstrained\_integer}(\mathtt{s})\end{array}}{\mathit{lowest\_common\_ancestor}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{unconstrained\_integer}}^{\mathtt{ty}}}$$

T_INT_PARAMETERIZED

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{ast\_label}(\mathtt{t}) = \mathit{ast\_label}(\mathtt{s}) = \mathsf{T\_Int} \\ \neg\mathsf{is\_unconstrained\_integer}(\mathtt{t}) \qquad \neg\mathsf{is\_unconstrained\_integer}(\mathtt{s}) \\ \mathsf{is\_parameterized\_integer}(\mathtt{t}) \vee \mathsf{is\_parameterized\_integer}(\mathtt{s}) \\ \mathit{to\_well\_constrained}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t1} \qquad \mathit{to\_well\_constrained}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s1} \\ \mathit{lowest\_common\_ancestor}(\mathtt{tenv},\mathtt{t1},\mathtt{s1}) \xrightarrow{\mathsf{type}} \mathtt{ty} \;\; /\!/ \;\; \mathtt{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathtt{t} = \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs\_t},\mathtt{p1})) \\ \mathtt{s} = \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs\_s},\mathtt{p2})) \qquad \mathtt{p} := \mathit{precision\_join}(\mathtt{p1},\mathtt{p2})\end{array}}{\mathit{lowest\_common\_ancestor}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs\_t} + \mathtt{cs\_s},\mathtt{p}))}^{\mathtt{ty}}}$$

T_BITS

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathit{expr\_equal}(\mathsf{tenv},\mathtt{e\_t},\mathtt{e\_s}) \xrightarrow{\text{type}} \mathtt{b\_equal} \qquad \mathit{check}(\mathtt{b\_equal},\mathtt{TE\_LCA}) \longrightarrow \mathtt{TRUE} \;//\; \mathtt{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv},\overbrace{\mathsf{T\_Bits}(\mathtt{e\_t},\_)}^{\mathtt{t}},\overbrace{\mathsf{T\_Bits}(\mathtt{e\_s},\_)}^{\mathtt{s}}) \xrightarrow{\text{type}} \overbrace{\mathsf{T\_Bits}(\mathtt{e\_t},[\,])}^{\mathtt{ty}}}$$

T_ARRAY

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathit{array\_length\_equal}(\mathsf{tenv},\mathtt{width\_t},\mathtt{width\_s}) \xrightarrow{\text{type}} \mathtt{b\_equal\_length} \;//\; \mathtt{\#TE} \\ \mathit{check}(\mathtt{b\_equal\_length},\mathtt{TE\_LCA}) \longrightarrow \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \mathit{lowest\_common\_ancestor}(\mathsf{tenv},\mathtt{ty\_t},\mathtt{ty\_s}) \xrightarrow{\text{type}} \mathtt{t1} \;//\; \mathtt{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv},\overbrace{\mathsf{T\_Array}(\mathtt{width\_t},\mathtt{ty\_t})}^{\mathtt{t}},\overbrace{\mathsf{T\_Array}(\mathtt{width\_s},\mathtt{ty\_s})}^{\mathtt{s}}) \xrightarrow{\text{type}} \overbrace{\mathsf{T\_Array}(\mathtt{width\_t},\mathtt{t1})}^{\mathtt{ty}}}$$

T_TUPLE

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathit{equal\_length}(\mathtt{lis\_t},\mathtt{lis\_s}) \xrightarrow{\text{type}} \mathtt{b} \qquad \mathit{check}(\mathtt{b},\mathtt{TE\_LCA}) \xrightarrow{\text{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \mathtt{i} \in \mathit{indices}(\mathtt{lis\_t}) : \mathit{lowest\_common\_ancestor}(\mathsf{tenv},\mathtt{lis\_t}[\mathtt{i}],\mathtt{lis\_s}[\mathtt{i}]) \xrightarrow{\text{type}} \mathtt{t}_\mathtt{i} \;//\; \mathtt{\#TE} \\ \mathtt{li} := [\mathtt{i} \in \mathit{indices}(\mathtt{lis\_t}) : \mathtt{t}_\mathtt{i}]\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv},\overbrace{\mathsf{T\_Tuple}(\mathtt{lis\_t})}^{\mathtt{t}},\overbrace{\mathsf{T\_Tuple}(\mathtt{lis\_s})}^{\mathtt{s}}) \xrightarrow{\text{type}} \overbrace{\mathsf{T\_Tuple}(\mathtt{li})}^{\mathtt{ty}}}$$

ERROR

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ (\mathit{ast\_label}(\mathtt{t}) \neq \mathit{ast\_label}(\mathtt{s})) \vee \mathit{ast\_label}(\mathtt{t}) \in \{\mathsf{T\_Enum},\mathsf{T\_Record},\mathsf{T\_Exception},\mathsf{T\_Collection}\}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_LCA})}$$

### TypingRule.ApplyUnopType

The function

$$\mathit{apply\_unop\_type}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}},\overbrace{\mathsf{unop}}^{\mathtt{op}},\overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{s}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

determines the result type of applying a unary operator when the type of its operand is known. Similarly, we determine the negation of integer constraints. Otherwise, the result is a type error.

### Example: Applying Unary Operations to Types

Listing 13.47 shows examples of typing applications of unary operations.

Listing 13.47: Applying unary operations to types

```
func main() => integer
begin
    //        result type                              input type
    var b   : boolean                       = ! TRUE    as boolean;
    var i1  : integer{-5}                   = - (5      as integer{5});
    var i2  : integer{-(-5)}                = - i1;
    var ci1 : integer{0..5, 9, 10..8}       = 4         as integer{0..5, 9, 10..8};
    var ci2 : integer{-9, -8..-10, -5..0}   = - (ci1    as integer{0..5, 9, 10..8});
    var ui1 : integer                       = 9         as integer;
    var ui2 : integer                       = - ui1     as integer;

    var r1  : real                          = - 5.0     as real;
    var r2  : real                          = - r1      as real;

    var bv1 : bits(8) {[0] flag}            = Zeros{8}  as bits(8) {[0] flag};
    var bv2 : bits(8) {[0] flag}            = NOT bv1   as bits(8) {[0] flag};
    return 0;
end;
```

### Prose

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**

BNOT_T_BOOL

$$\frac{\mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t1}, \mathsf{T\_Bool}) \xrightarrow{\text{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \mathtt{BNOT}, \mathtt{t1}) \xrightarrow{\text{type}} \mathsf{T\_Bool}}$$

NEG_ERROR

$$\frac{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathsf{unconstrained\_integer}) \xrightarrow{\text{type}} \mathtt{FALSE} \;//\; \mathtt{\#TE} \qquad \mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Real}) \xrightarrow{\text{type}} \mathtt{FALSE} \;//\; \mathtt{\#TE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NEG}}^{\text{op}}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_BO})}$$

NEG_T_REAL

$$\frac{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Real}) \xrightarrow{\text{type}} \mathtt{TRUE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NEG}}^{\text{op}}, \mathtt{t}) \xrightarrow{\text{type}} \overbrace{\mathsf{T\_Real}}^{\text{s}}}$$

NEG_T_INT_UNCONSTRAINED

$$\frac{\mathit{get\_well\_constrained\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{unconstrained\_integer} \;//\; \mathtt{\#TE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NEG}}^{\text{op}}, \mathtt{t}) \xrightarrow{\text{type}} \overbrace{\mathsf{unconstrained\_integer}}^{\text{s}}}$$

NEG_T_INT_WELL_CONSTRAINED

$$\frac{\mathit{get\_well\_constrained\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{vcs})) \quad \mathtt{c} \in \mathtt{vcs} : \mathit{negate\_constraint}(\mathtt{c}) \xrightarrow{\text{type}} \mathtt{neg_c} \quad \mathtt{cs\_new} := [\mathtt{c} \in \mathtt{vcs} : \mathtt{neg_c}]}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NEG}}^{\text{op}}, \mathtt{t}) \xrightarrow{\text{type}} \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs\_new}))}^{\text{s}}}$$

NOT_T_BITS

$$\frac{\mathit{check\_structure}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Bits}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NOT}}^{\text{op}}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t}}$$

### TypingRule.NegateConstraint

The helper function

$$\mathit{negate\_constraint}(\overbrace{\mathsf{int\_constraint}}^{\text{c}}) \longrightarrow \overbrace{\mathsf{int\_constraint}}^{\text{new\_c}}$$

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

$$\mathit{negate\_constraint}(\overbrace{\mathsf{Constraint\_Exact}(\mathtt{e})}^{\mathtt{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Exact}(\mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{e}))}^{\mathtt{new\_c}}$$

RANGE

$$\mathit{negate\_constraint}(\overbrace{\mathsf{Constraint\_Range}(\mathtt{v\_start}, \mathtt{v\_end})}^{\mathtt{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Range}(\mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{v\_end}), \mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{v\_start}))}^{\mathtt{new\_c}}$$

### TypingRule.ApplyBinopTypes

The function

$$\mathit{apply\_binop\_types}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{binop}}^{\mathtt{op}}, \overbrace{\mathsf{ty}}^{\mathtt{t1}}, \overbrace{\mathsf{ty}}^{\mathtt{t2}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{t}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

determines the result type `t` of applying the binary operator `op` to operands of type `t1` and `t2` in the static environment tenv. Otherwise, the result is a type error.

### Example: Applying Binary Operations to Types

Listing 13.48 shows examples of typing binary operations.

Listing 13.48: 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 13.49 shows examples of typing binary operations applied to constrained integer types.

Importantly, note that the AST for constraints is not closed under binary operations. For example, given a range constraint `A..B` and an exact constraint |2|, there is no AST to express `(A..B) * 2`. Therefore, they 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 13.49: 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 `l1` and `l2` are the AST labels of any

singular types:

| op | *ast_label*(t1_anon) | *ast_label*(t2_anon) |
|---|---|---|
| AND | T_Bool | T_Bool |
| OR | T_Bool | T_Bool |
| EQ_OP | T_Bool | T_Bool |
| IMPL | T_Bool | T_Bool |
| AND | T_Bits | T_Bits |
| OR | T_Bits | T_Bits |
| XOR | T_Bits | T_Bits |
| PLUS | T_Bits | T_Bits |
| MINUS | T_Bits | T_Bits |
| CONCAT | l1 | l2 |
| PLUS | T_Bits | T_Int |
| MINUS | T_Bits | T_Int |
| LEQ | T_Int | T_Int |
| GEQ | T_Int | T_Int |
| GT | T_Int | T_Int |
| LT | T_Int | T_Int |
| LEQ | T_Real | T_Real |
| GEQ | T_Real | T_Real |
| GT | T_Real | T_Real |
| LT | T_Real | T_Real |
| EQ_OP | T_Int | T_Int |
| NEQ | T_Int | T_Int |
| EQ_OP | T_Bool | T_Bool |
| NEQ | T_Bool | T_Bool |
| EQ_OP | T_Real | T_Real |
| NEQ | T_Real | T_Real |
| EQ_OP | T_String | T_String |
| NEQ | T_String | T_String |
| MUL | T_Int | T_Int |
| DIV | T_Int | T_Int |
| DIVRM | T_Int | T_Int |
| MOD | T_Int | T_Int |
| SHL | T_Int | T_Int |
| SHR | T_Int | T_Int |
| POW | T_Int | T_Int |
| PLUS | T_Int | T_Int |
| MINUS | T_Int | T_Int |
| PLUS | T_Real | T_Real |
| MINUS | T_Real | T_Real |
| MUL | T_Real | T_Real |
| RDIV | T_Real | T_Real |
| POW | T_Real | T_Int |
| PLUS | T_Real | T_Real |
| MINUS | T_Real | T_Real |
| MUL | T_Real | T_Real |
| POW | T_Real | T_Int |
| RDIV | T_Real | T_Real |

**Formally**

NAMED

$$\frac{\begin{array}{c} \mathit{ast\_label}(\texttt{t1}) = \texttt{T\_Named} \lor \mathit{ast\_label}(\texttt{t2}) = \texttt{T\_Named} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t1}) \xrightarrow{\text{type}} \texttt{t1\_anon} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{t2\_anon} \;/\!/\; \texttt{\#TE} \\ \mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1\_anon}, \texttt{t2\_anon}) \xrightarrow{\text{type}} \texttt{t} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{t}}$$

BOOLEAN

$$\frac{\texttt{op} \in \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}, \texttt{EQ\_OP}\}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Bool}}^{\texttt{t1}}, \overbrace{\texttt{T\_Bool}}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bool}}^{\texttt{t}}}$$

BITS_ARITH

$$\frac{\begin{array}{c} \texttt{op} \in \{\texttt{AND}, \texttt{OR}, \texttt{XOR}, \texttt{PLUS}, \texttt{MINUS}\} \\ \mathit{check\_bits\_equal\_width}(\texttt{tenv}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Bits}(\texttt{w1}, \_)}^{\texttt{t1}}, \overbrace{\texttt{T\_Bits}(\texttt{w2}, \_)}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bits}(\texttt{w1}, [\,])}^{\texttt{t}}}$$

BITS_INT

$$\frac{\texttt{op} \in \{\texttt{PLUS}, \texttt{MINUS}\}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Bits}(\texttt{w}, \_)}^{\texttt{t1}}, \overbrace{\texttt{T\_Int}(\_)}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bits}(\texttt{w}, [\,])}^{\texttt{t}}}$$

BITS_CONCAT

$$\frac{\texttt{w} := \texttt{E\_Binop}(\texttt{PLUS}, \texttt{w1}, \texttt{w2}) \qquad \mathit{normalize}(\texttt{tenv}, \texttt{w}) \xrightarrow{\text{type}} \texttt{w'}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{CONCAT}, \overbrace{\texttt{T\_Bits}(\texttt{w1}, \_)}^{\texttt{t1}}, \overbrace{\texttt{T\_Bits}(\texttt{w2}, \_)}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bits}(\texttt{w'}, [\,])}^{\texttt{t}}}$$

STRING_CONCAT

$$\frac{\begin{array}{c} \texttt{t1} \neq \texttt{T\_Bits}(\_, \_) \lor \texttt{t2} \neq \texttt{T\_Bits}(\_, \_) \\ \mathit{check}(\mathit{is\_singular}(\texttt{t1}), \texttt{TE\_UT}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{is\_singular}(\texttt{t2}), \texttt{TE\_UT}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{CONCAT}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_String}}^{\texttt{t}}}$$

REL

$$(\mathtt{op}, \mathtt{t1}, \mathtt{t2}) \in \left\{ \begin{array}{lll} (\mathtt{LEQ} & , \mathtt{T\_Int} & , \mathtt{T\_Int}) \\ (\mathtt{GEQ} & , \mathtt{T\_Int} & , \mathtt{T\_Int}) \\ (\mathtt{GT} & , \mathtt{T\_Int} & , \mathtt{T\_Int}) \\ (\mathtt{LT} & , \mathtt{T\_Int} & , \mathtt{T\_Int}) \\ (\mathtt{LEQ} & , \mathtt{T\_Real} & , \mathtt{T\_Real}) \\ (\mathtt{GEQ} & , \mathtt{T\_Real} & , \mathtt{T\_Real}) \\ (\mathtt{GT} & , \mathtt{T\_Real} & , \mathtt{T\_Real}) \\ (\mathtt{LT} & , \mathtt{T\_Real} & , \mathtt{T\_Real}) \\ (\mathtt{EQ\_OP} & , \mathtt{T\_Int} & , \mathtt{T\_Int}) \\ (\mathtt{NEQ} & , \mathtt{T\_Int} & , \mathtt{T\_Int}) \\ (\mathtt{EQ\_OP} & , \mathtt{T\_Bool} & , \mathtt{T\_Bool}) \\ (\mathtt{NEQ} & , \mathtt{T\_Bool} & , \mathtt{T\_Bool}) \\ (\mathtt{EQ\_OP} & , \mathtt{T\_Real} & , \mathtt{T\_Real}) \\ (\mathtt{NEQ} & , \mathtt{T\_Real} & , \mathtt{T\_Real}) \\ (\mathtt{EQ\_OP} & , \mathtt{T\_String} & , \mathtt{T\_String}) \\ (\mathtt{NEQ} & , \mathtt{T\_String} & , \mathtt{T\_String}) \end{array} \right\}$$

$$\mathit{apply\_binop\_types}(\mathtt{tenv}, \mathtt{op}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathtt{type}} \overbrace{\mathtt{T\_Bool}}^{\mathtt{t}}$$

EQ_NEQ_BITS

$$\frac{\mathtt{op} \in \{\mathtt{EQ\_OP}, \mathtt{NEQ}\} \qquad \mathit{check\_bits\_equal\_width}(\mathtt{tenv}, \mathtt{t1\_anon}, \mathtt{t2\_anon}) \xrightarrow{\mathtt{type}} \mathtt{TRUE} \ /\!/ \ \mathtt{\#TE}}{\mathit{apply\_binop\_types}(\mathtt{tenv}, \mathtt{op}, \overbrace{\mathtt{T\_Bits}(\mathtt{w1}, \_)}^{\mathtt{t1}}, \overbrace{\mathtt{T\_Bits}(\mathtt{w2}, \_)}^{\mathtt{t2}}) \xrightarrow{\mathtt{type}} \overbrace{\mathtt{T\_Bool}}^{\mathtt{t}}}$$

EQ_NEQ_ENUM

$$\frac{\mathtt{op} \in \{\mathtt{EQ\_OP}, \mathtt{NEQ}\} \qquad \mathit{check}(\mathtt{li1} = \mathtt{li2}, \mathtt{DifferentEnumLabels}) \longrightarrow \mathtt{TRUE} \ /\!/ \ \mathtt{\#TE}}{\mathit{apply\_binop\_types}(\mathtt{tenv}, \mathtt{op}, \overbrace{\mathtt{T\_Enum}(\mathtt{li1})}^{\mathtt{t1}}, \overbrace{\mathtt{T\_Enum}(\mathtt{li2})}^{\mathtt{t2}}) \xrightarrow{\mathtt{type}} \overbrace{\mathtt{T\_Bool}}^{\mathtt{t}}}$$

ARITH_T_INT_UNCONSTRAINED

$$\frac{\mathtt{op} \in \{\mathtt{MUL}, \mathtt{DIV}, \mathtt{DIVRM}, \mathtt{MOD}, \mathtt{SHL}, \mathtt{SHR}, \mathtt{POW}, \mathtt{PLUS}, \mathtt{MINUS}\} \qquad \mathtt{c1} = \mathtt{Unconstrained} \vee \mathtt{c2} = \mathtt{Unconstrained}}{\mathit{apply\_binop\_types}(\mathtt{tenv}, \mathtt{op}, \overbrace{\mathtt{T\_Int}(\mathtt{c1})}^{\mathtt{t1}}, \overbrace{\mathtt{T\_Int}(\mathtt{c2})}^{\mathtt{t2}}) \xrightarrow{\mathtt{type}} \mathtt{unconstrained\_integer}}$$

ARITH_T_INT_PARAMETERIZED

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{MUL}, \texttt{DIV}, \texttt{DIVRM}, \texttt{MOD}, \texttt{SHL}, \texttt{SHR}, \texttt{POW}, \texttt{PLUS}, \texttt{MINUS}\} \\ \mathit{ast\_label}(\texttt{c1}) = \mathsf{Parameterized} \vee \mathit{ast\_label}(\texttt{c2}) = \mathsf{Parameterized} \\ \mathit{ast\_label}(\texttt{c1}) \neq \mathsf{Unconstrained} \wedge \mathit{ast\_label}(\texttt{c2}) \neq \mathsf{Unconstrained} \\ \mathit{to\_well\_constrained}(\texttt{t1}) \xrightarrow{\mathsf{type}} \texttt{t1\_well\_constrained} \\ \mathit{to\_well\_constrained}(\texttt{t2}) \xrightarrow{\mathsf{type}} \texttt{t2\_well\_constrained} \\ \mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{t1\_well\_constrained}, \texttt{t2\_well\_constrained}) \xrightarrow{\mathsf{type}} \texttt{t} \;\; /\!\!/ \;\; \texttt{\#TE}\end{array}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\mathsf{T\_Int}(\texttt{c1})}^{\texttt{t1}}, \overbrace{\mathsf{T\_Int}(\texttt{c2})}^{\texttt{t2}}) \xrightarrow{\mathsf{type}} \texttt{t}}$$

ARITH_T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{MUL}, \texttt{POW}, \texttt{PLUS}, \texttt{MINUS}, \texttt{DIVRM}, \texttt{DIV}, \texttt{MOD}, \texttt{SHL}, \texttt{SHR}\} \\ \texttt{c1} = \mathsf{WellConstrained}(\texttt{cs2}, \texttt{p1}) \qquad \texttt{c2} = \mathsf{WellConstrained}(\texttt{cs1}, \texttt{p2}) \\ \mathit{annotate\_constraint\_binop}(\texttt{tenv}, \texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\mathsf{type}} (\texttt{cs}, \texttt{p3}) \;\; /\!\!/ \;\; \texttt{\#TE} \\ \texttt{p} = \mathit{precision\_join}(\texttt{p1}, \mathit{precision\_join}(\texttt{p2}, \texttt{p3}))\end{array}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\mathsf{T\_Int}(\texttt{c1})}^{\texttt{t1}}, \overbrace{\mathsf{T\_Int}(\texttt{c2})}^{\texttt{t2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{cs}, \texttt{p}))}^{\texttt{t}}}$$

ARITH_REAL

$$\frac{(\texttt{op}, \texttt{t1}, \texttt{t2}) \in \left\{ \begin{array}{lcll} (\texttt{PLUS} & , & \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\texttt{MINUS} & , & \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\texttt{MUL} & , & \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\texttt{POW} & , & \mathsf{T\_Real} & , \mathsf{T\_Int}) \\ (\texttt{RDIV} & , & \mathsf{T\_Real} & , \mathsf{T\_Real}) \end{array} \right\}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Real}}^{\texttt{t}}}$$

ERROR

$$\frac{\begin{array}{l} \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t1}) \xrightarrow{\mathsf{type}} \mathtt{t1\_anon} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{t2\_anon} \;/\!/\; \mathtt{\#TE} \\ (\mathtt{op}, \mathit{ast\_label}(\mathtt{t1\_anon}), \mathit{ast\_label}(\mathtt{t2\_anon})) \notin \left\{ \begin{array}{lll} (\mathtt{AND} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{OR} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{IMPL} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{AND} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{OR} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{XOR} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{PLUS} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{MINUS} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{CONCAT} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{PLUS} & , \mathsf{T\_Bits} & , \mathsf{T\_Int}) \\ (\mathtt{MINUS} & , \mathsf{T\_Bits} & , \mathsf{T\_Int}) \\ (\mathtt{LEQ} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{GEQ} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{GT} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{LT} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{LEQ} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{GEQ} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{GT} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{LT} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{NEQ} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{NEQ} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{NEQ} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_String} & , \mathsf{T\_String}) \\ (\mathtt{NEQ} & , \mathsf{T\_String} & , \mathsf{T\_String}) \\ (\mathtt{MUL} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{DIV} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{DIVRM} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{MOD} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{SHL} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{SHR} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{POW} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{PLUS} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{MINUS} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{PLUS} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{MINUS} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{MUL} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{RDIV} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{POW} & , \mathsf{T\_Real} & , \mathsf{T\_Int}) \\ (\mathtt{PLUS} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{MINUS} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{MUL} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{POW} & , \mathsf{T\_Real} & , \mathsf{T\_Int}) \\ (\mathtt{RDIV} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \end{array} \right\} \\ \cup \{(\mathtt{CONCAT}, \mathit{ast\_label}(\mathtt{t1}), \mathit{ast\_label}(\mathtt{t2})) \mid \mathit{is\_singular}(\mathtt{t1}) \wedge \mathit{is\_singular}(\mathtt{t2})\} \end{array}}{\mathit{apply\_binop\_types}(\mathtt{tenv}, \mathtt{op}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_BO})}$$

### TypingRule.FindNamedLCA

The function

$$\mathit{named\_lowest\_common\_ancestor}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathsf{ty}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

returns the lowest common named super type — `ty` — of the types `t` and `s` in tenv.

The helper function

$$\mathit{supers}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \mathcal{P}(\mathsf{ty})$$

returns the set of *named supertypes* of a type `t` in the subtypes function of a global static environment tenv:

$$\mathit{supers}(\mathsf{tenv}, \mathtt{t}) \triangleq \begin{cases} \{\mathtt{t}\} \cup \mathit{supers}(\mathtt{s}) & \text{if } G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{t}) = \mathtt{s} \\ \{\mathtt{t}\} & \text{otherwise (that is, } G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{t}) = \bot) \end{cases}$$

### Example: Finding Named Lowest Common Ancestors

In Listing 13.50, 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 13.50: 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` — `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**

FOUND

$$\frac{\mathit{supers}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_supers} \qquad \mathtt{s} \in \mathtt{t\_supers}}{\mathit{named\_lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s}}$$

SUPER

$$\frac{\mathit{supers}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_supers} \qquad \mathtt{s} \notin \mathtt{t\_supers} \qquad G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{s}) = \mathtt{s'} \qquad \mathit{named\_lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t}, \mathtt{s'}) \xrightarrow{\mathsf{type}} \mathtt{ty}}{\mathit{named\_lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}$$

NONE

$$\frac{\mathit{supers}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_supers} \qquad \mathtt{s} \notin \mathtt{t\_supers} \qquad G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{s}) = \bot}{\mathit{named\_lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{None}}$$

### TypingRule.AnnotateConstraintBinop

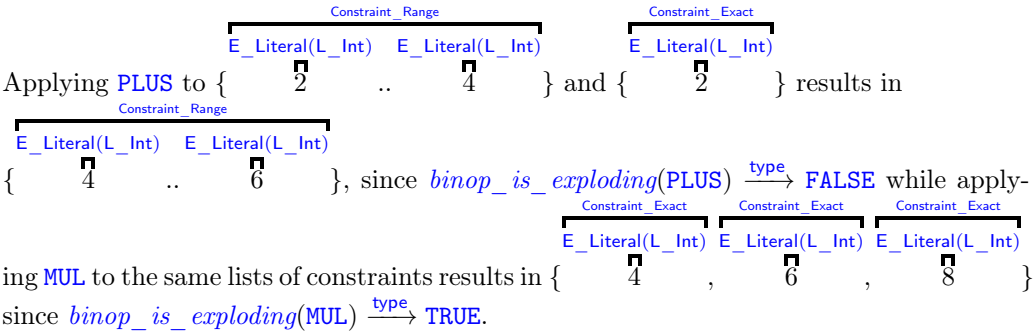
The function

$$\mathit{annotate\_constraint\_binop}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{binop}}^{\mathtt{op}}, \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{cs1}}, \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{cs2}}) \longrightarrow \left(\overbrace{\mathsf{int\_constraint}^*}^{\mathtt{annotated\_cs}}, \overbrace{\mathsf{precision\_loss\_indicator}}^{\mathtt{p}}\right) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

annotates the application of the binary operation `op` to the lists of integer constraints `cs1` and `cs2`, yielding a list of constraints — `annotated_cs`. Otherwise, the result is a type error.

The operator `op` is assumed to be only one of the operators in the following set: {`SHL`, `SHR`, `POW`, `MOD`, `DIVRM`, `MINUS`, `MUL`, `PLUS`, `DIV`}. The rule employs *binop_is_exploding* to decide whether range constraints can be maintained as range constraints or have to be converted to a list of exact constraints.

**Example: Annotating Constraints for Binary Operations**




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

$$\frac{\begin{array}{c}\mathit{binop\_filter\_rhs}(\mathtt{tenv},\mathtt{op},\mathtt{cs2}) \xrightarrow{\text{type}} \mathtt{cs2\_f} \\ \text{******** common prefix ********} \\ \mathit{binop\_is\_exploding}(\mathtt{op}) \xrightarrow{\text{type}} \mathtt{TRUE} \qquad \mathit{explode\_intervals}(\mathtt{tenv},\mathtt{cs1}) \xrightarrow{\text{type}} (\mathtt{cs1\_e},\mathtt{p1}) \\ \mathit{explode\_intervals}(\mathtt{tenv},\mathtt{cs2\_f}) \xrightarrow{\text{type}} (\mathtt{cs2\_e},\mathtt{p2}) \qquad \mathtt{p0} := \mathit{precision\_join}(\mathtt{p1},\mathtt{p2}) \\ \mathtt{expected\_constraint\_length} := \mathsf{choice}(\mathtt{op} = \mathtt{MOD}, |\mathtt{cs2\_e}|, |\mathtt{cs1\_e}| \times |\mathtt{cs2\_e}|) \\ (\mathtt{cs1\_arg},\mathtt{cs2\_arg},\mathtt{p}) := \begin{array}{ll}\textbf{if} & \mathtt{expected\_constraint\_length} < 2^{17} \text{ then} \\ & (\mathtt{cs1\_e},\mathtt{cs2\_e},\mathtt{p0}) \\ \textbf{else} & \\ & (\mathtt{cs1},\mathtt{cs2\_f},\mathsf{Precision\_Lost})\end{array} \\ \text{******** common suffix ********} \\ \mathit{constraint\_binop}(\mathtt{op},\mathtt{cs1\_arg},\mathtt{cs2\_arg}) \xrightarrow{\text{type}} \mathtt{cs\_vanilla} \\ \mathit{refine\_constraint\_for\_div}(\mathtt{op},\mathtt{cs\_vanilla}) \xrightarrow{\text{type}} \mathtt{refined\_cs} \;//\; \mathtt{\#TE} \\ \mathit{reduce\_constraints}(\mathtt{tenv},\mathtt{refined\_cs}) \xrightarrow{\text{type}} \mathtt{annotated\_cs}\end{array}}{\mathit{annotate\_constraint\_binop}(\mathtt{tenv},\mathtt{op},\mathtt{cs1},\mathtt{cs2}) \xrightarrow{\text{type}} \mathtt{annotated\_cs}}$$

NON_EXPLODING

$$\frac{\begin{array}{c}\mathit{binop\_filter\_rhs}(\mathtt{tenv},\mathtt{op},\mathtt{cs2}) \xrightarrow{\text{type}} \mathtt{cs2\_f} \\ \text{******** common prefix ********} \\ \mathit{binop\_is\_exploding}(\mathtt{op}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathtt{p} := \mathsf{Precision\_Full} \qquad (\mathtt{cs1\_arg},\mathtt{cs2\_arg}) := (\mathtt{cs1},\mathtt{cs2\_f}) \\ \text{******** common suffix ********} \\ \mathit{constraint\_binop}(\mathtt{op},\mathtt{cs1\_arg},\mathtt{cs2\_arg}) \xrightarrow{\text{type}} \mathtt{cs\_vanilla} \\ \mathit{refine\_constraint\_for\_div}(\mathtt{op},\mathtt{cs\_vanilla}) \xrightarrow{\text{type}} \mathtt{refined\_cs} \;//\; \mathtt{\#TE} \\ \mathit{reduce\_constraints}(\mathtt{tenv},\mathtt{refined\_cs}) \xrightarrow{\text{type}} \mathtt{annotated\_cs}\end{array}}{\mathit{annotate\_constraint\_binop}(\mathtt{tenv},\mathtt{op},\mathtt{cs1},\mathtt{cs2}) \xrightarrow{\text{type}} \mathtt{annotated\_cs}}$$

### TypingRule.BinopFilterRhs

The function

$$\mathit{binop\_filter\_rhs}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{binop}}^{\mathtt{op}}, \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{cs}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{new\_cs}}$$

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 13.49, 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 $\texttt{TRUE}^{/\!/\texttt{TE\_BO}}$.
- 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 $\texttt{TRUE}^{/\!/\texttt{TE\_BO}}$.
- All of the following apply (NO_FILTERING):
  - `op` is one of `MINUS`, `MUL`, and `PLUS`;
  - `new_cs` is `cs`.

### Formally

GREATER_OR_EQUAL

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{SHL}, \texttt{SHR}, \texttt{POW}\} \\ \texttt{f} := \mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \lambda x.\ x \geq 0) \\ \mathit{refine\_constraints}(\texttt{cs}, \texttt{f}) \xrightarrow{\text{type}} \texttt{new\_cs} \qquad \mathit{check}(\texttt{new\_cs} \neq [\,], \texttt{TE\_BO}) \xrightarrow{\text{type}} \texttt{TRUE}\ /\!/\ \texttt{\#TE}\end{array}}{\mathit{binop\_filter\_rhs}(\mathsf{tenv}, \mathsf{op}, \texttt{cs}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

GREATER_THAN

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{MOD}, \texttt{DIV}, \texttt{DIVRM}\} \\ \texttt{f} := \mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \lambda x.\ x > 0) \\ \mathit{refine\_constraints}(\texttt{cs}, \texttt{f}) \xrightarrow{\text{type}} \texttt{new\_cs} \qquad \mathit{check}(\texttt{new\_cs} \neq [\,], \texttt{TE\_BO}) \xrightarrow{\text{type}} \texttt{TRUE}\ /\!/\ \texttt{\#TE}\end{array}}{\mathit{binop\_filter\_rhs}(\mathsf{tenv}, \mathsf{op}, \texttt{cs}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

NO_FILTERING

$$\frac{\texttt{op} \in \{\texttt{MINUS}, \texttt{MUL}, \texttt{PLUS}\}}{\mathit{binop\_filter\_rhs}(\mathsf{op}, \texttt{cs}) \xrightarrow{\text{type}} \overbrace{\texttt{cs}}^{\texttt{new\_cs}}}$$

**TypingRule.RefineConstraintBySign**

The function

$$\mathit{refine\_constraint\_by\_sign}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{Z} \to \mathbb{B}}^{\texttt{p}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c}}) \longrightarrow \overbrace{\langle \mathsf{int\_constraint} \rangle}^{\texttt{c\_opt}}$$

takes a predicate `p` that returns `TRUE` based on the sign of its input. The function conservatively refines the constraint `c` in `tenv` by applying symbolic reasoning to yield a new constraint (inside an optional) that represents the values that satisfy the `c` and for which `p` holds. In this context, conservatively means that the new constraint may represent a superset of the values that a more precise reasoning may yield. If the set of those values is empty the result is `None`.

**Prose**

One of the following applies:

- All of the following apply (EXACT_REDUCES_TO_Z):
  - `c` is an exact constraint for the expression `e`, that is, Constraint_Exact(`e`);
  - applying *reduce_to_z_opt* to `e` in `tenv`, in order to symbolically simplify `e` to an integer, yields ⟨`z`⟩;
  - `c_opt` is ⟨`c`⟩ if `p` holds for `z` and `None` otherwise.
- All of the following apply (EXACT_DOES_NOT_REDUCE_TO_Z):
  - `c` is an exact constraint for the expression `e`, that is, Constraint_Exact(`e`);
  - applying *reduce_to_z_opt* to `e` in `tenv`, in order to symbolically simplify `e` to an integer, yields `None`;
  - `c_opt` is ⟨`c`⟩.
- All of the following apply (RANGE_BOTH_REDUCE_TO_Z):
  - `c` is a range constraint for the expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *reduce_to_z_opt* to `e1` in `tenv`, in order to symbolically simplify `e1` to an integer, yields ⟨`z1`⟩;
  - applying *reduce_to_z_opt* to `e2` in `tenv`, in order to symbolically simplify `e2` to an integer, yields ⟨`z2`⟩;
  - One of the following applies: (defining `c_opt`)
    - if `p` is `TRUE` for both `z1` and `z2`, define `c_opt` as ⟨`c`⟩;
    - if `p` is `FALSE` for `z1` and `TRUE` for `z2`, define `c_opt` as the optional range constraint where the bottom expression is the literal expression for 0 if `p` holds for 0 and the literal expression for 1 otherwise, and the top expression is `e2`;

  - if p is TRUE for z1 and FALSE for z2, define c_opt as the optional range constraint where the bottom expression is e1 and the top expression is the literal expression for 0 if p holds for 0 and the literal expression for −1 otherwise;
  - if p is FALSE for both z1 and z2, define c_opt as None.

- All of the following apply (ONLY_E1_REDUCES_TO_Z):
  * c is a range constraint for the expressions e1 and e2, that is, Constraint_Range(e1, e2);
  * applying *reduce_to_z_opt* to e1 in tenv, in order to symbolically simplify e1 to an integer, yields ⟨z1⟩;
  * applying *reduce_to_z_opt* to e2 in tenv, in order to symbolically simplify e2 to an integer, yields None;
  * One of the following applies: (defining c_opt):
    - if p is TRUE for z1, define c_opt as ⟨c⟩;
    - if p is FALSE for z1, define c_opt as the optional range constraint with the bottom expression as the literal expression for 0 if p holds for 0 and the literal expression for 1 otherwise, and the top expression e2.

- All of the following apply (ONLY_E2_REDUCES_TO_Z):
  * c is a range constraint for the expressions e1 and e2, that is, Constraint_Range(e1, e2);
  * applying *reduce_to_z_opt* to e1 in tenv, in order to symbolically simplify e1 to an integer, yields None;
  * applying *reduce_to_z_opt* to e2 in tenv, in order to symbolically simplify e2 to an integer, yields ⟨z2⟩;
  * One of the following applies (defining c_opt):
    - if p is TRUE for z2, define c_opt as ⟨c⟩;
    - if p is FALSE for z2, define c_opt as the optional range constraint with the bottom expression e1 and the top expression the literal expression for 0 if p holds for 0 and the literal expression for −1 otherwise.

- All of the following apply (NONE_REDUCE_TO_Z):
  * c is a range constraint for the expressions e1 and e2, that is, Constraint_Range(e1, e2);
  * applying *reduce_to_z_opt* to e1 in tenv, in order to symbolically simplify e1 to an integer, yields None;
  * applying *reduce_to_z_opt* to e2 in tenv, in order to symbolically simplify e2 to an integer, yields None;
  * define c_opt as c.

**Formally**

EXACT_REDUCES_TO_Z

$$\frac{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \langle \mathsf{z} \rangle \qquad \mathtt{c\_opt} := \mathsf{choice}(\mathsf{p}(\mathsf{z}), \langle \mathsf{c} \rangle, \mathsf{None})}{\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\mathsf{type}} \mathtt{c\_opt}}$$

EXACT_DOES_NOT_REDUCE_TO_Z

$$\frac{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{None}}{\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\mathsf{type}} \overbrace{\langle \mathsf{c} \rangle}^{\mathtt{c\_opt}}}$$

RANGE_BOTH_REDUCE_TO_Z

$$\frac{\begin{array}{c}\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \langle \mathsf{z1} \rangle \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \langle \mathsf{z2} \rangle \\ \mathtt{c\_opt} := \\ \hookrightarrow \begin{cases} \langle \mathsf{c} \rangle & \text{if } \mathsf{p}(\mathsf{z1}) \wedge \mathsf{p}(\mathsf{z2}) \\ \langle \mathsf{Constraint\_Range}(\mathsf{choice}(\mathsf{p}(0), \overset{\mathsf{E\_Literal(L\_Int)}}{0}, \overset{\mathsf{E\_Literal(L\_Int)}}{1}), \mathsf{e2}) \rangle & \text{if } \neg\mathsf{p}(\mathsf{z1}) \wedge \mathsf{p}(\mathsf{z2}) \\ \langle \mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{choice}(\mathsf{p}(0), \overset{\mathsf{E\_Literal(L\_Int)}}{0}, \overset{\mathsf{E\_Literal(L\_Int)}}{-1})) \rangle & \text{if } \mathsf{p}(\mathsf{z1}) \wedge \neg\mathsf{p}(\mathsf{z2}) \\ \mathsf{None} & \text{if } \neg\mathsf{p}(\mathsf{z1}) \wedge \neg\mathsf{p}(\mathsf{z2}) \end{cases} \end{array}}{\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) \xrightarrow{\mathsf{type}} \mathtt{c\_opt}}$$

ONLY_E1_REDUCES_TO_Z

$$\frac{\begin{array}{c}\mathsf{c} = \mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2}) \\ \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \langle \mathsf{z1} \rangle \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathsf{None} \\ \mathtt{c\_opt} := \\ \hookrightarrow \begin{cases} \langle \mathsf{c} \rangle & \text{if } \mathsf{p}(\mathsf{z1}) \\ \langle \mathsf{Constraint\_Range}(\mathsf{choice}(\mathsf{p}(0), \overset{\mathsf{E\_Literal(L\_Int)}}{0}, \overset{\mathsf{E\_Literal(L\_Int)}}{1}), \mathsf{e2}) \rangle & \text{else} \end{cases} \end{array}}{\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \mathsf{c}) \xrightarrow{\mathsf{type}} \mathtt{c\_opt}}$$

ONLY_E2_REDUCES_TO_Z

$$\frac{\begin{array}{c}\mathsf{c} = \mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2}) \\ \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathsf{None} \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \langle \mathsf{z2} \rangle \\ \mathtt{c\_opt} := \\ \hookrightarrow \begin{cases} \langle \mathsf{c} \rangle & \text{if } \mathsf{p}(\mathsf{z2}) \\ \langle \mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{choice}(\mathsf{p}(0), \overset{\mathsf{E\_Literal(L\_Int)}}{0}, \overset{\mathsf{E\_Literal(L\_Int)}}{-1})) \rangle & \text{else} \end{cases} \end{array}}{\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \mathsf{c}) \xrightarrow{\mathsf{type}} \mathtt{c\_opt}}$$

NONE_REDUCE_TO_Z

$$\frac{\mathsf{c} = \mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2}) \quad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} \mathtt{None} \quad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} \mathtt{None}}{\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \mathsf{c}) \xrightarrow{\text{type}} \overbrace{\mathsf{c}}^{\mathsf{c\_opt}}}$$

### TypingRule.ReduceToZOpt

The function

$$\mathit{reduce\_to\_z\_opt}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \overbrace{\langle \mathbb{Z} \rangle}^{\mathsf{z\_opt}}$$

returns an integer inside an optional if `e` can be symbolically simplified into an integer in `tenv` and `None` otherwise. The expression `e` is assumed to appear in a constraint for a type that has been successfully annotated, which means that applying *normalize* to it should not yield a type error.

### Prose

One of the following applies:

- All of the following apply (NORMALIZES_TO_Z):
  - symbolically simplifying `e` in `tenv` via *normalize* yields a literal expression for the integer `z`;
  - define `z_opt` as ⟨`z`⟩.
- All of the following apply (DOES_NOT_NORMALIZE_TO_Z):
  - symbolically simplifying `e` in `tenv` via *normalize* yields an expression that is not an integer literal;
  - define `z_opt` as `None`.

### Formally

NORMALIZES_TO_Z

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \overset{\mathsf{E\_Literal(L\_Int)}}{\boxed{\mathsf{z}}}}{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \overbrace{\langle \mathsf{z} \rangle}^{\mathsf{z\_opt}}}$$

DOES_NOT_NORMALIZE_TO_Z

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \mathtt{e'} \quad \forall \mathsf{z} \in \mathbb{Z}.\ \mathtt{e'} \neq \overset{\mathsf{E\_Literal(L\_Int)}}{\boxed{\mathsf{z}}}}{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \overbrace{\mathtt{None}}^{\mathsf{z\_opt}}}$$

### TypingRule.RefineConstraints

The function

$$\mathit{refine\_constraints}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{int\_constraint} \to \langle \mathsf{int\_constraint} \rangle}^{\texttt{f}}, \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\texttt{new\_cs}}$$

refines a list of constraints `cs` by applying the refinement function `f` to each constraint and retaining the constraints that do not refine to `None`. The resulting list of constraints is given in `new_cs`.

### Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `cs` is the empty list;
  - `new_cs` is the empty list.
- All of the following apply (NON_EMPTY_NONE):
  - `cs` is the list with `c` as its `head` and `cs1` as its `tail`;
  - applying `f` to `c` yields `None`;
  - applying *refine_constraints* to `f` and `cs1` yields `cs1'`;
  - `new_cs` is `cs1'`.
- All of the following apply (NON_EMPTY_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

EMPTY

$$\mathit{refine\_constraints}(\mathsf{tenv}, \mathtt{f}, \overbrace{[\,]}^{\texttt{cs}}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\texttt{new\_cs}}$$

NON_EMPTY_NONE

$$\frac{\mathtt{f}(\mathtt{c}) \xrightarrow{\text{type}} \mathtt{None} \qquad \mathit{refine\_constraints}(\mathtt{f}, \mathtt{cs1}) \xrightarrow{\text{type}} \mathtt{cs1'}}{\mathit{refine\_constraints}(\mathtt{f}, \overbrace{[\mathtt{c}] + \mathtt{cs1}}^{\texttt{cs}}) \xrightarrow{\text{type}} \overbrace{\mathtt{cs1'}}^{\texttt{new\_cs}}}$$

$$\frac{\text{NON\_EMPTY\_SOME}\quad \texttt{f(c)} \xrightarrow{\text{type}} \langle \texttt{c'} \rangle \qquad \textit{refine\_constraints}(\texttt{f}, \texttt{cs1}) \xrightarrow{\text{type}} \texttt{cs1'}}{\textit{refine\_constraints}(\texttt{f}, \overbrace{[\texttt{c}] + \texttt{cs1}}^{\texttt{cs}}) \xrightarrow{\text{type}} \overbrace{[\texttt{c'}] + \texttt{cs1'}}^{\texttt{new\_cs}}}$$

### TypingRule.RefineConstraintForDiv

The function

$$\textit{refine\_constraint\_for\_div}(\overbrace{\text{binop}}^{\texttt{op}}, \overbrace{\text{int\_constraint}^*}^{\texttt{cs}}) \longrightarrow \overbrace{\text{int\_constraint}^*}^{\texttt{res}} \cup \overbrace{\text{TTypeError}}^{\#\texttt{TE}}$$

filters the list of constraints `cs` for `op`, removing constraints that represents a division operation that will definitely fail when `op` is the division operation. Otherwise, the result is a type error.

### Prose

One of the following applies:

- All of the following apply (DIV):
  - `op` is DIV;
  - applying *filter_reduce_constraint_div* to each constraint `cs[i]`, for each `i` in *indices*(`cs`), yields the optional constraint $\texttt{c\_opt}_\texttt{i}$ $/\!/^{\#\texttt{TE}}$;
  - define `res` as the list made of constraints $\texttt{c'}_\texttt{i}$, for each `i` in *indices*(`cs`) such that $\texttt{c\_opt}_\texttt{i} = \langle \texttt{c'}_\texttt{i} \rangle$;
  - checking that `res` is not the empty list yields $\text{TRUE}^{/\!/\texttt{TE\_BO}}$.
- All of the following apply (NON_DIV):
  - `op` is not DIV;
  - define `res` as `cs`.

### Formally

$$\frac{\begin{array}{l}\text{DIV}\\ \texttt{op} = \text{DIV} \qquad \texttt{i} \in \textit{indices}(\texttt{cs}) : \textit{filter\_reduce\_constraint\_div}(\texttt{cs[i]}) \xrightarrow{\text{type}} \texttt{c\_opt}_\texttt{i} \;/\!/\; \#\texttt{TE}\\ \qquad\qquad \texttt{res} := [\texttt{i} \in \textit{indices}(\texttt{cs}) : \text{choice}(\texttt{c\_opt}_\texttt{i} = \langle \texttt{c'}_\texttt{i} \rangle, \texttt{c'}, \epsilon)]\\ \qquad\qquad \textit{check}(\texttt{res} \neq [\,], \texttt{TE\_BO}) \longrightarrow \text{TRUE} \;/\!/\; \#\texttt{TE}\end{array}}{\textit{refine\_constraint\_for\_div}(\texttt{op}, \texttt{cs}) \xrightarrow{\text{type}} \texttt{res}}$$

$$\frac{\text{NON\_DIV}\quad \texttt{op} \neq \text{DIV}}{\textit{refine\_constraint\_for\_div}(\texttt{op}, \texttt{cs}) \xrightarrow{\text{type}} \overbrace{\texttt{cs}}^{\texttt{res}}}$$

### TypingRule.FilterReduceConstraintDiv

The function

$$\mathit{filter\_reduce\_constraint\_div}(\overbrace{\mathsf{int\_constraint}}^{\texttt{c}}) \longrightarrow \overbrace{\langle \mathsf{int\_constraint} \rangle}^{\texttt{c\_opt}}$$

returns `None` if `c` is an exact constraint for a binary expression for dividing two integer literals where the denominator does not divide the numerator and an optional containing `c`. The result is returned in `c_opt`. This is used to conservatively test whether `c` would always fail dynamically.

### Prose

One of the following applies:

- All of the following apply (EXACT):
  - `c` is an exact constraint for the expression `e`, that is, Constraint_Exact(`e`);
  - applying *get_literal_div_opt* to `e` yields $\langle (\texttt{z1}, \texttt{z2}) \rangle^{/\!/\texttt{None}}$;
  - define `c_opt` as follows:
    - `None`, if `z2` is positive and `z2` does not divide `z1`;
    - $\langle \texttt{c} \rangle$, 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 (`z1`, `z2`) and `z2` is positive;
    - `None`, otherwise.
  - applying *get_literal_div_opt* to `e2` yields `e2_opt`;
  - define `z2_opt` as follows:
    - `z3` divided by `z4` and rounded down, if `e2_opt` is (`z3`, `z4`) and `z4` is positive;
    - `None`, otherwise.
  - define `c_opt` as follows:
    - the exact constraint for the literal integer `z5`, if `z1_opt` is $\langle \texttt{z5} \rangle$ and `z2_opt` is $\langle \texttt{z6} \rangle$ and `z5` is equal to `z6`;
    - the range constraint for the literal integer `z5` and `z6`, if `z1_opt` is $\langle \texttt{z5} \rangle$ and `z2_opt` is $\langle \texttt{z6} \rangle$ and `z5` is less than `z6`;
    - `None`, if `z1_opt` is $\langle \texttt{z5} \rangle$ and `z2_opt` is $\langle \texttt{z6} \rangle$ and `z5` is greater than `z6`;
    - the range constraint for the literal integer `z5` and `e2`, if `z1_opt` is $\langle \texttt{z5} \rangle$ and `z2_opt` is `None`;
    - the range constraint for `e1` and the literal integer `z6`, if `z1_opt` is `None` and `z2_opt` is $\langle \texttt{z6} \rangle$;
    - `c` if `z1_opt` is `None` and `z2_opt` is `None`.

**Formally**

EXACT

$$\frac{\mathit{get\_literal\_div\_opt}(\mathtt{e}) \xrightarrow{\text{type}} \langle(\mathtt{z1}, \mathtt{z2})\rangle \;/\!/\; \mathtt{None} \qquad \mathtt{c\_opt} := \begin{cases} \mathtt{None} & \text{if } \mathtt{z2} > 0 \wedge \frac{\mathtt{z1}}{\mathtt{z2}} \notin \mathbb{Z} \\ \langle \mathtt{c} \rangle & \text{else} \end{cases}}{\mathit{filter\_reduce\_constraint\_div}(\mathtt{tenv}, \overbrace{\mathsf{Constraint\_Exact}(\mathtt{e})}^{\mathtt{c}}) \xrightarrow{\text{type}} \mathtt{c\_opt}}$$

RANGE

$$\frac{\begin{array}{c} \mathit{get\_literal\_div\_opt}(\mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1\_opt} \\ \mathtt{z1\_opt} := \begin{cases} \left\lceil \frac{\mathtt{z1}}{\mathtt{z2}} \right\rceil & \text{if } \mathtt{e1\_opt} = \langle(\mathtt{z1}, \mathtt{z2})\rangle \wedge \mathtt{z2} > 0 \\ \mathtt{None} & \text{else} \end{cases} \\ \mathit{get\_literal\_div\_opt}(\mathtt{e2}) \xrightarrow{\text{type}} \mathtt{e2\_opt} \\ \mathtt{z2\_opt} := \begin{cases} \left\lceil \frac{\mathtt{z3}}{\mathtt{z4}} \right\rceil & \text{if } \mathtt{e2\_opt} = \langle(\mathtt{z3}, \mathtt{z4})\rangle \wedge \mathtt{z4} > 0 \\ \mathtt{None} & \text{else} \end{cases} \end{array} \qquad \mathtt{c\_opt} := \begin{cases} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int})}^{\mathsf{Constraint\_Exact}} & \\ \langle \quad \overbrace{\mathtt{z5}}^{} \quad \rangle & \text{if } \mathtt{z1\_opt} = \langle \mathtt{z5} \rangle \wedge \mathtt{z2\_opt} = \langle \mathtt{z6} \rangle \wedge \mathtt{z5} = \mathtt{z6} \\ \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int})\ \mathsf{E\_Literal}(\mathsf{L\_Int})}^{\mathsf{Constraint\_Range}} & \\ \langle \quad \overbrace{\mathtt{z5}}^{} \quad .. \quad \overbrace{\mathtt{z6}}^{} \quad \rangle & \text{if } \mathtt{z1\_opt} = \langle \mathtt{z5} \rangle \wedge \mathtt{z2\_opt} = \langle \mathtt{z6} \rangle \wedge \mathtt{z5} < \mathtt{z6} \\ \mathtt{None} & \text{if } \mathtt{z1\_opt} = \langle \mathtt{z5} \rangle \wedge \mathtt{z2\_opt} = \langle \mathtt{z6} \rangle \wedge \mathtt{z5} > \mathtt{z6} \\ \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int})}^{\mathsf{Constraint\_Range}} & \\ \langle \quad \overbrace{\mathtt{z5}}^{} \quad ..\mathtt{e2} \rangle & \text{if } \mathtt{z1\_opt} = \langle \mathtt{z5} \rangle \wedge \mathtt{z2\_opt} = \mathtt{None} \\ \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int})}^{\mathsf{Constraint\_Range}} & \\ \langle \mathtt{e1}.. \quad \overbrace{\mathtt{z6}}^{} \quad \rangle & \text{if } \mathtt{z1\_opt} = \mathtt{None} \wedge \mathtt{z2\_opt} = \langle \mathtt{z6} \rangle \\ \langle \mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2}) \rangle & \text{if } \mathtt{z1\_opt} = \mathtt{None} \wedge \mathtt{z2\_opt} = \mathtt{None} \end{cases}}{\mathit{filter\_reduce\_constraint\_div}(\mathtt{tenv}, \overbrace{\mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{c}}) \xrightarrow{\text{type}} \overbrace{\langle \mathtt{c} \rangle}^{\mathtt{c\_opt}}}$$

**TypingRule.GetLiteralDivOpt**

The function

$$\mathit{get\_literal\_div\_opt}(\overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\langle \mathbb{Z} \times \mathbb{Z} \rangle}^{\mathtt{range\_opt}}$$

matches the expression `e` to a binary operation expression over the division operation and two literal integer expressions. If `e` matches this pattern the result `range_opt` is an

optional containing the pair of integers appearing in the operand expressions. Otherwise, the result is `None`.

### Prose

The value `range_opt` is $\langle(\texttt{z1}, \texttt{z2})\rangle$ if `e` is a binary operation expression over the division operation and two literal integer expressions for the integers `z1` and `z2` and `None` otherwise.

### Formally

$$\frac{\texttt{range\_opt} := \mathsf{choice}(\texttt{e} = \mathsf{E\_Binop}(\texttt{DIV}, \overbrace{\lceil \texttt{z1} \rceil}^{\mathsf{E\_Literal(L\_Int)}}, \overbrace{\lceil \texttt{z2} \rceil}^{\mathsf{E\_Literal(L\_Int)}}), \langle(\texttt{z1}, \texttt{z2})\rangle, \texttt{None})}{\mathit{get\_literal\_div\_opt}(\texttt{e}) \xrightarrow{\mathsf{type}} \texttt{range\_opt}}$$

### TypingRule.ExplodeIntervals

The function

$$\mathit{explode\_intervals}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs}}) \longrightarrow \left(\overbrace{\mathsf{int\_constraint}^*}^{\texttt{new\_cs}}, \overbrace{\mathsf{precision\_loss\_indicator}}^{\texttt{p}}\right)$$

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`.

### 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**

EMPTY

$$explode\_intervals(\mathsf{tenv}, \overbrace{[\,]}^{\mathsf{cs}}) \xrightarrow{\mathsf{type}} \left(\overbrace{[\,]}^{\mathtt{new\_cs}}, \overbrace{\mathsf{Precision\_Full}}^{\mathtt{p}}\right)$$

NON_EMPTY

$$\frac{explode\_constraint(\mathsf{tenv}, \mathtt{c}) \xrightarrow{\mathsf{type}} (\mathtt{c'}, \mathtt{p1}) \quad explode\_intervals(\mathsf{tenv}, \mathtt{cs1}) \xrightarrow{\mathsf{type}} (\mathtt{cs1'}, \mathtt{p2}) \quad \mathtt{p} := precision\_join(\mathtt{p1}, \mathtt{p2}) \quad \mathtt{new\_cs} := \mathtt{c'} + \mathtt{cs1'}}{explode\_intervals(\mathsf{tenv}, \overbrace{[\mathtt{c}] + \mathtt{cs1}}^{\mathsf{cs}}) \xrightarrow{\mathsf{type}} (\mathtt{new\_cs}, \mathtt{p})}$$

**TypingRule.ExplodeConstraint**

The function

$$explode\_constraint(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\mathsf{c}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^{*}}^{\mathsf{vcs}}$$

expands the constraint `c` into the equivalent list of exact constraints if `c` matches a n ascending range constraint that is not too large in `tenv` and the singleton list for `c` otherwise.

**Prose**

One of the following applies:

- All of the following apply (EXACT):
  - `c` is an exact constraint;
  - `vcs` is the singleton list for `c`.
- All of the following apply (RANGE_REDUCED):
  - `c` is a range constraint for the expressions `a` and `b`;
  - applying *reduce_to_z_opt* to `a` in `tenv` yields ⟨`za`⟩;
  - applying *reduce_to_z_opt* to `b` in `tenv` yields ⟨`zb`⟩;
  - define `exploded_interval` as the list of exact constraints for each integer literal in the range starting at `za` and ending at `zb`, inclusively, which is empty if $\mathtt{zb} < \mathtt{za}$;
  - applying *interval_too_large* to `za` and `zb` yields `b_too_large`;
  - define `vcs` as the singleton list for `c` if `b_too_large` is `TRUE` and `exploded_interval` otherwise.

- All of the following apply (RANGE_NOT_REDUCED):
  - `c` is a range constraint for the expressions `a` and `b`;
  - applying *reduce_to_z_opt* to `a` in `tenv` yields `za_opt`;
  - applying *reduce_to_z_opt* to `b` in `tenv` yields `zb_opt`;
  - at least one of `za_opt` and `zb_opt` is `None`;
  - `vcs` is the singleton list for `c`.

**Formally**

$$\text{EXACT}\ \frac{\mathit{ast\_label}(\mathsf{c}) = \mathsf{Constraint\_Exact}}{\mathit{explode\_constraint}(\mathsf{tenv}, \mathsf{c}) \xrightarrow{\text{type}} \overbrace{[\mathsf{c}]}^{\mathtt{vcs}}}$$

$$\text{RANGE\_REDUCED}\ \frac{\begin{array}{c}\mathsf{c} = \mathsf{Constraint\_Range}(\mathsf{a}, \mathsf{b}) \\ \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{a}) \xrightarrow{\text{type}} \langle \mathtt{za} \rangle \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{b}) \xrightarrow{\text{type}} \langle \mathtt{zb} \rangle \\ \mathtt{exploded\_interval} := [\mathtt{z} \in \mathtt{za}..\mathtt{zb} : \mathsf{Constraint\_Exact}(\overbrace{\mathbb{Z}}^{\mathsf{E\_Literal(L\_Int)}})] \\ \mathit{interval\_too\_large}(\mathtt{za}, \mathtt{zb}) \xrightarrow{\text{type}} \mathtt{b\_too\_large} \\ \mathtt{vcs} := \mathsf{choice}(\mathtt{b\_too\_large}, [\mathsf{c}], \mathtt{exploded\_interval})\end{array}}{\mathit{explode\_constraint}(\mathsf{tenv}, \mathsf{c}) \xrightarrow{\text{type}} \mathtt{vcs}}$$

$$\text{RANGE\_NOT\_REDUCED}\ \frac{\begin{array}{c}\mathsf{c} = \mathsf{Constraint\_Range}(\mathsf{a}, \mathsf{b}) \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{a}) \xrightarrow{\text{type}} \mathtt{za\_opt} \\ \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{b}) \xrightarrow{\text{type}} \mathtt{zb\_opt} \qquad \mathtt{za\_opt} = \mathtt{None} \vee \mathtt{zb\_opt} = \mathtt{None}\end{array}}{\mathit{explode\_constraint}(\mathsf{tenv}, \mathsf{c}) \xrightarrow{\text{type}} \overbrace{[\mathsf{c}]}^{\mathtt{vcs}}}$$

### TypingRule.IntervalTooLarge

The function

$$\mathit{interval\_too\_large}(\overbrace{\mathbb{Z}}^{\mathtt{z1}}, \overbrace{\mathbb{Z}}^{\mathtt{z2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

determines whether the set of numbers between `z1` and `z2`, inclusive, contains more than $2^{14}$ integers, yielding the result in `b`.

### Prose

The value `b` is `TRUE` if and only if the absolute value of `z1` − `z2` is greater than $2^{14}$.

**Formally**

$$\mathit{interval\_too\_large}(\mathtt{z1}, \mathtt{z2}) \xrightarrow{\text{type}} \overbrace{\mathtt{z2} - \mathtt{z1} > 2^{14}}^{\mathtt{b}}$$

**TypingRule.BinopIsExploding**

The function

$$\mathit{binop\_is\_exploding}(\overbrace{\mathsf{binop}}^{\mathtt{op}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

determines whether the binary operation `op` should lead to applying *explode_intervals* when the `op` is applied to a pair of constraint lists. It is assumed that `op` is one of `MUL`, `SHL`, `POW`, `PLUS`, `DIV`, `MINUS`, `MOD`, `SHR`, and `DIVRM`.

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**

$$\mathit{binop\_is\_exploding}(\mathtt{op}) \xrightarrow{\text{type}} \overbrace{\mathtt{op} \in \{\mathtt{MUL}, \mathtt{SHL}, \mathtt{POW}, \mathtt{DIV}, \mathtt{DIVRM}, \mathtt{MOD}, \mathtt{SHR}\}}^{\mathtt{b}}$$

**TypingRule.BitFieldsIncluded**

The predicate

$$\mathit{bitfields\_included}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{bitfield}^*}^{\mathtt{bfs1}}, \overbrace{\mathsf{bitfield}^*}^{\mathtt{bfs2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

tests whether the set of bit fields `bfs1` is included in the set of bit fields `bfs2` in environment `tenv`, returning a type error, if one is detected.

**Prose**

All of the following apply:

- checking whether each field `bf` in `bfs1` exists in `bfs2` via *mem_bfs* yields $\mathtt{b}_{\mathtt{bf}}{}^{/\!/\,\#\mathtt{TE}}$;
- the result — `b` — is the conjunction of $\mathtt{b}_{\mathtt{bf}}$ for all bitfields `bf` in `bfs1`.

**Formally**

$$\frac{\begin{array}{c}\mathtt{bf} \in \mathtt{bfs1} : \mathit{mem\_bfs}(\mathtt{bfs2}, \mathtt{bf}) \xrightarrow{\text{type}} \mathtt{b}_{\mathtt{bf}} \;/\!/\; \#\mathtt{TE} \\ \mathtt{bf} := \bigwedge_{\in \mathtt{bfs1}} \mathtt{b}_{\mathtt{bf}}\end{array}}{\mathit{bitfields\_included}(\mathsf{tenv}, \mathtt{bfs1}, \mathtt{bfs2}) \xrightarrow{\text{type}} \mathtt{b}}$$

### TypingRule.MemBfs

The function

$$mem\_bfs(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{bitfield}^{+}}^{\texttt{bfs2}}, \overbrace{\mathsf{bitfield}}^{\texttt{bf1}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

checks whether the bitfield `bf` exists in `bfs2` in the context of `tenv`, returning the result in `b`.

### Prose

One of the following applies:

- All of the following apply (NONE):
  - the name associated with the bitfield `bf1` is `name`;
  - finding the bitfield associated with `name` in `bfs2` yields `None`;
  - `b` is `FALSE`.
- All of the following apply (SIMPLE_ANY):
  - the name associated with the bitfield `bf1` is `name`;
  - finding the bitfield associated with `name` in `bfs2` yields `bf2`;
  - `bf2` is a simple bitfield;
  - symbolically checking whether `bf1` is equivalent to `bf2` in `tenv` yields `b`.
- All of the following apply (NESTED_SIMPLE):
  - the name associated with the bitfield `bf1` is `name`;
  - finding the bitfield associated with `name` in `bfs2` yields `bf2`;
  - `bf2` is a nested bitfield with name `name2`, slices `slices2`, and bitfields `bfs2'`;
  - `bf1` is a simple bitfield;
  - symbolically checking whether `bf1` is equivalent to `bf2` in `tenv` yields `b`.
- All of the following apply (NESTED_NESTED):
  - the name associated with the bitfield `bf1` is `name`;
  - finding the bitfield associated with `name` in `bfs2` yields `bf2`;
  - `bf2` is a nested bitfield with name `name2`, slices `slices2`, and bitfields `bfs2'`;
  - `bf1` is a nested bitfield with name `name1`, slices `slice1`, and `bfs1`;
  - `b1` is true if and only if `name1` is equal to `name2`;
  - symbolically equating the slices `slices1` and `slices2` in `tenv` yields `b2`;
  - checking `bfs1` is included in `bfs2'` in the context of `tenv` yields `b3`;
  - `b` is defined as the conjunction of `b1`, `b2`, and `b3`.

- All of the following apply (NESTED_TYPED):
  - the name associated with the bitfield `bf1` is `name`;
  - finding the bitfield associated with `name` in `bfs2` yields `bf2`;
  - `bf2` is a nested bitfield with name `name2`, slices `slices2`, and bitfields `bfs2'`;
  - `bf1` is a typed bitfield;
  - `b` is `FALSE`.
- All of the following apply (TYPED_SIMPLE):
  - the name associated with the bitfield `bf1` is `name`;
  - finding the bitfield associated with `name` in `bfs2` yields `bf2`;
  - `bf2` is a typed bitfield with name `name2`, slices `slices2`, and type `ty2`;
  - `bf1` is a simple bitfield;
  - symbolically checking whether `bf1` is equivalent to `bf2` in `tenv` yields `b`.
- All of the following apply (TYPED_NESTED):
  - the name associated with the bitfield `bf1` is `name`;
  - finding the bitfield associated with `name` in `bfs2` yields `bf2`;
  - `bf2` is a typed bitfield with name `name2`, slices `slices2`, and type `ty2`;
  - `bf1` is a nested bitfield;
  - `b` is `FALSE`.
- All of the following apply (TYPED_TYPED):
  - the name associated with the bitfield `bf1` is `name`;
  - finding the bitfield associated with `name` in `bfs2` yields `bf2`;
  - `bf2` is a typed bitfield with name `name2`, slices `slices2`, and type `ty2`;
  - `bf1` is a typed bitfield with name `name1`, slices `slices1`, and type `ty1`;
  - `b1` is true if and only if `name1` is equal to `name2`;
  - symbolically equating the slices `slices1` and `slices2` in `tenv` yields `b2`;
  - checking whether `ty1` subtypes `ty2` in `tenv` yields `b3`;
  - `b` is defined as the conjunction of `b1`, `b2`, and `b3`.

**Formally**

NONE

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf1}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \texttt{None}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

SIMPLE_ANY

$$\frac{\begin{array}{c}\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \\ \mathit{ast\_label}(\texttt{bf2}) = \texttt{BitField\_Simple} \qquad \mathit{bitfields\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}\end{array}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{b}}$$

NESTED_SIMPLE

$$\frac{\begin{array}{c}\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \\ \texttt{bf2} = \texttt{BitField\_Nested}(\texttt{name2}, \texttt{slices2}, \texttt{bfs2'}) \\ \texttt{bf1} = \texttt{BitField\_Simple}(\_) \qquad \mathit{bitfields\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}\end{array}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \overbrace{\texttt{FALSE}}^{\texttt{b}}}$$

NESTED_NESTED

$$\frac{\begin{array}{c}\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \\ \texttt{bf2} = \texttt{BitField\_Nested}(\texttt{name2}, \texttt{slices2}, \texttt{bfs2'}) \\ \texttt{bf1} = \texttt{BitField\_Nested}(\texttt{name1}, \texttt{slices1}, \texttt{bfs1}) \\ \texttt{b1} := \texttt{name1} = \texttt{name2} \qquad \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b2} \\ \mathit{bitfields\_included}(\texttt{tenv}, \texttt{bfs1}, \texttt{bfs2'}) \xrightarrow{\text{type}} \texttt{b3} \qquad \texttt{b} := \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3}\end{array}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{b}}$$

NESTED_TYPED

$$\frac{\begin{array}{c}\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \\ \texttt{bf2} = \texttt{BitField\_Nested}(\texttt{name2}, \texttt{slices2}, \texttt{bfs2'}) \qquad \mathit{ast\_label}(\texttt{bf1}) = \texttt{BitField\_Type}\end{array}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \overbrace{\texttt{FALSE}}^{\texttt{b}}}$$

TYPED_SIMPLE

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \quad \texttt{bf2} = \text{BitField\_Type}(\texttt{name2}, \texttt{slices2}, \texttt{ty2}) \quad \texttt{bf1} = \text{BitField\_Simple}(\_) \qquad \mathit{bitfields\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{b}}$$

TYPED_NESTED

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \quad \texttt{bf2} = \text{BitField\_Type}(\texttt{name2}, \texttt{slices2}, \texttt{ty2}) \qquad \mathit{ast\_label}(\texttt{bf1}) = \text{BitField\_Nested}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \overbrace{\texttt{FALSE}}^{\texttt{b}}}$$

TYPED_TYPED

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \quad \texttt{bf2} = \text{BitField\_Type}(\texttt{name2}, \texttt{slices2}, \texttt{ty2}) \quad \texttt{bf1} = \text{BitField\_Type}(\texttt{name1}, \texttt{slices1}, \texttt{ty1}) \quad \texttt{b1} := \texttt{name1} = \texttt{name2} \qquad \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b2} \quad \mathit{subtype\_satisfies}(\texttt{tenv}, \texttt{ty1}, \texttt{ty2}) \xrightarrow{\text{type}} \texttt{b3} \ /\!/ \ \texttt{\#TE} \quad \texttt{b} := \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.CheckStructure

The function

$$\mathit{check\_structure}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{ty}}^{\texttt{t}}, \overbrace{\mathbb{L}}^{\texttt{l}}) \longrightarrow \{\texttt{TRUE}\} \cup \text{TTypeError}$$

returns TRUE is t is 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 t'$/\!/^{\texttt{\#TE}}$;
  - t' has the label l;
  - the result is TRUE;
- All of the following apply (ERROR):

* determining the structure of `t` yields `t'` $/\!/_{\texttt{\#TE}}$;
* `t'` does not have the label `l`;
* the result is a type error indicating that `t` was expected to have the structure of a type with the AST label `l`.

**Formally**

$$\text{OKAY}\quad \frac{\mathit{get\_structure}(\texttt{t}) \xrightarrow{\text{type}} \texttt{t'} \;/\!/\; \texttt{\#TE} \qquad \mathit{ast\_label}(\texttt{t'}) = \texttt{l}}{\mathit{check\_structure}(\texttt{tenv}, \texttt{t}, \texttt{l}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

$$\text{ERROR}\quad \frac{\mathit{get\_structure}(\texttt{t}) \xrightarrow{\text{type}} \texttt{t'} \qquad \mathit{ast\_label}(\texttt{t'}) \neq \texttt{l}}{\mathit{check\_structure}(\texttt{tenv}, \texttt{t}, \texttt{l}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_UT})}$$

**TypingRule.ToWellConstrained**

The function

$$\mathit{to\_well\_constrained}(\overbrace{\text{ty}}^{\texttt{t}}) \longrightarrow \overbrace{\text{ty}}^{\texttt{t'}}$$

returns the well-constrained version of a type `t` — `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-constrainted Integer Types**

The following table shows examples of applying *to_well_constrained* to various types:

| input type | output type |
|---|---|
| T_Int(Parameterized(x)) | T_Int(WellConstrained(Constraint_Exact(E_Var(x)))) |
| T_Int(unconstrained_integer) | T_Int(unconstrained_integer) |
| T_Real | T_Real |

**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**

T_INT_PARAMETERIZED

$$\mathit{to\_well\_constrained}(\mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{v}))) \xrightarrow{\mathsf{type}} \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathsf{Constraint\_Exact}(\mathsf{E\_Var}(\mathtt{v}))))$$

T_INT_OTHER

$$\frac{\mathit{ast\_label}(\mathtt{i}) \neq \mathsf{Parameterized}}{\mathit{to\_well\_constrained}(\mathsf{T\_Int}(\mathtt{i})) \xrightarrow{\mathsf{type}} \mathtt{t}}$$

OTHER

$$\frac{\mathit{ast\_label}(\mathtt{t}) \neq \mathsf{T\_Int}}{\mathit{to\_well\_constrained}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t}}$$

### TypingRule.GetWellConstrainedStructure

The function

$$\mathit{get\_well\_constrained\_structure}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{t'}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

returns the well-constrained structure of a type `t` in the static environment `tenv` — `t'`, which is defined as follows. Otherwise, the result is a type error.

**Prose**

All of the following apply:

- the structure of `t` in `tenv` is `t1`$^{//\#\mathsf{TE}}$;
- the well-constrained version of `t1` is `t'`.

**Formally**

$$\frac{\begin{array}{c}\mathit{get\_structure}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t1} \;//\; \#\mathsf{TE} \\ \mathit{to\_well\_constrained}(\mathtt{t1}) \xrightarrow{\mathsf{type}} \mathtt{t'}\end{array}}{\mathit{get\_well\_constrained\_structure}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t'}}$$

### TypingRule.GetBitvectorWidth

The function

$$\mathit{get\_bitvector\_width}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathsf{expr}}^{\mathtt{e}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

returns the expression `e`, which represents the width of the bitvector type `t` in the static environment `tenv`. $^{//\#\mathsf{TE}}$

### Prose

One of the following applies:

- All of the following apply (OKAY):
  - obtaining the structure of `t` in `tenv` yields a bitvector type with width expression `e`, that is, $\mathsf{T\_Bits}(\mathtt{e}, \_)^{/\!/ \#TE}$;
  - the result is `e`.
- All of the following apply (ERROR):
  - obtaining the structure of `t` in `tenv` yields a type that is not a bitvector type;
  - the result is a type error indicating that a bitvector type was expected.

### Formally

$$\text{OKAY}\quad \frac{\mathit{get\_structure}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\mathtt{e}, \_) \;/\!/\; \#TE}{\mathit{get\_bitvector\_width}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{e}}$$

$$\text{ERROR}\quad \frac{\mathit{get\_structure}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t'} \qquad \mathit{ast\_label}(\mathtt{t'}) \neq \mathsf{T\_Bits}}{\mathit{get\_bitvector\_width}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_UT})}$$

### TypingRule.GetBitvectorConstWidth

The function

$$\mathit{get\_bitvector\_const\_width}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathbb{N}}^{\mathtt{w}} \cup \overbrace{\mathsf{TTypeError}}^{\#TE}$$

returns the natural number `w`, which represents the width of the bitvector type `t` in the static environment `tenv`. Otherwise, the result is a type error.

### Prose

All of the following apply:

- applying *get_bitvector_width* to `t` in `tenv` yields $\mathtt{e\_width}^{/\!/ \#TE}$;
- statically evaluating the expression `e_width` in the static environment `tenv` yields the literal integer for $\mathtt{w}^{/\!/ \#TE}$.

### Formally

$$\frac{\mathit{get\_bitvector\_width}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{e\_width} \;/\!/\; \#TE \qquad \mathit{static\_eval}(\mathtt{tenv}, \mathtt{e\_width}) \xrightarrow{\text{type}} \mathsf{L\_Int}(\mathtt{w}) \;/\!/\; \#TE}{\mathit{get\_bitvector\_const\_width}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{w}}$$

### TypingRule.CheckBitsEqualWidth

The function

$$\mathit{check\_bits\_equal\_width}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{t1}}, \overbrace{\mathsf{ty}}^{\texttt{t2}}) \longrightarrow \{\texttt{TRUE}\} \cup \mathsf{TTypeError}$$

tests whether the types `t1` and `t2` are bitvector types of the same width. If the answer is positive, the result is `TRUE`. Otherwise, the result is a type error.

### Prose

All of the following apply:

- obtaining the width of `t1` in `tenv` (via *get_bitvector_width*) yields the expression $\texttt{n}^{/\!/\texttt{\#TE}}$;
- obtaining the width of `t2` in `tenv` (via *get_bitvector_width*) yields the expression $\texttt{m}^{/\!/\texttt{\#TE}}$;
- One of the following applies:
  - All of the following apply (TRUE):
    - symbolically checking whether the bitwidth expressions `n` and `m` are equal (via *bitwidth_equal*) yields `TRUE`;
    - the result is `TRUE`.
  - All of the following apply (ERROR):
    - symbolically checking whether the bitwidth expressions `n` and `m` are equal (via *bitwidth_equal*) yields `FALSE`;
    - the result is a type error indicating that the bitwidths are different.

### Formally

TRUE

$$\frac{\begin{array}{c}\mathit{get\_bitvector\_width}(\texttt{tenv},\texttt{t1}) \xrightarrow{\text{type}} \texttt{n}^{/\!/\texttt{\#TE}} \\ \mathit{get\_bitvector\_width}(\texttt{tenv},\texttt{t2}) \xrightarrow{\text{type}} \texttt{m}^{/\!/\texttt{\#TE}} \\ \mathit{bitwidth\_equal}(\texttt{tenv},\texttt{n},\texttt{m}) \xrightarrow{\text{type}} \texttt{TRUE}\end{array}}{\mathit{check\_bits\_equal\_width}(\texttt{tenv},\texttt{t1},\texttt{t2}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

ERROR

$$\frac{\begin{array}{c}\mathit{get\_bitvector\_width}(\texttt{tenv},\texttt{t1}) \xrightarrow{\text{type}} \texttt{n}^{/\!/\texttt{\#TE}} \\ \mathit{get\_bitvector\_width}(\texttt{tenv},\texttt{t2}) \xrightarrow{\text{type}} \texttt{m}^{/\!/\texttt{\#TE}} \\ \mathit{bitwidth\_equal}(\texttt{tenv},\texttt{n},\texttt{m}) \xrightarrow{\text{type}} \texttt{FALSE}\end{array}}{\mathit{check\_bits\_equal\_width}(\texttt{tenv},\texttt{t1},\texttt{t2}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{TE\_UT})}$$

### TypingRule.PrecisionJoin

The function

$$\mathit{precision\_join}(\overbrace{\texttt{p1}}^{\text{precision\_loss\_indicator}}, \overbrace{\texttt{p1}}^{\text{precision\_loss\_indicator}}) \longrightarrow \overbrace{\texttt{p}}^{\text{precision\_loss\_indicator}}$$

returns the precision loss indicator `p`, denoting whether `p1` or `p2` denote a precision loss.

### Example: Precision join

In Listing 13.51, 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 13.51: 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

$$\textsc{Loss}\ \frac{\texttt{p1} = \texttt{Precision\_Lost} \vee \texttt{p2} = \texttt{Precision\_Lost}}{\texttt{p} = \texttt{Precision\_Lost}}$$

$$\textsc{Full}\ \frac{\texttt{p1} = \texttt{Precision\_Full} \qquad \texttt{p2} = \texttt{Precision\_Full}}{\texttt{p} = \texttt{Precision\_Full}}$$

## 13.19 Base Values

Each type, with the exceptions stated below, have a base value, which is used to initialize storage elements (either local of 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 13.52 shows a specification with examples of well-typed base value for various types, followed by the output to the console.

Listing 13.52: 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 13.53 shows a specification that relies on the base value of a bitvector type whose width is a parameterized integer type.

Listing 13.53: 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 13.54 shows an ill-typed specification where the width of a bitvector is negative.

Listing 13.54: No Base Value for Bitvectors of Negative Width

```
var bits_base: bits(-3);
```

Listing 13.55 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 13.55: No Base Value for an Empty Integer Type

```
var x : integer{5..0};
```

The function

$$\mathit{base\_value}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{t}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{e\_init}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

returns the expression `e_init` which can be used to initialize a storage element of type `t` in the static environment `tenv`. Otherwise, the result is a type error.

### TypingRule.BaseValue

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 $\texttt{TRUE}^{/\!/\texttt{TE\_NBV}}$;
  - `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 `l` 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` ∈ *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):
    * `t` is a structured type with list of fields `fields`;
    * applying *base_value* to `t_field` in `tenv` for each (`name`, `t_field`) in `fields` yields $\texttt{e}_{\texttt{name}}{}^{\#TE}$;
    * `e_init` is the record construction expression assigning each field `name` where (`name`, `t_field`) is an element of `fields` to `t_field`, that is, E_Record((`name`, `t_field`) ∈ `fields` : (`name`, $\texttt{e}_{\texttt{name}}$)).
- All of the following apply (T_STRING):
    * `t` is the string type;
    * `e_init` is the string literal expression for the empty list of characters.

- All of the following apply (T_TUPLE):
  - `t` is the tuple type over the list of types $\texttt{t}_{1..k}$, that is, T_Tuple($\texttt{t}_{1..k}$);
  - applying *base_value* to each type $\texttt{t}_\texttt{i}$ in `tenv` for $\texttt{i} = 1..k$; yields the list of expressions $\texttt{e}_{1..k}$;
  - `e_init` is the tuple expression E_Tuple($\texttt{e}_{1..k}$).
- All of the following apply (T_ARRAY_ENUM):
  - `t` is the enumerated array type over for the enumeration `enum` and labels `labels` and element type `ty`, that is, T_Array(ArrayLength_Enum(`enum`, `labels`), `ty`) ;
  - applying *base_value* to `ty` in `tenv` yields the expression `value`$^{\#\texttt{TE}}$;
  - `e_init` is the array construction expression for an enumerated array with labels `labels` and initial value `value`, that is, E_EnumArray{labels : `labels`, value : `value`}.
- All of the following apply (T_ARRAY_EXPR):
  - `t` is the array type over an integer index expression `v_length` and element type `ty`, that is, T_Array(ArrayLength_Expr(`v_length`), `ty`) ;
  - applying *base_value* to `ty` in `tenv` yields the expression `value`$^{\#\texttt{TE}}$;
  - `e_init` is the array construction expression with length expression `v_length` and value expression `value`, that is, E_Array{length : `length`, value : `value`}.

**Formally**

T_BOOL

$$\textit{base\_value}(\text{tenv}, \overbrace{\text{T\_Bool}}^{\texttt{t}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Literal}(\text{L\_Bool}(\texttt{FALSE}))}^{\texttt{e\_init}}$$

T_BITS_STATIC

$$\frac{\textit{reduce\_to\_z\_opt}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \texttt{z\_opt} \qquad \texttt{z\_opt} \neq \text{None} \qquad \texttt{z\_opt} \overset{\text{is}}{=} \langle \texttt{length} \rangle \qquad \textit{check}(\texttt{length} \geq 0, \texttt{TE\_NBV}) \longrightarrow \texttt{TRUE} \;//\; \#\texttt{TE}}{\textit{base\_value}(\text{tenv}, \overbrace{\text{T\_Bits}(\text{e}, \_)}^{\texttt{t}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Literal}(\text{L\_Bitvector}(i = 1..\texttt{length} : 0))}^{\texttt{e\_init}}}$$

T_BITS_NON_STATIC

$$\frac{\textit{reduce\_to\_z\_opt}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \texttt{z\_opt} \qquad \texttt{z\_opt} = \text{None} \qquad \texttt{e\_init} := \text{E\_Slice}(\underset{\text{E\_Literal(L\_Int)}}{0}, [\text{Slice\_Length}(\underset{\text{E\_Literal(L\_Int)}}{0}, \text{e})])}{\textit{base\_value}(\text{tenv}, \overbrace{\text{T\_Bits}(\text{e}, \_)}^{\texttt{t}}) \xrightarrow{\text{type}} \texttt{e\_init}}$$

T_ENUM

$$\frac{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{name}) \xrightarrow{\mathsf{type}} \mathtt{l}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Enum}(\mathtt{name} + \_)}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathtt{l})}^{\mathtt{e\_init}}}$$

T_INT_UNCONSTRAINED

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{unconstrained\_integer}}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(0))}^{\mathtt{e\_init}}$$

T_INT_PARAMETERIZED

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{id}))}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_NBV})$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{c}\mathtt{cs} \overset{\mathsf{is}}{=} \mathtt{c}_{1..k} \\ \mathtt{z\_min\_list} := \mathit{constraint\_abs\_min}(\mathsf{tenv}, \mathtt{c}_1) + \ldots + \mathit{constraint\_abs\_min}(\mathsf{tenv}, \mathtt{c}_k) \\ \mathit{check}(\mathtt{z\_min\_list} \neq \emptyset, \mathtt{TE\_NBV}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;\; // \;\; \mathtt{\#TE} \\ \mathit{list\_min\_abs}(\mathtt{z\_min\_list}) \xrightarrow{\mathsf{type}} \mathtt{z\_min}\end{array}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs}))}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(\mathtt{z\_min}))}^{\mathtt{e\_init}}}$$

T_NAMED

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id})) \xrightarrow{\mathsf{type}} \mathtt{t'} \;\; // \;\; \mathtt{\#TE} \\ \mathit{base\_value}(\mathsf{tenv}, \mathtt{t'}) \xrightarrow{\mathsf{type}} \mathtt{e\_init} \;\; // \;\; \mathtt{\#TE}\end{array}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Named}(\mathtt{id})}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \mathtt{e\_init}}$$

T_REAL

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Real}}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Real}(0))}^{\mathtt{e\_init}}$$

STRUCTURED

$$\frac{\begin{array}{c}\mathit{is\_structured}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \qquad \mathtt{t} \overset{\mathsf{is}}{=} L(\mathtt{fields}) \\ (\mathtt{name}, \mathtt{t\_field}) \in \mathtt{fields} : \mathit{base\_value}(\mathsf{tenv}, \mathtt{t\_field}) \xrightarrow{\mathsf{type}} \mathtt{e}_{\mathtt{name}} \;\; // \;\; \mathtt{\#TE}\end{array}}{\mathit{base\_value}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Record}((\mathtt{name}, \mathtt{t\_field}) \in \mathtt{fields} : (\mathtt{name}, \mathtt{e}_{\mathtt{name}}))}^{\mathtt{e\_init}}}$$

T_STRING

$$base\_value(\text{tenv}, \overbrace{\text{T\_String}}^{\text{t}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Literal}(\text{L\_String}([\,]))}^{\text{e\_init}}$$

T_TUPLE

$$\frac{\texttt{i} = 1..k : base\_value(\text{tenv}, \texttt{t}_\texttt{i}) \xrightarrow{\text{type}} \texttt{e}_\texttt{i} \;/\!/\; \text{\#TE}}{base\_value(\text{tenv}, \overbrace{\text{T\_Tuple}}^{\texttt{t}_{1..k}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Tuple}(\texttt{e}_{1..k})}^{\text{e\_init}}}$$

T_ARRAY_ENUM

$$\frac{base\_value(\text{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{value} \;/\!/\; \text{\#TE}}{base\_value(\text{tenv}, \overbrace{\text{T\_Array}(\text{ArrayLength\_Enum}(\texttt{enum}, \texttt{labels}), \texttt{ty})}^{\text{t}}) \xrightarrow{\text{type}} \overbrace{\text{E\_EnumArray}\{\text{labels} : \texttt{labels}, \text{value} : \texttt{value}\}}^{\text{e\_init}}}$$

T_ARRAY_EXPR

$$\frac{base\_value(\text{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{value} \;/\!/\; \text{\#TE}}{base\_value(\text{tenv}, \overbrace{\text{T\_Array}(\text{ArrayLength\_Expr}(\texttt{length}), \texttt{ty})}^{\text{t}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Array}\{\text{length} : \texttt{length}, \text{value} : \texttt{value}\}}^{\text{e\_init}}}$$

### TypingRule.ConstraintAbsMin

The function

$$constraint\_abs\_min(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^{\text{c}}) \longrightarrow \overbrace{\mathbb{Z}^*}^{\text{zs}} \cup \overbrace{\text{TTypeError}}^{\text{TypeError(TE\_NBV)}}$$

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 TypeError(`TE_NBV`).

### Example: Minimal Absolute Value in a Constraint List

The minimal absolute value of `{7, -2}` is `-2`.
The minimal absolute value of `{2, -2}` is `2`.
The minimal absolute value of `{-2..2, 5}` is `0`.

**Prose**

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`$^{/\!/\texttt{TE\_NBV}}$;
  - 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`$^{/\!/\texttt{TE\_NBV}}$;
  - 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`$^{/\!/\texttt{TE\_NBV}}$;
  - 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**

$$\text{EXACT}\quad \frac{\begin{array}{c} \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \texttt{z\_opt} \\ \mathit{check}(\texttt{z\_opt} \neq \texttt{None}, \texttt{TE\_NBV}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{z\_opt} \overset{\text{is}}{=} \langle \mathsf{z} \rangle \end{array}}{\mathit{constraint\_abs\_min}(\overbrace{\text{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{[\mathsf{z}]}^{\mathsf{zs}}}$$

RANGE

$$\frac{\begin{array}{c} \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{z\_opt1} \\ \mathit{check}(\mathtt{z\_opt1} \neq \mathtt{None}, \mathtt{TE\_NBV}) \longrightarrow \mathtt{TRUE} \ /\!/ \ \mathtt{\#TE} \\ \mathtt{z\_opt1} \overset{\text{is}}{=} \langle \mathtt{v1} \rangle \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\text{type}} \mathtt{z\_opt2} \\ \mathit{check}(\mathtt{z\_opt2} \neq \mathtt{None}, \mathtt{TE\_NBV}) \longrightarrow \mathtt{TRUE} \ /\!/ \ \mathtt{\#TE} \\ \mathtt{z\_opt2} \overset{\text{is}}{=} \langle \mathtt{v2} \rangle \qquad \mathtt{zs} := \begin{cases} [\,] & \mathtt{v1} > \mathtt{v2} \\ [\mathtt{v2}] & \mathtt{v1} \leq \mathtt{v2} < 0 \\ [0] & \mathtt{v1} < 0 \leq \mathtt{v2} \\ [\mathtt{v1}] & 0 \leq \mathtt{v1} \leq \mathtt{v2} \end{cases} \end{array}}{\mathit{constraint\_abs\_min}(\overbrace{\mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{c}}) \xrightarrow{\text{type}} \mathtt{zs}}$$

### TypingRule.ListMinAbs

The function

$$\mathit{list\_min\_abs}(\overbrace{\mathbb{Z}^*}^{\mathtt{l}}) \longrightarrow \overbrace{\mathbb{Z}}^{\mathtt{z}}$$

returns `z` — the integer closest to 0 among the list of integers in the list `l`. The result is biased towards positive integers. That is, if two integers $x$ and $y$ have the same absolute value and $x$ is positive and $y$ is negative then $x$ is considered closer to 0.

### 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):
  - `l` is the single element list for `z`.
- All of the following apply (MORE_THAN_ONE):
  - `l` is the list where `z1` is its `head` and `l2` is its `tail`;
  - `l2` is not the empty list;
  - applying *list_min_abs* to `l2` yields `z2`;
  - define `z` based on `z1` and `z2` by the following cases:
    - `z1` if the absolute value of `z1` is less than the absolute value of `z2`;
    - `z2` if the absolute value of `z1` is greater than the absolute value of `z2`;
    - `z1` if `z1` is equal to `z2`;
    - the absolute value of `z1` if the absolute value of `z1` is equal to the absolute value of `z2` and `z1` is not equal to `z2`;

**Formally**

ONE

$$\mathit{list\_min\_abs}(\overbrace{[\mathtt{z}]}^{1}) \xrightarrow{\mathsf{type}} \mathtt{z}$$

MORE_THAN_ONE

$$\frac{\mathtt{l1} \neq [\,] \qquad \mathit{list\_min\_abs}(\mathtt{l2}) = \mathtt{z2} \qquad \mathtt{z} := \begin{cases} \mathtt{z1} & |\mathtt{z1}| < |\mathtt{z2}| \\ \mathtt{z2} & |\mathtt{z1}| > |\mathtt{z2}| \\ \mathtt{z1} & \mathtt{z1} = \mathtt{z2} \\ |\mathtt{z1}| & |\mathtt{z1}| = |\mathtt{z2}| \wedge \mathtt{z1} \neq \mathtt{z2} \end{cases}}{\mathit{list\_min\_abs}(\overbrace{[\mathtt{z1}] + \mathtt{l2}}^{1}) \xrightarrow{\mathsf{type}} \mathtt{z}}$$

# Chapter 14

# Bitfields

Bitvector types allow defining bitslices of bitvectors, to be treated as named fields, which can be read or written.

Individual bitfields are grammatically derived from `bitfield` and represented as ASTs by bitfield. Bitfields are not associated with a semantic relation.

**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 respected to its containing bitvector type. The absolute bitfield of a given bitfield consists of bits absolute name and its absolute slices.

Further, the bitfield scope of a bitfield is its absolute name, with the last identifier (the bitfield's name) removed. For example, the bitfield scope of a bitfield with the absolute name `fmt0.later1.remainder.moving` is `fmt0.later1.remainder` and the bitfield scope of a bitfield with the absolute name `fmt0` is the empty list.

We say that two bitfields exist in the same bitfield scope if the bitfield scope of one is a prefix of the bitfield scope of the other. For example, consider two bitfields whose corresponding absolute names are `fmt0.later1.remainder` and `fmt0.later1.remainder.moving`, respectively. They exist in the same scope, since `fmt0.later1` is a prefix of `fmt0.later1.remainder`. In contrast, a pair of bitfields with absolute names `fmt0.common` and `fmt1.common`, since neither of `fmt0` and `fmt1` is a prefix of the other.

**Guide.BitfieldAlignment** For every two bitfields of a given bitvector type, if they share the same name and exist in the same bitfield scope then their absolute slices must be equal. This is formalized in TypingRule.CheckCommonBitfieldsAlign and exemplified in Listing 14.2.

### Example: A bitvector type with nested bitfields

Listing 14.2 shows a bitvector type with nested bitfields, along with the absolute bitfield defined for each bitfield.

Listing 14.2: A bitvector type with nested bitfields

```
type Nested_Type of bits(32) {        // absolute fields
   [31:16] fmt0 {                     // [31:16] fmt0
        [15] common,                  // [31:31] fmt0.common
        [14:13, 12:2, 1, 0] layer1 {  // [30:16] fmt0.layer1
            [14:13] remainder {       // [30:29] fmt0.layer1.remainder
               [1]  moving,           // [30:30] fmt0.layer1.remainder.moving
               [0]  extra             // [29:29] fmt0.layer1.remainder.extra
            },
        },
        [13] extra                    // [29:29] fmt0.extra
   },
   [31:16] fmt1 {                     // [31:16] fmt1
       [15] common,                   // [31:31] fmt1.common
       [0]  moving                    // [16:16] fmt1.moving
```

```
    },
    [31] common,                        // [31:31] common
    [0]  fmt                            // [0:0] fmt
};

var nested : Nested_Type = '10101010101010101010101010101010';

// select the correct view of moving
// nested.fmt is '0'
//    nested.fmt0.moving is nested[30]
// nested.fmt is '1'
//    nested.fmt1.moving is nested[16]
let moving = if nested.fmt == '0' then nested.fmt0.layer1.remainder.moving
    else nested.fmt1.moving;

func main() => integer
begin
    // below are all equivalent to nested[31]
    let common = nested.common;
    let common_fmt0 = nested.fmt0.common;
    let common_fmt1 = nested.fmt1.common;
    assert common == common_fmt0;
    assert common == common_fmt1;
    return 0;
end;
```

### 14.1.1 Syntax

$$\begin{aligned}\texttt{bitfields} &\longrightarrow \texttt{"\{"}\ \texttt{tclist0}(\texttt{bitfield})\ \texttt{"\}"}\\ \texttt{bitfield} &\longrightarrow \texttt{slices}\ \mathbf{ID}\\ &\mid \texttt{slices}\ \mathbf{ID}\ \texttt{bitfields}\\ &\mid \texttt{slices}\ \mathbf{ID}\ \texttt{":"}\ \texttt{ty}\end{aligned}$$

### 14.1.2 Abstract Syntax

$$\begin{aligned}\texttt{bitfield} \longrightarrow\ & \texttt{BitField\_Simple}(\texttt{identifier}, \texttt{slice}^*)\\ \mid\ & \texttt{BitField\_Nested}(\texttt{identifier}, \texttt{slice}^*, \texttt{bitfield}^*)\\ \mid\ & \texttt{BitField\_Type}(\texttt{identifier}, \texttt{slice}^*, \texttt{ty})\end{aligned}$$

#### ASTRule.Bitfields

The function

$$\mathit{build\_bitfields}(\overbrace{\mathbb{PARSE}[\texttt{bitfields}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{bitfield}^*}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_tclist}[\mathit{build\_bitfield}](\texttt{bitfields}) \xrightarrow{\text{ast}} \texttt{bitfield\_asts}}{\mathit{build\_bitfields}(\texttt{bitfields}(\texttt{"\{"}, \texttt{bitfields} : \texttt{tclist0}(\texttt{bitfield}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{bitfield\_asts}}^{\texttt{ast\_node}}}$$

#### ASTRule.Bitfield

The function

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

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

SIMPLE

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{slices}, \mathbf{ID}(\texttt{x}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Simple}(\texttt{x}, \overline{\texttt{slices}})}^{\texttt{ast\_node}}$$

NESTED

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{slices}, \mathbf{ID}(\texttt{x}), \texttt{bitfields})) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Nested}(\texttt{x}, \overline{\texttt{slices}}, \overline{\texttt{bitfields}})}^{\texttt{ast\_node}}$$

TYPE

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{slices}, \mathbf{ID}(\texttt{x}), \texttt{":"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Type}(\texttt{x}, \overline{\texttt{slices}}, \overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

## 14.2 Typing Bitfields

#### TypingRule.TBitFields

The function

$$\mathit{annotate\_bitfields}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{expr}}^{\texttt{e\_width}}, \overbrace{\mathsf{bitfield}^*}^{\texttt{fields}}) \longrightarrow (\overbrace{\mathsf{bitfield}^*}^{\texttt{new\_fields}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a list of bitfields — `fields` — with an expression denoting the overall number of bits in the containing bitvector type — `e_width`, in an environment `tenv`, resulting in `new_fields` — the typed AST for `fields` and `e_width` as well as a set of side effect descriptors `ses`. Otherwise, the result is a type error.

#### 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

- symbolically simplifying `e_width` in `tenv` via *static_ eval* yields the literal integer for `width`$^{\text{// \#TE}}$;
- annotating each bitfield `f` in `fields` with width `width` in `tenv` yields the corresponding annotated bitfield `f'` and set of side effect descriptors $\texttt{xs}_\texttt{f}{}^{\text{// \#TE}}$;
- define `new_fields` as the list of annotated bitfields;
- define `ses` as the union of $\texttt{xs}_\texttt{f}$ for every field `f` in `fields`.

**Formally**

$$\frac{\begin{array}{c}\texttt{names} := [\texttt{field} \in \texttt{fields} : \mathit{bitfield\_get\_name}(\texttt{field})] \\ \mathit{check\_no\_duplicates}(\texttt{names}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{static\_eval}(\texttt{tenv}, \texttt{e\_width}) \xrightarrow{\text{type}} \texttt{L\_Int}(\texttt{width}) \;/\!/\; \texttt{\#TE} \\ \texttt{f} \in \texttt{fields} : \mathit{annotate\_bitfield}(\texttt{tenv}, \texttt{width}, \texttt{field}) \xrightarrow{\text{type}} (\texttt{f'}, \texttt{xs}_\texttt{f}) \;/\!/\; \texttt{\#TE} \\ \texttt{new\_fields} := [\texttt{f} \in \texttt{fields} : \texttt{f'}] \qquad \texttt{ses} := \bigcup_{\texttt{f} \in \texttt{fields}} \texttt{xs}_\texttt{f}\end{array}}{\mathit{annotate\_bitfields}(\texttt{tenv}, \texttt{e\_width}, \texttt{fields}) \xrightarrow{\text{type}} (\texttt{new\_fields}, \texttt{ses})}$$

**TypingRule.BitFieldGetName**

The function

$$\mathit{bitfield\_get\_name} : \overbrace{\mathsf{bitfield}}^{\texttt{bf}} \longrightarrow \overbrace{\mathsf{identifier}}^{\texttt{name}}$$

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

$$\mathit{bitfield\_get\_name}(\overbrace{\mathsf{BitField\_Simple}(\texttt{name}, \_)}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{name}$$

NESTED

$$\mathit{bitfield\_get\_name}(\overbrace{\mathsf{BitField\_Nested}(\texttt{name}, \_, \_)}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{name}$$

TYPE

$$\mathit{bitfield\_get\_name}(\overbrace{\mathsf{BitField\_Type}(\texttt{name}, \_, \_)}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{name}$$

**TypingRule.BitFieldGetSlices**

The function

$$\mathit{bitfield\_get\_slices} : \overbrace{\mathsf{bitfield}}^{\texttt{bf}} \longrightarrow \overbrace{\mathsf{slice}^*}^{\texttt{slices}}$$

returns the list of slices `slices` associated with the bitfield `bf`.

**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**

SIMPLE

$$\mathit{bitfield\_get\_slices}(\overbrace{\mathsf{BitField\_Simple}(\_, \texttt{slices})}^{\texttt{bf}}) \xrightarrow{\text{type}} \texttt{slices}$$

NESTED

$$\mathit{bitfield\_get\_slices}(\overbrace{\mathsf{BitField\_Nested}(\_, \texttt{slices}, \_)}^{\texttt{bf}}) \xrightarrow{\text{type}} \texttt{slices}$$

TYPE

$$\mathit{bitfield\_get\_slices}(\overbrace{\mathsf{BitField\_Type}(\_, \texttt{slices}, \_)}^{\texttt{bf}}) \xrightarrow{\text{type}} \texttt{slices}$$

### TypingRule.BitFieldGetNested

The function

$$\mathit{bitfield\_get\_nested} : \overbrace{\mathsf{bitfield}}^{\texttt{bf}} \longrightarrow \overbrace{\mathsf{bitfield}^*}^{\texttt{nested}}$$

returns the list of bitfields `nested` nested within the bitfield `bf`, if there are any, and an empty list if there are none.

### 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**

SIMPLE_TYPE

$$\frac{\mathit{ast\_label}(\texttt{bf}) \neq \mathsf{BitField\_Nested}}{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} [\,]}$$

NESTED

$$\mathit{bitfield\_get\_name}(\overbrace{\mathsf{BitField\_Nested}(\_, \_, \texttt{nested})}^{\texttt{bf}}) \xrightarrow{\text{type}} \texttt{nested}$$

### TypingRule.TBitField

The function

$$\mathit{annotate\_bitfield}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{Z}}^{\texttt{width}}, \overbrace{\mathsf{bitfield}}^{\texttt{field}}) \longrightarrow (\overbrace{\mathsf{bitfield}}^{\texttt{new\_field}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 $(\texttt{slices1}, \texttt{ses\_slices})^{/\!/\texttt{\#TE}}$;
- One of the following applies:
  - All of the following apply (SIMPLE):
    - checking whether the range of positions in `slices1` fits inside $0..\texttt{width}-1$ yields $\mathsf{TRUE}^{/\!/\texttt{\#TE}}$;
    - define `new_field` as the bitfield named `name` with list of slices `slices1`, that is, $\mathsf{BitField\_Simple}(\texttt{name}, \texttt{slices1})$.
  - All of the following apply (NESTED):

  - 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

$$\frac{\begin{array}{c}\mathit{annotate\_slices}(\mathsf{tenv},\mathtt{slices}) \xrightarrow{\mathsf{type}} (\mathtt{slices1},\mathtt{ses\_slices}) \;//\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{check\_slices\_in\_width}(\mathsf{tenv},\mathtt{width},\mathtt{slices1}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_bitfield}(\mathsf{tenv},\mathtt{width},\mathsf{BitField\_Simple}(\mathtt{name},\mathtt{slices})) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{BitField\_Simple}(\mathtt{name},\mathtt{slices1})}^{\mathtt{new\_field}},\overbrace{\mathtt{ses\_slices}}^{\mathtt{ses}})}$$

NESTED

$$\frac{\begin{array}{c}\mathit{annotate\_slices}(\mathtt{tenv}, \mathtt{slices}) \xrightarrow{\text{type}} (\mathtt{slices1}, \mathtt{ses\_slices}) \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{disjoint\_slices\_to\_positions}(\mathtt{tenv}, \mathtt{TRUE}, \mathtt{slices1}) \xrightarrow{\text{type}} \mathtt{positions} \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_positions\_in\_width}(\mathtt{tenv}, \mathtt{width}, \mathtt{positions}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{width'} := |\mathtt{positions}| \qquad \mathit{annotate\_bitfields}(\mathtt{tenv}, \mathtt{width'}, \mathtt{bitfields'}) \xrightarrow{\text{type}} \\ (\mathtt{bitfields''}, \mathtt{ses\_bitfields}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_slices} \cup \mathtt{ses\_bitfields}\end{array}}{\mathit{annotate\_bitfield}(\mathtt{tenv}, \mathtt{width}, \mathsf{BitField\_Nested}(\mathtt{name}, \mathtt{slices}, \mathtt{bitfields'})) \xrightarrow{\text{type}} (\overbrace{\mathsf{BitField\_Nested}(\mathtt{slices1}, \mathtt{bitfields''})}^{\mathtt{new\_field}}, \mathtt{ses})}$$

TYPE

$$\frac{\begin{array}{c}\mathit{annotate\_slices}(\mathtt{tenv}, \mathtt{slices}) \xrightarrow{\text{type}} (\mathtt{slices1}, \mathtt{ses\_slices}) \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{annotate\_type}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} (\mathtt{t'}, \mathtt{ses\_ty}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_slices\_in\_width}(\mathtt{tenv}, \mathtt{width}, \mathtt{slices1}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{disjoint\_slices\_to\_positions}(\mathtt{tenv}, \mathtt{TRUE}, \mathtt{slices1}) \xrightarrow{\text{type}} \mathtt{positions} \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_positions\_in\_width}(\mathtt{tenv}, \mathtt{slices1}, \mathtt{width}, \mathtt{positions}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{width'} := |\mathtt{positions}| \\ \mathit{check\_bits\_equal\_width}(\mathsf{T\_Bits}(\mathtt{width'}, [\,]), \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_slices} \cup \mathtt{ses\_ty}\end{array}}{\mathit{annotate\_bitfield}(\mathtt{tenv}, \mathtt{width}, \mathsf{BitField\_Type}(\mathtt{name}, \mathtt{slices}, \mathtt{t})) \xrightarrow{\text{type}} (\overbrace{\mathsf{BitField\_Type}(\mathtt{name}, \mathtt{slices1}, \mathtt{t'})}^{\mathtt{new\_field}}, \mathtt{ses})}$$

### TypingRule.CheckSlicesInWidth

The function

$$\mathit{check\_slices\_in\_width}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathbb{Z}}^{\mathtt{width}}, \overbrace{\mathsf{slice}^*}^{\mathtt{slices}}) \longrightarrow \{\mathtt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

checks whether the slices in `slices` fit within the bitvector width given by `width` in tenv, yielding `TRUE`. Otherwise, the result is a type error.

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

$$\frac{\mathit{disjoint\_slices\_to\_positions}(\mathsf{tenv}, \mathsf{TRUE}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{positions} \;/\!/\; \#TE \qquad \mathit{check\_positions\_in\_width}(\texttt{width}, \texttt{positions}) \xrightarrow{\text{type}} \mathsf{TRUE} \;/\!/\; \#TE}{\mathit{check\_slices\_in\_width}(\mathsf{tenv}, \texttt{width}, \texttt{slices}) \xrightarrow{\text{type}} \mathsf{TRUE}}$$

### TypingRule.CheckPositionsInWidth

The function

$$\mathit{check\_positions\_in\_width}(\overbrace{\mathbb{Z}}^{\texttt{width}}, \overbrace{\mathcal{P}(\mathbb{Z})}^{\texttt{positions}}) \longrightarrow \{\mathsf{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#TE}$$

checks whether the set of positions in `positions` fit within the bitvector width given by `width`, yielding TRUE. Otherwise, the result is a type error.

### 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**

$$\frac{\texttt{min\_pos} := \min(\texttt{positions}) \qquad \texttt{max\_pos} := \max(\texttt{positions}) \qquad \mathit{check}(0 \leq \texttt{min\_pos} \wedge \texttt{max\_pos} < \texttt{width}, \texttt{TE\_BS}) \xrightarrow{\text{type}} \texttt{TRUE} \ /\!/ \ \texttt{\#TE}}{\mathit{check\_positions\_in\_width}(\texttt{width}, \texttt{positions}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

**TypingRule.DisjointSlicesToPositions**

The function

$$\mathit{disjoint\_slices\_to\_positions}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathbb{B}}^{\text{is\_static}}, \overbrace{\mathsf{slice}^*}^{\text{slices}}) \longrightarrow \overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^{\text{positions}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

returns the set of integers defined by the list of slices in `slices` in `positions`. In particular, this rule checks that the 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 of 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 ⟨`s1`⟩ 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 `s1` if `positions2_opt` is ⟨`s2`⟩ 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**

EMPTY

$$\mathit{disjoint\_slices\_to\_positions}(\mathsf{tenv}, \mathtt{is\_static}, \overbrace{[\,]}^{\mathtt{slices}}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{positions}}$$

NON_EMPTY

$$\frac{\begin{array}{c}\mathit{bitfield\_slice\_to\_positions}(\mathsf{tenv}, \mathtt{is\_static}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{positions1\_opt} \;/\!/\; \mathtt{\#TE} \\ \mathtt{positions1} := \mathsf{choice}(\mathtt{positions1\_opt} = \langle \mathtt{s1} \rangle, \mathtt{s1}, \emptyset) \\ \mathit{disjoint\_slices\_to\_positions}(\mathsf{tenv}, \mathtt{is\_static}, \mathtt{slices1}) \xrightarrow{\mathsf{type}} \mathtt{positions2\_opt} \;/\!/\; \mathtt{\#TE} \\ \mathtt{positions2} := \mathsf{choice}(\mathtt{positions2\_opt} = \langle \mathtt{s2} \rangle, \mathtt{s2}, \emptyset) \\ \mathit{check}(\mathtt{positions1} \cap \mathtt{positions2} = \emptyset, \mathtt{TE\_BS}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{disjoint\_slices\_to\_positions}(\mathsf{tenv}, \mathtt{is\_static}, \overbrace{\mathtt{s} + \mathtt{slices1}}^{\mathtt{slices}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{positions1} \cup \mathtt{positions2}}^{\mathtt{positions}}}$$

**TypingRule.BitfieldSliceToPositions**

The function

$$\mathit{bitfield\_slice\_to\_positions}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathbb{B}}^{\mathtt{is\_static}}, \overbrace{\mathsf{slice}}^{\mathtt{slice}}) \longrightarrow \overbrace{\langle \mathcal{P}_{\mathrm{fin}}(\mathbb{Z}) \rangle}^{\mathtt{positions}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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`$^{/\!/\texttt{\#TE,None}}$;
- applying *eval_slice_expr* to `tenv`, `is_static`, and `e2`, yields the integer literal for `length`$^{/\!/\texttt{\#TE,None}}$;
- checking that `offset` is less than or equal to `offset` + `length` − 1 holds yields `TRUE`$^{/\!/\texttt{TE\_BS}}$;
- `positions` is the set of integers between `offset` and `offset` + `length` − 1, inclusive.

**Formally**

$$\frac{\begin{array}{c}\mathit{eval\_slice\_expr}(\texttt{tenv}, \texttt{is\_static}, \texttt{e1}) \xrightarrow{\text{type}} \langle \texttt{offset} \rangle \;/\!/\; \texttt{\#TE}, \texttt{None} \\ \mathit{eval\_slice\_expr}(\texttt{tenv}, \texttt{is\_static}, \texttt{e2}) \xrightarrow{\text{type}} \langle \texttt{length} \rangle \;/\!/\; \texttt{\#TE}, \texttt{None} \\ \mathit{check}(\texttt{offset} \leq \texttt{offset} + \texttt{length} - 1, \texttt{TE\_BS}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{bitfield\_slice\_to\_positions}(\texttt{tenv}, \texttt{is\_static}, \overbrace{\texttt{Slice\_Length}(\texttt{e1}, \texttt{e2})}^{\texttt{slice}}) \xrightarrow{\text{type}} \overbrace{\langle \{n \mid \texttt{offset} \leq n \leq \texttt{offset} + \texttt{length} - 1\} \rangle}^{\texttt{positions}}}$$

**TypingRule.EvalSliceExpr**

The function

$$\mathit{eval\_slice\_expr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{B}}^{\texttt{is\_static}}, \overbrace{\texttt{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\langle \mathbb{Z} \rangle}^{\texttt{z\_opt}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

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

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

STATIC

$$\frac{\mathit{static\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \texttt{z} \;/\!/\; \texttt{\#TE}}{\mathit{eval\_slice\_expr}(\mathsf{tenv}, \overbrace{\texttt{TRUE}}^{\texttt{is\_static}}, \mathsf{e}) \xrightarrow{\text{type}} \overbrace{\langle \texttt{z} \rangle}^{\texttt{z\_opt}}}$$

SYMBOLIC

$$\frac{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \texttt{z\_opt}}{\mathit{eval\_slice\_expr}(\mathsf{tenv}, \overbrace{\texttt{FALSE}}^{\texttt{is\_static}}, \mathsf{e}) \xrightarrow{\text{type}} \texttt{z\_opt}}$$

### TypingRule.CheckCommonBitfieldsAlign

The function

$$\mathit{check\_common\_bitfields\_align}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{bitfield}^*}^{\texttt{bitfields}}, \overbrace{\mathbb{N}}^{\texttt{width}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

checks 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 $\mathbb{ABF} := (\overbrace{\mathsf{identifier}^*}^{\texttt{name}}, \overbrace{\mathbb{N}^*}^{\texttt{slice}})$, where the component `name` is a list of identifiers corresponding to the absolute name, and the component

`slice` corresponds to an absolute slice by listing the indices into the containing bitvector type.[1]

Premises in TypingRule.TBits guarantee that `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 $\texttt{TRUE}/\!\!/^{\texttt{TE\_BS}}$;
  - the result is `TRUE`.

[1] An implementation of the type system may compactly represent the list of indices via a list of intervals, each represented by its limits.

**Formally**

$$\frac{\text{EMPTY}}{\texttt{bitfields} = [\,]}{\mathit{check\_common\_bitfields\_align}(\mathsf{tenv}, \texttt{bitfields}, \overbrace{0}^{\texttt{width}}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

$$\frac{\text{NON\_EMPTY}}{\begin{array}{c}\texttt{bitfields} \neq [\,] \\ \texttt{last\_index} := \texttt{width} - 1 \qquad \texttt{top\_absolute} := ([\,], \texttt{last\_index}..0) \\ \mathit{bitfields\_to\_absolute}(\mathsf{tenv}, \texttt{bitfields}, \texttt{top\_absolute}) \xrightarrow{\mathsf{type}} \texttt{fs} \\ \mathit{check}(\forall \texttt{f1}, \texttt{f2} \in \texttt{fs} : \mathit{absolute\_bitfields\_align}(\texttt{f1}, \texttt{f2}), \mathtt{TE\_BS}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;\; /\!\!/ \;\; \mathtt{\#TE}\end{array}}{\mathit{check\_common\_bitfields\_align}(\mathsf{tenv}, \texttt{bitfields}, \texttt{width}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

### TypingRule.BitfieldsToAbsolute

The function

$$\mathit{bitfields\_to\_absolute}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{bitfield}^{+}}^{\texttt{bitfields}}, \overbrace{\mathbb{ABF}}^{\texttt{absolute\_parent}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{ABF})}^{\texttt{abs\_bitfields}}$$

returns the set of absolute bitfields `abs_bitfields` that correspond to the list of bitfields `bitfields`, whose bitfield scope and absolute slice is given by `absolute_parent`, in the static environment tenv.

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 $\mathtt{a_f}$;
- define `abs_bitfields` as the union of the sets $\mathtt{a_f}$, for every `f` in `bitfields`.

**Formally**

$$\frac{\begin{array}{c}\texttt{f} \in \texttt{bitfields} : \mathit{bitfield\_to\_absolute}(\mathsf{tenv}, \texttt{f}, \texttt{absolute\_parent}) \xrightarrow{\mathsf{type}} \mathtt{a_f} \\ \texttt{abs\_bitfields} := \bigcup_{\texttt{f} \in \texttt{bitfields}} \mathtt{a_f}\end{array}}{\mathit{bitfields\_to\_absolute}(\mathsf{tenv}, \texttt{bitfields}, \texttt{absolute\_parent}) \xrightarrow{\mathsf{type}} \texttt{abs\_bitfields}}$$

### TypingRule.BitfieldToAbsolute

The function

$$\mathit{bitfield\_to\_absolute}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{bitfield}}^{\texttt{bf}}, \overbrace{\mathbb{ABF}}^{\texttt{absolute\_parent}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{ABF})}^{\texttt{abs\_bitfields}}$$

returns the set of absolute bitfields `abs_bitfields` that correspond to the bitfields nested in `bf`, including itself, where the bitfield scope and absolute slice of the bitfield containing `bf` are `absolute_parent`, in the static environment `tenv`.

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 $\texttt{indices}_\texttt{s}$;
- define `slices_as_indices` as the concatenation of the sequences $\texttt{indices}_\texttt{s}$, for each slice `s` in `slices`, in their order of appearance in `slices`;
- 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**

$$\frac{\begin{array}{c}\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \\ \texttt{bf\_name} := \texttt{absolute\_name} + [\texttt{name}] \qquad \mathit{bitfield\_get\_slices}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{slices} \\ \texttt{s} \in \texttt{slices} : \mathit{slice\_to\_indices}(\texttt{tenv}, \texttt{s}) \xrightarrow{\text{type}} \texttt{indices}_\texttt{s} \\ \texttt{slices\_as\_indices} := [\texttt{s} \in \texttt{slices} : \texttt{indices}_\texttt{s}] \\ \mathit{select\_indices\_by\_slices}(\texttt{absolute\_slices}, \texttt{slices\_as\_indices}) \xrightarrow{\text{type}} \texttt{bf\_indices} \\ \texttt{bf\_absolute} := (\texttt{bf\_name}, \texttt{bf\_indices}) \qquad \mathit{bitfield\_get\_nested}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{nested} \\ \mathit{bitfields\_to\_absolute}(\texttt{tenv}, \texttt{nested}, \texttt{bf\_absolute}) \xrightarrow{\text{type}} \texttt{abs\_bitfields1}\end{array}}{\mathit{bitfield\_to\_absolute}(\texttt{tenv}, \texttt{bf}, \overbrace{(\texttt{absolute\_name}, \texttt{absolute\_slices})}^{\texttt{absolute\_parent}}) \xrightarrow{\text{type}} \overbrace{\{\texttt{bf\_absolute}\} \cup \texttt{abs\_bitfields1}}^{\texttt{abs\_bitfields}}}$$

**TypingRule.SelectIndicesBySlices**

The function

$$\mathit{select\_indices\_by\_slices}(\ \overbrace{\mathbb{N}^+}^{\texttt{indices}}\ ,\ \overbrace{\mathbb{N}^+}^{\texttt{slice\_indices}}\ ) \longrightarrow \overbrace{\mathbb{N}^*}^{\texttt{absolute\_slice}}$$

considers the list `indices` as a list of indices into a bitvector type (essentially, a slice of it), and the list `slice_indices` as a list of indices into `indices` (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:

$$\begin{array}{lcl}\mathit{select\_indices\_by\_slices}([9,4,6,1,13],[4,3,2,1,0]) & \xrightarrow{\text{type}} & [9,4,6,1,13] \\ \mathit{select\_indices\_by\_slices}([9,4,6,1,13],[0,1,2,3,4]) & \xrightarrow{\text{type}} & [13,1,6,4,9] \\ \mathit{select\_indices\_by\_slices}([9,4,6,1,13],[3,2,4,0]) & \xrightarrow{\text{type}} & [4,6,9,13]\end{array}$$

**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 $I[S_m] \ldots I[S_0]$.

**Formally**

$$\mathit{select\_indices\_by\_slices}(\ \overbrace{I_{n..0}}^{\texttt{indices}}\ ,\ \overbrace{S_{m..0}}^{\texttt{slice\_indices}}\ ) \xrightarrow{\text{type}} \overbrace{I[S_m] \ldots I[S_0]}^{\texttt{absolute\_slice}}$$

### TypingRule.AbsoluteBitfieldsAlign

The function

$$\mathit{absolute\_bitfields\_align}(\overbrace{\mathbb{ABF}}^{\texttt{f}}, \overbrace{\mathbb{ABF}}^{\texttt{g}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

tests whether the absolute bitfields `f1` and `f2` share the same name and exist in the same scope. If they do, `b` indicates whether their absolute slices are equal. Otherwise, the result 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 $\texttt{f}_{1..k}$ and absolute slice `slice1`;
- `g` is an absolute bitfield with absolute name $\texttt{g}_{1..n}$ and absolute slice `slice2`;
- define `name1` to be the name of the bitfield corresponding to `f`, that is, $\texttt{f}_k$;
- define `name2` to be the name of the bitfield corresponding to `g`, that is, $\texttt{g}_n$;
- define `scope1` to be the bitfield scope of `f`, that is, $\texttt{f}_{1..k-1}$;
- define `scope2` to be the bitfield scope of `g`, that is, $\texttt{g}_{1..n-1}$;
- define `same_scope` as `TRUE` if and only if `scope1` is a prefix of `scope2` or vice versa;
- define `b` as `TRUE` if and only if `name1` and `name2` are equal and `same_scope` is `TRUE` implies that `slice1` is equal to `slice2`.

### Formally

$$\frac{\begin{array}{c}\texttt{name1} := \texttt{f}_k \qquad \texttt{name2} := \texttt{g}_n \qquad \texttt{scope1} := \texttt{f}_{1..k-1} \\ \texttt{scope2} := \texttt{g}_{1..n-1} \qquad \texttt{same\_scope} := \mathit{prefix}(\texttt{scope1}, \texttt{scope2}) \lor \mathit{prefix}(\texttt{scope2}, \texttt{scope1}) \\ \texttt{b} := (\texttt{name1} = \texttt{name2} \ \land \ \texttt{same\_scope}) \implies (\texttt{slice1} = \texttt{slice2})\end{array}}{\mathit{absolute\_bitfields\_align}(\overbrace{(\texttt{f}_{1..k}, \texttt{slice1})}^{\texttt{f}}, \overbrace{(\texttt{g}_{1..n}, \texttt{slice2})}^{\texttt{f}}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.SliceToIndices

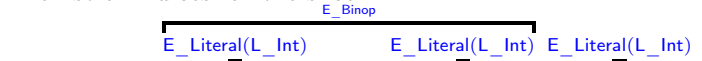
The function

$$\mathit{slice\_to\_indices}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{slice}}^{\texttt{s}}) \longrightarrow \overbrace{\mathbb{N}^*}^{\texttt{indices}}$$

returns the list of indices `indices` represented by the bitvector slice `s` in the static environment tenv.

### Example: Converting a Slice to a List of Indices

The list of indices for the slice




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 $\texttt{z}_\texttt{i}$;
- statically evaluating the expression `w` in the static environment `tenv` yields the literal the literal for the integer $\texttt{z}_\texttt{w}$;
- define `v_start` as $\texttt{z}_\texttt{i}$;
- define `v_end` as $\texttt{z}_\texttt{i} + \texttt{z}_\texttt{w} - 1$;
- define `indices` as the list of integers starting at `v_end` and counting down to `v_start`.

### Formally

$$\frac{\mathit{static\_eval}(\mathsf{tenv}, \texttt{i}) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(\texttt{z}_\texttt{i}) \qquad \mathit{static\_eval}(\mathsf{tenv}, \texttt{w}) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(\texttt{z}_\texttt{w}) \qquad \texttt{v\_start} := \texttt{z}_\texttt{i} \qquad \texttt{v\_end} := \texttt{z}_\texttt{i} + \texttt{z}_\texttt{w} - 1}{\mathit{slice\_to\_indices}(\mathsf{tenv}, \overbrace{\mathsf{Slice\_Length}(\texttt{i}, \texttt{w})}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\texttt{v\_end..v\_start}}^{\texttt{indices}}}$$

# 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

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

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

All expressions have a unique type (which can be a tuple type). The function

$$annotate\_expr(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{expr}}^{\texttt{e}}) \longrightarrow (\overbrace{\texttt{ty}}^{\texttt{t}} \times \overbrace{\texttt{expr}}^{\texttt{new\_e}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

specifies how to annotate an expression `e` in an environment `tenv`. The result of annotating the expression `e` in `tenv` is the tuple (`t`, `new_e`, `ses`), where `t` is the type inferred for `e`, `new_e` is the typed AST for `e`, also known as the *annotated expression*, and `ses` is the set of side effect descriptors inferred for `e`. Otherwise, the result is a type error.

The annotation rewrites the input expression in the following case, making the annotation of statements simpler: variables with constant values are substituted by their constant values.

The relation

$$eval\_expr(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\texttt{expr}}^{\texttt{e}}) \bigtimes \mathsf{Normal}((\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

evaluates the expression `e` in an environment `env` and 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

expr ⟶ value

### 15.1.2 Abstract Syntax

expr ⟶ E_Literal(literal)

**ASTRule.ELit**

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{value})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Literal}(\texttt{value})}^{\texttt{ast\_node}}$$

### 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**

$$\frac{\mathit{annotate\_literal}(\text{tenv}, \texttt{v}) \xrightarrow{\text{type}} \texttt{t}}{\mathit{annotate\_expr}(\text{tenv}, \overbrace{\text{E\_Literal}(\texttt{v})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \overbrace{\text{E\_Literal}(\texttt{v})}^{\texttt{new\_e}}, \overbrace{\emptyset}^{\texttt{ses}})}$$

### 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 `l`, that is, E_Literal(`l`)
- `v` is the native value corresponding to `l`;
- `g` is the empty graph, as literals do not yield any Read and Write Effects;
- `new_env` is `env`.

**Formally**

$$\mathit{eval\_expr}(\texttt{env}, \overbrace{\mathsf{E\_Literal}(\texttt{l})}^{\texttt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\overbrace{\texttt{NV\_Literal}(\texttt{l})}^{\texttt{v}}, \overbrace{\emptyset_{\mathsf{g}}}^{\texttt{g}}), \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

## 15.2 Variable Expressions

A variable expression consists of an identifier for a storage element.

### 15.2.1 Syntax

expr ⟶ **ID**

### 15.2.2 Abstract Syntax

$$\mathsf{expr} \longrightarrow \mathsf{E\_Var}(\ \overbrace{\mathsf{identifier}}^{\text{variable name}}\ )$$

**ASTRule.EVAR**

$$\mathit{build\_expr}(\overbrace{\mathsf{expr}(\mathbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{E\_Var}(\texttt{id})}^{\texttt{ast\_node}}$$

### 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
    constant LOCAL_CONSTANT = 7;
    var var_x = LOCAL_CONSTANT; // The annotated expression for LOCAL_CONSTANT is 7.
    var y = var_x; // The annotated expression for var_x is var_x.
    var local_non_constant = LOCAL_CONSTANT;
    var z = local_non_constant + GLOBAL_CONSTANT + global_non_constant;
    return 0;
end;
```

The type system annotates the expression `LOCAL_CONSTANT` as the literal for `7`, since it is declared as a `constant`, whereas the expression `var_x` on the right-hand-side of the assignment `var y = var_x;` is annotated as `var_x`, since `var_x` is not declared as a `constant`.

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_CONSTANT):
    - `x` is bound to the type `t` and local declaration keyword LDK_Constant via the local_storage_types map of the local environment component of tenv;
    - `x` is bound to the literal `v` via the constant_values map of the local environment of tenv;
    - define `new_e` as the literal expression for `v`, that is E_Literal(`v`);
    - define `ses` as the empty set.
  - All of the following apply (LOCAL_NON_CONSTANT):
    - `x` is bound to the type `t` and local declaration keyword $k$ via the local_storage_types map of the local environment component of tenv;
    - either $k$ is different from LDK_Constant or `x` is not bound in the constant_values map of the local environment of tenv;
    - define `new_e` as `e`;

  - define `ses` as the singleton set for the local read side effect descriptor for `x` the time frame of $k$ (*time_frame_ldk*) and the immutability status of $k$ (*ldk_is_immutable*).
* All of the following apply (GLOBAL_CONSTANT):
  - `x` is not bound via the local_storage_types map of the local component of tenv;
  - `x` is bound to (`ty`, GDK_Constant) via the global_storage_types map of the global component of tenv;
  - `x` is bound to `v` via the constant_values map of the global component of tenv;
  - define *newe* as the literal expression for `v`;
  - define `ses` as the empty set.
* All of the following apply (GLOBAL_NON_CONSTANT):
  - `x` is not bound via the local_storage_types map of the local component of tenv;
  - `x` is bound to (`ty`, $k$) via the global_storage_types map of the global component of tenv;
  - either `x` is not bound in the constant_values map of the global component of tenv or $k$ is not GDK_Constant;
  - define *newe* as `e`;
  - define `ses` as the singleton set for the global read side effect descriptor for `x` the time frame of $k$ (*time_frame_gdk*) and the immutability status of $k$ (*gdk_is_immutable*).
* All of the following apply (ERROR_UNDEFINED):
  - `x` is not bound via the local_storage_types map of the local component of tenv;
  - `x` is not bound via the global_storage_types map of the local component of tenv;
  - the result is a type error indicating that `x` is an undefined identifier (`TE_UI`).

**Formally**

LOCAL_CONSTANT

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{x}) = (\mathtt{t}, \mathsf{LDK\_Constant}) \qquad L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{x}) = \mathtt{v}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \overbrace{\mathsf{E\_Literal}(\mathtt{v})}^{\mathtt{new\_e}}, \overbrace{\emptyset}^{\mathtt{ses}})}$$

LOCAL_NON_CONSTANT

$$\frac{\begin{array}{c} L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{x}) = (\mathtt{t}, k) \\ L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{x}) = \bot \vee k \neq \mathsf{LDK\_Constant} \\ \mathtt{ses} := \{\ \mathsf{ReadLocal}(\mathtt{x}, \mathit{time\_frame\_ldk}(k), \mathit{ldk\_is\_immutable}(k))\ \} \end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{new\_e}}, \mathtt{ses})}$$

GLOBAL_CONSTANT

$$\frac{L^{\text{tenv}}.\text{local\_storage\_types}(\texttt{x}) = \bot \qquad G^{\text{tenv}}.\text{global\_storage\_types}(\texttt{x}) = (\texttt{ty}, \text{GDK\_Constant}) \qquad G^{\text{tenv}}.\text{constant\_values}(\texttt{x}) = \texttt{v}}{\mathit{annotate\_expr}(\text{tenv}, \overbrace{\text{E\_Var}(\texttt{x})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{ty}, \overbrace{\text{E\_Literal}(\texttt{v})}^{\texttt{new\_e}}, \overbrace{\emptyset}^{\texttt{ses}})}$$

GLOBAL_NON_CONSTANT

$$\frac{L^{\text{tenv}}.\text{local\_storage\_types}(\texttt{x}) = \bot \qquad G^{\text{tenv}}.\text{global\_storage\_types}(\texttt{x}) = (\texttt{ty}, k) \qquad G^{\text{tenv}}.\text{constant\_values}(\texttt{x}) = \bot \vee k \neq \text{GDK\_Constant} \qquad \texttt{ses} := \{\ \text{ReadGlobal}(\texttt{x}, \mathit{time\_frame\_gdk}(k), \mathit{gdk\_is\_immutable}(k))\ \}}{\mathit{annotate\_expr}(\text{tenv}, \overbrace{\text{E\_Var}(\texttt{x})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{ty}, \overbrace{\text{E\_Var}(\texttt{x})}^{\texttt{new\_e}}, \texttt{ses})}$$

ERROR_UNDEFINED

$$\frac{L^{\text{tenv}}.\text{local\_storage\_types}(\texttt{x}) = \bot \qquad G^{\text{tenv}}.\text{global\_storage\_types}(\texttt{x}) = \bot}{\mathit{annotate\_expr}(\text{tenv}, \overbrace{\text{E\_Var}(\texttt{x})}^{\texttt{e}}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_UI})}$$

### 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

LOCAL

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

GLOBAL

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

### Comments

When there exists a global variable `x`, the type system forbids having `x` as a local variable. This is enforced by TypingRule.LDVar in the Chapter "Typing of Local Declarations", and TypingRule.DeclareGlobalStorage and TypingRule.DeclareOneFunc, both in the Chapter "Typing of Global Declarations".

## 15.3 Binary Expressions

### 15.3.1 Syntax

expr $\longrightarrow$ expr binop expr

### 15.3.2 Abstract Syntax

expr $\longrightarrow$ E_Binop(binop, expr, expr)

#### ASTRule.EBinop

The following rule constructs a binary expression AST and checks that a requirement on *associative operators* holds (see ASTRule.CheckNotSamePrec).

$$\frac{\begin{array}{c} \mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \;/\!/\; \texttt{\#BE} \\ \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast} \;/\!/\; \texttt{\#BE} \\ \mathit{check\_not\_same\_prec}(\overline{\texttt{binop}}, \texttt{e1\_ast}) \xrightarrow{\text{ast}} \texttt{TRUE} \;/\!/\; \texttt{\#BE} \\ \mathit{check\_not\_same\_prec}(\overline{\texttt{binop}}, \texttt{e2\_ast}) \xrightarrow{\text{ast}} \texttt{TRUE} \;/\!/\; \texttt{\#BE} \end{array}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e1} : \texttt{expr}, \texttt{binop}, \texttt{e2} : \texttt{expr})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Binop}(\texttt{e1\_ast}, \overline{\texttt{binop}}, \texttt{e2\_ast})}^{\text{ast\_node}}}$$

#### ASTRule.CheckNotSamePrec

The set of *associative binary operators* consists of the following: BOR, BAND, IMPL, BEQ, EQ_OP, NEQ, PLUS, MINUS, OR, XOR, AND, MUL, DIV, DIVRM, RDIV, MOD, SHL, SHR, POW.

We define the helper function

$$\mathit{binop\_prec}(\overbrace{\texttt{binop}}^{\text{op}}) \longrightarrow \mathbb{N}$$

which assigns a precedence level to each binary operator op, as defined below:

$$\mathit{binop\_prec}(\texttt{op}) \xrightarrow{\text{ast}} \begin{cases} 0 & \text{if op} \in \{\texttt{GT}, \texttt{GEQ}, \texttt{LT}, \texttt{LEQ}\} \\ 1 & \text{if op} \in \{\texttt{BOR}, \texttt{BAND}, \texttt{IMPL}, \texttt{BEQ}\} \\ 2 & \text{if op} \in \{\texttt{EQ\_OP}, \texttt{NEQ}\} \\ 3 & \text{if op} \in \{\texttt{PLUS}, \texttt{MINUS}, \texttt{OR}, \texttt{XOR}, \texttt{AND}\} \\ 4 & \text{if op} \in \{\texttt{MUL}, \texttt{DIV}, \texttt{DIVRM}, \texttt{RDIV}, \texttt{MOD}, \texttt{SHL}, \texttt{SHR}\} \\ 5 & \text{if op} = \texttt{POW} \end{cases}$$

The function

$$\mathit{check\_not\_same\_prec}(\overbrace{\mathsf{binop}}^{\mathsf{op}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \{\mathtt{TRUE}\} \cup \overbrace{\mathsf{TBuildError}}^{\mathtt{\#BE}}$$

checks whether the expression AST node `e` is a binary operator of the same *precedence* as that of the binary operator `op`. If so, it is considered an error. Surrounding `e` by parenthesis fixes the error.

For example, `a + b + c` is considered legal, since the same binary operator (+) is used, whereas `a + b - c` is considered illegal, since `PLUS` and `MINUS` have the same precedence (3). To fix this, we can surround one of the subexpressions with parenthesis, for example: `(a + b) - c`.

#### Prose

One of the following applies:

- All of the following apply (NOT_BINOP):
  - `e` is not a binary operation expression;
  - the result is `TRUE`.
- All of the following apply (BINOP):
  - `e` is a binary operation expression for the operator `op'`;
  - checking whether `op` is different from `op'` implies that `op` and `op'` have different precedence levels yields $\mathtt{TRUE}^{/\!/\mathtt{BE\_BOP}}$.

#### Formally

$$\text{NOT\_BINOP}\ \frac{\mathit{ast\_label}(\mathsf{e}) \neq \mathsf{E\_Binop}}{\mathit{check\_not\_same\_prec}(\mathsf{op}, \mathsf{e}) \xrightarrow{\mathsf{ast}} \mathtt{TRUE}}$$

$$\text{BINOP}\ \frac{\mathit{check}(\mathtt{op} \neq \mathtt{op'} \implies \mathit{binop\_prec}(\mathtt{op}) \neq \mathit{binop\_prec}(\mathtt{op'}), \mathtt{BE\_BOP}) \longrightarrow \mathtt{TRUE}\ /\!/\ \mathtt{\#BE}}{\mathit{check\_not\_same\_prec}(\mathsf{op}, \overbrace{\mathsf{E\_Binop}(\mathtt{op'}, \_, \_)}^{\mathsf{e}}) \xrightarrow{\mathsf{ast}} \mathtt{TRUE}}$$

### 15.3.3 Typing

#### TypingRule.EBinop

#### Example: Typing of Binary Expressions

In Listing 15.6, the expression `3 DIV 0` results in a type error.

Listing 15.6: Evaluating a binary expression resulting in a type error

```
func main () => integer
begin

  let x = 3 DIV 0;

  return 0;
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`)$^{/\!/\texttt{\#TE}}$;
- annotating the expression `e2` in the static environment `tenv` yields (`t2`, `e2'`, `ses2`)$^{/\!/\texttt{\#TE}}$;
- applying `op` to the type `t1` and type `t2` in the static environment `tenv` yields the type `t`$^{/\!/\texttt{\#TE}}$;
- define `new_e` as the binary expression `op` over `e1'` and `e2'`;
- One of the following applies:
  - All of the following apply (ORDERED):
    - `op` is one of `BAND`, `BOR`, or `IMPL`;
    - define `ses` as the union of `ses1` and `ses2`.
  - All of the following apply (UNORDERED):
    - `op` is not one of `BAND`, `BOR`, or `IMPL`;
    - define `ses` as the union of `ses1` and `ses2`.

### Formally

ORDERED

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t1}, \texttt{e1'}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} (\texttt{t2}, \texttt{e2'}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{t} \;/\!/\; \texttt{\#TE} \\ \texttt{op} \in \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \text{******** common prefix ********} \\ \texttt{ses} := \texttt{ses1} \cup \texttt{ses2} \end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \overbrace{\text{E\_Binop}(\texttt{op}, \texttt{e1}', \texttt{e2}')}^{\texttt{new\_e}}, \texttt{ses})}$$

UNORDERED

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} (\mathtt{t1}, \mathtt{e1'}, \mathtt{ses1}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e2}) \xrightarrow{\text{type}} (\mathtt{t2}, \mathtt{e2'}, \mathtt{ses2}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{apply\_binop\_types}(\mathtt{tenv}, \mathtt{op}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\text{type}} \mathtt{t} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathtt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \text{******** common prefix ********} \\ \mathtt{ses} := \mathtt{ses1} \cup \mathtt{ses2}\end{array}}{\mathit{annotate\_expr}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{op}, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\text{type}} (\mathtt{t}, \overbrace{\mathsf{E\_Binop}(\mathtt{op}, \mathtt{e1'}, \mathtt{e2'})}^{\mathtt{new\_e}}, \mathtt{ses})}$$

### 15.3.4 Semantics

#### SemanticsRule.BinopAnd

#### Example: Evaluation of Binary And Expressions

In Listing 15.7, the expression `FALSE && fail()` evaluates to the value `FALSE`. Notice that the function `fail` is never called.

Listing 15.7: 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, $\mathsf{E\_Binop}(\texttt{BAND}, \mathtt{e1}, \mathtt{e2})$;
- $C$ is the result of the evaluation of the expression `if e1 then e2 else false` (see SemanticsRule.ECond).

#### Formally

$$\frac{\mathtt{false'} := \mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{FALSE})) \qquad \mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_Cond}(\mathtt{e1}, \mathtt{e2}, \mathtt{false'})) \xrightarrow{\text{eval}} C}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_Binop}(\texttt{BAND}, \mathtt{e1}, \mathtt{e2})) \xrightarrow{\text{eval}} C}$$

### 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 Binary Or Expressions

In Listing 15.8, the expression `(0 == 1) || (1 == 1)` evaluates to the value `TRUE`.

Listing 15.8: 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 SemanticsRule.ECond).

### Formally

$$\frac{\texttt{true'} := \mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{TRUE})) \qquad \mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Cond}(\texttt{e1}, \texttt{true'}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{BOR}, \texttt{e1}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}$$

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to `FALSE`, is `e2` evaluated.

### Comments

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 15.9, the expression `(0 == 1) -> (1 == 0)` evaluates to the value `TRUE`, according to the definition of implication.

Listing 15.9: 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

$$\frac{\texttt{true'} := \mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{TRUE})) \qquad \mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Cond}(\texttt{e1}, \texttt{e2}, \texttt{true'})) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{IMPL}, \texttt{e1}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}$$

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to `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 15.10, the expression `3 + 2` evaluates to the value `5`.

Listing 15.10: 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**

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

The rule above applies to many binary operators, including `EQ_OP` (which is used for `<->` as well as `==`).

**Comments**

## 15.4 Unary Expressions

### 15.4.1 Syntax

$\text{expr} \longrightarrow \text{unop expr}$

### 15.4.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E\_Unop}(\text{unop}, \text{expr})$

**ASTRule.EUnop**

$$\frac{\mathit{build\_expr}(\texttt{expr}) \xrightarrow{\text{ast}} \texttt{expr\_ast} \;/\!/\; \#\texttt{BE}}{\mathit{build\_expr}(\overbrace{\text{expr}(\text{unop}, \texttt{expr} : \text{expr})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Unop}(\overline{\text{unop}}, \texttt{expr\_ast})}^{\texttt{ast\_node}}}$$

### 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 $\text{E\_Unop}(\texttt{op}, \texttt{e'})$;
- annotating `e'` in `tenv` yields $(\texttt{t''}, \texttt{e''}, \texttt{ses})^{/\!/\#\texttt{TE}}$;

- checking compatibility of `op` with `t''` as per TypingRule.ApplyUnopType yields $\texttt{t}^{/\!/\texttt{\#TE}}$;
- define `new_e` as `op` over `e''`, that is, $\mathsf{E\_Unop}(\texttt{op}, \texttt{e''})$.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \texttt{e'}) \xrightarrow{\mathsf{type}} (\texttt{t''}, \texttt{e''}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{apply\_unop\_type}(\mathsf{tenv}, \texttt{op}, \texttt{t''}) \xrightarrow{\mathsf{type}} \texttt{t} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{E\_Unop}(\texttt{op}, \texttt{e'})) \xrightarrow{\mathsf{type}} (\texttt{t}, \mathsf{E\_Unop}(\texttt{op}, \texttt{e''}), \texttt{ses})}$$

### 15.4.4 Semantics

**SemanticsRule.Unop**

**Example: Evaluation of a Unary Operation Expression**

In Listing 15.11, the expression `NOT '1010'` evaluates to the value `'0101'`.

Listing 15.11: 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, $\mathsf{E\_Unop}(\texttt{op}, \texttt{e1})$;
- the evaluation of the expression `e1` in `env` yields $\mathsf{Normal}((\texttt{v1}, \texttt{g}), \texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- applying the unary operator `op` to `v1` is `v`.

**Formally**

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v1}, \texttt{g}), \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{unop}(\texttt{op}, \texttt{v1}) \xrightarrow{\mathsf{eval}} \texttt{v}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Unop}(\texttt{op}, \texttt{e1})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

## 15.5 Conditional Expressions

### 15.5.1 Syntax

expr ⟶ "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**

$$\begin{aligned} \textit{build\_expr}(\texttt{cond\_expr}) &\xrightarrow{\text{ast}} \texttt{cond\_expr\_ast} \;/\!/\; \#\text{BE} \\ \textit{build\_expr}(\texttt{then\_expr}) &\xrightarrow{\text{ast}} \texttt{then\_expr\_ast} \;/\!/\; \#\text{BE} \\ \textit{build\_expr}(\texttt{else\_expr}) &\xrightarrow{\text{ast}} \texttt{else\_expr\_ast} \;/\!/\; \#\text{BE} \end{aligned}$$

$$\textit{build\_expr}\left(\overbrace{\text{expr}\left(\begin{array}{l}\texttt{"if"}, \texttt{cond\_expr} : \text{expr}, \texttt{"then"}, \\ \hookrightarrow\ \texttt{then\_expr} : \text{expr}, \texttt{"else"}, \texttt{else\_expr} : \text{expr}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\text{E\_Cond}(\texttt{cond\_expr\_ast}, \texttt{then\_expr\_ast}, \texttt{else\_expr\_ast})}^{\texttt{ast\_node}}$$

### 15.5.3 Typing

**TypingRule.ECond**

Example: Lowest Common Ancestor shows 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 $(\texttt{t\_cond}, \texttt{e\_cond'}, \texttt{ses\_cond})^{/\!/\#\text{TE}}$;
- annotating `e_true` in `tenv` results in $(\texttt{t\_true}, \texttt{e\_true'}, \texttt{ses\_true})^{/\!/\#\text{TE}}$;
- annotating `e_false` in `tenv` results in $(\texttt{t\_false}, \texttt{e\_false'}, \texttt{ses\_false})$;
- obtaining the lowest common ancestor of `t_true` and `t_false` results in $\texttt{t}^{/\!/\#\text{TE}}$;

- `new_e` is the condition `e_cond'` with two options `e_true'` and `e_false'`, that is, E_Cond(`e_cond'`, `e_true'`, `e_false'`);
- define `ses` as the union of `ses_cond`, `ses_true`, and `ses_false`.

**Formally**

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e\_cond}) \xrightarrow{\text{type}} (\mathtt{t\_cond}, \mathtt{e\_cond}', \mathtt{ses\_cond}) \;/\!/\; \#\mathrm{TE} \\ \mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e\_true}) \xrightarrow{\text{type}} (\mathtt{t\_true}, \mathtt{e\_true}', \mathtt{ses\_true}) \;/\!/\; \#\mathrm{TE} \\ \mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e\_false}) \xrightarrow{\text{type}} (\mathtt{t\_false}, \mathtt{e\_false}', \mathtt{ses\_false}) \;/\!/\; \#\mathrm{TE} \\ \mathit{lowest\_common\_ancestor}(\mathtt{t\_true}, \mathtt{t\_false}) \xrightarrow{\text{type}} \mathtt{t} \;/\!/\; \#\mathrm{TE} \\ \mathtt{ses} := \mathtt{ses\_cond} \cup \mathtt{ses\_true} \cup \mathtt{ses\_false} \end{array}}{\begin{array}{c} \mathit{annotate\_expr}(\mathrm{E\_Cond}(\mathtt{e\_cond}, \mathtt{e\_true}, \mathtt{e\_false})) \xrightarrow{\text{type}} \\ (\mathtt{t}, \mathrm{E\_Cond}(\mathtt{e\_cond'}, \mathtt{e\_true'}, \mathtt{e\_false'}), \mathtt{ses}) \end{array}}$$

### 15.5.4 Semantics

**SemanticsRule.ECond**

**Example: Evaluation of Conditional Expressions**

In Listing 15.12, the expression `if FALSE then Return42() else 3` evaluates to the value 3.

Listing 15.12: 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 15.13, 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 15.13: 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, $\mathsf{E\_Cond}(\texttt{e\_cond}, \texttt{e1}, \texttt{e2})$;
- the evaluation of the conditional expression `e_cond` in `env` yields $\mathsf{Normal}(\texttt{m\_cond}, \texttt{env1})^{/\!/\texttt{\#T,\#DE}}$;
- `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 $\mathsf{Normal}((\texttt{v2}, \texttt{g2}), \texttt{new\_env})^{/\!/\texttt{\#T,\#DE}}$;
- `g` is the parallel composition of `g1` and `g2`.

### Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \texttt{e\_cond}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{m\_cond}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_cond} \overset{\mathsf{is}}{=} (\mathsf{Bool}(\texttt{b}), \texttt{g1}) \qquad \texttt{e'} := \mathsf{choice}(\texttt{b}, \texttt{e1}, \texttt{e2}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e'}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g2}), \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \xrightarrow{\mathsf{asl\_ctrl}} \texttt{g2} \end{array}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\mathsf{E\_Cond}(\texttt{e\_cond}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

### Comments

A conditional expression evaluates to its `then` expression if the condition expression evaluates to `TRUE`. If the condition expression evaluates to `FALSE` each `elsif` condition expression is evaluated sequentially until an `elsif` condition expression evaluates to `TRUE`; the conditional expression evaluates to the corresponding `elsif` expression. If no `elsif` expression evaluates to `TRUE` the conditional expression evaluates to the `else` expression.

## 15.6 Call Expressions

Listing 15.14: 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 ⟶ call

call ⟶ **ID** plist0(expr)
| **ID** "{" clist1(expr) "}"
| **ID** "{" clist1(expr) "}" plist0(expr)

### 15.6.2 Abstract Syntax

expr ⟶ E_Call(call)

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

**ASTRule.Call**

$$\frac{\mathit{build\_plist}[\mathit{build\_expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{arg\_asts}}{\mathit{build\_call}(\overbrace{\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{args} : \text{plist0}(\text{expr}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{id},\\ \text{params} & : & [\,],\\ \text{args} & : & \texttt{arg\_asts},\\ \text{call\_type} & : & \texttt{ST\_Function}\end{array}\right\}}^{\texttt{ast\_node}}}$$

$$\frac{\mathit{build\_list}[\mathit{build\_expr}](\texttt{params}) \xrightarrow{\text{ast}} \texttt{params\_ast}}{\mathit{build\_call}(\overbrace{\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{params} : \text{clist1}(\text{expr}), \texttt{"\}"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{id},\\ \text{params} & : & \texttt{params\_ast},\\ \text{args} & : & [\,],\\ \text{call\_type} & : & \texttt{ST\_Function}\end{array}\right\}}^{\texttt{ast\_node}}}$$

$$\frac{\begin{array}{c}\mathit{build\_plist}[\mathit{build\_expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{arg\_asts}\\ \mathit{build\_list}[\mathit{build\_expr}](\texttt{params}) \xrightarrow{\text{ast}} \texttt{params\_ast}\end{array}}{\mathit{build\_call}(\overbrace{\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{params} : \text{clist1}(\text{expr}), \texttt{"\}"}, \texttt{args} : \text{plist0}(\text{expr}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{id},\\ \text{params} & : & \texttt{params\_ast},\\ \text{args} & : & \texttt{arg\_asts},\\ \text{call\_type} & : & \texttt{ST\_Function}\end{array}\right\}}^{\texttt{ast\_node}}}$$

**ASTRule.SetCallType**

Above, `ST_Function` is inserted as a default call type for any parsed call. The helper function

$$\mathit{set\_call\_type}(\overbrace{\text{call}}^{\texttt{call}}, \overbrace{\text{sub\_program\_type}}^{\texttt{call\_type}}) \longrightarrow \overbrace{\text{call}}^{\texttt{call'}}$$

changes the call type of `call` to `call_type`.

$$\mathit{set\_call\_type}(\texttt{call}, \texttt{call\_type}) \longrightarrow \overbrace{\texttt{call}[\text{call\_type} \mapsto \texttt{call\_type}]}^{\texttt{call'}}$$

**ASTRule.ECall**

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{call})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Call}(\texttt{call})}^{\texttt{ast\_node}}$$

### 15.6.3 Typing

**TypingRule.ECall**

**Example: Typing Call Expressions**

Listing 15.14 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 $(\texttt{call'}, \langle \texttt{t} \rangle, \texttt{ses})^{/\!/ \#\text{TE}}$.
- `new_e` is the call using `call'`, that is, E_Call(`call'`).

**Formally**

$$\frac{\mathit{annotate\_call}(\texttt{call}) \xrightarrow{\text{type}} (\texttt{call'}, \langle \texttt{t} \rangle, \texttt{ses}) \;/\!/\; \#\text{TE}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_Call}(\texttt{call})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \overbrace{\text{E\_Call}(\texttt{call'})}^{\texttt{new\_e}}, \texttt{ses})}$$

### 15.6.4 Semantics

**SemanticsRule.ECall**

**Example: Evaluation of Call Expressions**

Listing 15.14 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`)$^{/\!/ \#\text{T}, \#\text{DE}}$;
- One of the following applies:
  - All of the following apply (SINGLE_RETURNED_VALUE):

  - `vms` consists of a single returned value $(\mathtt{v}, \mathtt{g})$, which goes into the output configuration Normal$((\mathtt{v}, \mathtt{g}), \mathtt{new_env})$.

* All of the following apply (MULTIPLE_RETURNED_VALUES):
  - `vms` consists of a list of returned value $(\mathtt{v}_i, \mathtt{g}_i)$, for $i = 1..k$;
  - `g` is the parallel composition of $\mathtt{g}_i$, for $i = 1..k$;
  - `v` is the native value vector of values $\mathtt{v}_i$, for $i = 1..k$;
  - the resulting configuration is Normal$((\mathtt{v}, \mathtt{g}), \mathtt{new_env})$.

**Formally**

SINGLE_RETURNED_VALUE

$$\frac{\mathit{eval\_call}(\mathtt{env}, \mathtt{call.name}, \mathtt{call.params}, \mathtt{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{vms}, \mathtt{new\_env}) \;/\!/\; \#\mathtt{T}, \#\mathtt{DE} \qquad \mathtt{vms} \overset{\text{is}}{=} [(\mathtt{v}, \mathtt{g})]}{\mathit{eval\_expr}(\mathtt{env}, \mathtt{E\_Call}(\mathtt{call})) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

MULTIPLE_RETURNED_VALUES

$$\frac{\mathit{eval\_call}(\mathtt{env}, \mathtt{call.name}, \mathtt{call.params}, \mathtt{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{vms}, \mathtt{new\_env}) \;/\!/\; \#\mathtt{T}, \#\mathtt{DE} \qquad \mathtt{vms} \overset{\text{is}}{=} [i = 1..k : (\mathtt{v}_i, \mathtt{g}_i)] \qquad \mathtt{g} := \mathtt{g}_1 \parallel \ldots \parallel \mathtt{g}_k \qquad \mathtt{v} := \mathtt{NV\_Vector}(\mathtt{v}_{1..k})}{\mathit{eval\_expr}(\mathtt{env}, \mathtt{E\_Call}(\mathtt{call})) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

## 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.

Listing 15.15: 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;
```

### 15.7.1 Syntax

`expr` ⟶ `expr slices`

### 15.7.2 Abstract Syntax

expr $\longrightarrow$ E_Slice(expr, slice$^*$)

**ASTRule.ESlice**

$$\frac{\textit{build\_expr}(\texttt{expr}) \xrightarrow{\text{ast}} \texttt{expr\_ast} \;/\!/\; \texttt{\#BE} \qquad \textit{build\_slices}(\texttt{slices}) \xrightarrow{\text{ast}} \texttt{slices\_ast} \;/\!/\; \texttt{\#BE}}{\textit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr} : \texttt{expr}, \texttt{slices} : \texttt{slices})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Slice}(\texttt{expr\_ast}, \texttt{slices\_ast})}^{\texttt{ast\_node}}}$$

### 15.7.3 Typing

**TypingRule.ESlice**

**Example: Typing of Slicing Expressions**

Listing 15.15 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'`).

**Prose**

All of the following apply:

- `e` denotes the slicing of expression `e'` by the slices `slices`, that is, E_Slice(`e'`, `slices`);
- annotating the expression `e'` in tenv yields (`t_e'`, `e''`, `ses1`)$^{/\!/\texttt{\#TE}}$;
- obtaining the structure of `t_e'` in tenv yields `struct_t_e'`$^{/\!/\texttt{\#TE}}$;
- `struct_t_e'` is either a bitvector or an integer;
- checking that `slices` is not empty yields TRUE$^{/\!/\texttt{TE_BS}}$;
- annotating `slices` in tenv yields (`slices'`, `ses2`)$^{/\!/\texttt{\#TE}}$;
- obtaining the width of `slices` in tenv via *slices_width* yields `w`$^{/\!/\texttt{\#TE}}$;
- `t` is the bitvector type of width `w`, that is, T_Bits(`w`, [ ]);
- define `new_e` as the slicing of expression `e''` by the slices `slices'`, that is, E_Slice(`e''`, `slices'`);
- define `ses` as the union of `ses1` and `ses2`.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t\_e'}, \texttt{e''}, \texttt{ses1}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \texttt{t\_e'}) \xrightarrow{\text{type}} \texttt{struct\_t\_e'} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{ast\_label}(\texttt{struct\_t\_e'}) \in \{\mathsf{T\_Int}, \mathsf{T\_Bits}\} \\ \mathit{check}(\texttt{slices} \neq [\,], \texttt{TE\_BS}) \xrightarrow{\text{type}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{annotate\_slices}(\mathsf{tenv}, \texttt{slices}) \xrightarrow{\text{type}} (\texttt{slices'}, \texttt{ses2}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{slices\_width}(\mathsf{tenv}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{w} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses1} \cup \texttt{ses2}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Slice}(\texttt{e'}, \texttt{slices})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{T\_Bits}(\texttt{w}, [\,])}^{\texttt{t}}, \overbrace{\mathsf{E\_Slice}(\texttt{e''}, \texttt{slices'})}^{\texttt{new\_e}}, \texttt{ses})}$$

**Comments**

The width of `slices` might be a symbolic expression if one of the widths references a `let` identifier with a non-compile-time-constant initializer expression.

**TypingRule.ESliceError**

**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**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t\_e'}, \texttt{e''}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \texttt{t\_e'}) \xrightarrow{\text{type}} \texttt{t'} \qquad \mathit{ast\_label}(\texttt{t'}) \notin \{\mathsf{T\_Int}, \mathsf{T\_Bits}\}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Slice}(\texttt{e'}, \texttt{slices})}^{\texttt{e}}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{TE\_BS})}$$

### 15.7.4 Semantics

**SemanticsRule.ESlice**

**Example: Evaluation of Slicing Expressions**

Listing 15.15 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]`.

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`)$^{/\!/\texttt{\#T,\#DE}}$;
- the evaluation of `slices` in `env` yields Normal(`m_positions`, `new_env`)$^{/\!/\texttt{\#T,\#DE}}$;
- `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$^{/\!/\texttt{\#DE}}$;
- `g` is the parallel composition of `g1` and `g2`.

**Formally**

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e\_bv}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_bv}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_bv} \overset{\text{is}}{=} (\texttt{v\_bv}, \texttt{g1}) \\ \mathit{eval\_slices}(\texttt{env1}, \texttt{slices}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_positions}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_positions} \overset{\text{is}}{=} (\texttt{positions}, \texttt{g2}) \\ \mathit{read\_from\_bitvector}(\texttt{v\_bv}, \texttt{positions}) \xrightarrow{\text{eval}} \texttt{v} \;/\!/\; \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Slice}(\texttt{e\_bv}, \texttt{slices})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

## 15.8 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 ⟶ expr "[[" expr "]]"

### 15.8.2 Abstract Syntax

expr ⟶ E_GetArray(expr, expr)

#### ASTRule.EGetArray

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \;/\!/\; \texttt{\#BE} \\ \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast} \;/\!/\; \texttt{\#BE}\end{array}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e1} : \texttt{expr}, \texttt{"[["}, \texttt{e2} : \texttt{expr}, \texttt{"]]"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_GetArray}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

#### 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 13.22 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 $(\texttt{t\_base}, \texttt{e\_base'}, \texttt{ses\_base})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_base` in `tenv` yields $\texttt{t\_anon\_base}^{/\!/\texttt{\#TE}}$;
- checking whether `t_anon_base` is an array type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- view `t_anon_base` as the array type with size expression `size` and element type `t_elem`, that is, 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**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e\_base}) \xrightarrow{\text{type}} (\mathtt{t\_base}, \mathtt{e\_base'}, \mathtt{ses\_base}) \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base}) \xrightarrow{\text{type}} \mathtt{t\_anon\_base} \;/\!/\; \mathtt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_anon\_base}) = \mathsf{T\_Array}, \mathtt{ExpectedArrayType}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{t\_anon\_base} \overset{\text{is}}{=} \mathsf{T\_Array}(\mathtt{size}, \mathtt{t\_elem}) \\ \mathit{annotate\_get\_array}(\mathsf{tenv}, (\mathtt{size}, \mathtt{t\_elem}), (\mathtt{e\_base'}, \mathtt{ses\_base}, \mathtt{e\_index})) \xrightarrow{\text{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses})\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetArray}(\mathtt{e\_base}, \mathtt{e\_index})}^{\mathtt{e}}) \xrightarrow{\text{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses})}$$

**TypingRule.AnnotateGetArray**

The helper function

$$\mathit{annotate\_get\_array}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, (\overbrace{\mathsf{expr}}^{\mathtt{size}} \times \overbrace{\mathsf{ty}}^{\mathtt{t\_elem}}), (\overbrace{\mathsf{expr}}^{\mathtt{e\_base}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses\_base}} \times \overbrace{\mathsf{expr}}^{\mathtt{e\_index}})) \longrightarrow (\overbrace{\mathsf{ty}}^{\mathtt{t}} \times \overbrace{\mathsf{expr}}^{\mathtt{new\_e}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}})$$

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 (`t_index'`, `e_index'`, `ses_index`)$^{/\!/\mathtt{\#TE}}$;
- applying *type_of_array_length* to `size`, to obtain the type of the array length, yields `wanted_t_index`;
- checking that `t_index'` type-satisfies `wanted_t_index` in `tenv` yields `TRUE`$^{/\!/\mathtt{\#TE}}$;
- 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**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e\_index}) \xrightarrow{\text{type}} (\texttt{t\_index'}, \texttt{e\_index'}, \texttt{ses\_index}) \;/\!/\; \texttt{\#TE} \\ \mathit{type\_of\_array\_length}(\texttt{size}) \xrightarrow{\text{type}} \texttt{wanted\_t\_index} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_index'}, \texttt{wanted\_t\_index}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_index} \cup \texttt{ses\_base} \\ \texttt{new\_e} := \begin{cases} \texttt{E\_GetArray}(\texttt{e\_base}, \texttt{e\_index'}) & \text{if } \mathit{ast\_label}(\texttt{size}) = \texttt{ArrayLength\_Expr} \\ \texttt{E\_GetEnumArray}(\texttt{e\_base}, \texttt{e\_index'}) & \text{if } \mathit{ast\_label}(\texttt{size}) = \texttt{ArrayLength\_Enum} \end{cases}\end{array}}{\mathit{annotate\_get\_array}(\texttt{tenv}, (\texttt{size}, \texttt{t\_elem}), (\texttt{e\_base}, \texttt{ses\_base}, \texttt{e\_index})) \xrightarrow{\text{type}} (\overbrace{\texttt{t\_elem}}^{\texttt{t}}, \texttt{new\_e}, \texttt{ses})}$$

**SemanticsRule.EGetArray**

### Example: Evaluation of Array Reading Expressions

In Listing 15.16, 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 15.16: 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 15.17, 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 15.17: 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` is `v`;
- `g` is the parallel composition of `g1` and `g2`.

### Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \texttt{e\_array}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_array}, \texttt{env1}) \;/\!/\; \#T, \#DE \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_index}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_index}, \texttt{new\_env}) \;/\!/\; \#T, \#DE \\ \texttt{m\_array} \overset{\text{is}}{=} (\texttt{v\_array}, \texttt{g1}) \qquad \texttt{m\_index} \overset{\text{is}}{=} (\texttt{index}, \texttt{g2}) \\ \texttt{index} \overset{\text{is}}{=} \text{Int}(\texttt{i}) \qquad \mathit{get\_index}(\texttt{i}, \texttt{v\_array}) \xrightarrow{\text{eval}} \texttt{v} \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2} \end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_GetArray}(\texttt{e\_array}, \texttt{e\_index})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

### SemanticsRule.EGetEnumArray

### Example: Evaluation of Reading from an Enumeration-indexed Array

In Listing 15.18, the enumeration-typed array `Arr` is accessed for reading and writing with indices taken from the enumeration type `Enum`.

Listing 15.18: 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 `l`;
- accessing the field `l` of `v_array`, which is a native record value, yields `v`;
- `g` is the parallel composition of `g1` and `g2`.

**Formally**

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \texttt{e\_array}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_array}, \texttt{env1}) \;/\!/\; \#T, \#DE \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_index}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_index}, \texttt{new\_env}) \;/\!/\; \#T, \#DE \\ \texttt{m\_array} \overset{\text{is}}{=} (\texttt{v\_array}, \texttt{g1}) \qquad \texttt{m\_index} \overset{\text{is}}{=} (\texttt{index}, \texttt{g2}) \\ \texttt{index} \overset{\text{is}}{=} \text{Label}(\texttt{l}) \qquad \mathit{get\_field}(\texttt{l}, \texttt{v\_array}) \xrightarrow{\text{eval}} \texttt{v} \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2} \end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_GetEnumArray}(\texttt{e\_array}, \texttt{e\_index})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

## 15.9 Field Reading Expressions

### 15.9.1 Syntax

expr ⟶ expr "." **ID**

### 15.9.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_GetField}(\overset{\text{record}}{\text{expr}}, \overset{\text{field name}}{\text{identifier}})$$

**ASTRule.EGetField**

$$\frac{\mathit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \;/\!/\; \#BE}{\mathit{build\_expr}(\overbrace{\text{expr}(\texttt{e} : \text{expr}, \texttt{"."}, \textbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_GetField}(\texttt{e\_ast}, \texttt{id})}^{\texttt{ast\_node}}}$$

### 15.9.3 Typing

#### TypingRule.EGetRecordField

#### Example: Typing Record Field Expressions

Listing 15.19 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 15.19: 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 $(\texttt{t\_e1}, \texttt{e2}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields $\texttt{t\_e2}^{/\!/\texttt{\#TE}}$;
- `t_e2` is a 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

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\text{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\text{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e2} = L(\texttt{fields}) \\ L \in \{\text{T\_Record}, \text{T\_Exception}\} \qquad \mathit{assoc\_opt}(\texttt{fields}, \texttt{field\_name}) \xrightarrow{\text{type}} \langle \texttt{t} \rangle \end{array}}{\begin{array}{r} \mathit{annotate\_expr}(\text{tenv}, \text{E\_GetField}(\texttt{e1}, \texttt{field\_name})) \xrightarrow{\text{type}} \\ (\texttt{t}, \text{E\_GetField}(\texttt{e2}, \texttt{field\_name}), \overbrace{\texttt{ses1}}^{\texttt{ses}}) \end{array}}$$

### TypingRule.EGetBadRecordField

#### Example: Ill-typed Record Field Expressions

Listing 15.20 shows an ill-typed field expression.

Listing 15.20: 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`)$^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields `t_e2`$^{/\!/\texttt{\#TE}}$;
- `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

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e2} = L(\texttt{fields}) \end{array} \qquad L \in \{\text{T\_Record}, \text{T\_Exception}\} \qquad \mathit{assoc\_opt}(\texttt{fields}, \texttt{field\_name}) \xrightarrow{\text{type}} \text{None}}{\mathit{annotate\_expr}(\texttt{tenv}, \text{E\_GetField}(\texttt{e1}, \texttt{field\_name})) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_BF})}$$

### TypingRule.EGetBadBitField

#### Example: Ill-typed Bitfield Expressions

Listing 15.21 shows an ill-typed bitfield expression.

Listing 15.21: 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`)$^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields `t_e2`$^{/\!/\texttt{\#TE}}$;
- `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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} (\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\text{type}} \mathtt{t\_e2} \;/\!/\; \texttt{\#TE} \\ \mathtt{t\_e2} = \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field\_name}) \xrightarrow{\text{type}} \mathtt{None}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \mathtt{field\_name})}^{\mathtt{e}}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_BF})}$$

### 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`, `field_name`);
- annotating the expression `e1` in `tenv` yields (`t_e1`, `e2`, `ses1`)$^{/\!/\texttt{\#TE}}$;

- obtaining the underlying type of `t_e1` yields `t_e2`$^{/\!/\texttt{\#TE}}$;
- `e2` is a variable expression for `base`, that is, E_Var(`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(`base`, [`field_name`]);
- define `ses` as `ses1`.

**Formally**

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}, \texttt{ses1}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{e2} = \mathsf{E\_Var}(\texttt{base}) \\ \texttt{t\_e2} = \mathsf{T\_Collection}(\texttt{fields}) \qquad \mathit{assoc\_opt}(\texttt{fields}, \texttt{field\_name}) \xrightarrow{\text{type}} \langle \texttt{t} \rangle \end{array}}{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \mathsf{E\_GetField}(\texttt{e1}, \texttt{field\_name})) \xrightarrow{\text{type}} \\ (\texttt{t}, \mathsf{E\_GetCollectionFields}(\texttt{base}, [\texttt{field\_name}]), \overbrace{\texttt{ses1}}^{\text{ses}}) \end{array}}$$

### TypingRule.EGetBitField

### Example: Well-typed Bitfield Expressions

Listing 15.22 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 15.22: 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`, `new_e`, `ses`)$^{/\!/\#TE}$.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} (\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1}) \;/\!/\; \#\mathrm{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\text{type}} \mathtt{t\_e2} \;/\!/\; \#\mathrm{TE} \\ \mathtt{t\_e2} = \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field\_name}) \xrightarrow{\text{type}} \langle \mathsf{BitField\_Simple}(\_, \mathtt{slices}) \rangle \\ \mathtt{e3} := \mathsf{E\_Slice}(\mathtt{e2}, \mathtt{slices}) \qquad \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e3}) \xrightarrow{\text{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses}) \;/\!/\; \#\mathrm{TE}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \mathtt{field\_name})}^{\mathtt{e}}) \xrightarrow{\text{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses})}$$

**TypingRule.EGetBitFieldNested**

**Example: Nested Bitfield Expressions**

Listing 15.22 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`, `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 nested bitfields `bitfields'`, that is, BitField_Nested(_, `slices`, `bitfields'`);
- `e3` denotes the slicing of the expression `e2` by the slices `slices`, that is, E_Slice(`e2`, `slices`);
- annotating `e3` in `tenv` yields (`t_e4`, `new_e`, `ses_new`)$^{\#TE}$;
- `t_e4` is a bitvector type with length expression `width`, that is, T_Bits(`width`, _);
- define `t` as a bitvector type with length expression `width` and bitfields `bitfields'`;
- define `ses` as `ses_new`.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}, \texttt{ses1}) \;/\!/\; \#TE \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;/\!/\; \#TE \\ \texttt{t\_e2} = \text{T\_Bits}(\_, \texttt{bitfields}) \\ \mathit{find\_bitfield\_opt}(\texttt{bitfields}, \texttt{field\_name}) \xrightarrow{\text{type}} \\ \langle \text{BitField\_Nested}(\_, \texttt{slices}, \texttt{bitfields'}) \rangle \\ \texttt{e3} := \text{E\_Slice}(\texttt{e2}, \texttt{slices}) \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e3}) \xrightarrow{\text{type}} (\texttt{t\_e4}, \texttt{new\_e}, \texttt{ses\_new}) \;/\!/\; \#TE \\ \texttt{t\_e4} \overset{\text{is}}{=} \text{T\_Bits}(\texttt{width}, \_) \qquad \texttt{t} := \text{T\_Bits}(\texttt{width}, \texttt{bitfields'}) \end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_GetField}(\texttt{e1}, \texttt{field\_name})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \overbrace{\texttt{ses}}^{\texttt{ses\_new}})}$$

**TypingRule.EGetBitFieldTyped**

**Example: Typed Bitfield Expressions**

Listing 15.22 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**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} (\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1}) \;//\; \#TE \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\text{type}} \mathtt{t\_e2} \;//\; \#TE \\ \mathtt{t\_e2} = \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field\_name}) \xrightarrow{\text{type}} \langle \mathsf{BitField\_Type}(\_, \mathtt{slices}, \mathtt{t}) \rangle \\ \mathtt{e3} := \mathsf{E\_Slice}(\mathtt{e2}, \mathtt{slices}) \\ \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e3}) \xrightarrow{\text{type}} (\mathtt{t\_e4}, \mathtt{new\_e}, \mathtt{ses\_new}) \;//\; \#TE \\ \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e4}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;//\; \#TE \end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \mathtt{field\_name})}^{\mathtt{e}}) \xrightarrow{\text{type}} (\mathtt{t}, \mathtt{new\_e}, \overbrace{\mathtt{ses}}^{\mathtt{ses\_new}})}$$

**TypingRule.EGetTupleItem**

**Example: Typing of a Tuple Item Expression**

In Listing 15.25, 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, ses1)$^{//\#TE}$;
- obtaining the underlying type of t_e1 yields t_e2$^{//\#TE}$;
- 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 TRUE$^{//\#TE}$;
- t is the type at position index of tys;

- `new_e` is the expression for obtaining the item at index `index` from the expression `e2`, that is, E_GetItem(`e2`, `index`);
- define `ses` as `ses1`.

**Formally**

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e2} = \texttt{T\_Tuple}(\texttt{tys}) \qquad \texttt{field\_name} = \texttt{item}+\mathit{num} \\ \mathit{num} \in \texttt{Lang}(\texttt{<int\_lit>}) \qquad \mathit{dec\_to\_lit}(\mathit{num}) = \textbf{INT\_LIT}(\texttt{index}) \\ \mathit{check}(0 \le \texttt{index} \le |\texttt{tys}|, \texttt{TE\_BTI}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{t} := \texttt{tys}[\texttt{index}] \qquad \texttt{new\_e} := \texttt{E\_GetItem}(\texttt{e2}, \texttt{index}) \end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\texttt{E\_GetField}(\texttt{e1}, \texttt{field\_name})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \overbrace{\texttt{ses1}}^{\texttt{ses}})}$$

**TypingRule.EGetBadField**

**Example: An Ill-typed Field Expression**

Listing 15.23 shows an example of an ill-typed field expression `a.f`.

Listing 15.23: 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 $(\texttt{t\_e1}, \texttt{e2}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields $\texttt{t\_e2}^{/\!/\texttt{\#TE}}$;
- `t_e2` is neither one of the following types: record, exception, bitvector, or tuple;
- the result is an error indicating that the type of `e1` is inappropriate for accessing the field `field_name`.

**Formally**

$$\frac{\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} (\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1}) \;/\!/\; \#\mathtt{TE} \quad \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\mathsf{type}} \mathtt{t\_e2} \;/\!/\; \#\mathtt{TE} \quad \mathit{ast\_label}(\mathtt{t\_e2}) \notin \{\mathsf{T\_Record}, \mathsf{T\_Exception}, \mathsf{T\_Bits}, \mathsf{T\_Tuple}\}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \mathtt{field\_name})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UT})}$$

### 15.9.4 Semantics

**SemanticsRule.EGetField**

**Example: Evaluation of a Field Read Expression**

In Listing 15.24, the expression `my_record.a` evaluates to the value 3.

Listing 15.24: 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, $\mathsf{E\_GetField}(\mathsf{E\_Record}, \mathtt{field\_name})$;
- the evaluation of $\mathsf{E\_Record}$ in `env` is $\mathsf{Normal}((\mathtt{v\_record}, \mathtt{g}), \mathtt{new\_env})^{/\!/\#\mathtt{T},\#\mathtt{DE}}$;
- `v` is the value mapped by `field_name` in the native record `v_record`.

**Formally**

$$\frac{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_Record}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v\_record}, \mathtt{g}), \mathtt{new\_env}) \;/\!/\; \#\mathtt{T}, \#\mathtt{DE} \quad \mathit{get\_field}(\mathtt{field\_name}, \mathtt{v\_record}) \xrightarrow{\mathsf{eval}} \mathtt{v}}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_GetField}(\mathsf{E\_Record}, \mathtt{field\_name})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

**SemanticsRule.EGetItem**

**Example: Evaluation of a Tuple Item Expression**

In Listing 15.25, the expression `t.item0` evaluates to the value `1` and the expression `t.item1` evaluates to the value `2`.

Listing 15.25: Evaluating a tuple item access expression

```
func main() => integer
begin
    var t = (1, 2);
    assert t.item0 + t.item1 == 3;

    // The following statement in comment is illegal: item01 is treated
    // by the type system as a field.
    // let - = t.item01;
    return 0;
end;
```

**Prose**

All of the following apply:

- `e` is 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**

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e\_tuple}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_tuple}, \texttt{g}), \texttt{new\_env}) \;//\; \#T, \#DE \qquad \mathit{get\_index}(\texttt{v\_tuple}, \texttt{index}) \xrightarrow{\text{eval}} \texttt{v}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_GetItem}(\texttt{e\_tuple}, \texttt{index})}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

## 15.10 Multi-field Reading Expressions

### 15.10.1 Syntax

expr ⟶ expr "." "[" clist1(**ID**) "]"

### 15.10.2 Abstract Syntax

expr ⟶ E_GetFields( $\overbrace{\text{expr}}^{\text{record}}$ , $\overbrace{\text{identifier}^*}^{\text{field names}}$ )

**ASTRule.EGetFields**

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{id\_asts} \qquad \mathit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \;\; /\!/ \;\; \#\text{BE}}{\mathit{build\_expr}(\overbrace{\text{expr}(\texttt{e} : \text{expr}, \texttt{"."}, \texttt{"["}, \texttt{ids} : \text{clist1}(\textbf{ID}), \texttt{"]"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_GetFields}(\texttt{e\_ast}, \texttt{id\_asts})}^{\texttt{ast\_node}}}$$

### 15.10.3 Typing

**TypingRule.EGetFields**

**Example: Typing Multi-field Expressions**

Listing 15.26 shows examples of well-typed multi-field expressions.

Listing 15.26: 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 $\texttt{slices}_{\texttt{name}}$[//#TE];
  - define `e_slice` as the slicing expression for `e_base` and lists of slices $\texttt{slices}_{\texttt{name}}$, for each `name` in `fields`;
  - annotating the expression `e_slice` in the static environment `tenv` yields $(\texttt{t}, \texttt{new\_e}, \texttt{ses})$[//#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 $\texttt{e\_width}_{\texttt{f}}$[//#TE];
  - applying *width_plus* to the list of expressions $\texttt{e\_width}_{\texttt{f}}$, 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 $\texttt{e\_width}_{\texttt{f}}$[//#TE];
  - applying *width_plus* to the list of expressions $\texttt{e\_width}_{\texttt{f}}$, 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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e\_base}) \xrightarrow{\mathsf{type}} (\mathtt{t\_base\_annot}, \mathtt{e\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \#\mathsf{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base\_annot}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \;/\!/\; \#\mathsf{TE} \\ \mathtt{name} \in \mathtt{fields} : \mathit{find\_bitfields\_slices}(\mathtt{name}, \mathtt{bitfields}) \xrightarrow{\mathsf{type}} \mathtt{slices}_{\mathtt{name}} \;/\!/\; \#\mathsf{TE} \\ \mathtt{e\_slice} := \mathsf{E\_Slice}(\mathtt{e\_base}, [\mathtt{name} \in \mathtt{fields} : \mathtt{slices}_{\mathtt{name}}]) \\ \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e\_slice}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses}) \;/\!/\; \#\mathsf{TE}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e\_base}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses})}$$

RECORD

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e\_base}) \xrightarrow{\mathsf{type}} (\mathtt{t\_base\_annot}, \mathtt{e\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \#\mathsf{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base\_annot}) \xrightarrow{\mathsf{type}} \mathsf{T\_Record}(\mathtt{base\_fields}) \;/\!/\; \#\mathsf{TE} \\ \mathtt{f} \in \mathtt{base\_fields} : \mathit{get\_bitfield\_width}(\mathsf{tenv}, \mathtt{f}, \mathtt{tfields}) \xrightarrow{\mathsf{type}} \mathtt{e\_width}_{\mathtt{f}} \;/\!/\; \#\mathsf{TE} \\ \mathit{width\_plus}(\mathsf{tenv}, [\mathtt{f} \in \mathtt{base\_fields} : \mathtt{e\_width}_{\mathtt{f}}]) \xrightarrow{\mathsf{type}} \mathtt{e\_slice\_width} \;/\!/\; \#\mathsf{TE}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e\_base}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{T\_Bits}(\mathtt{e\_slice\_width}, [\,])}^{\mathtt{t}}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e\_base\_annot}, \mathtt{fields})}^{\mathtt{new\_e}}, \overbrace{\mathtt{ses\_base}}^{\mathtt{ses}})}$$

COLLECTION

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e\_base}) \xrightarrow{\mathsf{type}} (\mathtt{t\_base\_annot}, \mathtt{e\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \#\mathsf{TE} \\ \mathtt{e\_base\_annot} = \mathsf{E\_Var}(\mathtt{base}) \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base\_annot}) \xrightarrow{\mathsf{type}} \mathsf{T\_Record}(\mathtt{base\_fields}) \;/\!/\; \#\mathsf{TE} \\ \mathtt{f} \in \mathtt{base\_fields} : \mathit{get\_bitfield\_width}(\mathsf{tenv}, \mathtt{f}, \mathtt{tfields}) \xrightarrow{\mathsf{type}} \mathtt{e\_width}_{\mathtt{f}} \;/\!/\; \#\mathsf{TE} \\ \mathit{width\_plus}(\mathsf{tenv}, [\mathtt{f} \in \mathtt{base\_fields} : \mathtt{e\_width}_{\mathtt{f}}]) \xrightarrow{\mathsf{type}} \mathtt{e\_slice\_width} \;/\!/\; \#\mathsf{TE}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e\_base}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{T\_Bits}(\mathtt{e\_slice\_width}, [\,])}^{\mathtt{t}}, \overbrace{\mathsf{E\_GetCollectionFields}(\mathtt{base}, \mathtt{fields})}^{\mathtt{new\_e}}, \overbrace{\mathtt{ses\_base}}^{\mathtt{ses}})}$$

ERROR

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} (\mathtt{t\_base\_annot}, \mathtt{e\_base\_annot}, \_) \;/\!/\; \#\mathsf{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base\_annot}) \xrightarrow{\mathsf{type}} \mathtt{t\_base\_annot\_anon} \;/\!/\; \#\mathsf{TE} \\ \mathit{ast\_label}(\mathtt{t\_base\_annot\_anon}) \notin \{\mathsf{T\_Bits}, \mathsf{T\_Record}\}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e1}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UT})}$$

### TypingRule.FindBitFieldsSlices

The helper function

$$\mathit{find\_bitfields\_slices}(\overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{bitfield}^*}^{\texttt{bitfields}}) \longrightarrow \overbrace{\mathsf{slice}^*}^{\texttt{slices}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 15.22, 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 $\mathit{vbitfieldsone}$ yields $\texttt{slices}^{/\!/\texttt{\#TE}}$.
- 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

FOUND

$$\frac{\mathit{bitfield\_get\_name}(\texttt{field}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{bitfield\_get\_slices}(\texttt{field}) \xrightarrow{\text{type}} \texttt{slices}}{\mathit{find\_bitfields\_slices}(\texttt{name}, \overbrace{[\texttt{field}] + \texttt{bitfields1}}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \texttt{slices}}$$

TAIL

$$\frac{\mathit{bitfield\_get\_name}(\texttt{field}) \xrightarrow{\text{type}} \texttt{name'} \qquad \texttt{name'} \neq \texttt{name} \qquad \mathit{find\_bitfields\_slices}(\texttt{name}, \texttt{bitfields1}) \xrightarrow{\text{type}} \texttt{slices} \;/\!/\; \texttt{\#TE}}{\mathit{find\_bitfields\_slices}(\texttt{name}, \overbrace{[\texttt{field}] + \texttt{bitfields1}}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \texttt{slices}}$$

EMPTY

$$\mathit{find\_bitfields\_slices}(\texttt{name}, \overbrace{[\,]}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{TE\_BF})$$

### TypingRule.GetBitfieldWidth

The helper function

$$\mathit{get\_bitfield\_width}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{field}^*}^{\texttt{tfields}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{e\_width}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

returns the expression `e_width` that describes the width of the bitfield named `name` in the list of fields `tfields`. Otherwise, the result is a type error.

### Example: Obtaining the Width of a Bitfield

In Listing 15.26, 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 $\texttt{e\_width}^{/\!/\texttt{\#TE}}$.
- 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`.

**Formally**

OKAY

$$\frac{\mathit{assoc\_opt}(\texttt{name}, \texttt{tfields}) \xrightarrow{\text{type}} \langle \texttt{t} \rangle \quad \mathit{get\_bitvector\_width}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{e\_width} \;/\!/\; \texttt{\#TE}}{\mathit{get\_bitfield\_width}(\texttt{tenv}, \texttt{name}, \texttt{tfields}) \xrightarrow{\text{type}} \texttt{e\_width}}$$

ERROR

$$\frac{\mathit{assoc\_opt}(\texttt{name}, \texttt{tfields}) \xrightarrow{\text{type}} \texttt{None}}{\mathit{get\_bitfield\_width}(\texttt{tenv}, \texttt{name}, \texttt{tfields}) \xrightarrow{\text{type}} \texttt{TypeError}(\texttt{TE\_BF})}$$

**TypingRule.WidthPlus**

The helper function

$$\mathit{width\_plus}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{expr}^*}^{\texttt{exprs}}) \xrightarrow{\text{type}} \overbrace{\texttt{expr}}^{\texttt{e\_width}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

generates the expression `e_width`, which represents the summation of all expressions in the list `exprs`, normalized in the static environment `tenv`. $/\!/_{\texttt{\#TE}}$

**Example: Summing Widths**

Summing the list of expression `2, 3`, yields the expression `2+3`, (in AST terms,

$$\overbrace{\texttt{E\_Literal(L\_Int)}\ \texttt{2}\quad \texttt{PLUS}\quad \texttt{E\_Literal(L\_Int)}\ \texttt{2}}^{\texttt{E\_Binop}}).$$

**Prose**

One of the following applies:

- All of the following apply (EMPTY):
  - `exprs` is an empty list;
  - `e_width` is the literal expression for 0.
- All of the following apply (NON_EMPTY):
  - `exprs` is the list with `head e` and `tail exprs1`;
  - applying *width_plus* to `exprs1` yields `e_width1`;
  - applying *normalize* to the binary operation for `PLUS` and `e` and `e_width1` yields `e_width`$/\!/_{\texttt{\#TE}}$.

**Formally**

$$\text{EMPTY}\quad \frac{}{\mathit{width\_plus}(\mathsf{tenv}, \overbrace{[\,]}^{\mathtt{exprs}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal(L\_Int)}}^{\mathtt{e\_width}}\ \underset{}{0}}$$

$$\text{NON\_EMPTY}\quad \frac{\mathit{width\_plus}(\mathtt{exprs1}) \xrightarrow{\mathsf{type}} \mathtt{e\_width1} \qquad \mathit{normalize}(\mathsf{tenv}, \mathsf{E\_Binop}(\mathtt{PLUS}, \mathtt{e}, \mathtt{e\_width1})) \xrightarrow{\mathsf{type}} \mathtt{e\_width}\ /\!/\ \mathtt{\#TE}}{\mathit{width\_plus}(\mathsf{tenv}, \overbrace{[\mathtt{e}] + \mathtt{exprs1}}^{\mathtt{exprs}}) \xrightarrow{\mathsf{type}} \mathtt{e\_width}}$$

### 15.10.4 Semantics

**SemanticsRule.EGetfields**

**Example: Evaluating Multi-field Expressions**

In Listing 15.26, evaluating the multi-field expression `bits_var.[bits3_2, bit1, bit0, info_and_bits]` yields the bitvector value `'00101010'`, and evaluating the multi-field expression `record_var.[bits3_2, bit1, bit0, info]` yields the bitvector value `'001010'`.

**Prose**

All of the following apply:

- `e` is the multi-field access expression for the expression E_Record and list of field names `field_names`;
- evaluating the expression E_Record in `env` yields $((\mathtt{v\_record}, \mathtt{g}), \mathtt{new\_env})^{/\!/\mathtt{\#T,\#DE}}$;
- obtaining the value associated with the field `field_name` in `v`, for each `field_name` in `field_names`, yields $\mathtt{v}_{\mathtt{field\_name}}$;
- define `v` as the concatenation of $\mathtt{v}_{\mathtt{field\_name}}$, for each `field_name` in `field_names`.

**Formally**

$$\frac{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_Record}) \xrightarrow{\mathsf{eval}} ((\mathtt{v\_record}, \mathtt{g}), \mathtt{new\_env})\ /\!/\ \mathtt{\#T}, \mathtt{\#DE} \qquad \mathtt{field\_name} \in \mathtt{field\_names} : \mathit{get\_field}(\mathtt{field\_name}, \mathtt{v}) \xrightarrow{\mathsf{eval}} \mathtt{v}_{\mathtt{field\_name}} \qquad \mathit{concat\_bitvectors}([\mathtt{field\_name} \in \mathtt{field\_names} : \mathtt{v}_{\mathtt{field\_name}}]) \xrightarrow{\mathsf{type}} \mathtt{v}}{\mathit{eval\_expr}(\mathtt{env}, \overbrace{\mathsf{E\_GetFields}(\mathsf{E\_Record}, \mathtt{field\_names})}^{\mathtt{e}}) \xrightarrow{\mathsf{eval}} ((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

**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 $\mathtt{v}_{\mathtt{field\_name}}$;
- define `v` as the concatenation of $\mathtt{v}_{\mathtt{field\_name}}$, for each `field_name` in `field_names`.
- `g` is the graph containing a Read Effect for each field `field_name` in `field_names`.

**Formally**

$$\frac{\begin{array}{c}\mathtt{env} \overset{\text{is}}{=} (\_, \mathsf{denv}) \qquad \mathtt{base} \in \mathrm{dom}(G^{\mathsf{denv}}.\mathsf{storage}) \qquad \mathtt{v} := G^{\mathsf{denv}}.\mathsf{storage}(\mathtt{base}) \\ \mathtt{field\_name} \in \mathtt{field\_names} : \mathit{get\_field}(\mathtt{field\_name}, \mathtt{v}) \xrightarrow{\mathsf{eval}} \mathtt{v}_{\mathtt{field\_name}} \\ \mathtt{g} := \{\mathtt{field\_name} \in \mathtt{field\_names} : \mathsf{ReadEffect}(\mathtt{base} + \texttt{"."} + \mathtt{field\_name})\} \\ \mathit{concat\_bitvectors}([\mathtt{field\_name} \in \mathtt{field\_names} : \mathtt{v}_{\mathtt{field\_name}}]) \xrightarrow{\mathsf{eval}} \mathtt{v}\end{array}}{\mathit{eval\_expr}(\mathtt{env}, \overbrace{\mathsf{E\_GetCollectionFields}(\mathtt{base}, \mathtt{field\_names})}^{\mathtt{e}}) \xrightarrow{\mathsf{eval}} ((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

## 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 subprogram. If the condition evaluates to `FALSE` it is a dynamic error.

Listing 15.27: 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

expr ⟶ expr "as" ty
| expr "as" constraint_kind

### 15.11.2 Abstract Syntax

expr ⟶ E_ATC (expr, ty )  (Type assertion; asserted type)

**ASTRule.ATC**

TYPE

$$\frac{\begin{array}{c}build\_expr(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \;\; /\!/ \;\; \texttt{\#BE} \\ build\_ty(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast} \;\; /\!/ \;\; \texttt{\#BE}\end{array}}{build\_expr(\overbrace{\texttt{expr}(\texttt{e}:\texttt{expr},\texttt{"as"},\texttt{t}:\texttt{ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_ATC}(\texttt{e\_ast},\texttt{t\_ast})}^{\texttt{ast\_node}}}$$

INT_CONSTRAINTS

$$\frac{\mathit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \;/\!/\; \texttt{\#BE} \qquad \mathit{build\_constraint\_kind}(\texttt{ics}) \xrightarrow{\text{ast}} \texttt{ics\_ast} \;/\!/\; \texttt{\#BE}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e} : \texttt{expr}, \texttt{"as"}, \texttt{ics} : \texttt{constraint\_kind})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_ATC}(\texttt{e\_ast}, \texttt{T\_Int}(\texttt{ics\_ast}))}^{\texttt{ast\_node}}}$$

### 15.11.3 Typing

#### TypingRule.ATC

#### Example: Well-typed Asserting Type Conversion Expressions

Listing 15.27 shows examples of ATC expressions and the corresponding annotated expressions in comments.

#### 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`)$^{/\!/\texttt{\#TE}}$;
- obtaining the structure of `t` in `tenv` yields `t_struct`$^{/\!/\texttt{\#TE}}$;
- annotating the type `ty` in `tenv` yields (`ty'`, `ses_ty`)$^{/\!/\texttt{\#TE}}$;
- obtaining the structure of `ty'` in `tenv` yields `ty_struct`$^{/\!/\texttt{\#TE}}$;
- applying *check_atc* to `t_struct` and `ty_struct` in `tenv` to check whether the type assertion will always fail yields TRUE$^{/\!/\texttt{\#TE}}$;
- define `ses'` as the union of `ses_ty`, `ses_e`, and the singleton set for PerformsAssertions;
- checking whether `t_struct` subtype-satisfies `ty_struct` in `tenv` yields `allways_succeeds`$^{/\!/\texttt{\#TE}}$ (if `allways_succeeds` holds then the type assertion will always succeed dynamically, and therefore can be omitted);
- `new_e` is `e''` if `allways_succeeds` is TRUE and E_ATC(`ty'`, `e''`) otherwise;
- `ses` is `ses_e` if `allways_succeeds` is TRUE and `ses` otherwise;
- `t` is `ty'`.

**Formally**

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e'}) \xrightarrow{\text{type}} (\mathtt{t}, \mathtt{e''}, \mathtt{ses\_e}) \ /\!/ \ \#\mathsf{TE} \\ \mathit{get\_structure}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \ /\!/ \ \#\mathsf{TE} \\ \mathit{annotate\_type}(\mathtt{tenv}, \mathtt{ty}) \xrightarrow{\text{type}} (\mathtt{ty'}, \mathtt{ses\_ty}) \ /\!/ \ \#\mathsf{TE} \\ \mathit{get\_structure}(\mathtt{tenv}, \mathtt{ty'}) \xrightarrow{\text{type}} \mathtt{ty\_struct} \ /\!/ \ \#\mathsf{TE} \\ \mathit{check\_atc}(\mathtt{tenv}, \mathtt{t\_struct}, \mathtt{ty\_struct}) \xrightarrow{\text{type}} \mathsf{TRUE} \ /\!/ \ \#\mathsf{TE} \\ \mathtt{ses'} := \mathtt{ses\_ty} \cup \mathtt{ses\_e} \cup \{\mathsf{PerformsAssertions}\} \\ \mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t\_struct}, \mathtt{ty\_struct}) \xrightarrow{\text{type}} \mathtt{allways\_succeeds} \ /\!/ \ \#\mathsf{TE} \\ (\mathtt{new\_e}, \mathtt{ses}) := \mathsf{choice}(\mathtt{allways\_succeeds}, (\mathtt{e''}, \mathtt{ses\_e}), (\mathsf{E\_ATC}(\mathtt{e''}, \mathtt{ty'}), \mathtt{ses'})) \end{array}}{\mathit{annotate\_expr}(\mathtt{tenv}, \overbrace{\mathsf{E\_ATC}(\mathtt{e'}, \mathtt{ty})}^{\mathtt{e}}) \xrightarrow{\text{type}} (\overbrace{\mathtt{ty'}}^{\mathtt{t}}, \mathtt{new\_e}, \mathtt{ses})}$$

### TypingRule.CheckATC

The helper function

$$\mathit{check\_atc}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t1}}, \overbrace{\mathsf{ty}}^{\mathtt{t2}}) \longrightarrow \{\mathsf{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 15.28 shows examples of ill-typed ATC expressions.

Listing 15.28: 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

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

DIFFERENT_LABELS_ERROR

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \quad \mathit{ast\_label}(\mathtt{t1}) \neq \mathit{ast\_label}(\mathtt{t2})}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_TAF})}$$

INT_BITS

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \quad \mathit{ast\_label}(\mathtt{t1}) = \mathit{ast\_label}(\mathtt{t2}) \quad \mathit{ast\_label}(\mathtt{t1}) \in \{\mathsf{T\_Int}, \mathsf{T\_Bits}\}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

TUPLE

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \quad \mathtt{t1} = \mathsf{T\_Tuple}(\mathtt{l1}) \quad \mathtt{t2} = \mathsf{T\_Tuple}(\mathtt{l2}) \quad \mathit{check}(|\mathtt{l1}| = |\mathtt{l2}|, \mathtt{TE\_TAF}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \quad \mathtt{i} \in \mathit{indices}(\mathtt{l1}) : \mathit{check\_atc}(\mathtt{l1[i]}, \mathtt{l2[i]}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

OTHER_ERROR

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \quad \mathit{ast\_label}(\mathtt{t1}) = \mathit{ast\_label}(\mathtt{t2}) \quad \mathit{ast\_label}(\mathtt{t1}) \notin \{\mathsf{T\_Int}, \mathsf{T\_Bits}, \mathsf{T\_Tuple}\}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_TAF})}$$

### 15.11.4 Semantics

#### SemanticsRule.ATC

#### Example: Evaluation of an Asserting Type Conversion Expressions

In Listing 15.29, both type assertions — `3 as integer` and `3 as integer{3..5}` — succeed.

Listing 15.29: 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 15.30, the asserting type conversion on `y` does not yield a dynamic error, since the invocation of `f1` returns `FALSE` when evaluated:

Listing 15.30: 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 Type Errors and Dynamic Errors

Listing 15.31 shows various type errors and 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 15.31: Various type errors and dynamic errors

```
func ErrorExample()
begin
  var a: integer{1, 2, 3} = 2 as integer{1, 2, 3}; // Legal
  var b: integer{4, 5, 6} = 2;                     // A type error
  var c: integer{4, 5, 6} = 2 as integer{4, 5, 6}; // A dynamic error
  if FALSE then
      var d: integer{4, 5, 6} = 2; // A type error
      // A dynamic error
      var e: integer{4, 5, 6} = 2 as integer{4, 5, 6};
  end;
end;
```

### Prose

All of the following apply:

- `e` denotes an asserted type conversion expression, $\mathsf{E\_ATC}(\texttt{e1}, \texttt{t})$;
- evaluating `e1` in `env` results in $\mathsf{Normal}((\texttt{v}, \texttt{g1}), \texttt{new\_env})^{/\!/\,\#\mathsf{T},\#\mathsf{DE}}$;
- evaluating whether `v` has type `t` in `env` results in $(\texttt{b}, \texttt{g2})^{/\!/\,\#\mathsf{DE}}$;
- One of the following applies:
  - All of the following apply (OKAY):
    - `b` is the native Boolean for `TRUE`;
    - `g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.
  - All of the following apply (ERROR):
    - `b` is the native Boolean for `FALSE`;
    - the result is a dynamic error indicating that the type assertion failed (`DE_TAF`).

### Formally

OKAY

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g1}), \texttt{new\_env}) \;/\!/\; \#\mathsf{T}, \#\mathsf{DE} \\ \mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \texttt{t}) \xrightarrow{\mathsf{eval}} (\texttt{b}, \texttt{g2}) \;/\!/\; \#\mathsf{DE} \\ \texttt{b} \overset{\mathsf{is}}{=} \mathsf{Bool}(\mathsf{TRUE}) \qquad \texttt{g} := \texttt{g1} \xrightarrow{\mathsf{asl\_data}} \texttt{g2}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_ATC}(\texttt{e1}, \texttt{t})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

ERROR

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \_), \_) \\ \mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \texttt{t}) \xrightarrow{\mathsf{eval}} (\texttt{b}, \_) \qquad \texttt{b} \overset{\mathsf{is}}{=} \mathsf{Bool}(\mathsf{FALSE})\end{array}}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_ATC}(\texttt{e1}, \texttt{t})) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathsf{DE\_TAF})}$$

### SemanticsRule.IsValOfType

### Prose

The relation

$$\mathit{is\_val\_of\_type}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathsf{ty}}^{\texttt{t}}) \bigtimes (\overbrace{\mathbb{B}}^{\texttt{b}} \times \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\mathsf{TDynError}}^{\#\mathsf{DE}}$$

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 15.31, 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 $\texttt{c}_{1..k}$;
  - `v` is the native value integer for $n$;
  - the evaluation of every constraint $\texttt{c}_i$ with $n$ in environment `env` yields a Boolean value $\texttt{b}_i$ and an execution graph $\texttt{g}_i$ //#DE;
  - `b` is the Boolean disjunction of all Boolean values $\texttt{b}_i$, for $i = 1..k$;
  - `g` is the parallel composition of all execution graphs $\texttt{g}_i$, for $i = 1..k$;
- All of the following apply (BITS):
  - `t` is a bitvector type with expression `e`, that is, T_Bits(`e`, _);
  - `v` is a native bitvector value for the sequence of bits `bits`, that is, Bitvector(`bits`);
  - evaluating the side-effect-free expression `e` in `env` yields Normal(`v'`, `g`) //#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):

* $\mathtt{t}$ is a tuple with types $\mathtt{t}_i$, for $i = 1..k$;
* the value at every index $i = 1..k$ of $\mathtt{v}$ is $\mathtt{u}_i$, for $i = 1..k$,
* the evaluation of *is_val_of_type* for every value $\mathtt{u}_i$ and corresponding type $\mathtt{t}_i$, for $i = 1..k$, results in a Boolean $\mathtt{b}_i$ and execution graph $\mathtt{g}_i$ $/\!/^{\#DE}$;
* $\mathtt{b}$ is the Boolean conjunction of all Boolean values $\mathtt{b}_i$, for $i = 1..k$;
* $\mathtt{g}$ is the parallel composition of all execution graphs $\mathtt{g}_i$, for $i = 1..k$; of the constraints.

**Formally**

TYPE_EQUAL

$$\frac{\mathit{ast\_label}(\mathtt{t}) \notin \{\mathsf{T\_Int}, \mathsf{T\_Bits}\}}{\mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{v}, \mathtt{t}) \xrightarrow{\mathsf{eval}} (\overbrace{\mathtt{TRUE}}^{\mathtt{b}}, \overbrace{\emptyset_{\mathtt{g}}}^{\mathtt{g}})}$$

INT_UNCONSTRAINED

$$\mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{v}, \overbrace{\mathsf{T\_Int}(\mathsf{Unconstrained})}^{\mathtt{t}}) \xrightarrow{\mathsf{eval}} (\overbrace{\mathtt{TRUE}}^{\mathtt{b}}, \overbrace{\emptyset_{\mathtt{g}}}^{\mathtt{g}})$$

INT_WELLCONSTRAINED

$$\frac{\mathtt{v} \overset{\mathsf{is}}{=} \mathtt{Int}(n) \qquad i = 1..k : \mathit{is\_constraint\_sat}(\mathtt{env}, \mathtt{c}_i, n) \xrightarrow{\mathsf{eval}} (\mathtt{b}_i, \mathtt{g}_i) \;/\!/\; \#DE \qquad \mathtt{b} := \bigvee_{i=1}^{k} \mathtt{b}_i \qquad \mathtt{g} := \|_{i=1}^{k}\, \mathtt{g}_i}{\mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{v}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{c}_{1..k}))}^{\mathtt{t}}) \xrightarrow{\mathsf{eval}} (\mathtt{b}, \mathtt{g})}$$

BITS

$$\frac{\mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{v'}, \mathtt{g}) \;/\!/\; \#DE}{\mathit{is\_val\_of\_type}(\mathtt{env}, \overbrace{\mathtt{Bitvector}(\mathtt{bits})}^{\mathtt{v}}, \overbrace{\mathsf{T\_Bits}(\mathtt{e}, \_)}^{\mathtt{t}}) \xrightarrow{\mathsf{eval}} (\overbrace{\mathtt{v'} = |\mathtt{bits}|}^{\mathtt{b}}, \mathtt{g})}$$

TUPLE

$$\frac{i = 1..k : \mathit{get\_index}(i, \mathtt{v}) \xrightarrow{\mathsf{eval}} \mathtt{u}_i \qquad i = 1..k : \mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{u}_i, \mathtt{t}_i) \xrightarrow{\mathsf{eval}} (\mathtt{b}_i, \mathtt{g}_i) \;/\!/\; \#DE \qquad \mathtt{b} := \bigwedge_{i=1}^{k} \mathtt{b}_i \qquad \mathtt{g} := \|_{i=1}^{k} \mathtt{g}_i}{\mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{v}, \overbrace{\mathsf{T\_Tuple}(i = 1..k : \mathtt{t}_i)}^{\mathtt{t}}) \xrightarrow{\mathsf{eval}} (\mathtt{b}, \mathtt{g})}$$

### 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

$$\mathit{is\_constraint\_sat}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c}}, \overbrace{\mathbb{Z}}^{n}) \bigtimes (\overbrace{\mathbb{B}}^{\texttt{b}} \times \overbrace{\mathcal{G}}^{\texttt{g}})$$

tests whether the integer value $n$ *satisfies the constraint* `c` (that is, whether $n$ is within the range of values defined by `c`) in the environment `env` and returns a Boolean answer `b` and the execution graph `g` resulting from evaluating the expressions appearing in `c`.

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 $\texttt{g}^{/\!/\texttt{\#DE}}$.
  - define `b` as `TRUE` if and only if $m$ is equal to $n$.
- All of the following apply (Constraint_Range_Sat):
  - `c` is a constraint for the expressions `e1` and `e2`;
  - evaluating the side-effect-free expression `e1` in `env` yields the concurrent native value given by the native integer value for $a$ and the execution graph $\texttt{g1}^{/\!/\texttt{\#DE}}$.
  - evaluating the side-effect-free expression `e2` in `env` yields the concurrent native value given by the native integer value for $b$ and the execution graph $\texttt{g2}^{/\!/\texttt{\#DE}}$.
  - define `b` as `TRUE` if and only if $n$ is greater or equal to $a$ and less than or equal to $b$;
  - define `g` as the parallel composition of `g1` and `g2`.

**Formally**

The use of *eval_expr_sef*() is justified by checks in *annotate_type*(), verifying that expressions appearing in types are all side-effect-free.

CONSTRAINT_EXACT_SAT

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} (\texttt{Int}(m), \texttt{g}) \;/\!/\; \texttt{\#DE}}{\mathit{is\_constraint\_sat}(\texttt{env}, \overbrace{\mathsf{Constraint\_Exact}(\texttt{e})}^{\texttt{c}}, n) \xrightarrow{\text{eval}} (\overbrace{m = n}^{\texttt{b}}, \texttt{g})}$$

CONSTRAINT_RANGE_SAT

$$\frac{\begin{array}{c}\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} (\texttt{Int}(a), \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e2}) \xrightarrow{\text{eval}} (\texttt{Int}(b), \texttt{g2}) \;/\!/\; \texttt{\#DE} \\ \texttt{b} := \mathsf{choice}(a \le n \wedge n \le b, \texttt{TRUE}, \texttt{FALSE}) \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\mathit{is\_constraint\_sat}(\texttt{env}, \overbrace{\mathsf{Constraint\_Range}(\texttt{e1}, \texttt{e2})}^{\texttt{c}}, n) \xrightarrow{\text{eval}} (\texttt{b}, \texttt{g})}$$

## 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 15.32: 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

$$\begin{aligned}\texttt{expr} \longrightarrow\ & \texttt{expr}\ \texttt{"IN"}\ \texttt{pattern\_set} \\ \mid\ & \texttt{expr}\ \texttt{"=="}\ \textbf{MASK\_LIT} \\ \mid\ & \texttt{expr}\ \texttt{"!="}\ \textbf{MASK\_LIT}\end{aligned}$$

### 15.12.2 Abstract Syntax

$$\texttt{expr} \longrightarrow \texttt{E\_Pattern}(\texttt{expr}, \texttt{pattern})$$

**ASTRule.EPattern**

$$\frac{\mathit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast}\ \ //\ \texttt{\#BE} \qquad \mathit{build\_pattern\_set}(\texttt{ps}) \xrightarrow{\text{ast}} \texttt{ps\_ast}\ \ //\ \texttt{\#BE}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e} : \texttt{expr}, \texttt{"IN"}, \texttt{ps} : \texttt{pattern\_set})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Pattern}(\texttt{e\_ast}, \texttt{ps\_ast})}^{\texttt{ast\_node}}}$$

EQ

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"="}, \textbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Pattern}(\overline{\texttt{expr}}, \texttt{Pattern\_Mask}(\texttt{m}))}^{\texttt{ast\_node}}$$

NEQ

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"!="}, \textbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Pattern}(\overline{\texttt{expr}}, \texttt{Pattern\_Not}(\texttt{Pattern\_Mask}(\texttt{m})))}^{\texttt{ast\_node}}$$

### 15.12.3 Typing

**TypingRule.EPattern**

Listing 15.32 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**

$$\frac{\begin{array}{c}\textit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e2}, \texttt{e2}, \texttt{ses\_e}) \;/\!/\; \texttt{\#TE} \\ \textit{annotate\_pattern}(\texttt{tenv}, \texttt{t\_e2}, \texttt{pat}) \xrightarrow{\text{type}} (\texttt{pat'}, \texttt{ses\_pat}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_e} \cup \texttt{ses\_pat}\end{array}}{\textit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_Pattern}(\texttt{e1}, \texttt{pat})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\text{T\_Bool}}^{\texttt{t}}, \overbrace{\text{E\_Pattern}(\texttt{e2}, \texttt{pat'})}^{\texttt{new\_e}}, \texttt{ses})}$$

### 15.12.4 Semantics

**SemanticsRule.EPattern**

**Example: Evaluation of Pattern Expressions**

In Listing 15.32, 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 Normal(`v`, `g2`) where is a native Boolean that determines whether the is indeed a match;
- `g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

**Formally**

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v1}, \texttt{g1}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_pattern}(\texttt{env}, \texttt{v1}, \texttt{p}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v}, \texttt{g2}) \qquad \texttt{g} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Pattern}(\texttt{e}, \texttt{p})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

## 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 ⟶ `"ARBITRARY"` `":"` ty

### 15.13.2 Abstract Syntax

expr ⟶ E_Arbitrary(ty)

**ASTRule.EArbitrary**

$$\frac{\mathit{build\_ty}(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast} \;\; /\!/ \;\; \texttt{\#BE}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{"ARBITRARY"}, \texttt{":"}, \texttt{t} : \texttt{ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Arbitrary}(\texttt{t\_ast})}^{\texttt{ast\_node}}}$$

### 15.13.3 Typing

**TypingRule.EArbitrary**

**Example: Well-typed Arbitrary Value Expressions**

Listing 15.33 show examples of well-typed arbitrary value expressions.

Listing 15.33: 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 $(\texttt{ty1}, \texttt{ses\_ty})^{/\!/\texttt{\#TE}}$;
- obtaining the structure of `ty1` in tenv yields $\texttt{ty2}^{/\!/\texttt{\#TE}}$;
- determining whether `ty2` is not a collection type in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `t` is `ty1`;
- define `new_e` as an expression `ARBITRARY` of type `ty2`, that is, E_Arbitrary(`ty2`);
- define `ses` as the union of `ses_ty` and the singleton set for the non-determinism side effect descriptor.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\text{tenv}, \texttt{ty}) \xrightarrow{\text{type}} (\texttt{ty1}, \texttt{ses\_ty}) \;/\!/\; \texttt{\#TE} \\ \mathit{get\_structure}(\text{tenv}, \texttt{ty1}) \xrightarrow{\text{type}} \texttt{ty2} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_is\_not\_collection}(\text{tenv}, \texttt{ty2}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_ty} \cup \{\text{NonDeterministic}\}\end{array}}{\mathit{annotate\_expr}(\text{tenv}, \text{E\_Arbitrary}(\texttt{ty})) \xrightarrow{\text{type}} (\texttt{ty1}, \text{E\_Arbitrary}(\texttt{ty2}), \texttt{ses})}$$

### 15.13.4 Semantics

**SemanticsRule.EArbitrary**

**Example: Evaluation of ARBITRARY for an Unconstrained Integer Type**

In Listing 15.34, the expression `ARBITRARY : integer` evaluates to an arbitrary integer value.

Listing 15.34: 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 15.35, the expression `ARBITRARY : integer {3, 42}` evaluates to either `Int`(3) or `Int`(42).

Listing 15.35: 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 15.36 demonstrates how to obtain an arbitrary integer-indexed array, `int_array`, and how to obtain an arbitrary enumeration-indexed array, `enum_array`.

Listing 15.36: 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**

$$\text{OKAY}\quad \frac{\mathsf{dyn\_dom}(\mathtt{env},\mathtt{t}) \neq \emptyset \qquad \mathtt{v} \in \mathsf{dyn\_dom}(\mathtt{env},\mathtt{t})}{\mathit{eval\_expr}(\mathtt{env}, \overbrace{\mathsf{E\_Arbitrary}(\mathtt{t})}^{\mathtt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \overbrace{\emptyset_{\mathsf{g}}}^{\mathtt{g}}), \overbrace{\mathtt{env}}^{\mathtt{new\_env}})}$$

$$\text{ERROR}\quad \frac{\mathsf{dyn\_dom}(\mathtt{env},\mathtt{t}) = \emptyset}{\mathit{eval\_expr}(\mathtt{env}, \overbrace{\mathsf{E\_Arbitrary}(\mathtt{t})}^{\mathtt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_AET})}$$

**Comments**

Notice that this rule introduces non-determinism.

## 15.14 Structured Type Construction Expressions

Listing 15.37: 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

$$\begin{aligned}\texttt{expr} &\longrightarrow \textbf{ID}\ \texttt{"\{"}\ \texttt{"-"}\ \texttt{"\}"}\\ &\mid \textbf{ID}\ \texttt{"\{"}\ \texttt{clist1(field\_assign)}\ \texttt{"\}"}\\ \texttt{field\_assign} &\longrightarrow \textbf{ID}\ \texttt{"="}\ \texttt{expr}\end{aligned}$$

### 15.14.2 Abstract Syntax

$$\texttt{expr} \longrightarrow \texttt{E\_Record}(\overbrace{\texttt{ty}}^{\text{record type}}, \overbrace{(\texttt{identifier}, \texttt{expr})^*}^{\text{field initializers}})$$

#### ASTRule.ERecord

$$\frac{}{\textit{build\_expr}(\overbrace{\texttt{expr}(\ \textbf{ID}(\texttt{t}), \texttt{"\{"}, \texttt{"-"}, \texttt{"\}"}\ )}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Record}(\texttt{T\_Named}(\texttt{t}), [\,])}^{\texttt{ast\_node}}}\ \text{EMPTY}$$

$$\frac{\textit{build\_clist}[\textit{build\_field\_assign}](\texttt{field\_assigns}) \xrightarrow{\text{ast}} \texttt{field\_assign\_asts}}{\textit{build\_expr}\left(\overbrace{\texttt{expr}\left(\begin{array}{l}\textbf{ID}(\texttt{t}), \texttt{"\{"},\\ \hookrightarrow\ \texttt{field\_assigns} : \texttt{clist1(field\_assign)},\\ \hookrightarrow\ \texttt{"\}"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Record}(\texttt{T\_Named}(\texttt{t}), \texttt{field\_assign\_asts})}^{\texttt{ast\_node}}}\ \text{NON\_EMPTY}$$

#### ASTRule.FieldAssign

The function

$$\textit{build\_field\_assign}(\overbrace{\mathbb{PARSE}[\texttt{field\_assign}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\texttt{identifier} \times \texttt{expr})}^{\texttt{ast\_node}}$$

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

$$\textit{build\_field\_assign}(\texttt{field\_assign}(\textbf{ID}(\texttt{id}), \texttt{"="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

### 15.14.3 Typing

#### TypingRule.ERecord

Listing 15.37 shows an example of a well-typed record construction expression and an example (in comment) where the same field is initialized twice, which is incalid..

**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`, $\mathtt{e}_{\mathtt{name}}$, $\mathtt{xs}_{\mathtt{name}}$)$^{//\#TE}$;
- define `fields'` as the list containing (`name`, $\mathtt{e}_{\mathtt{name}}$) 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 $\mathtt{xs}_{\mathtt{name}}$ for each `field` element (`name`, _) of `fields`.

**Formally**

$$\frac{\begin{array}{c}\mathit{check}(\mathit{ast\_label}(\texttt{ty}) = \mathsf{T\_Named}, \texttt{NamedTypeExpected}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{ty\_anon} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\texttt{ty\_anon}) \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\}, \texttt{TE\_UT}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ty\_anon} \overset{\text{is}}{=} L(\texttt{field\_types}) \qquad \texttt{initialized\_fields} := \{\texttt{name} \mid (\texttt{name}, \_) \in \texttt{fields}\} \\ \texttt{names} := \mathit{field\_names}(\texttt{field\_types}) \\ \mathit{check}(\{\texttt{names}\} = \{\texttt{initialized\_fields}\}, \texttt{TE\_BF}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_no\_duplicates}(\texttt{initialized\_fields}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ (\texttt{name}, \texttt{e'}) \in \texttt{fields} : \mathit{annotate\_field\_init}(\texttt{tenv}, (\texttt{name}, \texttt{e'}), \texttt{field\_types}) \xrightarrow{\text{type}} \\ (\texttt{name}, \texttt{e}_{\texttt{name}}, \texttt{xs}_{\texttt{name}}) \;/\!/\; \texttt{\#TE} \\ \texttt{fields'} := [(\texttt{name}, \texttt{e'}) \in \texttt{fields} : (\texttt{name}, \texttt{e}_{\texttt{name}})] \qquad \texttt{ses} := \bigcup_{(\texttt{name}, \_) \in \texttt{fields}} \texttt{xs}_{\texttt{name}}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\mathsf{E\_Record}(\texttt{ty}, \texttt{fields})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\texttt{ty}}^{\texttt{t}}, \overbrace{\mathsf{E\_Record}(\texttt{ty}, \texttt{fields'})}^{\texttt{new\_e}}, \texttt{ses})}$$

**TypingRule.AnnotateFieldInit**

The function

$$\mathit{annotate\_field\_init}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})}^{(\texttt{name,e'})}, \overbrace{\texttt{field}^*}^{\texttt{field\_types}}) \longrightarrow (\overbrace{\mathsf{identifier}}^{\texttt{name}} \times \overbrace{\mathsf{expr}}^{\texttt{e''}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})$$

annotates a field initializers $(\texttt{name}, \texttt{e'})$ 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 15.37 for an example.

**Prose**

All of the following apply:

- annotating the expression `e'` in tenv yields $(\texttt{t'}, \texttt{e''}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- checking whether there exists a type associated with `name` in `field_types` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- the unique type associated with `name` in `field_types` is `t_spec'`;
- determining whether `t'` type-satisfies `t_spec'` in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e'}) \xrightarrow{\text{type}} (\mathtt{t'}, \mathtt{e''}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check}(\mathit{field\_type}(\mathtt{field\_types}, \mathtt{name}) \neq \bot, \mathtt{TE\_BF}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{field\_type}(\mathtt{field\_types}, \mathtt{name}) = \mathtt{t\_spec'} \\ \mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{t'}, \mathtt{t\_spec'}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_field\_init}(\mathtt{tenv}, (\mathtt{name}, \mathtt{e'}), \mathtt{field\_types}) \xrightarrow{\text{type}} (\mathtt{name}, \mathtt{e''}, \mathtt{ses})}$$

### 15.14.4 Semantics

**SemanticsRule.ERecord**

**Example: Evaluation of Record Construction Expressions**

In Listing 15.37, the expression `MyRecordType{a=3, b=42}` evaluates to the native record value
$\mathtt{NV\_Record}(\mathtt{a} \mapsto \mathtt{Int}(3), \mathtt{b} \mapsto \mathtt{Int}(42))$.

**Prose**

All of the following apply:

- `e` denotes a record creation expression, $\mathsf{E\_Record}(\mathtt{names}, \mathtt{e\_fields})$;
- the names of the fields are $\mathtt{id}_{1..k}$;
- the expressions associated with the fields are $\mathtt{e}_{1..k}$;
- evaluating the expressions of `fields` in order yields $\mathsf{Normal}((\mathtt{v\_fields}, \mathtt{g}), \mathtt{new\_env})^{/\!/\mathtt{\#T},\mathtt{\#DE}}$;
- `v_fields` is a list of native values $\mathtt{v}_{1..k}$;
- `v` is the native record that maps $\mathtt{id}_i$ to $\mathtt{v}_i$, for $i = 1..k$.

**Formally**

$$\frac{\begin{array}{c}\mathtt{e\_fields} \overset{\text{is}}{=} [i = 1..k : (\mathtt{id}_i, \mathtt{e}_i)] \qquad \mathtt{names} := \mathtt{id}_{1..k} \qquad \mathtt{fields} := \mathtt{e}_{1..k} \\ \mathit{eval\_expr\_list}(\mathtt{env}, \mathtt{fields}) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v\_fields}, \mathtt{g}), \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{v\_fields} \overset{\text{is}}{=} \mathtt{v}_{1..k} \qquad \mathtt{v} := \mathtt{NV\_Record}(\{i = 1..k : \mathtt{id}_i \mapsto \mathtt{v}_i\})\end{array}}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_Record}(\_, \mathtt{e\_fields})) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

## 15.15 Tuple Expressions

Listing 15.38: Tuple expression

```
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;
```

### 15.15.1 Syntax

expr ⟶ plist2(expr)

### 15.15.2 Abstract Syntax

expr ⟶ E_Tuple(expr$^{+}$)

#### ASTRule.ETuple

$$\frac{\text{TUPLE} \qquad \mathit{build\_plist}[\mathit{build\_expr}](\texttt{exprs}) \xrightarrow{\text{ast}} \texttt{expr\_asts}}{\mathit{build\_expr}(\overbrace{\text{expr}(\texttt{exprs} : \text{plist2}(\text{expr}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Tuple}(\texttt{expr\_asts})}^{\texttt{ast\_node}}}$$

### 15.15.3 Typing

#### TypingRule.ETuple

Listing 15.38 shows an example of a well-typed tuple expressions.

#### Prose

One of the following applies:

- All of the following apply (PARENTHESIZED):
  - `e` denotes a tuple expression with list of expressions consisting solely of `e'`, that is, 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 $(\texttt{t}_i, \texttt{e}_i, \texttt{xs}_i)^{\#\texttt{TE}}$;
  - `t` is the tuple type with list of types $\texttt{t}_i$, for $i = 1..k$;
  - `new_e` is tuple expression over list of expressions $\texttt{e}_i$, for $i = 1..k$;
  - defining `ses` as the union of $\texttt{xs}_i$ for $i = 1..k$.

**Formally**

PARENTHESIZED

$$\frac{\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \texttt{ses}) \;/\!/\; \#\texttt{TE}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\texttt{E\_Tuple}(\texttt{e'})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \texttt{ses})}$$

LIST

$$\frac{|\texttt{li}| > 1 \qquad i = 1..k : \mathit{annotate\_expr}(\texttt{tenv}, \texttt{le}[i]) \xrightarrow{\text{type}} (\texttt{t}_i, \texttt{e}_i, \texttt{xs}_i) \;/\!/\; \#\texttt{TE} \qquad \texttt{ses} := \bigcup_{i=1..k} \texttt{xs}_i}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\texttt{E\_Tuple}(\texttt{li})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\texttt{T\_Tuple}(\texttt{t}_{1..k})}^{\texttt{t}}, \overbrace{\texttt{E\_Tuple}(\texttt{e}_{1..k})}^{\texttt{new\_e}}, \texttt{ses})}$$

### 15.15.4 Semantics

**SemanticsRule.ETuple**

**Example: Evaluation of Tuple Expressions**

In Listing 15.38, the expression `(3, Return42())` evaluates to the value `(3, 42)`.

**Prose**

All of the following apply:

- `e` denotes a tuple expression, E_Tuple(`e_list`);
- the evaluation of `e_list` in `env` is Normal$((\texttt{v_list}, \texttt{g}), \texttt{new_env})^{\#\texttt{T},\#\texttt{DE}}$;
- `v` is the native vector constructed from the values in `v_list`.

**Formally**

$$\frac{\mathit{eval\_expr\_list}(\texttt{env}, \texttt{e\_list}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env}) \;/\!/\; \#\texttt{T}, \#\texttt{DE} \qquad \texttt{v} := \texttt{NV\_Vector}(\texttt{v\_list})}{\mathit{eval\_expr}(\texttt{env}, \texttt{E\_Tuple}(\texttt{e\_list})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

## 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 ⟶ "(" 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**

SUB_EXPR

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{"("}, \texttt{expr}, \texttt{")"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Tuple}([\ \overline{\texttt{expr}}\ ])}^{\texttt{ast\_node}}$$

## 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

expr ⟶ E_Array{length : expr, value : expr}
| E_EnumArray{labels : identifier$^+$, value : 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 13.22, 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

$$\texttt{NV\_Vector}([\texttt{Int}(0), \texttt{Int}(0), \texttt{Int}(0), \texttt{Int}(0)]) \ .$$

### 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`)$^{/\!/\texttt{\#T,\#DE}}$;
- evaluating the side-effect-free expression `length` in `env` yields Normal((`v_length`, `g2`))$^{/\!/\texttt{\#DE}}$;
- `v_length` is a native integer value for `n_length`;
- checking that `n_length` is non-negative yields TRUE$^{/\!/\texttt{DE_NAL}}$;
- 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

$$\frac{\begin{array}{c} \textit{eval\_expr}(\texttt{env}, \texttt{e\_value}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{value}, \texttt{g1}), \texttt{new\_env}) \ /\!/ \ \texttt{\#T}, \texttt{\#DE} \\ \textit{eval\_expr\_sef}(\texttt{env}, \texttt{length}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_length}, \texttt{g2})) \ /\!/ \ \texttt{\#DE} \\ \texttt{v\_length} \overset{\text{is}}{=} \texttt{Int}(\texttt{n\_length}) \qquad \textit{check}(\texttt{n\_length} \geq 0, \texttt{DE\_NAL}) \longrightarrow \texttt{TRUE} \ /\!/ \ \texttt{\#DE} \\ \texttt{v} := \texttt{NV\_Vector}(i = 1..\texttt{n\_length} : \texttt{value}) \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2} \end{array}}{\textit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_Array}\{\text{length} : \texttt{length}, \text{value} : \texttt{e\_value}\}}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

### SemanticsRule.EEnumArray

In Listing 13.22, 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

$$\texttt{NV\_Record}(\{\textbf{BIG} \mapsto \texttt{Bitvector}(0000), \textbf{LITTLE} \mapsto \texttt{Bitvector}(0000)\}) \ .$$

**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 $\mathsf{Normal}((\texttt{value}, \texttt{g}), \texttt{new\_env})^{\#\texttt{T},\#\texttt{DE}}$;
- define `v` as the native record mapping each label $\texttt{l} \in \texttt{labels}$ to `value`.

**Formally**

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e\_value}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{value}, \texttt{g}), \texttt{new\_env}) \;/\!/\; \#\texttt{T}, \#\texttt{DE} \quad \texttt{v} := \texttt{NV\_Record}(\texttt{l} \in \texttt{labels} : [\texttt{l} \mapsto \texttt{value}])}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\mathsf{E\_EnumArray}\{\text{labels} : \texttt{labels}, \text{value} : \texttt{e\_value}\}}^{\texttt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

# 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 $\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\mathsf{type}} (\_, \_, \texttt{ses})$ and `ses` only contains side effect descriptors for reading storage (ReadLocal and ReadGlobal).

## 15.18.2 Semantics

**SemanticsRule.ESideEffectFreeExpr**

**Prose**

The helper relation

$$\mathit{eval\_expr\_sef}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{expr}}^{\texttt{e}}) \bigtimes \mathsf{Normal}(\overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\mathsf{TDynError}}^{\#\texttt{DE}}$$

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 $\mathsf{L\_Int}(1)$ in any environment `env`, yields $\mathsf{Normal}(\texttt{Int}(1), \emptyset_{\texttt{g}})$.

**Formally**

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{env}) \;\; /\!/ \;\; \texttt{\#DE}}{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v}, \texttt{g})}$$

Notice that the output configuration does not contain an environment, since side-effect-free expressions do not modify the environment.

## 15.19 Evaluating a List of Expressions

**SemanticsRule.EExprList**

The relation

$$\mathit{eval\_expr\_list}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\text{expr}^*}^{\texttt{le}}) \bigtimes \text{Normal}((\overbrace{\mathbb{V}^*}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\text{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

evaluates the list of expressions `le` in left-to-right order in the initial environment `env` and returns the resulting value `v`, the parallel composition of the execution graphs generated from evaluating each expression, and the new environment `new_env`. If the evaluation of any expression terminates abnormally then the abnormal configuration is returned.

**Example: Evaluating a List of Expressions**

In Listing 15.37, 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`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - evaluating the list of expressions `le1` in the environment `env1` yields Normal((`vs`, `g2`), `env2`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - define `g` as the parallel composition of `g1` and `g2`;
  - define `v` as the list with `head` `v1` and `tail` `vs`.

**Formally**

EMPTY

$$\mathit{eval\_expr\_list}(\mathtt{env}, \overbrace{[\,]}^{\mathtt{le}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\overbrace{[\,]}^{\mathtt{v}}, \overbrace{\emptyset_{\mathsf{g}}}^{\mathtt{g}}), \overbrace{\mathtt{env}}^{\mathtt{new\_env}})$$

NON_EMPTY

$$\frac{\mathtt{le} = [\mathtt{e}] + \mathtt{le1} \qquad \mathit{eval\_expr}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v1}, \mathtt{g1}), \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \quad \mathit{eval\_expr\_list}(\mathtt{env1}, \mathtt{le1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{vs}, \mathtt{g2}), \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \quad \mathtt{g} := \mathtt{g1} \parallel \mathtt{g2} \qquad \mathtt{v} := [\mathtt{v1}] + \mathtt{vs}}{\mathit{eval\_expr\_list}(\mathtt{env}, \mathtt{le}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

# 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` ⟶ `"["` clist0(`slice`) `"]"`

### 16.1.2 Abstract Syntax

#### ASTRule.Slices

The function

$$\mathit{build\_slices}(\overbrace{\mathbb{PARSE}[\texttt{slices}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{slice}^{+}}^{\texttt{ast\_node}}$$

transforms a parse node for a list of slices `parsed_node` into an AST node for a list of slices `ast_node`.

$$\frac{\mathit{build\_clist}[\mathit{build\_slice}](\texttt{slices}) \xrightarrow{\mathsf{ast}} \texttt{slice\_asts}}{\mathit{build\_slices}(\texttt{slices}(\texttt{"["}, \texttt{slices} : \mathsf{clist1}(\texttt{slice}), \texttt{"]"})) \xrightarrow{\mathsf{ast}} \overbrace{\texttt{slice\_asts}}^{\texttt{ast\_node}}}$$

### 16.1.3 Typing

#### TypingRule.Slices

The function

$$\mathit{annotate\_slices}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{slice}^{*}}^{\texttt{slices}}) \longrightarrow (\overbrace{\mathsf{slice}^{*}}^{\texttt{slices'}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})$$

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.

#### Prose

All of the following apply:

- annotating the slice $\texttt{slices}[\texttt{i}]$ in tenv, for each $\texttt{i} \in \mathit{indices}(\texttt{slices})$, yields $(\texttt{s}_\texttt{i}, \texttt{xs}_i)^{/\!/\texttt{\#TE}}$;
- define `slices'` as the list of slices $\texttt{s}_\texttt{i}$, for each $\texttt{i} \in \mathit{indices}(\texttt{slices})$;
- define `ses` as the union of all $\texttt{xs}_i$, for every index $i$ in the list of indices for `slices`.

#### Formally

$$\frac{i \in \mathit{indices}(\texttt{slices}) : \mathit{annotate\_slice}(\mathsf{tenv}, \texttt{slices}[i]) \xrightarrow{\mathsf{type}} (\texttt{s}_i, \texttt{xs}_i) \;/\!/\; \texttt{\#TE} \quad \texttt{slices'} := [i \in \mathit{indices}(\texttt{slices}) : \texttt{s}_i] \quad \texttt{ses} := \bigcup_{i \in \mathit{indices}(\texttt{slices})} \texttt{xs}_i}{\mathit{annotate\_slices}(\mathsf{tenv}, \texttt{slices}) \xrightarrow{\mathsf{type}} (\texttt{slices'}, \texttt{ses})}$$

### 16.1.4 Semantics

#### SemanticsRule.Slices

The relation

$$\mathit{eval\_slices}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices}}) \times \mathsf{Normal}(((\overbrace{\mathbb{V} \times \mathbb{V})^*}^{\texttt{ranges}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{Throwing}}^{\#T} \cup \overbrace{\mathsf{TDynError}}^{\#DE}$$

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 16.2, evaluating the list of slices `[2, 7:5, 0+:3]` yields the list of ranges $[(2, 1), (5, 2), (0, 3)]$.

Listing 16.2: 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 Normal((`range`, `g1`), `env1`)$^{\#T,\#DE}$;
  - evaluating the tail list `slices1` in `env1` results in Normal((`ranges1`, `g2`), `new_env`)$^{\#T,\#DE}$;
  - `ranges` is the concatenation of `range` to `ranges1`;
  - `new_g` is the parallel composition of `g1` and `g2`.

**Formally**

$$\text{EMPTY}\quad \frac{}{\mathit{eval\_slices}(\texttt{env}, [\,]) \xrightarrow{\text{eval}} \text{Normal}((([\,], \emptyset_g), \texttt{env})}$$

$$\text{NONEMPTY}\quad \frac{\begin{array}{c}\texttt{slices} \overset{\text{is}}{=} [\texttt{slice}] + \texttt{slices1} \\ \mathit{eval\_slice}(\texttt{env}, \texttt{slice}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{range}, \texttt{g1}), \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_slices}(\texttt{env1}, \texttt{slices1}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{ranges1}, \texttt{g2}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{ranges} := [\texttt{range}] + \texttt{ranges1} \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\mathit{eval\_slices}(\texttt{env}, \texttt{slices}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{ranges}, \texttt{new\_g}), \texttt{new\_env})}$$

## 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 ⟶ expr
        | expr ":" expr
        | expr "+:" expr
        | expr "*:" expr
        | ":" expr
```

### 16.2.2 Abstract Syntax

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

Note that the syntax `[:width]` is a shorthand for `x[width-1:0]`.

**ASTRule.Slice**

The function

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

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

SINGLE

$$\mathit{build\_slices}(\texttt{slice}(\texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Slice\_Single}(\overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

RANGE

$$\frac{\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \qquad \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast}}{\mathit{build\_slices}(\texttt{slice}(\texttt{e1} : \texttt{expr}, \texttt{":"}, \texttt{e2} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Slice\_Range}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

LENGTH

$$\frac{\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \qquad \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast}}{\mathit{build\_slices}(\texttt{slice}(\texttt{e1} : \texttt{expr}, \texttt{"+:"}, \texttt{e2} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Slice\_Length}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

SCALED

$$\frac{\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \qquad \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast}}{\mathit{build\_slices}(\texttt{slice}(\texttt{e1} : \texttt{expr}, \texttt{"*:"}, \texttt{e2} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Slice\_Star}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

WIDTH

$$\mathit{build\_slices}(\texttt{slice}(\texttt{":"}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Slice\_Length}(\mathsf{E\_Literal}(\mathsf{L\_Int}(0)), \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

### 16.2.3 Typing

**TypingRule.Slice**

the function

$$\mathit{annotate\_slice}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}}^{\texttt{s}}) \longrightarrow \overbrace{\mathsf{slice}}^{\texttt{s'}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a single slice `s` in the static environment tenv, resulting in an annotated slice `s'`. Otherwise, the result is a type error.

**Example: Annotating Slices**

The slices in Listing 16.3, Listing 16.3, Listing 16.4, Listing 16.5, Listing 16.6, and Listing 16.7 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`)$/\!/^{\#TE}$;
  - 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**

$$\text{SINGLE}\ \frac{\mathit{annotate\_slice}(\mathsf{Slice\_Length}(\mathtt{i}, \mathsf{E\_Literal}(1))) \xrightarrow{\mathsf{type}} \mathtt{s'}\ /\!/\ \mathtt{\#TE}}{\mathit{annotate\_slice}(\mathsf{tenv}, \overbrace{\mathsf{Slice\_Single}(\mathtt{i})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \mathtt{s'}}$$

RANGE

$$\frac{\begin{array}{c}\mathit{binop\_literals}(\texttt{MINUS}, \texttt{j}, \texttt{i}) \xrightarrow{\text{type}} \texttt{length'} \\ \mathit{binop\_literals}(\texttt{PLUS}, \texttt{length'}, \mathsf{E\_Literal}(1)) \xrightarrow{\text{type}} \texttt{length} \\ \mathit{annotate\_slice}(\mathsf{Slice\_Length}(\texttt{i}, \texttt{length})) \xrightarrow{\text{type}} \texttt{s'} \ /\!/ \ \texttt{\#TE}\end{array}}{\mathit{annotate\_slice}(\texttt{tenv}, \overbrace{\mathsf{Slice\_Range}(\texttt{j}, \texttt{i})}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{s'}}$$

LENGTH

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{offset}) \xrightarrow{\text{type}} (\texttt{t\_offset}, \texttt{offset'}, \texttt{ses\_offset}) \ /\!/ \ \texttt{\#TE} \\ \mathit{annotate\_symbolic\_constrained\_integer}(\texttt{tenv}, \texttt{length}) \xrightarrow{\text{type}} (\texttt{length'}, \texttt{ses\_length}) \ /\!/ \ \texttt{\#TE} \\ \mathit{check\_underlying\_integer}(\texttt{tenv}, \texttt{t\_offset}) \xrightarrow{\text{type}} \texttt{TRUE} \ /\!/ \ \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_offset} \cup \texttt{ses\_length}\end{array}}{\mathit{annotate\_slice}(\texttt{tenv}, \overbrace{\mathsf{Slice\_Length}(\texttt{offset}, \texttt{length})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Slice\_Length}(\texttt{offset'}, \texttt{length'})}^{\texttt{s'}}, \texttt{ses})}$$

SCALED

$$\frac{\begin{array}{c}\mathit{binop\_literals}(\texttt{MUL}, \texttt{factor}, \texttt{length}) \xrightarrow{\text{type}} \texttt{offset} \\ \mathit{annotate\_slice}(\mathsf{Slice\_Length}(\texttt{offset}, \texttt{length})) \xrightarrow{\text{type}} \texttt{s'} \ /\!/ \ \texttt{\#TE}\end{array}}{\mathit{annotate\_slice}(\texttt{tenv}, \overbrace{\mathsf{Slice\_Star}(\texttt{factor}, \texttt{length})}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{s'}}$$

### TypingRule.SlicesWidth

The helper function

$$\mathit{slices\_width}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{width}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

returns an expression `slices` that represents the width of all slices given by `slices` in the static environment tenv.

### Example: The Width of a List of Slices

In Listing 16.2 The width of the list of slices `[2, 7:5, 0+:3]` is 7.

### Prose

One of the following applies:

- All of the following apply (EMPTY):

  - `slices` is the empty list;
  - `width` is the literal integer expression for 0.

- All of the following apply (NON_EMPTY):
  - `slices` is the list with `head s` and `tail slices1`;
  - applying *slice_width* to `s` yields `e1`;
  - applying *slices_width* to `slices1` yields `e2`;
  - symbolically simplifying the binary expression summing `e1` with `e2` yields `width`$^{\#\texttt{TE}}$.

**Formally**

EMPTY

$$\textit{slices\_width}(\text{tenv}, \overbrace{[\,]}^{\texttt{slices}}) \xrightarrow{\text{type}} \overbrace{\underbrace{\text{E\_Literal(L\_Int)}}_{0}}^{\texttt{width}}$$

NON_EMPTY

$$\frac{\textit{slice\_width}(\texttt{s}) \xrightarrow{\text{type}} \texttt{e1} \qquad \textit{slices\_width}(\texttt{slices1}) \xrightarrow{\text{type}} \texttt{e2} \qquad \textit{normalize}(\overbrace{\texttt{e1 PLUS e2}}^{\text{E\_Binop}}) \xrightarrow{\text{type}} \texttt{width} \mathbin{//} \texttt{\#TE}}{\textit{slices\_width}(\text{tenv}, \overbrace{[\texttt{s}] + \texttt{slices1}}^{\texttt{slices}}) \xrightarrow{\text{type}} \texttt{width}}$$

**TypingRule.SliceWidth**

The helper function

$$\textit{slice\_width}(\overbrace{\text{slice}}^{\texttt{slice}}) \longrightarrow \overbrace{\text{expr}}^{\texttt{width}}$$

returns an expression `width` that represents the width of the slices given by `slice`.

**Example: The Width of a Slice**

In Listing 16.3, the width of the slice `[2]` is 1.
In Listing 16.4, the width of the slice `4:2` is 3.
In Listing 16.5, the width of the slice `2+:3` is 3.
In Listing 16.6, the width of the slice `x[3*:2]` is 2.
In Listing 16.7, the width of the slice `x[:3]` is 3.

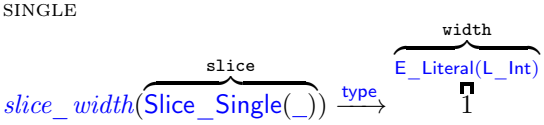
**Prose**

One of the following applies:

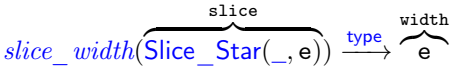
- All of the following apply (SINGLE):
  - `slice` is a single slice, that is, Slice_Single(_);
  - `width` is the literal integer expression for 1;
- All of the following apply (SCALED, LENGTH):
  - `slice` is either a slice of the form `_*:e` or `_+:e`;
  - `width` is `e`;
- All of the following apply (RANGE):
  - `slice` is a slice of the form `e1:e2`;
  - `width` is the expression for $1 + (\texttt{e1} - \texttt{e2})$.

**Formally**

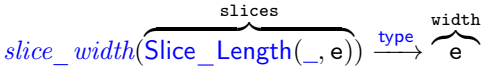
SINGLE



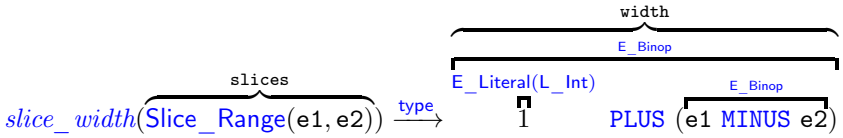

SCALED




LENGTH

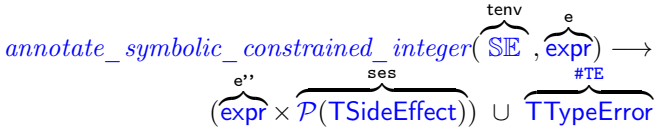



RANGE




**TypingRule.SymbolicConstrainedInteger**

The function




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 13.25, 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 13.36 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 $(\texttt{t}, \texttt{e'}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- determining whether `t` is a symbolically constrained integer in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- applying *normalize* to `e'` in `tenv` yields `e''`.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_symbolically\_evaluable\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} (\mathsf{t}, \mathsf{e'}, \mathsf{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{normalize}(\mathsf{tenv}, \mathsf{e'}) \xrightarrow{\text{type}} \mathsf{e''}\end{array}}{\mathit{annotate\_symbolic\_constrained\_integer}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} (\mathsf{e''}, \mathsf{ses})}$$

## 16.2.4 Semantics

**SemanticsRule.Slice**

The relation

$$\mathit{eval\_slice}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{slice}}^{\texttt{s}}) \times \underbrace{\mathsf{Throwing}}_{\texttt{\#T}} \cup \underbrace{\mathsf{TDynError}}_{\texttt{\#DE}} \cup \mathsf{Normal}(((\overbrace{\mathcal{Z}}^{\texttt{v\_start}} \times \overbrace{\mathcal{Z}}^{\texttt{v\_length}}) \times \overbrace{\mathcal{G}}^{\texttt{new\_g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup$$

evaluates an individual slice `s` in an environment `env` is, resulting either in $\mathsf{Normal}(((\texttt{v\_start}, \texttt{v\_length}), \texttt{g}), \texttt{new\_env})$, a throwing configuration, or a dynamic error configuration.

**Example: Evaluating Slices**

In Listing 16.3, the slice `[2]` evaluates to `(2, 1)`, that is, the slice of length 1 starting at index 2.

Listing 16.3: A single expression slice

```
func main () => integer
begin
  let x = '00000100';
  assert x[2] == '1';
  return 0;
end;
```

In Listing 16.4, `4:2` evaluates to `(2, 3)`.

Listing 16.4: A range slice

```
func main () => integer
begin
  let x = '00011100';
  assert x[4:2] == '111';
  return 0;
end;
```

In Listing 16.5, `2+:3` evaluates to `(2, 3)`.

Listing 16.5: A slice by length

```
func main () => integer
begin
  let x = '00011100';
  assert x[2+:3] == '111';
  return 0;
end;
```

In Listing 16.6, `x[3*:2]` evaluates to `'11'`.

Listing 16.6: A scaled slice

```
func main () => integer
begin
  let x = '11000000';

  assert x[3*:2] == '11';

  return 0;
end;
```

In Listing 16.7, `x[:3]` evaluates to `'100'`.

Listing 16.7: 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

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_start}, \texttt{new\_g}), \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{v\_length} := \texttt{Int}(1)}{\mathit{eval\_slice}(\texttt{env}, \text{Slice\_Single}(\texttt{e})) \xrightarrow{\text{eval}} \text{Normal}(((\texttt{v\_start}, \texttt{v\_length}), \texttt{new\_g}), \texttt{new\_env})}$$

RANGE

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e\_top}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_top}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{m\_top} \overset{\text{is}}{=} (\texttt{v\_top}, \texttt{g1}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_start}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_start}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_start} \overset{\text{is}}{=} (\texttt{v\_start}, \texttt{g2}) \qquad \mathit{binop}(\texttt{MINUS}, \texttt{v\_top}, \texttt{v\_start}) \xrightarrow{\text{eval}} \texttt{v\_diff} \\ \mathit{binop}(\texttt{PLUS}, \texttt{Int}(1), \texttt{v\_diff}) \xrightarrow{\text{eval}} \texttt{v\_length} \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\texttt{env}, \text{Slice\_Range}(\texttt{e\_top}, \texttt{e\_start})) \xrightarrow{\text{eval}} \\ \text{Normal}(((\texttt{v\_start}, \texttt{v\_length}), \texttt{new\_g}), \texttt{new\_env})\end{array}}$$

LENGTH

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e\_start}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_start}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{length}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_length}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_start} \overset{\text{is}}{=} (\texttt{v\_start}, \texttt{g1}) \qquad \texttt{m\_length} \overset{\text{is}}{=} (\texttt{v\_length}, \texttt{g2}) \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\texttt{env}, \text{Slice\_Length}(\texttt{e\_start}, \texttt{length})) \xrightarrow{\text{eval}} \\ \text{Normal}(((\texttt{v\_start}, \texttt{v\_length}), \texttt{new\_g}), \texttt{new\_env})\end{array}}$$

SCALED

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{factor}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_factor}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{m\_factor} \overset{\text{is}}{=} (\texttt{v\_factor}, \texttt{g1}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{length}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_length}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{m\_length} \overset{\text{is}}{=} (\texttt{v\_length}, \texttt{g2}) \\ \mathit{binop}(\texttt{MUL}, \texttt{v\_factor}, \texttt{v\_length}) \xrightarrow{\text{eval}} \texttt{v\_start} \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\texttt{env}, \text{Slice\_Star}(\texttt{factor}, \texttt{length})) \xrightarrow{\text{eval}} \\ \text{Normal}(((\texttt{v\_start}, \texttt{v\_length}), \texttt{new\_g}), \texttt{new\_env})\end{array}}$$

# Chapter 17

# Pattern Matching

Patterns are grammatically derived from `pattern` and represented as an AST by pattern.

The function

$$build\_pattern(\overbrace{\mathbb{PARSE}[\texttt{pattern}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{pattern}}^{\texttt{ast\_node}}$$

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

The function

$$annotate\_pattern(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{t}}, \overbrace{\mathsf{pattern}}^{\texttt{p}}) \longrightarrow (\overbrace{\mathsf{pattern}}^{\texttt{new\_p}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a pattern `p` in a static environment `tenv` given a type `t`, resulting in `new_p`, which is the typed AST node for `p` and the inferred set of side effect descriptors `ses`. Otherwise, the result is a type error..

The relation

$$eval\_pattern(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathsf{pattern}}^{\texttt{p}}) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{B}}^{\texttt{b}}, \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

determines whether a value `v` matches the pattern `p` in an environment `env` resulting in either Normal(`b`, `new_g`) or an abnormal configuration.

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 ⟶ `"-"`

### 17.1.2 Abstract Syntax

pattern ⟶ Pattern_All

**ASTRule.PAll**

$$\textit{build\_pattern}(\texttt{pattern}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_All}}^{\texttt{ast\_node}}$$

### 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**

$$\mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t}, \overbrace{\mathsf{Pattern\_All}}^{\texttt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_All}}^{\texttt{new\_p}}, \overbrace{\emptyset}^{\texttt{ses}})$$

### 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**

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

## 17.2 Matching a Single Value

### 17.2.1 Syntax

pattern ⟶ expr_pattern

### 17.2.2 Abstract Syntax

pattern ⟶ Pattern_Single(expr)

**ASTRule.PSingle**

$$build\_pattern(\texttt{pattern}(\texttt{expr\_pattern})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Single}(\overline{\texttt{expr\_pattern}})}^{\texttt{ast\_node}}$$

### 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, Pattern_Single(`e`);
- annotating the expression `e` in `tenv` yields $(\texttt{t\_e}, \texttt{e'}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- checking that `ses` is symbolically evaluable yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t` yields $\texttt{t\_struct}^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e` yields $\texttt{t\_e\_struct}^{/\!/\texttt{\#TE}}$;

- 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$^{/\!/\texttt{\#TE}}$;
  - 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$^{/\!/\texttt{\#TE}}$;
    - determining whether the bitwidths of `t_struct` and `t_e_struct` are equal yields TRUE$^{/\!/\texttt{\#TE}}$;
  - 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$^{/\!/\texttt{\#TE}}$;
    - determining whether the lists of enumeration literals of `t_struct` and `t_e_struct` are equal yields TRUE$^{/\!/\texttt{\#TE}}$;
  - All of the following apply (ERROR):
    - determining whether the labels of `t_struct` and `t_e_struct` are the same yields TRUE$^{/\!/\texttt{\#TE}}$;
    - the label of `t_struct` is not one of T_Bool, T_Real, T_Int, T_Bits, or T_Enum;
    - the result is a type error indicating that the types `t` and `t_e` are inappropriate for this pattern.
- `new_p` is the pattern that matches the expression `e'`, that is, Pattern_Single(`e'`).

**Formally**

T_BOOL, T_REAL, T_INT, T_STRING

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_symbolically\_evaluable}(\mathtt{ses}) \xrightarrow{\text{type}} \mathsf{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e}) \xrightarrow{\text{type}} \mathtt{t\_e\_struct} \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{ast\_label}(\mathtt{t\_struct}) \in \{\mathsf{T\_Bool}, \mathsf{T\_Real}, \mathsf{T\_Int}, \mathsf{T\_String}\} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}), \mathsf{TE\_BO}) \longrightarrow \mathsf{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Single}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Single}(\mathtt{e'})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

T_BITS

$$\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_symbolically\_evaluable}(\mathtt{ses}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} \mathtt{t\_e\_struct} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{ast\_label}(\mathtt{t\_struct}) = \mathsf{T\_Bits} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}), \mathtt{TE\_BO}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{bitwidth\_equal}(\mathsf{tenv}, \mathtt{t\_struct}, \mathtt{t\_e\_struct}) \xrightarrow{\mathsf{type}} \mathtt{b} \\ \mathit{check}(\mathtt{b}, \mathtt{BitvectorsDifferentWidths}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \end{array}$$

$$\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Single}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{Pattern\_Single}(\mathtt{e'})}^{\mathtt{new\_p}}, \mathtt{ses})$$

T_ENUM

$$\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_symbolically\_evaluable}(\mathtt{ses}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} \mathtt{t\_e\_struct} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{ast\_label}(\mathtt{t\_struct}) = \mathsf{T\_Enum} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}), \mathtt{TE\_BO}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{t\_struct} \overset{\mathsf{is}}{=} \mathsf{T\_Enum}(\mathtt{li1}) \qquad \mathtt{t\_e\_struct} \overset{\mathsf{is}}{=} \mathsf{T\_Enum}(\mathtt{li2}) \\ \mathit{check}(\mathtt{li1} = \mathtt{li2}, \mathtt{EnumDifferentLabels}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \end{array}$$

$$\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Single}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{Pattern\_Single}(\mathtt{e'})}^{\mathtt{new\_p}}, \mathtt{ses})$$

ERROR

$$\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_symbolically\_evaluable}(\mathtt{ses}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} \mathtt{t\_e\_struct} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}), \mathtt{TE\_BO}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{ast\_label}(\mathtt{t\_struct}) \notin \{\mathsf{T\_Bool}, \mathsf{T\_Real}, \mathsf{T\_Int}, \mathsf{T\_Bits}, \mathsf{T\_Enum}\} \end{array}$$

$$\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Single}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UT})$$

### 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`)$^{/\!/\,\texttt{\#DE}}$;
- `b` is the Boolean value corresponding to whether `v` is equal to `v1`.

**Formally**

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{new\_g}) \;/\!/\; \texttt{\#DE} \qquad \mathit{binop}(\texttt{EQ\_OP}, \texttt{v}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \overbrace{\text{Pattern\_Single}(\texttt{e})}^{\texttt{p}}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

## 17.3 Matching a Range of Integers

### 17.3.1 Syntax

```
pattern ⟶ 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**

$$\textit{build\_pattern}(\texttt{pattern}(\texttt{expr\_pattern}, \texttt{".."}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Range}(\texttt{expr\_pattern}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

### 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, Pattern_Range(`e1`, `e2`);
- annotating the symbolically evaluable expression `e1` in the static environment `tenv` yields (`t_e1`, `e1'`, `ses1`)[//#TE];

- annotating the symbolically evaluable expression `e2` in the static environment `tenv` yields $(\texttt{t\_e2}, \texttt{e2'}, \texttt{ses2})^{/\!/\texttt{\#TE}}$;
- define `ses` as the union of `ses1` and `ses2`;
- determining whether both `e1'` and `e2'` are compile-time constant expressions yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type for `t`, `t_e1`, and `t_e2` yields `t_struct`, `t_e1_struct`, and `t_e2_struct`, respectively$^{/\!/\texttt{\#TE}}$;
- a check the AST labels of `t_struct`, `t_e1_struct`, and `t_e2_struct` are all the same and are either T_Int or T_Real yields `TRUE`. Otherwise, the result is a type error, which short-circuits the entire rule. The type error indicates that the types of `e1`, `e2` and the type `t` must be either of integer type or of real type.
- `new_p` is a range pattern with bounds `e1'` and `e2'`, that is, Pattern_Range(`e1'`, `e2'`).

**Formally**

$$\frac{\begin{array}{c} \mathit{annotate\_symbolically\_evaluable\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e1'}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_symbolically\_evaluable\_expr}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} (\texttt{t\_e2}, \texttt{e2'}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses1} \cup \texttt{ses2} \qquad \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{t\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e1\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e2}) \xrightarrow{\text{type}} \texttt{t\_e2\_struct} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \mathit{ast\_label}(\texttt{t\_struct}) = \mathit{ast\_label}(\texttt{t\_e1\_struct}) = \mathit{ast\_label}(\texttt{t\_e2\_struct}) \wedge \\ \mathit{ast\_label}(\texttt{t\_struct}) \in \{\text{T\_Int}, \text{T\_Real}\} \\ \mathit{check}(\texttt{b}, \texttt{InvalidTypesForBinop}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t}, \overbrace{\text{Pattern\_Range}(\texttt{e1}, \texttt{e2})}^{\texttt{p}}) \xrightarrow{\text{type}} (\overbrace{\text{Pattern\_Range}(\texttt{e1'}, \texttt{e2'})}^{\texttt{new\_p}}, \texttt{ses})}$$

### 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**

$$\frac{\begin{array}{c} \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{g1}) \;/\!/\; \#DE \\ \mathit{binop}(\texttt{GEQ}, \texttt{v}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b1} \qquad \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v2}, \texttt{g2}) \;/\!/\; \#DE \\ \mathit{binop}(\texttt{LEQ}, \texttt{v}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{b2} \qquad \mathit{binop}(\texttt{BAND}, \texttt{b1}, \texttt{b2}) \xrightarrow{\text{eval}} \texttt{b} \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2} \end{array}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \text{Pattern\_Range}(\texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

## 17.4 Matching an Upper Bounded Range of Integers

### 17.4.1 Syntax

pattern ⟶ "<=" expr

### 17.4.2 Abstract Syntax

pattern ⟶ Pattern_Leq(expr)

**ASTRule.PLeq**

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{"<="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Leq}(\overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

### 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_BO`) is returned (indicating that the types of `t` and `t_e` are inappropriate for the `LEQ` operator), which short-circuits the entire rule;
- `new_p` is the pattern which matches anything less than or equal to `e'`.

**Formally**

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_symbolically\_evaluable}(\mathtt{ses}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e}) \xrightarrow{\text{type}} \mathtt{t\_e\_struct} \;/\!/\; \texttt{\#TE} \\ \mathtt{b} := \mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}) \wedge \\ \mathit{ast\_label}(\mathtt{t\_struct}) \in \{\mathsf{T\_Int}, \mathsf{T\_Real}\} \\ \mathit{check}(\mathtt{b}, \texttt{TE\_BO}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Leq}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Leq}(\mathtt{e'})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

### 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`)$^{/\!/\texttt{\#DE}}$;
- `b` is the Boolean value corresponding to whether `v` is less than or equal to `v1`.

**Formally**

$$\frac{\begin{array}{c} \mathit{eval\_expr\_sef}(\mathsf{env}, \mathtt{e}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{v1}, \mathtt{new\_g}) \;/\!/\; \texttt{\#DE} \\ \mathit{binop}(\texttt{LEQ}, \mathtt{v}, \mathtt{v1}) \xrightarrow{\text{eval}} \mathtt{b} \end{array}}{\mathit{eval\_pattern}(\mathsf{env}, \mathtt{v}, \mathsf{Pattern\_Leq}(\mathtt{e})) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{b}, \mathtt{new\_g})}$$

## 17.5 Matching a Lower Bounded Range of Integers

### 17.5.1 Syntax

pattern ⟶ ">=" expr

### 17.5.2 Abstract Syntax

pattern ⟶ Pattern_Geq(expr)

#### ASTRule.PGeq

$$build\_pattern(\texttt{pattern}(\texttt{">="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Geq}(\overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

### 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'`, `ses`)$^{/\!/\texttt{\#TE}}$;
- checking that `ses` is symbolically evaluable yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t` in `tenv` yields `t_struct`$^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e` in `tenv` yields `t_e_struct`$^{/\!/\texttt{\#TE}}$;
- `b` is true if and only if `t_struct` and `t_e_struct` are both integer types or both 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**

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_e}, \texttt{e'}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_symbolically\_evaluable}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{t\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e}) \xrightarrow{\text{type}} \texttt{t\_e\_struct} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \mathit{ast\_label}(\texttt{t\_struct}) = \mathit{ast\_label}(\texttt{t\_e\_struct}) \wedge \\ \mathit{ast\_label}(\texttt{t\_struct}) \in \{\mathsf{T\_Int}, \mathsf{T\_Real}\} \\ \mathit{check}(\texttt{b}, \texttt{InvalidTypesForBinop}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t}, \overbrace{\mathsf{Pattern\_Geq}(\texttt{e})}^{\texttt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Geq}(\texttt{e'})}^{\texttt{new\_p}}, \texttt{ses})}$$

### 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**

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{new\_g}) \;/\!/\; \#DE \qquad \mathit{binop}(\texttt{GEQ}, \texttt{v}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \text{Pattern\_Geq}(\texttt{e})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

## 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**

$$\mathit{build\_pattern}(\texttt{pattern}(\textbf{MASK\_LIT}(\texttt{m}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Mask}(\texttt{m})}^{\text{ast\_node}}$$

### 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$^{/\!/\text{\#TE}}$;
- `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$^{/\!/\text{\#TE}}$;
- `new_p` is `p`;
- define `ses` as the empty set.

#### Formally

$$\frac{\mathit{check\_structure}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Bits}) \xrightarrow{\text{type}} \mathsf{TRUE}\ /\!/\ \mathsf{\#TE} \qquad \mathtt{n} := |\mathtt{m}| \qquad \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Bits}(\mathtt{n}, [\,])) \xrightarrow{\text{type}} \mathsf{TRUE}\ /\!/\ \mathsf{\#TE}}{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Mask}(\mathtt{m})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Mask}(\mathtt{m})}^{\mathtt{new\_p}}, \overbrace{\emptyset}^{\mathtt{ses}})}$$

### 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 $\mathtt{u}_{1..n}$;
  - `b` is the native Boolean formed from the conjunction of Boolean values for each $i$, where the bit $\mathtt{u}_i$ is checked for matching the mask character $\mathtt{m}_i$;
  - `new_g` is the empty graph.

**Formally**

The helper function $\mathit{mask\_match} : \{0, 1, \mathtt{x}\} \times \{0, 1\} \to \mathbb{B}$, checks whether a bit value (second operand) matches a mask value (first operand), is defined by the following table:

| ***mask_match*** | 0 | 1 | x |
|---|---|---|---|
| 0 | TRUE | FALSE | TRUE |
| 1 | FALSE | TRUE | TRUE |

EMPTY

$$\mathit{eval\_pattern}(\mathtt{env}, \overbrace{\mathtt{Bitvector}([\,])}^{\mathtt{v}}, \mathsf{Pattern\_Mask}([\,])) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{TRUE}, \emptyset_{\mathsf{g}})$$

NON_EMPTY

$$\frac{\mathtt{m} = \mathtt{m}_{1..n} \qquad \mathtt{v} = \mathtt{Bitvector}(\mathtt{u}_{1..n}) \qquad \mathtt{b} := \mathtt{Bool}(\bigwedge_{i=1..n} \mathit{mask\_match}(\mathtt{m}_i, \mathtt{u}_i))}{\mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathsf{Pattern\_Mask}(\mathtt{m})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{b}, \emptyset_{\mathsf{g}})}$$

## 17.7 Matching a Tuple of Patterns

### 17.7.1 Syntax

$\mathtt{pattern} \longrightarrow \mathsf{plist2}(\mathtt{pattern})$

### 17.7.2 Abstract Syntax

$\mathsf{pattern} \longrightarrow \mathsf{Pattern\_Tuple}(\mathsf{pattern}^*)$

**ASTRule.PTuple**

$$\frac{\mathit{build\_plist}[\mathit{build\_pattern}](\mathtt{patterns}) \xrightarrow{\mathsf{ast}} \mathtt{pattern\_asts}}{\mathit{build\_pattern}(\mathtt{pattern}(\mathtt{patterns} : \mathsf{plist2}(\mathtt{pattern}))) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{Pattern\_Tuple}(\mathtt{pattern\_asts})}^{\mathtt{ast\_node}}}$$

### 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`$^{/\!/\texttt{\#TE}}$;
- determining whether `t_struct` is a tuple type yields `TRUE`$^{/\!/\texttt{TE\_UT}}$;
- `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`$^{/\!/\texttt{TE\_UT}}$;
- annotating each pattern in `li` with the corresponding type in `t_s` for each index $i$ in the list of indices for `li`, yields $(\texttt{li'}[i], \texttt{xs}_i)^{/\!/\texttt{\#TE}}$;
- `new_li` is the list of annotated patterns `li'`[i] at the same positions those of `li`;
- `new_p` is the pattern which matches the tuple `new_li`, that is, Pattern_Tuple(`new_li`);
- define `ses` as the union of all $\texttt{xs}_i$, for each index $i$ in the list of indices for `li`.

### Formally

$$\frac{\begin{array}{c} \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathsf{T\_Tuple}, \texttt{TE\_UT}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathtt{t\_struct} \overset{\text{is}}{=} \mathsf{T\_Tuple}(\mathtt{t\_s}) \qquad \mathit{check}(\mathit{equal\_length}(\mathtt{li}, \mathtt{t\_s}), \texttt{TE\_UT}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ i \in \mathit{indices}(\mathtt{li}) : \mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t\_s}[i], \mathtt{li}[i]) \xrightarrow{\text{type}} (\mathtt{li'}[i], \mathtt{xs}_i) \;/\!/\; \texttt{\#TE} \\ \mathtt{new\_li} := i \in \mathit{indices}(\mathtt{li}) : \mathtt{li'}[i] \qquad \mathtt{ses} := \bigcup_{i \in \mathit{indices}(\mathtt{li})} \mathtt{xs}_i \end{array}}{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Tuple}(\mathtt{li})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Tuple}(\mathtt{new\_li})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

### 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 $1 \leq i \leq n$, $\mathtt{b}_i$ is the evaluation result of $\mathtt{p}_i$ with respect to the value $\mathtt{v}_i$ in environment `env`;
- `bs` is the list of all $\mathtt{b}_i$ for $1 \leq i \leq k$;
- `b` is the conjunction of the Boolean values of `bs`.

**Formally**

$$\frac{\begin{array}{c}\mathtt{ps} \overset{\text{is}}{=} \mathtt{p}_{1..k} \qquad i = 1..k : \mathit{get\_index}(i, \mathtt{v}) \xrightarrow{\text{eval}} \mathtt{vs}_i \\ i = 1..k : \mathit{eval\_pattern}(\mathtt{env}, \mathtt{vs}_i, \mathtt{p}_i) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathsf{Bool}(\mathtt{bs}_i), \mathtt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \\ \mathtt{res} := \mathsf{Bool}(\bigwedge_{i=1..k} \mathtt{bs}_i) \qquad \mathtt{g} := \mathtt{g}_1 \parallel \ldots \parallel \mathtt{g}_k\end{array}}{\mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathsf{Pattern\_Tuple}(\mathtt{ps})) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{res}, \emptyset_{\mathtt{g}})}$$

## 17.8 Matching Any Pattern in a Set of Patterns

### 17.8.1 Syntax

$$\begin{aligned}
\texttt{pattern} &\longrightarrow \texttt{pattern_set} \\
\texttt{pattern_set} &\longrightarrow \texttt{"!"}\ \texttt{"\{"}\ \texttt{pattern_list}\ \texttt{"\}"} \\
&\quad \mid \texttt{"\{"}\ \texttt{pattern_list}\ \texttt{"\}"} \\
\texttt{pattern_list} &\longrightarrow \mathsf{clist1}(\texttt{pattern})
\end{aligned}$$

### 17.8.2 Abstract Syntax

$$\texttt{pattern} \longrightarrow \mathsf{Pattern\_Any}(\texttt{pattern}^*)$$

#### ASTRule.PatternSet

The function

$$\mathit{build\_pattern\_set}(\overbrace{\mathbb{PARSE}[\texttt{pattern\_set}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{pattern}}^{\texttt{ast\_node}}$$

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

NOT

$$\mathit{build\_pattern\_set}(\texttt{pattern\_set}(\texttt{"!"}, \texttt{"\{"}, \texttt{pattern\_list}, \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Pattern\_Not}(\texttt{pattern\_list})}^{\texttt{ast\_node}}$$

LIST

$$\mathit{build\_pattern\_set}(\texttt{pattern\_set}(\texttt{"\{"}, \texttt{pattern\_list}, \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{pattern\_list}}^{\texttt{ast\_node}}$$

#### ASTRule.PatternList

The function

$$\mathit{build\_pattern\_list}(\overbrace{\mathbb{PARSE}[\texttt{pattern\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{pattern}}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_clist}[\mathit{build\_pattern}](\texttt{patterns}) \xrightarrow{\text{ast}} \texttt{pattern\_asts}}{\mathit{build\_pattern\_list}(\texttt{pattern\_list}(\texttt{patterns} : \mathsf{clist1}(\texttt{pattern}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{Pattern\_Any}(\texttt{pattern\_asts})}^{\texttt{ast\_node}}}$$

### 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 $(\texttt{new\_l}_1, \texttt{xs}_1)^{/\!/\texttt{\#TE}}$;
- define `new_li` as the list of patterns $\texttt{new\_l}_1$, for each `l` in `li`;
- `new_p` is the pattern which matches anything in `new_li`, that is, Pattern_Any(`new_li`);
- define `ses` as the union of all $\texttt{xs}_1$, for each `l` in `li`.

#### Formally

$$\frac{\texttt{l} \in \texttt{li} : \mathit{annotate\_pattern}(\mathsf{tenv}, \texttt{t}, \texttt{l}) \xrightarrow{\mathsf{type}} (\texttt{new\_l}_1, \texttt{xs}_1) \;/\!/\; \texttt{\#TE} \qquad \texttt{new\_li} := [\texttt{l} \in \texttt{li} : \texttt{new\_l}_1] \qquad \texttt{ses} := \bigcup_{\texttt{l} \in \texttt{li}} \texttt{xs}_1}{\mathit{annotate\_pattern}(\mathsf{tenv}, \texttt{t}, \overbrace{\mathsf{Pattern\_Any}(\texttt{li})}^{\texttt{p}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{Pattern\_Any}(\texttt{new\_li})}^{\texttt{new\_p}}, \texttt{ses})}$$

### 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 $\mathtt{p}_{1..k}$;
- evaluating each pattern $\mathtt{p}_i$ in `env` results in $\mathsf{Normal}(\mathtt{Bool}(\mathtt{b}_i), \mathtt{g}_i)^{/\!/\#\mathsf{T},\#\mathsf{DE}}$;
- `b` is the native Boolean which is the disjunction of $\mathtt{b}_i$, for $i = 1..k$;
- `new_g` is the parallel composition of all execution graphs $\mathtt{g}_i$, for $i = 1..k$.

**Formally**

$$\frac{\mathtt{ps} \overset{\mathsf{is}}{=} \mathtt{p}_{1..k} \qquad i = 1..k : \mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathtt{p}_i) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Bool}(\mathtt{b}_i), \mathtt{g}_i) \;/\!/\; \#\mathsf{DE} \quad \mathtt{b} := \mathtt{Bool}(\bigvee_{i=1..k} \mathtt{b}_i) \qquad \mathtt{new\_g} := \mathtt{g}_1 \parallel \ldots \parallel \mathtt{g}_k}{\mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathsf{Pattern\_Any}(\mathtt{ps})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{b}, \mathtt{new\_g})}$$

## 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 $(\texttt{new\_q}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- `new_p` is pattern which matches the negation of `new_q`, that is, Pattern_Not(`new_q`).

#### Formally

$$\frac{\mathit{annotate\_pattern}(\texttt{tenv}, \texttt{q}) \xrightarrow{\text{type}} (\texttt{new\_q}, \texttt{ses}) \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t}, \overbrace{\text{Pattern\_Not}(\texttt{q})}^{\texttt{p}}) \xrightarrow{\text{type}} (\overbrace{\text{Pattern\_Not}(\texttt{new\_q})}^{\texttt{new\_p}}, \texttt{ses})}$$

### 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`)$^{/\!/\texttt{\#DE}}$;
- `b` is the Boolean negation of `b1`.

**Formally**

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{p1}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{b1}, \texttt{new\_g}) \;/\!/\; \texttt{\#DE} \qquad \mathit{unop}(\texttt{BNOT}, \texttt{b1}) \xrightarrow{\text{eval}} \texttt{b}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \mathsf{Pattern\_Not}(\texttt{p1})) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{b}, \texttt{new\_g})}$$

# 17.10 AST Rules for Pattern Expressions

## 17.10.1 ASTRule.ExprPattern

The function

$$\mathit{build\_expr\_pattern}(\overbrace{\mathbb{PARSE}[\texttt{expr\_pattern}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\mathsf{expr}}^{\text{ast\_node}}$$

transforms a pattern expression parse node `parsed_node` into a pattern AST node `ast_node`.

LITERAL

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{value})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Literal}(\overline{\texttt{value}})}^{\text{ast\_node}}$$

VAR

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Var}(\texttt{id})}^{\text{ast\_node}}$$

BINOP

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{binop}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Binop}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{binop}}, \overline{\texttt{expr}})}^{\text{ast\_node}}$$

UNOP

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{unop}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Unop}(\overline{\texttt{unop}}, \overline{\texttt{expr}})}^{\text{ast\_node}}$$

COND

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{cond\_expr}) \xrightarrow{\text{ast}} \texttt{cond\_expr\_ast} \\ \mathit{build\_expr}(\texttt{then\_expr}) \xrightarrow{\text{ast}} \texttt{then\_expr\_ast} \\ \mathit{build\_expr}(\texttt{else\_expr}) \xrightarrow{\text{ast}} \texttt{else\_expr\_ast}\end{array}}{\mathit{build\_expr\_pattern}\left(\texttt{expr\_pattern}\left(\begin{array}{l}\texttt{"if"}, \texttt{cond\_expr} : \texttt{expr}, \texttt{"then"}, \\ \hookrightarrow\ \texttt{then\_expr} : \texttt{expr}, \texttt{"else"}, \texttt{else\_expr} : \texttt{expr}\end{array}\right)\right) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Cond}(\texttt{cond\_expr\_ast}, \texttt{then\_expr\_ast}, \texttt{else\_expr\_ast})}^{\texttt{ast\_node}}}$$

CALL

$$\frac{\mathit{build\_plist}[\mathit{build\_expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{expr\_asts}}{\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{call})) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Call}(\overline{\texttt{call}})}^{\texttt{ast\_node}}}$$

SLICE

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{slice})) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Slice}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{slice}})}^{\texttt{ast\_node}}$$

SET_ARRAY

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"[["}, \texttt{expr}, \texttt{"]]"})) \xrightarrow{\text{ast}} \overbrace{\texttt{LE\_SetArray}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

GET_FIELD

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"."}, \textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_GetField}(\overline{\texttt{expr}}, \texttt{id})}^{\texttt{ast\_node}}$$

GET_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{id\_asts}}{\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"."}, \texttt{"["}, \texttt{ids} : \texttt{clist1}(\textbf{ID}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_GetFields}(\overline{\texttt{expr\_pattern}}, \texttt{id\_asts})}^{\texttt{ast\_node}}}$$

ATC

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"as"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_ATC}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

ATC_INT_CONSTRAINTS

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"as"}, \texttt{constraint\_kind})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_ATC}(\overline{\texttt{expr\_pattern}}, \mathsf{T\_Int}(\overline{\texttt{constraint\_kind}}))}^{\texttt{ast\_node}}$$

PATTERN_IN

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"IN"}, \texttt{pattern\_set})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{pattern\_set}})}^{\texttt{ast\_node}}$$

PATTERN_EQ

$$\mathit{build\_expr\_pattern}(\overbrace{\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"="}, \mathbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr\_pattern}}, \mathsf{Pattern\_Mask}(\texttt{m}))}^{\texttt{ast\_node}}$$

PATTERN_NEQ

$$\mathit{build\_expr\_pattern}(\overbrace{\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"!="}, \mathbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr\_pattern}}, \mathsf{Pattern\_Not}(\mathsf{Pattern\_Mask}(\texttt{m})))}^{\texttt{ast\_node}}$$

ARBITRARY

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{"ARBITRARY"}, \texttt{":"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Arbitrary}(\overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

RECORD_EMPTY

$$\mathit{build\_expr\_pattern}\left(\texttt{expr\_pattern}\left(\ \mathbf{ID}(\texttt{t}), \texttt{"\{"}, \texttt{"-"}, \texttt{"\}"}\ \right)\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Record}(\mathsf{T\_Named}(\texttt{t}), [\,])}^{\texttt{ast\_node}}$$

RECORD_NON_EMPTY

$$\frac{\mathit{build\_clist}[\mathit{build\_field\_assign}](\texttt{field\_assigns}) \xrightarrow{\text{ast}} \texttt{field\_assign\_asts}}{\mathit{build\_expr\_pattern}\left(\texttt{expr\_pattern}\left(\begin{array}{l}\mathbf{ID}(\texttt{t}), \texttt{"\{"},\\ \hookrightarrow\ \texttt{field\_assigns} : \mathsf{clist0}(\texttt{field\_assign}),\\ \hookrightarrow\ \texttt{"\}"}\end{array}\right)\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Record}(\mathsf{T\_Named}(\texttt{t}), \texttt{field\_assign\_asts})}^{\texttt{ast\_node}}}$$

SUB_EXPR

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{"("}, \texttt{expr\_pattern}, \texttt{")"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Tuple}([\ \overline{\texttt{expr\_pattern}}\ ])}^{\texttt{ast\_node}}$$

# Chapter 18

# Assignable Expressions

We refer to expressions that may appear on the left hand side of an assignment statement as assignable expressions. An assignable expression is grammatically derived from `lexpr` and is represented as an AST by lexpr.

We show the syntax relevant to assignable expressions in Section 18.1 and the rules need to build the AST for assignable expressions in Section 18.1.1. These rules rely on three further desugaring relations, defined in Section 18.1.2. We then define the abstract syntax, typing, and semantics of the different kinds of assignable expressions:

- Discarding assignment expressions (see Section 18.2)
- Variable assignment expressions (see Section 18.3)
- Multi-assignment expressions (see Section 18.4)
- Array assignment expressions (see Section 18.5)
- Bitvector slice assignment expressions (see Section 18.6)
- Structured type field assignment expressions (Section 18.7)
- Structured type multi-field assignment expressions (Section 18.8)
- Bitfield assignment expressions (see Section 18.9)

The function

$$\mathit{annotate\_lexpr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{lexpr}}^{\texttt{le}}, \overbrace{\mathsf{ty}}^{\texttt{t\_e}}) \longrightarrow (\overbrace{\mathsf{lexpr}}^{\texttt{new\_le}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \mathsf{TTypeError}$$

annotates a left-hand side expression `le` in an environment `tenv`, assuming `t_e` to be the type of the corresponding right-hand-side expression, resulting in an annotated expression `new_le` and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

The relation

$$\mathit{eval\_lexpr}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{lexpr}}^{\texttt{le}}, (\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}})) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\#\mathsf{T}} \cup \overbrace{\mathsf{TDynError}}^{\#\mathsf{DE}}$$

evaluates the assignment of a value `v` to the left-hand-side expression `le` in an environment `env`, resulting in either a configuration Normal(`new_g`, `env`) or an abnormal configuration.

**Semantics Rules Naming Convention:** In this chapter, variables containing `m` range over $\mathbb{V} \times \mathcal{G}$ while variables where the `m` is replaced with `v` correspond to their value component. For example, `rm_array` $\overset{\text{is}}{=}$ (`rv_array`, `g2`) and `m_index` $\overset{\text{is}}{=}$ (`index`, `g1`).

**Viewing Assignable Expressions as Right-hand-side Expressions:** Some of the typing rules and semantics rules require viewing assignable expressions as right-hand-side expressions. The correspondence is given by the function *rexpr* : lexpr → expr., defined in Section 8.7. For example, SemanticsRule.LESetField needs to evaluate the record subexpression `re_record`, which is an assignable expression. To achieve this, *rexpr*(`record`) is used to obtain an right-hand-side expression, which then allows using `eval_expr` to evaluate it.

## 18.1 Syntax

```
lexpr ⟶ "-"
        | 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.

$$\text{lhs\_access} \longrightarrow \left\{ \begin{array}{rcl} \text{access} & : & \text{field\_or\_array\_access}^*, \\ \text{slices} & : & \text{slice}^* \end{array} \right\}$$

$$\text{field\_or\_array\_access} \longrightarrow \text{FieldAccess}(\text{identifier}) \mid \text{ArrayAccess}(\text{expr})$$

#### ASTRule.LExpr

The function

$$\mathit{build\_lexpr}(\overbrace{\mathbb{PARSE}[\texttt{lexpr}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\text{lexpr}}^{\texttt{ast\_node}}$$

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

DISCARD

$$\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{LE\_Discard}}^{\texttt{ast\_node}}$$

BASIC_LEXPR

$$\frac{\mathit{desugar\_lhs\_access}(\overline{\texttt{basic\_lexpr}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{basic\_lexpr})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

MULTI_LEXPR

$$\frac{\begin{array}{c}\mathit{build\_clist}[\mathit{build\_discard\_or\_basic\_lexpr}](\texttt{lexprs}) \xrightarrow{\text{ast}} \texttt{lexpr\_asts} \\ \mathit{desugar\_lhs\_tuple}(\texttt{lexpr\_asts}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\begin{array}{r}\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{"("}, \texttt{lexprs} : \text{clist2}(\texttt{discard\_or\_basic\_lexpr}), \texttt{")"})) \\ \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}$$

CONCAT_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\texttt{fields}) \xrightarrow{\text{ast}} \texttt{field\_asts}}{\mathit{build\_lexpr}(\texttt{lexpr}(\textbf{ID}(\texttt{id}), \texttt{"."}, \texttt{"["}, \texttt{fields} : \text{clist2}(\textbf{ID}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\text{LE\_SetFields}(\text{LE\_Var}(\texttt{id}), \texttt{field\_asts})}^{\texttt{ast\_node}}}$$

TUPLE_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_discard\_or\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast} \quad \mathit{desugar\_lhs\_fields\_tuple}(\texttt{id}, \texttt{ids\_ast}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_lexpr}(\texttt{lexpr}(\textbf{ID}(\texttt{id}), \texttt{"."}, \texttt{"("}, \texttt{ids} : \mathsf{clist2}(\texttt{discard\_or\_identifier}), \texttt{")"})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

### ASTRule.BasicLexpr

The function

$$\mathit{build\_basic\_lexpr}(\overbrace{\mathbb{PARSE}[\texttt{basic\_lexpr}]}^{\texttt{parsed\_node}}) \longrightarrow (\overbrace{\mathsf{identifier}}^{\texttt{base}}, \overbrace{\mathsf{lhs\_access}}^{\texttt{lhs\_access}})$$

transforms a parse node `parsed_node` into a pair of AST nodes, `base` and `lhs_access`.

NO_SLICES

$$\frac{\texttt{lhs\_access} := \left\{ \begin{array}{rcl} \text{access} & : & \overline{\texttt{access}}, \\ \text{slices} & : & [\,] \end{array} \right\}}{\mathit{build\_basic\_lexpr}(\texttt{basic\_lexpr}(\textbf{ID}(\texttt{base}), \texttt{access})) \xrightarrow{\text{ast}} (\texttt{base}, \texttt{lhs\_access})}$$

SLICES

$$\frac{\texttt{lhs\_access} := \left\{ \begin{array}{rcl} \text{access} & : & \overline{\texttt{access}}, \\ \text{slices} & : & \overline{\texttt{slices}} \end{array} \right\}}{\mathit{build\_basic\_lexpr}(\texttt{basic\_lexpr}(\textbf{ID}(\texttt{base}), \texttt{access}, \texttt{slices})) \xrightarrow{\text{ast}} (\texttt{base}, \texttt{lhs\_access})}$$

### ASTRule.Access

The function

$$\mathit{build\_access}(\overbrace{\mathbb{PARSE}[\texttt{access}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{field\_or\_array\_access}^{*}}^{\texttt{ast\_node}}$$

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

EMPTY

$$\mathit{build\_access}(\texttt{access}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{ast\_node}}$$

FIELD_ACCESS

$$\mathit{build\_access}(\texttt{access}(\texttt{"."}, \textbf{ID}(\texttt{field}), \texttt{access})) \xrightarrow{\text{ast}} \overbrace{[\mathsf{FieldAccess}(\texttt{field})] + \overline{\texttt{access}}}^{\texttt{ast\_node}}$$

ARRAY_ACCESS

$$\mathit{build\_access}(\texttt{access}(\texttt{"[["}, \texttt{expr}, \texttt{"]]"}, \texttt{access})) \xrightarrow{\text{ast}} \overbrace{[\mathsf{ArrayAccess}(\overline{\texttt{expr}})] + \overline{\texttt{access}}}^{\texttt{ast\_node}}$$

**ASTRule.DiscardOrBasicLexpr**

The function

$$\mathit{build\_discard\_or\_basic\_lexpr}(\overbrace{\mathbb{PARSE}[\texttt{discard\_or\_basic\_lexpr}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle \mathsf{lhs\_access} \rangle}^{\texttt{ast\_node}}$$

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

DISCARD

$$\mathit{build\_discard\_or\_basic\_lexpr}(\texttt{discard\_or\_basic\_lexpr}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{None}}^{\texttt{ast\_node}}$$

BASIC

$$\mathit{build\_discard\_or\_basic\_lexpr}(\texttt{discard\_or\_basic\_lexpr}(\texttt{basic\_lexpr})) \xrightarrow{\text{ast}} \overbrace{\langle \texttt{basic\_lexpr} \rangle}^{\texttt{ast\_node}}$$

**ASTRule.DiscardOrIdentifier**

The function

$$\mathit{build\_discard\_or\_identifier}(\overbrace{\mathbb{PARSE}[\texttt{discard\_or\_identifier}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle \mathsf{identifier} \rangle}^{\texttt{ast\_node}}$$

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

NONE

$$\mathit{build\_discard\_or\_identifier}(\texttt{discard\_or\_identifier}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{None}}^{\texttt{ast\_node}}$$

SOME

$$\mathit{build\_discard\_or\_identifier}(\texttt{discard\_or\_identifier}(\textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\langle \texttt{id} \rangle}^{\texttt{ast\_node}}$$

### 18.1.2 Desugaring Assignable Expressions

This section defines three desugaring relations which produce assignable expression abstract syntax, that is, lexpr. They are used in Section 18.1.1 to build lexpr abstract syntax.

- *desugar_lhs_access*, which desugars 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

$$\mathit{desugar\_lhs\_access}(\overbrace{\text{identifier}}^{\texttt{name}}, \overbrace{\text{lhs\_access}}^{\texttt{lhs\_access}}) \longrightarrow \overbrace{\text{lexpr}}^{\texttt{lexpr}}$$

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;
```

$$\frac{\begin{array}{c}\texttt{lexprs}_0 := \text{LE\_Var}(\texttt{name}) \\ i \in 1..|\texttt{access}| : \texttt{lexprs}_i := \begin{cases} \text{LE\_SetField}(\texttt{lexprs}_{i-1}, \texttt{field}) \\ \quad \text{if } \texttt{access}_i = \text{FieldAccess}(\texttt{field}) \\ \text{LE\_SetArray}(\texttt{lexprs}_{i-1}, \texttt{expr}) \\ \quad \text{if } \texttt{access}_i = \text{ArrayAccess}(\texttt{expr}) \end{cases} \\ \texttt{lexpr} := \text{choice}(\texttt{slices} = [\,], \texttt{lexprs}_{|\texttt{access}|}, \text{LE\_Slice}(\texttt{lexprs}_{|\texttt{access}|}, \texttt{slices})) \end{array}}{\mathit{desugar\_lhs\_access}\overbrace{\left\{\begin{array}{rcl}\text{access} & : & \texttt{access}, \\ \text{slices} & : & \texttt{slices}\end{array}\right\}}^{\texttt{lhs\_access}} \xrightarrow{\text{ast}} \texttt{lexpr}}$$

### ASTRule.DesugarLHSTuple

The function

$$\mathit{desugar\_lhs\_tuple}(\overbrace{\langle \text{lhs\_access} \rangle^*}^{\texttt{lhs\_access\_opts}}) \longrightarrow \overbrace{\text{lexpr}}^{\texttt{lexpr}}$$

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;
```

$$\frac{i \in 1..|\texttt{lhs\_access\_opts}| : \mathit{desugar\_lhs\_access\_opt}(\texttt{lhs\_access\_opt}_i) \xrightarrow{\text{ast}} \texttt{lexpr}_i \qquad \texttt{lexprs} := [i \in 1..|\texttt{lhs\_access\_opts}| : \texttt{lexpr}_i]}{\mathit{desugar\_lhs\_tuple}(\texttt{lhs\_access\_opts}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Destructuring}(\texttt{lexprs})}^{\texttt{lexpr}}}$$

### ASTRule.DesugarLHSAccessOpt

The helper AST function

$$\mathit{desugar\_lhs\_access\_opt}(\overbrace{\langle \mathsf{lhs\_access} \rangle}^{\texttt{lhs\_access\_opt}}) \longrightarrow \overbrace{\mathsf{lexpr}}^{\texttt{lexpr}}$$

is defined via the following rules:

NONE

$$\mathit{desugar\_lhs\_access\_opt}(\overbrace{\quad\mathsf{None}\quad}^{\texttt{lhs\_access\_opt}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Discard}}^{\texttt{lexpr}}$$

SOME

$$\frac{\mathit{desugar\_lhs\_access}(\texttt{lhs\_access}) \xrightarrow{\text{ast}} \texttt{lexpr}}{\mathit{desugar\_lhs\_access\_opt}(\overbrace{\langle \texttt{lhs\_access} \rangle}^{\texttt{lhs\_access\_opt}}) \xrightarrow{\text{ast}} \texttt{lexpr}}$$

### ASTRule.DesugarLHSFieldsTuple

The function

$$\mathit{desugar\_lhs\_fields\_tuple}(\overbrace{\mathsf{identifier}}^{\texttt{id}}, \overbrace{\langle \mathsf{identifier} \rangle^*}^{\texttt{field\_opts}}) \longrightarrow \overbrace{\mathsf{lexpr}}^{\texttt{lexpr}}$$

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;
```

$$\frac{\begin{array}{c}\mathit{filter\_option\_list}(\texttt{field\_opts}) \longrightarrow \texttt{fields} \\ \mathit{check\_no\_duplicates}(\texttt{fields}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ i \in 1..|\texttt{field\_opts}| : \mathit{desugar\_lhs\_field\_opt}(\texttt{id}, \texttt{field\_opts}[i]) \xrightarrow{\text{ast}} \texttt{lexpr}_i \\ \texttt{lexprs} := [i \in 1..|\texttt{field\_opts}| : \texttt{lexpr}_i]\end{array}}{\mathit{desugar\_lhs\_fields\_tuple}(\texttt{id}, \texttt{field\_opts}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Destructuring}(\texttt{lexprs})}^{\texttt{lexpr}}}$$

### ASTRule.FilterOptionList

The parametric function

$$\mathit{filter\_option\_list}(\overbrace{\langle T \rangle^*}^{\texttt{v\_opts}}) \longrightarrow \overbrace{T^*}^{\texttt{vs}}$$

filters a list of optional elements, removing those which are `None` and unwrapping those which are $\langle v \rangle$ to $v$.

### 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 $[\langle \texttt{fld1} \rangle, \texttt{None}, \langle \texttt{fld2} \rangle]$. Before checking the list `fld1, -, fld2` for absence of duplicates, it is filtered as follows:
$\mathit{filter\_option\_list}([\langle \texttt{fld1} \rangle, \texttt{None}, \langle \texttt{fld2} \rangle]) \longrightarrow [\texttt{fld1}, \texttt{fld2}]$.

### 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 *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 *filter_option_list* to `v_opts'` yields `vs'`;
  - `vs` is the concatenation of `v` and `vs'`.

### Formally

EMPTY

$$\mathit{filter\_option\_list}(\overbrace{[\,]}^{\texttt{v\_opts}}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\texttt{vs}}$$

NON_EMPTY_NONE

$$\frac{\mathit{filter\_option\_list}(\texttt{v\_opts'}) \xrightarrow{\text{type}} \texttt{vs}}{\mathit{filter\_option\_list}(\overbrace{[\texttt{None}] + \texttt{v\_opts'}}^{\texttt{v\_opts}}) \xrightarrow{\text{type}} \texttt{vs}}$$

NON_EMPTY_SOME

$$\frac{\mathit{filter\_option\_list}(\texttt{v\_opts'}) \xrightarrow{\text{type}} \texttt{vs'}}{\mathit{filter\_option\_list}(\overbrace{[\langle \texttt{v} \rangle] + \texttt{v\_opts'}}^{\texttt{v\_opts}}) \xrightarrow{\text{type}} \overbrace{[\texttt{v}] + \texttt{vs'}}^{\texttt{vs}}}$$

**ASTRule.DesugarLHSFieldOpt**

The helper AST function

$$\mathit{desugar\_lhs\_field\_opt}(\overbrace{\mathsf{identifier}}^{\texttt{id}}, \overbrace{\langle \mathsf{identifier} \rangle}^{\texttt{field\_opt}}) \longrightarrow \overbrace{\mathsf{lexpr}}^{\texttt{lexpr}}$$

is defined via the following rules:

NONE

$$\mathit{desugar\_lhs\_field\_opt}(\texttt{id}, \overbrace{\mathsf{None}}^{\texttt{field\_opt}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Discard}}^{\texttt{lexpr}}$$

$$\mathit{desugar\_lhs\_field\_opt}(\texttt{id}, \overbrace{\langle \texttt{field} \rangle}^{\texttt{field\_opt}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_SetField}(\mathsf{LE\_Var}(\texttt{id}), \texttt{field})}^{\texttt{lexpr}}$$

## 18.2 Discarding Assignment Expressions

### 18.2.1 Abstract Syntax

$$\mathsf{lexpr} \longrightarrow \overbrace{\mathsf{LE\_Discard}}^{\texttt{"-"}}$$

### 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**

$$\mathit{annotate\_lexpr}(\text{tenv}, \overbrace{\text{LE\_Discard}}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE\_Discard}}^{\texttt{new\_le}}, \overbrace{\emptyset}^{\texttt{ses}})$$

### 18.2.3 Semantics

**SemanticsRule.LEDiscard**

**Example: 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**

$$\frac{\texttt{new\_g} := \texttt{g} \qquad \texttt{new\_env} := \texttt{env}}{\mathit{eval\_lexpr}(\texttt{env}, \text{LE\_Discard}, (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

## 18.3 Variable Assignment Expressions

### 18.3.1 Abstract Syntax

lexpr $\longrightarrow$ LE_Var(identifier)

### 18.3.2 Typing

**TypingRule.LEVar**

**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$^{\#\text{TE_AIM}}$;
  - determining whether `ty` type-satisfies `t_e` in tenv yields TRUE$^{\#\#\text{TE}}$;
  - `new_le` is `le`;
  - define `ses` as the local write side effect descriptor for `x`.

- 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 $\texttt{TRUE}^{/\!/\texttt{TE\_AIM}}$;
  - determining whether `ty` type-satisfies `t_e` in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - `new_le` is `le`;
  - define `ses` as the global write side effect descriptor for `x`.
- 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

$$\frac{\begin{array}{c} L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{x}) = (\mathtt{ty}, k) \\ \mathit{check}(k = \mathsf{LDK\_Var}, \mathtt{TE\_AIM}) \longrightarrow \mathtt{TRUE}\ /\!/\ \mathtt{\#TE} \\ \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}\ /\!/\ \mathtt{\#TE} \end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_Var}(\mathtt{x})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\overbrace{\mathtt{le}}^{\mathtt{new\_le}}, \{\mathsf{WriteLocal}(\mathtt{x})\})}$$

GLOBAL

$$\frac{\begin{array}{c} L^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{x}) = (\mathtt{ty}, k) \\ \mathit{check}(k = \mathsf{GDK\_Var}, \mathtt{AssignToImmutable}) \longrightarrow \mathtt{TRUE}\ /\!/\ \mathtt{\#TE} \\ \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}\ /\!/\ \mathtt{\#TE} \end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_Var}(\mathtt{x})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\overbrace{\mathtt{le}}^{\mathtt{new\_le}}, \{\mathsf{WriteGlobal}(\mathtt{x})\})}$$

ERROR_UNDEFINED

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{x}) = \bot \qquad L^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{x}) = \bot}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_Var}(\mathtt{x})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UI})}$$

### 18.3.3 Semantics

**SemanticsRule.LEVar**

**Example: Local Variable**

In 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 ($L^{\mathsf{denv}}$);
    - `new_env` is `env` where `x` is bound to `v` in the local dynamic environment ($L^{\mathsf{denv}}$).
  - All of the following apply (GLOBAL):
    - `x` is bound in the global dynamic environment ($G^{\mathsf{denv}}.\mathsf{storage}$);
    - `new_env` is `env` where `x` is bound to `v` in the storage map of the global dynamic environment $G^{\mathsf{denv}}$.
- `new_g` is the ordered composition of `g` and a Write Effect for `x` with the `asl_data` edge;

**Formally**

LOCAL

$$\frac{\texttt{env} \overset{\text{is}}{=} (\texttt{tenv}, \texttt{denv}) \qquad \texttt{x} \in \mathrm{dom}(L^{\texttt{denv}}) \qquad \texttt{new\_env} := (\texttt{tenv}, (G^{\texttt{denv}}, L^{\texttt{denv}}[\texttt{x} \mapsto \texttt{v}])) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \mathsf{WriteEffect}(\texttt{x})}{\mathit{eval\_lexpr}(\texttt{env}, \mathsf{LE\_Var}(\texttt{x}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

GLOBAL

$$\frac{\texttt{env} \overset{\text{is}}{=} (\texttt{tenv}, \texttt{denv}) \qquad \texttt{x} \in \mathrm{dom}(G^{\texttt{denv}}.\mathsf{storage}) \qquad \texttt{new\_env} := (\texttt{tenv}, (G^{\texttt{denv}}.\mathsf{storage}[\texttt{x} \mapsto \texttt{v}], L^{\texttt{denv}})) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \mathsf{WriteEffect}(\texttt{x})}{\mathit{eval\_lexpr}(\texttt{env}, \mathsf{LE\_Var}(\texttt{x}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

## 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;

  return 0;
end;
```

### 18.4.1 Abstract Syntax

$\mathsf{lexpr} \longrightarrow \mathsf{LE\_Destructuring}(\mathsf{lexpr}^*)$

### 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, $\mathsf{LE\_Destructuring}(\texttt{les})$;
- `les` is a list $\texttt{e}_{1..k}$;

- checking whether `t_e` is a tuple type yields $\texttt{TRUE}^{/\!/\texttt{TE\_UT}}$;
- `t_e` is a tuple type over the list of types `tys`, that is, T_Tuple(`tys`);
- determining whether `les` and `sub_tys` have the same length yields $\texttt{TRUE}^{/\!/\texttt{TE\_UT}}$;
- `sub_tys` is the list of types $\texttt{t}_{1..k}$;
- annotating the left-hand-side expression $\texttt{e}_i$ with the type $\texttt{t}_i$, for $i = 1..k$, yields $(\texttt{e'}_i, \texttt{xs}_i)^{/\!/\texttt{\#TE}}$;
- the list of expressions `les'` is $\texttt{e'}_i$, for $i = 1..k$;
- `new_le` is the list of left-hand-side expressions `les'`, that is, LE_Destructuring(`les'`);
- define `ses` as the union of all sets $\texttt{xs}_i$, for $i = 1..k$.

**Formally**

$$\frac{\begin{array}{c}\texttt{les} \overset{\text{is}}{=} [\texttt{e}_{1..k}] \qquad \mathit{check}(\mathit{ast\_label}(\texttt{t\_e}) = \text{T\_Tuple}, \texttt{TE\_UT}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e} \overset{\text{is}}{=} \text{T\_Tuple}(\texttt{tys}) \\ \mathit{equal\_length}(\texttt{les}, \texttt{tys}) \xrightarrow{\text{type}} \texttt{b} \qquad \mathit{check}(\texttt{b}, \texttt{TE\_UT}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{tys} \overset{\text{is}}{=} [\texttt{t}_{1..k}] \qquad i = 1..k : \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{e}_i, \texttt{t}_i) \xrightarrow{\text{type}} (\texttt{e'}_i, \texttt{xs}_i) \;/\!/\; \texttt{\#TE} \\ \texttt{les'} \overset{\text{is}}{=} [i = 1..k : \texttt{e'}_i] \qquad \texttt{ses} := \bigcup_{i=1..k} \texttt{xs}_i\end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\text{LE\_Destructuring}(\texttt{les})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE\_Destructuring}(\texttt{les'})}^{\texttt{new\_le}}, \texttt{ses})}$$

### 18.4.3 Semantics

**SemanticsRule.LEDestructuring**

**Example: Multi-assignments**

In Listing 18.10, the multi-assignment `(x, y) = (3, 42)` binds `x` to `Int`(3) and `y` to `Int`(42) in the environment where `x` is bound to `Int`(42) and `y` is bound to `Int`(3).

**Prose**

All of the following apply:

- `le` denotes a list of left-hand-side expressions, LE_Destructuring(`le_list`);
- `le_list` is the list of expressions $\texttt{le}_{1..n}$;
- getting the values from the native vector `v` at each index $i = 1..n$ results in $\texttt{v}_{i=1..n}$;
- `nmonads` is the list of pairs consisting of $\texttt{v}_i$ and `g` for $i = 1..n$;
- evaluating the multi-assignment between `le_list` and the list `nmonads` in `env` achieves the effects of assigning each value to the respective subexpressions, resulting in the output configuration $C$.

**Formally**

$$\frac{\texttt{le\_list} \overset{\text{is}}{=} [\texttt{le}_{1..n}] \qquad i = 1..n : \mathit{get\_index}(i, \texttt{v}) \xrightarrow{\text{eval}} \texttt{v}_i \qquad \texttt{nmonads} := [i = 1..n : (\texttt{v}_i, \texttt{g})] \qquad \mathit{multi\_assign}(\texttt{env}, \texttt{le\_list}, \texttt{nmonads}) \xrightarrow{\text{eval}} C}{\mathit{eval\_lexpr}(\texttt{env}, \mathsf{LE\_Destructuring}(\texttt{le\_list}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

**SemanticsRule.LEMultiAssign**

The helper relation

$$\mathit{multi\_assign}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{expr}^*}^{\texttt{le\_list}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vm\_list}}) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\#\text{T}} \cup \overbrace{\mathsf{TDynError}}^{\#\text{DE}}$$

evaluates multi-assignments. That is, the simultaneous assignment of the list of value-execution graph pairs `vm_list` to the corresponding list of left-hand side expressions `le_list`, in the environment `env`. The result is either the execution graph `g` and new environment `new_env` or an abnormal configuration.

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 $\mathsf{Normal}(\texttt{env1}, \texttt{g1})^{/\!/\#\text{T},\#\text{DE}}$;
  - applying *multi_assign* to `env1`, `le_list1`, and `vm_list1` yields $\mathsf{Normal}(\texttt{new\_env}, \texttt{g2})^{/\!/\#\text{T},\#\text{DE}}$;
  - define `new_g` as the ordered composition of `g1` and `g2` with the edge `asl_po`.

**Formally**

EMPTY

$$\mathit{multi\_assign}(\texttt{env}, \overbrace{[\,]}^{\texttt{le\_list}}, \overbrace{[\,]}^{\texttt{vm\_list}}) \xrightarrow{\text{eval}} \mathsf{Normal}(\overbrace{\emptyset_{\mathsf{g}}}^{\texttt{new\_g}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

$$\frac{\begin{array}{c}\text{NON\_EMPTY}\\ \texttt{le\_list} = [\texttt{le}] + \texttt{le\_list1}\\ \texttt{vm\_list} = [\texttt{m}] + \texttt{vm\_list1} \qquad \mathit{eval\_lexpr}(\texttt{env}, \texttt{le}, \texttt{m}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{env1}, \texttt{g1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE}\\ \mathit{multi\_assign}(\texttt{env1}, \texttt{le\_list1}, \texttt{vm\_list1}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{new\_env}, \texttt{g2}) \;/\!/\; \texttt{\#T}, \texttt{\#DE}\\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_po}} \texttt{g2}\end{array}}{\mathit{multi\_assign}(\texttt{env}, \texttt{le\_list}, \texttt{vm\_list}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

Notice that this rule is only defined when the lists `le_list` and `vm_list` have the same length. To see this, notice that to form a derivation tree, we must employ the NONEMPTY case, which ensures both lists have at least one element and shortens the lengths of both lists by one, until both lists become empty which is when the EMPTY axiom case is used.

## 18.5 Array Assignment Expressions

This section details the syntax, abstract syntax, semantics, and typing of array write expressions. In the untyped AST, a write to either an integer-indexed array or an enumeration-indexed array is represented the same way. 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

$$\begin{aligned}\mathsf{lexpr} \longrightarrow\ & \mathsf{LE\_SetArray}(\mathsf{lexpr}, \mathsf{expr})\\ \mid\ & \mathsf{LE\_SetEnumArray}(\overbrace{\mathsf{lexpr}}^{\text{base}}, \overbrace{\mathsf{expr}}^{\text{index}})\end{aligned}$$

### 18.5.2 Typing

#### TypingRule.LESetArray

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 $(\texttt{t\_base}, \_, \_)^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_base` in `tenv` yields $\texttt{t\_anon\_base}^{/\!/\texttt{\#TE}}$;
- checking that `t_anon_base` is an array type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- view `t_anon_base` as an array type of size `size` and element type `t_elem`, that is, $\texttt{T\_Array}(\texttt{size}, \texttt{t\_elem})$;
- annotating the left-hand-side expression `e_base` with type `t_base` in `tenv` yields $(\texttt{e\_base'}, \texttt{ses\_base})^{/\!/\texttt{\#TE}}$;
- applying *annotate_set_array* to $(\texttt{size}, \texttt{t\_elem})$, `t_e`, and $(\texttt{e\_base'}, \texttt{ses\_base}, \texttt{e\_index})$ in `tenv` yields $(\texttt{new\_le}, \texttt{ses})^{/\!/\texttt{\#TE}}$.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \mathit{rexpr}(\texttt{e\_base})) \xrightarrow{\text{type}} (\texttt{t\_base}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_base}) \xrightarrow{\text{type}} \texttt{t\_anon\_base} \;/\!/\; \texttt{\#TE} \\ \mathit{check}\,(\mathit{ast\_label}(\texttt{t\_anon\_base}) = \texttt{T\_Array}, \texttt{ExpectedArrayType}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_anon\_base} \overset{\text{is}}{=} \texttt{T\_Array}(\texttt{size}, \texttt{t\_elem}) \\ \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{e\_base}, \texttt{t\_base}) \xrightarrow{\text{type}} (\texttt{e\_base'}, \texttt{ses\_base}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_set\_array}(\texttt{tenv}, (\texttt{size}, \texttt{t\_elem}), \texttt{t\_e}, (\texttt{e\_base'}, \texttt{ses\_base}, \texttt{e\_index})) \xrightarrow{\text{type}} \\ (\texttt{new\_le}, \texttt{ses}) \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\texttt{LE\_SetArray}(\texttt{e\_base}, \texttt{e\_index})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

**TypingRule.AnnotateSetArray**

The helper function

$$\mathit{annotate\_set\_array}\left(\begin{array}{c}\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \\ (\overbrace{\texttt{array\_index}}^{\texttt{size}} \times \overbrace{\texttt{ty}}^{\texttt{t\_elem}}), \\ \overbrace{\texttt{ty}}^{\texttt{rhs\_ty}}, \\ (\overbrace{\texttt{expr}}^{\texttt{e\_base}} \times \overbrace{\mathcal{P}(\texttt{TSideEffect})}^{\texttt{ses\_base}} \times \overbrace{\texttt{expr}}^{\texttt{e\_index}})\end{array}\right) \longrightarrow (\overbrace{\texttt{lexpr}}^{\texttt{new\_le}} \times \overbrace{\mathcal{P}(\texttt{TSideEffect})}^{\texttt{ses}})$$

annotates an array update in the static environment `tenv` where `size` is the array index, `t_elem` is the type of array elements, `rhs_ty` is the type of the right-hand-side expression, `e_base` is the annotated expression for the array base, `ses_base` is the set of side effect descriptors inferred for `e_base`, and `e_index` is the index expression. The result is the

annotated assignable expression `new_le` and set of side effect descriptors for the annotated expression `ses`. $/\!/^{\#TE}$

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 $\texttt{TRUE}^{/\!/\#TE}$;
- annotating the index expression `e_index` in `tenv` yields $(\texttt{t\_index'}, \texttt{e\_index'}, \texttt{ses\_index})^{/\!/\#TE}$;
- 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 $\texttt{TRUE}^{/\!/\#TE}$;
- 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

$$\frac{\begin{array}{c}\mathit{checked\_typesat}(\texttt{tenv}, \texttt{rhs\_ty}, \texttt{t\_elem}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e\_index}) \xrightarrow{\text{type}} (\texttt{t\_index'}, \texttt{e\_index'}, \texttt{ses\_index}) \;/\!/\; \texttt{\#TE} \\ \mathit{type\_of\_array\_length}(\texttt{tenv}, \texttt{size}) \xrightarrow{\text{type}} \texttt{wanted\_t\_index} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_index'}, \texttt{wanted\_t\_index}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_base} \cup \texttt{ses\_index} \qquad \texttt{new\_le\_int} := \text{LE\_SetArray}(\texttt{e\_base'}, \texttt{e\_index'}) \\ \texttt{new\_le\_enum} := \text{LE\_SetEnumArray}(\texttt{e\_base'}, \texttt{e\_index'}) \\ \texttt{new\_le} := \begin{cases} \texttt{new\_le\_int} & \text{if } \mathit{ast\_label}(\texttt{size}) = \text{ArrayLength\_Expr} \\ \texttt{new\_le\_enum} & \text{if } \mathit{ast\_label}(\texttt{size}) = \text{ArrayLength\_Enum} \end{cases}\end{array}}{\mathit{annotate\_set\_array}(\texttt{tenv}, (\texttt{size}, \texttt{t\_elem}), \texttt{rhs\_ty}, (\texttt{e\_base}, \texttt{ses\_base}, \texttt{e\_index})) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

### 18.5.3 Semantics

#### SemanticsRule.LESetArray

#### Example: 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` is the native vector `v1`;
- `m1` is the pair consisting of `v1` and the parallel composition of `g1` and `g2`;
- the steps so far computed the updated array, but have not assigned it to the variable bound to the array given by `re_array`, which is achieved next. Evaluating the left-hand-side expression `re_array` in an environment `env2` with `m1` is the output configuration $C$.

**Formally**

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{re\_array})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{rm\_array}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_index}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_index}, \texttt{env2}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_index} \overset{\text{is}}{=} (\texttt{index}, \texttt{g1}) \\ \texttt{index} \overset{\text{is}}{=} \texttt{Int}(\texttt{i}) \qquad \texttt{rm\_array} \overset{\text{is}}{=} (\texttt{rv\_array}, \texttt{g2}) \qquad \mathit{set\_index}(\texttt{i}, \texttt{v}, \texttt{rv\_array}) \xrightarrow{\text{eval}} \texttt{v1} \\ \texttt{m1} := (\texttt{v1}, \texttt{g1} \parallel \texttt{g2}) \qquad \mathit{eval\_lexpr}(\texttt{env2}, \texttt{re\_array}, \texttt{m1}) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_lexpr}(\texttt{env}, \text{LE\_SetArray}(\texttt{re\_array}, \texttt{e\_index}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

**Comments**

If the declared type of the right-hand-side expression of a setter has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

We note that the index is guaranteed by the 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`)$^{/\!/\texttt{\#T,\#DE}}$;
- evaluating `e_index` in `env1` is Normal(`m_index`, `env2`)$^{/\!/\texttt{\#T,\#DE}}$;
- `m_index` consists of the native value `index` and the execution graph `g1`;
- `index` is the native label for `l`;
- `rm_array` consists of the native value `rv_array` and the execution graph `g2`;
- setting the value `v` of field `l` of `rv_array` is the native record `v1`;
- `m1` is the pair consisting of `v1` and the parallel composition of `g1` and `g2`;
- the steps so far computed the updated array, but have not assigned it to the variable bound to the array given by `re_array`, which is achieved next. Evaluating the left-hand-side expression `re_array` in an environment `env2` with `m1` is the output configuration $C$.

**Formally**

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{re\_array})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{rm\_array}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_index}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_index}, \texttt{env2}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_index} \overset{\text{is}}{=} (\texttt{index}, \texttt{g1}) \\ \texttt{index} \overset{\text{is}}{=} \text{Label}(\texttt{l}) \qquad \texttt{rm\_array} \overset{\text{is}}{=} (\texttt{rv\_array}, \texttt{g2}) \qquad \mathit{set\_field}(\texttt{l}, \texttt{v}, \texttt{rv\_array}) \xrightarrow{\text{eval}} \texttt{v1} \\ \texttt{m1} := (\texttt{v1}, \texttt{g1} \parallel \texttt{g2}) \qquad \mathit{eval\_lexpr}(\texttt{env2}, \texttt{re\_array}, \texttt{m1}) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_lexpr}(\texttt{env}, \text{LE\_SetEnumArray}(\texttt{re\_array}, \texttt{e\_index}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

## 18.6 Bitvector Slice Assignment Expressions

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

lexpr $\longrightarrow$ LE_Slice(lexpr, 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, LE_Slice(`le1`, `slices`);
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields (`t_le1`, _, _)$^{/\!/\#TE}$;
- obtaining the underlying type of `t_le1` in `tenv` yields `t_le1_anon`$^{/\!/\#TE}$;
- checking that `t_le1_anon` is a bitvector type yields TRUE$^{/\!/TE_UT}$;
- annotating the left-hand-side expression `le1` in `tenv` yields (`le2`, `ses1`)$^{/\!/\#TE}$;
- annotating `slices` in `tenv` yields (`slices_annotated`, `ses_slices`)$^{/\!/\#TE}$;
- 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 TRUE$^{/\!/\#TE}$;
- checking that the slices `slices_annotated` are all disjoint yields TRUE$^{/\!/\#TE}$;

- checking that `slices_annotated` is not empty yields TRUE$^{/\!/\texttt{TE_BS}}$;
- `new_le` is the slicing of `le2` by `slices_annotated`, that is, LE_Slice(`le2`, `slices_annotated`);
- define `ses` as the union of `ses1` and `ses_slices`.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \mathit{rexpr}(\texttt{le1})) \xrightarrow{\text{type}} (\texttt{t\_le1}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_le1}) \xrightarrow{\text{type}} \texttt{t\_le1\_anon} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\texttt{t\_le1\_anon}) = \mathsf{T\_Bits}, \texttt{TE\_UT}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{le1}, \texttt{t\_le1}) \xrightarrow{\text{type}} (\texttt{le2}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_slices}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} (\texttt{slices\_annotated}, \texttt{ses\_slices}) \;/\!/\; \texttt{\#TE} \\ \mathit{slices\_width}(\texttt{tenv}, \texttt{slices\_annotated}) \xrightarrow{\text{type}} \texttt{width'} \\ \mathit{normalize}(\texttt{tenv}, \texttt{width'}) \xrightarrow{\text{type}} \texttt{width} \\ \texttt{t} := \mathsf{T\_Bits}(\texttt{width}, [\,]) \qquad \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_e}, \texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_disjoint\_slices}(\texttt{tenv}, \texttt{slices\_annotated}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\texttt{slices\_annotated} \neq [\,], \texttt{TE\_BS}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{new\_le} := \mathsf{LE\_Slice}(\texttt{le2}, \texttt{slices\_annotated}) \qquad \texttt{ses} := \texttt{ses1} \cup \texttt{ses\_slices}\end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\mathsf{LE\_Slice}(\texttt{le1}, \texttt{slices})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

**TypingRule.CheckDisjointSlices**

The function

$$\mathit{check\_disjoint\_slices}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices}}) \longrightarrow \texttt{TRUE} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

tests whether the list of slices `slices` do not overlap in `tenv`, yielding TRUE. Otherwise, the result is a type error.

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$^{/\!/\texttt{\#TE}}$.
- the result is TRUE.

**Formally**

$$\frac{\mathit{disjoint\_slices\_to\_positions}(\mathsf{tenv}, \texttt{FALSE}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{positions} \;/\!/\; \texttt{\#TE}}{\mathit{check\_disjoint\_slices}(\mathsf{tenv}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

### 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 *rexpr* to `e_bv`) in `env` is Normal(`m_bv`, `env1`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- evaluating `slices` in `env1` is Normal(`m_sliceranges`, `env2`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `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`$^{/\!/\texttt{\#DE}}$;
- `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`$^{/\!/\texttt{\#DE}}$;
- `g3` is the parallel composition of `g1`, and `g2`;
- `new_m_bv` is a pair consisting of `v1` and the execution graph `g3`;
- the steps so far computed the updated bitvector, but have not assigned it to the variable bound to the bitvector given by `e_bv`, which is achieved next. Evaluating the left-hand-side expression `e_bv` with `new_m_bv` in an environment `env2` is the output configuration $C$,

**Formally**

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{e\_bv})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_bv}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_slices}(\texttt{env1}, \texttt{slices}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_sliceranges}, \texttt{env2}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_sliceranges} \overset{\text{is}}{=} (\texttt{slice\_ranges}, \texttt{g1}) \qquad \texttt{m\_bv} \overset{\text{is}}{=} (\texttt{v\_bv}, \texttt{g2}) \\ \mathit{check\_non\_overlapping\_slices}(\texttt{slice\_ranges}) \xrightarrow{\text{eval}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#DE} \\ \mathit{write\_to\_bitvector}(\texttt{slice\_ranges}, \texttt{v}, \texttt{v\_bv}) \xrightarrow{\text{eval}} \texttt{v1} \;\; /\!/ \;\; \texttt{\#DE} \\ \texttt{g3} := \texttt{g1} \parallel \texttt{g2} \qquad \texttt{new\_m\_bv} := (\texttt{v1}, \texttt{g3}) \qquad \mathit{eval\_lexpr}(\texttt{env2}, \texttt{e\_bv}, \texttt{new\_m\_bv}) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_lexpr}(\texttt{env2}, \text{LE\_Slice}(\texttt{e\_bv}, \texttt{slices}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

**Comments**

If the declared type of the right-hand-side expression of a setter has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

**SemanticsRule.CheckNonOverlappingSlices**

The helper function

$$\mathit{check\_non\_overlapping\_slices}(\overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\texttt{value\_ranges}}) \xrightarrow{\text{eval}} \{\texttt{TRUE}\} \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

checks whether the sets of integers represented by the list of ranges `value_ranges` overlap, yielding `TRUE`. 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 `0` 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 == '00';
  return 0;
end;
```

### Prose

All of the following apply:

- view `value_ranges` as the list $\texttt{range}_{1..k}$;
- for every pair of indices `i` and `j` such that $1 \le \texttt{i} < \texttt{j} \le k$, applying *check_two_ranges_non_overlapping* to $\texttt{range}_\texttt{i}$ and $\texttt{range}_\texttt{j}$ yields $\texttt{TRUE}^{/\!/\,\#\texttt{DE}}$;
- the result is `TRUE`.

### Formally

$$\frac{1 \le \texttt{i} < \texttt{j} \le k : \mathit{check\_two\_ranges\_non\_overlapping}(\texttt{range}_\texttt{i}, \texttt{range}_\texttt{j}) \xrightarrow{\text{eval}} \texttt{TRUE} \;/\!/\; \#\texttt{DE}}{\mathit{check\_non\_overlapping\_slices}(\overbrace{\texttt{range}_{1,,k}}^{\texttt{value\_ranges}}) \xrightarrow{\text{eval}} \texttt{TRUE}}$$

### SemanticsRule.CheckTwoRangesNonOverlapping

The helper function

$$\mathit{check\_two\_ranges\_non\_overlapping}((\overbrace{\mathbb{Z}}^{\texttt{s1}} \times \overbrace{\mathbb{Z}}^{\texttt{l1}},),(\overbrace{\mathbb{Z}}^{\texttt{s2}} \times \overbrace{\mathbb{Z}}^{\texttt{l2}},)) \xrightarrow{\text{eval}} \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TDynError}}^{\#\texttt{DE}}$$

checks whether two sets of integers represented by the ranges (`s1`, `l1`) and (`s2`, `l2`) do not intersect, yielding `TRUE`. $/\!/^{\texttt{\#DE}}$

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 $\texttt{TRUE}^{/\!/\texttt{DE\_OSA}}$;
- the result is `TRUE`.

**Formally**

$$\frac{\begin{array}{cc} \mathit{binop}(\texttt{PLUS}, \texttt{s1}, \texttt{l1}) \xrightarrow{\text{eval}} \texttt{s1l1} & \mathit{binop}(\texttt{LEQ}, \texttt{s1l1}, \texttt{s2}) \xrightarrow{\text{eval}} \texttt{s1l1s2} \\ \mathit{binop}(\texttt{PLUS}, \texttt{s2}, \texttt{l2}) \xrightarrow{\text{eval}} \texttt{s2l2} & \mathit{binop}(\texttt{LEQ}, \texttt{s2l2}, \texttt{s1}) \xrightarrow{\text{eval}} \texttt{s2l2s1} \\ \mathit{binop}(\texttt{BOR}, \texttt{s1l1s2}, \texttt{s2l2s1}) \xrightarrow{\text{eval}} \texttt{Bool}(\texttt{b}) & \mathit{check}(\texttt{b}, \texttt{DE\_OSA}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#DE} \end{array}}{\mathit{check\_two\_ranges\_non\_overlapping}((\texttt{s1}, \texttt{l1}), (\texttt{s2}, \texttt{l2})) \xrightarrow{\text{eval}} \texttt{TRUE}}$$

## 18.7 Structured Type Field Assignment Expressions

### 18.7.1 Abstract Syntax

lexpr $\longrightarrow$ LE_SetField(lexpr, identifier)

### 18.7.2 Typing

**TypingRule.LESetBadField**

**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 $(\texttt{t\_le1}, \_, \_)^{/\!/\texttt{\#TE}}$;
- annotating the left-hand-side expression `le1` in `tenv` yields $(\texttt{le2}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_le1` in `tenv` yields a type $\texttt{t\_le1\_anon}^{/\!/\texttt{\#TE}}$;
- 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

TUPLE

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\text{tenv}, \mathit{rexpr}(\texttt{le1})) \xrightarrow{\text{type}} (\texttt{t\_le1}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\text{tenv}, \texttt{le1}, \texttt{t\_le1}) \xrightarrow{\text{type}} (\texttt{le2}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\text{tenv}, \texttt{t\_le1}) \xrightarrow{\text{type}} \texttt{t\_le1\_anon} \;/\!/\; \texttt{\#TE} \\ \overbrace{\qquad\qquad\qquad\qquad}^{\text{******** common prefix ********}} \\ \mathit{ast\_label}(\texttt{t\_le1\_anon}) = \text{T\_Tuple} \end{array}}{\mathit{annotate\_lexpr}(\text{tenv}, \overbrace{\text{LE\_SetField}(\texttt{le1}, \texttt{field})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_AIM})}$$

STRUCTURED

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le1})) \xrightarrow{\mathsf{type}} (\mathtt{t\_le1}, \_, \_) \;/\!/\; \#\mathsf{TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le1}, \mathtt{t\_le1}) \xrightarrow{\mathsf{type}} (\mathtt{le2}, \mathtt{ses}) \;/\!/\; \#\mathsf{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_le1}) \xrightarrow{\mathsf{type}} \mathtt{t\_le1\_anon} \;/\!/\; \#\mathsf{TE} \\ \overset{\text{******** common prefix ********}}{\mathit{ast\_label}(\mathtt{t\_le1\_anon}) \notin \{\mathsf{T\_Tuple}, \mathsf{T\_Exception}, \mathsf{T\_Record}, \mathsf{T\_Collection}, \mathsf{T\_Bits}\}}\end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetField}(\mathtt{le1}, \mathtt{field})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UT})}$$

### 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 $(\mathtt{t\_le1}, \_, \_)^{/\!/\#\mathsf{TE}}$;
- annotating the left-hand-side expression `le1` with type `t_le1` in tenv yields $(\mathtt{le2}, \mathtt{ses})^{/\!/\#\mathsf{TE}}$;
- obtaining the underlying type of `t_le1` in tenv yields a record or exception type with fields `fields`$^{/\!/\#\mathsf{TE}}$;
- checking that there exists a type associated with the field `field` in `fields` TRUE $^{/\!/\mathsf{TE\_BF}}$;
- the type associated with the field `field` in `fields` is `t`;
- determining whether `t_e` type-satisfies `t` yields TRUE$^{/\!/\#\mathsf{TE}}$;
- `new_le` is the access to the field `field` in `le2`, that is, LE_SetField(`le2`, `field`).

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \mathit{rexpr}(\texttt{le1})) \xrightarrow{\text{type}} (\texttt{t\_le1}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{le1}, \texttt{t\_le1}) \xrightarrow{\text{type}} (\texttt{le2}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_le1}) \xrightarrow{\text{type}} L(\texttt{fields}) \;/\!/\; \texttt{\#TE} \\ L \in \{\texttt{T\_Exception}, \texttt{T\_Record}\} \qquad \mathit{assoc\_opt}(\texttt{fields}, \texttt{field}) \xrightarrow{\text{type}} \texttt{ty\_opt} \\ \mathit{check}(\texttt{ty\_opt} \neq \texttt{None}, \texttt{TE\_BF}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ty\_opt} \overset{\text{is}}{=} \langle \texttt{t} \rangle \qquad \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_e}, \texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{new\_le} := \texttt{LE\_SetField}(\texttt{le2}, \texttt{field}) \end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\texttt{LE\_SetField}(\texttt{le1}, \texttt{field})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

**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 $(\texttt{t\_le1}, \_, \_)^{/\!/\texttt{\#TE}}$;
- annotating the left-hand-side expression `le1` with type `t_le1` in `tenv` yields $(\texttt{le2}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_le1` in `tenv` yields a collection type with fields $\texttt{fields}^{/\!/\texttt{\#TE}}$;
- `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 $/\!/_{\texttt{TE\_BF}}$;
- the type associated with the field `field` in `fields` is `t`;
- determining whether `t_e` type-satisfies `t` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `new_le` is the access to the field `field` in `le2`, that is, LE_SetCollectionFields(`base`, [`field`]).

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le1})) \xrightarrow{\text{type}} (\mathtt{t\_le1}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le1}, \mathtt{t\_le1}) \xrightarrow{\text{type}} (\mathtt{le2}, \mathtt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_le1}) \xrightarrow{\text{type}} \mathsf{T\_Collection}(\mathtt{fields}) \;/\!/\; \texttt{\#TE} \\ \mathit{assoc\_opt}(\mathtt{fields}, \mathtt{field}) \xrightarrow{\text{type}} \mathtt{ty\_opt} \\ \mathit{check}(\mathtt{ty\_opt} \neq \mathsf{None}, \mathsf{TE\_BF}) \longrightarrow \mathsf{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathtt{ty\_opt} \overset{\text{is}}{=} \langle \mathtt{t} \rangle \qquad \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathtt{new\_le} := \mathsf{LE\_SetCollectionFields}(\mathtt{base}, [\mathtt{field}]) \end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetField}(\mathtt{le1}, \mathtt{field})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

## 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 :: x.time :: x.data == '1 0000000000000000 11111111';

    var y : Message;
    assert y == '00000000 0000000000000000 0';
    y.[data, time, status] = Ones{8} :: Zeros{16} :: '1';
    assert y == '11111111 0000000000000000 1';
    return 0;
end;
```

### 18.8.1 Abstract Syntax

$\mathsf{lexpr} \longrightarrow \mathsf{LE\_SetFields}(\mathsf{lexpr}, \mathsf{identifier}^*)$

### 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 $(\texttt{t\_base}, \_, \_)^{/\!/\texttt{\#TE}}$;
- annotating the assignable expression `le_base` with the right-hand side type `t_base` in the static environment tenv yields $(\texttt{le\_base\_annot}, \texttt{ses\_base})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_base` in the static environment tenv yields $\texttt{t\_base\_anon}^{/\!/\texttt{\#TE}}$;
- One of the following applies:
  - All of the following apply (BITS):
    - `t_anon_base` is a bitvector type with list of bitfields `bitfields`;
    - applying *find_bitfields_slices* to `name` and `bitfields`, for every `name` in `le_fields`, yields $\texttt{slices}_{\texttt{name}}{}^{/\!/\texttt{\#TE}}$;
    - define `le_slice` as the left-hand-side slicing expression for the base expression `le_base_annot` and list of slices formed by concatenating all slice lists $\texttt{slices}_{\texttt{name}}$, for every `name` in `le_fields`;
    - annotating the assignable expression `le_slice` with the right-hand side type `t_e` in the static environment tenv yields $(\texttt{new\_le}, \texttt{ses})^{/\!/\texttt{\#TE}}$.
  - All of the following apply (RECORD):
    - `t_anon_base` is a record type with list of fields `base_fields`;
    - applying *fold_bitvector_fields* to `le_fields` and `base_fields` in tenv yields $(\texttt{v\_length}, \texttt{slices})^{/\!/\texttt{\#TE}}$;
    - define `t_lhs` as the bitvector type of length `v_length` and no bitfields, that is, T_Bits( E_Literal(L_Int) `v_length` , [ ]);
    - checking that `t_e` type-satisfies `t_lhs` in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
    - define `new_le` as the assignable expression of the list of fields `le_fields` to the base expression `le_base`, that is, LE_SetFields(`le_base`, `le_fields`);
    - define `ses` as `ses_base`.
  - All of the following apply (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 $(\texttt{v\_length}, \texttt{slices})^{/\!/\texttt{\#TE}}$;
    - define `t_lhs` as the bitvector type of length `v_length` and no bitfields, that is, T_Bits( E_Literal(L_Int) `v_length` , [ ]);
    - checking that `t_e` type-satisfies `t_lhs` in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
    - define `new_le` as the assignable expression of the list of fields `le_fields` to the 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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le\_base})) \xrightarrow{\mathsf{type}} (\mathtt{t\_base}, \_, \_) \;/\!/\; \#\mathsf{TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le\_base}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} (\mathtt{le\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \#\mathsf{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} \mathtt{t\_base\_anon} \;/\!/\; \#\mathsf{TE} \\ \overset{\text{******** common prefix ********}}{\mathtt{t\_base\_anon} = \mathsf{T\_Bits}(\_, \mathtt{bitfields})} \\ \mathtt{name} \in \mathtt{le\_fields} : \mathit{find\_bitfields\_slices}(\mathtt{name}, \mathtt{bitfields}) \xrightarrow{\mathsf{type}} \mathtt{slices}_{\mathtt{name}} \;/\!/\; \#\mathsf{TE} \\ \mathtt{le\_slice} := \mathsf{LE\_Slice}(\mathtt{le\_base\_annot}, [\mathtt{name} \in \mathtt{le\_fields} : \mathtt{slices}_{\mathtt{name}}]) \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le\_slice}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses}) \;/\!/\; \#\mathsf{TE}\end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base}, \mathtt{le\_fields})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

RECORD

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le\_base})) \xrightarrow{\mathsf{type}} (\mathtt{t\_base}, \_, \_) \;/\!/\; \#\mathsf{TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le\_base}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} (\mathtt{le\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \#\mathsf{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} \mathtt{t\_base\_anon} \;/\!/\; \#\mathsf{TE} \\ \overset{\text{******** common prefix ********}}{\mathtt{t\_base\_anon} = \mathsf{T\_Record}(\mathtt{base\_fields})} \\ \mathit{fold\_bitvector\_fields}(\mathsf{tenv}, \mathtt{base\_fields}, \mathtt{le\_fields}) \xrightarrow{\mathsf{type}} (\mathtt{v\_length}, \mathtt{slices}) \;/\!/\; \#\mathsf{TE} \\ \mathtt{t\_lhs} := \mathsf{T\_Bits}(\overbrace{\mathtt{v\_length}}^{\mathsf{E\_Literal(L\_Int)}}, [\,]) \\ \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{t\_lhs}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \#\mathsf{TE}\end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base}, \mathtt{le\_fields})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base\_annot}, \mathtt{le\_fields}, \mathtt{slices})}^{\mathtt{new\_le}}, \overbrace{\mathtt{ses\_base}}^{\mathtt{ses}})}$$

COLLECTION

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le\_base})) \xrightarrow{\text{type}} (\mathtt{t\_base}, \_, \_) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le\_base}, \mathtt{t\_base}) \xrightarrow{\text{type}} (\mathtt{le\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base}) \xrightarrow{\text{type}} \mathtt{t\_base\_anon} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathtt{t\_base\_anon} = \mathsf{T\_Collection}(\mathtt{base\_fields}) \qquad \mathtt{le\_base} = \mathsf{LE\_Var}(\mathtt{base}) \\ \mathit{fold\_bitvector\_fields}(\mathsf{tenv}, \mathtt{base\_fields}, \mathtt{le\_fields}) \xrightarrow{\text{type}} (\mathtt{v\_length}, \mathtt{slices}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{t\_lhs} := \mathsf{T\_Bits}(\overbrace{\mathtt{v\_length}}^{\mathsf{E\_Literal(L\_Int)}}, [\,]) \\ \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{t\_lhs}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base}, \mathtt{le\_fields})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\overbrace{\mathsf{LE\_SetCollectionFields}(\mathtt{base}, \mathtt{le\_fields}, \mathtt{slices})}^{\mathtt{new\_le}}, \overbrace{\mathtt{ses\_base}}^{\mathtt{ses}})}$$

ERROR

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le\_base})) \xrightarrow{\text{type}} (\mathtt{t\_base}, \_, \_) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le\_base}, \mathtt{t\_base}) \xrightarrow{\text{type}} (\mathtt{le\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base}) \xrightarrow{\text{type}} \mathtt{t\_base\_anon} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_base\_anon}) \notin \{\mathsf{T\_Bits}, \mathsf{T\_Record}, \mathsf{T\_Collection}\}, \mathtt{TE\_UT}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base}, \mathtt{le\_fields})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

### TypingRule.FoldBitvectorFields

The helper function

$$\mathit{fold\_bitvector\_fields}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{field}^*}^{\mathtt{base\_fields}}, \overbrace{\mathsf{bitfield}^*}^{\mathtt{le\_fields}}) \longrightarrow (\overbrace{\mathbb{N}}^{\mathtt{v\_length}} \times \overbrace{(\mathbb{N} \times N)^*}^{\mathtt{slices}})$$

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 $(\texttt{v\_start}, \texttt{slices1})^{/\!/\texttt{\#TE}}$;
  - applying *assoc_opt* to `base_fields` and `field` yields `ty_opt`;
  - checking that `ty_opt` is different to `None` yields $\texttt{TRUE}^{/\!/\texttt{TE\_BF}}$;
  - view `ty_opt` as $\langle \texttt{t\_field} \rangle$;
  - applying *get_bitvector_const_width* to `t_field` in `tenv` yields $\texttt{field\_width}^{/\!/\texttt{\#TE}}$;
  - define `v_length` as $\texttt{v\_start} + \texttt{field\_width}$;
  - define `slices` as the list with `head` $(\texttt{v\_start}, \texttt{field\_width})$ and `tail` `slices1`.

### Formally

EMPTY

$$\mathit{fold\_bitvector\_fields}(\mathsf{tenv}, \texttt{base\_fields}, \overbrace{[\,]}^{\texttt{le\_fields}}) \xrightarrow{\text{type}} (\overbrace{0}^{\texttt{v\_length}}, \overbrace{[\,]}^{\texttt{slices}})$$

NON_EMPTY

$$\begin{gathered}\mathit{fold\_bitvector\_fields}(\mathsf{tenv}, \texttt{base\_fields}, \texttt{le\_fields1}) \xrightarrow{\text{type}} (\texttt{v\_start}, \texttt{slices1}) \;/\!/\; \texttt{\#TE} \\ \mathit{assoc\_opt}(\texttt{base\_fields}, \texttt{field}) \xrightarrow{\text{type}} \texttt{ty\_opt} \\ \mathit{check}(\texttt{ty\_opt} \neq \texttt{None}, \texttt{TE\_BF}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ty\_opt'} \overset{\text{is}}{=} \langle \texttt{t\_field} \rangle \\ \mathit{get\_bitvector\_const\_width}(\mathsf{tenv}, \texttt{t\_field}) \xrightarrow{\text{type}} \texttt{field\_width} \;/\!/\; \texttt{\#TE}\end{gathered}$$

---

$$\mathit{fold\_bitvector\_fields}(\mathsf{tenv}, \texttt{base\_fields}, \overbrace{\texttt{le\_fields1} + [\texttt{field}]}^{\texttt{le\_fields}}) \xrightarrow{\text{type}} (\overbrace{\texttt{v\_start} + \texttt{field\_width}}^{\texttt{v\_length}}, \overbrace{[(\texttt{v\_start}, \texttt{field\_width})] + \texttt{slices1}}^{\texttt{slices}})$$

### 18.8.3 Semantics

#### SemanticsRule.LESetFields

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 $((\texttt{v\_record}, \texttt{g1}), \texttt{env1})^{/\!/ \texttt{\#T,\#DE}}$;
- applying *assign_bitvector_fields* to `v`, `v_record`, `fields`, and `slices`, yields $\texttt{v2}^{/\!/ \texttt{\#DE}}$;
- 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

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{le\_record}), \texttt{rm\_record\_new}) \xrightarrow{\text{eval}} ((\texttt{v\_record}, \texttt{g1}), \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{assign\_bitvector\_fields}(\texttt{v}, \texttt{v\_record}, \texttt{fields}, \texttt{slices}) \xrightarrow{\text{eval}} \texttt{v2} \;/\!/\; \texttt{\#DE} \\ \texttt{m2} := (\texttt{v2}, \texttt{g} \parallel \texttt{g1}) \qquad \mathit{eval\_lexpr}(\texttt{env1}, \texttt{le\_record}, \texttt{m2}) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_lexpr}(\texttt{env}, \overbrace{\mathsf{LE\_SetFields}(\texttt{le\_record}, \texttt{fields}, \texttt{slices})}^{\texttt{le}}, (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

#### SemanticsRule.AssignBitvectorFields

The helper function

$$\mathit{assign\_bitvector\_fields}(\overbrace{\mathcal{BV}}^{\texttt{bitvector}}, \overbrace{\mathcal{REC}}^{\texttt{record}}, \overbrace{\mathsf{identifier}^*}^{\texttt{fields}}, \overbrace{(\mathbb{N} \times \mathbb{N})^*}^{\texttt{slices}}) \longrightarrow \overbrace{\mathcal{REC}}^{\texttt{res}}$$

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`('11111111000000000000000001').

**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

$$\mathit{assign\_bitvector\_fields}(\texttt{bitvector}, \texttt{record}, \overbrace{[\,]}^{\texttt{fields}}, \overbrace{[\,]}^{\texttt{slices}}) \xrightarrow{\text{eval}} \overbrace{\texttt{record}}^{\texttt{res}}$$

NON_EMPTY

$$\texttt{slice} := [(\texttt{Int}(\texttt{i1}), \texttt{Int}(\texttt{i2}))]$$
$$\mathit{read\_from\_bitvector}(\texttt{bitvector}, \texttt{slice}) \xrightarrow{\text{eval}} \texttt{record\_slices} \;/\!/\; \texttt{\#DE}$$
$$\mathit{set\_field}(\texttt{field\_name}, \texttt{record\_slices}, \texttt{record}) \xrightarrow{\text{eval}} \texttt{record1}$$
$$\mathit{assign\_bitvector\_fields}(\texttt{bitvector}, \texttt{record1}, \texttt{fields1}, \texttt{slices1}) \xrightarrow{\text{eval}} \texttt{res} \;/\!/\; \texttt{\#DE}$$

---

$$\mathit{assign\_bitvector\_fields}(\texttt{bitvector}, \texttt{record}, \overbrace{[\texttt{field\_name}] + \texttt{fields1}}^{\texttt{fields}}, \overbrace{[(\texttt{i1}, \texttt{i2})] + \texttt{slices1}}^{\texttt{slices}}) \xrightarrow{\text{eval}} \texttt{res}$$

**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

```
type MYCOLLECTION of collection {
  field1: bits(3),
  field2: bits(4),
};

var MyCollection: MYCOLLECTION;

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 ($G^{\text{denv}}$.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 $G^{\text{denv}}$.
- `new_g` is the ordered composition with the `asl_data` edge of `g` and `g0`;

### Formally

$$\texttt{env} \overset{\text{is}}{=} (\texttt{tenv}, \texttt{denv}) \qquad \texttt{base} \in \text{dom}(G^{\text{denv}}) \qquad \texttt{record} := G^{\text{denv}}[\texttt{base}]$$

$$\textit{assign\_bitvector\_fields}(\texttt{v}, \texttt{record}, \texttt{fields}, \texttt{slices}) \xrightarrow{\text{eval}} \texttt{record1} \;/\!/\; \#DE$$

$$\texttt{g0} := \{\texttt{field\_name} \in \textit{field\_names} : \text{WriteEffect}(\texttt{base} + \texttt{"."} + \texttt{field\_name})\}$$

$$\texttt{new\_env} := (\texttt{tenv}, (G^{\text{denv}}[\texttt{x} \mapsto \texttt{record1}], L^{\text{denv}})) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \texttt{g0}$$

---

$$\textit{eval\_lexpr}(\texttt{env}, \overbrace{\text{LE\_SetCollectionFields}(\texttt{base}, \texttt{fields}, \texttt{slices})}^{\texttt{le}}, (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})$$

## 18.9 Bitfield Assignable Expressions

### 18.9.1 Abstract Syntax

lexpr $\longrightarrow$ LE_Slice(lexpr, 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

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le1})) \xrightarrow{\mathsf{type}} (\mathtt{t\_le1}, \_, \_) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le1}, \mathtt{t\_le1}) \xrightarrow{\mathsf{type}} (\mathtt{le2}, \mathtt{ses}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t\_le1}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field}) \xrightarrow{\mathsf{type}} \mathsf{None} \end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetField}(\mathtt{le1}, \mathtt{field})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_BF})}$$

FIELD_SIMPLE

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le1})) \xrightarrow{\mathsf{type}} (\mathtt{t\_le1}, \_, \_) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le1}, \mathtt{t\_le1}) \xrightarrow{\mathsf{type}} (\mathtt{le2}, \mathtt{ses}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t\_le1}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field}) \xrightarrow{\mathsf{type}} \langle \mathsf{BitField\_Simple}(\_, \mathtt{slices}) \rangle \\ \mathit{slices\_width}(\mathsf{tenv}, \mathtt{slices}) \xrightarrow{\mathsf{type}} \mathtt{w} \qquad \mathtt{t} := \mathsf{T\_Bits}(\mathtt{w}, [\,]) \\ \text{******** common suffix ********} \\ \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathtt{le2} := \mathsf{LE\_Slice}(\mathtt{le1}, \mathtt{slices}) \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le2}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses}) \;\; /\!/ \;\; \mathtt{\#TE} \end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetField}(\mathtt{le1}, \mathtt{field})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

FIELD_NESTED

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\mathtt{le1})) \xrightarrow{\mathsf{type}} (\mathtt{t\_le1}, \_, \_) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le1}, \mathtt{t\_le1}) \xrightarrow{\mathsf{type}} (\mathtt{le2}, \mathtt{ses}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t\_le1}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field}) \xrightarrow{\mathsf{type}} \langle \mathsf{BitField\_Nested}(\_, \mathtt{slices}, \mathtt{bitfields'}) \rangle \\ \mathit{slices\_width}(\mathsf{tenv}, \mathtt{slices}) \xrightarrow{\mathsf{type}} \mathtt{w} \qquad \mathtt{t} := \mathsf{T\_Bits}(\mathtt{w}, \mathtt{bitfields'}) \\ \text{******** common suffix ********} \\ \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathtt{le2} := \mathsf{LE\_Slice}(\mathtt{le1}, \mathtt{slices}) \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le2}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses}) \;\; /\!/ \;\; \mathtt{\#TE} \end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetField}(\mathtt{le1}, \mathtt{field})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

FIELD_TYPED

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \mathit{rexpr}(\texttt{le1})) \xrightarrow{\text{type}} (\texttt{t\_le1}, \_, \_) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{le1}, \texttt{t\_le1}) \xrightarrow{\text{type}} (\texttt{le2}, \texttt{ses}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{get\_structure}(\texttt{tenv}, \texttt{t\_le1}) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\_, \texttt{bitfields}) \;\; /\!/ \;\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{find\_bitfield\_opt}(\texttt{bitfields}, \texttt{field}) \xrightarrow{\text{type}} \langle \mathsf{BitField\_Type}(\_, \texttt{slices}, \texttt{t}) \rangle \\ \mathit{slices\_width}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{w} \\ \texttt{t'} := \mathsf{T\_Bits}(\texttt{w}, [\,]) \qquad \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t'}, \texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#TE} \\ \text{******** common suffix ********} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_e}, \texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{le2} := \mathsf{LE\_Slice}(\texttt{le1}, \texttt{slices}) \\ \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{le2}, \texttt{t\_e}) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses}) \;\; /\!/ \;\; \texttt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\mathsf{LE\_SetField}(\texttt{le1}, \texttt{field})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

### 18.9.3 Semantics

The semantics for assigning to individual bitvector bitfields is covered by SemanticsRule.LESlice as the type system transforms the untyped AST for assigning to an individual bitfield into an LE_Slice typed AST.

#### SemanticsRule.LESetField

#### Example: 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`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `rm_record` is a pair consisting of the native record `rv_record` and the execution graph `g1`;
- setting the field `field_name` in the native record `rv_record` to `v` is the updated native record `v1`;
- `m1` is the pair consisting of the native vector `v1` and the execution graph that is, the parallel composition of `g` and `g1`;
- the steps so far computed the updated record, but have not assigned it to the variable holding the record given by `record`, which is achieved next. Evaluating the left-hand-side expression `re_record` in an environment `env1` with `m1` is the output configuration $C$.

### Formally

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{re\_record})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{rm\_record}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{rm\_record} \overset{\text{is}}{=} (\texttt{rv\_record}, \texttt{g1}) \qquad \mathit{set\_field}(\texttt{field\_name}, \texttt{v}, \texttt{rv\_record}) \xrightarrow{\text{eval}} \texttt{v1} \\ \texttt{m1} := (\texttt{v1}, \texttt{g} \parallel \texttt{g1}) \qquad \mathit{eval\_lexpr}(\texttt{env1}, \texttt{re\_record}, \texttt{m1}) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_lexpr}(\texttt{env}, \text{LE\_SetField}(\texttt{re\_record}, \texttt{field\_name}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

### Comments

We note that the 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.

A local declaration keyword is one of `var`, `let`, and `constant`. A local declaration item is an element derived from `decl_item`. A local declaration consists of a local declaration item and a local declaration keyword.

We show the syntax relevant to local declarations in Section 19.1 and the AST rule and rules need to build the AST for assignable expressions in Section 19.2. We then define the typing and semantics of the different kinds of local declarations:

- Variable declarations (see Section 19.3)
- Tuple declarations (see Section 19.4)

**Typing:** The function

$$\mathit{annotate\_local\_decl\_item}\left(\begin{array}{c}\overbrace{\mathbb{SE}}^{\texttt{tenv}},\\ \overbrace{\mathsf{ty}}^{\texttt{ty}},\\ \overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}},\\ \overbrace{\langle \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect})\rangle}^{\texttt{e\_opt}},\\ \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ldi}}\end{array}\right) \longrightarrow \left(\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}\right) \cup \overbrace{\mathsf{TTypeError}}^{\#\texttt{TE}}$$

annotates a local declaration item `ldi` with a local declaration keyword `ldk`, given a type `ty`, and optionally `e_opt` — an initializing expression and set of side effect descriptors, in a static environment `tenv` results in `new_env`, the modified static environment. Otherwise, the result is a type error.

**Semantics:** The relation

$$\mathit{eval\_local\_decl}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ldi}}, \overbrace{\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g1}}}^{\texttt{m}}) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}})$$

evaluates a local declaration item `ldi` in an environment `env` with an initialization value `m`. That is, the right-hand side of the declaration has already been evaluated, yielding `m` (see, for example, SemanticsRule.SDeclSome). Evaluation of the local variables `ldi` in an environment `env` is either Normal(`g`, `new_env`) or an abnormal configuration.

While there are three different categories of local storage elements — constants, mutable variables (declared via `var`), and immutable variables (declared via `let`) — from the perspective of the semantics of local storage elements, they are all treated the same way.

## 19.1 Syntax

Declaring a local storage element is done via the following grammar rules:

```
stmt ⟶ local_decl_keyword_non_var decl_item option(as_ty) "=" expr ";"
     | "var" decl_item option(as_ty) option("=" expr) ";"
     | "var" clist2(ID) as_ty ";"

local_decl_keyword_non_var ⟶ "let" | "constant"
                 decl_item ⟶ ID
                             | plist2(ignored_or_identifier)
ignored_or_identifier ⟶ "-" | ID
```

## 19.2 Abstract Syntax

```
local_decl_keyword ⟶ LDK_Var | LDK_Constant | 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 19.1 are illegal, as they discard all declared storage elements.

Listing 19.1: 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

$$\mathit{build\_local\_decl\_keyword\_non\_var}(\overbrace{\mathbb{PARSE}[\texttt{local\_decl\_keyword\_non\_var}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ast\_node}}$$

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

LET

$$\mathit{build\_local\_decl\_keyword\_non\_var}(\overbrace{\texttt{local\_decl\_keyword\_non\_var}(\texttt{"let"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDK\_Let}}^{\texttt{ast\_node}}$$

CONSTANT

$$\mathit{build\_local\_decl\_keyword\_non\_var}(\overbrace{\texttt{local\_decl\_keyword\_non\_var}(\texttt{"constant"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDK\_Constant}}^{\texttt{ast\_node}}$$

### ASTRule.DeclItem

The function

$$\mathit{build\_decl\_item}(\overbrace{\mathbb{PARSE}[\texttt{decl\_item}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ast\_node}}$$

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

VAR

$$\mathit{build\_decl\_item}(\texttt{decl\_item}(\textbf{ID}(\texttt{name}))) \xrightarrow{\text{ast}} \overbrace{\texttt{name}}^{\texttt{ast\_node}}$$

TUPLE

$$\frac{\texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{ids})} \texttt{ids}_i = \texttt{"-"} \qquad \mathit{check}(\neg\texttt{b}, \texttt{BE\_BD}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#BE} \qquad \mathit{build\_clist}[\mathit{build\_ignored\_or\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_decl\_item}(\texttt{decl\_item}(\texttt{ids} : \texttt{plist2}(\texttt{ignored\_or\_identifier}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDI\_Tuple}(\texttt{ids\_ast})}^{\texttt{ast\_node}}}$$

**ASTRule.IgnoredOrIdentifier**

The relation

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{ignored\_or\_identifier}]}^{\texttt{parsed\_node}}) \times \overbrace{\mathsf{identifier}}^{\texttt{ast\_node}}$$

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

DISCARD

$$\frac{\texttt{id} \in \mathsf{identifier} \text{ is fresh}}{\mathit{build\_ignored\_or\_identifier}(\overbrace{\texttt{ignored\_or\_identifier}(\texttt{"-"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{id}}^{\texttt{ast\_node}}}$$

ID

$$\mathit{build\_ignored\_or\_identifier}(\overbrace{\texttt{ignored\_or\_identifier}(\textbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{id}}^{\texttt{ast\_node}}$$

## 19.3 Variable Declarations

### 19.3.1 Typing

**TypingRule.LDVar**

**Example: Well-typed Local Variable Declarations**

In Listing 19.2, 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 19.2: 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- determining whether `x` is not declared in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

- 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**

$$\frac{\begin{array}{c}\mathit{check\_var\_not\_in\_env}(\texttt{tenv}, \texttt{x}) \xrightarrow{\text{type}} \text{TRUE} \;/\!/\; \#TE \\ \mathit{check\_no\_precision\_loss}(\texttt{ty}) \xrightarrow{\text{type}} \text{TRUE} \;/\!/\; \#TE \\ \mathit{add\_local}(\texttt{tenv}, \texttt{x}, \texttt{ty}, \texttt{ldk}) \xrightarrow{\text{type}} \texttt{tenv2} \\ \mathit{add\_immutable\_expr}(\texttt{tenv2}, \texttt{ldk}, \texttt{e\_opt}, \texttt{x}) \xrightarrow{\text{type}} \texttt{new\_tenv}\end{array}}{\mathit{annotate\_local\_decl\_item}(\texttt{tenv}, \texttt{ty}, \texttt{ldk}, \texttt{e\_opt}, \overbrace{\text{LDI\_Var}(\texttt{x})}^{\text{ldi}}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

**TypingRule.CheckIsNotCollection**

The helper function

$$\mathit{check\_is\_not\_collection}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \xrightarrow{\text{type}} \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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 19.3, the statement `var test: MyCollection;` fails with a type error because TypingRule.LDVar calls TypingRule.CheckIsNotCollection.

Listing 19.3: Declaring a local variable with a collection type

```
type MyCollection of collection {
  field1: bits(2),
  field2: bits(3),
};

func main () => integer
begin
  var test: MyCollection; // 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.

**Formally**

$$\textsc{Collection}\quad \frac{\mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \#\mathsf{TE} \qquad \mathit{ast\_label}(\mathtt{t\_struct}) = \mathsf{T\_Collection}}{\mathit{check\_is\_not\_collection}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_UT})}$$

$$\textsc{Not-Collection}\quad \frac{\mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \#\mathsf{TE} \qquad \mathit{ast\_label}(\mathtt{t\_struct}) \neq \mathsf{T\_Collection}}{\mathit{check\_is\_not\_collection}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE}}$$

### 19.3.2 Semantics

**SemanticsRule.LDVar**

**Example: Evaluation of a Local Variable Declaration**

The statement `var x = 3;` in Listing 19.2 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**

$$\frac{\texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g1}) \qquad \textit{declare\_local\_identifier}(\texttt{env}, \texttt{x}, \texttt{v}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g2}) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2}}{\textit{eval\_local\_decl}(\texttt{env}, \text{LDI\_Var}(\texttt{x}), \texttt{m}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

## 19.4 Tuple Declarations

### 19.4.1 Typing

**TypingRule.LDTuple**

**Example: Well-typed Tuple Declarations**

Listing 19.4: 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 $\texttt{ids}_{1..k}$, that is, LDI_Tuple($\texttt{ids}_{1..k}$);
- obtaining the underlying type of `ty` in `tenv` yields `t'`$^{\text{// \#TE}}$;
- determining whether `t'` is a tuple type yields TRUE$^{\text{// \#TE}}$;
- determining whether the number of elements of `t'` is $k$ yields TRUE$^{\text{// \#TE}}$;
- declaring the identifiers in `ids` in the static environment `tenv` from right to left with their corresponding (that is, with the same index) types $t_{1..k}$ in `tenv`, propagating static environments from one declaration to the next, yields the resulting environment `new_tenv`$^{\text{// \#TE}}$.

**Formally**

$$
\begin{array}{c}
\mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{t'} \;/\!/\; \texttt{\#TE} \\
\mathit{check}(\mathit{ast\_label}(\texttt{t'}) = \mathsf{T\_Tuple}, \texttt{TupleTypeExpected}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\
\texttt{t'} \overset{\text{is}}{=} \mathsf{T\_Tuple}([\texttt{t}_{1..n}]) \\
\mathit{check}(k = n, \texttt{InvalidArity}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\
\texttt{new\_tenv}_k := \texttt{tenv} \\
i = k..1 : \\
\mathit{check\_var\_not\_in\_env}(\texttt{new\_tenv}_i, \texttt{ids}_i) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\
\mathit{add\_local}(\texttt{tenv}, \texttt{ids}_i, \texttt{t}_i, \texttt{ldk}) \xrightarrow{\text{type}} \texttt{new\_tenv}_{i-1} \\
\texttt{new\_tenv} := \texttt{new\_tenv}_0 \\
\hline
\mathit{annotate\_local\_decl\_item}(\texttt{tenv}, \texttt{ty}, \texttt{ldk}, \texttt{e\_opt}, \overbrace{\mathsf{LDI\_Tuple}(\texttt{ids}_{1..k})}^{\texttt{ldi}}) \xrightarrow{\text{type}} \quad \texttt{new\_tenv}
\end{array}
$$

## 19.4.2 Semantics

**SemanticsRule.LDTuple**

**Example: Evaluation of Tuple Declarations**

In Listing 19.5, `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 19.5: 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 $\texttt{id}_{1..k}$;
- the value at each index of `v` is $\texttt{v}_i$, for $i = 1..k$;
- `liv` is the list of pairs $(\texttt{v}_i, \texttt{g})$, for $i = 1..k$;
- the output configuration is obtained by declaring each identifier $\texttt{id}_i$ with the corresponding value (`m` component) $(\texttt{v}_i, \texttt{g})$.

**Formally**

$$\frac{\texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g}) \qquad \texttt{ldis} \overset{\text{is}}{=} \texttt{id}_{1..k} \qquad i = 1..k : \mathit{get\_index}(i, \texttt{v}) \xrightarrow{\text{eval}} \texttt{v}_i \qquad \texttt{liv} \overset{\text{is}}{=} [i = 1..k : (\texttt{v}_i, \texttt{g})] \qquad \mathit{ldi\_tuple\_folder}(\texttt{env}, \texttt{ids}, \texttt{liv}) \xrightarrow{\text{eval}} C}{\mathit{eval\_local\_decl}(\texttt{env}, \mathsf{LDI\_Tuple}(\texttt{ids}), \texttt{m}) \xrightarrow{\text{eval}} C}$$

**SemanticsRule.LDITupleFolder**

The helper semantic relation

$$\mathit{ldi\_tuple\_folder}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{identifier}^*}^{\texttt{ids}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{liv}}) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}})$$

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 `asl_data` edge, and `g3`.

**Formally**

EMPTY

$$\mathit{ldi\_tuple\_folder}(\texttt{env}, \overbrace{[\,]}^{\texttt{ids}}, \overbrace{[\,]}^{\texttt{liv}}) \xrightarrow{\text{eval}} \mathsf{Normal}(\emptyset_{\texttt{g}}, \texttt{env})$$

NON_EMPTY

$$\frac{\texttt{ids} = [\texttt{id}] + \texttt{ids}' \qquad \texttt{liv} = [\texttt{m}] + \texttt{liv}' \qquad \texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g1}) \qquad \mathit{declare\_local\_identifier}(\texttt{env}, \texttt{id}, \texttt{v}) \xrightarrow{\text{eval}} (\texttt{env1}, \texttt{g2}) \qquad \mathit{ldi\_tuple\_folder}(\texttt{env1}, \texttt{ids}', \texttt{liv}') \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{g3}, \texttt{new\_env}) \qquad \texttt{new\_g} := (\texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2}) \parallel \texttt{g3}}{\mathit{ldi\_tuple\_folder}(\texttt{env}, \texttt{ids}, \texttt{liv}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

# Chapter 20

# Statements

Statements update storage elements and determine the flow of control of a subprogram.

Statements are grammatically derived from `stmt` and represented as ASTs by stmt.

The function

$$build\_stmt(\overbrace{\mathbb{PARSE}[\texttt{stmt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

The function

$$annotate\_stmt(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{stmt}}^{\texttt{s}}) \longrightarrow (\overbrace{\mathsf{stmt}}^{\texttt{new\_s}} \times \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a statement `s` in an environment `tenv`, resulting in `new_s` — the typed AST for `s`, which is also known as the *annotated statement* — a modified environment `new_tenv`, and set of side effect descriptors `ses`. Otherwise, the result is a type error.

The relation

$$eval\_stmt(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{stmt}}^{\texttt{s}}) \times \begin{pmatrix} \overbrace{\mathsf{TReturning}}^{\mathsf{Returning}(\texttt{(vs,new\_g),new\_env})} & \cup \\ \overbrace{\mathsf{TContinuing}}^{\mathsf{Continuing}(\texttt{new\_g,new\_env})} & \cup \\ \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} & \cup \\ \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}} & \end{pmatrix}$$

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 ⟶ "pass" ";"

### 20.1.2 Abstract Syntax

stmt ⟶ S_Pass

#### ASTRule.SPass

$$build\_stmt(\overbrace{\texttt{stmt}(\texttt{"pass"}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Pass}}^{\texttt{ast\_node}}$$

### 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

$$annotate\_stmt(\text{tenv}, \text{S\_Pass}) \xrightarrow{\text{type}} (\text{S\_Pass}, \text{tenv}, \overbrace{\emptyset}^{\texttt{ses}})$$

### 20.1.4 Semantics

**SemanticsRule.SPass**

**Example: 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**

$$\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Pass}) \xrightarrow{\text{eval}} \mathsf{Continuing}(\overbrace{\emptyset_g}^{\texttt{new\_g}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

## 20.2 Assignment Statements

### 20.2.1 Syntax

$$\mathsf{stmt} \longrightarrow \mathsf{lexpr}\ \texttt{"="}\ \mathsf{expr}\ \texttt{";"}$$

### 20.2.2 Abstract Syntax

$$\mathsf{stmt} \longrightarrow \mathsf{S\_Assign}(\mathsf{lexpr}, \mathsf{expr})$$

**ASTRule.SAssign**

$$\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\mathsf{lexpr}, \texttt{"="}, \mathsf{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Assign}(\overline{\mathsf{lexpr}}, \overline{\mathsf{expr}})}^{\texttt{ast\_node}}$$

### 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`)$^{/\!/\texttt{\#TE}}$;
- annotating the assignable expression `le` with the type `t_re` in `tenv` yields (`le1`, `ses_le`)$^{/\!/\texttt{\#TE}}$;
- `new_s` is the assignment `le1 = re1`, that is, S_Assign(`le1`, `re1`);
- `new_tenv` is `tenv`;
- define `ses` as the union of `ses_re` and `ses_le`.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{re}) \xrightarrow{\text{type}} (\mathtt{t\_re}, \mathtt{re1}, \mathtt{ses\_re}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \mathtt{le}, \mathtt{t\_re}) \xrightarrow{\text{type}} (\mathtt{le1}, \mathtt{ses\_le}) \;/\!/\; \texttt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_re} \cup \mathtt{ses\_le}\end{array}}{\mathit{annotate\_stmt}(\mathsf{tenv}, \overbrace{\mathsf{S\_Assign}(\mathtt{le}, \mathtt{re})}^{\mathtt{s}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{S\_Assign}(\mathtt{le1}, \mathtt{re1})}^{\mathtt{new\_s}}, \overbrace{\mathsf{tenv}}^{\mathtt{new\_tenv}}, \mathtt{ses})}$$

### 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:

$$\mathit{lexpr\_is\_var}(\mathsf{lexpr}) \longrightarrow \texttt{TRUE}$$

which holds when a left-hand side expression represents a variable or a discarded left-hand-side expression:

$$\mathit{lexpr\_is\_var}(\texttt{le}) \xrightarrow{\text{eval}} \mathit{ast\_label}(\texttt{le}) \in \{\mathsf{LE\_Var}, \mathsf{LE\_Discard}\}$$

The inference rules defining the evaluation of assigning from a subprogram call are as follows:

$$\frac{\texttt{les} := \texttt{le}_{1..k} \qquad i = 1..k : \mathit{lexpr\_is\_var}(\texttt{le}_i) \xrightarrow{\text{eval}} \texttt{TRUE} \quad \mathit{eval\_call}(\texttt{env}, \texttt{call.name}, \texttt{call.params}, \texttt{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms}, \texttt{env1}) \;//\; \texttt{\#T}, \texttt{\#DE} \quad \mathit{multi\_assign}(\texttt{env1}, \texttt{les}, \texttt{vms}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env}) \;//\; \texttt{\#T}, \texttt{\#DE}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Assign}(\text{LE\_Destructuring}(\texttt{les}), \text{E\_Call}(\texttt{call}))) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

**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)$^{\texttt{\#T,\#DE}}$;
- evaluating the assignable expression `le` with `m` in `env1`, as per Chapter 18, yields Normal(`new_g`, `new_env`)$^{\texttt{\#T,\#DE}}$.

**Formally**

$$\frac{\begin{array}{c}\left(\begin{array}{c}\mathit{ast\_label}(\texttt{le}) \neq \mathsf{LE\_Destructuring} \lor \\ \mathit{ast\_label}(\texttt{re}) \neq \mathsf{E\_Call} \lor \\ \texttt{le} = \mathsf{LE\_Destructuring}(\texttt{les}) \land \exists \texttt{i} \in \mathit{indices}(\texttt{le}).\ \neg \mathit{lexpr\_is\_var}(\texttt{le}[\texttt{i}])\end{array}\right) \\ \mathit{eval\_expr}(\texttt{env}, \texttt{re}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{m}, \texttt{env1}) \ \ /\!/ \ \ \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_lexpr}(\texttt{env1}, \texttt{le}, \texttt{m}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env}) \ \ /\!/ \ \ \texttt{\#T}, \texttt{\#DE}\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Assign}(\texttt{le}, \texttt{re})) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

## 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

$$\mathit{make\_setter}(\overbrace{\mathsf{call}}^{\texttt{call}}, \overbrace{\mathsf{expr}}^{\texttt{arg}}) \longrightarrow \overbrace{\mathsf{call}}^{\texttt{call'}}$$

constructs a setter call `call'` using a base call `call` and right-hand side `arg`.

$$\mathit{make\_setter}(\texttt{call}, \texttt{arg}) \longrightarrow \overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{call}.\text{name}, \\ \text{params} & : & \texttt{call}.\text{params}, \\ \text{args} & : & [\texttt{arg}] + \texttt{call}.\text{args}, \\ \text{call\_type} & : & \mathsf{ST\_Setter}\end{array}\right\}}^{\texttt{call'}}$$

#### ASTRule.DesugarSetter

The helper function

$$\mathit{desugar\_setter}(\overbrace{\mathsf{call}}^{\texttt{call}}, \overbrace{\mathsf{lhs\_access}}^{\texttt{lhs\_access}}, \overbrace{\mathsf{expr}}^{\texttt{rhs}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

builds a statement `new_s` from an assignment of expression `rhs` to a setter invocation `call`.name with accesses given by `lhs_access`.

$$\frac{\begin{array}{c}\text{EMPTY}\\ \texttt{lhs\_access}.\text{access} = [\,] \qquad \texttt{lhs\_access}.\text{slices} = [\,]\\ \mathit{make\_setter}(\texttt{call}, \texttt{rhs}) \longrightarrow \texttt{call'}\end{array}}{\mathit{desugar\_setter}(\texttt{call}, \texttt{lhs\_access}, \texttt{rhs}) \xrightarrow{\text{ast}} \mathsf{S\_Call}(\texttt{call'})}$$

$$\frac{\begin{array}{c}\text{NON\_EMPTY}\\ \texttt{lhs\_access}.\text{access} \neq [\,] \vee \texttt{lhs\_access}.\text{slices} \neq [\,]\\ \texttt{x} \in \mathbb{I} \text{ is fresh}\\ \mathit{desugar\_lhs\_access}(\texttt{x}, \texttt{lhs\_access}) \xrightarrow{\text{ast}} \texttt{lhs}\\ \texttt{modify} := \mathsf{S\_Assign}(\texttt{lhs}, \texttt{rhs}) \qquad \mathit{read\_modify\_write}(\texttt{call}, \texttt{x}, \texttt{modify}) \xrightarrow{\text{ast}} \texttt{call'}\end{array}}{\mathit{desugar\_setter}(\texttt{call}, \texttt{lhs\_access}, \texttt{rhs}) \xrightarrow{\text{ast}} \texttt{call'}}$$

### ASTRule.DesugarSetterSetfields

The helper function

$$\mathit{desugar\_setter\_setfields}(\overbrace{\mathsf{call}}^{\texttt{call}}, \overbrace{\mathsf{identifier}^*}^{\texttt{fields}}, \overbrace{\mathsf{expr}}^{\texttt{rhs}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

builds a statement `new_s` from an assignment of `rhs` to a setter invocation `call`.name with concatenated field accesses given by `fields`.

$$\frac{\begin{array}{c}\texttt{x} \in \mathbb{I} \text{ is fresh}\\ \texttt{lhs} := \mathsf{LE\_SetFields}(\mathsf{LE\_Var}(\texttt{x}), \texttt{fields})\\ \texttt{modify} := \mathsf{S\_Assign}(\texttt{lhs}, \texttt{rhs}) \qquad \mathit{read\_modify\_write}(\texttt{call}, \texttt{x}, \texttt{modify}) \xrightarrow{\text{ast}} \texttt{call'}\end{array}}{\mathit{desugar\_setter\_setfields}(\texttt{call}, \texttt{fields}, \texttt{rhs}) \xrightarrow{\text{ast}} \texttt{call'}}$$

### ASTRule.ReadModifyWrite

The helper function

$$\mathit{read\_modify\_write}(\overbrace{\mathsf{call}}^{\texttt{call}}, \overbrace{\mathsf{identifier}}^{\texttt{x}}, \overbrace{\mathsf{stmt}}^{\texttt{modify}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

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.

$$\frac{\begin{array}{c}\mathit{set\_call\_type}(\texttt{call}, \mathsf{ST\_Getter}) \longrightarrow \texttt{getter}\\ \texttt{read} := \mathsf{S\_Decl}(\mathsf{LDK\_Var}, \mathsf{LDI\_Var}(\texttt{x}), \mathsf{None}, \langle \mathsf{E\_Call}(\texttt{getter}) \rangle)\\ \mathit{make\_setter}(\texttt{call}, \mathsf{E\_Var}(\texttt{x})) \longrightarrow \texttt{setter}\end{array}}{\mathit{read\_modify\_write}(\texttt{call}, \texttt{x}, \texttt{modify}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Seq}(\mathsf{S\_Seq}(\texttt{read}, \texttt{modify}), \mathsf{S\_Call}(\texttt{setter}))}^{\texttt{new\_s}}}$$

**ASTRule.SetterAssign**

NO_SLICES

$$\frac{\begin{array}{c}\mathit{build\_access}(\texttt{setter\_access}) \xrightarrow{\text{ast}} \texttt{access} \\ \texttt{lhs\_access} := \left\{\begin{array}{lcl}\text{access} & : & \texttt{access},\\ \text{slices} & : & [\,]\end{array}\right\} \\ \mathit{desugar\_setter}(\overline{\texttt{call}}, \texttt{lhs\_access}, \overline{\texttt{expr}}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{call}, \texttt{setter\_access} : \texttt{setter\_access}, \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

SLICES

$$\frac{\begin{array}{c}\mathit{build\_access}(\texttt{setter\_access}) \xrightarrow{\text{ast}} \texttt{access} \\ \texttt{lhs\_access} := \left\{\begin{array}{lcl}\text{access} & : & \texttt{access},\\ \text{slices} & : & \overline{\texttt{slices}}\end{array}\right\} \\ \mathit{desugar\_setter}(\overline{\texttt{call}}, \texttt{lhs\_access}, \overline{\texttt{expr}}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{call}, \texttt{setter\_access} : \texttt{setter\_access}, \texttt{slices}, \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

SETFIELDS

$$\frac{\begin{array}{c}\mathit{build\_clist}[\mathit{build\_identity}](\texttt{fields}) \xrightarrow{\text{ast}} \texttt{field\_asts} \\ \mathit{desugar\_setter\_setfields}(\overline{\texttt{call}}, \texttt{field\_asts}, \overline{\texttt{expr}}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{call}, \texttt{"."}, \texttt{"["}, \texttt{fields} : \text{clist2}(\textbf{ID}), \texttt{"]"}, \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

### 20.3.2 Typing and semantics

As given by applying the relevant rules to the desugared AST.

## 20.4 Declaration Statements

### 20.4.1 Syntax

```
stmt ⟶ local_decl_keyword_non_var decl_item option(as_ty) "=" expr ";"
     | "var" decl_item option(as_ty) option("=" expr) ";"
     | "var" clist2(ID) as_ty ";"
```

### 20.4.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_Decl}(\text{local\_decl\_keyword}, \text{local\_decl\_item}, \text{ty}?, \text{expr}?)$$

**ASTRule.SDecl**

LET_CONSTANT

$$\frac{\mathit{build\_option}[\mathit{build\_as\_ty}](\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{local\_decl\_keyword\_non\_var}, \texttt{decl\_item}, \texttt{t} : \texttt{option}(\texttt{as\_ty}), \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Decl}(\overline{\texttt{local\_decl\_keyword}}, \overline{\texttt{decl\_item}}, \texttt{t\_ast}, \langle\overline{\texttt{expr}}\rangle)}^{\texttt{ast\_node}}}$$

VAR

$$\frac{\mathit{build\_option}[\mathit{build\_as\_ty}](\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast} \qquad \mathit{build\_option}[\mathit{build\_expr}](\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"var"}, \texttt{decl\_item}, \texttt{t} : \texttt{option}(\texttt{as\_ty}), \texttt{e} : \texttt{option}(\texttt{"="}, \texttt{expr}), \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Decl}(\texttt{LDK\_Var}, \overline{\texttt{decl\_item}}, \texttt{t\_ast}, \texttt{e\_ast})}^{\texttt{ast\_node}}}$$

MULTI_VAR

$$\frac{\begin{array}{c}\mathit{build\_clist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast} \qquad \mathit{build\_as\_ty}(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast} \\ \texttt{stmts} := [\texttt{x} \in \texttt{ids\_ast} : \texttt{S\_Decl}(\texttt{LDK\_Var}, \texttt{x}, \texttt{t\_ast}, \texttt{None})] \\ \mathit{stmt\_from\_list}(\texttt{stmts}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"var"}, \texttt{ids} : \texttt{clist2}(\textbf{ID}), \texttt{t} : \texttt{as\_ty}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

### 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
    constant c1: integer{1..1000} = 42;
    constant 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 and Listing 20.6 are ill-typed, since constant storage elements and immutable local storage elements require an initializing expression.

Listing 20.5: An uninitialized constant declaration

```
func main() => integer
begin
    // Illegal: constant storage elements require initialization.
    constant c3;
    return 0;
end;
```

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:
  - `s` is a declaration with an initializing expression `e`, that is, S_Decl(`ldk`, `ldi`, `ty_opt`, ⟨`e`⟩);
  - annotating the right-hand-side expression `e` in tenv yields (`t_e`, `e'`, `ses_e`)[//#TE];
  - applying *annotate_local_decl_type_annot* to the environment tenv, type annotation `ty_opt`, type `t_e`, local declaration keyword `ldk`, expression `e'`, and local declaration item `ldi` yields (`tenv1`, `ty_opt'`, `ses_ldi`)[//#TE];
  - define `ses` as the union of `ses_e` and `ses_ldi`;
  - One of the following applies:
    - All of the following apply (CONSTANT):
      - `ldk` indicates a local constant declaration, that is, LDK_Constant;
      - checking that all time frames in `ses_e` are before Constant yields TRUE[//#TE];
      - symbolically simplifying `e` in `tenv1` yields the literal `v`[//#TE];
      - declaring a local constant with literal `v` and local declaration item `ldi` in `tenv1` yields `new_tenv`;
      - `new_s` is a declaration with `ldk`, `ldi`, type annotation `ty_opt'`, and an expression `e'`.
    - All of the following apply (NON_CONSTANT):
      - `ldk` indicates that this is not a local constant declaration, that is, `ldk` ≠ LDK_Constant;
      - `new_s` is a declaration with `ldk`, `ldi`, type annotation `ty_opt'`, and an expression `e'`;
      - `new_tenv` is `tenv1`.

- All of the following apply (NONE):
  - `s` is a local declaration statement with a variable keyword and no initializing expression, that is, S_Decl(LDK_Var, `ldi`, `ty_opt`, `None`) (local declarations of `let` variables and constants require an initializing expression, otherwise they are rejected by an ASL parser);
  - `ty_opt` is $\langle$`t`$\rangle^{\text{// \#TE}}$;
  - annotating `t` in `tenv` yields (`t'`, `ses`)$^{\text{// \#TE}}$;
  - applying *base_value* to `t'` in `tenv` yields `e_init`$^{\text{// \#TE}}$;
  - annotating the local declaration item `ldi` with the type `t'` and local declaration keyword LDI_Var yields `new_tenv`$^{\text{// \#TE}}$;
  - 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`, $\langle$`e_init`$\rangle$).

**Formally**

CONSTANT

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses\_e}) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_local\_decl\_type\_annot}(\mathsf{tenv}, \mathtt{ty\_opt}, \mathtt{t\_e}, \mathtt{ldk}, \mathtt{e'}, \mathtt{ldi}) \xrightarrow{\text{type}} \\ (\mathtt{tenv1}, \mathtt{ty\_opt'}, \mathtt{ses\_ldi}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_e} \cup \mathtt{ses\_ldi} \\ \text{******** common prefix ********} \\ \mathtt{ldk} = \mathsf{LDK\_Constant} \\ \mathit{check}(\mathit{ses\_is\_before}(\mathtt{ses\_e}, \mathsf{Constant}), \mathtt{TE\_SEV}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{static\_eval}(\mathtt{tenv1}, \mathtt{e}) \xrightarrow{\text{type}} \mathtt{v} \;/\!/\; \mathtt{\#TE} \\ \mathit{declare\_local\_constant}(\mathtt{tenv1}, \mathtt{v}, \mathtt{ldi}) \xrightarrow{\text{type}} \mathtt{new\_tenv} \\ \mathtt{new\_s} := \mathsf{S\_Decl}(\mathsf{LDK\_Constant}, \mathtt{ldi}, \mathtt{ty\_opt'}, \langle \mathtt{e'} \rangle)\end{array}}{\mathit{annotate\_stmt}(\mathsf{tenv}, \overbrace{\mathsf{S\_Decl}(\mathtt{ldk}, \mathtt{ldi}, \mathtt{ty\_opt}, \langle \mathtt{e} \rangle)}^{\mathtt{s}}) \xrightarrow{\text{type}} (\mathtt{new\_s}, \mathtt{new\_tenv}, \mathtt{ses})}$$

NON_CONSTANT

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses\_e}) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_local\_decl\_type\_annot}(\mathsf{tenv}, \mathtt{ty\_opt}, \mathtt{t\_e}, \mathtt{ldk}, \mathtt{e'}, \mathtt{ldi}) \xrightarrow{\text{type}} \\ (\mathtt{tenv1}, \mathtt{ty\_opt'}, \mathtt{ses\_ldi}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_e} \cup \mathtt{ses\_ldi} \\ \text{******** common prefix ********} \\ \mathtt{ldk} \neq \mathsf{LDK\_Constant} \qquad \mathtt{new\_s} := \mathsf{S\_Decl}(\mathtt{ldk}, \mathtt{ldi}, \mathtt{ty\_opt'}, \langle \mathtt{e'} \rangle)\end{array}}{\mathit{annotate\_stmt}(\mathsf{tenv}, \overbrace{\mathsf{S\_Decl}(\mathtt{ldk}, \mathtt{ldi}, \mathtt{ty\_opt}, \langle \mathtt{e} \rangle)}^{\mathtt{s}}) \xrightarrow{\text{type}} (\mathtt{new\_s}, \overbrace{\mathtt{tenv1}}^{\mathtt{new\_tenv}}, \mathtt{ses})}$$

NONE

$$\frac{\begin{array}{c}\mathit{check}(\texttt{ty\_opt} = \langle \_ \rangle, \mathsf{TypeError}(\texttt{TE\_BD})) \xrightarrow{\text{type}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{ty\_opt} \overset{\text{is}}{=} \langle \texttt{t} \rangle \qquad \mathit{annotate\_type}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} (\texttt{t'}, \texttt{ses}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{base\_value}(\texttt{tenv}, \texttt{t'}) \xrightarrow{\text{type}} \texttt{e\_init} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{annotate\_local\_decl\_item}(\texttt{tenv}, \texttt{t'}, \mathsf{LDK\_Var}, \mathsf{None}, \texttt{ldi'}) \xrightarrow{\text{type}} \texttt{new\_tenv} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{new\_s} := \mathsf{S\_Decl}(\mathsf{LDK\_Var}, \texttt{ldi}, \langle \texttt{t'} \rangle, \langle \texttt{e\_init} \rangle)\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\mathsf{S\_Decl}(\mathsf{LDK\_Var}, \texttt{ldi}, \texttt{ty\_opt}, \mathsf{None})}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{new\_s}, \texttt{new\_tenv}, \texttt{ses})}$$

### TypingRule.DeclareLocalConstant

The helper function

$$\mathit{declare\_local\_constant}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{literal}}^{\texttt{v}}, \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ldi}}) \xrightarrow{\text{type}} \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

adds the literal `v` with the local declaration item `ldi` as a constant to the local component of the static environment `tenv`, yielding the modified static environment `new_tenv`.

### Example: Typing Local Constant Declarations

In Listing 20.4, the declaration statement `constant c1 : integer{1..1000} = 42;` updates the static environment by binding `c1` to $\mathsf{L\_Int}(42)$.

### Prose

One of the following applies:

- All of the following apply (VAR):
  - `ldi` corresponds to a variable declaration for `x`, that is, $\mathsf{LDI\_Var}(\texttt{x})$;
  - applying *add_local_constant* to `x` and `v` in `tenv` yields `new_tenv`.
- All of the following apply (TUPLE):
  - `ldi` corresponds to a tuple declaration, that is, $\mathsf{LDI\_Var}(\_)$;
  - this case is not yet implemented.

### Formally

VAR

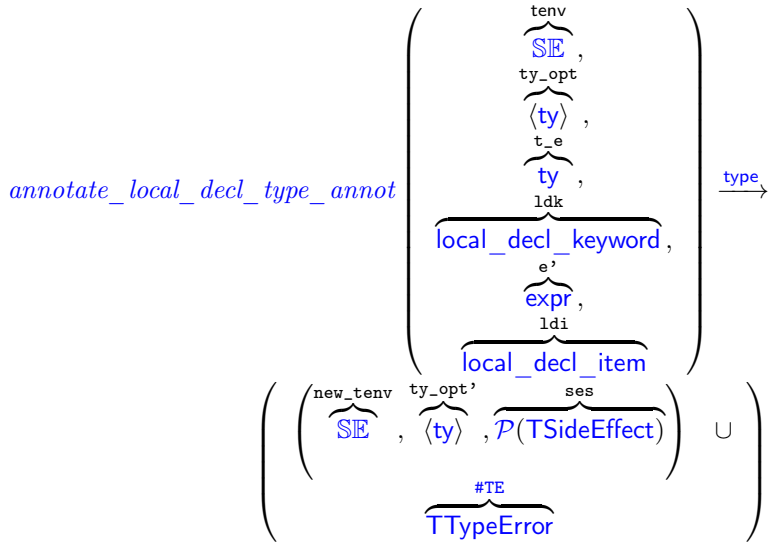
$$\frac{\mathit{add\_local\_constant}(\texttt{tenv}, \texttt{x}, \texttt{v}) \xrightarrow{\text{type}} \texttt{new\_tenv}}{\mathit{declare\_local\_constant}(\texttt{tenv}, \texttt{v}, \overbrace{\mathsf{LDI\_Var}(\texttt{x})}^{\texttt{ldi}}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

TUPLE

$$\mathit{declare\_local\_constant}(\texttt{tenv}, \texttt{v}, \overbrace{\mathsf{LDI\_Tuple}(\_)}^{\texttt{ldi}}) \xrightarrow{\text{type}} \texttt{not implemented yet}$$

### TypingRule.AnnotateLocalDeclTypeAnnot

The helper function




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`$^{/\!/\texttt{\#TE}}$;
  - `new_tenv` is the result of *annotate_local_decl_item*(`tenv`, `t_e`, `ldk`, ⟨`e'`⟩, `ldi`)$^{/\!/\texttt{\#TE}}$;
  - `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'`$^{/\!/\texttt{\#TE}}$;

* 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

$$\frac{\mathit{annotate\_local\_decl\_item}(\texttt{tenv}, \texttt{t\_e}, \texttt{ldk}, \langle \texttt{e'} \rangle, \texttt{ldi}) \xrightarrow{\text{type}} \texttt{new\_tenv} \;/\!/\; \#TE}{\mathit{annotate\_local\_decl\_type\_annot}(\texttt{tenv}, \texttt{None}, \texttt{t\_e}, \texttt{ldk}, \texttt{e'}, \texttt{ldi}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \overbrace{\texttt{None}}^{\texttt{ty\_opt'}}, \overbrace{\emptyset}^{\texttt{ses}})}$$

SOME

$$\frac{\begin{array}{c}\mathit{get\_structure}(\texttt{tenv}, \texttt{t\_e}) \xrightarrow{\text{type}} \texttt{t\_e'} \;/\!/\; \#TE \\ \mathit{inherit\_integer\_constraints}(\texttt{t}, \texttt{t\_e'}) \xrightarrow{\text{type}} \texttt{t'} \;/\!/\; \#TE \\ \mathit{annotate\_type}(\texttt{tenv}, \texttt{t'}) \xrightarrow{\text{type}} (\texttt{t''}, \texttt{ses}) \;/\!/\; \#TE \\ \mathit{check\_can\_be\_initialized\_with}(\texttt{tenv}, \texttt{t''}, \texttt{t\_e}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \#TE \\ \mathit{annotate\_local\_decl\_item}(\texttt{tenv}, \texttt{t''}, \texttt{ldk}, \langle \texttt{e'} \rangle, \texttt{ldi'}) \xrightarrow{\text{type}} \texttt{new\_tenv} \;/\!/\; \#TE\end{array}}{\mathit{annotate\_local\_decl\_type\_annot}(\texttt{tenv}, \langle \texttt{t} \rangle, \texttt{t\_e}, \texttt{ldk}, \texttt{e'}, \texttt{ldi}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \overbrace{\langle \texttt{t''} \rangle}^{\texttt{ty\_opt'}}, \texttt{ses})}$$

**TypingRule.InheritIntegerConstraints**

The helper function

$$\mathit{inherit\_integer\_constraints}(\overbrace{\mathsf{ty}}^{\texttt{lhs}}, \overbrace{\mathsf{ty}}^{\texttt{rhs}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{lhs'}} \cup \overbrace{\mathsf{TTypeError}}^{\#TE}$$

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 13.3 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

INT

$$\frac{\begin{array}{c}\mathit{check\_no\_precision\_loss}(\texttt{rhs}) \xrightarrow{\text{type}} \texttt{TRUE}\ /\!/\ \texttt{\#TE} \\ \mathit{check}(\mathsf{is\_well\_constrained\_integer}(\texttt{rhs}), \texttt{TE\_UT}) \longrightarrow \texttt{TRUE}\ /\!/\ \texttt{\#TE}\end{array}}{\mathit{inherit\_integer\_constraints}(\overbrace{\mathsf{T\_Int}(\mathsf{PendingConstrained})}^{\texttt{lhs}}, \texttt{rhs}) \xrightarrow{\text{type}} \overbrace{\texttt{rhs}}^{\texttt{lhs'}}}$$

TUPLE

$$\frac{\texttt{lhs} = \mathsf{T\_Tuple}(\texttt{lhs\_tys}) \qquad \texttt{rhs} = \mathsf{T\_Tuple}(\texttt{rhs\_tys}) \qquad \mathit{check}(\mathit{equal\_length}(\texttt{lhs\_tys}, \texttt{rhs\_tys}), \mathsf{TE\_UT}) \xrightarrow{\text{type}} \mathsf{TRUE} \;//\; \#\mathsf{TE} \qquad \texttt{i} \in \mathit{indices}(\texttt{lhs}) : \mathit{inherit\_integer\_constraints}(\texttt{lhs\_tys}_{\texttt{i}}, \texttt{rhs\_tys}_{\texttt{i}}) \xrightarrow{\text{type}} \texttt{lhs\_tys'}_{\texttt{i}} \;//\; \#\mathsf{TE}}{\mathit{inherit\_integer\_constraints}(\texttt{lhs}, \texttt{rhs}) \xrightarrow{\text{type}} \overbrace{\mathsf{T\_Tuple}(\texttt{lhs\_tys'})}^{\texttt{lhs'}}}$$

OTHER

$$\frac{\texttt{lhs} \neq \mathsf{T\_Int}(\mathsf{PendingConstrained}) \lor \mathit{ast\_label}(\texttt{lhs}) \neq \mathsf{T\_Tuple} \lor \mathit{ast\_label}(\texttt{rhs}) \neq \mathsf{T\_Tuple}}{\mathit{inherit\_integer\_constraints}(\texttt{lhs}, \texttt{rhs}) \xrightarrow{\text{type}} \overbrace{\texttt{lhs}}^{\texttt{lhs'}}}$$

### TypingRule.CheckNoPrecisionLoss

The helper function

$$\mathit{check\_no\_precision\_loss}(\overbrace{\mathsf{ty}}^{\texttt{t}}) \xrightarrow{\text{type}} \{\mathsf{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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**

$$\textsc{Well-Constrained}\ \frac{\mathit{check}(p = \mathsf{Precision\_Full}, \texttt{PrecisionLostDefining})}{\mathit{check\_no\_precision\_loss}(\overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\_, p))}^{\texttt{t}}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

$$\textsc{Integer}\ \frac{\mathit{ast\_label}(\texttt{c}) \neq \mathsf{WellConstrained}}{\mathit{check\_no\_precision\_loss}(\overbrace{\mathsf{T\_Int}(\texttt{c})}^{\texttt{t}}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

$$\textsc{Other}\ \frac{\mathit{ast\_label}(\texttt{t}) \neq \mathsf{T\_Int}}{\mathit{check\_no\_precision\_loss}(\texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

**TypingRule.CheckCanBeInitializedWith**

The helper function

$$\mathit{check\_can\_be\_initialized\_with}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{s}}, \overbrace{\mathsf{ty}}^{\texttt{t}}) \xrightarrow{\text{type}} \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\texttt{TE}}$$

checks whether an expression of type `s` can be used to initialize a storage element of type `t` in the static environment `tenv`. If the answer if `FALSE`, the result is a type error.

See Example: Type-satisfaction Examples.

**Prose**

One of the following applies:

- All of the following apply (OKAY):
  - testing whether `t` type-satisfies `s` in `tenv` yields `TRUE`;
  - the result is `TRUE`.

- All of the following apply (ERROR):
  - testing whether `t` type-satisfies `s` in `tenv` yields `FALSE`;
  - the result is a type error indicating that an expression of type `s` cannot be used to initialize a storage element of type `t`.

**Formally**

OKAY

$$\frac{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{TRUE}}{\mathit{check\_can\_be\_initialized\_with}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE}}$$

ERROR

$$\frac{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE}}{\mathit{check\_can\_be\_initialized\_with}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_TSF})}$$

### 20.4.4 Semantics

**SemanticsRule.SDeclSome**

**Example: Declaration With an Initializing Value**

In Listing 20.9, `let x = 3;` binds `x` to `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

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m}, \texttt{env1}) \;/\!/\; \#T, \#DE \qquad \mathit{eval\_local\_decl}(\texttt{env1}, \texttt{ldi}, \texttt{m}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Decl}(\_, \texttt{ldi}, \_, \langle \texttt{e} \rangle)) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

NONE

$$\mathit{eval\_stmt}(\texttt{env}, \text{S\_Decl}(\_, \texttt{ldi}, \_, \texttt{None})) \xrightarrow{\text{eval}} \text{DynError}(\texttt{UninitialisedDecl})$$

## 20.5 Declaration statements with an elided parameter

### 20.5.1 Syntax

```
stmt ⟶ local_decl_keyword_non_var decl_item as_ty "="
         ↪ elided_param_call ";"
       | "var" decl_item as_ty "=" elided_param_call ";"
```

```
elided_param_call ⟶ ID "{" "}"
                    | 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`.

$$\frac{\mathit{build\_call}(\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"("}, \texttt{")"})) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_elided\_param\_call}(\texttt{elided\_param\_call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{"\}"})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

$$\frac{\mathit{build\_call}(\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{args})) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\begin{array}{r}\mathit{build\_elided\_param\_call}(\texttt{elided\_param\_call}\\ (\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{"\}"}, \texttt{args} : \mathsf{plist0}(\mathsf{expr}))) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}$$

$$\frac{\mathit{build\_call}(\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{params}, \texttt{"\}"})) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\begin{array}{r}\mathit{build\_elided\_param\_call}(\texttt{elided\_param\_call}\\ (\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{","}, \texttt{params} : \mathsf{clist1}(\mathsf{expr}), \texttt{"\}"})) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}$$

$$\frac{\mathit{build\_call}(\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{params}, \texttt{"\}"}, \texttt{args})) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\begin{array}{r}\mathit{build\_elided\_param\_call}(\texttt{elided\_param\_call}\\ (\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{","}, \texttt{params} : \mathsf{clist1}(\mathsf{expr}), \texttt{"\}"}, \texttt{args} : \mathsf{plist0}(\mathsf{expr}))) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}$$

### ASTRule.DesugarElidedParameter

The helper function

$$\mathit{desugar\_elided\_parameter}(\overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}, \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ldi}}, \overbrace{\mathsf{ty}}^{\texttt{t}}, \overbrace{\mathsf{call}}^{\texttt{call}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}} \cup \overbrace{\mathsf{TBuildError}}^{\#\text{BE\_PE}}$$

builds a declaration statement `new_s` from an assignment of the call `call` to the left-hand side `ldi` with keyword `ldk` and type annotation `t`, where the call has an elided parameter. Otherwise, the result is a parse error.

### 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 - : bits(64) = Zeros{}; // illegal: parsing conflict with empty record

    - = UInt('1111'); // equivalent to UInt{4}('1111');
    let d : bits(64) = ZeroExtend{64}('11'); // equivalent to ZeroExtend{64,2}
    let e : bits(64) = ZeroExtend{}('11'); // can also elide the output parameter N
    return 0;
end;
```

$$\frac{\mathit{check}(\texttt{t} = \mathsf{T\_Bits}(\_,\_), \texttt{\#BE\_PE}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#BE\_PE} \qquad \texttt{t} \overset{\text{is}}{=} \mathsf{T\_Bits}(\texttt{e},\_) \qquad \texttt{call'} := \texttt{call}[\text{params} \mapsto [\texttt{e}] + \texttt{call.params}]}{\mathit{desugar\_elided\_parameter}(\texttt{ldk},\texttt{ldi},\texttt{t},\texttt{call}) \xrightarrow{\text{ast}} \mathsf{S\_Decl}(\texttt{ldk},\texttt{ldi},\langle\texttt{t}\rangle,\mathsf{E\_Call}(\texttt{call'}))}$$

**ASTRule.ElidedParamDecl**

$$\frac{\mathit{build\_elided\_param\_call}(\texttt{call}) \xrightarrow{\text{ast}} \texttt{call\_ast} \qquad \mathit{desugar\_elided\_parameter}(\overline{\texttt{local\_decl\_keyword}},\overline{\texttt{decl\_item}},\overline{\texttt{as\_ty}},\texttt{call\_ast}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_stmt}\left(\texttt{stmt}\left(\begin{array}{l}\texttt{local\_decl\_keyword\_non\_var},\\ \hookrightarrow \texttt{decl\_item},\texttt{as\_ty},\texttt{"="},\\ \hookrightarrow \texttt{call} : \texttt{elided\_param\_call},\texttt{";"}\end{array}\right)\right) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

$$\frac{\mathit{build\_elided\_param\_call}(\texttt{call}) \xrightarrow{\text{ast}} \texttt{call\_ast} \qquad \mathit{desugar\_elided\_parameter}(\mathsf{LDK\_Var},\overline{\texttt{decl\_item}},\overline{\texttt{as\_ty}},\texttt{call\_ast}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_stmt}(\texttt{stmt}(\texttt{"var"},\texttt{decl\_item},\texttt{as\_ty},\texttt{"="},\texttt{call} : \texttt{elided\_param\_call},\texttt{";"})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

### 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

$\texttt{stmt\_list} \longrightarrow \mathsf{list1}(\texttt{stmt})$

### 20.6.2 Abstract Syntax

$\mathsf{stmt} \longrightarrow \mathsf{S\_Seq}(\mathsf{stmt}, \mathsf{stmt})$

#### ASTRule.StmtList

The function

$$\mathit{build\_stmt\_list}(\overbrace{\mathbb{PARSE}[\texttt{stmt\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_list}[\texttt{stmt}](\texttt{stmts}) \xrightarrow{\text{ast}} \texttt{stmt\_list} \qquad \mathit{stmt\_from\_list}(\texttt{stmt\_list}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_stmt\_list}(\texttt{stmt\_list}(\texttt{stmts} : \mathsf{list1}(\texttt{stmt}))) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

#### ASTRule.StmtFromList

The helper function

$$\mathit{stmt\_from\_list}(\overbrace{\mathsf{stmt}^*}^{\texttt{stmts}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

builds a statement `new_s` from a possibly-empty list of statements `stmts`.

EMPTY

$$\frac{}{\mathit{stmt\_from\_list}(\overbrace{[\,]}^{\texttt{stmts}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pass}}^{\texttt{new\_s}}}$$

NON_EMPTY

$$\frac{\mathit{stmt\_from\_list}(\texttt{stmts1}) \xrightarrow{\text{ast}} \texttt{s1} \qquad \mathit{sequence\_stmts}(\texttt{s}, \texttt{s1}) \xrightarrow{\text{ast}} \texttt{new\_s}}{\mathit{stmt\_from\_list}(\overbrace{[\texttt{s}] + \texttt{stmts1}}^{\texttt{stmts}}) \xrightarrow{\text{ast}} \texttt{new\_s}}$$

### ASTRule.SequenceStmts

The helper function

$$\mathit{sequence\_stmts}(\overbrace{\mathsf{stmt}}^{\texttt{s1}}, \overbrace{\mathsf{stmt}}^{\texttt{s2}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

Combines the statement `s1` with `s2` into the statement `new_s`, while filtering away instances of S_Pass.

S1_SPASS

$$\mathit{sequence\_stmts}(\overbrace{\mathsf{S\_Pass}}^{\texttt{s1}}, \texttt{s2}) \xrightarrow{\text{ast}} \overbrace{\texttt{s2}}^{\texttt{new\_s}}$$

S2_SPASS

$$\frac{\texttt{s1} \neq \mathsf{S\_Pass}}{\mathit{sequence\_stmts}(\texttt{s1}, \overbrace{\mathsf{S\_Pass}}^{\texttt{s2}}) \xrightarrow{\text{ast}} \overbrace{\texttt{s1}}^{\texttt{new\_s}}}$$

NO_SPASS

$$\frac{\texttt{s1} \neq \mathsf{S\_Pass} \qquad \texttt{s2} \neq \mathsf{S\_Pass}}{\mathit{sequence\_stmts}(\texttt{s1}, \texttt{s2}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Seq}(\texttt{s1}, \texttt{s2})}^{\texttt{new\_s}}}$$

## 20.6.3 Typing

### TypingRule.SSeq

### 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 $(\texttt{new\_s2}, \texttt{new\_tenv}, \texttt{ses2})^{/\!/ \texttt{\#TE}}$;
- `new_s` is the AST node for the sequence of statements `new_s1` and `new_s2`, that is, S_Seq(`new_s1`, `new_s2`);
- define `ses` as the union of `ses1` and `ses2`.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_stmt}(\texttt{tenv}, \texttt{s1}) \xrightarrow{\text{type}} (\texttt{new\_s1}, \texttt{tenv1}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_stmt}(\texttt{tenv1}, \texttt{s2}) \xrightarrow{\text{type}} (\texttt{new\_s2}, \texttt{new\_tenv}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses1} \cup \texttt{ses2}\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\text{S\_Seq}(\texttt{s1}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Seq}(\texttt{new\_s1}, \texttt{new\_s2})}^{\texttt{new\_s}}, \texttt{new\_tenv}, \texttt{ses})}$$

### 20.6.4 Semantics

**SemanticsRule.SSeq**

**Example: 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 $(\text{Returning}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env}))^{/\!/ \texttt{\#T,\#DE}}$;
- evaluating `s2` in `env1` yields a non-abnormal configuration (either Normal or Continuing) $C^{/\!/ \texttt{\#T,\#DE}}$;
- `new_g` is the ordered composition of `g1` and the execution graph of $C$ with the `asl_po` edge;
- $D$ is the configuration $C$ with the execution graph component replaced with `new_g`.

**Formally**

$$\frac{\begin{array}{c}\mathit{eval\_stmt}(\texttt{env}, \texttt{s1}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g1}, \texttt{env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_stmt}(\texttt{env1}, \texttt{s2}) \xrightarrow{\text{eval}} C \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_po}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{new\_g})\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Seq}(\texttt{s1}, \texttt{s2})) \xrightarrow{\text{eval}} D}$$

## 20.7 Call Statements

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 ⟶ call";"

### 20.7.2 Abstract Syntax

stmt ⟶ S_Call(call)

#### ASTRule.SCall

$$\frac{\mathit{build\_call}(\texttt{call}) \xrightarrow{\text{ast}} \texttt{call\_ast} \qquad \mathit{set\_call\_type}(\texttt{call\_ast}) \longrightarrow \texttt{call'}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{call} : \texttt{call}, ";")}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Call}(\texttt{call'})}^{\texttt{ast\_node}}}$$

### 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`)$^{/\!/\texttt{\#TE}}$;
- `new_s` is the call using `call'`, that is, S_Call(`call'`);
- `new_tenv` is tenv.

#### Formally

$$\frac{\mathit{annotate\_call}(\texttt{call}) \xrightarrow{\text{type}} (\texttt{call'}, \texttt{None}, \texttt{ses}) \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Call}(\texttt{call})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\texttt{S\_Call}(\texttt{call'})}^{\texttt{new\_s}}, \texttt{tenv}, \texttt{ses})}$$

### 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`)$^{/\!/\texttt{\#T,\#DE}}$;
- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

**Formally**

$$\frac{\mathit{eval\_call}(\texttt{env}, \texttt{call.name}, \texttt{call.params}, \texttt{call.args}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE}}{\mathit{eval\_stmt}(\texttt{env}, \overbrace{\mathsf{S\_Call}(\texttt{call})}^{\texttt{s}}) \xrightarrow{\text{eval}} \mathsf{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

## 20.8 Conditional Statements

### 20.8.1 Syntax

```
  stmt ⟶ "if" expr "then" stmt_list s_else "end" ";"
s_else ⟶ "elsif" expr "then" stmt_list s_else
         | "else" stmt_list
         | ϵ
```

### 20.8.2 Abstract Syntax

$$\mathsf{stmt} \longrightarrow \mathsf{S\_Cond}(\mathsf{expr}, \mathsf{stmt}, \mathsf{stmt})$$

**ASTRule.SCond**

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"if"}, \texttt{expr}, \texttt{"then"}, \texttt{stmt\_list}, \texttt{s\_else}, \texttt{"end"}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Cond}(\overline{\texttt{expr}}, \overline{\texttt{stmt\_list}}, \overline{\texttt{else}})}^{\texttt{ast\_node}}$$

**ASTRule.SElse**

The function

$$\mathit{build\_s\_else}(\overbrace{\mathbb{PARSE}[\texttt{s\_else}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

ELSEIF

$$\mathit{build\_s\_else}(\texttt{s\_else}(\texttt{"elsif"}, \texttt{expr}, \texttt{"when"}, \texttt{stmt\_list}, \texttt{s\_else})) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Cond}(\overline{\texttt{expr}}, \overline{\texttt{stmt\_list}}, \overline{\texttt{s\_else}})}^{\texttt{ast\_node}}$$

PASS

$$\mathit{build\_s\_else}(\texttt{s\_else}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pass}}^{\texttt{ast\_node}}$$

ELSE

$$\mathit{build\_s\_else}(\texttt{s\_else}(\texttt{"else"}, \texttt{stmt\_list})) \xrightarrow{\text{ast}} \overbrace{\texttt{stmt\_list}}^{\texttt{ast\_node}}$$

### 20.8.3 Typing

#### TypingRule.SCond

The specifications in Listing 20.14, Listing 20.15, Listing 20.16, and Listing 20.17 are all well-typed.

#### Prose

All of the following apply:

- `s` is a condition `e` with the statements `s1` and `s2`, that is, S_Cond(`e`, `s1`, `s2`);
- annotating the right-hand-side expression `e` in `tenv` yields (`t_cond`, `e_cond`, `ses_cond`)$^{/\!/\texttt{\#TE}}$;
- checking that `t_cond` type-satisfies T_Bool yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- annotating the statement `s1` in `tenv` yields (`s1'`, `ses1`)$^{/\!/\texttt{\#TE}}$;
- annotating the statement `s2` in `tenv` yields (`s2'`, `ses2`)$^{/\!/\texttt{\#TE}}$;
- `new_s` is the condition `e_cond` with the statements `s1'` and `s2'`, that is, S_Cond(`e_cond`, `s1'`, `s2'`);
- `new_tenv` is `tenv`;
- define `ses` as the union of `ses_cond`, `ses1`, and `ses2`.

#### Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\text{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_cond}, \texttt{e\_cond}, \texttt{ses\_cond}) \;/\!/\; \texttt{\#TE} \\ \mathit{checked\_typesat}(\text{tenv}, \texttt{t\_cond}, \texttt{T\_Bool}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_block}(\text{tenv}, \texttt{s1}) \xrightarrow{\text{type}} (\texttt{s1'}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_block}(\text{tenv}, \texttt{s2}) \xrightarrow{\text{type}} (\texttt{s2'}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \text{ses} := \texttt{ses\_cond} \cup \texttt{ses1} \cup \texttt{ses2}\end{array}}{\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\texttt{S\_Cond}(\texttt{e}, \texttt{s1}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\texttt{S\_Cond}(\texttt{e\_cond}, \texttt{s1'}, \texttt{s2'})}^{\texttt{new\_s}}, \overbrace{\text{tenv}}^{\texttt{new\_tenv}})}$$

### 20.8.4 Semantics

#### SemanticsRule.SCond

#### Example: Conditional Statements

The specification in Listing 20.14 does not result in any Assertion Error.

Listing 20.14: Evaluating a conditional statement

```
func main () => integer
begin

  if TRUE then
    assert TRUE;
  else
    assert FALSE;
  end;

  return 0;
end;
```

The specification in Listing 20.15 does not result in any error.

Listing 20.15: 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 20.16 results in an Assertion Error.

Listing 20.16: 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 20.17 does not result in any error.

Listing 20.17: 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` is `b` is `TRUE` and `s2` otherwise (so that `s1` will be evaluated if the condition evaluates to `TRUE` and otherwise `s2` will be evaluated);
- evaluating `s'` in `env1` as per Chapter 21 is a non-abnormal configuration (either Normal or Continuing) $C^{\#T,\#DE}$;
- `g` is the ordered composition of `g1` and the execution graph of the configuration $C$;
- $D$ is the configuration $C$ with the execution graph component updated to be `g`.

### Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\mathtt{env},\mathtt{e}) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v},\mathtt{g1}),\mathtt{env1}) \;//\; \#T,\#DE \\ \mathtt{v} \overset{\text{is}}{=} \text{Bool}(\mathtt{b}) \qquad \mathtt{s'} := \text{choice}(\mathtt{b},\mathtt{s1},\mathtt{s2}) \qquad \mathit{eval\_block}(\mathtt{env1},\mathtt{s'}) \xrightarrow{\text{eval}} C \;//\; \#T,\#DE \\ \mathtt{g} := \mathtt{g1} \xrightarrow{\text{asl\_ctrl}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \mathtt{g}) \end{array}}{\mathit{eval\_stmt}(\mathtt{env}, \overbrace{\text{S\_Cond}(\mathtt{e},\mathtt{s1},\mathtt{s2})}^{\mathtt{s}}) \xrightarrow{\text{eval}} D}$$

## 20.9 Case Statements

Case statements allow executing different statements, based on which condition an expression satisfies.

Listing 20.18 shows an example of a case statement and the output to a console when it is evaluated.

Listing 20.18: 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 20.18 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 20.18 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 20.18 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 20.18 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 20.19 demonstrates how the otherwise case is evaluated.

Listing 20.19: 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 20.20 shows a specification that, when evaluated, yields a dynamic error.

Listing 20.20: 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
```

### 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 20.21 shows an example of how a `case` statement can be transformed into a corresponding compound condition statement.

Listing 20.21: 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 20.22 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 a call to `Unreachable()`, the transformation ensures that a dynamic error occurs when no case alternative is selected.

Listing 20.22: Transforming a case statement with a non-variable as the case discriminant and no otherwise case

```
func main () => integer
begin
  var x : integer;
  // The following case statement:
  case 3 of
    when 42 => x = 42;
    when <= 42 => x = 0;
  end;
  // can be desugared into the following statement:
  let discriminant_var: integer {3} = 3;
  if discriminant_var IN {42} then
    x = 42;
  else
    if discriminant_var IN {<= 42} then
      x = 0;
    else
      Unreachable();
    end;
  end;

  return x;
end;
```

The untyped AST contains non-terminals for case alternatives, which exist only as a data type used by *desugar_case_stmt* and do not later appear in the untyped AST:

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

### ASTRule.SCase

To satisfy Guide.CaseNoOtherwiseError, when no otherwise case exists, S_Unreachable is used instead:

NO_OTHERWISE

$$\frac{\begin{array}{c}\textit{build\_list}[\textit{build\_case\_alt}](\texttt{case\_alt\_list}) \xrightarrow{\text{ast}} \texttt{case\_alt\_list\_ast} \\ \textit{build\_expr}(\texttt{e\_discriminant}) \xrightarrow{\text{ast}} \texttt{e\_discriminant\_ast} \\ \textit{desugar\_case\_stmt}(\texttt{e\_discriminant\_ast}, \texttt{case\_alt\_list\_ast}, \text{S\_Unreachable}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\textit{build\_stmt}\left(\overbrace{\text{stmt}\left(\begin{array}{l}\texttt{"case"}, \texttt{e\_discriminant} : \text{expr}, \texttt{"of"}, \\ \hookrightarrow\ \texttt{case\_alt\_list} : \texttt{case\_alt\_list}, \\ \hookrightarrow\ \texttt{"end"}, \texttt{";"}\end{array}\right)}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

OTHERWISE

$$\frac{\begin{array}{c}\mathit{build\_list}[\mathit{build\_case\_alt}](\texttt{case\_alt\_list}) \xrightarrow{\text{ast}} \texttt{case\_alt\_list\_ast} \\ \mathit{build\_expr}(\texttt{e\_discriminant}) \xrightarrow{\text{ast}} \texttt{e\_discriminant\_ast} \\ \mathit{build\_stmt\_list}(\texttt{otherwise}) \xrightarrow{\text{ast}} \texttt{otherwise\_ast} \\ \mathit{desugar\_case\_stmt}(\texttt{e\_discriminant\_ast}, \texttt{case\_alt\_list\_ast}, \texttt{otherwise\_ast}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{l}\texttt{"case"}, \texttt{e\_discriminant} : \texttt{expr}, \texttt{"of"}, \\ \hookrightarrow\ \texttt{case\_alt\_list} : \texttt{case\_alt\_list}, \\ \hookrightarrow\ \texttt{"otherwise"}, \texttt{"=>"}, \texttt{otherwise} : \texttt{stmt\_list}, \\ \hookrightarrow\ \texttt{"end"}, \texttt{";"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

### ASTRule.CaseAltList

The function

$$\mathit{build\_case\_alt\_list}(\overbrace{\mathbb{PARSE}[\texttt{case\_alt\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{case\_alt}^{+}}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_clist}[\mathit{build\_case\_alt}](\texttt{cases}) \xrightarrow{\text{type}} \texttt{ast\_node}}{\mathit{build\_case\_alt\_list}(\overbrace{\texttt{case\_alt\_list}(\texttt{cases} : \mathsf{clist1}(\texttt{case\_alt}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

### ASTRule.CaseAlt

The function

$$\mathit{build\_case\_alt}(\overbrace{\mathbb{PARSE}[\texttt{case\_alt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{case\_alt}}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_option}[\mathit{build\_expr}](\texttt{where\_opt}) \xrightarrow{\text{ast}} \texttt{where\_ast}}{\mathit{build\_case\_alt}\left(\overbrace{\texttt{case\_alt}\left(\begin{array}{l}\texttt{"when"}, \texttt{pattern\_list}, \\ \hookrightarrow\ \texttt{where\_opt} : \mathsf{option}(\texttt{"where"}, \texttt{expr}), \texttt{"=>"}, \\ \hookrightarrow\ \texttt{stmts} : \texttt{stmt\_list}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{case\_alt}(\text{pattern} : \overline{\texttt{pattern\_list}}, \text{where} : \texttt{where\_ast}, \text{stmt} : \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}}$$

### ASTRule.DesugarCaseStmt

The relation

$$\mathit{desugar\_case\_stmt}(\overbrace{\mathsf{expr}}^{\texttt{e0}}, \overbrace{\mathsf{case\_alt}^*}^{\texttt{cases}}, \overbrace{\mathsf{stmt}}^{\texttt{otherwise}}) \bigtimes \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

transforms a case discriminant `e0`, a list of case alternatives `cases`, and a statement `otherwise` into a statement `new_s`.

VAR

$$\frac{\mathit{ast\_label}(\texttt{e0}) = \mathsf{E\_Var} \qquad \mathit{cases\_to\_cond}(\texttt{e0}, \texttt{cases}, \texttt{otherwise}) \xrightarrow{\text{type}} \texttt{new\_s}}{\mathit{desugar\_case\_stmt}(\texttt{e0}, \texttt{cases}, \texttt{otherwise}) \xrightarrow{\text{ast}} \texttt{new\_s}}$$

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 20.22 for an example):

NON_VAR

$$\frac{\begin{array}{c}\mathit{ast\_label}(\texttt{e0}) \neq \mathsf{E\_Var} \\ \texttt{x} \in \mathbb{I} \text{ is fresh} \qquad \texttt{decl\_x} \overset{\text{is}}{=} \mathsf{S\_Decl}(\mathsf{LDK\_Let}, \mathsf{LDI\_Var}(\texttt{x}), \texttt{None}, \langle \texttt{e0} \rangle) \\ \mathit{cases\_to\_cond}(\mathsf{E\_Var}(\texttt{x}), \texttt{cases}, \texttt{otherwise}) \xrightarrow{\text{type}} \texttt{s\_cond}\end{array}}{\mathit{desugar\_case\_stmt}(\texttt{e0}, \texttt{cases}, \texttt{otherwise}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Seq}(\texttt{decl\_x}, \texttt{s\_cond})}^{\texttt{new\_s}}}$$

### ASTRule.CasesToCond

The function

$$\mathit{cases\_to\_cond}(\overbrace{\mathsf{expr}}^{\texttt{e}}, \overbrace{\mathsf{case\_alt}^*}^{\texttt{cases}}, \overbrace{\mathsf{stmt}}^{\texttt{otherwise}}) \bigtimes \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

transforms an expression `e`, a list of `case` alternatives `cases`, and a statement `otherwise` into a statement `new_s`.

LAST

$$\frac{\mathit{case\_to\_cond}(\texttt{e}, \texttt{case}, \texttt{otherwise}) \xrightarrow{\text{ast}} \texttt{new\_s}}{\mathit{cases\_to\_cond}(\texttt{e}, \overbrace{[\texttt{case}]}^{\texttt{cases}}, \texttt{otherwise}) \xrightarrow{\text{ast}} \texttt{new\_s}}$$

NOT_LAST

$$\frac{\begin{array}{c}\texttt{cases1} \neq [\,] \\ \mathit{cases\_to\_cond}(\texttt{e}, \texttt{cases1}) \xrightarrow{\text{type}} \texttt{s1} \qquad \mathit{case\_to\_cond}(\texttt{e}, \texttt{case}, \texttt{s1}) \xrightarrow{\text{type}} \texttt{new\_s}\end{array}}{\mathit{cases\_to\_cond}(\texttt{e}, \overbrace{[\texttt{case}] + \texttt{cases1}}^{\texttt{cases}}, \texttt{otherwise}) \xrightarrow{\text{ast}} \texttt{new\_s}}$$

**ASTRule.CaseToCond**

The function

$$\mathit{case\_to\_cond}(\overbrace{\mathsf{expr}}^{\texttt{e0}}, \overbrace{\mathsf{case\_alt}}^{\texttt{case}}, \overbrace{\mathsf{stmt}}^{\texttt{tail}}) \times \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

transforms an expression `e0` (the condition used for a `case` statement), a single `case` alternative `case`, and a statement `tail`, which represents a list of `case` alternatives already converted to conditionals, into a condition statement `new_s`.

$$\frac{\begin{array}{c}\texttt{case} \overset{\text{is}}{=} \{\text{pattern} : \texttt{pattern}, \text{where} : \texttt{where}, \text{stmt} : \texttt{stmt}\} \\ \texttt{e\_pattern} := \mathsf{E\_Pattern}(\texttt{e0}, \texttt{pattern}) \\ \texttt{v\_cond} := \mathsf{choice}(\texttt{where} = \langle \texttt{e\_where} \rangle, \mathsf{E\_Binop}(\texttt{BAND}, \texttt{e\_pattern}, \texttt{e\_where}), \texttt{e\_pattern})\end{array}}{\mathit{case\_to\_cond}(\texttt{e0}, \texttt{case}, \texttt{tail}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Cond}(\texttt{v\_cond}, \texttt{stmt}, \texttt{tail})}^{\texttt{new\_s}}}$$

### 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.

## 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 20.23 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 20.23: 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 ⟶ "assert" expr ";"

### 20.10.2 Abstract Syntax

stmt ⟶ S_Assert(expr)

#### ASTRule.SAssert

$$\textit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"assert"}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Assert}(\overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

### 20.10.3 Typing

#### TypingRule.SAssert

#### Example: Typing Assertion Statements

The specifications in Listing 20.25 and Listing 20.26 are well-typed.

The assertion statements in Listing 20.24 are ill-typed, since the expression needs to be both of a boolean type and pure.

Listing 20.24: 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 pure.
    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 pure via *ses_is_pure* 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 the union of `ses_e` and the singleton set for assertion side effect descriptor.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e'}, \mathtt{e'}, \mathtt{ses\_e}) \;/\!/\; \#TE \\ \mathit{check}\left(\mathit{ses\_is\_pure}(\mathtt{ses\_e}) \xrightarrow{\text{type}} \mathtt{TRUE}, \mathtt{TE\_SEV}\right) \\ \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e'}, \mathsf{T\_Bool}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \#TE \\ \mathtt{ses} := \mathtt{ses\_e} \cup \{\mathsf{PerformsAssertions}\}\end{array}}{\mathit{annotate\_stmt}(\mathsf{tenv}, \overbrace{\mathsf{S\_Assert}(\mathtt{e})}^{\mathtt{s}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{S\_Assert}(\mathtt{e'})}^{\mathtt{new\_s}}, \overbrace{\mathsf{tenv}}^{\mathtt{new\_tenv}}, \mathtt{ses})}$$

### 20.10.4 Semantics

**SemanticsRule.SAssert**

**Example: Evaluation of Assertions: Success**

In Listing 20.25, `assert (42 != 3);` ensures that `3` is not equal to `42`.

Listing 20.25: Evaluating an assertion that succeeds

```
func main () => integer
begin

  assert (42 != 3);

  return 0;
end;
```

**Example: Evaluation of Assertions: Failure**

In Listing 20.26, evaluating `assert (42 == 3);` results in an DE_DAF error.

Listing 20.26: 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

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{new\_g}), \texttt{new\_env}) \;//\; \#T, \#DE \qquad \texttt{v} \overset{\text{is}}{=} \text{Bool}(\texttt{TRUE})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Assert}(\texttt{e})) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

ERROR

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \_), \_) \qquad \texttt{v} \overset{\text{is}}{=} \text{Bool}(\texttt{FALSE})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Assert}(\texttt{e})) \xrightarrow{\text{eval}} \text{DynError}(\texttt{DE\_DAF})}$$

## 20.11 While Statements

Listing 20.27: A `while` statement

```
func limit_loop() => integer{4}
begin
  println("evaluated limit = ", 4);
  return 4;
end;

func test_condition(i: integer) => boolean
begin
  constant 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 20.27 specifies a limit via the expression `limit_loop()`.

### 20.11.1 Syntax

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

### 20.11.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_While}(\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{expr?}}^{\text{loop limit}}, \overbrace{\text{stmt}}^{\text{loop body}})$$

**ASTRule.SWhile**

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{limit\_expr}) \xrightarrow{\text{ast}} \texttt{limit\_expr\_ast} \\ \mathit{build\_looplimit}(\texttt{opt\_limit}) \xrightarrow{\text{ast}} \texttt{opt\_limit\_ast}\end{array}}{\mathit{build\_stmt}\left(\overbrace{\text{stmt}\left(\begin{array}{l}\texttt{"while"}, \texttt{e\_cond} : \text{expr}, \texttt{opt\_limit} : \text{loop\_limit}, \texttt{"do"}, \\ \hookrightarrow\ \text{stmt\_list}, \texttt{"end"}, \texttt{";"}\end{array}\right)}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\text{S\_While}(\overline{\texttt{expr}}, \texttt{opt\_limit\_ast}, \overline{\texttt{stmt\_list}})}^{\text{ast\_node}}}$$

**ASTRule.LoopLimit**

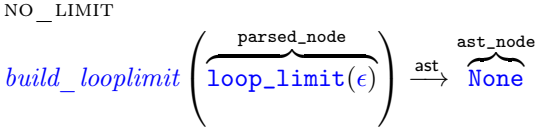
The function

$$\mathit{build\_looplimit}(\overbrace{\mathbb{PARSE}[\texttt{loop\_limit}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\langle \text{expr} \rangle}^{\text{ast\_node}}$$

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

LIMIT

$$\mathit{build\_looplimit}\left(\overbrace{\texttt{loop\_limit}(\texttt{"looplimit"}, \text{expr})}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\langle \overline{\texttt{expr}} \rangle}^{\text{ast\_node}}$$




### 20.11.3 Typing

#### TypingRule.SWhile

#### Example: Typing While Loops

The specification in Listing 20.28 is well-typed and shows two while statement — the first one without a loop limit and the second one with a loop limit.

Listing 20.28: Typing while statements

```
func scan{N}(x: bits(N)) => integer{0..N}
begin
    var res : integer = 0;
    var i: integer = 0;
    // N is a constrained integer, since N is a parameter,
    // and thus can be used as a limit expression.
    while i < N looplimit N do
        if x[i] == '1' then
            res = res + 1;
        end;
        i = i + 1;
    end;
    return res as integer{0..N};
end;

func main () => integer
begin
    var x = Ones{20};
    println(scan{20}(x));

    var i: integer = 0;
    var ones: integer = 0;
    while i < 20 do
        assert i < 20;
        if x[i] == '1' then
            ones = ones + 1;
        end;
        i = i + 1;
    end;
    println(ones);
    return 0;
end;
```

The specification in Listing 20.29 is ill-typed, since the loop limit is not a constrained integer.

Listing 20.29: 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`)$^{/\!/\texttt{\#TE}}$;
- annotating the optional limit expression `limit1` via *annotate_limit_expr* in tenv yields (`limit2`, `ses_limit`)$^{/\!/\texttt{\#TE}}$;
- checking that `t` type-satisfies T_Bool in tenv yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- annotating `s1` as a block statement as per TypingRule.Block in tenv yields (`s2`, `ses_block`)$^{/\!/\texttt{\#TE}}$;
- `new_s` is a `while` statement with expression `e2`, optional limit expression `limit2`, and statement block `s2`, that is, S_While(`e2`, `s2`);
- `new_tenv` is tenv;
- define `ses` as the union of `ses_block`, `ses_e`, and `ses_limit`.

### Formally

$$\frac{\begin{array}{c}\textit{annotate\_expr}(\text{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{e2}, \texttt{ses\_e}) \;/\!/\; \texttt{\#TE} \\ \textit{annotate\_limit\_expr}(\text{tenv}, \texttt{limit1}) \xrightarrow{\text{type}} (\texttt{limit2}, \texttt{ses\_limit}) \;/\!/\; \texttt{\#TE} \\ \textit{checked\_typesat}(\text{tenv}, \texttt{t}, \text{T\_Bool}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \textit{annotate\_block}(\text{tenv}, \texttt{s1}) \xrightarrow{\text{type}} (\texttt{s2}, \texttt{ses\_block}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_block} \cup \texttt{ses\_e} \cup \texttt{ses\_limit}\end{array}}{\textit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_While}(\texttt{e1}, \texttt{limit1}, \texttt{s1})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_While}(\texttt{e2}, \texttt{limit2}, \texttt{s2})}^{\texttt{new\_s}}, \overbrace{\text{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

### TypingRule.AnnotateLimitExpr

The function

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

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 20.28 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 20.29 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 $(\mathtt{limit'}, \mathtt{ses})^{/\!/ \mathtt{\#TE}}$;
  - define `e'` as ⟨`limit'`⟩.

### Formally

NONE

$$\mathit{annotate\_limit\_expr}(\mathsf{tenv}, \overbrace{\mathtt{None}}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathtt{None}}^{\mathtt{e'}}, \overbrace{\emptyset}^{\mathtt{ses}})$$

SOME

$$\frac{\mathit{annotate\_symbolic\_constrained\_integer}(\mathsf{tenv}, \mathtt{limit}) \xrightarrow{\mathsf{type}} (\mathtt{limit'}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE}}{\mathit{annotate\_limit\_expr}(\mathsf{tenv}, \overbrace{\langle \mathtt{limit} \rangle}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\overbrace{\langle \mathtt{limit'} \rangle}^{\mathtt{e'}}, \mathtt{ses})}$$

### 20.11.4 Semantics

#### SemanticsRule.SWhile

#### Example: Evaluation of While Statements

The specification in Listing 20.27 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 20.30 yields a dynamic error once the loop limit for the while statement is reached.

Listing 20.30: 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 $(\texttt{limit\_opt}, \texttt{g1})^{/\!/\texttt{\#DE}}$;
- evaluating the loop as per SemanticsRule.Loop in an environment `env`, with the arguments `TRUE` (which conveys that this is a `while` statement), `limit_opt`, `e`, and `body` yields the (non-error configuration) $C^{/\!/\texttt{\#DE}}$;
- `g2` is the ordered composition of `g1` and `g2` with the `asl_data` edge;
- the output configuration $D$ is the output configuration $C$ with its execution graph substituted with `g2`.

**Formally**

$$\frac{\begin{array}{c} \mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} (\texttt{limit\_opt}, \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{eval\_loop}(\texttt{env}, \texttt{TRUE}, \texttt{limit\_opt}, \texttt{e}, \texttt{body}) \xrightarrow{\text{eval}} C \;/\!/\; \texttt{\#DE} \\ \texttt{g2} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{g2}) \end{array}}{\mathit{eval\_stmt}(\texttt{env}, \overbrace{\text{S\_While}(\texttt{e}, \texttt{e\_limit\_opt}, \texttt{body})}^{\texttt{s}}) \xrightarrow{\text{eval}} D}$$

**SemanticsRule.Loop**

The relation

$$\mathit{eval\_loop}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{B}}^{\texttt{is\_while}}, \overbrace{\mathbb{N}?}^{\texttt{limit\_opt}}, \overbrace{\text{expr}}^{\texttt{e\_cond}}, \overbrace{\text{stmt}}^{\texttt{body}}) \times \left( \begin{array}{ll} \text{Continuing}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) & \cup \\ \overbrace{\text{TReturning}}^{\texttt{\#R}} & \cup \\ \overbrace{\text{TThrowing}}^{\texttt{\#T}} & \cup \\ \overbrace{\text{TDynError}}^{\texttt{\#DE}} & \end{array} \right)$$

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 20.31 does not result in any Assertion Error and the specification terminates with exit code 0.

Listing 20.31: 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 $\texttt{b} \neq \texttt{is\_while}$ 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

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{cond\_m}, \texttt{new\_env}) \;/\!/\; \#T, \#DE \qquad \texttt{cond\_m} \overset{\text{is}}{=} (\text{Bool}(\texttt{b}), \texttt{new\_g}) \qquad \texttt{b} \neq \texttt{is\_while}}{\mathit{eval\_loop}(\texttt{env}, \texttt{is\_while}, \texttt{limit\_opt}, \texttt{e\_cond}, \texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

CONTINUE

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env},\texttt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{cond\_m},\texttt{env1}) \qquad \texttt{cond\_m} \overset{\text{is}}{=} (\texttt{Bool}(\texttt{b}),\texttt{g1}) \\ \texttt{b} = \texttt{is\_while} \qquad \mathit{tick\_loop\_limit}(\texttt{limit\_opt}) \xrightarrow{\text{eval}} \texttt{limit\_opt'} \;/\!/\; \texttt{\#DE} \\ \mathit{eval\_block}(\texttt{env1},\texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g2},\texttt{env2}) \;/\!/\; \texttt{\#R},\texttt{\#T},\texttt{\#DE} \\ \mathit{eval\_loop}(\texttt{env2},\texttt{is\_while},\texttt{limit\_opt'},\texttt{e\_cond},\texttt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\texttt{g3},\texttt{new\_env}) \;/\!/\; \texttt{\#R},\texttt{\#T},\texttt{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_ctrl}} \texttt{g2} \xrightarrow{\texttt{asl\_po}} \texttt{g3}\end{array}}{\mathit{eval\_loop}(\texttt{env},\texttt{is\_while},\texttt{limit\_opt},\texttt{e\_cond},\texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g},\texttt{new\_env})}$$

### SemanticsRule.EvalLimit

The relation

$$\mathit{eval\_limit}(\overbrace{\texttt{env}}^{\mathbb{E}},\overbrace{\texttt{e\_limit\_opt}}^{\text{expr?}}) \longrightarrow (\overbrace{\langle\mathbb{N}\rangle}^{\texttt{v\_opt}},\overbrace{\texttt{g}}^{\mathcal{G}}) \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

evaluates the optional expression `e_limit_opt` in the environment `env`, yielding the optional integer value `v_opt` and execution graph `g`. //#DE

The evaluation uses the function *eval_expr_sef*() because limit expressions are guaranteed side-effect-free by the 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 $\langle$`v`$\rangle$.

**Formally**

NONE

$$\mathit{eval\_limit}(\texttt{env},\ \overbrace{\texttt{None}}^{\texttt{e\_limit\_opt}}\ ) \xrightarrow{\text{eval}} (\overbrace{\texttt{None}}^{\texttt{v\_opt}},\ \overbrace{\emptyset_{\texttt{g}}}^{\texttt{g}}\ )$$

SOME

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e\_limit}) \xrightarrow{\text{eval}} (\texttt{Int}(\texttt{v}), \texttt{g})\ /\!/\ \texttt{\#DE}}{\mathit{eval\_limit}(\texttt{env}, \overbrace{\langle \texttt{e\_limit} \rangle}^{\texttt{e\_limit\_opt}}) \xrightarrow{\text{eval}} (\overbrace{\langle \texttt{v} \rangle}^{\texttt{v\_opt}}, \texttt{g})}$$

**SemanticsRule.TickLoopLimit**

The relation

$$\mathit{tick\_loop\_limit}(\overbrace{\texttt{v\_opt}}^{\mathbb{N}?}) \longrightarrow \overbrace{\langle \texttt{v\_opt'} \rangle}^{\mathbb{N}?} \cup \overbrace{\texttt{TDynError}}^{\texttt{\#DE}}$$

decrements the optional integer `v_opt`, yielding the optional integer value `v_opt`. If the value is 0, the result is a dynamic error.

**Example: Decrementing a Loop Limit Value**

In Listing 20.27, 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 $\langle \texttt{v} - 1 \rangle$.
- All of the following apply (SOME_ERROR):
  - `v_opt` is the integer 0;
  - the result is a dynamic error indicating that a limit has been reached

**Formally**

NONE

$$\mathit{tick\_loop\_limit}(\overbrace{\mathtt{None}}^{\mathtt{v\_opt}}) \xrightarrow{\mathsf{eval}} \overbrace{\mathtt{None}}^{\mathtt{v\_opt'}}$$

SOME_OK

$$\frac{\mathrm{v} > 0}{\mathit{tick\_loop\_limit}(\overbrace{\langle \mathrm{v} \rangle}^{\mathtt{v\_opt}}) \xrightarrow{\mathsf{eval}} \overbrace{\langle \mathrm{v} - 1 \rangle}^{\mathtt{v\_opt'}}}$$

SOME_ERROR

$$\mathit{tick\_loop\_limit}(\overbrace{\langle 0 \rangle}^{\mathtt{v\_opt}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_LE})$$

## 20.12 Repeat Statements

Listing 20.32: 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 ⟶ "repeat" stmt_list "until" expr loop_limit ";"

### 20.12.2 Abstract Syntax

stmt ⟶ S_Repeat( stmt , expr , expr? )
(loop body: stmt; condition: expr; loop limit: expr?)

**ASTRule.SRepeat**

$$\frac{\textit{build\_expr}(\texttt{limit\_expr}) \xrightarrow{\text{ast}} \texttt{limit\_expr\_ast}}{\textit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{l}\texttt{"looplimit"},\texttt{"("},\texttt{limit\_expr} : \texttt{expr},\texttt{")"},\texttt{"repeat"},\\ \quad\hookrightarrow \texttt{stmt\_list},\texttt{"until"},\texttt{expr},\texttt{loop\_limit},\texttt{";"}\end{array}\right)}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Repeat}(\overline{\texttt{stmt\_list}},\overline{\texttt{expr}},\overline{\texttt{looplimit}})}^{\text{ast\_node}}}$$

### 20.12.3 Typing

**TypingRule.SRepeat**

**Example: Typing a Repeat Statement**

The repeat statements in Listing 20.32 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`)$^{/\!/\texttt{\#TE}}$;
- annotating the optional limit expression `limit1` via *annotate_limit_expr* in `tenv` yields (`limit2`, `ses_limit`)$^{/\!/\texttt{\#TE}}$;
- annotating the right-hand-side expression `e1` in `tenv` yields (`t`, `e2`, `ses_e`)$^{/\!/\texttt{\#TE}}$;
- checking that `t` type-satisfies T_Bool in `tenv` yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- `new_s` is a `repeat` statement with statement block `s2`, optional limit expression `limit2`, and condition expression `e2` and , that is, S_Repeat(`s2`, `e2`, `limit2`);
- `new_tenv` is `tenv`;
- define `ses` as the union of `ses_block`, `ses_e`, and `ses_limit`.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\texttt{tenv}, \texttt{s1}) \xrightarrow{\text{type}} (\texttt{s2}, \texttt{ses\_block}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_limit\_expr}(\texttt{tenv}, \texttt{limit1}) \xrightarrow{\text{type}} (\texttt{limit2}, \texttt{ses\_limit}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{e2}, \texttt{ses\_e}) \;/\!/\; \texttt{\#TE} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t}, \texttt{T\_Bool}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_block} \cup \texttt{ses\_e} \cup \texttt{ses\_limit}\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Repeat}(\texttt{s1}, \texttt{e1}, \texttt{limit1})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\texttt{S\_Repeat}(\texttt{s2}, \texttt{e2}, \texttt{limit2})}^{\texttt{new\_s}}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

### 20.12.4 Semantics

**SemanticsRule.SRepeat**

**Example: Evaluation of Repeat Statements**

The specification in Listing 20.32 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((`vs`, `new_g`), `new_env`) or an output configuration $D$.

All of the following apply:

- `s` is a `repeat` statement, S_Repeat(`e`, `body`, `e_limit_opt`);
- evaluating the optional limit expression `e_limit_opt` via *eval_limit* in `env` yields (`limit_opt`, `g1`)$^{/\!/\texttt{\#DE}}$;
- decrementing the optional loop limit value `limit_opt1` yields the updated optional limit value `limit_opt2`$^{/\!/\texttt{\#DE}}$;
- evaluating `body` in `env` as per Chapter 21 yields Continuing(`g2`, `env1`)$^{/\!/\texttt{\#R,\#T,\#DE}}$;

- evaluating the loop as per Section 20.11.4 in an environment `env1`, with the arguments `FALSE` (which conveys that this is a `repeat` statement), `limit_opt2`, `e`, and `body` results in $C$;
- `g3` is the ordered composition of `g1` and `g2` with the `asl_data` and the graph of $C$ with the `asl_po` edge;
- the output configuration $D$ is the output configuration $C$ with its execution graph substituted with `g3`.

**Formally**

$$\frac{\begin{array}{c} \mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} (\texttt{limit\_opt1}, \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{tick\_loop\_limit}(\texttt{limit\_opt1}) \xrightarrow{\text{eval}} \texttt{limit\_opt2} \;/\!/\; \texttt{\#DE} \\ \mathit{eval\_block}(\texttt{env}, \texttt{body}) \xrightarrow{\text{eval}} \mathsf{Continuing}(\texttt{g2}, \texttt{env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_loop}(\texttt{env1}, \texttt{FALSE}, \texttt{limit\_opt2}, \texttt{e}, \texttt{body}) \xrightarrow{\text{eval}} C \\ \texttt{g3} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2} \xrightarrow{\texttt{asl\_po}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{g3}) \end{array}}{\mathit{eval\_stmt}(\texttt{env}, \overbrace{\mathsf{S\_Repeat}(\texttt{e}, \texttt{body}, \texttt{e\_limit\_opt})}^{\texttt{s}}) \xrightarrow{\text{eval}} D}$$

## 20.13 For Statements

Listing 20.33: `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

```
    stmt ⟶ "for" ID "=" expr direction expr loop_limit "do"
           ↪ stmt_list "end" ";"
direction ⟶ "to" | "downto"
```

### 20.13.2 Abstract Syntax

```
for_direction ⟶ Up | Down
```

$$\texttt{stmt} \longrightarrow \texttt{S\_For} \left\{ \begin{array}{rcl} \text{index\_name} & : & \texttt{identifier}, \\ \text{start\_e} & : & \texttt{expr}, \\ \text{dir} & : & \texttt{for\_direction}, \\ \text{end\_e} & : & \texttt{expr}, \\ \text{body} & : & \texttt{stmt}, \\ \text{limit} & : & \texttt{expr?} \end{array} \right\}$$

**ASTRule.SFor**

$$\frac{\textit{build\_expr}(\texttt{start\_e}) \xrightarrow{\text{ast}} \texttt{start\_e\_ast} \qquad \textit{build\_expr}(\texttt{end\_e}) \xrightarrow{\text{ast}} \texttt{end\_e\_ast}}{\textit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{l}\texttt{"for"}, \texttt{ID}(\texttt{index\_name}), \texttt{"="}, \texttt{start\_e} : \texttt{expr}, \\ \hookrightarrow\ \texttt{direction}, \texttt{end\_e} : \texttt{expr}, \texttt{loop\_limit}, \texttt{"do"}, \\ \hookrightarrow\ \texttt{stmt\_list}, \texttt{"end"}, \texttt{";"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_For}\left(\left\{\begin{array}{rcl} \text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e\_ast} \\ \text{dir} & : & \overline{\texttt{direction}} \\ \text{end\_e} & : & \texttt{end\_e\_ast} \\ \text{body} & : & \overline{\texttt{stmt\_list}} \\ \text{limit} & : & \overline{\texttt{looplimit}} \end{array}\right\}\right)}^{\texttt{ast\_node}}}$$

**ASTRule.Direction**

The function

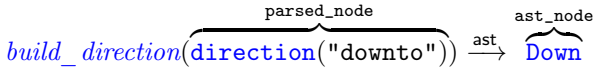
$$\textit{build\_direction}(\overbrace{\mathbb{PARSE}[\texttt{direction}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{for\_direction}}^{\texttt{ast\_node}}$$

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

TO

$$\textit{build\_direction}(\overbrace{\texttt{direction}(\texttt{"to"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{Up}}^{\texttt{ast\_node}}$$

DOWNTO




### 20.13.3 Typing

#### TypingRule.SFor

The `for` loops in Listing 20.33 are well-typed.

#### Example: Ill-typed for loop

Listing 20.34, Listing 20.35, Listing 20.36, and Listing 20.37 show examples of ill-typed `for` loops.

Listing 20.34: 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 20.35: 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 20.36: 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 20.37: Ill-typed for loop 4

```
var g : integer = 0;
func upper_bound() => integer
begin
    g = 1; // writing to a global variables implies impurity.
```

```
    return 5;
end;

func ill_typed_for_loop_3()
begin
    // Illegal: 'upper_bound()' is not a pure 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 $\left\{\begin{array}{rcl} \text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{dir} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \texttt{limit} \end{array}\right\}$;
- annotating the right-hand-side expression `start_e` in `tenv` yields $(\texttt{start\_t}, \texttt{start\_e'}, \texttt{ses\_start})^{/\!/\texttt{\#TE}}$;
- annotating the right-hand-side expression `end_e` in `tenv` yields $(\texttt{end\_t}, \texttt{end\_e'}, \texttt{ses\_end})^{/\!/\texttt{\#TE}}$;
- annotating the optional loop limit expression `limit` via *annotate_limit_expr* in `tenv` yields $(\texttt{limit'}, \texttt{ses\_limit})^{/\!/\texttt{\#TE}}$;
- checking that `ses_start` is pure via *ses_is_pure* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- checking that `ses_start` is deterministic via *ses_is_deterministic* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- checking that `ses_end` is pure via *ses_is_pure* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- checking that `ses_end` is deterministic via *ses_is_deterministic* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- define `ses_cond` as the union of `ses_start`, `ses_end`, and `ses_limit`;
- obtaining the underlying type of `start_t` in `tenv` yields $\texttt{start\_struct}^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `end_t` in `tenv` yields $\texttt{end\_struct}^{/\!/\texttt{\#TE}}$;
- 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 $\texttt{cs}^{/\!/\texttt{\#TE}}$;
- `ty` is the integer type with constraints `cs`;
- checking that `index_name` is not already declared in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

- 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**

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{start\_e}) \xrightarrow{\mathsf{type}} (\mathtt{start\_t}, \mathtt{start\_e'}, \mathtt{ses\_start}) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{end\_e}) \xrightarrow{\mathsf{type}} (\mathtt{end\_t}, \mathtt{end\_e'}, \mathtt{ses\_end}) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_limit\_expr}(\mathsf{tenv}, \mathtt{limit}) \xrightarrow{\mathsf{type}} (\mathtt{limit'}, \mathtt{ses\_limit}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check}(\mathit{ses\_is\_pure}(\mathtt{ses\_start}), \mathsf{TE\_SEV}) \xrightarrow{\mathsf{type}} \;/\!/\; \mathtt{\#TE} \\ \mathit{check}(\mathit{ses\_is\_deterministic}(\mathtt{ses\_start}), \mathsf{TE\_SEV}) \xrightarrow{\mathsf{type}} \;/\!/\; \mathtt{\#TE} \\ \mathit{check}(\mathit{ses\_is\_pure}(\mathtt{ses\_end}), \mathsf{TE\_SEV}) \xrightarrow{\mathsf{type}} \;/\!/\; \mathtt{\#TE} \\ \mathit{check}(\mathit{ses\_is\_deterministic}(\mathtt{ses\_end}), \mathsf{TE\_SEV}) \xrightarrow{\mathsf{type}} \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses\_cond} := \mathtt{ses\_start} \cup \mathtt{ses\_end} \cup \mathtt{ses\_limit} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{start\_t}) \xrightarrow{\mathsf{type}} \mathtt{start\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{end\_t}) \xrightarrow{\mathsf{type}} \mathtt{end\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{for\_constraints}(\mathsf{tenv}, \mathtt{start\_struct}, \mathtt{end\_struct}, \mathtt{start\_e'}, \mathtt{end\_e'}, \mathtt{dir}) \xrightarrow{\mathsf{type}} \\ \mathtt{cs} \;/\!/\; \mathtt{\#TE} \\ \mathtt{ty} := \mathsf{T\_Int}(\mathtt{cs}) \qquad \mathit{check\_var\_not\_in\_env}(\mathsf{tenv}, \mathtt{index\_name}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{add\_local}(\mathsf{tenv}, \mathtt{ty}, \mathtt{index\_name}, \mathsf{LDK\_Let}) \xrightarrow{\mathsf{type}} \mathtt{tenv'} \\ \mathit{annotate\_block}(\mathtt{tenv'}, \mathtt{body}) \xrightarrow{\mathsf{type}} (\mathtt{body'}, \mathtt{ses\_block}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_block} \cup \mathtt{ses\_cond}\end{array}}{\mathit{annotate\_stmt}\left(\mathsf{tenv}, \mathsf{S\_For}\overbrace{\left\{\begin{array}{rcl}\mathsf{index\_name} & : & \mathtt{index\_name} \\ \mathsf{start\_e} & : & \mathtt{start\_e} \\ \mathsf{for\_direction} & : & \mathtt{dir} \\ \mathsf{end\_e} & : & \mathtt{end\_e} \\ \mathsf{body} & : & \mathtt{body} \\ \mathsf{limit} & : & \mathtt{limit}\end{array}\right\}}^{\mathtt{s}}\right) \xrightarrow{\mathsf{type}} \left(\mathsf{S\_For}\overbrace{\left\{\begin{array}{rcl}\mathsf{index\_name} & : & \mathtt{index\_name} \\ \mathsf{start\_e} & : & \mathtt{start\_e'} \\ \mathsf{for\_direction} & : & \mathtt{dir} \\ \mathsf{end\_e} & : & \mathtt{end\_e'} \\ \mathsf{body} & : & \mathtt{body'} \\ \mathsf{limit} & : & \mathtt{limit'}\end{array}\right\}}^{\mathtt{new\_s}}, \overbrace{\mathsf{tenv}}^{\mathtt{new\_tenv}}, \mathtt{ses}\right)}$$

**TypingRule.SForConstraints**

The function

$$\mathit{for\_constraints}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{struct1}}, \overbrace{\mathsf{ty}}^{\mathtt{struct2}}, \overbrace{\mathsf{expr}}^{\mathtt{e1'}}, \overbrace{\mathsf{expr}}^{\mathtt{e2'}}, \overbrace{\mathtt{dir}}^{\mathtt{dir}}) \longrightarrow \overbrace{\mathsf{constraint\_kind} \cup \mathsf{TTypeError}}^{\mathtt{vis}}{}^{\mathtt{\#TE}}$$

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 20.33 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 $\texttt{e1\_n}^{\texttt{//\#TE}}$;
  - symbolically simplifying `e2'` in `tenv` yields $\texttt{e2\_n}^{\texttt{//\#TE}}$;
  - define `ics_up` as the single range constraint with expressions `e1_n` and `e2_n`;
  - define `ics_down` as the single range constraint with expressions `e2_n` and `e1_n`;
  - define `vis` as `ics_up` if `dir` is `Up` and `ics_down` otherwise.

**Formally**

NOT_INTEGERS

$$\frac{\mathit{ast\_label}(\texttt{struct1}) \neq \mathsf{T\_Int} \vee \mathit{ast\_label}(\texttt{struct2}) \neq \mathsf{T\_Int}}{\mathit{for\_constraints}(\texttt{tenv}, \texttt{struct1}, \texttt{struct2}, \texttt{e1'}, \texttt{e2'}, \texttt{dir}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{TE\_UT})}$$

UNCONSTRAINED

$$\frac{\begin{array}{c}\mathit{ast\_label}(\texttt{struct1}) = \mathsf{T\_Int} \wedge \mathit{ast\_label}(\texttt{struct2}) = \mathsf{T\_Int} \\ \texttt{struct1} = \mathsf{unconstrained\_integer} \vee \texttt{struct2} = \mathsf{unconstrained\_integer}\end{array}}{\mathit{for\_constraints}(\texttt{tenv}, \texttt{struct1}, \texttt{struct2}, \texttt{e1'}, \texttt{e2'}, \texttt{dir}) \xrightarrow{\text{type}} \overbrace{\mathsf{Unconstrained}}^{\texttt{vis}}}$$

WELL_CONSTRAINED

$$\frac{\begin{array}{c}\mathit{ast\_label}(\texttt{struct1}) = \mathsf{T\_Int} \wedge \mathit{ast\_label}(\texttt{struct2}) = \mathsf{T\_Int} \\ \texttt{struct1} \neq \mathsf{unconstrained\_integer} \wedge \texttt{struct2} \neq \mathsf{unconstrained\_integer} \\ \mathit{normalize}(\texttt{tenv}, \texttt{e1'}) \xrightarrow{\text{type}} \texttt{e1\_n} \;//\; \texttt{\#TE} \\ \mathit{normalize}(\texttt{tenv}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2\_n} \;//\; \texttt{\#TE} \\ \texttt{ics\_up} := \mathsf{WellConstrained}([\mathsf{Constraint\_Range}(\texttt{e1\_n}, \texttt{e2\_n})]) \\ \texttt{ics\_down} := \mathsf{WellConstrained}([\mathsf{Constraint\_Range}(\texttt{e2\_n}, \texttt{e1\_n})]) \\ \texttt{vis} := \mathsf{choice}(\texttt{dir} = \mathsf{Up}, \texttt{ics\_up}, \texttt{ics\_down})\end{array}}{\mathit{for\_constraints}(\texttt{tenv}, \texttt{struct1}, \texttt{struct2}, \texttt{e1'}, \texttt{e2'}, \texttt{dir}) \xrightarrow{\text{type}} \texttt{vis}}$$

### 20.13.4 Semantics

#### SemanticsRule.SFor

Evaluating a `for` statement involves introducing an index variable to the environment. The type system ensures, via TypingRule.SFor, that the index variable is not already declared in the scope of the subprogram containing the `for` statement.

#### Example: Evaluation of For Statements

The specification in Listing 20.33 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
```

### Prose

All of the following apply:

- `s` is a `for` statement, S_For $\left\{ \begin{array}{rcl} \text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{dir} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \_ \end{array} \right\}$;
- evaluating the side-effect-free expression `start_e` in `env` yields Normal(`start_v`, `g1`)$^{\#\text{DE}}$;
- evaluating the side-effect-free expression `end_e` in `env` yields Normal(`end_v`, `g2`)$^{\#\text{DE}}$;
- evaluating the limit expression `e_limit_opt` in the static environment `env` yields Normal(`limit_opt`, `g3`)$^{\#\text{DE}}$;
- declaring the local identifier `index_name` in `env` with value `start_v` is (`g4`, `env1`);
- evaluating the `for` loop with arguments (`index_name`, `limit_opt`, `start_v`, `dir`, `end_v`, `body`) in `env1`, as per SemanticsRule.EvalFor yields Normal(`g5`, `env2`)$^{\#\text{T},\#\text{DE}}$;
- removing the local `index_name` from `env2` is `env3`;
- `new_g` is formed as follows: the parallel composition of `g1`, `g2`, and `g3`; followed by the ordered composition of the result with `g4` using the `asl_data` edge; followed by the ordered composition of the result with `g5` using the `asl_po` edge.
- `new_env` is `env3`.
- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

### Formally

Recall that the expressions for the `for` loop range are side-effect-free, as guaranteed by TypingRule.SFor, which is why they are evaluated via the rule for evaluating side-effect-

free expressions.

$$\frac{\begin{array}{c} \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{start\_e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{start\_v}, \texttt{g1}) \;\; /\!/ \;\; \texttt{\#DE} \\ \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{end\_e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{end\_v}, \texttt{g2}) \;\; /\!/ \;\; \texttt{\#DE} \\ \mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{limit\_opt}, \texttt{g3}) \;\; /\!/ \;\; \texttt{\#DE} \\ \mathit{declare\_local\_identifier}(\texttt{env}, \texttt{index\_name}, \texttt{start\_v}) \xrightarrow{\text{eval}} (\texttt{g4}, \texttt{env1}) \\ \mathit{eval\_for}(\texttt{env1}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{start\_v}, \texttt{dir}, \texttt{end\_v}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Normal}(\texttt{g5}, \texttt{env2}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{remove\_local}(\texttt{env2}, \texttt{index\_name}) \xrightarrow{\text{eval}} \texttt{env3} \\ \texttt{new\_g} := (\texttt{g1} \parallel \texttt{g2} \parallel \texttt{g3}) \xrightarrow{\texttt{asl\_data}} \texttt{g4} \xrightarrow{\texttt{asl\_po}} \texttt{g5} \qquad \texttt{new\_env} := \texttt{env3} \end{array}}{\mathit{eval\_stmt}(\texttt{env}, \overbrace{\text{S\_For}\left\{\begin{array}{rcl} \text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{dir} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \texttt{e\_limit\_opt} \end{array}\right\}}^{\texttt{s}}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

**SemanticsRule.EvalFor**

The relation

$$\mathit{eval\_for}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{index\_name}}, \overbrace{\langle \mathcal{Z} \rangle}^{\texttt{limit\_opt}}, \overbrace{\mathcal{Z}}^{\texttt{v\_start}}, \overbrace{\{\text{Up}, \text{Down}\}}^{\texttt{dir}}, \overbrace{\mathcal{Z}}^{\texttt{v\_end}}, \overbrace{\text{stmt}}^{\texttt{body}}) \times \left(\begin{array}{ll} \overbrace{\text{TReturning}}^{\texttt{\#R}} & \cup \\ \overbrace{\text{TContinuing}}^{\texttt{\#C}} & \cup \\ \overbrace{\text{TThrowing}}^{\texttt{\#T}} & \cup \\ \overbrace{\text{TDynError}}^{\texttt{\#DE}} & \end{array}\right)$$

evaluates the `for` loop with the index variable `index_name`, optional limit value `limit_opt`, starting from the value `v_start` going in the direction given by `dir` until the value given by `v_end`, executing `body` on each iteration. The evaluation utilizes two helper relations: *eval_for_step* and *eval_for_loop*.

The helper relation

$$\mathit{eval\_for\_step}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{index\_name}}, \overbrace{\langle \mathcal{Z} \rangle}^{\texttt{limit\_opt}}, \overbrace{\mathcal{Z}}^{\texttt{v\_start}}, \overbrace{\{\text{Up}, \text{Down}\}}^{\texttt{dir}}) \times ((\overbrace{\mathcal{Z}}^{\texttt{v\_step}} \times \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

either increments or decrements the index variable, returning the new value of the index variable, the modified environment, and the resulting execution graph.

The helper relation

$$\mathit{eval\_for\_loop}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{index\_name}}, \overbrace{\langle \mathcal{Z} \rangle}^{\texttt{limit\_opt}}, \overbrace{\mathcal{Z}}^{\texttt{v\_start}}, \overbrace{\{\texttt{Up}, \texttt{Down}\}}^{\texttt{dir}}, \overbrace{\mathcal{Z}}^{\texttt{v\_end}}, \overbrace{\mathsf{stmt}}^{\texttt{body}}) \times \begin{pmatrix} \overbrace{\overbrace{\mathsf{TContinuing}}^{\#\mathsf{R}}}^{\mathsf{Continuing}(\texttt{new\_g},\texttt{new\_env})} & \cup \\ \overbrace{\mathsf{TReturning}}^{\#\mathsf{T}} & \cup \\ \overbrace{\mathsf{TThrowing}}^{\#\mathsf{DE}} & \cup \\ \mathsf{TDynError} & \end{pmatrix}$$

executes one iteration of the loop body and then uses `eval_for` to execute the remaining iterations.

### Prose

#### Stepping the Index Variable

All of the following apply:

- `op_for_dir` is either `PLUS` when `dir` is `Up` or `MINUS` when `dir` is `Down`;
- reading `v_start` into the identifier `index_name` gives `g1`;
- applying the binary operator `op_for_dir` to `v_start` and the native integer for 1 is `v_step`;
- the execution graph for writing `v_step` into the identifier `index_name` gives `g2`;
- updating the local component of the dynamic environment of `env` by binding `index_name` to `v_step` gives `new_env`;
- `new_g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

#### Running the Loop Body

All of the following apply:

- evaluating `body` as a block statement (see SemanticsRule.Block) in `env` yields $\mathsf{Continuing}(\texttt{g1}, \texttt{env1})^{/\!/\#\mathsf{R},\#\mathsf{T},\#\mathsf{DE}}$;
- stepping the index `index_name` with `v_start` and the direction `dir` in `env1`, that is, $\mathit{eval\_for\_step}(\texttt{env1}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir})$ yields $((\texttt{v\_step}, \texttt{env2}), \texttt{g2})$;
- evaluating the `for` loop with $(\texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_step}, \texttt{dir}, \texttt{v\_end}, \texttt{body})$ in `env2` results in a continuing configuration $\mathsf{Continuing}(\texttt{g3}, \texttt{new\_env})^{/\!/\#\mathsf{R},\#\mathsf{T},\#\mathsf{DE}}$;
- `new_g` is the ordered composition of `g1`, `g2`, and `g3` with the `asl_po` edge.

### Overall Evaluation

#### Example: Overall Evaluation of For Statements

The specification in Listing 20.33 does not result in any assertion error, and terminates with exit-code 0.

Evaluating (`index_name`, `v_start`, `dir`, `v_end`, `body`) in `env` yields either a continuing configuration Continuing(`new_g`, `new_env`), or a returning configuration (in case the body of the loop results in an early return), or an abnormal configuration.

All of the following apply:

- decrementing the optional loop limit value `limit_opt` yields the updated optional limit value `next_limit_opt`$^{\#DE}$;
- `comp_for_dir` is either `LT` when `dir` is `Up` or `GT` when `dir` is `Down`;
- reading `v_start` into the identifier `index_name` gives `g1`;
- One of the following applies:
  - All of the following apply (RETURN):
    - using `comp_for_dir` to compare `v_end` to `v_start` gives the native Boolean for `TRUE`;
    - `new_g` is `g1`;
    - `new_env` is `env`;
    - the result of the entire evaluation is Continuing(`new_g`, `new_env`).
  - All of the following apply (CONTINUE):
    - using `comp_for_dir` to compare `v_end` to `v_start` gives the native Boolean for `FALSE`;
    - evaluating the loop body via *eval_for_loop* with (`index_name`, `limit_opt`, `v_start`, `dir`, `v_end`, `body`) in `env` is Continuing(`g2`, `new_env`)$^{\#R,\#T,\#DE}$;
    - `new_g` is the ordered composition of `g1` and `g2` with the `asl_ctrl` label.

### Formally

Advancing the loop counter one step towards the end of its range is achieved via the following rule:

$$\frac{\begin{array}{c}\texttt{op\_for\_dir} := \mathsf{choice}(\texttt{dir} = \texttt{Up}, \texttt{PLUS}, \texttt{MINUS}) \\ \textit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \texttt{g1} \\ \textit{binop}(\texttt{op\_for\_dir}, \texttt{v\_start}, \texttt{Int}(1)) \xrightarrow{\mathsf{eval}} \texttt{v\_step} \\ \textit{write\_identifier}(\texttt{index}, \texttt{v\_step}) \xrightarrow{\mathsf{eval}} \texttt{g2} \qquad \texttt{env} \overset{\mathsf{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \\ \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}[\texttt{index\_name} \mapsto \texttt{v\_step}])) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2}\end{array}}{\begin{array}{r}\textit{eval\_for\_step}(\texttt{env}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}) \xrightarrow{\mathsf{eval}} \\ ((\texttt{v\_step}, \texttt{new\_env}), \texttt{new\_g})\end{array}}$$

Running the loop body is achieved via the following rule:

$$\frac{\begin{array}{c}\mathit{eval\_block}(\texttt{env}, \texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g1}, \texttt{env1}) \;/\!/\; \#\text{R}, \#\text{T}, \#\text{DE} \\ \mathit{eval\_for\_step}(\texttt{env1}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}) \xrightarrow{\text{eval}} ((\texttt{v\_step}, \texttt{env2}), \texttt{g2}) \\ \mathit{eval\_for}(\texttt{env2}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_step}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\texttt{g3}, \texttt{new\_env}) \;/\!/\; \#\text{R}, \#\text{T}, \#\text{DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_po}} \texttt{g2} \xrightarrow{\texttt{asl\_po}} \texttt{g3}\end{array}}{\begin{array}{r}\mathit{eval\_for\_loop}(\texttt{env}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})\end{array}}$$

Finally, the rules for evaluating a `for` loop utilize both *eval_for_step* and *eval_for_loop* (the latter in a mutually recursive manner):

RETURN

$$\frac{\begin{array}{c}\mathit{tick\_loop\_limit}(\texttt{limit\_opt}) \xrightarrow{\text{eval}} \texttt{next\_limit\_opt} \;/\!/\; \#\text{DE} \\ \texttt{comp\_for\_dir} := \text{choice}(\texttt{dir} = \text{Up}, \text{LT}, \text{GT}) \\ \mathit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\text{eval}} \texttt{g1} \\ \text{******** common prefix ********} \\ \mathit{binop}(\texttt{comp\_for\_dir}, \texttt{v\_end}, \texttt{v\_start}) \xrightarrow{\text{eval}} \text{Bool}(\text{TRUE})\end{array}}{\begin{array}{r}\mathit{eval\_for}(\texttt{env}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\overbrace{\texttt{g1}}^{\texttt{new\_g}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})\end{array}}$$

CONTINUE

$$\frac{\begin{array}{c}\mathit{tick\_loop\_limit}(\texttt{limit\_opt}) \xrightarrow{\text{eval}} \texttt{next\_limit\_opt} \;/\!/\; \#\text{DE} \\ \texttt{comp\_for\_dir} := \text{choice}(\texttt{dir} = \text{Up}, \text{LT}, \text{GT}) \\ \mathit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\text{eval}} \texttt{g1} \\ \text{******** common prefix ********} \\ \mathit{binop}(\texttt{comp\_for\_dir}, \texttt{v\_end}, \texttt{v\_start}) \xrightarrow{\text{eval}} \text{Int}(\text{FALSE}) \\ \mathit{eval\_for\_loop}(\texttt{env}, \texttt{index\_name}, \texttt{next\_limit\_opt}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\texttt{g2}, \texttt{new\_env}) \;/\!/\; \#\text{R}, \#\text{T}, \#\text{DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_ctrl}} \texttt{g2}\end{array}}{\begin{array}{r}\mathit{eval\_for}(\texttt{env}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})\end{array}}$$

## 20.14 Throw Statements

### 20.14.1 Syntax

```
stmt ⟶ "throw" expr ";"
       | "throw" ";"
```

### 20.14.2 Abstract Syntax

stmt $\longrightarrow$ S_Throw(expr?)

#### ASTRule.SThrow

THROW_SOME

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"throw"}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Throw}(\langle \overline{\texttt{expr}} \rangle)}^{\texttt{ast\_node}}$$

THROW_NONE

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"throw"}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Throw}(\texttt{None})}^{\texttt{ast\_node}}$$

### 20.14.3 Typing

#### TypingRule.SThrow

Listing 20.38 and Listing 20.39 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 singleton set for ThrowException(_)
- All of the following apply (SOME):
  - `s` is a throw statement with expression `e`, that is, S_Throw(⟨e⟩);
  - annotating the right-hand-side expression `e` in tenv yields $(\texttt{t\_e}, \texttt{e'}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
  - checking that `t_e` has the structure of an exception type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - view `t_e` as the named type for `exn_name`;
  - `new_s` is a throw statement with expression `e'` and type `t_e`, that is, S_Throw(⟨(e', t_e)⟩);
  - `new_tenv` is tenv;
  - define `ses` as the union of `ses1` and the singleton set for ThrowException(exn_name).

**Formally**

NONE

$$\mathit{annotate\_stmt}(\mathsf{tenv}, \overbrace{\mathsf{S\_Throw}(\mathtt{None})}^{\mathtt{s}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{S\_Throw}(\mathtt{None})}^{\mathtt{new\_s}}, \overbrace{\mathsf{tenv}}^{\mathtt{new\_tenv}}, \overbrace{\{\mathsf{ThrowException}(\_)\}}^{\mathtt{ses}})$$

SOME

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses1}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{check\_structure}(\mathsf{tenv}, \mathtt{t\_e}, \mathsf{T\_Exception}) \xrightarrow{\text{type}} \mathtt{TRUE} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathtt{t\_e} \overset{\text{is}}{=} \mathsf{T\_Named}(\mathtt{exn\_name}) \qquad \mathtt{ses} := \mathtt{ses1} \cup \{\mathsf{ThrowException}(\mathtt{exn\_name})\}\end{array}}{\mathit{annotate\_stmt}(\mathsf{tenv}, \overbrace{\mathsf{S\_Throw}(\langle \mathtt{e} \rangle)}^{\mathtt{s}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{S\_Throw}(\langle (\mathtt{e'}, \mathtt{t\_e}) \rangle)}^{\mathtt{new\_s}}, \overbrace{\mathsf{tenv}}^{\mathtt{new\_tenv}}, \mathtt{ses})}$$

## 20.14.4 Semantics

### SemanticsRule.SThrow

### Example: Rethrowing an Exception

The specification in Listing 20.38 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 20.38: 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 20.39 terminates successfully. That is, no dynamic error occurs.

Listing 20.39: 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

NONE

$$\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Throw}(\mathsf{None})) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}((\mathsf{None}, \emptyset_g), \texttt{env})$$

SOME

$$\frac{\mathit{eval\_expr}(\texttt{env},\texttt{e}) \xrightarrow{\text{eval}} \mathsf{Normal}((\texttt{v},\texttt{g1}),\texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T},\texttt{\#DE} \qquad \texttt{name} \in \mathbb{I} \text{ is fresh} \qquad \texttt{g2} := \mathsf{WriteEffect}(\texttt{name}) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2} \qquad \texttt{ex} := \langle(\mathsf{value\_read\_from}(\texttt{v},\texttt{name}),\texttt{t})\rangle}{\mathit{eval\_stmt}(\texttt{env},\mathsf{S\_Throw}(\langle(\texttt{e},\texttt{t})\rangle)) \xrightarrow{\text{eval}} \mathsf{Throwing}((\texttt{ex},\texttt{new\_g}),\texttt{new\_env})}$$

# 20.15 Try Statements

## 20.15.1 Syntax

$$\begin{aligned}\texttt{stmt} &\longrightarrow \texttt{"try"}\ \texttt{stmt\_list}\ \texttt{"catch"}\ \mathsf{list1}(\texttt{catcher})\ \texttt{otherwise\_opt} \\ &\hookrightarrow\ \texttt{"end"}\ \texttt{";"} \\ \texttt{otherwise\_opt} &\longrightarrow \texttt{"otherwise"}\ \texttt{"=>"}\ \texttt{stmt\_list} \\ &\mid \epsilon\end{aligned}$$

## 20.15.2 Abstract Syntax

$$\texttt{stmt} \longrightarrow \mathsf{S\_Try}(\texttt{stmt},\texttt{catcher}^*,\ \overbrace{\texttt{stmt}?}^{\text{otherwise}}\ )$$

### ASTRule.STry

$$\frac{\mathit{build\_list}[\texttt{catcher}] \xrightarrow{\text{ast}} \texttt{catcher\_list\_ast}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{r}\texttt{"try"},\texttt{stmt\_list},\texttt{"catch"},\\ \hookrightarrow\ \texttt{catcher\_list} : \mathsf{list1}(\texttt{catcher}),\\ \hookrightarrow\ \texttt{otherwise\_opt},\texttt{"end"},\texttt{";"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Try}(\overline{\texttt{stmt\_list}},\texttt{catcher\_list\_ast},\overline{\texttt{otherwise\_opt}})}^{\texttt{ast\_node}}}$$

### ASTRule.OtherwiseOpt

The function

$$\mathit{build\_otherwise\_opt}(\overbrace{\mathbb{PARSE}[\texttt{otherwise\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle\texttt{stmt}\rangle}^{\texttt{ast\_node}}$$

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

NON_EMPTY

$$\frac{\mathit{build\_stmt\_list}(\texttt{stmts}) \xrightarrow{\text{ast}} \texttt{stmts\_ast}}{\mathit{build\_otherwise\_opt}(\overbrace{\texttt{otherwise\_opt}(\texttt{"otherwise"}, \texttt{"=>"}, \texttt{stmts} : \texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{stmts\_ast}}$$

EMPTY

$$\mathit{build\_otherwise\_opt}(\overbrace{\texttt{otherwise\_opt}(\epsilon)}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{None}$$

### 20.15.3 Typing

**Example: Typing Try Statements**

**TypingRule.STry**

In Listing 22.4, Listing 22.5, Listing 22.6, Listing 22.7, and Listing 20.40, 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 $(\texttt{s''}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- annotating each catcher `catchers`[`i`], for each `i` in *indices*(`catchers`) in `tenv` yields `c_i` and $\texttt{xs}_i{}^{/\!/\texttt{\#TE}}$;
- `catchers'` is the list of annotated catchers `c_i` for each $\texttt{i} \in \mathit{indices}(\texttt{catchers})$;
- define `ses_catchers` as the union of all $\texttt{xs}_\texttt{i}$, for index `i` in the list of indices for `catchers`;
- One of the following applies:
  - All of the following apply (NO_OTHERWISE):
    - there is no `otherwise` statement;
    - `new_s` is a try statement with statement `s''`, list catchers `catchers'` and no `otherwise` statement, that is 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'` $/\!/_{\texttt{\#TE}}$;
    - `new_s` is a try statement with statement `s''`, list catchers `catchers'` and `otherwise` statement `otherwise'`, that is $\texttt{S\_Try}(\texttt{s''}, \texttt{catchers'}, \langle \texttt{otherwise'} \rangle)$;
    - define `ses_otherwise` as `ses_block`;
    - define `ses3` as `ses2`, excluding any exception side effect descriptor.
  - define `ses` as the union of `ses3`, `ses_catchers`, and `ses_otherwise`.

- `new_tenv` is `tenv`.

### Formally

NO_OTHERWISE

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\texttt{tenv}, \texttt{s'}) \xrightarrow{\text{type}} (\texttt{s''}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \texttt{i} \in \mathit{indices}(\texttt{catchers}) : \mathit{annotate\_catcher}(\texttt{tenv}, \texttt{catchers}[\texttt{i}]) \xrightarrow{\text{type}} (\texttt{c}_\texttt{i}, \texttt{xs}_v i) \;/\!/\; \texttt{\#TE} \\ \texttt{catchers'} := [\texttt{i} \in \mathit{indices}(\texttt{catchers}) : \texttt{c}_\texttt{i}] \\ \texttt{ses\_catchers} := \bigcup_{\texttt{i} \in \mathit{indices}(\texttt{catchers})} \texttt{xs}_v i \\ \text{******** common prefix ********} \\ \texttt{new\_s} := \texttt{S\_Try}(\texttt{s''}, \texttt{catchers'}, \texttt{None}) \quad \texttt{ses\_otherwise} := \emptyset \quad \texttt{ses3} := \texttt{ses2} \\ \text{******** common suffix ********} \\ \texttt{ses} := \texttt{ses3} \cup \texttt{ses\_catchers} \cup \texttt{ses\_otherwise}\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Try}(\texttt{s'}, \texttt{catchers}, \texttt{None})}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{new\_s}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

OTHERWISE

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\texttt{tenv}, \texttt{s'}) \xrightarrow{\text{type}} (\texttt{s''}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \texttt{i} \in \mathit{indices}(\texttt{catchers}) : \mathit{annotate\_catcher}(\texttt{tenv}, \texttt{catchers}[\texttt{i}]) \xrightarrow{\text{type}} (\texttt{c}_\texttt{i}, \texttt{xs}_v i) \;/\!/\; \texttt{\#TE} \\ \texttt{catchers'} := [\texttt{i} \in \mathit{indices}(\texttt{catchers}) : \texttt{c}_\texttt{i}] \\ \texttt{ses\_catchers} := \bigcup_{\texttt{i} \in \mathit{indices}(\texttt{catchers})} \texttt{xs}_i \\ \text{******** common prefix ********} \\ \mathit{annotate\_block}(\texttt{tenv}, \texttt{otherwise}) \xrightarrow{\text{type}} (\texttt{otherwise'}, \texttt{ses\_block}) \;/\!/\; \texttt{\#TE} \\ \texttt{new\_s} := \texttt{S\_Try}(\texttt{s''}, \texttt{catchers'}, \texttt{otherwise}') \quad \texttt{ses\_otherwise} := \texttt{ses\_block} \\ \texttt{ses3} := \texttt{ses2} \setminus \{\texttt{s} \in \mathcal{P}(\texttt{TSideEffect}) \mid \texttt{config\_dom}(\texttt{s}) = \texttt{ThrowException}\} \\ \text{******** common suffix ********} \\ \texttt{ses} := \texttt{ses\_catchers} \cup \texttt{ses\_otherwise}\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Try}(\texttt{s'}, \texttt{catchers}, \langle \texttt{otherwise} \rangle)}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{new\_s}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

### 20.15.4 Semantics

**SemanticsRule.STry**

**Example: Evaluation of Try Statements**

Evaluating the specification in Listing 20.40 does not result in any Assertion error, and the specification terminates with the exit code 0.

Listing 20.40: 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 `s1` in `env` as per Chapter 21 yields the configuration $\texttt{s\_m}^{/\!/\texttt{\#DE}}$;
- evaluating (`catchers`, `otherwise_opt`, `s_m`) as per Chapter 22 is $C$, which is the result of the entire evaluation.

**Formally**

$$\frac{\mathit{eval\_block}(\texttt{env}, \texttt{s1}) \xrightarrow{\text{eval}} \texttt{s\_m} \;/\!/\; \texttt{\#DE} \qquad \mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\text{eval}} C}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Try}(\texttt{s1}, \texttt{catchers}, \texttt{otherwise\_opt})) \xrightarrow{\text{eval}} C}$$

## 20.16 Return Statements

### 20.16.1 Syntax

stmt ⟶ "return" option(expr) ";"

### 20.16.2 Abstract Syntax

stmt ⟶ S_Return(expr?)

#### ASTRule.SReturn

$$\frac{\mathit{build\_option}[\texttt{expr}](\texttt{expr}) \xrightarrow{\text{ast}} \texttt{expr\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"return"}, \texttt{expr} : \texttt{option}(\texttt{expr}), \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Return}(\texttt{expr\_ast})}^{\texttt{ast\_node}}}$$

### 20.16.3 Typing

#### TypingRule.SReturn

#### Example: Typing Return Statements

The return statements in Listing 20.42, Listing 20.43, and Listing 20.44 are all well-typed.

The return statement `return 0;` in Listing 20.41 is ill-typed, since `proc` is not a function but a procedure.

Listing 20.41: 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`)$^{/\!/\texttt{\#TE}}$;
  - checking whether `t_e'` type-satisfies `t` in tenv yields TRUE$^{/\!/\texttt{\#TE}}$;
  - `new_s` is a `return` statement with value `e'`, that is, S_Return(⟨`e'`⟩);
  - `new_tenv` is tenv.

**Formally**

$$\text{ERROR}\quad \frac{\texttt{b} := (L^{\texttt{tenv}}.\texttt{return\_type} = \texttt{None} \leftrightarrow \texttt{e\_opt} = \texttt{None}) \qquad \texttt{b} = \texttt{FALSE}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Return}(\texttt{e\_opt})}^{\texttt{s}}) \xrightarrow{\texttt{type}} \texttt{TypeError}(\texttt{TE\_BSPD})}$$

$$\text{NONE}\quad \frac{L^{\texttt{tenv}}.\texttt{return\_type} = \texttt{None}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Return}(\texttt{None})}^{\texttt{s}}) \xrightarrow{\texttt{type}} (\overbrace{\texttt{S\_Return}(\texttt{None})}^{\texttt{new\_s}}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \overbrace{\emptyset}^{\texttt{ses}})}$$

$$\text{SOME}\quad \frac{L^{\texttt{tenv}}.\texttt{return\_type} = \langle \texttt{t} \rangle \qquad \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\texttt{type}} (\texttt{t\_e'}, \texttt{e'}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \qquad \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_e'}, \texttt{t}) \xrightarrow{\texttt{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Return}(\langle \texttt{e} \rangle)}^{\texttt{s}}) \xrightarrow{\texttt{type}} (\overbrace{\texttt{S\_Return}(\langle \texttt{e'} \rangle)}^{\texttt{new\_s}}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

### 20.16.4 Semantics

**SemanticsRule.SReturn**

**Example: No Return Value**

The specification in Listing 20.42 exits the procedure `println_me` by evaluating the `return;` statement.

Listing 20.42: Evaluating a `return` statement with no value

```
func println_me ()
begin
  for i = 0 to 42 do
    if i >= 3 then
      return;
    end;
```

```
  end;
  assert FALSE;

end;

func main () => integer
begin
    println_me ();

    return 0;
end;
```

### Example: Returning a Single Value

In Listing 20.43, `return 3;` exits the function `f` with value `3`.

Listing 20.43: Evaluating a `return` statement with a single value

```
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 20.44, `return (3, 42);` exits the function `f` with the value `(3, 42)`.

Listing 20.44: 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

NONE

$$\mathit{eval\_stmt}(\texttt{env}, \texttt{S\_Return}(\texttt{None})) \xrightarrow{\text{eval}} \texttt{Returning}((\texttt{[ ]}, \emptyset_g), \texttt{env})$$

ONE

$$\frac{\texttt{wid} \in \mathbb{I} \text{ is fresh} \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \texttt{Normal}((\texttt{v}, \texttt{g1}), \texttt{new\_env}) \;//\; \#\texttt{T}, \#\texttt{DE} \qquad \mathit{write\_identifier}(\texttt{wid}, \texttt{v}) \xrightarrow{\text{eval}} \texttt{g2} \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2}}{\mathit{eval\_stmt}(\texttt{env}, \texttt{S\_Return}(\langle \texttt{e} \rangle)) \xrightarrow{\text{eval}} \texttt{Returning}((\texttt{[v]}, \texttt{new\_g}), \texttt{new\_env})}$$

TUPLE

$$\frac{\mathit{eval\_expr\_list\_m}(\texttt{env}, \texttt{es}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{ms}, \texttt{new\_env}) \;/\!/\; \#\text{T}, \#\text{DE} \qquad \mathit{write\_folder}(\texttt{ms}) \xrightarrow{\text{eval}} (\texttt{vs}, \texttt{new\_g})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Return}(\langle \text{E\_Tuple}(\texttt{es}) \rangle)) \xrightarrow{\text{eval}} \text{Returning}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env})}$$

### SemanticsRule.EExprListM

The helper relation

$$\mathit{eval\_expr\_list\_m}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\text{expr}^*}^{\texttt{es}}) \bigtimes \text{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\text{TThrowing}}^{\#\text{T}} \cup \overbrace{\text{TDynError}}^{\#\text{DE}}$$

evaluates a list of expressions `es` in left-to-right in the initial environment `env` and returns the list of values associated with graphs `vms` and the new environment `new_env`. If the evaluation of any expression terminates abnormally then the abnormal configuration is returned.

### Example: Seperately Evaluating a List of Expressions

In Listing 20.44, 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):
  - `es` is an empty list;
  - `vms` is then empty list.
- All of the following apply (NON_EMPTY):
  - `es` is a list with `head` `e` and `tail` `es1`;
  - evaluating `e` in `env` yields $\text{Normal}(\texttt{m1}, \texttt{env1})^{/\!/\#\text{T},\#\text{DE}}$;
  - evaluating `es1` in `env1` via *eval_expr_list_m* yields $\text{Normal}(\texttt{vms1}, \texttt{new\_env})^{/\!/\#\text{T},\#\text{DE}}$;
  - the result is the normal configuration with the list consisting of `m1` as its `head` and `vms1` as its `tail` and `new_env`.

### Formally

EMPTY

$$\mathit{eval\_expr\_list\_m}(\texttt{env}, \overbrace{[\,]}^{\texttt{es}}) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{[\,]}^{\texttt{vms}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

**Semantics**

NON_EMPTY

$$\frac{\texttt{es} = [\texttt{e}] + \texttt{es1} \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_expr\_list\_m}(\texttt{env1}, \texttt{es1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms1}, \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE}}{\mathit{eval\_expr\_list\_m}(\texttt{env}, \texttt{es}) \xrightarrow{\text{eval}} \text{Normal}([\texttt{m1}] + \texttt{vms1}, \texttt{new\_env})}$$

**SemanticsRule.WriteFolder**

The helper relation

$$\mathit{write\_folder}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms}}) \bigtimes (\overbrace{\mathbb{V}^*}^{\texttt{vs}}, \overbrace{\mathcal{G}}^{\texttt{new\_g}}) \ ,$$

concatenates the input values in `vms` and generates an execution graph by composing the graphs in `vms` with Write Effects for the respective values.

**Example: Folding a List of Pairs with Values and Execution Graphs**

In Listing 20.44, the statement `return (3 , 42);` uses *write_folder* to generate $([\texttt{Int}(3), \texttt{Int}(42)], \emptyset_{\mathsf{g}})$. 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**

$$\text{EMPTY}\quad \mathit{write\_folder}(\overbrace{[\,]}^{\texttt{vms}}) \xrightarrow{\text{eval}} (\overbrace{[\,]}^{\texttt{vs}}, \overbrace{\emptyset_{\texttt{g}}}^{\texttt{new\_g}})$$

$$\text{NON\_EMPTY}\quad \frac{\texttt{vms} = [\texttt{m}] + \texttt{vms1} \qquad \texttt{m} := (\texttt{v}, \texttt{g}) \qquad \texttt{wid} \in \mathbb{I} \text{ is fresh}}{\mathit{write\_identifier}(\texttt{wid}, \texttt{v}) \xrightarrow{\text{eval}} \texttt{g1} \qquad \mathit{write\_folder}(\texttt{vms1}, \texttt{g1}) \xrightarrow{\text{eval}} (\texttt{vs1}, \texttt{g2})}$$
$$\frac{\texttt{vs} := [\texttt{v}] + \texttt{vs1} \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_po}} \texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2}}{\mathit{write\_folder}(\texttt{vms}) \xrightarrow{\text{eval}} (\texttt{vs}, \texttt{new\_g})}$$

# 20.17 Print Statements

## 20.17.1 Syntax

stmt ⟶ "print" plist0(expr) ";"
stmt ⟶ "println" plist0(expr) ";"

## 20.17.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_Print}(\overbrace{\text{expr}^*}^{\text{args}}, \overbrace{\mathbb{B}}^{\text{newline}})$$

**ASTRule.SPrint**

$$\frac{\mathit{build\_plist}[\text{expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{args\_ast} \qquad \texttt{newline} := \texttt{FALSE}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"print"}, \texttt{args} : \text{plist0}(\text{expr}), \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_Print}(\texttt{args\_ast}, \texttt{newline})}^{\texttt{ast\_node}}}$$

$$\frac{\mathit{build\_plist}[\text{expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{args\_ast} \qquad \texttt{newline} := \texttt{TRUE} \qquad \texttt{debug} := \texttt{FALSE}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"println"}, \texttt{args} : \text{plist0}(\text{expr}), \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_Print}(\texttt{args\_ast}, \texttt{newline})}^{\texttt{ast\_node}}}$$

## 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 $\texttt{args}_i$ in tenv yields $(\texttt{t}_i, \texttt{args'}_i, \texttt{xs}_i)^{/\!/\texttt{\#TE}}$;
- checking for each index $i$ in the list of indices for `args`, that $\texttt{t}_i$ is a singular type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `new_s` denotes the print statement with arguments `args'` and newline indicator `newline`;
- `new_tenv` is tenv;
- define `ses` as the union of $\texttt{xs}_i$, index $i$ in the list of indices for `args`.

**Formally**

$$\frac{\begin{array}{c}\texttt{i} \in \mathit{indices}(\texttt{args}) : \mathit{annotate\_expr}(\texttt{args}[i], \texttt{tenv}) \xrightarrow{\text{type}} (\texttt{t}_i, \texttt{args'}[i], \texttt{xs}_i) \;/\!/\; \texttt{\#TE} \\ \texttt{i} \in \mathit{indices}(\texttt{args}) : \mathit{check}(\mathit{is\_singular}(\texttt{t}_i), \texttt{TE\_UT}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \bigcup_{\texttt{i} \in \mathit{indices}(\texttt{args})} \texttt{xs}_i\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Print}(\texttt{args}, \texttt{newline})}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{S\_Print}(\texttt{args'}, \texttt{newline}), \texttt{tenv}, \texttt{ses})}$$

## 20.17.4 Semantics

**SemanticsRule.SPrint**

**Example: Printing Literals**

Listing 20.45 shows examples of printing various types of literals, followed by the output to the console resulting from running the specification.

Listing 20.45: 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___12345567890);
  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
5317089828729272873049957 8000
TRUE
FALSE
123456789001234567 89/10000000000
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$((\texttt{v_list}, \texttt{g}), \texttt{new_env})^{/\!/\texttt{\#T,\#DE}}$;
  - 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$(\texttt{e_list}, \texttt{FALSE})$ yields the configuration Continuing$(\texttt{g}, \texttt{env1})^{/\!/\texttt{\#T,\#DE}}$;
  - outputs a newline character to the console, if one exists;
- the resulting configuration is Continuing$(\texttt{g}, \texttt{new_env})$.

**Formally**

PRINT

$$\frac{\mathit{eval\_expr\_list}(\texttt{env}, \texttt{e\_list}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{env}_1) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{i} \in \mathit{indices}(\texttt{v\_list}) : \mathit{output\_to\_console}(\texttt{env}_\texttt{i}, \texttt{v\_list}[i]) \xrightarrow{\text{eval}} \texttt{env}_{\texttt{i}+1} \quad n := |\texttt{v\_list}| \quad \texttt{new\_env} := \texttt{env}_{n+1}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Print}(\texttt{e\_list}, \texttt{FALSE})) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g}, \texttt{new\_env})}$$

We define the newline character $\texttt{newline} \triangleq \text{ASCII}\{\texttt{10}\}$.

PRINTLN

$$\frac{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Print}(\texttt{e\_list}, \texttt{FALSE})) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \mathit{output\_to\_console}(\texttt{env1}, \texttt{String}(\texttt{newline})) \xrightarrow{\text{eval}} \texttt{new\_env}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Print}(\texttt{e\_list}, \texttt{TRUE})) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g}, \texttt{new\_env})}$$

Not all ASL runtimes support printing to a console (see Guide.Printing). Therefore, the semantics is parameterized by the function

$$\mathit{output\_to\_console}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \longrightarrow \overbrace{\mathbb{E}}^{\texttt{new\_env}}$$

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 $\mathit{literal\_to\_string} : \text{literal} \longrightarrow \mathbb{S}$, which defines how a literal is represented by a string, is defined by Table. 20.1. Please note that surrounding quotations mark for $\text{L\_String}(S)$ are not included in $\mathit{literal\_to\_string}(S)$, 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` | ***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 *literal_to_string*(`l`).

**Formally**

$$\frac{\texttt{env} \overset{\text{is}}{=} (\texttt{tenv}, \texttt{denv}, \texttt{consolestream}) \qquad \texttt{new\_env} := (\texttt{tenv}, \texttt{denv}, \texttt{consolestream} + \mathit{literal\_to\_string}(\texttt{l}))}{\mathit{output\_to\_console}(\texttt{env}, \texttt{NV\_Literal}(\texttt{l})) \xrightarrow{\text{eval}} (\texttt{new\_env})}$$

**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 20.45 evaluate their list of expressions with no other effect.

**Prose**

Define `new_env` as `env`.

**Formally**

$$\mathit{output\_to\_console}(\texttt{env}, \_) \xrightarrow{\text{eval}} \overbrace{\texttt{env}}^{\texttt{new\_env}}$$

## 20.18 The Unreachable Statement

Listing 20.46 shows an example of using an unreachable statement to implement a custom form of assertion checking.

Listing 20.46: 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 ⟶ `"Unreachable" "(" ")" ";"`

### 20.18.2 Abstract Syntax

stmt ⟶ S_Unreachable

#### ASTRule.SUnreachable

$$build\_stmt(\overbrace{\texttt{stmt}(\texttt{"Unreachable"},\texttt{"("},\texttt{")"},\texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_Unreachable}}^{\text{ast\_node}}$$

#### TypingRule.SUnreachable

**Example: Typing an Unreachable Statement**

The unreachable statement in Listing 20.46 is well-typed.

**Prose**

Annotating S_Unreachable in the static environment tenv yields (S_Unreachable, tenv, ∅).

**Formally**

$$annotate\_stmt(\text{tenv}, \text{S\_Unreachable}) \xrightarrow{\text{type}} (\text{S\_Unreachable}, \text{tenv}, \overbrace{\emptyset}^{\text{ses}})$$

**SemanticsRule.SUnreachable**

**Example: Evaluating an Unreachable Statement**

Evaluating the specification in Listing 20.46 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**

$$\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Unreachable}) \xrightarrow{\text{eval}} \mathsf{DynError}(\mathtt{DE\_UNR})$$

## 20.19 Pragma Statements

Listing 20.47: A pragma statement

```
func internal_function(x : integer)
begin
    pragma implementation_hidden x + 1;
end;
```

### 20.19.1 Syntax

$$\mathtt{stmt} \longrightarrow \texttt{"pragma"}\ \mathbf{ID}\ \mathsf{clist0}(\mathtt{expr})\ \texttt{";"}$$

### 20.19.2 Abstract Syntax

$$\mathtt{stmt} \longrightarrow \mathsf{S\_Pragma}(\mathbf{ID}, \overbrace{\mathtt{expr}^{*}}^{\text{args}})$$

**ASTRule.SPragma**

$$\frac{\mathit{build\_clist}[\mathtt{expr}](\mathtt{args}) \xrightarrow{\text{ast}} \mathtt{args\_ast}}{\mathit{build\_stmt}(\overbrace{\mathtt{stmt}(\texttt{"pragma"}, \mathbf{ID}(\mathtt{id}), \mathtt{args} : \mathsf{clist0}(\mathtt{expr}), \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pragma}(\mathtt{id}, \mathtt{args\_ast})}^{\text{ast\_node}}}$$

**TypingRule.SPragma**

**Example: Typing a Pragma Statement**

The pragma statement in Listing 20.47 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 $(\_, \_, \texttt{xs}_i)^{\#\texttt{TE}}$;
- define `ses` as the union of $\texttt{xs}_i$, 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 `sess`.

### Formally

$$\frac{i \in \mathit{indices}(\texttt{args}) : \mathit{annotate\_expr}(\text{tenv}, \texttt{args}[i]) \xrightarrow{\text{type}} (\_, \_, \texttt{xs}_i) \;/\!/\; \#\texttt{TE} \qquad \texttt{ses} := \bigcup_{i \in \mathit{indices}(\texttt{args})} \texttt{xs}_i}{\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_Pragma}(\texttt{id}, \texttt{args})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Pass}}^{\texttt{new\_s}}, \text{tenv})}$$

## 20.19.3 Semantics

### Prose

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

$$\mathit{annotate\_block}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{stmt}}^{\texttt{s}}) \longrightarrow (\overbrace{\mathsf{stmt}}^{\texttt{new\_stmt}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a block statement `s` in static environment `tenv` and returns the annotated statement `new_stmt` and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

**TypingRule.Block**

See Example: Block Statements.

**Prose**

All of the following apply:

- annotating the statement `s` in `tenv` yields $(\texttt{new\_stmt}, \texttt{new\_tenv}, \texttt{ses})^{/\!/ \texttt{\#TE}}$;
- the modified environment `new_tenv` is dropped.

**Formally**

$$\frac{\mathit{annotate\_stmt}(\texttt{tenv}, \texttt{s}) \xrightarrow{\text{type}} (\texttt{new\_stmt}, \_, \texttt{ses}) \;\; /\!/ \;\; \texttt{\#TE}}{\mathit{annotate\_block}(\texttt{tenv}, \texttt{s}) \xrightarrow{\text{type}} (\texttt{new\_stmt}, \texttt{ses})}$$

### 21.1.1 Comments

A local identifier declared in a block statement (with `var`, `let`, or `constant`) is in scope from the point immediately after its declaration until the end of the immediately enclosing block. This means, we can discard the environment at the end of an enclosing block, which has the effect of dropping bindings of the identifiers declared inside the block.

## 21.2 Semantics

The relation

$$\mathit{eval\_block}(\overbrace{\mathbb{E}}^{\texttt{env}} \times \overbrace{\mathsf{stmt}}^{\texttt{stm}}) \bigtimes \overbrace{\mathsf{TContinuing}}^{\mathsf{Continuing}(\texttt{new\_g},\texttt{new\_env})} \cup \overbrace{\mathsf{TReturning}}^{\texttt{\#R}} \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

evaluates a statement `stm` as a *block*. That is, `stm` is evaluated in a fresh local environment, which drops back to the original local environment of `env` when the evaluation terminates.

**SemanticsRule.Block**

See Example: Block Statements.

We first define the helper function *pop_local_scope*:

$$\mathit{pop\_local\_scope} : \overbrace{\mathbb{DE}}^{\texttt{outer\_env}} \times \overbrace{\mathbb{DE}}^{\texttt{inner\_env}} \to \mathbb{DE}$$
$$\mathit{pop\_local\_scope}(\texttt{outer\_env}, \texttt{inner\_env}) \triangleq (G^{\texttt{inner\_env}}, L^{\texttt{inner\_env}}|_{\mathrm{dom}(L^{\texttt{outer\_env}})})$$

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

$$\frac{\begin{array}{c}\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{eval\_stmt}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \texttt{res} \\ \text{******** common prefix ********} \\ \texttt{res} = \text{Returning}((\texttt{vs}, \texttt{new\_g}), \texttt{env\_ret}) \\ \texttt{env\_ret} \overset{\text{is}}{=} (\texttt{tenv1}, \text{denv1}) \qquad \texttt{new\_env} := (\text{tenv}, \mathit{pop\_local\_scope}(\text{denv}, \text{denv1}))\end{array}}{\mathit{eval\_block}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \text{Returning}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env})}$$

CONTINUING

$$\frac{\begin{array}{c}\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{eval\_stmt}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \texttt{res} \\ \text{******** common prefix ********} \\ \texttt{res} = \text{Continuing}(\texttt{new\_g}, \texttt{env\_cont}) \\ \texttt{env\_cont} \overset{\text{is}}{=} (\texttt{tenv1}, \text{denv1}) \qquad \texttt{new\_env} := (\text{tenv}, \mathit{pop\_local\_scope}(\text{denv}, \text{denv1}))\end{array}}{\mathit{eval\_block}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

THROWING

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{eval\_stmt}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \texttt{res} \qquad \texttt{res} = \text{Throwing}((\texttt{v}, \texttt{new\_g}), \texttt{env\_throw}) \qquad \texttt{env\_throw} \overset{\text{is}}{=} (\texttt{tenv1}, \text{denv1}) \qquad \texttt{new\_env} := (\text{tenv}, \mathit{pop\_local\_scope}(\text{denv}, \text{denv1}))}{\mathit{eval\_block}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \text{Throwing}((\texttt{v}, \texttt{new\_g}), \texttt{new\_env})}$$

********* common prefix *********

That is, evaluating a block discards the bindings for variables declared inside `stm`.

# Chapter 22

# Catching Exceptions

Exception catchers are grammatically derived from `catcher` and represented in the untyped AST by catcher.

## 22.1 Syntax

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

## 22.2 Abstract Syntax

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

### ASTRule.Catcher

The function

$$\mathit{build\_catcher}(\overbrace{\mathbb{PARSE}[\texttt{catcher}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\text{catcher}}^{\texttt{ast\_node}}$$

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

NAMED

$$\mathit{build\_catcher}(\overbrace{\texttt{catcher}(\texttt{"when"}, \textbf{ID}(\texttt{id}), \texttt{":"}, \texttt{ty}, \texttt{"=>"}, \texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\langle \texttt{id} \rangle, \overline{\texttt{ty}}, \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}$$

UNNAMED

$$\mathit{build\_catcher}(\overbrace{\texttt{catcher}(\texttt{"when"}, \texttt{ty}, \texttt{"=>"}, \texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{None}, \overline{\texttt{ty}}, \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}$$

## 22.3 Typing

The function

$$\mathit{annotate\_catcher}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_in}}, \overbrace{\mathsf{catcher}}^{\texttt{c}}) \longrightarrow (\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_filtered}} \times (\overbrace{\mathsf{catcher}}^{\texttt{new\_catcher}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

annotates a catcher `c` in the static environment `tenv` and set of side effect descriptors `ses_in`. The result is the set of side effect descriptors `ses_filtered`, the annotated catcher `new_catcher` and the set of side effect descriptors `ses`. Otherwise, the result is a type error.

### TypingRule.Catcher

### Example: Annotating Catch Clauses

Listing 22.1 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 22.1: 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 ( $\overbrace{\texttt{None}}^{\texttt{name\_opt}}$ , `ty`, `stmt`);
  - annotating the type `ty` in `tenv` yields (`ty'`, `ses_ty`)$^{/\!/\texttt{\#TE}}$;
  - determining whether `ty'` has the structure of an exception type yields TRUE$^{/\!/\texttt{\#TE}}$;
  - annotating the block `stmt` in `tenv` yields `new_stmt`;
  - define `new_catcher` as ( $\overbrace{\texttt{None}}^{\texttt{name\_opt}}$ , `ty'`, `new_stmt`);
- All of the following apply (SOME):
  - the catcher has a named identifier, that is, `c` is (⟨`name`⟩, `ty`, `stmt`);
  - annotating the type `ty` in `tenv` yields (`ty'`, `ses_ty`)$^{/\!/\texttt{\#TE}}$;
  - determining whether `ty'` has the structure of an exception type yields TRUE$^{/\!/\texttt{\#TE}}$;
  - the identifier `name` is not bound in `tenv`;
  - binding `name` in the local environment of `tenv` with the type `ty'` as an immutable variable (that is, with the local declaration keyword LDK_Let), yields the static environment `tenv'`;
  - annotating the block `stmt` in `tenv'` yields `new_stmt`;
  - define `new_catcher` as ($\overbrace{\langle\texttt{name}\rangle}^{\texttt{name\_opt}}$, `ty'`, `new_stmt`);
- define `ses_filtered` as `ses_in` where every exception side effect descriptor with an exception `name` such that *is_subtype*(`tenv`, T_Named(`name`), `ty'`) holds removed;
- define `ses` as the union of `ses_block` and `ses_ty`.

**Formally**

NONE

$$\frac{\begin{array}{c} \mathtt{c} = (\overbrace{\mathsf{None}}^{\mathtt{name\_opt}}, \mathtt{ty}, \mathtt{stmt}) \qquad \mathit{annotate\_type}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} (\mathtt{ty'}, \mathtt{ses\_ty}) \;/\!/\; \#\mathsf{TE} \\ \mathit{check\_structure}(\mathtt{tenv}, \mathtt{ty'}, \mathsf{T\_Exception}) \xrightarrow{\mathsf{type}} \mathsf{TRUE} \;/\!/\; \#\mathsf{TE} \\ \mathit{annotate\_block}(\mathtt{tenv}, \mathtt{stmt}) \xrightarrow{\mathsf{type}} (\mathtt{new\_stmt}, \mathtt{ses\_block}) \;/\!/\; \#\mathsf{TE} \\ \mathtt{new\_catcher} := (\overbrace{\mathsf{None}}^{\mathtt{name\_opt}}, \mathtt{ty'}, \mathtt{new\_stmt}) \\ \text{******** common suffix ********} \\ \mathtt{ses\_filtered} := \mathtt{ses\_in} \setminus \{\mathsf{ThrowException}(\mathtt{name}) \mid \mathit{is\_subtype}(\mathtt{tenv}, \mathsf{T\_Named}(\mathtt{name}), \mathtt{ty'})\} \\ \mathtt{ses} := \mathtt{ses\_block} \cup \mathtt{ses\_ty} \end{array}}{\mathit{annotate\_catcher}(\mathtt{tenv}, \mathtt{ses\_in}, \mathtt{c}) \xrightarrow{\mathsf{type}} (\mathtt{ses\_filtered}, \mathtt{new\_catcher}, \mathtt{ses})}$$

SOME

$$\frac{\begin{array}{c} \mathtt{c} = (\overbrace{\langle \mathtt{name} \rangle}^{\mathtt{name\_opt}}, \mathtt{ty}, \mathtt{stmt}) \qquad \mathit{annotate\_type}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} (\mathtt{ty'}, \mathtt{ses\_ty}) \;/\!/\; \#\mathsf{TE} \\ \mathit{check\_structure}(\mathtt{tenv}, \mathtt{ty'}, \mathsf{T\_Exception}) \xrightarrow{\mathsf{type}} \mathsf{TRUE} \;/\!/\; \#\mathsf{TE} \\ \mathit{check\_var\_not\_in\_env}(\mathtt{tenv}, \mathtt{name}) \xrightarrow{\mathsf{type}} \mathsf{TRUE} \;/\!/\; \#\mathsf{TE} \\ \mathit{add\_local}(\mathtt{tenv}, \mathtt{name}, \mathtt{ty'}, \mathsf{LDK\_Let}) \xrightarrow{\mathsf{type}} \mathtt{tenv'} \\ \mathit{annotate\_block}(\mathtt{tenv'}, \mathtt{stmt}) \xrightarrow{\mathsf{type}} (\mathtt{new\_stmt}, \mathtt{ses\_block}) \;/\!/\; \#\mathsf{TE} \\ \mathtt{new\_catcher} := (\overbrace{\langle \mathtt{name} \rangle}^{\mathtt{name\_opt}}, \mathtt{ty'}, \mathtt{new\_stmt}) \\ \text{******** common suffix ********} \\ \mathtt{ses\_filtered} := \mathtt{ses\_in} \setminus \{\mathsf{ThrowException}(\mathtt{name}) \mid \mathit{is\_subtype}(\mathtt{tenv}, \mathsf{T\_Named}(\mathtt{name}), \mathtt{ty'})\} \\ \mathtt{ses} := \mathtt{ses\_block} \cup \mathtt{ses\_ty} \end{array}}{\mathit{annotate\_catcher}(\mathtt{tenv}, \mathtt{ses\_in}, \mathtt{c}) \xrightarrow{\mathsf{type}} (\mathtt{ses\_filtered}, \mathtt{new\_catcher}, \mathtt{ses})}$$

## 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:

$$\mathsf{TOutConfig} \triangleq \mathsf{TNormal} \cup \mathsf{TThrowing} \cup \mathsf{TContinuing} \cup \mathsf{TReturning} \ .$$

The relation

$$\mathit{eval\_catchers}(\overbrace{\mathbb{E}}^{\mathtt{env}}, \overbrace{\mathsf{catcher}^*}^{\mathtt{catchers}}, \overbrace{\langle \mathsf{stmt} \rangle}^{\mathtt{otherwise\_opt}}, \overbrace{\mathsf{TOutConfig}}^{\mathtt{s\_m}}) \times \begin{pmatrix} \mathsf{TReturning} & \cup \\ \mathsf{TContinuing} & \cup \\ \mathsf{TThrowing} & \cup \\ \mathsf{TDynError} & \end{pmatrix}$$

evaluates a list of `catch` clauses `catchers`, an 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 22.2, 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 22.2: 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 22.3 terminates successfully. That is, no dynamic error occurs.

Listing 22.3: 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((⟨value_read_from(v, e_id), v_ty⟩, s_g), env_throw);
- `env` consists of the static environment tenv and dynamic environment denv;
- `env_throw` consists of the static environment tenv and dynamic environment `denv_throw`;
- finding the first catcher with the static environment tenv, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that does not declare a name (None) and gives a statement `s`;
- evaluating `s` in `env_throw` as a block (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

$$\frac{\begin{array}{c}\texttt{s\_m} \overset{\text{is}}{=} \text{Throwing}((\langle \text{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \\ \texttt{env} \overset{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \qquad \texttt{env\_throw} \overset{\text{is}}{=} (\text{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \\ \textit{find\_catcher}(\text{tenv}, \texttt{v\_ty}, \texttt{catchers}) \overset{\text{is}}{=} \langle(\text{None}, \_, \texttt{s})\rangle \\ \textit{eval\_block}(\texttt{env\_throw}, \texttt{s}) \xrightarrow{\text{eval}} C \;/\!/\; \texttt{\#DE} \\ D := \textit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, C) \qquad \texttt{new\_g} := \texttt{s\_g} \xrightarrow{\texttt{asl\_po}} \textit{graph}(D)\end{array}}{\textit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\text{eval}} D(\textit{graph} \mapsto \texttt{new\_g})}$$

### SemanticsRule.CatchNamed

### Example: Catching a Named Exception

The specification in Listing 22.4, prints `My exception with my message`.

Listing 22.4: 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((⟨value_read_from(`v`, `e_id`), `v_ty`⟩, `s_g`), `env_throw`);
- `env` consists of the static environment `tenv` and dynamic environment `denv`;
- `env_throw` consists of the static environment `tenv` and dynamic environment `denv_throw`;
- finding the first catcher with the static environment `tenv`, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives a statement `s`;
- `g1` is the execution graph resulting from reading `v` into the identifier `e_id`;
- declaring a local identifier `name` with (`e1`, `g1`) in `env_throw` gives (`env2`, `g2`);
- evaluating `s` in `env2` as a block (SemanticsRule.Block) is not an error configuration $C^{/\!/\texttt{\#DE}}$;
- `env3` is the environment of the configuration $C$;
- removing the binding for `name` from the local component of the dynamic environment in `env3` gives `env4`;
- substituting the environment of $C$ with `env4` gives $D$;
- editing potential implicit throwing configurations via *rethrow_implicit*(`v`, `v_ty`, $D$) gives the configuration $E$;

- `new_g` is the ordered composition of `s_g`, `g1`, `g2`, and the graph of $E$, with the `asl_po` edges;
- the result of the entire evaluation is $E$ with its graph substituted with `new_g`.

**Formally**

$$\frac{\begin{array}{c}\texttt{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \\ \texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{env\_throw} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \\ \mathit{find\_catcher}(\mathsf{tenv}, \texttt{v\_ty}, \texttt{catchers}) \overset{\text{is}}{=} \langle(\langle\texttt{name}\rangle, \_, \texttt{s})\rangle \qquad \texttt{g1} := \mathit{read\_identifier}(\texttt{e\_id}, \texttt{v}) \\ \mathit{declare\_local\_identifier\_m}(\texttt{env\_throw}, \texttt{name}, (\texttt{e1}, \texttt{g1})) \xrightarrow{\text{eval}} (\texttt{env2}, \texttt{g2}) \\ \mathit{eval\_block}(\texttt{env2}, \texttt{s}) \xrightarrow{\text{eval}} C \ \ /\!/ \ \texttt{\#DE} \\ \mathsf{env3} := \mathit{environ}(C) \qquad \mathit{remove\_local}(\mathsf{env3}, \texttt{name}) \xrightarrow{\text{eval}} \mathsf{env4} \\ D := C(\mathit{environ} \mapsto \mathsf{env4}) \qquad E := \mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, D) \\ \texttt{new\_g} := \texttt{s\_g} \xrightarrow{\texttt{asl\_po}} \texttt{g1} \xrightarrow{\texttt{asl\_po}} \texttt{g2} \xrightarrow{\texttt{asl\_po}} \mathit{graph}(E)\end{array}}{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\text{eval}} E(\mathit{graph} \mapsto \texttt{new\_g})}$$

**SemanticsRule.CatchOtherwise**

**Example: Evaluation of a Catch with an Otherwise**

The specification in Listing 22.5 prints `Otherwise`.

Listing 22.5: 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 $\mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw})$;
- `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 $C^{/\!/\texttt{\#DE}}$;
- editing potential implicit throwing configurations via $\mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, C)$ gives the configuration $D$;
- `new_g` is the ordered composition of `s_g` and the graph of $D$, with the `asl_po` edge;
- the result of the entire evaluation is $D$ with its graph substituted with `new_g`.

**Formally**

$$\frac{\begin{array}{c}\texttt{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \\ \texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{env\_throw} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \\ \mathit{find\_catcher}(\mathsf{tenv}, \texttt{v\_ty}, \texttt{catchers}) = \texttt{None} \qquad \mathit{eval\_block}(\texttt{env\_throw}, \texttt{s}) \xrightarrow{\text{eval}} C \;/\!/\; \texttt{\#DE} \\ D := \mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, C) \qquad \texttt{g} := \texttt{s\_g} \xrightarrow{\texttt{asl\_po}} \mathit{graph}(D)\end{array}}{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \langle \texttt{s}\rangle, \texttt{s\_m}) \xrightarrow{\text{eval}} D(\mathit{graph} \mapsto \texttt{g})}$$

**SemanticsRule.CatchNone**

### Example: Evaluation of an Uncaught Exception

The specification in Listing 22.6 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 22.6: 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;
```

```
  return 0;
end;
```

### Prose

All of the following apply:

- `s_m` is Throwing((⟨value_read_from(v, e_id), v_ty⟩, s_g), env_throw);
- `env` consists of the static environment tenv and dynamic environment denv;
- `env_throw` consists of the static environment tenv and dynamic environment `denv_throw`;
- finding the first catcher with the static environment tenv, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives `None` (that is, neither of the `catch` clauses matches the raised exception);
- since there no `otherwise` clause, the result is `s_m`.

### Formally

$$\frac{\texttt{s\_m} \overset{\text{is}}{=} \text{Throwing}((\langle \text{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \quad \texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \mathit{find\_catcher}(\text{tenv}, \texttt{v\_ty}, \texttt{catchers}) = \text{None}}{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \text{None}, \texttt{s\_m}) \xrightarrow{\text{eval}} \texttt{s\_m}}$$

### SemanticsRule.CatchNoThrow

### Example: Evaluation of a Try Statement that Does Not Raise an Exception

The specification in Listing 22.7 prints `No exception raised`.

Listing 22.7: 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

$$\frac{\texttt{s\_m} = \mathsf{Throwing}((\mathsf{None}, \texttt{s\_g}), \texttt{env\_throw}) \lor \mathsf{config\_dom}(\texttt{s\_m}) = \mathsf{Normal}}{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \_, \texttt{s\_m}) \xrightarrow{\mathsf{eval}} \texttt{s\_m}}$$

### SemanticsRule.FindCatcher

The (recursively-defined) helper relation

$$\mathit{find\_catcher}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathsf{v\_ty}}, \overbrace{\mathsf{catcher}^*}^{\mathsf{catchers}}) \bigtimes \langle \mathsf{catcher} \rangle \ ,$$

returns the first catcher clause in `catchers` that matches the type `v_ty` (as a singleton set), or an empty set (None), by invoking *type_satisfies* with the static environment tenv.

### Example: Finding a Catch Clause

In Listing 22.1, 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 22.1, no catch clause is matched for the type of the exception value thrown by `update_and_throw`, resulting in a dynamic error.

Listing 22.8: 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 (`name_opt`, `e_ty`, `s`);
  - `v_ty` does not subtype `e_ty` in the static environment `tenv`;
  - the result of finding a catcher for `v_ty` with the type environment `tenv` in the tail list `catchers1` is $d$;
  - the result is $d$.

### Formally

EMPTY

$$\mathit{find\_catcher}(\mathsf{tenv}, \mathtt{v\_ty}, \overbrace{[\,]}^{\mathtt{catchers}}) \xrightarrow{\mathsf{eval}} \mathtt{None}$$

MATCH

$$\frac{\mathtt{catchers} = [\mathtt{c}] + \mathtt{catchers1} \qquad \mathtt{c} \overset{\mathsf{is}}{=} (\mathtt{name\_opt}, \mathtt{e\_ty}, \mathtt{s}) \qquad \mathsf{subtypes}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{e\_ty})}{\mathit{find\_catcher}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{catchers}) \xrightarrow{\mathsf{eval}} \langle \mathtt{c} \rangle}$$

NO_MATCH

$$\frac{\mathtt{catchers} = [\mathtt{c}] + \mathtt{catchers1} \qquad \mathtt{c} \overset{\mathsf{is}}{=} (\mathtt{name\_opt}, \mathtt{e\_ty}, \mathtt{s}) \qquad \neg\mathsf{subtypes}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{e\_ty}) \qquad d := \mathit{find\_catcher}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{catchers1})}{\mathit{find\_catcher}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{catchers}) \xrightarrow{\mathsf{eval}} d}$$

### Comments

When the `catch` of a `try` statement is executed, then the thrown exception is caught by the first catcher in that `catch` which it type-satisfies or the `otherwise_opt` in that catch if it exists.

### SemanticsRule.RethrowImplicit

An expressionless `throw` statement causes the exception which the currently executing catcher caught to be thrown.

The helper relation

$$\mathit{rethrow\_implicit}(\overbrace{\mathsf{value\_read\_from}(\mathbb{V},\mathbb{I})}^{\mathtt{v}},\overbrace{\mathsf{ty}}^{\mathtt{v\_ty}},\overbrace{\mathsf{TOutConfig}}^{\mathtt{res}}) \bigtimes \mathsf{TOutConfig}$$

changes *implicit throwing configurations* into *explicit throwing configurations*. That is, configurations of the form $\mathsf{Throwing}((\mathtt{None},\mathtt{g}),\mathtt{env\_throw1})$.

*rethrow_implicit* leaves non-throwing configurations, and *explicit throwing configurations*, which have the form $\mathsf{Throwing}(\langle(\mathsf{value\_read\_from}(\mathtt{v}',\mathtt{e\_id}),\mathtt{v\_ty}')\rangle,\mathtt{g})$, 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 22.9 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 22.9: 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;
  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

$$\mathit{rethrow\_implicit}(\mathtt{v}, \mathtt{v\_ty}, \mathsf{Throwing}((\mathsf{None}, \mathtt{g}), \mathtt{env\_throw1})) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}((\langle(\mathsf{value\_read\_from}(\mathtt{v}, \mathtt{e\_id}), \mathtt{v\_ty})\rangle, \mathtt{g}), \mathtt{env\_throw1})$$

EXPLICIT_THROWING

$$\mathit{rethrow\_implicit}(\mathtt{v}, \mathtt{v\_ty}, \mathsf{Throwing}((\langle(\mathtt{v}', \mathtt{v\_ty}')\rangle, \mathtt{g}), \mathtt{env\_throw1})) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}((\langle(\mathtt{v}', \mathtt{v\_ty}')\rangle, \mathtt{g}), \mathtt{env\_throw1})$$

NON_THROWING

$$\frac{\mathtt{config\_dom}(C) \neq \mathsf{Throwing}}{\mathit{rethrow\_implicit}(\_, \_, C, \_) \xrightarrow{\mathsf{eval}} C}$$

# Chapter 23

# Subprogram Calls

This chapter is concerned with subprogram calls, which appear both in expressions and in statements. Specifically:

- how subprogram calls are represented in syntax (Section 23.1);
- how subprogram calls are represented in abstract syntax (Section 23.2);
- how subprogram calls are typed (Section 23.3), including a discussion on parameter elision (Section 23.3.1); and
- what is the dynamic semantics of subprogram calls (Section 23.4).

## 23.1 Syntax

expr ⟶ **ID** plist0(expr)
stmt ⟶ **ID** plist0(expr) ";"

## 23.2 Abstract Syntax

expr ⟶ E_Call(call)
stmt ⟶ S_Call(call)

## 23.3 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

$$\mathit{annotate\_call}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{call}}^{\texttt{call}}) \longrightarrow (\overbrace{\mathsf{call}}^{\texttt{call'}} \times \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ret\_ty\_opt}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})$$

annotates the call `call` to a subprogram with call type `call_type`, resulting in the following:

- `call'` — the updated call, with all arguments/parameters annotated and `call`.name updated to uniquely identify the call among the set of overloading subprograms declared with the same name;
- `ret_ty_opt` — the optional annotated return type;
- `ses` — the set of side effect descriptors inferred for `call`.

Otherwise, the result is a type error.

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`$^{/\!/\texttt{\#TE}}$;
- applying *annotate_exprs* to annotate the expression list `call`.params in `tenv` yields `params`$^{/\!/\texttt{\#TE}}$;
- applying *annotate_call_actuals_typed* to `call`.name, `params`, `args`, and `call`.call_type in `tenv` yields $(\texttt{call'}, \texttt{ret\_ty}, \texttt{ses})^{/\!/\texttt{\#TE}}$.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_exprs}(\mathsf{tenv}, \texttt{call}.\mathsf{args}) \xrightarrow{\mathsf{type}} \texttt{args} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_exprs}(\mathsf{tenv}, \texttt{call}.\mathsf{params}) \xrightarrow{\mathsf{type}} \texttt{params} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_call\_actuals\_typed}(\mathsf{tenv}, \texttt{call}.\mathsf{name}, \texttt{params}, \texttt{args}, \texttt{call}.\mathsf{call\_type}) \xrightarrow{\mathsf{type}} \\ (\texttt{call'}, \texttt{ret\_ty}, \texttt{ses}) \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_call}(\mathsf{tenv}, \texttt{call}) \xrightarrow{\mathsf{type}} (\texttt{call'}, \texttt{ret\_ty})}$$

**TypingRule.AnnotateCallActualsTyped**

The function

$$\mathit{annotate\_call\_actuals\_typed}\left(\begin{array}{l}\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \\ \overbrace{\mathsf{identifier}}^{\texttt{name}}, \\ \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{params}}, \\ \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{typed\_args}}, \\ \overbrace{\mathsf{sub\_program\_type}}^{\texttt{call\_type}}\end{array}\right) \longrightarrow \begin{array}{l}(\overbrace{\mathsf{call}}^{\texttt{call}}, \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ret\_ty\_opt}}) \\ \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}\end{array}$$

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 23.1, the call expression `xor_extend{64}(bv1, bv2)` is ill-typed, since `xor_extend` has two parameters and only one was supplied.

Listing 23.1: 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 23.2, the call expression `xor_extend{64, 32}(bv1)` is ill-typed, since `xor_extend` has two arguments and only one was supplied.

Listing 23.2: 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 23.3, 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 23.3: 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;
```

**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 $(\texttt{name'}, \texttt{func\_sig}, \texttt{ses\_call})^{/\!/\texttt{\#TE}}$;
- 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 $\texttt{TRUE}^{/\!/\texttt{TE\_BC}}$;
- 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 $\texttt{TRUE}^{/\!/\texttt{TE\_BC}}$;
- checking that the lengths of `func_sig`.args and `args` are the same yields $\texttt{TRUE}^{/\!/\texttt{TE\_BC}}$;
- applying *check_params_typesat* to `params1` to check that the actual parameters have correct types with respect to `func_sig`.parameters in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- define `eqs` as the association of declared parameter names in `func_sig`.parameters with actual parameters `params1`;
- applying *check_args_typesat* to `arg_types` and `eqs` to check that the actual arguments have correct types with respect to `func_sig`.args in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- applying *annotate_ret_ty* to `eqs`, `call_type`, and `func_sig`.return_type to check that the two call types match and to substitute actual parameter arguments in the formal return type yields $\texttt{ret\_ty\_opt}^{/\!/\texttt{\#TE}}$;
- define `call` as the call with name `name'`, parameters taken from `params1`, arguments `args`, and call type `func_sig`.subprogram_type.

**Formally**

$$\frac{\begin{array}{c}\mathit{unzip3}(\texttt{typed\_args}) = (\texttt{arg\_types}, \texttt{args}, \texttt{sess\_args}) \qquad \texttt{ses\_args} := \bigcup \texttt{sess\_args} \\ \mathit{subprogram\_for\_name}(\texttt{tenv}, \texttt{name}, \texttt{arg\_types}) \xrightarrow{\text{type}} (\texttt{name1}, \texttt{func\_sig}, \texttt{ses\_call}) \;/\!/\; \texttt{\#TE} \\ \mathit{check}\left(\left(\begin{array}{l}\texttt{func\_sig.subprogram\_type} = \texttt{call\_type} \vee \\ (\texttt{func\_sig.subprogram\_type} = \texttt{ST\_Getter} \wedge \\ \quad \texttt{call\_type} = \texttt{ST\_Function})\end{array}\right), \texttt{TE\_BC}\right) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_args} \cup \texttt{ses\_call} \\ \mathit{insert\_stdlib\_param}(\texttt{func\_sig}, \texttt{params}, \texttt{arg\_types}) \xrightarrow{\text{type}} \texttt{params1} \\ \mathit{equal\_length}(\texttt{func\_sig.parameters}, \texttt{params1}) \xrightarrow{\text{type}} \texttt{param\_arity\_match} \\ \mathit{check}(\texttt{param\_arity\_match}, \texttt{TE\_BC}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{equal\_length}(\texttt{func\_sig.args}, \texttt{args}) \xrightarrow{\text{type}} \texttt{arity\_match} \\ \mathit{check}(\texttt{arity\_match}, \texttt{TE\_BC}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_params\_typesat}(\texttt{tenv}, \texttt{func\_sig.parameters}, \texttt{params1}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{eqs} := [(\texttt{x}_i, \_) \in \texttt{func\_sig.func\_sig\_params}_i,\ (\_, \texttt{v}_i, \_) \in \texttt{params1} : (\texttt{x}_i, \texttt{v}_i)] \\ \mathit{check\_args\_typesat}(\texttt{tenv}, \texttt{func\_sig.args}, \texttt{arg\_types}, \texttt{eqs}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_ret\_ty}(\texttt{tenv}, \texttt{call\_type}, \texttt{func\_sig.return\_type}, \texttt{eqs}) \xrightarrow{\text{type}} \texttt{ret\_ty\_opt} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_call\_actuals\_typed}(\texttt{tenv}, \texttt{name}, \texttt{params}, \texttt{typed\_args}, \texttt{call\_type}) \xrightarrow{\text{type}} \left(\overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{name'}, \\ \text{params} & : & [(\_, \texttt{v}_i, \_) \in \texttt{params1} : \texttt{v}_i], \\ \text{args} & : & \texttt{args}, \\ \text{call\_type} & : & \texttt{func\_sig.subprogram\_type}\end{array}\right\}}^{\texttt{call}}, \texttt{ret\_ty\_opt}, \texttt{ses}\right)}$$

### TypingRule.InsertStdlibParam

The function

$$\mathit{insert\_stdlib\_param}(\overbrace{\mathsf{func}}^{\texttt{func\_sig}}, \overbrace{(\mathsf{ty} \times \mathsf{expr})^*}^{\texttt{params}}, \overbrace{\mathsf{ty}^*}^{\texttt{arg\_types}}) \longrightarrow \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{params1}}$$

inserts the (optionally) omitted input parameter of a standard library function call.

Note that this function relies on all standard library functions with input parameters having one of two simple forms:

```
func stdlibA{N}  (arg1: bits(N), ...) => ...
func stdlibB{M,N}(arg1: bits(N), ...) => bits(...M...)
```

### Example: Inserting Parameters in Calls to Standard Library Subprograms

The specification in Listing 23.4 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 23.4: 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 *paramtype* 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

CAN_INSERT

$$\texttt{can\_insert\_stdlib\_param} := \begin{pmatrix} \textit{can\_omit\_stdlib\_param}(\texttt{func\_sig}) & \wedge \\ |\texttt{params}| < |\texttt{func\_sig.parameters}| & \wedge \\ \texttt{arg\_types} \neq [\,] \end{pmatrix}$$

$$\texttt{can\_insert\_stdlib\_param} = \texttt{TRUE}$$

$$\texttt{arg\_types} \overset{\text{is}}{=} [\texttt{t}] + \_ \qquad \textit{get\_bitvector\_width}(\texttt{tenv}, \texttt{t}) \xrightarrow[\text{Constraint\_Exact}]{\text{type}} \texttt{width} \;\; /\!/ \;\; \texttt{\#TE}$$

$$\texttt{param\_type} := \texttt{T\_Int}(\texttt{WellConstrained}([\;\overbrace{\texttt{width}}\;]))$$

$$\texttt{params1} := \texttt{params} + [(\texttt{param\_type}, \texttt{width}, \emptyset)]$$

---

$$\textit{insert\_stdlib\_param}(\texttt{func\_sig}, \texttt{params}, \texttt{arg\_types}) \longrightarrow \texttt{params1}$$

CANNOT_INSERT

$$\texttt{can\_insert\_stdlib\_param} := \left( \begin{array}{ll} \textit{can\_omit\_stdlib\_param}(\texttt{func\_sig}) & \wedge \\ |\texttt{params}| < |\texttt{func\_sig.parameters}| & \wedge \\ \texttt{arg\_types} \neq [\,] & \end{array} \right)$$

$$\texttt{can\_insert\_stdlib\_param} = \texttt{FALSE}$$

---

$$\textit{insert\_stdlib\_param}(\texttt{func\_sig}, \texttt{params}, \texttt{arg\_types}) \longrightarrow \overbrace{\texttt{params}}^{\texttt{params1}}$$

### TypingRule.CanOmmitStdlibParam

The function

$$\textit{can\_omit\_stdlib\_param}(\ \overbrace{\mathsf{func}}^{\texttt{func\_sig}}\ ) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

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**

$$\texttt{declared\_param} := \begin{cases} \texttt{func\_sig}.\text{builtin} & \\ \langle \texttt{n} \rangle & \text{if } \texttt{func\_sig}.\text{parameters} = [(\texttt{n}, \_)] \\ \langle \texttt{n} \rangle & \text{if } \texttt{func\_sig}.\text{parameters} = [\_, (\texttt{n}, \_)] \\ \texttt{None} & \text{else} \end{cases}$$

$$\frac{\texttt{b} := \texttt{declared\_param} = \langle \texttt{n} \rangle \wedge \texttt{func\_sig}.\text{args} = (\_, \mathsf{T\_Bits}(\mathsf{E\_Var}(\texttt{n}), \_))}{\mathit{can\_omit\_stdlib\_param}(\texttt{func\_sig}) \xrightarrow{\text{type}} \texttt{b}}$$

**TypingRule.CheckParamsTypeSat**

The function

$$\mathit{check\_params\_typesat}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\texttt{func\_sig\_params}}, \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{params}}) \longrightarrow \{\mathsf{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 23.5, 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 $\mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{N}))$ — and there is no need to check that the type of `EIGHT`, which is `integer{8}`, type-satisfies $\mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{N}))$.

Listing 23.5: 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - checking that `ty_actual` represents a constrained integer yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - One of the following applies:
    - All of the following apply (PARAMETERIZED):
      - `ty_actual` is a parameterized integer type for the parameter `x`, that is, $\langle \mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{x}))\rangle$.
    - All of the following apply (OTHER):
      - `ty_decl_opt` is not `None`, that is, $\langle$`ty_decl`$\rangle$;
      - `ty_decl` is not the parameterized integer type for the parameter `x`;
      - checking that `ty_actual` type-satisfies `ty_decl` in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - applying *check_params_typesat* to `func_sig_params1` and `params1` in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$.

**Formally**

EMPTY

$$\mathit{check\_params\_typesat}(\text{tenv}, \overbrace{[\,]}^{\texttt{func\_sig\_params}}, \_) \xrightarrow{\text{type}} \texttt{TRUE}$$

PARAMETERIZED

$$\frac{\begin{array}{c}\texttt{func\_sig\_params} = [(\texttt{x}, \texttt{ty\_decl\_opt})] + \texttt{func\_sig\_params1} \\ \texttt{params} = [(\texttt{ty\_actual}, \texttt{e\_actual}, \texttt{ses\_actual})] + \texttt{params1} \\ \mathit{check\_symbolically\_evaluable}(\texttt{ses\_actual}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_constrained\_integer}(\text{tenv}, \texttt{ty\_actual}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \texttt{ty\_decl\_opt} = \langle \mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{x}))\rangle \\ \mathit{check\_params\_typesat}(\text{tenv}, \texttt{func\_sig\_params1}, \texttt{params1}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{check\_params\_typesat}(\text{tenv}, \texttt{func\_sig\_params}, \texttt{params}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

OTHER

$$\frac{\begin{array}{c}\texttt{func\_sig\_params} = [(\texttt{x}, \texttt{ty\_decl\_opt})] + \texttt{func\_sig\_params1} \\ \texttt{params} = [(\texttt{ty\_actual}, \texttt{e\_actual}, \texttt{ses\_actual})] + \texttt{params1} \\ \mathit{check\_symbolically\_evaluable}(\texttt{ses\_actual}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_constrained\_integer}(\text{tenv}, \texttt{ty\_actual}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \texttt{ty\_decl\_opt} \overset{\text{is}}{=} \langle \texttt{ty\_decl}\rangle \qquad \texttt{ty\_decl} \neq \mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{x})) \\ \mathit{checked\_typesat}(\text{tenv}, \texttt{ty\_actual}, \texttt{ty\_decl}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_params\_typesat}(\text{tenv}, \texttt{func\_sig\_params1}, \texttt{params1}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{check\_params\_typesat}(\text{tenv}, \texttt{func\_sig\_params}, \texttt{params}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

### TypingRule.RenameTyEqs

The function

$$\mathit{rename\_ty\_eqs}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\texttt{eqs}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{new\_ty}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 23.6, 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 23.6: 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 $\texttt{new\_c}_\texttt{i}$;
  - define `new_constraints` as the list of constraints $\texttt{new\_c}_\texttt{i}$, for `i` in *indices*(`constraints`);
  - define `new_ty` as the well-constrained integer type with constraints `new_constraints`.
- All of the following apply (T_INT_PARAMETERIZED):
  - `ty` is a parameterized integer type for the parameter `name`;
  - applying *subst_expr_normalize* to `eqs` and the expression E_Var(`name`) yields `e`;
  - define `new_ty` as the well-constrained integer type with the single constraint for `e`, that is, T_Int(WellConstrained(Constraint_Exact(`e`))).
- All of the following apply (T_TUPLE):
  - `ty` is the tuple type over the list of tuples `tys`, that is, T_Tuple(`tys`);

  - applying *rename_ty_eqs* to `eqs` and the type `tys[i]`, for each `i` in *indices*(`tys`), yields the type $\texttt{new\_ty}_\texttt{i}$;
  - define `new_tys` as the list of types $\texttt{new\_ty}_\texttt{i}$, for each `i` in *indices*(`tys`);
  - define `new_ty` as the tuple type over `new_tys`, that is, T_Tuple(`new_tys`).

- All of the following apply (OTHER):
  - `ty` is not one of the types in the previous cases, that is, `ty` is not a bitvector type, nor an integer type, nor a tuple type;
  - `new_ty` is `ty`.

**Formally**

T_BITS

$$\frac{\textit{subst\_expr\_normalize}(\mathsf{tenv}, \mathsf{eqs}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}{\textit{rename\_ty\_eqs}(\mathsf{tenv}, \mathsf{eqs}, \overbrace{\mathsf{T\_Bits}(\mathsf{e}, \mathtt{fields})}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Bits}(\mathtt{new\_e}, \mathtt{fields})}^{\mathtt{new\_ty}}}$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{c}\mathtt{i} \in \textit{indices}(\mathtt{constraints}) : \textit{subst\_constraint}(\mathsf{tenv}, \mathtt{constraints[i]}) \xrightarrow{\mathsf{type}} \mathtt{new\_c_i} \\ \mathtt{new\_constraints} := [\mathtt{i} \in \textit{indices}(\mathtt{constraints}) : \mathtt{new\_c_i}] \\ \mathtt{new\_ty} := \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{new\_constraints}))\end{array}}{\textit{rename\_ty\_eqs}(\mathsf{tenv}, \mathsf{eqs}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{constraints}))}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \mathtt{new\_ty}}$$

T_INT_PARAMETERIZED

$$\frac{\begin{array}{c}\textit{subst\_expr\_normalize}(\mathsf{eqs}, \mathsf{E\_Var}(\mathtt{name})) \xrightarrow{\mathsf{type}} \mathtt{e} \\ \mathtt{new\_ty} := \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathsf{Constraint\_Exact}(\mathtt{e})))\end{array}}{\textit{rename\_ty\_eqs}(\mathsf{tenv}, \mathsf{eqs}, \overbrace{\mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{name}))}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \mathtt{new\_ty}}$$

T_TUPLE

$$\frac{\begin{array}{c}\mathtt{i} \in \textit{indices}(\mathtt{tys}) : \textit{rename\_ty\_eqs}(\mathtt{eqs}, \mathtt{tys[i]}) \xrightarrow{\mathsf{type}} \mathtt{new\_ty_i} \\ \mathtt{new\_tys} := [\mathtt{i} \in \textit{indices}(\mathtt{tys}) : \mathtt{new\_ty_i}]\end{array}}{\textit{rename\_ty\_eqs}(\mathsf{tenv}, \mathsf{eqs}, \overbrace{\mathsf{T\_Tuple}(\mathtt{tys})}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Tuple}(\mathtt{new\_tys})}^{\mathtt{new\_ty}}}$$

OTHER

$$\frac{\textit{ast\_label}(\mathtt{ty}) \notin \{\mathsf{T\_Bits}, \mathsf{T\_Int}, \mathsf{T\_Tuple}\}}{\textit{rename\_ty\_eqs}(\mathsf{tenv}, \mathsf{eqs}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{ty}}^{\mathtt{new\_ty}}}$$

### TypingRule.SubstExprNormalize

The function

$$\mathit{subst\_expr\_normalize}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\texttt{new\_e}}^{\text{expr}}$$

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 23.7, 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 23.7: 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

$$\frac{\mathit{subst\_expr}(\text{tenv}, \texttt{e}) \xrightarrow{\text{type}} \texttt{e1} \qquad \mathit{normalize}(\text{tenv}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{new\_e}}{\mathit{subst\_expr\_normalize}(\text{tenv}, \texttt{eqs}, \texttt{e}) \xrightarrow{\text{type}} \texttt{new\_e}}$$

### TypingRule.SubstExpr

The function

$$\mathit{subst\_expr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\texttt{substs}}, \overbrace{\mathsf{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{new\_e}}$$

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 unary operator expression for the operator `op` and expression `e1'`, that is, E_Unop(`op`, `e1'`).

- All of the following apply (E_COND):
  - `e` is the conditional expression for expressions `e1`, `e2`, and `e3`, that is, E_Cond(`e1`, `e2`, `e3`);
  - applying *subst_expr* to `substs` and `e1` in `tenv` yields `e1'`;
  - applying *subst_expr* to `substs` and `e2` in `tenv` yields `e2'`;
  - applying *subst_expr* to `substs` and `e3` in `tenv` yields `e3'`;
  - define `new_e` as the conditional expression for expressions `e1'`, `e2'`, and `e3'`, that is, E_Cond(`e1'`, `e2'`, `e3'`).
- All of the following apply (E_CALL):
  - `e` is the call expression for subprogram `x` with arguments `args` and parameter expressions `param_args`, that is, E_Call(`x`, `args`, `param_args`);
  - applying *subst_expr* to `substs` and every argument expression `args`[`i`], for `i` in *indices*(`args`) yields $\texttt{e}_\texttt{i}$;
  - define `args'` as $\texttt{e}_\texttt{i}$ for each `i` in *indices*(`args`);
  - define `new_e` as the call expression for subprogram `x` with arguments `args'` and parameter expressions `param_args`, that is, E_Call(`x`, `args'`, `param_args`).
- All of the following apply (E_GETARRAY):
  - `e` is the array access expression for base expression `e1` and index expression `e2`, that is, E_GetArray(`e1`, `e2`);
  - applying *subst_expr* to `substs` and `e1` in `tenv` yields `e1'`;
  - applying *subst_expr* to `substs` and `e2` in `tenv` yields `e2'`;
  - define `new_e` as the array access expression for base expression `e1'` and index expression `e2'`, that is, E_GetArray(`e1'`, `e2'`).
- All of the following apply (E_GETENUMARRAY):
  - `e` is the array access expression for base expression `e1` and an enumeration-typed index expression `e2`, that is, E_GetEnumArray(`e1`, `e2`);
  - applying *subst_expr* to `substs` and `e1` in `tenv` yields `e1'`;
  - applying *subst_expr* to `substs` and `e2` in `tenv` yields `e2'`;
  - define `new_e` as the array access expression for base expression `e1'` and enumeration-typed index expression `e2'`, that is, E_GetEnumArray(`e1'`, `e2'`).
- All of the following apply (E_GETFIELD):
  - `e` is the field access expression for base expression `e` and field `x`, that is, E_GetField(`e1`, `x`);
  - applying *subst_expr* to `substs` and `e1` in `tenv` yields `e1'`;

    * define `new_e` as the field access expression for base expression `e` and field `x`, that is, E_GetField(`e1'`, `x`).

- All of the following apply (E_GETFIELDS):
    * `e` is the access to fields `fields` with base expression `e1`, that is, E_GetFields(`e1`, `fields`);
    * applying *subst_expr* to `substs` and `e1` in `tenv` yields `e1'`;
    * define `new_e` as the access to fields `fields` with base expression `e1'`, that is, E_GetFields(`e1'`, `fields`).
- All of the following apply (E_GETITEM):
    * `e` is the access to tuple item `i` of the tuple expression `e1`, that is, E_GetItem(`e1`, `i`);
    * applying *subst_expr* to `substs` and `e1` in `tenv` yields `e1'`;
    * define `new_e` as the access to tuple item `i` of the tuple expression `e1'`, that is, E_GetItem(`e1'`, `i`).
- All of the following apply (E_PATTERN):
    * `e` is the pattern expression of expression `e1` and patterns `ps`, that is, E_Pattern(`e1`, `ps`);
    * applying *subst_expr* to `substs` and `e1` in `tenv` yields `e1'`;
    * define `new_e` as the pattern expression of expression `e1'` and patterns `ps`, that is, E_Pattern(`e1'`, `ps`).
- All of the following apply (E_RECORD):
    * `e` is the record expression of record type `t` and list of fields `fields`;
    * for every pair (`x`, `e1`) in `fields`, applying *subst_expr* to `substs` `e1` in `tenv` yields $\texttt{e1'}_{\texttt{x}}$;
    * define `fields'` as the list of pairs $(\texttt{x}, \texttt{e1'}_{\texttt{x}})$ for every pair (`x`, `e1`) in `fields`;
    * define `new_e` as the record expression of record type `t` and list of fields `fields'`.
- All of the following apply (E_SLICE):
    * `e` is the slicing expression for subexpression `e1` and list of slices `slices`, that is, E_Slice(`e1`, `slices`);
    * applying *subst_expr* to `e1` in `tenv` yields `e1'`;
    * define `new_e` as slicing expression for subexpression `e1'` and list of slices `slices`, that is, E_Slice(`e1'`, `slices`).
- All of the following apply (E_TUPLE):
    * `e` is the tuple expression of expressions `e_s`, that is, E_Tuple(`e_s`);

  - applying *subst_expr* to `substs` and every expression `e_s`[`i`] in `tenv`, for every `i` in *indices*(`e_s`) yields $\texttt{new\_e}_{\texttt{i}}$;
  - define `es'` as the list of expressions $\texttt{new\_e}_{\texttt{i}}$, for every `i` in *indices*(`e_s`);
  - define `new_e` as the tuple expression of expressions `es'`, that is, E_Tuple(`es'`).

- All of the following apply (E_ARRAY):
  - `e` is an array construction expression with length expression `length` and value expression `value`, that is, E_Array{length : `length`, value : `value`};
  - applying *subst_expr* to `substs` and `length` in `tenv` yields `length'`;
  - applying *subst_expr* to `substs` and `value` in `tenv` yields `value'`;
  - define `new_e` as the array construction expression with length expression `length'` and initial element value expression `value'`, that is, E_Array{length : `length'`, value : `value'`}.

- All of the following apply (E_ENUMARRAY):
  - `e` is an array construction expression for an enumeration-typed index with list of labels `labels` and initial element value expression `value`, that is, E_EnumArray{labels : `labels`, value : `value`};
  - applying *subst_expr* to `substs` and `value` in `tenv` yields `value'`;
  - define `new_e` as the array construction expression with list of labels `labels` and value expression `value'`, that is, E_EnumArray{labels : `labels`, value : `value'`}.

- All of the following apply (E_ATC):
  - `e` is the type assertion of expression `e1` and type `t`, that is, E_ATC(`e1`, `t`);
  - applying *subst_expr* to `substs` and `e1` in `tenv` yields `e1'`;
  - define `new_e` as the type assertion of expression `e1'` and type `t`, that is, E_ATC(`e1'`, `t`).

- All of the following apply (OTHER):
  - `e` is either a literal expression or an arbitrary value expression;
  - define `new_e` as `e`.

**Formally**

E_VAR_IN_SUBSTS

$$\frac{\mathit{assoc\_opt}(\texttt{s}, \texttt{substs}) \xrightarrow{\text{type}} \langle \texttt{new\_e} \rangle}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\text{E\_Var}(\texttt{s})}^{\texttt{e}}) \xrightarrow{\text{type}} \texttt{new\_e}}$$

E_VAR_NOT_IN_SUBSTS

$$\frac{\mathit{assoc\_opt}(\texttt{s}, \texttt{substs}) \xrightarrow{\text{type}} \texttt{None}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\texttt{E\_Var}(\texttt{s})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\texttt{e}}^{\texttt{new\_e}}}$$

E_UNOP

$$\frac{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\texttt{E\_Unop}(\texttt{op}, \texttt{e1})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\texttt{E\_Unop}(\texttt{op}, \texttt{e1'})}^{\texttt{new\_e}}}$$

E_BINOP

$$\frac{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'} \qquad \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2'}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{e1'}, \texttt{e2'})}^{\texttt{new\_e}}}$$

E_COND

$$\frac{\begin{array}{c}\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'} \\ \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2'} \qquad \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e3'}) \xrightarrow{\text{type}} \texttt{e3'}\end{array}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\texttt{E\_Cond}(\texttt{e1}, \texttt{e2}, \texttt{e3})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\texttt{E\_Cond}(\texttt{e1'}, \texttt{e2'}, \texttt{e3'})}^{\texttt{new\_e}}}$$

E_CALL

$$\frac{\begin{array}{c}\texttt{i} \in \mathit{indices}(\texttt{args}) : \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{args}[\texttt{i}]) \xrightarrow{\text{type}} \texttt{e}_\texttt{i} \\ \texttt{args'} := [\texttt{i} \in \mathit{indices}(\texttt{args}) : \texttt{e}_\texttt{i}]\end{array}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\texttt{E\_Call}(\texttt{x}, \texttt{args}, \texttt{param\_args})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\texttt{E\_Call}(\texttt{x}, \texttt{args'}, \texttt{param\_args})}^{\texttt{new\_e}}}$$

E_GETARRAY

$$\frac{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'} \qquad \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2'}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\texttt{E\_GetArray}(\texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\texttt{E\_GetArray}(\texttt{e1'}, \texttt{e2'})}^{\texttt{new\_e}}}$$

E_GETENUMARRAY

$$\frac{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'} \qquad \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2'}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\texttt{E\_GetEnumArray}(\texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\texttt{E\_GetEnumArray}(\texttt{e1'}, \texttt{e2'})}^{\texttt{new\_e}}}$$

E_GETFIELD

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \mathtt{x})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_GetField}(\mathtt{e1'}, \mathtt{x})}^{\mathtt{new\_e}}}$$

E_GETFIELDS

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e1}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_GetFields}(\mathtt{e1'}, \mathtt{fields})}^{\mathtt{new\_e}}}$$

E_GETITEM

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \overbrace{\mathsf{E\_GetItem}(\mathtt{e1}, \mathtt{i})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_GetItem}(\mathtt{e1'}, \mathtt{i})}^{\mathtt{new\_e}}}$$

E_PATTERN

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \overbrace{\mathsf{E\_Pattern}(\mathtt{e1}, \mathtt{ps})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Pattern}(\mathtt{e1'}, \mathtt{ps})}^{\mathtt{new\_e}}}$$

E_RECORD

$$\frac{\begin{array}{c}(\mathtt{x}, \mathtt{e1}) \in \mathtt{fields} : \mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{e1}_{\mathtt{x}} \\ \mathtt{fields'} := [(\mathtt{x}, \mathtt{e1}) \in \mathtt{fields} : (\mathtt{x}, \mathtt{e1}_{\mathtt{x}})]\end{array}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \overbrace{\mathsf{E\_Record}(\mathtt{t}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Record}(\mathtt{t}, \mathtt{fields'})}^{\mathtt{new\_e}}}$$

E_SLICE

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \overbrace{\mathsf{E\_Slice}(\mathtt{e1}, \mathtt{slices})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Slice}(\mathtt{e1'}, \mathtt{slices})}^{\mathtt{new\_e}}}$$

E_TUPLE

$$\frac{\begin{array}{c}\mathtt{i} \in \mathit{indices}(\mathtt{e\_s}) : \mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \mathtt{e\_s}[\mathtt{i}]) \xrightarrow{\mathsf{type}} \mathtt{new\_e}_{\mathtt{i}} \\ \mathtt{es'} := [\mathtt{i} \in \mathit{indices}(\mathtt{e\_s}) : \mathtt{new\_e}_{\mathtt{i}}]\end{array}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathsf{substs}, \overbrace{\mathsf{E\_Tuple}(\mathtt{e\_s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Tuple}(\mathtt{es'})}^{\mathtt{new\_e}}}$$

E_ARRAY

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathtt{length}) \xrightarrow{\text{type}} \mathtt{length'} \quad \mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathtt{value}) \xrightarrow{\text{type}} \mathtt{value'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_Array}\{\text{length} : \mathtt{length}, \text{value} : \mathtt{value}\}}^{\mathsf{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_Array}\{\text{length} : \mathtt{length}, \text{value} : \mathtt{value'}\}}^{\mathsf{new\_e}}}$$

E_ENUMARRAY

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathtt{value}) \xrightarrow{\text{type}} \mathtt{value'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_EnumArray}\{\text{labels} : \mathtt{labels}, \text{value} : \mathtt{value}\}}^{\mathsf{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_EnumArray}\{\text{labels} : \mathtt{length}, \text{value} : \mathtt{value'}\}}^{\mathsf{new\_e}}}$$

E_ATC

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_ATC}(\mathtt{e1}, \mathtt{t})}^{\mathsf{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_ATC}(\mathtt{e1'}, \mathtt{t})}^{\mathsf{new\_e}}}$$

OTHER

$$\frac{\mathit{ast\_label}(\mathsf{e}) \in \{\mathsf{E\_Literal}, \mathsf{E\_Arbitrary}\}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathsf{e}) \xrightarrow{\text{type}} \overbrace{\mathsf{e}}^{\mathsf{new\_e}}}$$

### TypingRule.SubstConstraint

The function

$$\mathit{subst\_constraint}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\text{eqs}}, \overbrace{\mathsf{int\_constraint}}^{\mathsf{c}}) \longrightarrow \overbrace{\mathtt{new\_c}}^{\text{int\_constraint}}$$

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 23.7, 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, Constraint_Exact(`e`);
  - applying *subst_expr_normalize* in `tenv` to `eqs` and `e` yields `new_e`;
  - define `new_c` as the exact constraint for the expression `new_e`, that is, Constraint_Exact(`new_e`).
- All of the following apply (RANGE):
  - `c` is a range constraint for the expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *subst_expr_normalize* in `tenv` to `eqs` and `e1` yields `e1'`;
  - applying *subst_expr_normalize* in `tenv` to `eqs` and `e2` yields `e2'`;
  - define `new_c` as the range constraint for the expressions `e1'` and `e2'`, that is, Constraint_Range(`e1'`, `e2'`).

**Formally**

EXACT

$$\frac{\mathit{subst\_expr\_normalize}(\mathsf{tenv}, \mathtt{eqs}, \mathtt{e}) \xrightarrow{\text{type}} \mathtt{new\_e}}{\mathit{subst\_constraint}(\mathsf{tenv}, \mathtt{eqs}, \overbrace{\mathsf{Constraint\_Exact}(\mathtt{e})}^{\mathtt{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Exact}(\mathtt{new\_e})}^{\mathtt{new\_c}}}$$

RANGE

$$\frac{\begin{array}{c}\mathit{subst\_expr\_normalize}(\mathsf{tenv}, \mathtt{eqs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1'} \\ \mathit{subst\_expr\_normalize}(\mathsf{tenv}, \mathtt{eqs}, \mathtt{e2}) \xrightarrow{\text{type}} \mathtt{e2'}\end{array}}{\mathit{subst\_constraint}(\mathsf{tenv}, \mathtt{eqs}, \overbrace{\mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Range}(\mathtt{e1'}, \mathtt{e2'})}^{\mathtt{new\_c}}}$$

**TypingRule.CheckArgsTypeSat**

The function

$$\mathit{check\_args\_typesat}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\mathtt{func\_sig\_args}}, \overbrace{\mathsf{ty}^*}^{\mathtt{arg\_types}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\mathtt{eqs}}) \longrightarrow \{\mathtt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

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 23.6, 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.

$$\text{EMPTY}\quad \frac{}{\mathit{check\_args\_typesat}(\text{tenv}, \overbrace{[\,]}^{\texttt{func\_sig\_args}}, \overbrace{[\,]}^{\texttt{arg\_types}}, \text{eqs}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

$$\text{NON\_EMPTY}\quad \frac{\begin{array}{c}\texttt{func\_sig\_args} = [(\_, \texttt{ty\_decl})] + \texttt{func\_sig\_args1} \\ \texttt{arg\_types} = [\texttt{ty\_actual}] + \texttt{arg\_types1} \\ \mathit{rename\_ty\_eqs}(\text{tenv}, \text{eqs}, \texttt{ty\_decl}) \xrightarrow{\text{type}} \texttt{ty\_decl'} \;//\; \texttt{\#TE} \\ \mathit{checked\_typesat}(\text{tenv}, \texttt{ty\_actual}, \texttt{ty\_decl'}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ \mathit{check\_args\_typesat}(\text{tenv}, \texttt{func\_sig\_args1}, \texttt{arg\_types1}, \text{eqs}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE}\end{array}}{\mathit{check\_args\_typesat}(\text{tenv}, \texttt{func\_sig\_args}, \texttt{arg\_types}, \text{eqs}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

### TypingRule.AnnotateRetTy

The function

$$\mathit{annotate\_ret\_ty}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{sub\_program\_type}}^{\texttt{call\_type}}, \overbrace{\langle \text{ty} \rangle}^{\texttt{func\_sig\_ret\_ty\_opt}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\texttt{eqs3}}) \longrightarrow \overbrace{\langle \text{ty} \rangle}^{\texttt{ret\_ty\_opt}} \cup \overbrace{\mathsf{T}\,\text{TypeError}}^{\texttt{\#TE}}$$

annotates the optional return type `func_sig_ret_ty_opt` given with the subprogram type `call_type` with respect to the parameter expressions `eqs`, yielding the optional annotated type `ret_ty_opt`. Otherwise, the result is a type error.

### Example: Annotating the Return Type of a Subprogram Call

In Listing 23.8, 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 23.8: 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 23.9 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 23.9: 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 ⟨`ty`⟩;
  - applying *rename_ty_eqs* to `eqs` and `ty` yields `ty1`[//#TE];
  - `ret_ty_opt` is ⟨`ty1`⟩.
- All of the following apply (PROCEDURE_OR_SETTER):
  - `call_type` is one of `ST_Procedure` or `ST_Setter`;
  - `func_sig_ret_ty_opt` is `None`;
  - define `ret_ty_opt` as `None`.
- All of the following apply (RET_TYPE_MISMATCH):
  - the condition that `call_type` is one of `ST_Procedure` or `ST_Setter` if and only if `func_sig_ret_ty_opt` is `None` does not hold;
  - the result is a type error indicating the mismatch.

**Formally**

FUNCTION_OR_GETTER

$$\frac{\texttt{call\_type} \in \{\texttt{ST\_Function}, \texttt{ST\_Getter}\} \qquad \mathit{rename\_ty\_eqs}(\texttt{eqs}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{ty1} \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_ret\_ty}(\texttt{tenv}, \texttt{call\_type}, \overbrace{\langle \texttt{ty} \rangle}^{\texttt{func\_sig\_ret\_ty\_opt}}, \texttt{eqs}) \xrightarrow{\text{type}} \overbrace{\langle \texttt{ty1} \rangle}^{\texttt{ret\_ty\_opt}}}$$

PROCEDURE_OR_SETTER

$$\frac{\texttt{call\_type} \in \{\texttt{ST\_Procedure}, \texttt{ST\_Setter}\}}{\mathit{annotate\_ret\_ty}(\texttt{tenv}, \texttt{call\_type}, \overbrace{\texttt{None}}^{\texttt{func\_sig\_ret\_ty\_opt}}, \texttt{eqs}) \xrightarrow{\text{type}} \overbrace{\texttt{None}}^{\texttt{ret\_ty\_opt}}}$$

RET_TYPE_MISMATCH

$$\frac{\neg \left( \begin{array}{c} \texttt{call\_type} \in \{\texttt{ST\_Procedure}, \texttt{ST\_Setter}\} \leftrightarrow \\ \texttt{func\_sig\_ret\_ty\_opt} = \texttt{None} \end{array} \right)}{\mathit{annotate\_ret\_ty}(\texttt{tenv}, \texttt{call\_type}, \texttt{func\_sig\_ret\_ty\_opt}, \texttt{eqs}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_BC})}$$

### TypingRule.SubprogramForName

The function

$$\mathit{subprogram\_for\_name}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{S}}^{\texttt{name}}, \overbrace{\texttt{ty}^*}^{\texttt{caller\_arg\_types}}) \longrightarrow (\overbrace{\mathbb{S}}^{\texttt{name'}}, \overbrace{\text{func}}^{\texttt{callee}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\text{TTypeError}}^{\texttt{\#TE}}$$

looks up the static environment `tenv` for a subprogram associated with `name` and the list of argument types `callee_arg_types` and determines which one of the following cases holds:

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 23.10 shows an example where all subprogram calls match subprogram declarations.

Listing 23.10: 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 23.11 shows an example where the subprogram call `add_10(5)` is illegal, since no subprogram named `add_10` is declared.

Listing 23.11: Subprogram name undefined

```
func main() => integer
begin
    - = add_10(5);
    return 0;
end;
```

Listing 23.10 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 23.12: 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 ($G^{\text{tenv}}$.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$^{\#\text{TE}}$;
  - 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`$^{\#\text{TE}}$;
  - `matching_renamings` contains a single element — (`name'`, `callee`)$^{\#\text{TE}}$;

**Formally**

UNDEFINED

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = \bot}{\textit{subprogram\_for\_name}(\text{tenv}, \texttt{name}, \texttt{caller\_arg\_types}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_UI})}$$

NO_CANDIDATES

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = (\texttt{renaming\_set}, \texttt{ses}) \quad \textit{filter\_call\_candidates}(\text{tenv}, \texttt{caller\_arg\_types}, \texttt{renaming\_set}) \xrightarrow{\text{type}} \emptyset \;//\; \#\text{TE}}{\textit{subprogram\_for\_name}(\text{tenv}, \texttt{name}, \texttt{caller\_arg\_types}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_BC})}$$

ONE_CANDIDATE

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = (\texttt{renaming\_set}, \texttt{ses}) \quad \textit{filter\_call\_candidates}(\text{tenv}, \texttt{caller\_arg\_types}, \texttt{renaming\_set}) \xrightarrow{\text{type}} \texttt{matching\_renamings} \;//\; \#\text{TE} \quad \texttt{matching\_renamings} = [(\texttt{name'}, \texttt{callee})]}{\textit{subprogram\_for\_name}(\text{tenv}, \texttt{name}, \texttt{caller\_arg\_types}) \xrightarrow{\text{type}} (\texttt{name'}, \texttt{callee}, \texttt{ses})}$$

### TypingRule.FilterCallCandidates

The helper function

$$\mathit{filter\_call\_candidates}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}^*}^{\texttt{formal\_types}}, \overbrace{\mathcal{P}(\mathbb{S})}^{\texttt{candidates}}) \longrightarrow \overbrace{(\mathbb{S} \times \mathsf{func})^*}^{\texttt{matches}}$$

iterates over the list of unique subprogram names in `candidates` and checks whether their lists of arguments clash with the types in `formal_types` in `tenv`. The result is the set of pairs consisting of the names and function definitions of the subprograms whose arguments clash in `candidates`. Otherwise, the result is a type error.

The names `candidates` are assumed to exist in $G^{\mathsf{tenv}}.\mathsf{subprograms}$.

### Example: Filtering Subprograms Matching a Call

In Listing 23.10, 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 23.12, 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 $\mathtt{b}^{/\!/\texttt{\#TE}}$;
  - filtering the call candidates in `candidates1` with `formal_types` in `tenv` yields $\mathtt{matches1}^{/\!/\texttt{\#TE}}$;
  - if `b` is `TRUE` then `matches` is the list with `head` (`name`, `func_def`) and `tail` `matches1`, and otherwise it is `matches1`.

### Formally

NO_CANDIDATES

$$\mathit{filter\_call\_candidates}(\mathsf{tenv}, \mathtt{formal\_types}, \overbrace{[\,]}^{\texttt{candidates}}) \xrightarrow{\mathsf{type}} \overbrace{[\,]}^{\texttt{matches}}$$

CANDIDATES_EXIST

$$\frac{\begin{array}{c}\texttt{func\_def} := G^{\texttt{tenv}}.\mathsf{subprograms}(\texttt{name}) \\ \mathit{has\_arg\_clash}(\texttt{tenv}, \texttt{formal\_types}, \texttt{func\_def.args}) \xrightarrow{\text{type}} \texttt{b} \;//\; \texttt{\#TE} \\ \mathit{filter\_call\_candidates}(\texttt{tenv}, \texttt{formal\_types}, \texttt{candidates1}) \xrightarrow{\text{type}} \texttt{matches1} \;//\; \texttt{\#TE} \\ \texttt{matches} := \mathsf{choice}(\texttt{b},\ [(\texttt{name}, \texttt{func\_def})] + \texttt{matches1},\ \texttt{matches1})\end{array}}{\mathit{filter\_call\_candidates}(\texttt{tenv}, \texttt{formal\_types}, \overbrace{[\texttt{name}] + \texttt{candidates1}}^{\texttt{candidates}}) \xrightarrow{\text{type}} \texttt{matches}}$$

### TypingRule.HasArgClash

The function

$$\mathit{has\_arg\_clash}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}^*}^{\texttt{f\_tys}}, \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{args}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

checks whether a list of types `f_tys` clashes with the list of types appearing in the list of arguments `args` in `tenv`, yielding the result in `b`. Otherwise, the result is a type error.

### Example: Argument Clashing

Listing 23.13 shows examples of types — the arguments of procedures – that do not clash and types that do clash (shown in comments).

Listing 23.13: 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 $\texttt{TRUE}^{/\!/\texttt{FALSE},\#\texttt{TE}}$;
- `b` is `TRUE` (unless the rule short-circuited with `FALSE` or a type error).

### Formally

$$\frac{\begin{array}{c}\mathit{equal\_length}(\texttt{formal\_types},\texttt{args}) \xrightarrow{\text{type}} \texttt{TRUE} \mathbin{/\!/} \texttt{FALSE} \\ \texttt{a\_tys} := [(\_,\texttt{t}) \in \texttt{args} : \texttt{t}] \\ \texttt{i} \in \mathit{indices}(\texttt{f\_tys}) : \mathit{type\_clashes}(\texttt{tenv},\texttt{f\_tys}[\texttt{i}],\texttt{a\_tys}[\texttt{i}]) \xrightarrow{\text{type}} \texttt{TRUE} \mathbin{/\!/} \texttt{FALSE},\#\texttt{TE}\end{array}}{\mathit{has\_arg\_clash}(\texttt{tenv},\texttt{f\_tys},\texttt{args}) \xrightarrow{\text{type}} \overbrace{\texttt{TRUE}}^{\texttt{b}}}$$

### TypingRule.TypeClash

The helper function

$$\mathit{type\_clashes}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{t}}, \overbrace{\mathsf{ty}}^{\texttt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\texttt{TE}}$$

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 23.14, 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 23.14: 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 types, 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 $\texttt{t}_{1..k}$;
  - obtaining the structure of `s` in `tenv` yields a tuple type with element types $\texttt{s}_{1..n}$;
  - if $n \neq k$ the rule short-circuits with `b` = `FALSE`;
  - `b` is `TRUE` if and only if $\texttt{t}_i$ type-clashes with $\texttt{s}_i$, for all $i = 1..k$.

- All of the following apply (OTHERWISE_DIFFERENT_LABELS):
  - neither `s` subtypes `t` nor `t` subtypes `s`;
  - obtaining the structure of `t` in `tenv` yields `t_struct`;
  - obtaining the structure of `s` in `tenv` yields `s_struct`;
  - `s_struct` and `t_struct` have different AST labels;
  - `b` is `FALSE`;

- All of the following apply (OTHERWISE_STRUCTURED):
  - neither `s` subtypes `t` nor `t` subtypes `s`;
  - obtaining the structure of `t` in `tenv` yields `t_struct`;
  - obtaining the structure of `s` in `tenv` yields `s_struct`;
  - `s_struct` and `t_struct` have the same AST label;
  - `t_struct` (and thus `s_struct`) is a structured type;
  - `b` is `FALSE`;

**Formally**

SUBTYPE

$$\frac{(\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}) \vee (\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE})}{\mathit{type\_clashes}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{TRUE}}^{\mathsf{b}}}$$

SIMPLE

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{s\_struct}) \\ \mathit{ast\_label}(\mathtt{t\_struct}) \in \{\mathsf{T\_Bool}, \mathsf{T\_Int}, \mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Bits}\}\end{array}}{\mathit{type\_clashes}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{TRUE}}^{\mathsf{b}}}$$

T_ENUM

$$\frac{\mathit{is\_subtype}(\mathtt{tenv},\mathtt{s},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Enum}(\_,\mathtt{lis\_s}) \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{T\_Enum}(\_,\mathtt{lis\_t})}{\mathit{type\_clashes}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{lis\_s} = \mathtt{lis\_t}}^{\mathtt{b}}}$$

T_ARRAY

$$\frac{\mathit{is\_subtype}(\mathtt{tenv},\mathtt{s},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Array}(\_,\mathtt{ty\_t}) \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{T\_Array}(\_,\mathtt{ty\_s}) \qquad \mathit{type\_clashes}(\mathtt{tenv},\mathtt{ty\_t},\mathtt{ty\_s}) \xrightarrow{\mathsf{type}} \mathtt{b}}{\mathit{type\_clashes}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

T_TUPLE

$$\frac{\mathit{is\_subtype}(\mathtt{tenv},\mathtt{s},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Tuple}(\mathtt{t}_{1..k}) \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{T\_Tuple}(\mathtt{s}_{1..n}) \\ \mathit{bool\_transition}(n = k) \longrightarrow \mathtt{TRUE} \mathbin{/\!/} \mathtt{FALSE} \\ i = 1..k : \mathit{type\_clashes}(\mathtt{tenv},\mathtt{t}_i,\mathtt{s}_i) \xrightarrow{\mathsf{type}} \mathtt{b}_i \qquad \mathtt{b} := \bigwedge_{\mathtt{i}=1}^{k} \mathtt{b}_i}{\mathit{type\_clashes}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

OTHERWISE_DIFFERENT_LABELS

$$\frac{\mathit{is\_subtype}(\mathtt{tenv},\mathtt{s},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_struct} \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s\_struct} \qquad \mathit{ast\_label}(\mathtt{t\_struct}) \neq \mathit{ast\_label}(\mathtt{s\_struct})}{\mathit{type\_clashes}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

OTHERWISE_STRUCTURED

$$\frac{\mathit{is\_subtype}(\mathtt{tenv},\mathtt{s},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_struct} \\ \mathit{get\_structure}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s\_struct} \qquad \mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{s\_struct}) \\ \mathit{ast\_label}(\mathtt{t\_struct}) \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}, \mathsf{T\_Collection}\}}{\mathit{type\_clashes}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

### 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

$$\mathit{annotate\_exprs}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{expr}^*}^{\texttt{exprs}}) \longrightarrow \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{typed\_exprs}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 23.15, the list of expressions `10, 20.0` is annotated as

$$\left[\begin{array}{llll} & \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}}^{\mathsf{Constraint\_Exact}} & & \\ (\mathsf{T\_Int}(\mathsf{WellConstrained}([ & 10 & ])), & \mathsf{E\_Literal}(\mathsf{L\_Int}(10)), \quad \emptyset), \\ (\mathsf{T\_Real}, & & & \mathsf{E\_Literal}(\mathsf{L\_Real}(20)), \quad \emptyset) \end{array}\right]$$

Listing 23.15: 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**

EMPTY

$$\mathit{annotate\_exprs}(\mathsf{tenv}, \overbrace{[\,]}^{\mathsf{exprs}}) \xrightarrow{\mathsf{type}} \overbrace{[\,]}^{\mathsf{typed\_exprs}}$$

NON_EMPTY

$$\frac{\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{typed\_expr}\ \mathtt{//\ \#TE} \qquad \mathit{annotate\_exprs}(\mathsf{tenv}, \mathtt{exprs1}) \xrightarrow{\mathsf{type}} \mathtt{typed\_exprs1}\ \mathtt{//\ \#TE}}{\mathit{annotate\_exprs}(\mathsf{tenv}, \overbrace{[\mathsf{e}] + \mathtt{exprs1}}^{\mathsf{exprs}}) \xrightarrow{\mathsf{type}} \overbrace{[\mathtt{typed\_expr}] + \mathtt{typed\_exprs1}}^{\mathsf{typed\_exprs}}}$$

### 23.3.1 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 23.16.

Listing 23.16: 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 23.17 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 23.17: 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 23.18 is ill-typed, as an empty parameter list for a standard library function must be entirely omitted.

Listing 23.18: 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;
```

## 23.4 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

$$\mathit{eval\_call}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathsf{expr}^*}^{\texttt{params}}, \overbrace{\mathsf{expr}^*}^{\texttt{args}}) \times \mathsf{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms2}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\#T} \cup \overbrace{\mathsf{TDynError}}^{\#DE}$$

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 23.19, calling `non_throwing_func` terminates normally, while calling `throwing_func` terminates by throwing an exception.

Listing 23.19: 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 $\mathsf{Normal}(\texttt{vparams}, \texttt{env1})^{/\!/\#T,\#DE}$;
- evaluating each expression in `args` separately in `env1` as per SemanticsRule.EExprListM is $\mathsf{Normal}(\texttt{vargs}, \texttt{env2})^{/\!/\#T,\#DE}$;

- env2 consists of the static environment tenv and the dynamic environment denv2;
- applying *incr_statck_size* to $G^{\mathsf{denv2}}$ 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)$^{/\!/\#\mathrm{DE}}$;
    - applying the helper relation *read_value_from* to each element of the list vms yields the list vms2;
    - applying *decr_statck_size* to global and name yields genv2;
    - define new_env as the environment where the static environment is tenv and the dynamic environment consists of the dynamic global environment genv2 and the dynamic local environment is taken from denv2 (that is, we restore the local environment to that of the caller and drop the local environment of the callee).
    - the entire evaluation results in Normal(vms2, new_env).
  - All of the following apply (THROWING):
    - evaluating the subprogram named name with arguments vargs and parameters vparams in denv2′ is Throwing(v, env_throw)$^{/\!/\#\mathrm{DE}}$;
    - 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_statck_size* to global and name yields genv2;
    - define new_env as the environment where the static environment is tenv and the dynamic environment consists of the dynamic global environment genv2 and the dynamic local environment is taken from denv2 (that is, we restore the local environment to that of the caller and drop the local environment of the callee).
    - the entire evaluation results in Normal(vms2, new_env).

**Formally**

NORMAL

$$\frac{\begin{array}{c} \mathit{eval\_expr\_list\_m}(\texttt{env},\texttt{params}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vparams},\texttt{env1}) \;/\!/\; \#\text{T},\#\text{DE} \\ \mathit{eval\_expr\_list\_m}(\texttt{env1},\texttt{args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vargs},\texttt{env2}) \;/\!/\; \#\text{T},\#\text{DE} \\ \texttt{env2} \overset{\text{is}}{=} (\texttt{tenv},\texttt{denv2}) \qquad \mathit{incr\_statck\_size}(G^{\texttt{denv2}},\texttt{name}) \xrightarrow{\text{eval}} \texttt{genv} \\ \texttt{env2}' := (\texttt{tenv},(\texttt{genv},\emptyset_\lambda)) \\ \text{******** common prefix ********} \\ \mathit{eval\_subprogram}(\texttt{env2}',\texttt{name},\texttt{vparams},\texttt{vargs}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms},(\texttt{global},\_)) \;/\!/\; \#\text{DE} \\ \texttt{vms2} := [\texttt{m} \in \texttt{vms} : \mathit{read\_value\_from}(\texttt{vms})] \\ \mathit{decr\_statck\_size}(\texttt{global},\texttt{name}) \xrightarrow{\text{eval}} \texttt{genv2} \qquad \texttt{new\_env} := (\texttt{tenv},(\texttt{genv2},L^{\texttt{denv2}})) \end{array}}{\mathit{eval\_call}(\texttt{env},\texttt{name},\texttt{params},\texttt{args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms2},\texttt{new\_env})}$$

THROWING

$$\frac{\begin{array}{c} \mathit{eval\_expr\_list\_m}(\texttt{env},\texttt{args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vargs},\texttt{env1}) \;/\!/\; \#\text{T},\#\text{DE} \\ \mathit{eval\_expr\_list\_m}(\texttt{env1},\texttt{params}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vparams},\texttt{env2}) \;/\!/\; \#\text{T},\#\text{DE} \\ \texttt{env2} \overset{\text{is}}{=} (\texttt{tenv},\texttt{denv2}) \qquad \mathit{incr\_statck\_size}(G^{\texttt{denv2}},\texttt{name}) \xrightarrow{\text{eval}} \texttt{genv} \\ \texttt{env2}' := (\texttt{tenv},(\texttt{genv},\emptyset_\lambda)) \\ \text{******** common prefix ********} \\ \mathit{eval\_subprogram}(\texttt{env2}',\texttt{name},\texttt{vparams},\texttt{vargs}) \xrightarrow{\text{eval}} \text{Throwing}(\texttt{v},\texttt{env\_throw}) \;/\!/\; \#\text{DE} \\ \texttt{env\_throw} \overset{\text{is}}{=} (\texttt{tenv},(\texttt{global},\_)) \\ \mathit{decr\_statck\_size}(\texttt{global},\texttt{name}) \xrightarrow{\text{eval}} \texttt{genv2} \qquad \texttt{new\_env} := (\texttt{tenv},(\texttt{genv2},L^{\texttt{denv2}})) \end{array}}{\mathit{eval\_call}(\texttt{env},\texttt{name},\texttt{params},\texttt{args}) \xrightarrow{\text{eval}} \text{Throwing}(\texttt{v},\texttt{new\_env})}$$

**SemanticsRule.EvalSubprogram**

The relation

$$\mathit{eval\_subprogram}(\overbrace{\mathbb{E}}^{\texttt{env}},\overbrace{\mathbb{I}}^{\texttt{name}},\overbrace{(\mathbb{V}\times\mathcal{G})^*}^{\texttt{params}},\overbrace{(\mathbb{V}\times\mathcal{G})^*}^{\texttt{args}}) \times \text{Normal}(\overbrace{(\mathbb{V}^*,\mathcal{G})}^{\texttt{vs}},\overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\text{TThrowing}}^{\#\text{T}} \cup \overbrace{\text{TDynError}}^{\#\text{DE}}$$

evaluates the subprogram named `name` in the environment `env`, with `actual_args` the list of actual arguments, and `params` the list of arguments deduced by type equality. The result is either a normal configuration or an abnormal configuration. In the case of a normal configuration, it consists of a list of pairs with a value and an identifier, and a new environment `new_env`. The values represent values returned by the subprogram call and the identifiers are used in generating execution graph constraints for the returned values.

**Example: Subprogram Calls**

In Listing 23.20, the function `main` calls the function `foo` and the procedure `bar`.

Listing 23.20: Evaluating subprogram calls

```
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 $\texttt{g1}^{/\!/\texttt{\#DE}}$;
- define `arg_names` as the identifiers appearing as the first component of each pair in `arg_decls`;
- assigning the actual arguments with $((\texttt{env1}, \emptyset_{\texttt{g}}), \texttt{arg\_names}, \texttt{args})$ as per SemanticsRule.AssignArgs gives $(\texttt{env2}, \texttt{g2})$ 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 $((\texttt{env2}, \emptyset_{\texttt{g}}), \texttt{param\_names}, \texttt{param\_decls})$ as per SemanticsRule.AssignArgs gives $(\texttt{env3}, \texttt{g3})$ 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**

$$\frac{\begin{array}{c}\mathsf{env} \overset{\text{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \\ G^{\mathsf{tenv}}.\mathsf{subprograms}(\mathtt{name}) \overset{\text{is}}{=} \left\{\begin{array}{rcl} \text{body} & : & \mathtt{body}, \\ \text{args} & : & \mathtt{arg\_decls}, \\ \text{parameters} & : & \mathtt{param\_decls}, \\ \text{recurse\_limit} & : & \mathtt{recurse\_limit}, \\ & & \ldots \end{array}\right\} \\ \mathtt{env1} := (\mathsf{tenv}, (G^{\mathsf{denv}}, \emptyset_\lambda)) \\ \mathit{check\_recurse\_limit}(\mathtt{env1}, \mathtt{name}, \mathtt{recurse\_limit}) \xrightarrow{\text{eval}} \mathtt{g1} \;//\; \#DE \\ \mathtt{arg\_names} := [(\mathtt{x}, \_) \in \mathtt{arg\_decls} : \mathtt{x}] \\ \mathit{assign\_args}((\mathtt{env1}, \emptyset_{\mathtt{g}}), \mathtt{arg\_names}, \mathtt{actual\_args}) \xrightarrow{\text{eval}} (\mathtt{env2}, \mathtt{g2}) \\ \mathtt{param\_names} := [(\mathtt{x}, \_) \in \mathtt{params} : \mathtt{x}] \\ \mathit{assign\_args}((\mathtt{env2}, \mathtt{g2}), \mathtt{param\_names}, \mathtt{param\_decls}) \xrightarrow{\text{eval}} (\mathtt{env3}, \mathtt{g3}) \\ \mathit{eval\_stmt}(\mathtt{env3}, \mathtt{body}) \xrightarrow{\text{eval}} \mathtt{res} \;//\; \#T, \#DE \\ \mathit{match\_func\_res}(\mathtt{res}) \xrightarrow{\text{eval}} C \qquad \mathtt{new\_g} := \mathtt{g1} \xrightarrow{\mathtt{asl\_data}} \mathtt{g2} \xrightarrow{\mathtt{asl\_po}} \mathtt{g3} \end{array}}{\mathit{eval\_subprogram}(\mathtt{env}, \mathtt{name}, \mathtt{actual\_args}, \mathtt{params}) \xrightarrow{\text{eval}} C(\mathit{graph} \mapsto \mathtt{new\_g})}$$

**Comments**

It is not an error for execution of a procedure or setter to end without a return statement.

**SemanticsRule.CheckRecurseLimit**

The helper relation

$$\mathit{check\_recurse\_limit}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathsf{identifier}}^{\text{name}}, \overbrace{\mathsf{expr?}}^{\text{e\_limit\_opt}}) \longrightarrow \overbrace{\mathcal{G}}^{\text{g}} \cup \overbrace{\mathsf{TDynError}}^{\#DE}$$

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 23.21, the function `factorial` is specified without a limit expression, and evaluating `main` terminates normally.

Listing 23.21: A recursive function with no limit expression

```
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 23.22, the function `factorial` specifies the limit expression `11`, and evaluating `main` terminates normally.

Listing 23.22: A recursive function with a limit expression

```
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 23.23, the function `factorial` specifies the limit expression `10`, and evaluating `main` terminates with a dynamic error (`DE_LE`), since the limit is exceeded.

Listing 23.23: A recursive call exceeding the specified limit

```
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 $(\langle \texttt{limit} \rangle, \texttt{g})^{/\!/ \texttt{\#DE}}$;
    * view env as (tenv, denv);
    * applying *get_statck_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 $(\langle \texttt{limit} \rangle, \texttt{g})^{/\!/ \texttt{\#DE}}$;
    * view env as (tenv, denv);
    * applying *get_statck_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**

NONE

$$\frac{\mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} (\texttt{None}, \texttt{g}) \;/\!/\; \texttt{\#DE}}{\mathit{check\_recurse\_limit}(\texttt{env}, \texttt{name}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} \overbrace{\emptyset_{\texttt{g}}}^{\texttt{g}}}$$

SOME_OK

$$\frac{\begin{array}{c}\mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} (\langle \texttt{limit} \rangle, \texttt{g}) \;/\!/\; \texttt{\#DE} \\ \texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \\ \mathit{get\_statck\_size}(\text{denv}, \texttt{name}) \xrightarrow{\text{eval}} \texttt{stack\_size} \qquad \texttt{stack\_size} < \texttt{limit}\end{array}}{\mathit{check\_recurse\_limit}(\texttt{env}, \texttt{name}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} \texttt{g}}$$

LIMIT_EXCEEDED

$$\frac{\begin{array}{c}\mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} (\langle \texttt{limit} \rangle, \texttt{g}) \;/\!/\; \texttt{\#DE} \\ \texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \\ \mathit{get\_statck\_size}(\text{denv}, \texttt{name}) \xrightarrow{\text{eval}} \texttt{stack\_size} \qquad \texttt{stack\_size} \geq \texttt{limit}\end{array}}{\mathit{check\_recurse\_limit}(\texttt{env}, \texttt{name}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} \text{DynError}(\texttt{DE\_LE})}$$

**SemanticsRule.ReadValueFrom**

The helper relation

$$\mathit{read\_value\_from}(\mathbb{V}, \mathbb{I}) \bigtimes (\mathbb{V} \times \mathcal{G})$$

generates an execution graph for reading the given value to a variable given by the identifier, and pairs it with the given value.

### Prose

All of the following apply:

- reading the value `v` into the variable named `id` gives `new_g`;
- the result is $(\texttt{v}, \texttt{new\_g})$.

### Formally

$$\frac{\mathit{read\_identifier}(\texttt{v}, \texttt{id}) \xrightarrow{\text{eval}} \texttt{new\_g}}{\mathit{read\_value\_from}(\texttt{v}, \texttt{id}) \xrightarrow{\text{eval}} (\texttt{v}, \texttt{new\_g})}$$

### SemanticsRule.AssignArgs

The relation

$$\mathit{assign\_args}((\overbrace{\mathbb{E}}^{\texttt{env}} \times \overbrace{\mathcal{G}}^{\texttt{g1}}), \overbrace{\mathbb{I}^*}^{\texttt{ids}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{actuals}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

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 $(\texttt{new\_env}, \texttt{new\_g})$.

### Example: Assigning Arguments

In Listing 23.24, 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 23.24: 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**

EMPTY

$$\mathit{assign\_args}((\texttt{env}, \texttt{g1}), \overbrace{\texttt{ids}}^{[\,]}, \overbrace{[\,]}^{\texttt{actuals}}) \xrightarrow{\text{eval}} (\overbrace{\texttt{env}}^{\texttt{new\_env}}, \overbrace{\texttt{g1}}^{\texttt{new\_g}})$$

NON_EMPTY

$$\frac{\mathit{declare\_local\_identifier\_mm}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{env1}, \texttt{g2}) \qquad \mathit{assign\_args}((\texttt{env1}, \texttt{g1} \xrightarrow{\texttt{asl\_po}} \texttt{g2}), \texttt{ids1}, \texttt{actuals1}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g})}{\mathit{assign\_args}((\texttt{env}, \texttt{g1}), \overbrace{[\texttt{x}] + \texttt{ids1}}^{\texttt{ids}}, \overbrace{[\texttt{m}] + \texttt{actuals1}}^{\texttt{actuals}}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g})}$$

### SemanticsRule.MatchFuncRes

The helper relation

$$\mathit{match\_func\_res}(\mathsf{TContinuing} \cup \mathsf{TReturning}) \bigtimes \mathsf{Normal}(((\mathbb{I} \times \mathbb{V})^* \times \mathcal{G}), \mathbb{E})$$

converts continuing configurations and returning configurations into corresponding normal configurations that can be returned by a subprogram evaluation.

### Example: Converting Configurations Upon Subprogram Return

In Listing 23.25, 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 23.25: 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), env).
- All of the following apply (RETURNING):
  - the given configuration is Returning(xs, ret_env), which is the case of a function;
  - xs is the list $\mathtt{v}_i$, for $i = 1..k$;
  - define the list of fresh identifiers $\mathtt{id}_i$, for $i = 1..k$;
  - define vs to be $(\mathtt{v}_i, \mathtt{id}_i)$, for $i = 1..k$;
  - the result is Normal((vs, $\emptyset_{\mathsf{g}}$), ret_env).

### Formally

CONTINUING

$$\mathit{match\_func\_res}(\mathsf{Continuing}(\mathtt{g}, \mathtt{env})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{[\,]}, \mathtt{g}), \mathtt{env})$$

RETURNING

$$\frac{\mathtt{xs} \overset{\mathsf{is}}{=} [i = 1..k : \mathtt{v}_i] \qquad i = 1..k : \mathtt{id}_i \in \mathbb{I} \text{ is fresh} \qquad \mathtt{vs} := [i = 1..k : (\mathtt{v}_i, \mathtt{id}_i)]}{\mathit{match\_func\_res}(\mathsf{Returning}(\mathtt{xs}, \mathtt{ret\_env})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{vs}, \emptyset_{\mathsf{g}}), \mathtt{ret\_env})}$$

# 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.

## 24.1 Syntax

Subprogram declarations:

```
          decl ⟶ "func" ID params_opt func_args return_type func_body
                 | "func" ID params_opt func_args func_body
                 | "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_let_or_constant ID
         ↪ option(":" ty) "="expr ";"
       | "config" ID ":" ty "=" expr ";"
       | "var" ID ":" ty ";"
```

Pragmas:

```
decl ⟶ "pragma" ID clist0(expr) ";"
```

## 24.2 Abstract Syntax

```
decl ⟶ D_Func(func)
       | D_GlobalStorage(global_decl)
       | D_TypeDecl(ID, ty, (ID, field*)?)
       | D_Pragma(ID, expr*)
```

(with fields: field*; args: expr*)

The relation

$$build\_decl : \overbrace{\mathbb{PARSE}[\texttt{decl}]}^{\texttt{parsed\_node}} \times \overbrace{\texttt{decl}^*}^{\texttt{ast\_node}}$$

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

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

GLOBAL_PRAGMA

$$\frac{build\_clist[\texttt{expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{args\_ast}}{build\_decl(\overbrace{\texttt{decl}(\texttt{"pragma"}, \texttt{ID}(\texttt{id}), \texttt{args} : \texttt{clist0}(\texttt{expr}), \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{D\_Pragma}(\texttt{id}, \texttt{args\_ast})]}^{\texttt{ast\_node}}}$$

## 24.3 Typing Global Declarations

The function

$$\mathit{typecheck\_decl}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{decl}}^{\texttt{d}}) \longrightarrow (\overbrace{\mathsf{decl}}^{\texttt{new\_d}} \times \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}}) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 24.1 exemplifies various kinds of global declarations — types, global storage elements, and subprograms. All global declarations in Listing 24.1 are well-typed.

Listing 24.1: Typing global declarations

```
type MyRecord of record {
    high_bits: bits(32),
    low_bits: bits(32),
};

type MyException of exception {
    msg: string,
};

type MyCollection of collection {
    high_bits: bits(32),
    low_bits: bits(32),
};

var rec: MyRecord;
var exc: MyException;
var coll: MyCollection;

accessor Rec() <=> values: bits (64)
begin
    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`$^{/\!/\texttt{\#TE}}$;
  - 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`$^{/\!/\texttt{\#TE}}$;
  - define `ses_f_no_recursives` as `ses_f` with every recursive call side effect descriptor removed;
  - applying *add_subprogram* to `tenv1`, `new_f`.name, `new_f`, and `ses_f_no_recursives` 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'`$^{/\!/\texttt{\#TE}}$;
  - define `new_d` as the global storage declaration with description `gsd'`, that is, `D_GlobalStorage(gsd')`.
- All of the following apply (TYPE):
  - `d` is a type declaration with identifier `x`, type `ty`, and optional field initializers `s`, that is, `D_TypeDecl(x, ty, s)`;
  - declaring the type described by `(x, ty, s)` in `genv` as per TypingRule.DeclaredType yields the modified global static environment `new_genv`$^{/\!/\texttt{\#TE}}$;
  - define `new_d` as `d`.

### Formally

FUNC

$$\frac{\begin{array}{c}\mathit{annotate\_and\_declare\_func}(\texttt{genv}, \texttt{f}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{f1}, \texttt{ses\_func\_sig}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_subprogram}(\texttt{tenv1}, \texttt{f1}, \texttt{ses\_func\_sig}) \xrightarrow{\text{type}} (\texttt{new\_f}, \texttt{ses\_f}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses\_f\_no\_recursives} := \texttt{ses\_f} \setminus \{\texttt{s} \mid \texttt{config\_dom}(\texttt{s}) = \texttt{RecursiveCall}\} \\ \mathit{add\_subprogram}(\texttt{tenv1}, \texttt{new\_f.name}, \texttt{new\_f}, \texttt{ses\_f\_no\_recursives}) \xrightarrow{\text{type}} \texttt{new\_tenv}\end{array}}{\mathit{typecheck\_decl}(\texttt{genv}, \overbrace{\texttt{D\_Func(f)}}^{\texttt{d}}) \xrightarrow{\text{type}} (\overbrace{\texttt{D\_Func(new\_f)}}^{\texttt{new\_d}}, \overbrace{G^{\texttt{new\_tenv}}}^{\texttt{new\_genv}})}$$

GLOBAL_STORAGE

$$\frac{\mathit{declare\_global\_storage}(\texttt{genv}, \texttt{gsd}) \xrightarrow{\text{type}} (\texttt{new\_genv}, \texttt{gsd'}) \;/\!/\; \texttt{\#TE}}{\mathit{typecheck\_decl}(\texttt{genv}, \overbrace{\texttt{D\_GlobalStorage}(\texttt{gsd})}^{\texttt{d}}) \xrightarrow{\text{type}} (\overbrace{\texttt{D\_GlobalStorage}(\texttt{gsd'})}^{\texttt{new\_d}}, \texttt{new\_genv})}$$

TYPE

$$\frac{\mathit{declare\_type}(\texttt{genv}, \texttt{x}, \texttt{ty}, \texttt{s}) \xrightarrow{\text{type}} \texttt{new\_genv} \;/\!/\; \texttt{\#TE}}{\mathit{typecheck\_decl}(\texttt{genv}, \overbrace{\texttt{D\_TypeDecl}(\texttt{x}, \texttt{ty}, \texttt{s})}^{\texttt{d}}) \xrightarrow{\text{type}} (\overbrace{\texttt{d}}^{\texttt{new\_d}}, \texttt{new\_genv})}$$

### TypingRule.Subprogram

The function

$$\mathit{annotate\_subprogram}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{func}}^{\texttt{f}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_func\_sig}}) \longrightarrow (\overbrace{\mathsf{func}}^{\texttt{f'}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a subprogram `f` in an environment `tenv` and set of side effect descriptors `ses_func_sig`, resulting in an annotated subprogram `f'` and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

Note that the return type in `f` has already been annotated by *annotate_func_sig*.

### Example: Annotating Subprograms

Listing 24.2 shows an example of a well-typed procedure — `my_procedure` and an example of a well-typed function — `flip_bits`.

Listing 24.2: 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`.body in `tenv` as per TypingRule.Block yields (`new_body`, `ses_body`)$^{/\!/\texttt{\#TE}}$;
- One of the following applies:
  - All of the following apply (PROCEDURE):
    - `f`.return_type is `None`;
  - All of the following apply (FUNCTION):
    - `f`.return_type is not `None`;
    - applying *check_stmt_returns_or_throws* to `new_body` yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- `f'` is `f` with the subprogram body substituted with `new_body`;
- define `ses` as the union of `ses_func_sig` and `ses_body` with every instance of a local read side effect descriptor or a local write side effect descriptor removed.

### Formally

PROCEDURE

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\mathsf{tenv}, \texttt{f.body}) \xrightarrow{\mathsf{type}} (\texttt{new\_body}, \texttt{ses\_body}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \texttt{f.return\_type} = \texttt{None} \\ \text{******** common suffix ********} \\ \texttt{f'} := \mathit{subst\_record\_field}(\texttt{f}, \text{body}, \texttt{new\_body}) \\ \texttt{ses} := \texttt{ses\_func\_sig} \cup (\texttt{ses\_body} \setminus \{\texttt{s} \mid \texttt{config\_dom}(\texttt{s}) \in \{\mathsf{ReadLocal}, \mathsf{WriteLocal}\}\})\end{array}}{\mathit{annotate\_subprogram}(\mathsf{tenv}, \texttt{f}, \texttt{ses\_func\_sig}) \xrightarrow{\mathsf{type}} (\texttt{f'}, \texttt{ses})}$$

FUNCTION

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\mathsf{tenv}, \texttt{f.body}) \xrightarrow{\mathsf{type}} (\texttt{new\_body}, \texttt{ses\_body}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \texttt{f.return\_type} \neq \texttt{None} \\ \mathit{check\_stmt\_returns\_or\_throws}(\texttt{new\_body}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \text{******** common suffix ********} \\ \texttt{f'} := \mathit{subst\_record\_field}(\texttt{f}, \text{body}, \texttt{new\_body}) \\ \texttt{ses} := \texttt{ses\_func\_sig} \cup (\texttt{ses\_body} \setminus \{\texttt{s} \mid \texttt{config\_dom}(\texttt{s}) \in \{\mathsf{ReadLocal}, \mathsf{WriteLocal}\}\})\end{array}}{\mathit{annotate\_subprogram}(\mathsf{tenv}, \texttt{f}, \texttt{ses\_func\_sig}) \xrightarrow{\mathsf{type}} (\texttt{f'}, \texttt{ses})}$$

### TypingRule.CheckStmtReturnsOrThrows

The helper function

$$\mathit{check\_stmt\_returns\_or\_throws}(\overbrace{\mathsf{stmt}}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \{\mathtt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

checks whether all control-flow paths defined by the statement `s` terminate by either a statement returning a value, a `throw` statement, or the `Unreachable()` statement.

### Example: Ensuring All Terminating Paths Terminate Correctly

In Listing 24.3, 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 24.3: All terminating paths terminate correctly

```
type invalid_state of exception;

func all_terminating_paths_correct{N}(v: bits(N), flag: boolean) => bits(N)
begin
    if v != Zeros{N} then
        if flag then
            return Ones{N} XOR v;
        else
            Unreachable();
        end;
    else
        if flag then
            return v;
        else
            throw invalid_state{-};
        end;
    end;
end;
```

In Listing 24.4, 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 24.4: 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;
```

### Prose

All of the following apply:

- applying *control_flow_from_stmt* to `s` yields a control flow state `ctrl_flow`;
- checking that `ctrl_flow` is different from MayNotInterrupt yields $\texttt{TRUE}^{/\!/\texttt{TE\_BSPD}}$;
- the result is `TRUE`.

### Formally

$$\frac{\mathit{control\_flow\_from\_stmt}(\texttt{s}) \xrightarrow{\text{type}} \texttt{ctrl\_flow} \qquad \mathit{check}(\texttt{ctrl\_flow} \neq \mathsf{MayNotInterrupt}, \texttt{TE\_BSPD}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_stmt\_returns\_or\_throws}(\texttt{s}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

### TypingRule.ControlFlowFromStmt

We define control flow states as follows:

$$\mathsf{ControlFlow} := \{\mathsf{AssertedNotInterrupt}, \mathsf{Interrupt}, \mathsf{MayNotInterrupt}\}$$

control flow states are totally ordered via the relation $<_{\mathsf{CF}}$, defined as follows:

$$\mathsf{AssertedNotInterrupt} <_{\mathsf{CF}} \mathsf{Interrupt} <_{\mathsf{CF}} \mathsf{MayNotInterrupt} \ .$$

### Example: Determining the Control-flow State of Statements

In Listing 24.5, the function bodies of all functions terminate by either returning a value, throwing an exception, or executing the unreachable statement.

Listing 24.5: Determining the control-flow state of statements

```
func falls_through()
begin
    pass;
    var x : integer = 5;
    x = 7;
    assert x == 7;
    println(x);
    pragma require_positive x;
end;

func unreachable() => integer
begin
    Unreachable();
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 control-flow state is determined by the first statement.
    throw invalid_state{-};
    var x = 5;
end;

func sequencing2() => integer
begin
    // The control-flow state is determined by the second statement.
    pass;
    return 5;
end;

func conditional(flag: boolean) => integer
begin
    // The control-flow state is determined by "joining"
    // the control-flow states 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 2^128;
    return 0;
end;
```

```
func throwing_function() => integer
begin
    try
        return repeat_loop(1000);
    catch
        when invalid_state => return 0;
        otherwise => return 1;
    end;
end;

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

In Listing 24.6, the function body of `loop_forever` is ill-typed, since the conservative analysis of TypingRule.ControlFlowFromStmt 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 24.6: 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;
```

The helper function

$$\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{stmt}}^{\mathtt{s}}) \xrightarrow{\text{type}} \overbrace{\mathsf{ControlFlow}}^{\mathtt{ctrl\_flow}}$$

statically analyzes the statement `s` and determines the control flow state `ctrl_flow` to be one of the following:

**AssertedNotInterrupt** evaluating `s` in any environment will evaluate `Unreachable()`;

**Interrupt** evaluating `s` in any environment will always end by either evaluating a return statement with an expression, or evaluating a `throw` statement;

**MayNotInterrupt** evaluating `s` in any environment might not end by evaluating a `return` statement with an expression or a `throw` statement.

**Prose**

One of the following applies:

- All of the following apply (FALLS_THROUGH):
  - the AST label of `s` is S_Pass, S_Decl, S_Assign, S_Assert, S_Call, S_Print or S_Pragma;
  - `ctrl_flow` is MayNotInterrupt;
- All of the following apply (UNREACHABLE):
  - `s` is S_Unreachable;
  - `ctrl_flow` is AssertedNotInterrupt;
- All of the following apply (RETURN_THROW):
  - the AST label of `s` is either S_Return or S_Throw;
  - `ctrl_flow` is Interrupt;
- All of the following apply (S_SEQ):
  - `s` is the sequencing statement for `s1` and `s2`;
  - applying *control_flow_from_stmt* to `s1` yields `ctrl_flow1`;
  - applying *control_flow_from_stmt* to `s2` yields `ctrl_flow2`;
  - applying *control_flow_seq* to `ctrl_flow1` and `ctrl_flow2` yields `ctrl_flow`.
- All of the following apply (S_COND):
  - `s` is the conditional statement for an expression and statements `s1` and `s2`;
  - applying *control_flow_from_stmt* to `s1` yields `ctrl_flow1`;
  - applying *control_flow_from_stmt* to `s2` yields `ctrl_flow2`;
  - applying *control_flow_join* to `ctrl_flow1` and `ctrl_flow2` yields `ctrl_flow`.
- All of the following apply (S_WHILE_FOR):
  - `s` is either a while statement or a for statement;
  - define `ctrl_flow` as MayNotInterrupt.
- All of the following apply (S_REPEAT):
  - `s` is the repeat statement with the body statement `body`;
  - applying *control_flow_from_stmt* to `body` yields `ctrl_flow`.
- 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_opt`;

* applying *control_flow_from_stmt* to `body` yields `res0`;
* define `res1` as the application of *control_flow_join* to *control_flow_from_stmt*(`o`) and `res0`, if `otherwise_opt` = ⟨`o`⟩, and `res0`, otherwise;
* for each catcher in `catchers` associated with a statement `s`, applying *control_flow_from_stmt* to `s` yields $\texttt{cf}_\texttt{s}$;
* define `ctrl_flow` as the application of *control_flow_join* to `res1`, and $\texttt{cf}_\texttt{s}$, for each catcher in `catchers` associated with a statement `s`.

**Formally**

FALLS_THROUGH

$$\frac{\mathit{ast\_label}(\texttt{s}) \in \{\mathsf{S\_Pass}, \mathsf{S\_Decl}, \mathsf{S\_Assign}, \mathsf{S\_Assert}, \mathsf{S\_Call}, \mathsf{S\_Print}, \mathsf{S\_Pragma}\}}{\mathit{control\_flow\_from\_stmt}(\texttt{s}) \xrightarrow{\text{type}} \overbrace{\mathsf{MayNotInterrupt}}^{\texttt{ctrl\_flow}}}$$

UNREACHABLE

$$\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Unreachable}}^{\texttt{s}}) \xrightarrow{\text{type}} \overbrace{\mathsf{AssertedNotInterrupt}}^{\texttt{ctrl\_flow}}$$

RETURN_THROW

$$\frac{\mathit{ast\_label}(\texttt{s}) \in \{\mathsf{S\_Return}, \mathsf{S\_Throw}\}}{\mathit{control\_flow\_from\_stmt}(\texttt{s}) \xrightarrow{\text{type}} \overbrace{\mathsf{Interrupt}}^{\texttt{ctrl\_flow}}}$$

S_SEQ

$$\frac{\begin{array}{c}\mathit{control\_flow\_from\_stmt}(\texttt{s1}) \xrightarrow{\text{type}} \texttt{ctrl\_flow1} \\ \mathit{control\_flow\_from\_stmt}(\texttt{s2}) \xrightarrow{\text{type}} \texttt{ctrl\_flow2} \\ \mathit{control\_flow\_seq}(\texttt{ctrl\_flow1}, \texttt{ctrl\_flow2}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}\end{array}}{\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Seq}(\texttt{s1}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}$$

S_COND

$$\frac{\begin{array}{c}\mathit{control\_flow\_from\_stmt}(\texttt{s1}) \xrightarrow{\text{type}} \texttt{ctrl\_flow1} \\ \mathit{control\_flow\_from\_stmt}(\texttt{s2}) \xrightarrow{\text{type}} \texttt{ctrl\_flow2} \\ \mathit{control\_flow\_join}(\{\texttt{ctrl\_flow1}, \texttt{ctrl\_flow2}\}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}\end{array}}{\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Cond}(\_, \texttt{s1}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}$$

The reasoning for the next case is as follows:

- Conservatively, that is, without attempting to reason about the condition expressions, a S_While loop, and a S_For are like a conditional statement that either executes S_Pass (when the loop condition does not hold) or executes the body and then loops back.
- The control flow state for S_Pass is MayNotInterrupt.
- The overall control flow state is then *control_flow_join*({MayNotInterrupt, _}), which is always MayNotInterrupt, since MayNotInterrupt is the maximal element, with respect to $<_{\mathsf{CF}}$.

S_WHILE_FOR

$$\frac{\mathit{ast\_label}(\mathtt{s}) \in \{\mathsf{S\_While}, \mathsf{S\_For}\}}{\mathit{control\_flow\_from\_stmt}(\mathtt{s}) \xrightarrow{\text{type}} \overbrace{\mathsf{MayNotInterrupt}}^{\mathtt{ctrl\_flow}}}$$

The reasoning for the next case is as follows:

- A statement S_Repeat(body, v_cond, _) is equivalent (ignoring limits) to S_Seq(body, S_While(v_cond, _, body));
- Let us denote *control_flow_from_stmt*(body) $\xrightarrow{\text{type}}$ ctrl_flow. Observe that by case S_WHILE_FOR above, we have that *control_flow_from_stmt*(S_While(v_cond, _, body)) $\xrightarrow{\text{type}}$ MayNotInterrupt holds;
- Applying the rule for the S_SEQ case above, requires applying *control_flow_seq* to ctrl_flow and MayNotInterrupt, which always yields ctrl_flow (to see this, consider the case ctrl_flow = MayNotInterrupt and the case ctrl_flow ≠ MayNotInterrupt).

S_REPEAT

$$\frac{\mathit{control\_flow\_from\_stmt}(\mathtt{body}) \xrightarrow{\text{type}} \mathtt{ctrl\_flow}}{\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Repeat}(\mathtt{body}, \_, \_)}^{\mathtt{s}}) \xrightarrow{\text{type}} \mathtt{ctrl\_flow}}$$

The rule for S_TRY conservatively approximates the results from all control flows passing through the statement by returning the maximal *controlflowstate*, with respect to $<_{\mathsf{CF}}$, computed for each control flow path.

S_TRY

$$\frac{\begin{array}{c}\mathit{control\_flow\_from\_stmt}(\mathtt{body}) \xrightarrow{\text{type}} \mathtt{res0} \\ \mathtt{res1} := \begin{cases} \mathit{control\_flow\_join}(\{\mathit{control\_flow\_from\_stmt}(\mathtt{o}), \mathtt{res0}\}) & \text{if } \mathtt{otherwise\_opt} = \langle \mathtt{o} \rangle \\ \mathtt{res0} & \text{otherwise} \end{cases} \\ (\_, \_, \mathtt{s}) \in \mathtt{catchers} : \mathit{control\_flow\_from\_stmt}(\mathtt{s}) \xrightarrow{\text{type}} \mathtt{cf_s} \\ \mathtt{ctrl\_flow} := \mathit{control\_flow\_join}(\{(\_, \_, \mathtt{s}) \in \mathtt{catchers} : \mathtt{cf_s}\} \cup \{\mathtt{res1}\}) \end{array}}{\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Try}(\mathtt{body}, \mathtt{catchers}, \mathtt{otherwise\_opt})}^{\mathtt{s}}) \xrightarrow{\text{type}} \mathtt{ctrl\_flow}}$$

### TypingRule.ControlFlowSeq

The helper function

$$\mathit{control\_flow\_seq}(\overbrace{\mathsf{ControlFlow}}^{\texttt{t1}}, \overbrace{\mathsf{ControlFlow}}^{\texttt{t2}}) \longrightarrow \overbrace{\mathsf{ControlFlow}}^{\texttt{ctrl\_flow}}$$

combines two control flow states considering them as part of a control flow path where the analysis of the path prefix yields `t1` and the analysis of the path suffix yields `t2`, into a single control flow state `ctrl_flow`.

### Example: Determining the Control-flow State of a Sequence of Statements

In Listing 24.5, the function body of `sequencing1` is determined to have the control-flow state Interrupt by the statement `throw invalid_state{}`, effectively ignoring the succeeding statement `var x = 5;`

On the other hand, in the function body of `sequencing2`, analysis of the pass statement yields the control-flow state MayNotInterrupt, but analysis of the statement `return 5;` yields control-flow state Interrupt, which is why the analysis of the sequence of these two statements yields Interrupt.

### Prose

Define `ctrl_flow` as `t2` if `t1` is MayNotInterrupt and `t1`, otherwise.

### Formally

$$\frac{\texttt{ctrl\_flow} := \mathsf{choice}(\texttt{t1} = \mathsf{MayNotInterrupt}, \texttt{t2}, \texttt{t1})}{\mathit{control\_flow\_seq}(\texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}$$

### TypingRule.ControlFlowJoin

The helper function

$$\mathit{control\_flow\_join}(\overbrace{\mathcal{P}(\mathsf{ControlFlow})}^{\texttt{s}}) \longrightarrow \overbrace{\mathsf{ControlFlow}}^{\texttt{ctrl\_flow}}$$

returns the maximal element in the set of control flow states `s`, with respect to $<_{\mathsf{CF}}$ in `ctrl_flow`.

### Example: Determining the Control-flow State of a Conditional Statement

In Listing 24.5, the function body for `conditional` is well-typed, since the control-flow state analysis for both statements `return 5;` and `throw invalid_state{};` yields the control-flow state Interrupt.

### Prose

define `ctrl_flow` as the maximal element in the set of control flow states `s`, with respect to $<_{\mathsf{CF}}$.

### Formally

$$\mathit{control\_flow\_join}(\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\max_{<_{\mathsf{CF}}}(\mathtt{s})}^{\mathtt{ctrl\_flow}}$$

# 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.

## 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 time frame of `config` storage elements is Execution, so their values are not relied upon by typechecking.
- Values of `config` storage elements must have time frame less than or equal to Constant, 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 19.1 are illegal, as they discard their declared storage elements.

Listing 25.1: Illegal global storage declarations that discard storage elements

```
let - = 42;
var - = TRUE;
constant - = "abc";
config - = 1.0;
```

```
decl ⟶ global_let_or_constant ID option(":" ty)
         ↪ "=" expr ";"
       | "config" ID ":" ty "=" expr ";"
       | "var" ID option(":" ty) "="
         ↪ expr ";"
       | "var" ID ":" ty ";"
       | "pragma" ID clist0(expr) ";"

global_let_or_constant ⟶ "let" | "constant"
```

## 25.3 Abstract Syntax

$$\text{decl} \longrightarrow \text{D\_GlobalStorage}(\text{global\_decl})$$

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

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

### ASTRule.GlobalStorageDecl

GLOBAL_STORAGE_LET_OR_CONSTANT

$$\frac{\begin{array}{c}\mathit{build\_global\_decl\_let\_or\_constant}(\texttt{keyword}) \xrightarrow{\text{ast}} \overline{\texttt{keyword}} \\ \mathit{build\_option}[\mathit{build\_as\_ty}](\texttt{ty}) \xrightarrow{\text{ast}} \texttt{ty'} \\ \mathit{build\_expr}(\texttt{initial\_value}) \xrightarrow{\text{type}} \overline{\texttt{initial\_value}}\end{array}}{\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{r}\texttt{keyword} : \texttt{global\_let\_or\_constant}, \\ \hookrightarrow \textbf{ID}(\texttt{name}), \\ \hookrightarrow \texttt{ty} : \texttt{option}(\texttt{as\_ty}), \texttt{"="}, \texttt{initial\_value} : \texttt{expr}, \texttt{";"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\left[\texttt{D\_GlobalStorage}\left(\left\{\begin{array}{rcl}\text{keyword} & : & \overline{\texttt{keyword}}, \\ \text{name} & : & \overline{\texttt{name}}, \\ \text{ty} & : & \texttt{ty'}, \\ \text{initial\_value} & : & \overline{\texttt{initial\_value}}\end{array}\right\}\right)\right]}^{\texttt{ast\_node}}}$$

GLOBAL_STORAGE_VAR

$$\frac{\begin{array}{c}\mathit{build\_option}[\mathit{build\_as\_ty}](\texttt{ty}) \xrightarrow{\text{ast}} \texttt{ty'} \\ \mathit{build\_expr}(\texttt{initial\_value}) \xrightarrow{\text{type}} \overline{\texttt{initial\_value}}\end{array}}{\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{r}\texttt{"var"}, \textbf{ID}(\texttt{name}), \\ \hookrightarrow \texttt{ty} : \texttt{option}(\texttt{as\_ty}), \texttt{"="}, \texttt{initial\_value} : \texttt{expr}, \texttt{";"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\left[\texttt{D\_GlobalStorage}\left(\left\{\begin{array}{rcl}\text{keyword} & : & \texttt{GDK\_Var}, \\ \text{name} & : & \overline{\texttt{name}}, \\ \text{ty} & : & \texttt{ty'}, \\ \text{initial\_value} & : & \overline{\texttt{initial\_value}}\end{array}\right\}\right)\right]}^{\texttt{ast\_node}}}$$

GLOBAL_UNINIT_VAR

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"var"}, \textbf{ID}(\texttt{name}), \texttt{as\_ty}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\left[\texttt{D\_GlobalStorage}\left(\left\{\begin{array}{rcl}\text{keyword} & : & \texttt{GDK\_Var}, \\ \text{name} & : & \texttt{name}, \\ \text{ty} & : & \langle\overline{\texttt{as\_ty}}\rangle, \\ \text{initial\_value} & : & \texttt{None}\end{array}\right\}\right)\right]}^{\texttt{ast\_node}}$$

GLOBAL_STORAGE_CONFIG

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"config"}, \textbf{ID}(\texttt{name}), \texttt{":"}, \texttt{ty}, \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_GlobalStorage}(\{\text{keyword} : \texttt{GDK\_Config}, \text{name} : \texttt{name}, \text{ty} : \langle\overline{\texttt{t}}\rangle, \text{initial\_value} : \langle\overline{\texttt{expr}}\rangle\})}^{\texttt{ast\_node}}$$

**ASTRule.GlobalDeclKeyword**

The function

$$\mathit{build\_global\_decl\_let\_or\_constant}(\overbrace{\mathbb{PARSE}[\texttt{global\_let\_or\_constant}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{global\_decl\_keyword}}^{\texttt{ast\_node}}$$

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

LET

$$\mathit{build\_global\_decl\_let\_or\_constant}(\overbrace{\texttt{global\_let\_or\_constant}(\texttt{"let"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{GDK\_Let}}^{\texttt{ast\_node}}$$

CONSTANT

$$\mathit{build\_global\_decl\_let\_or\_constant}(\overbrace{\texttt{global\_let\_or\_constant}(\texttt{"constant"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{GDK\_Constant}}^{\texttt{ast\_node}}$$

## 25.4 Typing

We also define the following helper rules:

- TypingRule.DeclareGlobalStorage
- TypingRule.AnnotateTyOptInitialValue
- TypingRule.AddGlobalStorage

**TypingRule.DeclareGlobalStorage**

The function

$$\mathit{declare\_global\_storage}(\overbrace{\mathbb{GSE}}^{\texttt{tenv}}, \overbrace{\mathsf{global\_decl}}^{\texttt{gsd}}) \longrightarrow (\overbrace{\mathbb{GSE}}^{\texttt{new\_genv}}, \overbrace{\mathsf{global\_decl}}^{\texttt{new\_gsd}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the global storage declaration `gsd` in the global static environment `genv`, yielding a modified global static environment `new_genv` and annotated global storage declaration `new_gsd`. Otherwise, the result is a type error.

**Example: Declaring Global Storage**

Listing 25.2 shows examples of well-typed global storage declarations and ill-typed global storage declarations in comments, focused on `config` storage elements.

Listing 25.2: 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 25.3 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 25.3: 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 25.4 is illegal since `config` storage elements must have an initializing expression.

Listing 25.4: Uninitialized configuration storage declaration

```
// Illegal (parse error): 'config' storage must be initialized.
config uninitialized_config : integer;
```

The specifications in Listing 25.5 and Listing 25.6 are ill-typed since `config` storage elements have the Constant time frame, which requires that all expressions used in its type annotation and initializing expression also have the Constant time frame, whereas `x` has the Execution time frame.

Listing 25.5: Ill-typed configuration storage declaration

```
let x = 5;
// Illegal: initializing expression must be constant-time.
config c : integer{1..5} = x;
```

Listing 25.6: 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 `target_time_frame` as Constant if `keyword` is GDK_Constant or GDK_Config, and Execution otherwise;
- applying *annotate_ty_opt_initial_value* to `keyword`, `target_time_frame`, `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 `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**

$$\frac{\begin{array}{c}\texttt{gsd} \overset{\text{is}}{=} \{\text{keyword} : \texttt{keyword}, \text{initial\_value} : \texttt{initial\_value}, \text{ty} : \texttt{ty\_opt}, \text{name} : \texttt{name}\} \\ \mathit{check\_var\_not\_in\_genv}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv} \\ \texttt{target\_time\_frame} := \\ \text{choice}(\texttt{keyword} \in \{\text{GDK\_Constant}, \text{GDK\_Config}\}, \text{Constant}, \text{Execution}) \\ \mathit{annotate\_ty\_opt\_initial\_value}(\texttt{tenv}, \texttt{keyword}, \texttt{target\_time\_frame}, \texttt{ty\_opt}, \texttt{initial\_value}) \\ \xrightarrow{\text{type}} (\texttt{typed\_initial\_value}, \texttt{ty\_opt'}, \texttt{declared\_t}) \;/\!/\; \texttt{\#TE} \\ \mathit{add\_global\_storage}(\texttt{genv}, \texttt{name}, \texttt{keyword}, \texttt{declared\_t}) \xrightarrow{\text{type}} \texttt{genv1} \;/\!/\; \texttt{\#TE} \\ \mathit{with\_empty\_local}(\texttt{genv1}) \xrightarrow{\text{type}} \texttt{tenv1} \\ \texttt{typed\_initial\_value} \overset{\text{is}}{=} (\_, \texttt{initial\_value'}, \texttt{ses\_initial\_value}) \\ \mathit{update\_global\_storage}\begin{pmatrix}\texttt{tenv1},\\ \texttt{name},\\ \texttt{keyword},\\ \texttt{typed\_initial\_value},\\ \texttt{ses\_initial\_value}\end{pmatrix} \xrightarrow{\text{type}} \texttt{tenv2} \;/\!/\; \texttt{\#TE} \\ \texttt{new\_gsd} := \left\{\begin{array}{rcl}\text{keyword} & : & \texttt{keyword},\\ \text{initial\_value} & : & \langle\texttt{typed\_initial\_value}\rangle,\\ \text{ty} & : & \texttt{ty\_opt'},\\ \text{name} & : & \texttt{name}\end{array}\right\}\end{array}}{\mathit{declare\_global\_storage}(\texttt{genv}, \texttt{gsd}) \xrightarrow{\text{type}} (\overbrace{G^{\texttt{tenv2}}}^{\texttt{new\_genv}}, \texttt{new\_gsd})}$$

**TypingRule.AnnotateTyOptInitialValue**

The helper function

$$\mathit{annotate\_ty\_opt\_initial\_value}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{global\_decl\_keyword}}^{\texttt{gdk}}, \overbrace{\text{TimeFrame}}^{\texttt{target\_time\_frame}}, \overbrace{\langle\text{ty}\rangle}^{\texttt{ty\_opt'}}, \overbrace{\langle\text{expr}\rangle}^{\texttt{initial\_value}}) \longrightarrow (\overbrace{(\text{expr} \times \text{ty} \times \mathcal{P}(\text{TSideEffect}))}^{\texttt{typed\_initial\_value}} \times \overbrace{\langle\text{ty}\rangle}^{\texttt{ty\_opt'}} \times \overbrace{\text{ty}}^{\texttt{declared\_t}}) \cup \overbrace{\text{TTypeError}}^{\texttt{\#TE}}$$

is used in the context of a declaration of a global storage element with optional type annotation `ty_opt'` and optional initializing expression `initial_value`, in the static environment `tenv`. It determines `typed_initial_value`, which consists of an expression, a type, and a set of side effect descriptors, the annotation of the type in `ty_opt'` (in case there is a type), and the type that should be associated with the storage element `declared_t`.

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 $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})^{//\#\texttt{TE}}$;
  * annotating the type `t` in `tenv` yields $(\texttt{t'}, \texttt{ses\_t})^{//\#\texttt{TE}}$;
  * define `typed_e` as $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})$;
  * checking that `t_e` type-satisfies `t'` in `tenv` yields $\texttt{TRUE}^{//\#\texttt{TE}}$;
  * checking that all time frames in `ses_t` and `ses_e` are less than or equal to `target_time_frame` via *ses_is_before* yields $\texttt{TRUE}^{//\#\texttt{TE}}$;
  * define `typed_initial_value` as `typed_e`;
  * define `ty_opt'` as the singleton set for `t'`;
  * define `declared_t` as `t'`;
- All of the following apply (SOME_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 $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})^{//\#\texttt{TE}}$;
  * determining the structure of `t_e` in `tenv` yields $\texttt{t\_e'}^{//\#\texttt{TE}}$;
  * propagating integer constraints from `t_e'` to `t` using *inherit_integer_constraints* yields $\texttt{t''}^{//\#\texttt{TE}}$;
  * annotating the type `t''` in `tenv` yields $(\texttt{t'}, \texttt{ses\_t})^{//\#\texttt{TE}}$;
  * define `typed_e` as $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})$;
  * checking that `t_e` type-satisfies `t'` in `tenv` yields $\texttt{TRUE}^{//\#\texttt{TE}}$;
  * checking that all time frames in `ses_t` and `ses_e` are less than or equal to `target_time_frame` via *ses_is_before* yields $\texttt{TRUE}^{//\#\texttt{TE}}$;
  * define `typed_initial_value` as `typed_e`;
  * define `ty_opt'` as the singleton set for `t'`;
  * define `declared_t` as `t'`;
- All of the following apply (SOME_NONE):
  * `ty_opt'` is the singleton set for the type `t`;
  * `initial_value` is None;
  * annotating the type `t` in `tenv` yields $(\texttt{t'}, \texttt{ses\_t})^{//\#\texttt{TE}}$;

  - checking that all time frames in `ses_t` are less than or equal to `target_time_frame` via *ses_is_before* yields `TRUE`$^{\#TE}$;
  - obtaining the base value of `t'` in `tenv` yields `e'`$^{\#TE}$;
  - define `typed_initial_value` as $(\texttt{t'}, \texttt{e'}, \emptyset)$;
  - define `ty_opt'` as the singleton set for `t'`;
  - define `declared_t` as `t'`;

- All of the following apply (NONE_SOME):
  - `ty_opt'` is `None`;
  - `initial_value` is the singleton set for the expression `e`;
  - annotating the expression `e` in `tenv` yields $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})^{\#TE}$;
  - determining whether `t_e` has been computed with no precision loss via *check_no_precision_loss*() yields `TRUE`$^{\#TE}$;
  - checking that all time frames in `ses_e` are less than or equal to `target_time_frame` via *ses_is_before* yields `TRUE`$^{\#TE}$;
  - define `typed_e` as $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})$;
  - define `typed_initial_value` as `typed_e`;
  - define `ty_opt'` as `None`;
  - define `declared_t` as `t_e`;

The case where both `ty_opt` and `initial_value` are `None` is considered a syntax error.

### Formally

SOME_SOME_CONFIG

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} (\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e}) \;/\!/\; \#TE \\ \mathit{annotate\_type}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\text{type}} (\texttt{t'}, \texttt{ses\_t}) \;/\!/\; \#TE \\ \texttt{typed\_e} := (\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e}) \qquad \mathit{checked\_typesat}(\mathsf{tenv}, \texttt{t\_e}, \texttt{t'}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \#TE \\ \mathit{check}(\mathit{ses\_is\_before}(\texttt{ses\_t} \cup \texttt{ses\_e}, \texttt{target\_time\_frame}), \texttt{TE\_SEV}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \#TE \end{array}}{\begin{array}{l} \mathit{annotate\_ty\_opt\_initial\_value}(\mathsf{tenv}, \overbrace{\mathsf{GDK\_Config}}^{\text{gdk}}, \texttt{target\_time\_frame}, \overbrace{\langle \mathsf{t} \rangle}^{\texttt{ty\_opt'}}, \overbrace{\langle \mathsf{e} \rangle}^{\texttt{initial\_value}}) \\ \qquad \xrightarrow{\text{type}} (\overbrace{\texttt{typed\_e}}^{\texttt{typed\_initial\_value}}, \overbrace{\langle \texttt{t'} \rangle}^{\texttt{ty\_opt'}}, \overbrace{\texttt{t'}}^{\texttt{declared\_t}}) \end{array}}$$

SOME_SOME

$$\frac{\begin{array}{c}\texttt{gdk} \neq \text{GDK\_Config} \qquad \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e}) \;/\!/\; \#\text{TE} \\ \mathit{get\_structure}(\texttt{tenv}, \texttt{t\_e}) \xrightarrow{\text{type}} \texttt{t\_e'} \;/\!/\; \#\text{TE} \\ \mathit{inherit\_integer\_constraints}(\texttt{t}, \texttt{t\_e'}) \xrightarrow{\text{type}} \texttt{t''} \;/\!/\; \#\text{TE} \\ \mathit{annotate\_type}(\texttt{tenv}, \texttt{t''}) \xrightarrow{\text{type}} (\texttt{t'}, \texttt{ses\_t}) \;/\!/\; \#\text{TE} \\ \texttt{typed\_e} := (\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e}) \qquad \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_e}, \texttt{t'}) \xrightarrow{\text{type}} \text{TRUE} \;/\!/\; \#\text{TE} \\ \mathit{check}(\mathit{ses\_is\_before}(\texttt{ses\_t} \cup \texttt{ses\_e}, \texttt{target\_time\_frame}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \;/\!/\; \#\text{TE}\end{array}}{\mathit{annotate\_ty\_opt\_initial\_value}(\texttt{tenv}, \texttt{gdk}, \texttt{target\_time\_frame}, \overbrace{\langle \texttt{t} \rangle}^{\texttt{ty\_opt'}}, \overbrace{\langle \texttt{e} \rangle}^{\texttt{initial\_value}}) \xrightarrow{\text{type}} (\overbrace{\texttt{typed\_e}}^{\texttt{typed\_initial\_value}}, \overbrace{\langle \texttt{t'} \rangle}^{\texttt{ty\_opt'}}, \overbrace{\texttt{t'}}^{\texttt{declared\_t}})}$$

SOME_NONE

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} (\texttt{t'}, \texttt{ses\_t}) \;/\!/\; \#\text{TE} \\ \mathit{check}(\mathit{ses\_is\_before}(\texttt{ses\_t}, \texttt{target\_time\_frame}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \;/\!/\; \#\text{TE} \\ \mathit{base\_value}(\texttt{tenv}, \texttt{t'}) \xrightarrow{\text{type}} \texttt{e'} \;/\!/\; \#\text{TE} \\ \texttt{typed\_initial\_value} := (\texttt{t'}, \texttt{e'}, \emptyset)\end{array}}{\mathit{annotate\_ty\_opt\_initial\_value}(\texttt{tenv}, \texttt{gdk}, \texttt{target\_time\_frame}, \overbrace{\langle \texttt{t} \rangle}^{\texttt{ty\_opt'}}, \overbrace{\text{None}}^{\texttt{initial\_value}}) \xrightarrow{\text{type}} (\texttt{typed\_initial\_value}, \overbrace{\langle \texttt{t'} \rangle}^{\texttt{ty\_opt'}}, \overbrace{\texttt{t'}}^{\texttt{declared\_t}})}$$

NONE_SOME

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e}) \;/\!/\; \#\text{TE} \\ \mathit{check\_no\_precision\_loss}(\texttt{t\_e}) \xrightarrow{\text{type}} \text{TRUE} \;/\!/\; \#\text{TE} \\ \texttt{typed\_e} := (\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e}) \\ \mathit{check}(\mathit{ses\_is\_before}(\texttt{ses\_e}, \texttt{target\_time\_frame}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \;/\!/\; \#\text{TE}\end{array}}{\mathit{annotate\_ty\_opt\_initial\_value}(\texttt{tenv}, \texttt{gdk}, \texttt{target\_time\_frame}, \overbrace{\text{None}}^{\texttt{ty\_opt'}}, \overbrace{\langle \texttt{e} \rangle}^{\texttt{initial\_value}}) \xrightarrow{\text{type}} (\overbrace{\texttt{typed\_e}}^{\texttt{typed\_initial\_value}}, \overbrace{\text{None}}^{\texttt{ty\_opt'}}, \overbrace{\texttt{t\_e}}^{\texttt{declared\_t}})}$$

### TypingRule.UpdateGlobalStorage

The helper function

$$\mathit{update\_global\_storage}\left(\begin{array}{c}\overbrace{\mathbb{SE}}^{\texttt{tenv}},\\ \overbrace{\mathsf{identifier}}^{\texttt{name}},\\ \overbrace{\mathsf{global\_decl\_keyword}}^{\texttt{gdk}},\\ \overbrace{(\mathsf{ty}\times\mathsf{expr}\times\mathcal{P}(\mathsf{TSideEffect}))}^{\texttt{typed\_initial\_value}}\end{array}\right)\longrightarrow\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

updates the static environment `tenv` for the global storage element named `name` with global declaration keyword `gdk`, and a tuple (obtained via TypingRule.AnnotateTyOptInitialValue) `typed_initial_value`, which consists a type for the initializing value, the annotated initializing expression, and the inferred set of side effect descriptors for the initializing value. The result is the updated static environment `new_tenv`. Otherwise, the result is a type error.This helper function is applied following *add_global_storage*(`tenv`, `name`, `gdk`, `t`) where `t` is the type associated with `name`.

### Example: Updating the Global Storage for a Constant

The specification in Listing 25.7 binds `y` to L_Int(5040) in the constant_values map of the global static environment.

Listing 25.7: Updating the global storage for a constant

```
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 25.8 shows well-typed global storage `config` elements.

Listing 25.8: 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 25.9 is ill-typed, since only a singular type can be used for `config` storage elements.

Listing 25.9: 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 $G^{\texttt{tenv}}$, `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 $\langle$`e'`$\rangle^{/\!/\texttt{\#TE}}$;
  - applying *add_global_immutable_expr* to `name` and `e'` in `tenv` yields `new_tenv`.
- All of the following apply (LET_NON_NORMALIZABLE):
  - `gdk` is GDK_Let;
  - applying *normalize_opt* to `e` in `tenv` yields `None`$^{/\!/\texttt{\#TE}}$;
  - define `new_tenv` as `tenv`.
- All of the following apply (CONFIG):
  - `gdk` is GDK_Config;

  - ∗ view `typed_initial_value` as (`initial_value_type`, `initial_value'`, `ses_initial_value`);
  - ∗ checking that `initial_value_type` is a singular type yields `TRUE`$^{/\!/\texttt{\#TE}}$;
  - ∗ define `new_tenv` as tenv.
- All of the following apply (VAR):
  - ∗ `gdk` is GDK_Var;
  - ∗ define `new_tenv` as tenv.

### Formally

CONSTANT

$$\frac{\begin{array}{c}\texttt{typed\_initial\_value} \overset{\text{is}}{=} (\_, \texttt{initial\_value'}, \texttt{ses\_initial\_value}) \\ \mathit{static\_eval}(\mathsf{tenv}, \texttt{initial\_value'}) \xrightarrow{\text{type}} \mathtt{v} \\ \mathit{add\_global\_constant}(G^{\mathsf{tenv}}, \mathtt{name}, \mathtt{v}) \xrightarrow{\text{type}} G' \qquad \texttt{new\_tenv} := (G', L^{\mathsf{tenv}})\end{array}}{\mathit{update\_global\_storage}(\mathsf{tenv}, \mathtt{name}, \overbrace{\mathsf{GDK\_Constant}}^{\mathtt{gdk}}, \mathtt{typed\_initial\_value}) \xrightarrow[\texttt{new\_tenv}]{\text{type}}}$$

LET_NORMALIZABLE

$$\frac{\begin{array}{c}\mathit{normalize\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \langle \mathtt{e'} \rangle \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{add\_global\_immutable\_expr}(\mathsf{tenv}, \mathtt{name}, \mathtt{e'}) \xrightarrow{\text{type}} \texttt{new\_tenv}\end{array}}{\mathit{update\_global\_storage}(\mathsf{tenv}, \mathtt{name}, \overbrace{\mathsf{GDK\_Let}}^{\mathtt{gdk}}, \mathtt{typed\_initial\_value}) \xrightarrow[\texttt{new\_tenv}]{\text{type}}}$$

LET_NON_NORMALIZABLE

$$\frac{\mathit{normalize\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \texttt{None}}{\mathit{update\_global\_storage}(\mathsf{tenv}, \mathtt{name}, \overbrace{\mathsf{GDK\_Let}}^{\mathtt{gdk}}, \mathtt{typed\_initial\_value}) \xrightarrow[\overbrace{\texttt{new\_tenv}}^{\mathsf{tenv}}]{\text{type}}}$$

CONFIG

$$\frac{\begin{array}{c}\texttt{typed\_initial\_value} \overset{\text{is}}{=} (\texttt{initial\_value\_type}, \texttt{initial\_value'}, \texttt{ses\_initial\_value}) \\ \mathit{is\_singular}(\mathsf{tenv}, \texttt{initial\_value\_type}) \xrightarrow{\text{type}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#TE}\end{array}}{\mathit{update\_global\_storage}(\mathsf{tenv}, \mathtt{name}, \overbrace{\mathsf{GDK\_Config}}^{\mathtt{gdk}}, \mathtt{typed\_initial\_value}) \xrightarrow[\overbrace{\texttt{new\_tenv}}^{\mathsf{tenv}}]{\text{type}}}$$

VAR

$$\mathit{update\_global\_storage}(\mathsf{tenv}, \mathtt{name}, \overbrace{\mathsf{GDK\_Var}}^{\mathtt{gdk}}, \mathtt{typed\_initial\_value}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{tenv}}^{\mathtt{new\_tenv}}$$

### TypingRule.AddGlobalStorage

The function

$$\mathit{add\_global\_storage}(\overbrace{\mathbb{GSE}}^{\mathtt{genv}}, \overbrace{\mathsf{identifier}}^{\mathtt{name}}, \overbrace{\mathsf{global\_decl\_keyword}}^{\mathtt{keyword}}, \overbrace{\mathsf{ty}}^{\mathtt{declared\_t}}) \longrightarrow \overbrace{\mathbb{GSE}}^{\mathtt{new\_genv}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

returns a global static environment `new_genv` which is identical to the global static environment `genv`, except that the identifier `name`, which is assumed to name a global storage element, is bound to the global storage keyword `keyword` and type `declared_t`. Otherwise, the result is a type error.

### Example: Adding Global Storage Elements to the Static Environment

Adding the global storage elements in Listing 25.10, adds the following bindings to the global_storage_types map of a given static environments:

$$\begin{array}{rcl} \mathtt{PI} & \mapsto & (\mathsf{T\_Real}, \mathsf{GDK\_Constant}) \\ \mathtt{x} & \mapsto & (\mathsf{unconstrained\_integer}, \mathsf{GDK\_Var}) \\ \mathtt{c} & \mapsto & (\mathsf{T\_Int}(\mathsf{WellConstrained}(\overbrace{\mathsf{L\_Int}(42)}^{\mathsf{Constraint\_Exact}}))), \mathsf{GDK\_Config}) \end{array}$$

### Prose

All of the following apply:

- checking that `name` is not declared in the global environment of tenv yields $\mathtt{TRUE}^{/\!/\#\mathsf{TE}}$;
- `new_genv` is the global static environment of tenv with its global_storage_types component updated by binding `name` to (`declared_t`, `keyword`).

### Formally

$$\frac{\mathit{check\_var\_not\_in\_genv}(\mathtt{genv}, \mathtt{name}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \#\mathsf{TE} \qquad \mathtt{new\_genv} := \mathtt{genv}.\mathsf{global\_storage\_types}[\mathtt{name} \mapsto (\mathtt{declared\_t}, \mathtt{keyword})]}{\mathit{add\_global\_storage}(\mathtt{genv}, \mathtt{name}, \mathtt{keyword}, \mathtt{declared\_t}) \xrightarrow{\mathsf{type}} \mathtt{new\_genv}}$$

## 25.5 Semantics

This section defines the following relations:

- SemanticsRule.EvalGlobals
- SemanticsRule.DeclareGlobal

### SemanticsRule.EvalGlobals

The relation

$$\mathit{eval\_globals}(\overbrace{\mathsf{decl}^*}^{\texttt{decls}}, (\overbrace{\overbrace{\mathbb{E}}^{\texttt{env}} \times \overbrace{\mathcal{G}}^{\texttt{g1}}}^{\texttt{envm}})) \bigtimes \overbrace{(\mathbb{E} \times \mathcal{G})}^{C} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

updates the input environment and execution graph by initializing the global storage declarations.

### Example: Evaluating a List of Global Declarations

Evaluating the global storage declarations in Listing 25.10 results in updating the storage map of global dynamic environment by binding `PI` is bound to `Real`(157/50), `x` is bound to `Int`(5), and `c` is bound to `Int`(42).

Listing 25.10: Evaluating a list of global declarations

```
constant PI = 3.14;
var x : integer = 5;
config c : integer = 42;

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 (env, g1);
* the evaluation of the expression e in env yields Normal((v, g2), env2)$^{//\#T,\#DE}$;
* 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**

$$\text{EMPTY}\quad \frac{}{\mathit{eval\_globals}(\overbrace{[\,]}^{\texttt{decls}}, \texttt{envm}) \xrightarrow{\text{eval}} \overbrace{\texttt{envm}}^{C}}$$

$$\text{NON\_EMPTY}\quad \frac{\begin{array}{c}\texttt{d} \overset{\text{is}}{=} \texttt{D\_GlobalStorage}(\{\text{initial\_value} : \langle \texttt{e} \rangle, \text{name} : \texttt{name}, \ldots\}) \\ \texttt{envm} \overset{\text{is}}{=} (\texttt{env}, \texttt{g1}) \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g2}), \texttt{env2}) \;//\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{declare\_global}(\texttt{name}, \texttt{v}, \texttt{env2}) \xrightarrow{\text{eval}} \texttt{env3} \\ \mathit{eval\_globals}(\texttt{decls}', (\texttt{env3}, \texttt{g1} \xrightarrow{\texttt{asl\_po}} \texttt{g2})) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_globals}(\overbrace{[\texttt{d}] + \texttt{decls}'}^{\texttt{decls}}, \texttt{envm}) \xrightarrow{\text{eval}} C}$$

**SemanticsRule.DeclareGlobal**

**Prose**

The relation

$$\mathit{declare\_global}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathbb{E}}^{\texttt{env}}) \times \overbrace{\mathbb{E}}^{\texttt{new\_env}}$$

updates the environment env by mapping name to v in the storage map of the global dynamic environment $G^{\text{denv}}$.

**Example: Declaring a Global Storage Element**

Evaluating the specification in Listing 25.11, results in updating the storage map of the global dynamic environment by binding x to Int(5).

Listing 25.11: Declaring a global storage element

```
var x : integer = 5;

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

**Formally**

$$\frac{\texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}.\mathsf{storage}[\texttt{name} \mapsto \texttt{v}], L^{\mathsf{denv}}))}{\textit{declare\_global}(\texttt{name}, \texttt{v}, \texttt{env}) \xrightarrow{\text{eval}} \texttt{new\_env}}$$

# Chapter 26

# Type Declarations

Type declarations are grammatically derived from `decl` via the subset of productions shown in Section 26.1 and represented 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) "}"
      fields_opt ⟶ fields | ϵ
typed_identifier ⟶ ID as_ty
           as_ty ⟶ ":" ty
```

## 26.2 Abstract Syntax

$$\text{decl} \longrightarrow \texttt{D\_TypeDecl}(\text{identifier}, \text{ty}, (\text{identifier}, \overbrace{\text{field}^*}^{\text{with fields}})?)$$

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

### ASTRule.TypeDecl

TYPE_DECL

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"type"}, \mathbf{ID}(\texttt{x}), \texttt{"of"}, \texttt{ty\_decl}, \texttt{subtype\_opt}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{D\_TypeDecl}(\texttt{x}, \overline{\texttt{t}}, \overline{\texttt{subtype\_opt}})]}^{\texttt{ast\_node}}$$

SUBTYPE_DECL

$$\frac{\mathit{build\_subtype}(\texttt{subtype}) \xrightarrow{\text{ast}} \texttt{s} \qquad \texttt{s} \overset{\text{is}}{=} (\texttt{name}, \texttt{fields})}{\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"type"}, \mathbf{ID}(\texttt{x}), \texttt{"of"}, \texttt{subtype}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{D\_TypeDecl}(\texttt{x}, \mathsf{T\_Named}(\texttt{name}), \langle(\texttt{name}, \texttt{fields})\rangle)]}^{\texttt{ast\_node}}}$$

### ASTRule.Subtype

The function

$$\mathit{build\_subtype}(\overbrace{\mathbb{PARSE}[\texttt{subtype}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\text{identifier} \times (\text{identifier} \times \text{ty})^*)}^{\texttt{ast\_node}}$$

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

WITH_FIELDS

$$\mathit{build\_subtype}(\overbrace{\texttt{subtype}(\texttt{"subtypes"}, \mathbf{ID}(\texttt{id}), \texttt{"with"}, \texttt{fields})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{fields}})}^{\texttt{ast\_node}}$$

NO_FIELDS

$$\mathit{build\_subtype}(\overbrace{\texttt{subtype}(\texttt{"subtypes"}, \mathbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, [\,])}^{\texttt{ast\_node}}$$

### ASTRule.Subtypeopt

The function

$$\mathit{build\_subtype\_opt}(\overbrace{\mathbb{PARSE}[\texttt{subtype\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle(\text{identifier} \times \langle(\text{identifier} \times \text{ty})^*\rangle)\rangle}^{\texttt{ast\_node}}$$

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

SUBTYPE_OPT

$$\frac{\mathit{build\_option}[\texttt{subtype}](\texttt{subtype\_opt}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_subtype\_opt}(\overbrace{\texttt{subtype\_opt}(\texttt{subtype\_opt} : \text{option}(\texttt{subtype}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

**ASTRule.Fields**

The function

$$\mathit{build\_fields}(\overbrace{\mathbb{PARSE}[\texttt{fields}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{field}^*}^{\texttt{ast\_node}}$$

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

EMPTY

$$\mathit{build\_fields}(\texttt{fields}(\texttt{"\{"}, \texttt{"-"}, \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{ast\_node}}$$

NON_EMPTY

$$\frac{\mathit{build\_tclist}[\mathit{build\_typed\_identifier}](\texttt{fields}) \xrightarrow{\text{ast}} \texttt{field\_asts}}{\mathit{build\_fields}(\texttt{fields}(\texttt{"\{"}, \texttt{fields} : \mathsf{tclist1}(\texttt{typed\_identifier}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{field\_asts}}^{\texttt{ast\_node}}}$$

**ASTRule.FieldsOpt**

The function

$$\mathit{build\_fields\_opt}(\overbrace{\mathbb{PARSE}[\texttt{fields\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{field}^*}^{\texttt{ast\_node}}$$

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

FIELDS

$$\mathit{build\_fields\_opt}(\texttt{fields\_opt}(\texttt{fields})) \xrightarrow{\text{ast}} \overbrace{\texttt{fields}}^{\texttt{ast\_node}}$$

EMPTY

$$\mathit{build\_fields\_opt}(\texttt{fields\_opt}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{ast\_node}}$$

## 26.3 Typing

We also define the following helper rules:

- TypingRule.AnnotateTypeOpt
- TypingRule.AnnotateExprOpt
- TypingRule.AddGlobalStorage
- TypingRule.DeclareType
- TypingRule.AnnotateExtraFields
- TypingRule.DeclareEnumLabels
- TypingRule.DeclareConst

### TypingRule.DeclareType

The function

$$\mathit{declare\_type}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}, \overbrace{\langle(\mathsf{identifier} \times \texttt{field}^*)\rangle}^{\texttt{s}}) \longrightarrow \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 only side effect descriptors for `Record` is ReadGlobal(`num_bits`, Constant, `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;
type Collection of 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- define tenv as the static environment whose global component is `genv` and its local component is the empty local static environment;
- annotating the optional extra fields `s` for `ty` in tenv yields via *annotate_extra_fields* yields the modified environment `tenv1` and type $\texttt{t1}^{/\!/\texttt{\#TE}}$;

- annotating `t1` in `tenv1` yields $(\texttt{t2}, \texttt{ses\_t})^{/\!/\texttt{\#TE}}$;
- applying *max_time_frame* to `ses_t` yields `time_frame`;
- applying *add_type* to `name`, `t2`, and `time_frame` in `tenv` yields `tenv2`;
- `tenv2` is `tenv1` with its declared_types component updated by binding `name` to `t2`;
- One of the following applies:
  - All of the following apply (ENUM):
    - `t2` is an enumeration type with labels `ids`, that is, T_Enum(`ids`);
    - applying *declare_enum_labels* to `t2` in `tenv2` $\texttt{new\_tenv}^{/\!/\texttt{\#TE}}$.
  - All of the following apply (NOT_ENUM):
    - `t2` is not an enumeration type;
    - `new_tenv` is `tenv2`.

**Formally**

ENUM

$$\frac{\begin{array}{c}\textit{check\_var\_not\_in\_genv}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \textit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv} \\ \textit{annotate\_extra\_fields}(\texttt{tenv}, \texttt{name}, \texttt{ty}, \texttt{s}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{t1}) \\ \textit{annotate\_type}(\texttt{TRUE}, \texttt{tenv1}, \texttt{t1}) \xrightarrow{\text{type}} (\texttt{t2}, \texttt{ses\_t}) \;/\!/\; \texttt{\#TE} \\ \textit{max\_time\_frame}(\texttt{ses\_t}) \xrightarrow{\text{type}} \texttt{time\_frame} \\ \textit{add\_type}(\texttt{tenv1}, \texttt{name}, \texttt{t2}, \texttt{time\_frame}) \xrightarrow{\text{type}} \texttt{tenv2} \\ \text{******** common prefix ********} \\ \texttt{t2} = \texttt{T\_Enum}(\texttt{ids}) \qquad \textit{declare\_enum\_labels}(\texttt{tenv2}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{new\_tenv} \;/\!/\; \texttt{\#TE}\end{array}}{\textit{declare\_type}(\texttt{genv}, \texttt{name}, \texttt{ty}, \texttt{s}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

NOT_ENUM

$$\frac{\begin{array}{c}\textit{check\_var\_not\_in\_genv}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \textit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv} \\ \textit{annotate\_extra\_fields}(\texttt{tenv}, \texttt{name}, \texttt{ty}, \texttt{s}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{t1}) \\ \textit{annotate\_type}(\texttt{TRUE}, \texttt{tenv1}, \texttt{t1}) \xrightarrow{\text{type}} (\texttt{t2}, \texttt{ses\_t}) \;/\!/\; \texttt{\#TE} \\ \textit{max\_time\_frame}(\texttt{ses\_t}) \xrightarrow{\text{type}} \texttt{time\_frame} \\ \textit{add\_type}(\texttt{tenv1}, \texttt{name}, \texttt{t2}, \texttt{time\_frame}) \xrightarrow{\text{type}} \texttt{tenv2} \\ \text{******** common prefix ********} \\ \textit{ast\_label}(\texttt{t2}) \neq \texttt{T\_Enum}\end{array}}{\textit{declare\_type}(\texttt{genv}, \texttt{name}, \texttt{ty}, \texttt{s}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv2}}^{\texttt{new\_tenv}}}$$

### TypingRule.AnnotateExtraFields

The function

$$\mathit{annotate\_extra\_fields}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}, \overbrace{\langle\langle \overbrace{\mathsf{identifier}}^{\texttt{super}} \times \overbrace{\texttt{field}^*}^{\texttt{extra\_fields}} \rangle\rangle}^{\texttt{s}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathsf{ty}}^{\texttt{new\_ty}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the type `ty` with the optional extra fields `s` in `tenv`, yielding the modified environment `new_tenv` and type `new_ty`. Otherwise, the result is a type error.

### 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 $\langle$(`super`, `extra_fields`)$\rangle$;
  - checking that `ty` subtype-satisfies the named type `super` (that is, T_Named(`super`)) yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - `extra_fields` is the empty list;
  - `new_tenv` is `tenv` with its subtypes component updated by binding `name` to `super`;
  - `new_ty` is `ty`.
- All of the following apply (NO_SUPER):

  - `s` is $\langle$(`super`, `extra_fields`)$\rangle$;
  - checking that `ty` subtype-satisfies the named type `super` (that is, T_Named(`super`)) yields TRUE$^{//\#TE}$;
  - `extra_fields` is not an empty list;
  - `super` is not bound to a type in `tenv`;
  - the result is a type error indicating that `super` is not a declared type.

- All of the following apply (STRUCTURED):
  - `s` is $\langle$(`super`, `extra_fields`)$\rangle$;
  - checking that `ty` subtype-satisfies the named type `super` (that is, T_Named(`super`)) yields TRUE$^{//\#TE}$;
  - `extra_fields` is not an empty list;
  - `super` is bound to a type `t` in `tenv`;
  - checking that `t` is a structured type yields TRUE or a type error indicating that a structured type was expected, thereby short-circuiting the entire rule;
  - `t` has AST label $L$ and fields `fields`;
  - `new_ty` is the type with AST label $L$ and list fields that is the concatenation of `fields` and `extra_fields`;
  - `new_tenv` is `tenv` with its subtypes component updated by binding `name` to `super`.

**Formally**

NONE

$$\mathit{annotate\_extra\_fields}(\mathsf{tenv}, \mathtt{name}, \mathtt{ty}, \overbrace{\mathsf{None}}^{\mathtt{s}}) \xrightarrow{\text{type}} (\ \overbrace{\mathtt{tenv}}^{\mathtt{new\_tenv}}\ ,\ \overbrace{\mathtt{ty}}^{\mathtt{new\_ty}}\ )$$

EMPTY_FIELDS

$$\frac{\mathit{subtype\_satisfies}(\mathtt{ty}, \mathsf{T\_Named}(\mathtt{super})) \xrightarrow{\text{type}} \mathtt{b} \qquad \mathit{check}(\mathtt{b}, \mathtt{TE\_UT}) \xrightarrow{\text{type}} \mathtt{TRUE}\ //\ \mathtt{\#TE} \qquad \mathtt{extra\_fields} = [\,] \qquad \mathtt{new\_tenv} := (G^{\mathsf{tenv}}.\mathsf{subtypes}[\mathtt{name} \mapsto \mathtt{super}], L^{\mathsf{tenv}})}{\mathit{annotate\_extra\_fields}(\mathsf{tenv}, \mathtt{name}, \mathtt{ty}, \overbrace{\langle(\mathtt{super}, \mathtt{extra\_fields})\rangle}^{\mathtt{s}}) \xrightarrow{\text{type}} (\mathtt{new\_tenv}, \overbrace{\mathtt{ty}}^{\mathtt{new\_ty}})}$$

NO_SUPER

$$\frac{\mathit{subtype\_satisfies}(\mathtt{ty}, \mathsf{T\_Named}(\mathtt{super})) \xrightarrow{\text{type}} \mathtt{b} \qquad \mathit{check}(\mathtt{b}, \mathtt{TE\_UT}) \xrightarrow{\text{type}} \mathtt{TRUE}\ //\ \mathtt{\#TE} \qquad \mathtt{extra\_fields} \neq [\,] \qquad G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{super}) = \bot}{\mathit{annotate\_extra\_fields}(\mathsf{tenv}, \mathtt{name}, \mathtt{ty}, \overbrace{\langle(\mathtt{super}, \mathtt{extra\_fields})\rangle}^{\mathtt{s}}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_UI})}$$

STRUCTURED

$$\frac{\begin{array}{c}\mathit{subtype\_satisfies}(\texttt{ty}, \mathsf{T\_Named}(\texttt{super})) \xrightarrow{\text{type}} \texttt{b} \qquad \mathit{check}(\texttt{b}, \texttt{TE\_UT}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ \texttt{extra\_fields} \neq [\,] \\ G^{\text{tenv}}.\mathsf{declared\_types}(\texttt{super}) = (\texttt{t}, \_) \\ \mathit{check}(\mathit{ast\_label}(\texttt{t}) \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\}, \texttt{ExpectedStructuredType}) \xrightarrow{\text{type}} \\ \texttt{TRUE} \;//\; \texttt{\#TE} \\ \texttt{t} \overset{\text{is}}{=} L(\texttt{fields}) \qquad \texttt{new\_ty} := L(\texttt{fields} + \texttt{extra\_fields}) \\ \texttt{new\_tenv} := (G^{\text{tenv}}.\mathsf{subtypes}[\texttt{name} \mapsto \texttt{super}], L^{\text{tenv}})\end{array}}{\mathit{annotate\_extra\_fields}(\texttt{tenv}, \texttt{name}, \texttt{ty}, \overbrace{\langle(\texttt{super}, \texttt{extra\_fields})\rangle}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_ty})}$$

### TypingRule.AnnotateTypeOpt

The function

$$\mathit{annotate\_type\_opt}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{t}}) \xrightarrow{\text{type}} \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ty\_opt'}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the type `t` inside an optional `ty_opt`, if there is one, and leaves it as is if `ty_opt` is `None`. Otherwise, the result is a type error.

### Example: Annotating Optional Types

The following are examples of annotating a type inside an optional with the empty static environment:

$$\begin{array}{rcl} \mathit{annotate\_type\_opt}(\emptyset_{\mathbb{SE}}, \texttt{None}) & \xrightarrow{\text{type}} & \texttt{None} \\ \mathit{annotate\_type\_opt}(\emptyset_{\mathbb{SE}}, \langle \mathsf{unconstrained\_integer} \rangle) & \xrightarrow{\text{type}} & \langle \mathsf{unconstrained\_integer} \rangle \end{array}$$

### Prose

One of the following applies:

- All of the following apply (NONE):
  - `ty_opt` is `None`;
  - `ty_opt'` is `ty_opt`.
- All of the following apply (SOME):
  - `ty_opt` contains the type `t`;
  - annotating `t` in `tenv` yields $\texttt{t1}^{//\texttt{\#TE}}$;
  - `ty_opt'` is $\langle \texttt{t1} \rangle$.

**Formally**

NONE

$$\mathit{annotate\_type\_opt}(\mathsf{tenv}, \overbrace{\mathtt{None}}^{\mathtt{ty\_opt}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{ty\_opt}}^{\mathtt{ty\_opt'}}$$

SOME

$$\frac{\mathit{annotate\_type}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t1} \;/\!/\; \#\mathsf{TE}}{\mathit{annotate\_type\_opt}(\mathsf{tenv}, \overbrace{\langle \mathtt{t} \rangle}^{\mathtt{ty\_opt}}) \xrightarrow{\mathsf{type}} \overbrace{\langle \mathtt{t1} \rangle}^{\mathtt{ty\_opt'}}}$$

**TypingRule.AnnotateExprOpt**

The function

$$\mathit{annotate\_expr\_opt}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\langle \mathsf{expr} \rangle}^{\mathtt{expr\_opt}}) \longrightarrow \overbrace{(\langle \mathsf{expr} \rangle \times \langle \mathsf{ty} \rangle)}^{\mathtt{res}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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:

$$\mathit{annotate\_expr\_opt}(\emptyset_{\mathbb{SE}}, \mathtt{None}) \xrightarrow{\mathsf{type}} (\mathtt{None}, \mathtt{None})$$

$$\mathit{annotate\_expr\_opt}(\emptyset_{\mathbb{SE}}, \langle \mathsf{L\_Int}(5) \rangle) \xrightarrow{\mathsf{type}} (\langle \mathsf{L\_Int}(5) \rangle, \langle \mathsf{T\_Int}(\mathsf{WellConstrained}(\overbrace{\mathsf{L\_Int}(5)}^{\mathsf{Constraint\_Exact}}))) \rangle)$$

**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 $(\mathtt{t}, \mathtt{e'})^{/\!/ \#\mathsf{TE}}$;
  - `res` is $(\langle \mathtt{t} \rangle, \langle \mathtt{e'} \rangle)$.

**Formally**

NONE

$$\mathit{annotate\_expr\_opt}(\mathsf{tenv},\ \overbrace{\texttt{None}}^{\texttt{expr\_opt}}\ ) \xrightarrow{\text{type}} (\texttt{None}, \texttt{None})$$

SOME

$$\frac{\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} (\mathsf{t}, \mathsf{e'})\ \ /\!/\ \texttt{\#TE}}{\mathit{annotate\_expr\_opt}(\mathsf{tenv},\ \overbrace{\langle \mathsf{e} \rangle}^{\texttt{expr\_opt}}\ ) \xrightarrow{\text{type}} \overbrace{(\langle \mathsf{t} \rangle, \langle \mathsf{e'} \rangle)}^{\texttt{res}}}$$

**TypingRule.DeclaredType**

The helper function

$$\mathit{declared\_type}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{id}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{t}} \cup \mathsf{TTypeError}$$

retrieves the type associated with the identifier `id` in the static environment `tenv`. If the identifier is not associated with a declared type, 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**

$$\frac{\overset{\text{EXISTS}}{G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{id}) = (\mathtt{t}, \_)}}{\mathit{declared\_type}(\mathsf{tenv}, \mathtt{id}) \xrightarrow{\mathsf{type}} \mathtt{t}}$$

$$\frac{\overset{\text{TYPE\_NOT\_DECLARED}}{G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{id}) = \bot}}{\mathit{declared\_type}(\mathsf{tenv}, \mathtt{id}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UI})}$$

**TypingRule.DeclareEnumLabels**

The function

$$\mathit{declare\_enum\_labels}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{identifier}}^{\mathtt{name}}, \overbrace{\mathsf{identifier}^{+}}^{\mathtt{ids}}, \longrightarrow \overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}})$$

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 13.19, the declaration of the `Color` enumeration type updates the static environment with the following constant variables:

| identifier | associated type | associated constant value |
|---|---|---|
| `RED` | `T_Named(Color)` | `L_Label(RED)` |
| `GREEN` | `T_Named(Color)` | `L_Label(GREEN)` |
| `BLUE` | `T_Named(Color)` | `L_Label(BLUE)` |

**Prose**

All of the following apply:

- `ids` is the (non-empty) list of labels $\mathtt{id}_{1..k}$;
- $\mathsf{tenv}_0$ is tenv;
- declaring the constant $\mathtt{id}_i$ with the type `T_Named(name)` and literal `L_Label(`$\mathtt{id}_i$`)` in $\mathsf{tenv}_{i-1}$ via *declare_const* yields $\mathsf{tenv}_i$, for $i = 1$ to $k$ (if $k > 1$)$^{\#\mathtt{TE}}$;
- `new_tenv` is $\mathsf{tenv}_k$.

**Formally**

$$\frac{\texttt{ids} \overset{\text{is}}{=} \texttt{id}_{1..k} \qquad \mathsf{tenv}_0 := \mathsf{tenv} \qquad \texttt{i} = 1..k : \mathit{declare\_const}(\mathsf{tenv}_{\texttt{i}-1}, \texttt{id}_\texttt{i}, \mathsf{T\_Named}(\texttt{name}), \mathsf{L\_Label}(\texttt{id}_\texttt{i})) \xrightarrow{\text{type}} \mathsf{tenv}_\texttt{i} \;/\!/\; \#\mathsf{TE}}{\mathit{declare\_enum\_labels}(\mathsf{tenv}, \texttt{name}, \texttt{ids}) \xrightarrow{\text{type}} \overbrace{\mathsf{tenv}_k}^{\texttt{new\_tenv}}}$$

**TypingRule.DeclareConst**

The function

$$\mathit{declare\_const}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}, \overbrace{\mathsf{literal}}^{vv}) \longrightarrow \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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:

$$\mathit{declare\_const}(G^{\emptyset_{\mathbb{SE}}}, \texttt{PI}, \mathsf{T\_Real}, \mathsf{L\_Real}(157/50)) \xrightarrow{\text{type}} G^{\emptyset_{\mathbb{SE}}} \left[\begin{array}{l} \mathsf{global\_storage\_types} \mapsto \{\texttt{PI} \mapsto (\mathsf{T\_Real}, \mathsf{GDK\_Constant})\}, \\ \mathsf{constant\_values}[\texttt{PI} \mapsto \mathsf{L\_Real}(157/50)] \end{array}\right]$$

(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 $\texttt{genv1}^{/\!/\#\mathsf{TE}}$;
- applying *add_global_constant* to `name` and `v` in `genv1` yields `new_genv`.

**Formally**

$$\frac{\mathit{add\_global\_storage}(\texttt{genv}, \texttt{name}, \mathsf{GDK\_Constant}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{genv1} \;/\!/\; \#\mathsf{TE} \qquad \mathit{add\_global\_constant}(\texttt{genv1}, \texttt{name}, \texttt{v}) \xrightarrow{\text{type}} \texttt{new\_genv}}{\mathit{declare\_const}(\texttt{genv}, \texttt{name}, \texttt{ty}, \texttt{v}) \xrightarrow{\text{type}} \texttt{new\_genv}}$$

# Chapter 27

# Subprogram Declarations

Subprogram declarations are grammatically derived from `decl` via the subset of productions shown in Section 27.1 and represented as ASTs via the production of `decl` shown in Section 27.2. Subprogram declarations are typed via *annotate_and_declare_func*, which is defined in TypingRule.AnnotateAndDeclareFunc. Subprogram declarations have no associated semantics.

## 27.1 Syntax

```
          decl ⟶ override "func" ID params_opt func_args return_type
                   ↪ recurse_limit func_body
                 | 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  ⟶ "getter" maybe_empty_stmt_list "end" ";"
                        ↪"setter" maybe_empty_stmt_list "end" ";"
                      | "setter" maybe_empty_stmt_list "end" ";"
                        ↪"getter" maybe_empty_stmt_list "end" ";"
           override   ⟶(inline) ϵ | "impdef" | "implementation"
```

## 27.2 Abstract Syntax

$$\text{decl} \longrightarrow \texttt{D\_Func}(\text{func})$$

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

$$\begin{aligned} \text{typed\_identifier} &\longrightarrow (\text{identifier}, \text{ty}) \\ \text{sub\_program\_type} &\longrightarrow \texttt{ST\_Procedure} \mid \texttt{ST\_Function} \\ &\quad \mid \texttt{ST\_Getter} \mid \texttt{ST\_Setter} \\ \text{override\_info} &\longrightarrow \text{Impdef} \mid \text{Implementation} \end{aligned}$$

### ASTRule.GlobalDecl

The relation

$$\mathit{build\_decl} : \overbrace{\mathbb{PARSE}[\texttt{decl}]}^{\texttt{parsed\_node}} \times \overbrace{\texttt{decl}^*}^{\texttt{ast\_node}}$$

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

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.

$$\text{accessor\_pair} \longrightarrow \left\{ \begin{array}{rcl} \text{getter} & : & \text{stmt}, \\ \text{setter} & : & \text{stmt} \end{array} \right\}$$

FUNC_DECL

$$\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{l}\texttt{override}, \texttt{"func"}, \textbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{func\_args}, \texttt{return\_type}, \\ \hookrightarrow \texttt{recurse\_limit}, \texttt{func\_body}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}}$$

$$\overbrace{\left[\texttt{D\_Func}\left(\left\{\begin{array}{rcl} \text{name} & : & \overline{\texttt{name}}, \\ \text{parameters} & : & \overline{\texttt{params\_opt}}, \\ \text{args} & : & \overline{\texttt{func\_args}}, \\ \text{body} & : & \overline{\texttt{func\_body}}, \\ \text{return\_type} & : & \langle\overline{\texttt{return\_type}}\rangle, \\ \text{subprogram\_type} & : & \texttt{ST\_Function}, \\ \text{recurse\_limit} & : & \langle\overline{\texttt{recurse\_limit}}\rangle \\ \text{builtin} & : & \texttt{FALSE} \\ \text{override} & : & \overline{\texttt{override}} \end{array}\right\}\right)\right]}^{\texttt{ast\_node}}$$

PROCEDURE_DECL

$$\mathit{build\_decl}\left(\overbrace{\texttt{decl}(\texttt{override}, \texttt{"func"}, \textbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{func\_args}, \texttt{func\_body})}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}}$$

$$\overbrace{\left[\texttt{D\_Func}\left(\left\{\begin{array}{rcl} \text{name} & : & \overline{\texttt{name}}, \\ \text{parameters} & : & \overline{\texttt{params\_opt}}, \\ \text{args} & : & \overline{\texttt{func\_args}}, \\ \text{body} & : & \overline{\texttt{func\_body}}, \\ \text{return\_type} & : & \texttt{None}, \\ \text{subprogram\_type} & : & \texttt{ST\_Procedure}, \\ \text{recurse\_limit} & : & \texttt{None} \\ \text{builtin} & : & \texttt{FALSE} \\ \text{override} & : & \overline{\texttt{override}} \end{array}\right\}\right)\right]}^{\texttt{ast\_node}}$$

ACCESSOR

$$\mathit{build\_accessor\_body}(\texttt{body}) \xrightarrow{\text{ast}} \texttt{accessor\_pair}$$

$$\mathit{desugar\_accessor\_pair}\left(\begin{array}{l}\texttt{override},\\ \texttt{name},\\ \overline{\texttt{params\_opt}},\\ \overline{\texttt{func\_args}},\\ \texttt{setter\_arg},\\ \overline{\texttt{ty}},\\ \texttt{accessor\_pair}\end{array}\right) \xrightarrow{\text{ast}} \texttt{ast\_node}$$

---

$$\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{l}\texttt{override}, \texttt{"accessor"}, \mathbf{ID}(\texttt{name}), \texttt{params\_opt},\\ \hookrightarrow \texttt{func\_args}, \texttt{"<=>"}, \mathbf{ID}(\texttt{setter\_arg}), \texttt{as\_ty}, \texttt{body} : \texttt{accessor\_body}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \texttt{ast\_node}$$

### 27.2.1 Example

Listing 27.1 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 27.1: 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
  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;
```

#### ASTRule.AccessorBody

The relation

$$\mathit{build\_accessor\_body} : \overbrace{\mathbb{PARSE}[\texttt{accessor\_body}]}^{\texttt{parsed\_node}} \times \overbrace{\texttt{accessor\_pair}}^{\texttt{accessor\_pair}}$$

transforms a parse node `parsed_node` into an accessor_pair.

$$\frac{\mathit{build\_accessors}(\texttt{accessors}) \xrightarrow{\text{ast}} \texttt{accessor\_pair}}{\mathit{build\_accessor\_body}\left(\overbrace{\texttt{decl}\,(\ \texttt{"begin"}, \texttt{accessors} : \texttt{accessors}, \texttt{"end"}, \texttt{";"}\ )}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \texttt{accessor\_pair}}$$

### ASTRule.RecurseLimit

The function

$$\mathit{build\_recurselimit}(\overbrace{\mathbb{PARSE}[\texttt{recurse\_limit}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{accessor\_pair}}^{\texttt{accessor\_pair}}$$

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

LIMIT

$$\mathit{build\_recurselimit}\left(\overbrace{\texttt{recurse\_limit}(\texttt{"recurselimit"}, \texttt{expr})}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\langle\overline{\texttt{expr}}\rangle}^{\texttt{ast\_node}}$$

NO_LIMIT

$$\mathit{build\_recurselimit}\left(\overbrace{\texttt{recurse\_limit}(\epsilon)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{None}}^{\texttt{ast\_node}}$$

### ASTRule.TypedIdentifier

The function

$$\mathit{build\_typed\_identifier}(\overbrace{\mathbb{PARSE}[\texttt{typed\_identifier}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\texttt{identifier} \times \texttt{ty})}^{\texttt{ast\_node}}$$

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

$$\mathit{build\_typed\_identifier}(\overbrace{\texttt{typed\_identifier}(\textbf{ID}(\texttt{id}), \texttt{as\_ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{as\_ty}})}^{\texttt{ast\_node}}$$

### ASTRule.OptTypedIdentifier

The function

$$\mathit{build\_opt\_typed\_identifier}(\overbrace{\mathbb{PARSE}[\texttt{opt\_typed\_identifier}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\texttt{identifier} \times \langle\texttt{ty}\rangle)}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_option}[\texttt{as\_ty}](\texttt{as\_ty\_opt}) \xrightarrow{\text{ast}} \texttt{as\_ty\_opt\_ast}}{\mathit{build\_opt\_typed\_identifier}(\overbrace{\texttt{typed\_identifier}(\textbf{ID}(\texttt{id}), \texttt{as\_ty\_opt} : \mathsf{option}(\texttt{as\_ty}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \texttt{as\_ty\_opt\_ast})}^{\texttt{ast\_node}}}$$

### ASTRule.ReturnType

The function

$$\mathit{build\_return\_type}(\overbrace{\mathbb{PARSE}[\texttt{return\_type}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{ast\_node}}$$

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

$$\mathit{build\_return\_type}(\overbrace{\texttt{return\_type}(\texttt{"=>"}, \mathsf{ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\overline{\mathsf{ty}}}^{\texttt{ast\_node}}$$

### ASTRule.ParamsOpt

The function

$$\mathit{build\_params\_opt}(\overbrace{\mathbb{PARSE}[\texttt{params\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\texttt{ast\_node}}$$

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

EMPTY

$$\mathit{build\_params\_opt}(\overbrace{\texttt{params\_opt}(\mathsf{epsilon\_node})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{ast\_node}}$$

NON_EMPTY

$$\frac{\mathit{build\_clist}[\texttt{opt\_typed\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_params\_opt}(\overbrace{\texttt{params\_opt}(\texttt{"\{"}, \texttt{ids} : \mathsf{clist0}(\texttt{opt\_typed\_identifier}), \texttt{"\}"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{ids\_ast}}^{\texttt{ast\_node}}}$$

### ASTRule.FuncArgs

The function

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{func\_args}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_clist}[\texttt{typed\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_func\_args}(\overbrace{\texttt{func\_args}(\texttt{"("}, \texttt{ids} : \mathsf{clist0}(\texttt{typed\_identifier}), \texttt{")"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{ids\_ast}}^{\texttt{ast\_node}}}$$

### ASTRule.MaybeEmptyStmtList

The function

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\mathbb{PARSE}[\texttt{maybe\_empty\_stmt\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

EMPTY

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\texttt{maybe\_empty\_stmt\_list}(\mathsf{epsilon\_node})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pass}}^{\texttt{ast\_node}}$$

NON_EMPTY

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\texttt{maybe\_empty\_stmt\_list}(\texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{stmt\_list}}^{\texttt{ast\_node}}$$

### ASTRule.FuncBody

The function

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{func\_body}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

$$\mathit{build\_func\_body}(\overbrace{\texttt{func\_body}(\texttt{"begin"}, \texttt{stmts} : \texttt{maybe\_empty\_stmt\_list}, \texttt{"end"}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{maybe\_empty\_stmt\_list}}^{\texttt{ast\_node}}$$

### ASTRule.Accessors

The function

$$\mathit{build\_accessors}(\overbrace{\mathbb{PARSE}[\texttt{accessors}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{accessor\_pair}}^{\texttt{accessor\_pair}}$$

transforms a parse node `parsed_node` into an accessor_pair.

$$\frac{\begin{array}{r}\mathit{build\_stmt}(\texttt{getter}) \xrightarrow{\text{ast}} \texttt{getter\_ast} \\ \mathit{build\_stmt}(\texttt{setter}) \xrightarrow{\text{ast}} \texttt{setter\_ast} \\ \texttt{accessor\_pair} := \left\{ \begin{array}{lcl} \text{getter} & : & \texttt{getter\_ast}, \\ \text{setter} & : & \texttt{setter\_ast} \end{array} \right\}\end{array}}{\begin{array}{r}\mathit{build\_accessors}(\texttt{"getter"}, \texttt{getter} : \texttt{maybe\_empty\_stmt\_list}, \texttt{"end"}, \texttt{";"}, \\ \texttt{"setter"}, \texttt{setter} : \texttt{maybe\_empty\_stmt\_list}, \texttt{"end"}, \texttt{";"}) \\ \xrightarrow{\text{ast}} \texttt{accessor\_pair}\end{array}}$$

$$\frac{\begin{array}{r}\mathit{build\_stmt}(\texttt{getter}) \xrightarrow{\text{ast}} \texttt{getter\_ast} \\ \mathit{build\_stmt}(\texttt{setter}) \xrightarrow{\text{ast}} \texttt{setter\_ast} \\ \texttt{accessor\_pair} := \left\{ \begin{array}{lcl} \text{getter} & : & \texttt{getter\_ast}, \\ \text{setter} & : & \texttt{setter\_ast} \end{array} \right\}\end{array}}{\begin{array}{r}\mathit{build\_accessors}(\texttt{"setter"}, \texttt{setter} : \texttt{maybe\_empty\_stmt\_list}, \texttt{"end"}, \texttt{";"}, \\ \texttt{"getter"}, \texttt{getter} : \texttt{maybe\_empty\_stmt\_list}, \texttt{"end"}, \texttt{";"}) \\ \xrightarrow{\text{ast}} \texttt{accessor\_pair}\end{array}}$$

### ASTRule.DesugarAccessorPair

The function

$$\mathit{desugar\_accessor\_pair} \left( \begin{array}{l} \overbrace{\mathsf{override\_info}}^{\texttt{override}}, \\ \overbrace{\mathsf{identifier}}^{\texttt{name}}, \\ \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\texttt{parameters}}, \\ \overbrace{\mathsf{typed\_identifier}^*}^{\texttt{args}}, \\ \overbrace{\mathsf{identifier}}^{\texttt{setter\_arg}}, \\ \overbrace{\mathsf{ty}}^{\texttt{ty}}, \\ \overbrace{\mathsf{accessor\_pair}}^{\texttt{accessor\_pair}} \end{array} \right) \longrightarrow \overbrace{\mathsf{decl}^*}^{\texttt{ast\_node}}$$

transforms an accessor_pair into an AST node `ast_node`.

$$\begin{array}{l}\texttt{getter} := \texttt{D\_Func}\left\{\begin{array}{rcl}\text{name} & : & \texttt{name},\\ \text{parameters} & : & \texttt{parameters},\\ \text{args} & : & \texttt{args},\\ \text{body} & : & \texttt{accessor\_pair}.\text{getter},\\ \text{return\_type} & : & \langle \texttt{ty} \rangle,\\ \text{subprogram\_type} & : & \texttt{ST\_Getter},\\ \text{recurse\_limit} & : & \texttt{None}\\ \text{builtin} & : & \texttt{FALSE}\\ \text{override} & : & \texttt{override}\end{array}\right\}\\ \texttt{setter\_args} := [(\texttt{setter\_arg}, \texttt{ty})] + \texttt{args}\\ \texttt{setter} := \texttt{D\_Func}\left\{\begin{array}{rcl}\text{name} & : & \texttt{name},\\ \text{parameters} & : & \texttt{parameters},\\ \text{args} & : & \texttt{setter\_args},\\ \text{body} & : & \texttt{accessor\_pair}.\text{setter},\\ \text{return\_type} & : & \texttt{None},\\ \text{subprogram\_type} & : & \texttt{ST\_Setter},\\ \text{recurse\_limit} & : & \texttt{None}\\ \text{builtin} & : & \texttt{FALSE}\\ \text{override} & : & \texttt{override}\end{array}\right\}\\ \hline \textit{desugar\_accessor\_pair}\left(\begin{array}{l}\texttt{override},\\ \texttt{name},\\ \texttt{parameters},\\ \texttt{args},\\ \texttt{setter\_arg},\\ \texttt{ty},\\ \texttt{accessor\_pair}\end{array}\right) \xrightarrow{\text{ast}} [\texttt{getter}, \texttt{setter}]\end{array}$$

### ASTRule.Override

The function

$$\textit{build\_override}(\overbrace{\mathbb{PARSE}[\texttt{override}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle \texttt{override\_info} \rangle}^{\texttt{ast\_node}}$$

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

$$\textit{build\_override}(\overbrace{\texttt{override}(\epsilon)}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{None}$$

$$\textit{build\_override}(\overbrace{\texttt{override}(\texttt{"impdef"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \langle \text{Impdef} \rangle$$

$$\textit{build\_override}(\overbrace{\texttt{override}(\texttt{"implementation"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \langle \text{Implementation} \rangle$$

## 27.3 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.DeclareOneFunc
- TypingRule.SubprogramClash
- TypingRule.AddNewFunc
- TypingRule.AddSubprogram

### TypingRule.AnnotateAndDeclareFunc

The function

$$\mathit{annotate\_and\_declare\_func}(\overbrace{\mathbb{GSE}}^{\texttt{tenv}},\ \overbrace{\mathsf{func}}^{\texttt{func\_sig}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{tenv}} \times \underbrace{\mathsf{TTypeError}}_{\texttt{\#TE}} \times \overbrace{\mathsf{func}}^{\texttt{new\_func\_sig}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup$$

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 $(\texttt{tenv1}, \texttt{func\_sig\_f1}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- declaring the subprogram defined by `func_sig_f1` in `tenv1` with `ses_f1` as per TypingRule.DeclareOneFunc yields the environment `new_tenv` and new func node $\texttt{new\_func\_sig}^{/\!/\texttt{\#TE}}$;
- define `ses` as `ses_f1`.

### Formally

$$\frac{\mathit{annotate\_func\_sig}(\texttt{genv}, \texttt{func\_sig}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{func\_sig\_f1}, \texttt{ses\_f1}) \;/\!/\; \texttt{\#TE} \qquad \mathit{declare\_one\_func}(\texttt{tenv1}, \texttt{func\_sig\_f1}, \texttt{ses\_f1}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_func\_sig}) \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_and\_declare\_func}(\texttt{genv}, \texttt{func\_sig}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_func\_sig}, \overbrace{\texttt{ses\_f1}}^{\texttt{ses}})}$$

### TypingRule.AnnotateFuncSig

The function

$$\mathit{annotate\_func\_sig}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{func}}^{\texttt{func\_sig}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathsf{func}}^{\texttt{new\_func\_sig}} \times \overbrace{\mathsf{TSideEffect}}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the signature of a function definition `func_sig` in the global static environment `genv`, yielding a new function definition `new_func_sig`, a modified static environment `new_tenv`, and an inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

### 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 27.2 shows a subprogram with an associated recursion limit.

Listing 27.2: 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 27.3 is ill-typed, since the recursion limit expression `W` is not a constrained expression.

Listing 27.3: 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 all instances of ReadLocal and local write side effect descriptor removed;
- `new_func_sig` is `func_sig` with the annotated parameters `params`, annotated arguments `args`, annotated return type `return_type`, and `recurse_limit` as its recursion limit.

**Formally**

$$\frac{\begin{array}{c}\mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv} \\ \mathit{annotate\_limit\_expr}(\texttt{tenv1}, \texttt{func\_sig.recurse\_limit}) \xrightarrow{\text{type}} (\texttt{recurse\_limit}, \texttt{ses\_recurse\_limit}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_params}(\texttt{tenv}, \texttt{func\_sig.parameters}, (\texttt{tenv}, [\,])) \xrightarrow{\text{type}} \\ (\texttt{tenv\_with\_params}, \texttt{ses\_with\_params}, \texttt{params}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_param\_decls}(\texttt{tenv}, \texttt{func\_sig}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_args}(\texttt{tenv\_with\_params}, \texttt{func\_sig.args}, (\texttt{tenv\_with\_params}, [\,]), \texttt{ses\_with\_params}) \xrightarrow{\text{type}} \\ (\texttt{tenv\_with\_args}, \texttt{ses\_with\_args}, \texttt{args}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_return\_type}(\texttt{tenv\_with\_args}, \texttt{tenv\_with\_params}, \texttt{func\_sig.return\_type}, \texttt{ses\_with\_args}) \xrightarrow{\text{type}} \\ (\texttt{new\_tenv}, \texttt{return\_type}, \texttt{ses\_with\_return}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_with\_return} \setminus \{\texttt{s} \mid \texttt{config\_dom}(\texttt{s}) \in \{\texttt{ReadLocal}, \texttt{WriteLocal}\}\} \\ \texttt{new\_func\_sig} := \left\{\begin{array}{rcl} \text{name} & : & \texttt{func\_sig.name}, \\ \text{parameters} & : & \texttt{parameters}, \\ \text{args} & : & \texttt{args}, \\ \text{body} & : & \texttt{func\_sig.body}, \\ \text{return\_type} & : & \texttt{return\_type}, \\ \text{subprogram\_type} & : & \texttt{func\_sig.subprogram\_type}, \\ \text{recurse\_limit} & : & \texttt{recurse\_limit} \\ \text{builtin} & : & \texttt{func\_sig.builtin} \end{array}\right\}\end{array}}{\mathit{annotate\_func\_sig}(\texttt{genv}, \texttt{func\_sig}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_func\_sig}, \texttt{ses})}$$

### TypingRule.AnnotateParams

The function

$$\mathit{annotate\_params}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\texttt{params}}, (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{acc}})) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{tenv\_with\_params}} \times \overbrace{\mathsf{identifier} \times \mathsf{ty}}^{\texttt{params1}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates each parameter in `params` with respect to `tenv`, and declares it in environment `new_tenv`. It returns the updated environment `tenv_with_params` and the annotated parameters `params1`, together with any annotated parameters already in the accumulator `acc`. Otherwise, the result is a type error.

### Example: Annotating Parameters

In Listing 27.4, 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 27.4: 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 $(\texttt{x}, \texttt{ty\_opt})$ as its `head` and `params'` as its `tail`;
  - applying *annotate_one_param* to the parameter $(\texttt{x}, \texttt{ty\_opt})$ with `tenv` and `new_tenv` yields the pair `new_tenv'` and $\texttt{ty}^{\#\texttt{TE}}$;
  - define `acc'` as the concatenation of `acc` and the pair $(\texttt{x}, \texttt{ty})$;
  - applying *annotate_params* to `params'` with `tenv`, `new_tenv'`, and `acc'` yields `tenv_with_params` and `params1`.

### Formally

EMPTY

$$\mathit{annotate\_params}(\texttt{tenv}, \overbrace{[\,]}^{\texttt{params}}, (\texttt{new\_tenv}, \texttt{acc})) \xrightarrow{\texttt{type}} (\overbrace{\texttt{new\_tenv}}^{\texttt{tenv\_with\_params}}, \overbrace{\texttt{acc}}^{\texttt{params1}})$$

NON_EMPTY

$$\frac{\begin{array}{c}\texttt{params} = [(\texttt{x}, \texttt{ty\_opt})] + \texttt{params'} \\ \mathit{annotate\_one\_param}(\texttt{tenv}, \texttt{new\_tenv}, (\texttt{x}, \texttt{ty\_opt})) \xrightarrow{\texttt{type}} (\texttt{new\_tenv'}, \texttt{ty}) \;/\!/\; \#\texttt{TE} \\ \texttt{acc'} := \texttt{acc} + [(\texttt{x}, \texttt{ty})] \\ \mathit{annotate\_params}(\texttt{tenv}, \texttt{params'}, (\texttt{new\_tenv'}, \texttt{acc'})) \xrightarrow{\texttt{type}} (\texttt{tenv\_with\_params}, \texttt{params1}) \;/\!/\; \#\texttt{TE}\end{array}}{\mathit{annotate\_params}(\texttt{tenv}, \texttt{params}, (\texttt{new\_tenv}, \texttt{acc})) \xrightarrow{\texttt{type}} (\texttt{tenv\_with\_params}, \texttt{params1})}$$

### TypingRule.AnnotateOneParam

The function

$$\mathit{annotate\_one\_param}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}, \overbrace{(\mathsf{identifier} \times \overbrace{\langle \texttt{ty} \rangle}^{\texttt{ty\_opt}})}^{\texttt{x}}) \longrightarrow \underbrace{(\overbrace{\mathbb{SE}}^{\texttt{new\_tenv'}} \times \overbrace{\texttt{ty}}^{\texttt{ty}})}_{\mathsf{TTypeError}}{}^{\#\texttt{TE}} \cup$$

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 27.5, 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 27.5: 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 27.6, the declaration of the parameter `A` is illegal, since `A` is also declared as a global storage element.

Listing 27.6: 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 ⟨`ty'`⟩, which is not the unconstrained integer type;
    - annotating `ty'` in `tenv` yields `ty`$^{\#TE}_{\#TE}$.
- checking that `x` is not defined in `new_tenv` yields `TRUE`$^{\#TE}_{\#TE}$;
- checking that `ty` is a constrained integer in `new_tenv` via *check_constrained_integer* yields `TRUE`$^{\#TE}_{\#TE}$;
- adding the local storage element given by the identifier `x`, type `ty`, and local declaration keyword LDK_Let in `new_tenv` yields `new_tenv'`.

**Formally**

$$\frac{\begin{array}{c}\text{TYPE\_PARAMETERIZED}\\ (\texttt{ty\_opt} = \mathsf{None} \lor \texttt{ty\_opt} = \langle \mathsf{unconstrained\_integer} \rangle)\\ \texttt{ty} := \mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{x}))\\ \text{******** common suffix ********}\\ \mathit{check\_var\_not\_in\_env}(\texttt{new\_tenv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE}\\ \mathit{check\_constrained\_integer}(\texttt{new\_tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE}\\ \mathit{add\_local}(\texttt{new\_tenv}, \texttt{x}, \texttt{ty}, \mathsf{LDK\_Let}) \xrightarrow{\text{type}} \texttt{new\_tenv'}\end{array}}{\mathit{annotate\_one\_param}(\texttt{tenv}, \texttt{new\_tenv}, (\texttt{x}, \texttt{ty\_opt})) \xrightarrow{\text{type}} (\texttt{new\_tenv'}, \texttt{ty})}$$

$$\frac{\begin{array}{c}\text{TYPE\_ANNOTATED}\\ \texttt{ty'} \neq \mathsf{unconstrained\_integer} \qquad \mathit{annotate\_type}(\texttt{FALSE}, \texttt{tenv}, \texttt{ty'}) \xrightarrow{\text{type}} \texttt{ty} \;//\; \texttt{\#TE}\\ \text{******** common suffix ********}\\ \mathit{check\_var\_not\_in\_env}(\texttt{new\_tenv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE}\\ \mathit{check\_constrained\_integer}(\texttt{new\_tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE}\\ \mathit{add\_local}(\texttt{new\_tenv}, \texttt{x}, \texttt{ty}, \mathsf{LDK\_Let}) \xrightarrow{\text{type}} \texttt{new\_tenv'}\end{array}}{\mathit{annotate\_one\_param}(\texttt{tenv}, \texttt{new\_tenv}, (\texttt{x}, \langle \texttt{ty'} \rangle)) \xrightarrow{\text{type}} (\texttt{new\_tenv'}, \texttt{ty})}$$

### TypingRule.CheckParamDecls

The function

$$\mathit{check\_param\_decls}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{func}}^{\text{func\_sig}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

checks the validity of the parameters declared in `func_sig`.

### Example: Checking Parameter Declarations

In Listing 27.10, the list of extracted parameters `B, C, D, E, F, G` is equal to the list of declared parameters.

The specification in Listing 27.7 is ill-typed, since the list of extracted `D, A, B, C` while the list of declared parameters is `A, B, C, D`.

Listing 27.7: 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`$^{/\!/\texttt{\#TE}}$;
- 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`$^{/\!/\texttt{\#TE}}$.

### Formally

$$\frac{\begin{array}{c}\mathit{extract\_parameters}(\mathsf{tenv}, \texttt{func\_sig}) \xrightarrow{\mathsf{type}} \texttt{inferred\_parameters} \;/\!/\; \texttt{\#TE} \\ \texttt{declared\_parameters} := [(\mathsf{p}, \_) \in \texttt{func\_sig}.\mathsf{parameters} : \mathsf{p}] \\ \mathit{check}(\texttt{inferred\_parameters} = \texttt{declared\_parameters}, \texttt{TE\_BSPD}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{check\_param\_decls}(\mathsf{tenv}, \texttt{func\_sig}) \xrightarrow{\mathsf{type}} \mathsf{b}}$$

### TypingRule.ExtractParameters

The function

$$\mathit{extract\_parameters}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{func}}^{\texttt{func\_sig}}) \longrightarrow \overbrace{\mathsf{identifier}^*}^{\texttt{unique\_parameters}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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: Extracting Parameters from Subprogram Signatures

The list of parameters extracted from the signature of `parameters_of_expressions` in Listing 27.10 is `B, C, D, E, F, G`.

### 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 $\texttt{t}_{1..k}$;
  - applying *parameters_of_ty* to tenv and `types`[i], for every $\texttt{i} = 1..k$, yields the list of parameter names $\texttt{params}_\texttt{i}{}^{/\!/\texttt{\#TE}}$;

* define `all_parameters` as the concatenation of lists $\texttt{params}_\texttt{i}$, for every $\texttt{i} = 1..k$;
* define `unique_parameters` as the list of parameters in `all_parameters`, with repeating parameters removed.

**Formally**

$$\text{EMPTY}\quad \frac{\mathit{func\_sig\_types}(\texttt{func\_sig}) \xrightarrow{\text{type}} [\,]}{\mathit{extract\_parameters}(\texttt{tenv}, \texttt{func\_sig}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\texttt{unique\_parameters}}}$$

$$\text{NON\_EMPTY}\quad \frac{\begin{array}{c}\mathit{func\_sig\_types}(\texttt{func\_sig}) \xrightarrow{\text{type}} \texttt{t}_{1..k} \\ \texttt{i} = 1..k : \mathit{parameters\_of\_ty}(\texttt{tenv}, \texttt{types[i]}) \xrightarrow{\text{type}} \texttt{params}_\texttt{i} \;\; /\!/ \;\texttt{\#TE} \\ \texttt{all\_parameters} := \texttt{params}_1 + \ldots + \texttt{params}_k \\ \texttt{unique\_parameters} := \mathit{unique}(\texttt{all\_parameters})\end{array}}{\mathit{extract\_parameters}(\texttt{tenv}, \texttt{func\_sig}) \xrightarrow{\text{type}} \texttt{unique\_parameters}}$$

### TypingRule.FuncSigTypes

The function

$$\mathit{func\_sig\_types}(\overbrace{\mathsf{func}}^{\texttt{func\_sig}}) \longrightarrow \overbrace{\mathsf{ty}^*}^{\texttt{tys}}$$

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 27.8 shows the list of signature types for a procedure and a function in comments above their declarations.

Listing 27.8: 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

$$\frac{\texttt{return\_type} := \begin{cases} [\texttt{ty'}] & \text{if } \texttt{func\_sig}.\texttt{return\_type} = \langle \texttt{ty'} \rangle \\ [\,] & \text{if } \texttt{func\_sig}.\texttt{return\_type} = \texttt{None} \end{cases} \qquad \texttt{arg\_types} := [(\_, \texttt{ty'}) \in \texttt{func\_sig}.\texttt{args} : \texttt{ty'}]}{\mathit{func\_sig\_types}(\texttt{func\_sig}) \xrightarrow{\text{type}} \overbrace{\texttt{return\_type} + \texttt{arg\_types}}^{\text{tys}}}$$

### TypingRule.ParametersOfTy

The function

$$\mathit{parameters\_of\_ty}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{identifier}^*}^{\text{ids}} \cup \overbrace{\text{TTypeError}}^{\#\text{TE}}$$

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 27.9 are as follows:

| Type | Parameters |
|---|---|
| `bits(A)` | [A] |
| `(bits(B), bits(C))` | [B, C] |
| `integer{D..E}` | [D, E] |
| `real` | [ ] |
| `integer` | [ ] |

Listing 27.9: 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 $\texttt{ids}^{/\!/\texttt{\#TE}}$.
- 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 $\texttt{ty}_i$ in `tys` yields $\texttt{ids}_i{}^{/\!/\texttt{\#TE}}$;
  - `ids` is the concatenation of all the $\texttt{ids}_i$.
- 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 $\texttt{c}_i$ in `cs` yields $\texttt{ids}_i{}^{/\!/\texttt{\#TE}}$;
  - `ids` is the concatenation of all the $\texttt{ids}_i$.
- 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**

$$\text{TBITS}\quad \frac{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \mathtt{ids}}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathsf{T\_Bits}(\mathsf{e}, \_)) \xrightarrow{\text{type}} \mathtt{ids}}$$

$$\text{TTUPLE}\quad \frac{\mathtt{ty}_i \in \mathtt{tys} : \mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathtt{ty}_i) \xrightarrow{\text{type}} \mathtt{ids}_i \;/\!/\; \#\mathsf{TE} \qquad \mathtt{ids} := \mathtt{ids}_1 + \ldots + \mathtt{ids}_{|\mathtt{tys}|}}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathsf{T\_Tuple}(\mathtt{tys})) \xrightarrow{\text{type}} \mathtt{ids}}$$

$$\text{TINT\_CONSTRAINED\_PRECISE}\quad \frac{\mathtt{c}_i \in \mathtt{cs} : \mathit{parameters\_of\_constraint}(\mathsf{tenv}, \mathtt{c}_i) \xrightarrow{\text{type}} \mathtt{ids}_i \;/\!/\; \#\mathsf{TE} \qquad \mathtt{ids} := \mathtt{ids}_1 + \ldots + \mathtt{ids}_{|\mathtt{tys}|}}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs}, \mathtt{TRUE}))) \xrightarrow{\text{type}} \mathtt{ids}}$$

$$\text{OTHER}\quad \frac{\begin{array}{ll} \mathit{ast\_label}(\mathtt{ty}) \in \{\mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Bool}, \mathsf{T\_Array}, \mathsf{T\_Named}\} & \vee \\ \mathtt{ty} = \mathsf{unconstrained\_integer} & \vee \\ \mathtt{ty} = \mathsf{T\_Int}(\mathsf{WellConstrained}(\_, \mathtt{FALSE})) & \vee \\ \mathtt{ty} = \mathsf{T\_Int}(\mathsf{Parameterized}(\_)) & \end{array}}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathtt{ty}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\mathtt{ids}}}$$

$$\text{ERROR}\quad \frac{\mathit{is\_structured}(\mathtt{ty}) \vee \mathit{ast\_label}(\mathtt{ty}) = \mathsf{T\_Enum}}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathtt{ty}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathsf{TE\_BSPD})}$$

### TypingRule.ParametersOfExpr

The function

$$\mathit{parameters\_of\_expr}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{identifier}^*}^{\mathtt{ids}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

extracts the list of parameters appearing in the expression `e`. It assumes that `e` appears as $\mathsf{T\_Bits}(\mathsf{e}, \_)$ 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 27.10, extracting parameters from expressions appearing in the types of signature arguments (starting with the return type) yield the following lists of parameters:

| Expression | Parameters |
|---|---|
| `A` | [ ] |
| `B` | [B] |
| `C` | [C] |
| `-D` | [D] |
| `E+F` | [E, F] |
| `(G)` | [G] |
| `if H == 0 then I else J` | [H, I, J] |

Notice that, since `A` is declared as a global storage element, it is not extracted as a parameter for the expression `A` appearing in the return type.

Listing 27.10: 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 27.11 shows examples of expressions that are illegal in types of arguments.

Listing 27.11: 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`$^{//\texttt{\#TE}}$;
  - applying *parameters_of_expr* to `e2` in `tenv` yields `ids2`$^{//\texttt{\#TE}}$;
  - 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`$^{//\texttt{TE\_BSPD}}$;
  - view `es` as the singleton list for the expression `e`;
  - applying *parameters_of_expr* to `e` `ids`$^{//\texttt{\#TE}}$.
- 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'`$^{//\texttt{\#TE}}$;
  - applying *parameters_of_expr* to `e1` in `tenv` yields `ids1`$^{//\texttt{\#TE}}$;
  - applying *parameters_of_expr* to `e2` in `tenv` yields `ids2`$^{//\texttt{\#TE}}$;
  - 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

$$\frac{\mathit{is\_undefined}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\text{type}} \mathtt{b} \qquad \mathtt{ids} := \mathsf{choice}(\mathtt{b}, [\mathtt{x}], [\,])}{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{E\_Var}(\mathtt{x})) \xrightarrow{\text{type}} \mathtt{ids}}$$

EUNOP

$$\frac{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{ids} \;//\; \mathtt{\#TE}}{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{E\_Unop}(\_, \mathtt{e1})) \xrightarrow{\text{type}} \mathtt{ids}}$$

EBINOP

$$\frac{\begin{array}{l}\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{ids1} \;//\; \mathtt{\#TE} \\ \mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\text{type}} \mathtt{ids2} \;//\; \mathtt{\#TE}\end{array}}{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{E\_Binop}(\_, \mathtt{e1}, \mathtt{e2})) \xrightarrow{\text{type}} \overbrace{\mathtt{ids1} + \mathtt{ids2}}^{\mathtt{ids}}}$$

ETUPLE

$$\frac{\mathit{check}(|\texttt{es}| = 1, \texttt{TE\_BSPD}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \qquad \texttt{es} \overset{\text{is}}{=} [\texttt{e}] \qquad \mathit{parameters\_of\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} \texttt{ids} \;/\!/\; \texttt{\#TE}}{\mathit{parameters\_of\_expr}(\texttt{tenv}, \texttt{E\_Tuple}(\texttt{es})) \xrightarrow{\text{type}} \texttt{ids}}$$

ECOND

$$\frac{\mathit{parameters\_of\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} \texttt{ids'} \;/\!/\; \texttt{\#TE} \qquad \mathit{parameters\_of\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{ids1} \;/\!/\; \texttt{\#TE} \qquad \mathit{parameters\_of\_expr}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{ids2} \;/\!/\; \texttt{\#TE}}{\mathit{parameters\_of\_expr}(\texttt{tenv}, \texttt{E\_Cond}(\texttt{e}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{type}} \overbrace{\texttt{ids'} + \texttt{ids1} + \texttt{ids2}}^{\texttt{ids}}}$$

OTHER

$$\frac{\mathit{ast\_label}(\texttt{e}) \notin \{\texttt{E\_Var}, \texttt{E\_Unop}, \texttt{E\_Binop}, \texttt{E\_Tuple}\}}{\mathit{parameters\_of\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} \texttt{TypeError}(\texttt{TE\_BSPD})}$$

### TypingRule.ParametersOfConstraint

The function

$$\mathit{parameters\_of\_constraint}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{int\_constraint}}^{\texttt{c}}) \xrightarrow{\text{type}} \overbrace{\texttt{identifier}^*}^{\texttt{ids}}$$

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 27.9, the list of parameters extracted from the constraint `{D..E}` is $[\texttt{D}, \texttt{E}]$.
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 an range constraint, that is, Constraint_Range(`e1`, `e2`);
  - applying *parameters_of_expr* to `e1` in `tenv` yields `ids1`;
  - applying *parameters_of_expr* to `e2` in `tenv` yields `ids2`;
  - `ids` is the concatenation of `ids1` and `ids2`.

**Formally**

EXACT

$$\frac{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{ids}}{\mathit{parameters\_of\_constraint}(\mathsf{tenv}, \mathsf{Constraint\_Exact}(\mathtt{e})) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

RANGE

$$\frac{\begin{array}{c}\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{ids1} \\ \mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{ids2}\end{array}}{\mathit{parameters\_of\_constraint}(\mathsf{tenv}, \mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2})) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{ids1} + \mathtt{ids2}}^{\mathtt{ids}}}$$

### 27.3.1 Example

In Listing 27.4, 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 |
|---|---|---|
| `bits(A)` | {`A`} | `A` is a variable expression and `A` is not defined in the environment. |
| `bits(W)` | ∅ | `W` is defined in the environment. |
| `bits(A+B)` | {`A`, `B`} | `A` and `B` are variables and neither is defined in the environment. |

#### TypingRule.AnnotateArgs

The function

$$\mathit{annotate\_args}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\mathtt{args}}, (\overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}} \times \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\mathtt{acc}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses\_in}})) \longrightarrow (\overbrace{\mathbb{SE}}^{\mathtt{tenv\_with\_args}} \times \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\mathtt{new\_args}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

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 27.4, 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_tenv', ty', ses_ty)` // #TE;
  - define `acc'` as the concatenation of `acc` and the pair `(x, ty')`;
  - applying *annotate_args* to `args'` with `tenv`, `new_tenv'`, and `acc'` yields `(tenv_with_args, new_args,`
  - define `ses` as the union of `ses_ty` and `new_ses`.

### Formally

EMPTY

$$\mathit{annotate\_args}(\mathtt{tenv}, \overbrace{[\,]}^{\mathtt{args}}, (\mathtt{new\_tenv}, \mathtt{acc}), \mathtt{ses\_in}) \xrightarrow{\mathsf{type}} (\overbrace{\mathtt{new\_tenv}}^{\mathtt{tenv\_with\_args}}, \overbrace{\mathtt{acc}}^{\mathtt{new\_args}}, \overbrace{\mathtt{ses\_in}}^{\mathtt{ses}})$$

NON_EMPTY

$$\frac{\begin{array}{c}\mathtt{args} = [(\mathtt{x}, \mathtt{ty})] + \mathtt{args'} \\ \mathit{annotate\_one\_arg}(\mathtt{tenv}, \mathtt{new\_tenv}, (\mathtt{x}, \mathtt{ty})) \xrightarrow{\mathsf{type}} (\mathtt{new\_tenv'}, \mathtt{ty'}, \mathtt{ses\_ty}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{acc'} := \mathtt{acc} + [(\mathtt{x}, \mathtt{ty'})] \\ \mathit{annotate\_args}(\mathtt{tenv}, \mathtt{args'}, (\mathtt{new\_tenv'}, \mathtt{acc'}), \mathtt{ses\_in}) \xrightarrow{\mathsf{type}} (\mathtt{tenv\_with\_args}, \mathtt{new\_args}, \mathtt{new\_ses}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_ty} \cup \mathtt{new\_ses}\end{array}}{\mathit{annotate\_args}(\mathtt{tenv}, \mathtt{args}, (\mathtt{new\_tenv}, \mathtt{acc}), \mathtt{ses\_in}) \xrightarrow{\mathsf{type}} (\mathtt{tenv\_with\_args}, \mathtt{new\_args}, \mathtt{ses})}$$

### TypingRule.AnnotateOneArg

The function

$$\mathit{annotate\_one\_arg}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}}, (\overbrace{\mathsf{identifier}}^{\mathtt{x}} \times \overbrace{\mathsf{ty}}^{\mathtt{ty}})) \longrightarrow (\overbrace{\mathbb{SE}}^{\mathtt{new\_tenv'}} \times \overbrace{\mathsf{ty}}^{\mathtt{ty'}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

annotates the argument given by the identifier `x` and the type `ty` with respect to `tenv` and then declares the parameter `x` in environment `new_tenv`. The result is the updated environment `new_tenv'`, annotated argument type `ty'`, and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

### Example: Annotating a Subprogram Argument

The specification in Listing 27.12 shows well-typed argument declarations and demonstrates how they exist as local storage elements in the subprogram body.

Listing 27.12: Well-typed subprogram arguments

```
func arguments(b: boolean, i: integer, r: real)
begin
    - = b;
    - = i;
    - = r;
end;
```

The specification in Listing 27.13 is illegal, since arguments cannot share a name with global storage elements, `b` in this example.

Listing 27.13: 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 27.14 is illegal, since arguments cannot be typed as collection types.

Listing 27.14: An ill-typed subprogram argument

```
type MyCollection of collection;

// Illegal: collection types are not allowed as arguments.
func arguments(b: MyCollection)
begin pass; end;
```

### Prose

All of the following apply:

- annotating the type `ty` in `tenv` yields $(\texttt{ty'}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- determining whether `ty` is not a collection type in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- checking that `x` is not defined in `new_tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- adding the local storage element given by the identifier `x`, type `ty'`, and local declaration keyword LDK_Let in `new_tenv` yields `new_tenv'`.

**Formally**

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} (\texttt{ty'}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_is\_not\_collection}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_var\_not\_in\_env}(\texttt{new\_tenv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{add\_local}(\texttt{new\_tenv}, \texttt{x}, \texttt{ty'}, \texttt{LDK\_Let}) \xrightarrow{\text{type}} \texttt{new\_tenv'}\end{array}}{\mathit{annotate\_one\_arg}(\texttt{tenv}, \texttt{new\_tenv}, (\texttt{x}, \texttt{ty})) \xrightarrow{\text{type}} (\texttt{new\_tenv'}, \texttt{ty'}, \texttt{ses})}$$

**TypingRule.AnnotateReturnType**

The function

$$\mathit{annotate\_return\_type}(\overbrace{\mathbb{SE}}^{\texttt{tenv\_with\_params}}, \overbrace{\mathbb{SE}}^{\texttt{tenv\_with\_args}}, \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{return\_type}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_in}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathsf{ty}}^{\texttt{new\_return\_type}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the optional return type `return_type` in the context of the static environment `tenv_with_params`, where all parameters have been declared, and the set of side effect descriptors `ses_in`. The result is `new_tenv`, which is the input `tenv_with_args` (where all parameters and arguments have been declared) with the optional annotated return type `new_return_type` added and the inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

**Example: Annotating Subprogram Return Types**

In Listing 23.25, 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 27.15 is ill-typed, since collection types are not allowed as return types.

Listing 27.15: An ill-typed return type

```
type MyCollection of collection;

// Illegal: collection types are not allowed as return types.
func returns_value() => MyCollection
begin
    return ARBITRARY: MyCollection;
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`)$^{/\!/\texttt{\#TE}}$;
  - determining whether `ty'` is not a collection type in `tenv` yields `TRUE`$^{/\!/\texttt{\#TE}}$;
  - `new_return_type` is ⟨`ty'`⟩;
  - `new_tenv` is `tenv_with_args` with its local environment updated by binding its return_type field to `new_return_type`;
  - define `ses` as the union of `ses_in` and `ses_ty`.

**Formally**

NO_RETURN_TYPE

$$\mathit{annotate\_return\_type}(\texttt{tenv\_with\_params}, \texttt{tenv\_with\_args}, \overbrace{\texttt{None}}^{\texttt{return\_type}}, \texttt{ses\_in}) \xrightarrow{\texttt{type}} (\overbrace{\texttt{tenv\_with\_args}}^{\texttt{new\_tenv}}, \overbrace{\texttt{None}}^{\texttt{new\_return\_type}}, \overbrace{\texttt{ses\_in}}^{\texttt{ses}})$$

HAS_RETURN_TYPE

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\texttt{tenv\_with\_params}, \texttt{ty}) \xrightarrow{\texttt{type}} (\texttt{ty'}, \texttt{ses\_ty}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_is\_not\_collection}(\texttt{tenv}, \texttt{ty'}) \xrightarrow{\texttt{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{new\_return\_type} := \langle \texttt{ty'} \rangle \\ \texttt{new\_tenv} := (G^{\texttt{tenv\_with\_args}}, L^{\texttt{tenv\_with\_args}}[\text{return\_type} \mapsto \texttt{new\_return\_type}]) \\ \texttt{ses} := \texttt{ses\_in} \cup \texttt{ses\_ty}\end{array}}{\mathit{annotate\_return\_type}(\texttt{tenv\_with\_params}, \texttt{tenv\_with\_args}, \overbrace{\langle \texttt{ty} \rangle}^{\texttt{return\_type}}, \texttt{ses\_in}) \xrightarrow{\texttt{type}} (\texttt{new\_tenv}, \texttt{new\_return\_type}, \texttt{ses})}$$

### TypingRule.DeclareOneFunc

The function

$$\mathit{declare\_one\_func}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{func}}^{\texttt{func\_sig}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_func\_sig}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\texttt{func}}^{\texttt{new\_func\_sig}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

checks that a subprogram defined by `func_sig` and associated with the set of side effect descriptors `ses_func_sig` can be added to the static environment `tenv`, resulting in

an annotated function definition `new_func_def` and new static environment `new_tenv`. Otherwise, the result is a type error.

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,

$$\begin{aligned}\texttt{func\_sig} := \{ \quad & \text{name} : \texttt{name}, \\ & \text{parameters} : \texttt{p}, \\ & \text{args} : \texttt{args}, \\ & \text{body} : \texttt{bd}, \\ & \text{return\_type} : \texttt{t}, \\ & \text{subprogram\_type} : \text{sub\_program\_type}, \\ & \text{builtin} : \texttt{b} \\ \} \quad & ;\end{aligned}$$

- adding a new subprogram with `name`, `args`, and sub_program_type to `tenv` yields the new environment `tenv1` and new name `name'`[//#TE];
- checking that `name'` is not already declared in the global environment of `tenv1` yields TRUE[//#TE];
- `func_sig1` is `func_sig` with `name` substituted by `name1`;
- define `init_ses` as the union of `ses_func_sig` and the singleton set for a recursive call side effect descriptor for `name'`;
- adding a subprogram with name `name'`, definition `func_sig1`, and set of side effect descriptors `init_ses` to `tenv1` yields `new_tenv`[//#TE].

**Formally**

$$\frac{\begin{array}{l}\texttt{func\_sig} := \{\ \text{name} : \texttt{name},\ \text{parameters} : \texttt{p},\ \text{args} : \texttt{args},\ \text{body} : \texttt{bd},\ \text{return\_type} : \texttt{t},\ \text{subprogram\_type} : \mathit{sub\_program\_type},\ \text{builtin} : \texttt{b}\ \} \\ \mathit{add\_new\_func}(\texttt{tenv}, \texttt{name}, \texttt{args}, \mathit{sub\_program\_type}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{name'})\ /\!/\ \texttt{\#TE} \\ \mathit{check}\left(G^{\texttt{tenv1}}.\mathit{subprograms}(\texttt{name'}) = \bot, \texttt{TE\_IAD}\right) \longrightarrow \texttt{TRUE}\ /\!/\ \texttt{TE\_IAD} \\ \texttt{new\_func\_sig} := \{\ \text{name} : \texttt{name'},\ \text{parameters} : \texttt{p},\ \text{args} : \texttt{args},\ \text{body} : \texttt{bd},\ \text{return\_type} : \texttt{t},\ \text{subprogram\_type} : \mathit{sub\_program\_type},\ \text{builtin} : \texttt{b}\ \} \\ \texttt{init\_ses} := \texttt{ses\_func\_sig} \cup \{\mathit{RecursiveCall}(\texttt{name'})\} \\ \mathit{add\_subprogram}(\texttt{tenv1}, \texttt{name'}, \texttt{func\_sig1}, \texttt{init\_ses}) \xrightarrow{\text{type}} \texttt{new\_tenv}\ /\!/\ \texttt{\#TE}\end{array}}{\mathit{declare\_one\_func}(\texttt{tenv}, \texttt{func\_sig}, \texttt{ses\_func\_sig}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_func\_sig})}$$

### TypingRule.SubprogramClash

The function

$$\mathit{subprogram\_clash}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{S}}^{\texttt{name}}, \overbrace{\mathit{sub\_program\_type}}^{\texttt{subpgm\_type}}, \overbrace{\mathit{ty}^*}^{\texttt{formal\_types}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathit{TTypeError}}^{\texttt{\#TE}}$$

checks whether the unique subprogram associated with `name` clashes with another subprogram that has subprogram type `subpgm_type` and list of formal types `formal_types`, yielding a Boolean value in `b`. Otherwise, the result is a type error.

The function assumes there exists a binding for `name` in the subprograms map of `tenv`.

### Example: Subprogram Clashing

In Listing 27.16, the getter for `X` and the setter for `X` do not clash, since they have non-clashing subprogram types (see TypingRule.SubprogramTypesClash).

In Listing 27.17, the getter for `X` and the function `X` clash, since their subprogram types clash and their signatures clash (see TypingRule.HasArgClash).

In Listing 27.18, the function `X` and the procedure `X` clash since their subprogram types clash and their signatures clash.

### Prose

All of the following apply:

- the identifier `name` is bound to the func AST node `other_func_sig` in the subprograms map of the static global environment of `tenv` (ignoring the associated side effect descriptors);
- applying *subprogram_types_clash* to the subprogram type of `other_func_sig` (`other_func_sig`.sub_program_type) and `subpgm_type` yields `TRUE`$^{/\!/\texttt{FALSE}}$ (that is, if both subprogram types are `ST_Getter` or both are `ST_Setter` then the subprogram types are considered to be non-clashing and the entire rule short-circuits to `FALSE`);
- determining whether there is an argument clash between `formal_types` and the formal arguments of `other_func_sig` (`other_func_sig`.args) in `tenv` yields `b`$^{/\!/\#\texttt{TE}}$.

### Formally

$$\frac{\begin{array}{c} G^{\texttt{tenv}}.\mathsf{subprograms}(\texttt{name}) = (\texttt{other\_func\_sig}, \_) \\ \mathit{subprogram\_types\_clash}(\texttt{other\_func\_sig.sub\_program\_type}, \texttt{subpgm\_type}) \xrightarrow{\text{type}} \texttt{b1} \\ \mathit{has\_arg\_clash}(\texttt{formal\_types}, \texttt{other\_func\_sig.args}) \xrightarrow{\text{type}} \texttt{b2} \end{array}}{\mathit{subprogram\_clash}(\texttt{tenv}, \texttt{name}, \texttt{subpgm\_type}, \texttt{formal\_types}) \xrightarrow{\text{type}} \overbrace{\texttt{b1} \wedge \texttt{b2}}^{\texttt{b}}}$$

### TypingRule.SubprogramTypesClash

The function

$$\mathit{subprogram\_types\_clash}(\overbrace{\mathsf{sub\_program\_type}}^{\texttt{s1}}, \overbrace{\mathsf{sub\_program\_type}}^{\texttt{s2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

defines whether the subprogram types `s1` and `s2` clash, yielding the result in `b`.

### Example: Clashing Subprogram Types

The specification in Listing 27.16 contains an accessor for `X`. The subprogram type of its getter does not clash with the subprogram type of its setter.

Listing 27.16: Non-clashing subprogram types

```
accessor X() <=> value_in: bits (4)
begin
    getter
        return '1000';
    end;
    setter
        Unreachable();
    end;
end;
```

The specification in Listing 27.17 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 27.17: Clashing subprogram types

```
accessor X() <=> value_in: bits (4)
begin
    getter
        return '1000';
    end;
    setter
        Unreachable();
    end;
end;

func X() // Illegal as accessor `X` also declared.
begin pass; end;
```

The specification in Listing 27.18 contains a function `X` and a procedure `X` whose subprogram types (`ST_Function` and `ST_Procedure`, respectively) clash.

Listing 27.18: 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

$$\frac{\texttt{b} := (\texttt{s1}, \texttt{s2}) \notin \left\{ \begin{array}{l} (\texttt{ST\_Getter}, \texttt{ST\_Setter}), \\ (\texttt{ST\_Setter}, \texttt{ST\_Getter}) \end{array} \right\}}{\mathit{subprogram\_types\_clash}(\texttt{subpgm\_type1}, \texttt{subpgm\_type2}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.AddNewFunc

The function

$$\mathit{add\_new\_func}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{typed\_identifier}^*}^{\texttt{formals}}, \overbrace{\mathsf{sub\_program\_type}}^{\texttt{subpgm\_type}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathbb{S}}^{\texttt{new\_name}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

ensures that the subprogram given by the identifier `name`, list of formals `formals`, and subprogram type `subpgm_type` has a unique name among all the potential subprograms that overload `name`. The result is the unique subprogram identifier `new_name`, which is used

to distinguish it in the set of overloaded subprograms (that is, other subprograms that share the same name) and the environment `new_tenv`, which is updated with `new_name`. Otherwise, the result is a type error.

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 28.5 contains two subprograms named `add_10` with the signatures `func add_10(x: integer) => integer` and `func add_10(x: real) => real`, which for brevity we will refer to by $I$ and $R$, respectively. These subprograms are added to the static environment based on the dependency ordering (see TypingRule.DeclDependencies). In this case, there is no ordering between these two subprograms. Assuming $I$ is processed before $R$, adding $I$ to the static environment, the static environment has overloaded_subprograms(`add_10`) $= \bot$ 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 $R$.

### 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

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = \bot \qquad \texttt{new\_tenv} := (G^{\text{tenv}}.\text{overloaded\_subprograms}[\texttt{name} \mapsto \{\texttt{name}\}], L^{\text{tenv}})}{\textit{add\_new\_func}(\text{tenv}, \texttt{name}, \texttt{formals}, \texttt{subpgm\_type}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \overbrace{\texttt{name}}^{\texttt{new\_name}})}$$

NAME_EXISTS

$$\frac{\begin{array}{c} G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = \texttt{other\_names} \\ k := |\texttt{other\_names}| \qquad \texttt{new\_name} := \texttt{name} + \texttt{"-"} + \texttt{string\_of\_nat}(k) \\ \texttt{formal\_types} := [(\texttt{id}, \texttt{t}) \in \texttt{formals} : \texttt{t}] \\ \left( \begin{array}{l} \texttt{name'} \in \texttt{other\_names} : \\ \textit{subprogram\_clash}(\text{tenv}, \texttt{name'}, \texttt{subpgm\_type}, \texttt{formal\_types}) \xrightarrow{\text{type}} \texttt{b}_{\texttt{name'}} \ \ /\!/ \ \texttt{\#TE} \end{array} \right) \\ \texttt{name'} \in \texttt{other\_names} : \textit{check}(\neg \texttt{b}_{\texttt{name'}}, \texttt{TE\_BSPD}) \xrightarrow{\text{type}} \texttt{TRUE} \ /\!/ \ \texttt{\#TE} \\ \texttt{new\_tenv} := (G^{\text{tenv}}.\text{overloaded\_subprograms}[\texttt{name} \mapsto \texttt{other\_names} \cup \{\texttt{new\_name}\}], L^{\text{tenv}}) \end{array}}{\textit{add\_new\_func}(\text{tenv}, \texttt{name}, \texttt{formals}, \texttt{subpgm\_type}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_name})}$$

# 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 in given by the relation *eval_ spec*, which is defined in SemanticsRule.EvalSpec.

## 28.1 Syntax

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

## 28.2 Abstract Syntax

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

### ASTRule.AST

The relation

$$\mathit{build\_ ast} : \overbrace{\mathbb{PARSE}[\mathtt{spec}]}^{\mathtt{parsed\_node}} \times \overbrace{\mathsf{spec}}^{\mathtt{ast\_node}}$$

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.

AST

$$\frac{\mathit{build\_ list}[\mathit{build\_ decl}](\mathtt{decls}) \xrightarrow{\mathsf{ast}} \mathtt{decls1} \qquad \mathit{concat}(\mathtt{decls1}) \xrightarrow{\mathsf{ast}} \mathtt{adecls}}{\mathit{build\_ ast}(\overbrace{\mathsf{spec}(\mathtt{decls} : \mathsf{list}^*(\mathtt{decl}))}^{\mathtt{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathtt{adecls}}^{\mathtt{ast\_node}}}$$

## 28.3 Typing Specifications

The untyped AST of an ASL specification consists of a list of global declarations. Typechecking 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

$$\mathit{type\_check\_ast}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \bigtimes ( \overbrace{\mathsf{decl}^*}^{\texttt{new\_decls}} \times \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} ) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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`$^{/\!/\texttt{\#TE}}$.
- for each `d` in `pragmas`, checking the global pragma `d` for correctness in the static environment `new_tenv` yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- `pragmas` is ignored;

**Formally**

$$\frac{\begin{array}{c}\mathit{override\_subprograms}(\texttt{decls}) \xrightarrow{\text{type}} \texttt{decls'} \;/\!/\; \texttt{\#TE} \\ \texttt{pragmas} := [\ \texttt{d} \mid \texttt{d} \in \texttt{decls'} \ \wedge \mathit{ast\_label}(\texttt{d}) = \texttt{D\_Pragma}\ ] \\ \texttt{others} := [\ \texttt{d} \mid \texttt{d} \in \texttt{decls'} \ \wedge \mathit{ast\_label}(\texttt{d}) \neq \texttt{D\_Pragma}\ ] \\ \mathit{build\_dependencies}(\texttt{others}) \xrightarrow{\text{type}} (\texttt{defs}, \texttt{depends}) \\ \texttt{rev\_deps} := [(a,b) \in \texttt{depends} : (b,a)] \qquad \mathit{SCC}(\texttt{defs}, \texttt{rev\_deps}) = \texttt{comps} \\ \texttt{orderedcomps} \in \mathit{topological\_ordering\_comps}(\texttt{comps}, \texttt{rev\_deps}) \\ \texttt{comp\_decls} := [\texttt{c} \in \texttt{orderedcomps} : \mathit{list\_set}(\texttt{c})] \\ \mathit{annotate\_decl\_comps}(\texttt{genv}, \texttt{comp\_decls}) \xrightarrow{\text{type}} (\texttt{new\_decls}, \texttt{new\_tenv}) \;/\!/\; \texttt{\#TE} \\ \texttt{d} \in \texttt{pragmas} : \mathit{check\_global\_pragma}(\texttt{new\_tenv}, \texttt{d}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{type\_check\_ast}(\texttt{genv}, \texttt{decls}) \xrightarrow{\text{type}} (\texttt{new\_decls}, \texttt{new\_tenv})}$$

**TypingRule.AnnotateDeclComps**

The function

$$\mathit{annotate\_decl\_comps}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{(\mathsf{decl}^*)^*}^{\texttt{comps}}) \longrightarrow (\ \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}} \times \overbrace{\mathsf{decl}^*}^{\texttt{new\_decls}}\ ) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a list of declaration components `comps` (a list of lists) in the global static environment `genv`, yielding the annotated list of declarations `new_decls` and modified global static environment `new_genv`. Otherwise, the result is a type error.

We note that a strongly-connected component containing just a single declaration may contain any kind of global declaration — a type declaration, a global storage declaration, or a subprogram declaration — whereas a strongly-connected component containing multiple declarations must be checked to contain only subprograms. This is because the only type of mutually-recursive declarations allows in ASL are between subprograms. The rules below handle these cases separately (SINGLE for single declarations and MUTUALLY_RECURSIVE for more than one declaration).

### 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 $(\texttt{d1}, \texttt{genv1})^{/\!/\texttt{\#TE}}$;
  - applying *annotate_ decl_ comps* to `comps1` in `genv1` yields $(\texttt{new\_genv}, \texttt{decls1})^{/\!/\texttt{\#TE}}$;
  - define `new_decls` as the list with `head` `d1` and `tail` `decls1`.
- All of the following apply (MUTUALLY_RECURSIVE):
  - `comps` is a list with `head` `comp` and `tail` `comps1`;
  - `comp` is a list with more than one declaration (that is, a list of mutually-recursive declarations);
  - applying *type_ check_ mutually_ rec* to `comp` in `genv` yields $(\texttt{decls1}, \texttt{genv1})^{/\!/\texttt{\#TE}}$;
  - applying *annotate_ decl_ comps* to `comps1` in `genv1` yields $(\texttt{new\_genv}, \texttt{decls2})^{/\!/\texttt{\#TE}}$;
  - define `new_decls` as the concatenation of `decls1` and `decls2`.

**Formally**

EMPTY

$$\mathit{annotate\_decl\_comps}(\texttt{genv}, \overbrace{[\,]}^{\texttt{comps}}) \longrightarrow (\overbrace{\texttt{genv}}^{\texttt{new\_genv}}, \overbrace{[\,]}^{\texttt{new\_decls}})$$

SINGLE

$$\frac{\texttt{comp} = [\texttt{d}] \qquad \mathit{typecheck\_decl}(\texttt{genv}, \texttt{d}) \xrightarrow{\text{type}} (\texttt{d1}, \texttt{genv1}) \;/\!/\; \texttt{\#TE} \qquad \mathit{annotate\_decl\_comps}(\texttt{genv1}, \texttt{comps1}) \xrightarrow{\text{type}} (\texttt{new\_genv}, \texttt{decls1}) \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_decl\_comps}(\texttt{genv}, \overbrace{[\texttt{comp}] + \texttt{comps1}}^{\texttt{comps}}) \longrightarrow (\texttt{new\_genv}, \overbrace{[\texttt{d1}] + \texttt{decls1}}^{\texttt{new\_decls}})}$$

MUTUALLY_RECURSIVE

$$\frac{|\texttt{comp}| > 1 \qquad \mathit{type\_check\_mutually\_rec}(\texttt{genv}, \texttt{comp}) \xrightarrow{\text{type}} (\texttt{decls1}, \texttt{genv1}) \;/\!/\; \texttt{\#TE} \qquad \mathit{annotate\_decl\_comps}(\texttt{genv1}, \texttt{comps1}) \xrightarrow{\text{type}} (\texttt{new\_genv}, \texttt{decls2}) \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_decl\_comps}(\texttt{genv}, \overbrace{[\texttt{comp}] + \texttt{comps1}}^{\texttt{comps}}) \longrightarrow (\texttt{new\_genv}, \overbrace{\texttt{decls1} + \texttt{decls2}}^{\texttt{new\_decls}})}$$

### TypingRule.TypeCheckMutuallyRec

The function

$$\mathit{type\_check\_mutually\_rec}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \longrightarrow (\overbrace{\mathsf{decl}^*}^{\texttt{new\_decls}} \times \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a list of mutually recursive declarations `decls` in the global static environment `genv`, yielding the annotated list of subprogram declarations `new_decls` and modified global static environment `new_genv`.

### Example: Checking Mutually-recursive Declarations

In Example: Typechecking a List of Global Declarations, the strongly-connected component
{Subprogram(`rotate0`), Subprogram(`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 {Other(`g`), 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;
```

### Prose

All of the following apply:

- checking that each declaration in `d` is a subprogram declaration yields `TRUE`$^{\#\texttt{TE\_BD}}$;
- applying *annotate_func_sig* to each node `f` in `genv`, where `D_Func(f)` is a declaration in `decls`, yields $(\texttt{tenv}_\texttt{f}, \texttt{d}_\texttt{f}, \texttt{ses}_\texttt{f})^{\#\texttt{TE}}$;
- define `env_and_fs1` as the list of pairs, each consisting of the local environment component of $\texttt{tenv}_\texttt{f}$ the annotated subprogram $\texttt{d}_\texttt{f}$, and the set of side effect descriptors $\texttt{ses}_\texttt{f}$, for each subprogram declaration `D_Func(f)` in `decls`;
- applying *declare_subprograms* to `genv` and `env_and_fs1` yields $(\texttt{genv2}, \texttt{env\_and\_fs2})^{\#\texttt{TE}}$;
- for tuple in `env_and_fs2` consisting of a local static environment, an element of func, and a set of side effect descriptors, $(\texttt{lenv2}, \texttt{f}, \texttt{ses}_\texttt{f})$, applying *annotate_subprogram* to `f` and $\texttt{ses}_\texttt{f}$ in the static environment $(\texttt{genv2}, \texttt{lenv2})$ yields $(\texttt{new}_\texttt{f}, \texttt{ses'}_\texttt{f})^{\#\texttt{TE}}$;
- define `new_decls` as the list of $\texttt{D\_Func}(\texttt{new}_\texttt{f})$ for all $(\_, \texttt{f}, \_)$ in `env_and_fs2`;
- define `sess` as the list of $(\texttt{new}_\texttt{f}, \texttt{ses'}_\texttt{f})$ for all $(\_, \texttt{f}, \_)$ in `env_and_fs2`;
- applying *propagate_recursive_calls_sess* on `sess` yields `sess_prop`;
- define `tenv2` as the environment with `genv2` as its static global environment and an empty static local environment;
- applying *add_subprogram_decls* to `tenv2` and `sess_prop` yields `tenv3`;
- define `new_tenv` as the global static environment of `tenv3`.

### Formally

$$\frac{\begin{array}{c}\texttt{d} \in \texttt{decls} : \mathit{check}(\mathit{ast\_label}(\texttt{d}) = \texttt{D\_Func}, \texttt{TE\_BD}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \#\texttt{TE} \\ \texttt{D\_Func}(\texttt{f}) \in \texttt{decls} : \mathit{annotate\_func\_sig}(\texttt{genv}, \texttt{f}) \xrightarrow{\text{type}} (\texttt{tenv}_\texttt{f}, \texttt{d}_\texttt{f}, \texttt{ses}_\texttt{f}) \;/\!/\; \#\texttt{TE} \\ \texttt{env\_and\_fs1} := [\texttt{D\_Func}(\texttt{f}) \in \texttt{decls} : (L^{\texttt{tenv}_\texttt{f}}, \texttt{d}_\texttt{f}, \texttt{ses}_\texttt{f})] \\ \mathit{declare\_subprograms}(\texttt{genv}, \texttt{env\_and\_fs1}) \xrightarrow{\text{type}} (\texttt{genv2}, \texttt{env\_and\_fs2}) \;/\!/\; \#\texttt{TE} \\ (\texttt{lenv2}, \texttt{f}, \texttt{ses}_\texttt{f}) \in \texttt{env\_and\_fs2} : \mathit{annotate\_subprogram}((\texttt{genv2}, \texttt{lenv2}), \texttt{f}, \texttt{ses}_\texttt{f}) \xrightarrow{\text{type}} (\texttt{new}_\texttt{f}, \texttt{ses'}_\texttt{f}) \;/\!/\; \#\texttt{TE} \\ \texttt{new\_decls} := [(\_, \texttt{f}, \_) \in \texttt{env\_and\_fs2} : \texttt{D\_Func}(\texttt{new}_\texttt{f})] \\ \texttt{sess} := [(\_, \texttt{f}, \_) \in \texttt{env\_and\_fs2} : (\texttt{new}_\texttt{f}, \texttt{ses'}_\texttt{f})] \\ \mathit{propagate\_recursive\_calls\_sess}(\texttt{sess}) \xrightarrow{\text{type}} \texttt{sess\_prop} \\ \texttt{tenv2} := (\texttt{genv2}, \emptyset_\lambda) \qquad \mathit{add\_subprogram\_decls}(\texttt{tenv2}, \texttt{sess\_prop}) \xrightarrow{\text{type}} \texttt{tenv3}\end{array}}{\mathit{type\_check\_mutually\_rec}(\texttt{genv}, \texttt{decls}) \xrightarrow{\text{type}} (\texttt{new\_decls}, \overbrace{G^{\texttt{tenv3}}}^{\texttt{new\_genv}})}$$

### TypingRule.CheckGlobalPragma

The function

$$\mathit{check\_global\_pragma}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{decl}}^{\texttt{d}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\texttt{TE}}$$

typechecks a global pragma declaration `d` in the global static environment `genv`, yielding `TRUE`. Otherwise, the result is a type error.

### Example: Checking Global Pragmas

The specification in Listing 28.3 shows a well-typed global pragma, whereas the specification in Listing 28.4 shows an ill-typed global pragma where the expression `x` is ill-typed, since the identifier `x` is undefined and the expression `(2==3.0)` uses the operator `==` without matching any operation.

Listing 28.3: A well-typed global pragma

```
var x = 5;
pragma good_pragma 1, (2==3), x;
```

Listing 28.4: An ill-typed global pragma

```
// Illegal: x is undefined and (2==3.0) does not match any operation.
pragma bad_pragma 1, (2==3.0), x;
```

### Prose

All of the following apply:

- `d` is a global pragma declaration with any identifier and expression list `args`. that is, `D_Pragma(_, args)`;
- applying *with_empty_local* to `genv` yields `tenv`;
- applying *annotate_exprs* to `args` in `tenv` yields `args'` $/\!/^{\#\texttt{TE}}$;
- `args'` is ignored;

### Formally

$$\frac{\mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv} \qquad \mathit{annotate\_exprs}(\texttt{tenv}, \texttt{args}) \xrightarrow{\text{type}} \texttt{args'} \;/\!/\; \#\texttt{TE}}{\mathit{check\_global\_pragma}(\texttt{genv}, \overbrace{\texttt{D\_Pragma}(\_, \texttt{args})}^{\texttt{d}}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

### TypingRule.DeclareSubprograms

The function

$$\mathit{declare\_subprograms}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{(\mathbb{LSE} \times \mathsf{func})^*}^{\texttt{env\_and\_fs}}) \longrightarrow \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}} \times \overbrace{(\mathbb{LSE} \times \mathsf{func} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{new\_env\_and\_fs}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

processes a list of pairs, each consisting of a local static environment and a subprogram declaration, `env_and_fs`, in the context of a global static environment `genv`, declaring each subprogram in the environment consisting of `genv` and the static local environment associated with each subprogram. The result is a modified global static environment `new_genv` and list of tuples `new_env_and_fs` consisting of local static environment, annotated func AST node, and sets of side effect descriptors.

### Example: Updating Static Environments for Subprogram Declarations

Applying *declare_subprograms* for the subprogram declarations in Listing 28.5 adds the following bindings to the overloaded_subprograms map

$$\begin{array}{rcl} \texttt{flip\_bits} & \mapsto & \{\texttt{flip\_bits}\} \\ \texttt{add\_10} & \mapsto & \{\texttt{add\_10}, \texttt{add\_10-1}\} \\ \texttt{factorial} & \mapsto & \{\texttt{factorial}\} \end{array}$$

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 28.5: Updating static environments for subprogram declarations

```
func flip_bits{N}(bv: bits(N)) => bits(N)
begin
    return Ones{N} XOR bv;
end;

func add_10(x: integer) => integer
begin
    return x + 10;
end;

func add_10(x: real) => real
begin
```

```
    return x + 10.0;
end;

func factorial(x: integer) => integer recurselimit 100
begin
    return if x == 0 then 1 else x * factorial(x - 1);
end;
```

### Prose

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

EMPTY

$$\mathit{declare\_subprograms}(\texttt{genv}, \overbrace{[\,]}^{\texttt{env\_and\_fs}}) \xrightarrow{\text{type}} (\overbrace{\texttt{genv}}^{\texttt{new\_genv}}, \overbrace{[\,]}^{\texttt{new\_env\_and\_fs}})$$

NON_EMPTY

$$\frac{\texttt{tenv} \overset{\text{is}}{=} (\texttt{genv}, \texttt{lenv}) \qquad \mathit{declare\_one\_func}(\texttt{tenv}, \texttt{f}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{f1}) \;/\!/\; \#TE \qquad \mathit{declare\_subprograms}(G^{\texttt{tenv1}}, \texttt{env\_and\_fs1}) \xrightarrow{\text{type}} (\texttt{new\_genv}, \texttt{env\_and\_fs2}) \;/\!/\; \#TE \qquad \texttt{new\_env\_and\_fs} := [(\texttt{lenv}, \texttt{f1}, \texttt{ses\_f})] + \texttt{env\_and\_fs2}}{\mathit{declare\_subprograms}(\texttt{genv}, \overbrace{[(\texttt{lenv}, \texttt{f}, \texttt{ses\_f})] + \texttt{env\_and\_fs1}}^{\texttt{env\_and\_fs}}) \xrightarrow{\text{type}} (\overbrace{\texttt{genv}}^{\texttt{new\_genv}}, \texttt{new\_env\_and\_fs})}$$

### TypingRule.AddSubprogramDecls

The function

$$\mathit{add\_subprogram\_decls}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{(\texttt{func} \times \mathcal{P}(\texttt{TSideEffect}))^*}^{\texttt{funcs}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

adds each subprogram definition given by a func AST node in `funcs` to the subprograms map of $G^{\texttt{tenv}}$, yielding `new_tenv`.

### Example: Adding Subprogram Declarations

The specification in Listing 28.6 contains two subprogram declarations, which are added to the static environment. The subprogram with the integer type argument is named `increment` and the subprogram with the real type argument is named `increment-1`.

Listing 28.6: Adding subprogram declarations

```
func increment(x: integer) => integer
begin
    return x + 1;
end;

func increment(r: real) => real
begin
    return r + 1.0;
end;
```

### Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `funcs` is the empty list;
  - `new_tenv` is `tenv`.
- All of the following apply (NON_EMPTY):
  - `funcs` is the list with `head` (`f`, `ses_f`) and `tail` `funcs1`;
  - applying *add_subprogram* to `f`.name, `f`, and `ses_f` in `tenv` yields `tenv1`;
  - applying *add_subprogram_decls* to `tenv1` and `funcs1` yields `new_tenv`.

### Formally

EMPTY

$$\mathit{add\_subprogram\_decls}(\texttt{tenv}, \overbrace{[\,]}^{\texttt{funcs}}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}$$

NON_EMPTY

$$\frac{\mathit{add\_subprogram}(\texttt{tenv}, \texttt{f}.\text{name}, \texttt{f}, \texttt{ses\_f}) \xrightarrow{\text{type}} \texttt{tenv1} \qquad \mathit{add\_subprogram\_decls}(\texttt{tenv1}, \texttt{funcs1}) \xrightarrow{\text{type}} \texttt{new\_tenv}}{\mathit{add\_subprogram\_decls}(\texttt{tenv}, \overbrace{[(\texttt{f}, \texttt{ses\_f})] + \texttt{funcs1}}^{\texttt{funcs}}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

### TypingRule.PropagateRecursiveCallsSess

The helper relation

$$\mathit{propagate\_recursive\_calls\_sess}(\overbrace{(\mathsf{func} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{sess}}) \bigtimes \overbrace{(\mathsf{func} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{sess\_new}}$$

accepts a list that associates func AST nodes to sets of side effect descriptors and generates a list that associates each func AST nodes to the union of all side effect descriptors associated with the recursive functions it may transitively call.

We assume each func AST node occurs in exactly one pair of `sess`.

### Example: Propagating Recursive Call Side Effects

Consider the specification in Listing 28.7.

Listing 28.7: Propagating recursive call side effects

```
var g1 : integer;
var g2 : integer;
var g3 : integer;

func main() => integer
begin
    - = count_g1(10);
    return 0;
end;

func increment_g3()
begin
    g3 = g3 + 1;
end;

func count_g1(counter: integer) => integer recurselimit 100
begin
    g1 = g1 + 1;
    if counter > 0 then
        - = count_g2(counter - 1);
    end;
    return g1;
end;

func count_g2(counter: integer) => integer recurselimit 100
begin
    g2 = g2 + 1;
    increment_g3();
    - = count_g1(counter - 1);
    return g2;
end;
```

TypingRule.TypeCheckMutuallyRec applies *propagate_recursive_calls_sess* to the following subprograms and their associated side effect descriptors are inferred for each subprogram separately (`func_id_to_ses` in the rules below):

$$\left[\begin{array}{ll} \texttt{count\_g1}, & \left\{\begin{array}{l} \mathsf{ReadGlobal}(\texttt{g1}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{WriteGlobal}(\texttt{g1}), \\ \mathsf{RecursiveCall}(\texttt{count\_g2}) \end{array}\right\} \\ , & \\ \texttt{count\_g2}, & \left\{\begin{array}{l} \mathsf{ReadGlobal}(\texttt{g2}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{WriteGlobal}(\texttt{g2}), \\ \mathsf{ReadGlobal}(\texttt{g3}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{RecursiveCall}(\texttt{g1}) \end{array}\right\} \end{array}\right]$$

*propagate_recursive_calls_sess* then infers the following relation between the subprograms based on their associated sets of side effect descriptors (`call_graph` in the rules below):

| | | |
|---|---|---|
| (`count_g1` | , | `count_g2`) |
| (`count_g2` | , | `count_g1`) |

*propagate_recursive_calls_sess* then computes the transitive closure of the relation above (`transitive_call_graph` in the rules below):

| | | |
|---|---|---|
| (`count_g1` | , | `count_g1`) |
| (`count_g1` | , | `count_g2`) |
| (`count_g2` | , | `count_g1`) |
| (`count_g2` | , | `count_g2`) |

Then, the side effect descriptor are accumulated, excluding recursive call side effect descriptors, which yields the final result:

$$\left[\begin{array}{ll} \texttt{count\_g1}, & \left\{\begin{array}{l} \mathsf{ReadGlobal}(\texttt{g1}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{WriteGlobal}(\texttt{g1}), \\ \mathsf{ReadGlobal}(\texttt{g2}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{WriteGlobal}(\texttt{g2}), \\ \mathsf{ReadGlobal}(\texttt{g3}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{WriteGlobal}(\texttt{g3}), \end{array}\right\} \\ , & \\ \texttt{count\_g2}, & \left\{\begin{array}{l} \mathsf{ReadGlobal}(\texttt{g1}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{WriteGlobal}(\texttt{g1}), \\ \mathsf{ReadGlobal}(\texttt{g2}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{WriteGlobal}(\texttt{g2}), \\ \mathsf{ReadGlobal}(\texttt{g3}, \mathsf{Execution}, \texttt{FALSE}), \\ \mathsf{WriteGlobal}(\texttt{g3}), \end{array}\right\} \end{array}\right]$$

### Prose

All of the following apply:

- define `func_id_to_ses` as the function that maps a subprogram name of `f` to its associated sets of side effect descriptors `s`, for each pair `(f, s)` in `sess`;
- define `call_graph` as the relation between subprogram names `f1` and `f2` such that the sets of side effect descriptors associated with `f1` includes the recursive call side effect descriptor for `f2`;
- define `transitive_call_graph` as the transitive closure of `call_graph`;
- define `func_id_to_ses_minus_rec` as the function where for each mapping of subprogram name `id` to sets of side effect descriptors `s` in `func_id_to_ses`, there exists a mapping of `id` to `s` with all recursive call side effect descriptors removed;
- define `callees` as the function where for each pair `(f, ses)` in `sess` there exists a mapping of `f` to the set of subprogram names such that for each such subprogram name `succ`, `(f`.name, `succ)` is a member of `transitive_call_graph`;
- define `sess_new` as the list constructed from each pair `(f, s)` in `sess` by associating `f` to the union of sets of side effect descriptors given by the sets of side effect descriptors of `func_id_to_ses_minus_rec(c)` for each successor of `f` in `callees`, with all recursive call side effect descriptors removed.

**Formally**

$$\begin{aligned}
\texttt{func_id_to_ses} &:= \{\texttt{f}.\text{name} \mapsto \texttt{s} \mid (\texttt{f}, \texttt{s}) \in \texttt{sess}\} \\
\texttt{call_graph} &:= \{(\texttt{f1}, \texttt{f2}) \mid (\texttt{f1}, \texttt{ses}) \in \texttt{func_id_to_ses} \wedge \mathsf{RecursiveCall}(\texttt{f2}) \in \texttt{ses}\} \\
\texttt{transitive_call_graph} &:= \texttt{call_graph}^{+} \\
\texttt{func_id_to_ses_minus_rec} &:= \\
&\{\texttt{id} \mapsto \texttt{s} \setminus \mathsf{RecursiveCall}(\mathbb{I}) \mid (\texttt{id}, \texttt{s}) \in \mathtt{func_graph}(\texttt{func_id_to_ses})\} \\
\texttt{callees} &:= \\
&\{\texttt{f} \mapsto \{\texttt{succ} \mid (\texttt{f}.\text{name}, \texttt{succ}) \in \texttt{transitive_call_graph}\} \mid (\texttt{f}, \texttt{ses}) \in \texttt{sess}\} \\
\texttt{sess_new} &:= \\
&\left[(\texttt{f}, \texttt{s}) \in \texttt{sess} : \left(\texttt{f}, \bigcup_{\texttt{c} \in \texttt{callees(f)}} \texttt{func_id_to_ses_minus_rec(c)} \setminus \mathsf{RecursiveCall}(\mathbb{I})\right)\right] \\
\hline
&\mathit{propagate_recursive_calls_sess}(\texttt{sess}) \xrightarrow{\text{type}} \texttt{sess_new}
\end{aligned}$$

## 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;
- 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

$$\mathit{override\_subprograms}(\overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \longrightarrow \overbrace{\mathsf{decl}^*}^{\texttt{decls'}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

- processing overrides in `impdefs` and `impls` yields `impdefs'` and `discarded`$^{//\text{\#TE}}$;
- 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**

$$\texttt{impdefs} := [\texttt{d} : \texttt{D\_Func}(\texttt{d}) \in \texttt{decls} \wedge \texttt{d.override} = \mathsf{Impdef}]$$
$$\texttt{impls} := [\texttt{d} : \texttt{D\_Func}(\texttt{d}) \in \texttt{decls} \wedge \texttt{d.override} = \mathsf{Implementation}]$$
$$\texttt{normal} := [\texttt{d} : \texttt{D\_Func}(\texttt{d}) \in \texttt{decls} \wedge \texttt{d.override} \notin \{\mathsf{Impdef}, \mathsf{Implementation}\}]$$
$$\textit{check\_implementations\_unique}(\texttt{impls}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE}$$
$$\textit{process\_overrides}(\texttt{impdefs}, \texttt{impls}) \xrightarrow{\text{type}} (\texttt{impdefs'}, \texttt{discarded}) \;//\; \texttt{\#TE}$$
$$\textit{rename\_subprograms}(\texttt{discarded}) \xrightarrow{\text{type}} \texttt{renamed\_discarded}$$
$$\texttt{overridden} := \texttt{impdefs'} + \texttt{impls}$$
$$\frac{\texttt{decls'} := [\texttt{D\_Func}(\texttt{d}) : \texttt{d} \in \texttt{overridden}] + [\texttt{D\_Func}(\texttt{d}) : \texttt{d} \in \texttt{renamed\_discarded}] + \texttt{normal}}{\textit{override\_subprograms}(\texttt{decls}) \xrightarrow{\text{type}} \texttt{decls'}}$$

**TypingRule.CheckImplementationsUnique**

The function

$$\textit{check\_implementations\_unique}(\overbrace{\mathsf{func}^*}^{\texttt{impls}}) \longrightarrow \{\texttt{TRUE}\} \cup \mathsf{TTypeError}$$

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`$^{//\text{TE\_OE}}$;
  - checking that implementations are unique in `t` yields `TRUE`$^{//\text{\#TE}}$;
  - the result is `TRUE`.

**Formally**

EMPTY

$$\mathit{check\_implementations\_unique}(\overbrace{[\,]}^{\texttt{impls}}) \xrightarrow{\text{type}} \texttt{TRUE}$$

NON_EMPTY

$$\frac{i \in \mathit{indices}(\texttt{t}) : \mathit{signatures\_match}(\texttt{h}, \texttt{t}[i]) \xrightarrow{\text{type}} \texttt{FALSE} \;/\!/\; \texttt{TE\_OE} \qquad \mathit{check\_implementations\_unique}(\texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_implementations\_unique}(\overbrace{[\texttt{h}] + \texttt{t}}^{\texttt{impls}}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

**TypingRule.SignaturesMatch**

The function

$$\mathit{signatures\_match}(\overbrace{\mathsf{func}}^{\texttt{func1}}, \overbrace{\mathsf{func}}^{\texttt{func2}}) \longrightarrow \mathbb{B}$$

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 $/\!/^{\texttt{FALSE}}$;
- checking that the declared arguments of `func1` and `func2` are equal in both name and type yields TRUE $/\!/^{\texttt{FALSE}}$;
- checking that the declared parameters of `func1` and `func2` are equal in both name and type yields TRUE $/\!/^{\texttt{FALSE}}$;
- checking that the return types of `func1` and `func2` are equal yields TRUE $/\!/^{\texttt{FALSE}}$.

**Formally**

$$\frac{\begin{array}{c}\mathit{bool\_transition}(\texttt{func1.name} = \texttt{func2.name}) \longrightarrow \texttt{TRUE} \mathbin{/\!\!/} \texttt{FALSE} \\ \mathit{equal\_length}(\texttt{func1.args}, \texttt{func2.args}) \xrightarrow{\text{type}} \texttt{TRUE} \mathbin{/\!\!/} \texttt{FALSE} \\ i \in \texttt{indices}(\texttt{func1.args}),\ \texttt{func1.args}[i] \overset{\text{is}}{=} (\texttt{id1}, \texttt{ty1}),\ \texttt{func2.args}[i] \overset{\text{is}}{=} (\texttt{id2}, \texttt{ty2}) : \\ \mathit{bool\_transition}(\texttt{id1} = \texttt{id2} \wedge \texttt{ty1} = \texttt{ty2}) \longrightarrow \texttt{b1}_i \\ \mathit{bool\_transition}(\bigwedge_{i \in \texttt{indices}(\texttt{func1.args})} \texttt{b1}_i) \longrightarrow \texttt{TRUE} \mathbin{/\!\!/} \texttt{FALSE} \\ i \in \texttt{indices}(\texttt{func1.parameters}), \\ \texttt{func1.parameters}[i] \overset{\text{is}}{=} (\texttt{id1}, \texttt{ty1}),\ \texttt{func2.parameters}[i] \overset{\text{is}}{=} (\texttt{id2}, \texttt{ty2}) : \\ \mathit{bool\_transition}(\texttt{id1} = \texttt{id2} \wedge \texttt{ty1} = \texttt{ty2}) \longrightarrow \texttt{b2}_i \\ \mathit{bool\_transition}(\bigwedge_{i \in \texttt{indices}(\texttt{func1.parameters})} \texttt{b2}_i) \longrightarrow \texttt{TRUE} \mathbin{/\!\!/} \texttt{FALSE} \\ \mathit{bool\_transition}(\texttt{func1.return\_type} = \texttt{func2.return\_type}) \longrightarrow \texttt{TRUE} \mathbin{/\!\!/} \texttt{FALSE}\end{array}}{\mathit{signatures\_match}(\texttt{func1}, \texttt{func2}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

**TypingRule.ProcessOverrides**

The function

$$\mathit{process\_overrides}(\overbrace{\mathsf{func}^*}^{\texttt{impdefs}}, \overbrace{\mathsf{func}^*}^{\texttt{impls}}) \longrightarrow (\overbrace{\mathsf{func}^*}^{\texttt{impdefs'}} \times \overbrace{\mathsf{func}^*}^{\texttt{discarded}}) \cup \overbrace{\mathsf{TTypeError}}^{\#\text{TE}}$$

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 $\texttt{b}_i$;
  - define `matching` as the sublist of `impdefs` for which $\texttt{b}_i$ is `TRUE`;

* define `nonmatching` as the sublist of `impdefs` for which $\texttt{b}_i$ is `FALSE`;
* checking the length of `matching` is 1 yields $\texttt{TRUE}^{/\!/\texttt{TE\_OE}}$;
* processing overrides in `impls` and `nonmatching` yields `impdefs'` and `discarded'`;
* define `discarded` as the concatenation of `matching` and `discarded'`.

**Formally**

$$\frac{}{\mathit{process\_overrides}(\texttt{impdefs}, \overbrace{[\,]}^{\texttt{impls}}) \xrightarrow{\text{type}} (\overbrace{\texttt{impdefs}}^{\texttt{impdefs'}}, [\,])}\ \text{EMPTY}$$

$$\frac{\begin{array}{c} i \in \mathit{indices}(\texttt{impdefs}) : \mathit{signatures\_match}(\texttt{h}, \texttt{impdefs}[i]) \xrightarrow{\text{type}} \texttt{b}_i \\ \texttt{matching} := [\texttt{impdefs}[i] \mid i \in \mathit{indices}(\texttt{impdefs}) \wedge \texttt{b}_i = \texttt{TRUE}] \\ \texttt{nonmatching} := [\texttt{impdefs}[i] \mid i \in \mathit{indices}(\texttt{impdefs}) \wedge \texttt{b}_i = \texttt{FALSE}] \\ \mathit{check}(|\texttt{matching}| = 1, \texttt{TE\_OE}) \longrightarrow \texttt{TRUE} /\!/ \ \texttt{TE\_OE} \\ \mathit{process\_overrides}(\texttt{nonmatching}, \texttt{t}) \xrightarrow{\text{type}} (\texttt{impdefs'}, \texttt{discarded'})\texttt{discarded} := \texttt{matching} + \texttt{discarded'} \end{array}}{\mathit{process\_overrides}(\texttt{impdefs}, \overbrace{[\texttt{h}] + \texttt{t}}^{\texttt{impls}}) \xrightarrow{\text{type}} (\texttt{impdefs'}, \texttt{discarded})}\ \text{NON\_EMPTY}$$

**TypingRule.RenameSubprograms**

The function

$$\mathit{rename\_subprograms}(\overbrace{\mathit{func}^*}^{\texttt{discarded}}) \longrightarrow \overbrace{\mathit{func}^*}^{\texttt{renamed\_discarded}}$$

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**

EMPTY

$$\mathit{rename\_subprograms}(\overbrace{[\,]}^{\texttt{discarded}}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\texttt{renamed\_discarded}}$$

NON_EMPTY

$$\frac{\mathit{rename\_subprograms}(\texttt{t}) \xrightarrow{\text{type}} \texttt{t'} \qquad \texttt{name} \in \mathbb{I} \text{ is fresh} \qquad \texttt{h'} := \texttt{h}[\text{name} \mapsto \texttt{name}]}{\mathit{rename\_subprograms}(\overbrace{[\texttt{h}] + \texttt{t}}^{\texttt{discarded}}) \xrightarrow{\text{type}} \overbrace{[\texttt{h'}] + \texttt{t'}}^{\texttt{renamed\_discarded}}}$$

## 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;

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:

$$\mathsf{def\_use\_name} \longrightarrow \mathsf{Subprogram}(\mathsf{identifier}) \mid \mathsf{Other}(\mathsf{identifier})$$

### TypingRule.BuildDependencies

The function

$$\mathit{build\_dependencies}(\overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \longrightarrow (\overbrace{\mathsf{def\_use\_name}^*}^{\texttt{defs}}, \overbrace{(\mathsf{def\_use\_name} \times \mathsf{def\_use\_name})^*}^{\texttt{depends}})$$

takes a set of declarations `decls` and returns a graph whose set of nodes — `defs` — consists of the identifiers that are used to name declarations and whose set of edges `depends` consists of pairs $(a, b)$ where the declaration of $a$ uses an identifier defined by the declaration of $b$. We refer to this graph as the *def-use dependency graph* (of `decls`).

### 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:

$$\left(\begin{array}{l} \{\mathsf{Other}(\texttt{g}), \mathsf{Other}(\texttt{MyRecord}), \mathsf{Other}(\texttt{Color}), \mathsf{Other}(\texttt{WORD\_SIZE}), \mathsf{Subprogram}(\texttt{main})\}, \\ \left\{\begin{array}{rcl} (\mathsf{Other}(\texttt{g}) & , & \mathsf{Other}(\texttt{MyRecord})), \\ (\mathsf{Other}(\texttt{MyRecord}) & , & \mathsf{Other}(\texttt{WORD\_SIZE})), \\ (\mathsf{Other}(\texttt{Color}) & , & \mathsf{Other}(\texttt{RED})), \\ (\mathsf{Other}(\texttt{Color}) & , & \mathsf{Other}(\texttt{GREEN})), \\ (\mathsf{Other}(\texttt{Color}) & , & \mathsf{Other}(\texttt{BLUE})), \\ (\mathsf{Subprogram}(\texttt{main}) & , & \mathsf{Other}(\texttt{g})) \end{array}\right\} \end{array}\right)$$

### 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

$$\frac{\texttt{defs} := \{\mathit{def\_\ decl}(\texttt{d}) \mid \texttt{d} \in \texttt{decls}\} \cup \bigcup_{\texttt{d}\in\texttt{decls}} \mathit{def\_\ enum\_\ labels}(\texttt{d}) \qquad \texttt{depends} := \bigcup_{\texttt{d}\in\texttt{decls}} \mathit{decl\_\ dependencies}(\texttt{d})}{\mathit{build\_\ dependencies}(\texttt{decls}) \xrightarrow{\text{type}} (\texttt{defs}, \texttt{depends})}$$

### TypingRule.DeclDependencies

The function

$$\mathit{decl\_\ dependencies}(\overbrace{\mathsf{decl}}^{\texttt{d}}) \longrightarrow \overbrace{(\mathsf{def\_use\_name} \times \mathsf{def\_use\_name})^*}^{\texttt{depends}}$$

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 $d_1$ `->` $d_2$.

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

$$\frac{\begin{array}{lll}\texttt{depends} & := & \{(\texttt{id1},\texttt{id2}) \mid \texttt{id1} = \mathit{def\_\ decl}(\texttt{d}) \wedge \texttt{id2} \in \mathit{def\_\ enum\_\ labels}(\texttt{d})\} \quad \cup \\ & & \{(\texttt{id1},\texttt{id2}) \mid \texttt{id1} = \mathit{def\_\ decl}(\texttt{d}) \wedge \texttt{id2} \in \mathit{use\_\ decl}(\texttt{d})\}\end{array}}{\mathit{decl\_\ dependencies}(\texttt{d}) \xrightarrow{\text{type}} \texttt{depends}}$$

### TypingRule.DefDecl

The function

$$\mathit{def\_\ decl}(\overbrace{\text{decl}}^{\text{d}}) \longrightarrow \overbrace{\text{def\_use\_name}}^{\text{name}}$$

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**

D_FUNC

$$def\_\ decl(\overbrace{\texttt{D\_Func}(\text{name} : \texttt{id}, \ldots)}^{\texttt{d}}) \xrightarrow{\text{type}} \overbrace{\text{Subprogram}(\texttt{id})}^{\texttt{name}}$$

D_GLOBALSTORAGE

$$def\_\ decl(\overbrace{\texttt{D\_GlobalStorage}(\text{name} : \texttt{id}, \ldots)}^{\texttt{d}}) \xrightarrow{\text{type}} \overbrace{\text{Other}(\texttt{id})}^{\texttt{name}}$$

D_TYPEDECL

$$def\_\ decl(\overbrace{\texttt{D\_TypeDecl}(\texttt{id}, \_, \_)}^{\texttt{d}}) \xrightarrow{\text{type}} \overbrace{\text{Other}(\texttt{id})}^{\texttt{name}}$$

**TypingRule.DefEnumLabels**

The function

$$def\_\ enum\_\ labels(\overbrace{\text{decl}}^{\texttt{d}}) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{\texttt{labels}}$$

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 {Other(`RED`), Other(`GREEN`), Other(`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

$$\frac{\texttt{d} = \texttt{D\_TypeDecl}(\texttt{name}, \texttt{T\_Enum}(\texttt{labels1}, \_)) \quad \texttt{labels} := \{\texttt{Other}(\texttt{label}) \mid \texttt{label} \in \texttt{labels1}\}}{\mathit{def\_enum\_labels}(\texttt{d}) \xrightarrow{\text{type}} \texttt{labels}}$$

OTHER

$$\frac{\texttt{d} \neq \texttt{D\_TypeDecl}(\texttt{name}, \texttt{T\_Enum}(\_, \_))}{\mathit{def\_enum\_labels}(\texttt{d}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\texttt{labels}}}$$

### TypingRule.UseDecl

The function

$$\mathit{use\_decl}(\overbrace{\mathsf{decl}}^{\texttt{d}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\texttt{ids}}$$

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**

D_TypeDecl

$$use\_decl(\overbrace{\texttt{D\_TypeDecl}(\_, \texttt{ty}, \texttt{fields})}^{\texttt{d}}) \xrightarrow{\text{type}} \overbrace{use\_ty(\texttt{ty}) \cup use\_subtypes(\texttt{fields})}^{\texttt{ids}}$$

D_GlobalStorage

$$\frac{\texttt{ids} := use\_e(\texttt{initial\_value}) \cup use\_ty(\texttt{ty})}{use\_decl(\overbrace{\texttt{D\_GlobalStorage}(\{\text{initial\_value} : \texttt{initial\_value}, \text{ty} : \texttt{ty} \ldots\})}^{\texttt{d}}) \xrightarrow{\text{type}} \texttt{ids}}$$

D_Func

$$\frac{\begin{array}{lll} \texttt{ids} & := & \{(\_, \texttt{t}) \in \texttt{args} : use\_ty(\texttt{t})\} \cup \\ & & use\_ty(\texttt{ret\_ty\_opt}) \cup \\ & & \{(\_, \texttt{t}) \in \texttt{params} : use\_ty(\texttt{t})\} \cup \\ & & use\_s(\texttt{body}) \cup \\ & & use\_e(\texttt{recurse\_limit}) \end{array}}{use\_decl\left(\texttt{D\_Func}\left(\overbrace{\left\{\begin{array}{l} \text{body} : \texttt{body}, \\ \text{args} : \texttt{args}, \\ \text{return\_type} : \texttt{ret\_ty\_opt}, \\ \text{parameters} : \texttt{params}, \\ \text{recurse\_limit} : \texttt{recurse\_limit}, \\ \ldots \end{array}\right\}}^{\texttt{d}}\right)\right) \xrightarrow{\text{type}} \texttt{ids}}$$

**TypingRule.UseTy**

The function

$$use\_ty(\overbrace{\text{ty} \cup \langle \text{ty} \rangle}^{\texttt{t}}) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{\texttt{ids}}$$

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 ⟨`ty`⟩;
  - applying *use_ty* to `ty` yields `ids`.
- All of the following apply (SIMPLE):
  - `t` is one of the following types: enumeration, Boolean, real, or string;
  - define `ids` as the empty set.
- All of the following apply (T_NAMED):
  - `t` is the named type for `s`;
  - define `ids` as the singleton set for 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):

  - `t` is the tuple type with list of types `li`;
  - define `ids` as the union of applying *use_constraint* to each constraint in `vcs`.
- All of the following apply (STRUCTURED):
  - `t` is a structured type with fields `fields`;
  - define `ids` as the union of applying *use_ty* to each field type in `fields`.
- All of the following apply (ARRAY_EXPR):
  - `t` is an array expression with length expression `e` and element type `t'`;
  - define `ids` as the union of applying *use_e* to `e` and applying *use_ty* to `t'`.
- All of the following apply (ARRAY_ENUM):
  - `t` is an array expression with enumeration type `s` and element type `t'`;
  - define `ids` as the union of the singleton set for Other(`s`) and applying *use_ty* to `t'`.
- All of the following apply (T_BITS):
  - `t` is a bitvector type with width expression `e` and bitfields `bitfields`;
  - define `ids` as the union of applying *use_e* to `e` and applying *use_bitfield* to each field in `bitfields`.

**Formally**

NONE

$$use\_ty(\overbrace{\mathsf{None}}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

SOME

$$\frac{use\_ty(\mathtt{ty}) \xrightarrow{\text{type}} \mathtt{ids}}{use\_ty(\overbrace{\langle \mathtt{ty} \rangle}^{\mathtt{t}}) \xrightarrow{\text{type}} \mathtt{ids}}$$

SIMPLE

$$\frac{ast\_label(\mathtt{t}) \in \{\mathsf{T\_Enum}, \mathsf{T\_Bool}, \mathsf{T\_Real}, \mathsf{T\_String}\}}{use\_ty(\mathtt{t}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathtt{ids}}}$$

T_NAMED

$$use\_ty(\overbrace{\mathsf{T\_Named}(\mathtt{s})}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\{\mathsf{Other}(\mathtt{s})\}}^{\mathtt{ids}}$$

INT_NO_CONSTRAINTS

$$\frac{ast\_label(\mathtt{c}) \in \{\mathsf{Unconstrained}, \mathsf{Parameterized}\}}{use\_ty(\overbrace{\mathsf{T\_Int}(\mathtt{c})}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathtt{ids}}}$$

INT_WELL_CONSTRAINED

$$\overbrace{\mathit{use\_ty}(\overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{vcs}))}^{\mathtt{t}})}^{} \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{c}\in\mathtt{vcs}} \mathit{use\_constraint}(\mathtt{c})}^{\mathtt{ids}}$$

T_TUPLE

$$\overbrace{\mathit{use\_ty}(\overbrace{\mathsf{T\_Tuple}(\mathtt{li})}^{\mathtt{t}})}^{} \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{t}\in\mathtt{li}} \mathit{use\_ty}(\mathtt{t})}^{\mathtt{ids}}$$

STRUCTURED

$$\frac{L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\}}{\overbrace{\mathit{use\_ty}(\overbrace{L(\mathtt{fields})}^{\mathtt{t}})}^{} \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{(\_,\mathtt{t})\in\mathtt{fields}} \mathit{use\_ty}(\mathtt{t})}^{\mathtt{ids}}}$$

ARRAY_EXPR

$$\overbrace{\mathit{use\_ty}(\overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Expr}(\mathtt{e}), \mathtt{t'})}^{\mathtt{t}})}^{} \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\mathtt{e}) \cup \mathit{use\_ty}(\mathtt{t'})}^{\mathtt{ids}}$$

ARRAY_ENUM

$$\overbrace{\mathit{use\_ty}(\overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Enum}(\mathtt{s}, \_), \mathtt{t'})}^{\mathtt{t}})}^{} \xrightarrow{\mathsf{type}} \overbrace{\{\mathsf{Other}(\mathtt{s})\} \cup \mathit{use\_ty}(\mathtt{t'})}^{\mathtt{ids}}$$

T_BITS

$$\overbrace{\mathit{use\_ty}(\overbrace{\mathsf{T\_Bits}(\mathtt{e}, \mathtt{bitfields})}^{\mathtt{t}})}^{} \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\mathtt{e}) \cup \bigcup_{\mathtt{f}\in\mathtt{bitfields}} \mathit{use\_bitfield}(\mathtt{f})}^{\mathtt{ids}}$$

### TypingRule.UseSubtypes

The function

$$\mathit{use\_subtypes}(\langle\langle \overbrace{\overbrace{\mathsf{identifier}}^{\mathtt{x}} \times \overbrace{\mathsf{field}^*}^{\mathtt{subfields}}}^{\mathtt{fields}} \rangle\rangle) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\mathtt{ids}}$$

returns the set of identifiers `ids` which the optional pair consisting of identifier `x` (the type being subtyped) and fields `subfields` depends on.

### 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 $\langle(\texttt{x}, \texttt{subfields})\rangle$;
  - define `ids` as the union of the singleton set for $\mathsf{Other}(\texttt{x})$ and the union of applying *use_ty* to each field type in `subfields`.

### Formally

NONE

$$\mathit{use\_subtypes}(\texttt{None}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}$$

SOME

$$\frac{\texttt{ids} := \{\mathsf{Other}(\texttt{x})\} \cup \bigcup_{(\_,\texttt{t}) \in \texttt{subfields}} \mathit{use\_ty}(\texttt{t})}{\mathit{use\_subtypes}(\langle(\texttt{x}, \texttt{subfields})\rangle) \xrightarrow{\text{type}} \texttt{ids}}$$

### TypingRule.UseExpr

The function

$$\mathit{use\_e}(\overbrace{\mathsf{expr}}^{\texttt{e}} \cup \langle\overbrace{\mathsf{expr}}^{\texttt{e}}\rangle) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\text{ids}}$$

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 ⟨`e1`⟩;
  - applying *use_e* to `e1` yields `ids`.
- All of the following apply (E_LITERAL):
  - `e` is a literal expression;
  - define `ids` as the empty set.
- All of the following apply (E_ATC):
  - `e` is the typing assertion for expression `e` and type `ty`;
  - define `ids` as the union of applying *use_e* to `e1` and applying *use_ty* to `ty`.
- All of the following apply (E_VAR):
  - `e` is the variable expression for identifier `x`;
  - define `ids` as the singleton set for 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**

NONE

$$use\_e(\overbrace{\mathsf{None}}^{e}) \xrightarrow{type} \overbrace{\emptyset}^{ids}$$

SOME

$$\frac{use\_e(\mathtt{e1}) \xrightarrow{type} \mathtt{ids}}{use\_e(\overbrace{\langle \mathtt{e1} \rangle}^{e}) \xrightarrow{type} \mathtt{ids}}$$

E_LITERAL

$$use\_e(\overbrace{\mathsf{E\_Literal}(\_)}^{e}) \xrightarrow{type} \overbrace{\emptyset}^{ids}$$

E_ATC

$$use\_e(\overbrace{\mathsf{E\_ATC}(\mathtt{e1}, \mathtt{ty})}^{e}) \xrightarrow{type} \overbrace{use\_e(\mathtt{e1}) \cup use\_ty(\mathtt{ty})}^{ids}$$

E_VAR

$$use\_e(\overbrace{\mathsf{E\_Var}(\mathtt{x})}^{e}) \xrightarrow{type} \overbrace{\{\mathsf{Other}(\mathtt{x})\}}^{ids}$$

E_GETARRAY

$$use\_e(\overbrace{\mathsf{E\_GetArray}(\mathtt{e1},\mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

E_GETENUMARRAY

$$use\_e(\overbrace{\mathsf{E\_GetEnumArray}(\mathtt{e1},\mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

E_BINOP

$$use\_e(\overbrace{\mathsf{E\_Binop}(\_,\mathtt{e1},\mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

E_UNOP

$$use\_e(\overbrace{\mathsf{E\_Unop}(\_,\mathtt{e1})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_CALL

$$\frac{\mathtt{ids} := \{\mathsf{Subprogram}(\mathtt{x})\} \cup \bigcup_{\mathtt{e1} \in \mathtt{args}} use\_e(\mathtt{e1}) \cup \bigcup_{(\_,\mathtt{t}) \in \mathtt{named\_args}} use\_ty(\mathtt{t})}{use\_e(\overbrace{\mathsf{E\_Call}(\mathtt{x},\mathtt{args},\mathtt{named\_args})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

E_SLICE

$$use\_e(\overbrace{\mathsf{E\_Slice}(\mathtt{e1},\mathtt{slices})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup \bigcup_{\mathtt{s} \in \mathtt{slices}} use\_slice(\mathtt{s})}^{\mathtt{ids}}$$

E_COND

$$use\_e(\overbrace{\mathsf{E\_Cond}(\mathtt{e1},\mathtt{e2},\mathtt{e3})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2}) \cup use\_e(\mathtt{e3})}^{\mathtt{ids}}$$

E_GETITEM

$$use\_e(\overbrace{\mathsf{E\_GetItem}(\mathtt{e1},\_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_GETFIELD

$$use\_e(\overbrace{\mathsf{E\_GetField}(\mathtt{e1},\_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_GETFIELDS

$$use\_e(\overbrace{\mathsf{E\_GetFields}(\mathtt{e1},\_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_RECORD

$$use\_e(\overbrace{\mathsf{E\_Record}(\mathtt{ty}, \mathtt{li})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_ty(\mathtt{ty}) \cup \bigcup_{(\_,\mathtt{t}) \in \mathtt{li}} use\_ty(\mathtt{t})}^{\mathtt{ids}}$$

E_TUPLE

$$use\_e(\overbrace{\mathsf{E\_Tuple}(\mathtt{e\_s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{e1} \in \mathtt{e\_s}} use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_ARRAY

$$use\_e(\overbrace{\mathsf{E\_Array}\{\text{length} : \mathtt{e1}, \text{value} : \mathtt{e2}\}}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

E_ENUMARRAY

$$\frac{\mathtt{ids1} := \bigcup_{\mathtt{label} \in \mathtt{labels}} \mathsf{Other}(\mathtt{label})}{use\_e(\overbrace{\mathsf{E\_EnumArray}\{\text{labels} : \mathtt{labels}, \text{value} : \mathtt{value}\}}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\{\mathtt{labels}\} \cup use\_e(\mathtt{value})}^{\mathtt{ids}}}$$

E_ARBITRARY

$$use\_e(\overbrace{\mathsf{E\_Arbitrary}(\mathtt{t})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_ty(\mathtt{t})}^{\mathtt{ids}}$$

E_PATTERN

$$use\_e(\overbrace{\mathsf{E\_Pattern}(\mathtt{e1}, \mathtt{p})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_pattern(\mathtt{p})}^{\mathtt{ids}}$$

### TypingRule.UseLexpr

The function

$$use\_le(\overbrace{\mathsf{lexpr}}^{\mathtt{le}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\mathtt{ids}}$$

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**

LE_VAR

$$use\_le(\overbrace{\mathsf{LE\_Var}(\mathtt{x})}^{\mathtt{le}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Other}(\mathtt{x})}^{\mathtt{ids}}$$

LE_DESTRUCTURING

$$use\_le(\overbrace{\mathsf{LE\_Destructuring}(\mathtt{les})}^{\mathtt{le}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\mathtt{e} \in \mathtt{les}} use\_le(\mathtt{e})}^{\mathtt{ids}}$$

LE_DISCARD

$$use\_le(\overbrace{\mathsf{LE\_Discard}}^{\mathtt{le}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

LE_SETARRAY

$$use\_le(\overbrace{\mathsf{LE\_SetArray}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{le}}) \xrightarrow{\text{type}} \overbrace{use\_le(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

LE_SETENUMARRAY

$$use\_le(\overbrace{\mathsf{LE\_SetEnumArray}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{le}}) \xrightarrow{\text{type}} \overbrace{use\_le(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

LE_SETFIELD

$$use\_le(\overbrace{\mathsf{LE\_SetField}(\mathtt{e1}, \_)}^{\mathtt{le}}) \xrightarrow{\text{type}} \overbrace{use\_le(\mathtt{e1})}^{\mathtt{ids}}$$

LE_SETFIELDS

$$use\_le(\overbrace{\mathsf{LE\_SetFields}(\mathtt{e1}, \_)}^{\mathtt{le}}) \xrightarrow{\text{type}} \overbrace{use\_le(\mathtt{e1})}^{\mathtt{ids}}$$

LE_SLICE

$$use\_le(\overbrace{\mathsf{LE\_Slice}(\texttt{e1}, \texttt{slices})}^{\texttt{le}}) \xrightarrow{\text{type}} \overbrace{use\_le(\texttt{e1}) \cup \bigcup_{\texttt{s} \in \texttt{slices}} use\_slice(\texttt{s})}^{\texttt{ids}}$$

### TypingRule.UsePattern

The function

$$use\_pattern(\overbrace{\mathsf{pattern}}^{\texttt{p}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\texttt{ids}}$$

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**

MASK_ALL

$$\frac{ast\_label(\mathtt{p}) \in \{\mathsf{Pattern\_Mask}, \mathsf{Pattern\_All}\}}{use\_pattern(\mathtt{p}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{ids}}}$$

TUPLE

$$use\_pattern(\overbrace{\mathsf{Pattern\_Tuple}(\mathtt{li})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{p1} \in \mathtt{li}} use\_pattern(\mathtt{p1})}^{\mathtt{ids}}$$

ANY

$$use\_pattern(\overbrace{\mathsf{Pattern\_Any}(\mathtt{li})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{p1} \in \mathtt{li}} use\_pattern(\mathtt{p1})}^{\mathtt{ids}}$$

SINGLE

$$use\_pattern(\overbrace{\mathsf{Pattern\_Single(e)}}^{\mathsf{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathsf{e})}^{\mathsf{ids}}$$

GEQ

$$use\_pattern(\overbrace{\mathsf{Pattern\_Geq(e)}}^{\mathsf{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathsf{e})}^{\mathsf{ids}}$$

LEQ

$$use\_pattern(\overbrace{\mathsf{Pattern\_Leq(e)}}^{\mathsf{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathsf{e})}^{\mathsf{ids}}$$

NOT

$$use\_pattern(\overbrace{\mathsf{Pattern\_Not(p1)}}^{\mathsf{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_pattern(\mathsf{p1})}^{\mathsf{ids}}$$

RANGE

$$use\_pattern(\overbrace{\mathsf{Pattern\_Range(e1, e2)}}^{\mathsf{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathsf{e1}) \cup use\_e(\mathsf{e2})}^{\mathsf{ids}}$$

### TypingRule.UseSlice

The function

$$use\_slice(\overbrace{\mathsf{slice}}^{\mathsf{s}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\mathsf{ids}}$$

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

SINGLE

$$use\_slice(\overbrace{\mathsf{Slice\_Single(e)}}^{\mathtt{s}}) \xrightarrow{\text{type}} \overbrace{use\_e(\mathsf{e})}^{\mathtt{ids}}$$

START_LENGTH_RANGE

$$\frac{L \in \{\mathsf{Slice\_Star}, \mathsf{Slice\_Length}, \mathsf{Slice\_Range}\}}{use\_slice(\overbrace{L(\mathsf{e1}, \mathsf{e2})}^{\mathtt{s}}) \xrightarrow{\text{type}} \overbrace{use\_e(\mathsf{e1}) \cup use\_e(\mathsf{e2})}^{\mathtt{ids}}}$$

### TypingRule.UseBitfield

The function

$$use\_bitfield(\overbrace{\mathsf{decl}}^{\mathtt{bf}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\mathtt{ids}}$$

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

$$use\_bitfield(\overbrace{\mathsf{BitField\_Simple}(\_, \texttt{slices})}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\texttt{s} \in \texttt{slices}} use\_slice(\texttt{s})}^{\texttt{ids}}$$

NESTED

$$\frac{\texttt{ids} := \bigcup_{\texttt{bf1} \in \texttt{bitfields}} use\_bitfield(\texttt{s}) \cup \bigcup_{\texttt{s} \in \texttt{slices}} use\_slice(\texttt{s})}{use\_bitfield(\overbrace{\mathsf{BitField\_Nested}(\_, \texttt{slices}, \texttt{bitfields})}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{ids}}$$

TYPE

$$\frac{\texttt{ids} := \bigcup_{\texttt{s} \in \texttt{slices}} use\_slice(\texttt{s}) \cup use\_ty(\texttt{ty})}{use\_bitfield(\overbrace{\mathsf{BitField\_Type}(\_, \texttt{slices}, \texttt{ty})}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{ids}}$$

**TypingRule.UseConstraint**

The function

$$use\_constraint(\overbrace{\mathsf{int\_constraint}}^{\texttt{c}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\texttt{ids}}$$

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

EXACT

$$use\_constraint(\overbrace{\mathsf{Constraint\_Exact(e)}}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{use\_e(\mathsf{e})}^{\mathsf{ids}}$$

RANGE

$$use\_constraint(\overbrace{\mathsf{Constraint\_Range(e1, e2)}}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{use\_e(\mathsf{e1}) \cup use\_e(\mathsf{e2})}^{\mathsf{ids}}$$

### TypingRule.UseStmt

The function

$$use\_s(\overbrace{\mathsf{stmt}}^{\mathsf{s}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\mathsf{ids}}$$

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 looplimit 2^g2 do // { Other(FIVE), Other(SEVEN), Other(g2) }
        pass;
    end;
    var y : integer = g2; // { Other(g2) }
    try // { Other(g), Other(g2), Other(MyException) }
        y = g;
        throw_stmt();
```

```
    catch
        when MyException => // { Other(MyException), Other(g2) }
            y = g2;
            println("caught MyExeption");
    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):
  - `s` is the for statement S_For $\left\{ \begin{array}{rcl} \text{index\_name} & : & \_ \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{direction} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \texttt{limit} \end{array} \right\}$;
  - define `ids` as the union of applying *use_e* to `limit`, `start_e`, and `end_e` and applying *use_s* to `s1`.
- All of the following apply (WHILE_REPEAT):
  - `s` is either a while statement or repeat statement, each with expression `e`, body statement `s1`, and optional limit expression `limit`;
  - define `ids` as the union of applying *use_e* to `limit` and to `e`, and applying *use_s* to `s1`.
- All of the following apply (S_DECL):
  - `s` is a declaration statement with optional type annotation `t` and optional initialization expression `e`;
  - define `ids` as the union of applying *use_ty* to `t` and *use_e* to `e`.
- All of the following apply (S_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()`;
  - define `ids` as the empty set.

**Formally**

PASS_RETURN_NONE_THROW_NONE

$$\frac{\mathtt{s} = \mathsf{S\_Pass} \vee \mathtt{s} = \mathsf{S\_Return}(\mathtt{None}) \vee \mathtt{s} = \mathsf{S\_Throw}(\mathtt{None})}{use\_s(\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{ids}}}$$

S_SEQ

$$use\_s(\overbrace{\mathsf{S\_Seq}(\mathtt{s1}, \mathtt{s2})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{use\_s(\mathtt{s1}) \cup use\_s(\mathtt{s2})}^{\mathtt{ids}}$$

ASSERT_RETURN_SOME

$$\frac{\mathtt{s} = \mathsf{S\_Assert}(\mathtt{e}) \vee \mathtt{s} = \mathsf{S\_Return}(\langle \mathtt{e} \rangle)}{use\_s(\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e})}^{\mathtt{ids}}}$$

S_ASSIGN

$$use\_s(\overbrace{\mathsf{S\_Assign}(\mathtt{le}, \mathtt{e})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{use\_le(\mathtt{le}) \cup use\_e(\mathtt{e})}^{\mathtt{ids}}$$

S_CALL

$$\frac{\mathtt{ids} := \{\mathsf{Subprogram}(\mathtt{x})\} \cup \bigcup_{\mathtt{e} \in \mathtt{args}} use\_e(\mathtt{e}) \cup \bigcup_{(\_,\mathtt{e}) \in \mathtt{named\_args}} use\_e(\mathtt{e})}{use\_s(\overbrace{\mathsf{S\_Call}(\mathtt{x}, \mathtt{args}, \mathtt{named\_args})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

S_COND

$$use\_s(\overbrace{\mathsf{S\_Cond}(\mathtt{e}, \mathtt{s1}, \mathtt{s2})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e}) \cup use\_s(\mathtt{s1}) \cup use\_s(\mathtt{s2})}^{\mathtt{ids}}$$

S_FOR

$$\frac{\mathtt{ids} := use\_e(\mathtt{limit}) \cup use\_e(\mathtt{start\_e}) \cup use\_e(\mathtt{end\_e}) \cup use\_s(\mathtt{body})}{use\_s\left(\overbrace{\mathsf{S\_For}\left\{\begin{array}{rcl} \mathrm{index\_name} & : & \_ \\ \mathrm{start\_e} & : & \mathtt{start\_e} \\ \mathsf{for\_direction} & : & \mathtt{direction} \\ \mathrm{end\_e} & : & \mathtt{end\_e} \\ \mathrm{body} & : & \mathtt{body} \\ \mathrm{limit} & : & \mathtt{limit} \end{array}\right\}}^{\mathtt{s}}\right) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

WHILE_REPEAT

$$\frac{\mathtt{s} = \mathsf{S\_While}(\mathtt{e}, \mathtt{limit}, \mathtt{s}) \vee \mathtt{s} = \mathsf{S\_Repeat}(\mathtt{s}, \mathtt{e}, \mathtt{limit})}{use\_s(\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{limit}) \cup use\_e(\mathtt{e}) \cup use\_s(\mathtt{s1})}^{\mathtt{ids}}}$$

S_DECL

$$use\_s(\overbrace{\mathsf{S\_Decl}(\_,\_,\mathtt{t},\mathtt{e})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{use\_ty(\mathtt{t}) \cup use\_e(\mathtt{e})}^{\mathtt{ids}}$$

S_THROW_SOME

$$use\_s(\overbrace{\mathsf{S\_Throw}(\langle \mathtt{e} \rangle)}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e})}^{\mathtt{ids}}$$

S_TRY

$$\frac{\mathtt{ids} := use\_s(\mathtt{s1}) \cup \bigcup_{\mathtt{c} \in \mathtt{catchers}} use\_catcher(\mathtt{c}) \cup use\_s(\mathtt{s2})}{use\_s(\overbrace{\mathsf{S\_Try}(\mathtt{s1},\mathtt{catchers},\mathtt{s2})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

S_PRINT

$$use\_s(\overbrace{\mathsf{S\_Print}(\mathtt{args},\_)}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{e} \in \mathtt{args}} use\_e(\mathtt{e})}^{\mathtt{ids}}$$

S_UNREACHABLE

$$use\_s(\overbrace{\mathsf{S\_Unreachable}}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

### TypingRule.UseCatcher

The function

$$use\_catcher(\overbrace{\mathsf{catcher}}^{\mathtt{c}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{def\_use\_name})}^{\mathtt{ids}}$$

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**

$$\mathit{use\_catcher}(\overbrace{(\_, \texttt{ty}, \texttt{s})}^{\texttt{c}}) \xrightarrow{\texttt{type}} \overbrace{\mathit{use\_ty}(\texttt{ty}) \cup \mathit{use\_s}(\texttt{s})}^{\texttt{ids}}$$

## 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:*

$$\mathit{SCC}(V, E) \triangleq \{C \subseteq V \mid \forall u, v \in C.\ (u, v), (v, u) \in E^*\}\ .$$

**Definition 44 (Topological Ordering of Components)** *For a non-empty graph $G = (V, E)$ and its strongly connected components $\texttt{comps} \triangleq \mathit{SCC}(V, E)$, a listing of $\texttt{comps}$ — $C_{1..k}$ — is a* topological ordering of components*, denoted $C_{1..k} \in \mathit{topological\_ordering\_comps}(\texttt{comps}, E)$, if the following condition holds:*

$$\forall 1 \le i \le j \le k.\ \exists c_i \in C_i.\ c_j \in C_j.\ (c_i, c_j) \in E^* \implies i \le j\ .$$

## 28.7 Semantics of Specifications

The semantics of specifications is defined via the relation *eval_spec*, which is defined next.

### SemanticsRule.EvalSpec

The relation

$$\mathit{eval\_spec}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{spec}}^{\texttt{spec}}) \bigtimes ((\overbrace{\mathcal{Z}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\mathsf{TDynError}}^{\#\texttt{DE}})$$

evaluates the specification `spec` with the static environment `tenv`, yielding the native integer value `v` and execution graph `g`. Otherwise, the result is a dynamic error.

Example: Returning a Value from the Entry Point shows a specification whose evaluation results in returning the value `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 $\texttt{g}^{/\!/\#\texttt{DE}}$;

- One of the following applies:
  - All of the following apply (NORMAL):
    - evaluating the subprogram `main` with an empty list of actual arguments and empty list of parameters in `env` yields $\mathsf{Normal}([(\mathtt{v}, \mathtt{g2})], \_)^{/\!/\mathtt{\#DE}}$;
    - `g` is the ordered composition of `g1` and `g2` with the `asl_po` edge;
    - the result of the entire evaluation is $(\mathtt{v}, \mathtt{g})$.
  - 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 $\mathsf{Throwing}(\mathtt{v\_opt}, \_)$, which is an uncaught exception;
    - the result of the entire evaluation is an error indicating that an exception was not caught.

**Formally**

NORMAL

$$\frac{\begin{array}{c} \mathit{build\_genv}(\mathtt{tenv}, \mathtt{spec}) \xrightarrow{\mathsf{eval}} (\mathtt{env}, \mathtt{g1}) \;\; /\!/ \;\; \mathtt{\#DE} \\ \mathit{eval\_subprogram}(\mathtt{env}, \mathtt{main}, [\,], [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Normal}([(\mathtt{v}, \mathtt{g2})], \_) \;\; /\!/ \;\; \mathtt{\#DE} \\ \mathtt{g} := \mathtt{g1} \xrightarrow{\mathtt{asl\_po}} \mathtt{g2} \end{array}}{\mathit{eval\_spec}(\mathtt{tenv}, \mathtt{spec}) \xrightarrow{\mathsf{eval}} (\mathtt{v}, \mathtt{g})}$$

THROWING

$$\frac{\begin{array}{c} \mathit{build\_genv}(\mathtt{tenv}, \mathtt{spec}) \xrightarrow{\mathsf{eval}} (\mathtt{env}, \mathtt{g1}) \;\; /\!/ \;\; \mathtt{\#DE} \\ \mathit{eval\_subprogram}(\mathtt{env}, \mathtt{main}, [\,], [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}(\mathtt{v\_opt}, \_) \end{array}}{\mathit{eval\_spec}(\mathtt{tenv}, \mathtt{spec}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_UE})}$$

### SemanticsRule.BuildGlobalEnv

The helper relation

$$\mathit{build\_genv}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{spec}}^{\mathtt{typed\_spec}}) \bigtimes (\overbrace{\mathbb{E}}^{\mathtt{new\_env}} \times \overbrace{\mathcal{G}}^{\mathtt{new\_g}}) \cup \overbrace{\mathsf{TDynError}}^{\mathtt{\#DE}}$$

populates the environment `env` and output execution graph `new_g` with the global storage declarations in `typed_spec`, starting from the static environment `tenv`. This works by traversing the global storage declarations and updating the environment accordingly. Otherwise, the result is a dynamic error.

It is assumed that `typed_spec` lists the declarations in reverse order with respect to the def-use dependency order (see TypingRule.TypeCheckAST).

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 $\emptyset_{\mathbb{DE}}$;
- evaluating the global storage declarations in `typed_spec` in `env` with the empty execution graph is $(\texttt{new\_env}, \texttt{new\_g})^{/\!/_{\texttt{\#DE}}}$.
- the result of the entire evaluation is $(\texttt{new\_env}, \texttt{new\_g})$.

### Formally

$$\frac{\texttt{env} := (\texttt{tenv}, \emptyset_{\mathbb{DE}}) \qquad \mathit{eval\_globals}(\texttt{typed\_spec}, (\texttt{env}, \emptyset_{\texttt{g}})) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g}) \;\; /\!/ \;\; \texttt{\#DE}}{\mathit{build\_genv}(\texttt{tenv}, \texttt{typed\_spec}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

# Chapter 29

# Top Level

In previous chapters, we defined the following components:

- Lexical analysis (Chapter 6),
- Parsing (Chapter 7),
- AST building (Section 8.5),
- 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

$$\mathit{check\_and\_interpret}(\overbrace{\mathbb{S}}^{\texttt{spec\_text}}, \overbrace{\mathbb{S}}^{\texttt{std\_text}}) \bigtimes \left( \begin{array}{c} \overbrace{(\overbrace{\mathcal{Z}}^{\texttt{ret}} \times \overbrace{\mathcal{G}}^{\texttt{g}})} \\ \cup \\ \{\ \texttt{\#BE\_LE}, \texttt{\#BE\_PE}, \texttt{\#BE}\ \} \\ \cup \\ \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}} \\ \cup \\ \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}} \end{array} \right)$$

accepts a textual description of a specification in `spec_text` and a textual description of the standard library in `std_text`. The descriptions are statically checked for validity. If

found invalid, an error configuration corresponding to the phase where the error exists is returned — scanning, parsing, or AST building. If found valid, an AST is built and semantically evaluated, yielding an integer return code `ret` and an execution graph `g`, or a dynamic error.

### TopLevelRule.CheckAndInterpret

#### Prose

All of the following apply:

- applying lexical analysis to `std_text` with the lexical specification SPEC_TOKEN yields the list of tokens `std_tokens`//#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**

$$\frac{\begin{array}{c}\mathit{scan}(\text{SPEC\_TOKEN}, \texttt{std\_text}) \xrightarrow{\text{scan}} \texttt{std\_tokens} \;/\!/\; \#\text{BE\_LE} \\ \mathit{asl\_parse}(\texttt{std\_tokens}) \xrightarrow{\text{parse}} \texttt{std\_parse} \;/\!/\; \#\text{BE\_PE} \\ \mathit{build\_ast}(\texttt{std\_parse}) \xrightarrow{\text{ast}} \texttt{std\_ast} \;/\!/\; \#\text{BE} \\ \mathit{rename\_locals}(\texttt{std\_ast}) \xrightarrow{\text{ast}} \texttt{std\_ast\_renamed} \\ \texttt{i} \in \mathit{indices}(\texttt{std\_ast\_renamed}) : \mathit{set\_builtin}(\texttt{std\_ast\_renamed[i]}) \xrightarrow{\text{type}} \\ \qquad\qquad \texttt{std\_decl\_ast}_i \;/\!/\; \#\text{BE} \\ \texttt{std\_as\_builtin} := [\texttt{i} \in \mathit{indices}(\texttt{std\_ast}) : \texttt{std\_decl\_ast}_i] \\ \mathit{scan}(\text{SPEC\_TOKEN}, \texttt{spec\_text}) \xrightarrow{\text{scan}} \texttt{spec\_tokens} \;/\!/\; \#\text{BE\_LE} \\ \mathit{asl\_parse}(\texttt{spec\_tokens}) \xrightarrow{\text{parse}} \texttt{spec\_parse} \;/\!/\; \#\text{BE\_PE} \\ \mathit{build\_ast}(\texttt{spec\_parse}) \xrightarrow{\text{ast}} \texttt{spec\_ast} \;/\!/\; \#\text{BE} \\ \texttt{untyped\_ast} := \texttt{std\_as\_builtin} + \texttt{spec\_ast} \\ \mathit{type\_check\_ast}(G^{\emptyset_{\text{SE}}}, \texttt{untyped\_ast}) \xrightarrow{\text{type}} (\texttt{typed\_ast}, \texttt{tenv}) \;/\!/\; \#\text{TE} \\ \mathit{eval\_spec}(\texttt{tenv}, \texttt{typed\_ast}) \xrightarrow{\text{eval}} (\texttt{Int}(\texttt{ret}), \texttt{g}) \;/\!/\; \#\text{DE}\end{array}}{\mathit{check\_and\_interpret}(\texttt{spec\_text}, \texttt{std\_text}) \longrightarrow (\texttt{Int}(\texttt{ret}), \texttt{g})}$$

## 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

$$\mathit{rename\_locals}(\overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \longrightarrow \overbrace{\mathsf{decl}^*}^{\texttt{new\_decls}}$$

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 $\texttt{new\_decl}_\texttt{i}$, for each index `i` in the list of indices for `decls`;

- define `new_decls` as the list of declarations $\texttt{new\_decl}_\texttt{i}$, for each index `i` in the list of indices for `decls`.

**Formally**

$$\frac{\texttt{i} \in \mathit{indices}(\texttt{decls}) : \mathit{rename\_locals\_decl}(\texttt{decls}[\texttt{i}]) \xrightarrow{\text{ast}} \texttt{new\_decl}_\texttt{i}}{\mathit{rename\_locals}(\texttt{decls}) \xrightarrow{\text{ast}} \overbrace{[\texttt{i} \in \mathit{indices}(\texttt{decls}) : \texttt{new\_decl}_\texttt{i}]}^{\texttt{new\_decls}}}$$

**ASTRule.RenameLocalsDecl**

The helper function

$$\mathit{rename\_locals\_decl}(\overbrace{\texttt{decl}}^{\texttt{decl}}) \longrightarrow \overbrace{\texttt{decl}^*}^{\texttt{new\_decl}}$$

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**

SUBPROGRAM

$$\frac{\mathit{rename\_locals\_func}(\texttt{f}) \xrightarrow{\text{ast}} \texttt{f\_new}}{\mathit{rename\_locals\_decl}(\overbrace{\texttt{D\_Func}(\texttt{f})}^{\texttt{decl}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_Func}(\texttt{f\_new})}^{\texttt{new\_decl}}}$$

OTHER

$$\frac{\mathit{ast\_label}(\texttt{decl}) \neq \texttt{D\_Func}}{\mathit{rename\_locals\_decl}(\texttt{decl}) \xrightarrow{\text{ast}} \overbrace{\texttt{decl}}^{\texttt{new\_decl}}}$$

### ASTRule.RenameLocalsFunc

The helper function

$$\mathit{rename\_locals\_func}(\overbrace{\mathsf{func}}^{\texttt{f}}) \longrightarrow \overbrace{\mathsf{func}}^{\texttt{f\_new}}$$

renames the local storage elements appearing in the subprogram description `f`, yielding the subprogram description `f_new`.

### Prose

All of the following apply:

- view `f` as a subprogram description consisting of a subprogram named `name`, list of parameters `params`, list of arguments `args`, body `body`, optional return type `ret_ty_opt`, optional subtype `subtype`, optional limit expression `limit_expr`, and builtin flag `builtin`;
- renaming the list local arguments `args` yields the list of arguments `new_args`;
- renaming the local storage elements in the list of parameters `params` yields the list of parameters `new_params`;
- renaming the local storage elements in the statement `body` yields the statement `new_body`;
- applying *rename_locals_ty* to the optional value `ret_ty_opt` via optional yields `new_ret_ty_opt`;
- define `f_new` as the subprogram description consisting of a subprogram named `name`, list of parameters `new_params`, list of arguments `new_args`, body `new_body`, optional return type `new_ret_ty_opt`, optional subtype `subtype`, optional limit expression `limit_expr`, and builtin flag `builtin`.

**Formally**

$$\texttt{f} \overset{\text{is}}{=} \left\{ \begin{array}{rcl} \text{name} & : & \texttt{name}, \\ \text{parameters} & : & \texttt{params}, \\ \text{args} & : & \texttt{args}, \\ \text{body} & : & \texttt{body}, \\ \text{return\_type} & : & \texttt{ret\_ty\_opt}, \\ \text{subprogram\_type} & : & \texttt{subtype} \\ \text{recurse\_limit} & : & \texttt{limit\_expr} \\ \text{builtin} & : & \texttt{builtin} \\ \text{override} & : & \texttt{override} \end{array} \right\}$$

$$\textit{rename\_locals\_args}(\texttt{args}) \xrightarrow{\text{ast}} \texttt{new\_args}$$

$$\textit{rename\_locals\_named\_args}(\texttt{params}) \xrightarrow{\text{ast}} \texttt{new\_params}$$

$$\textit{rename\_locals\_stmt}(\texttt{body}) \xrightarrow{\text{ast}} \texttt{new\_body}$$

$$\text{optional}[\textit{rename\_locals\_ty}](\texttt{ret\_ty\_opt}) \xrightarrow{\text{ast}} \texttt{new\_ret\_ty\_opt}$$

$$\texttt{f\_new} := \left\{ \begin{array}{rcl} \text{name} & : & \texttt{name}, \\ \text{parameters} & : & \texttt{new\_params}, \\ \text{args} & : & \texttt{new\_args}, \\ \text{body} & : & \texttt{new\_body}, \\ \text{return\_type} & : & \texttt{new\_ret\_ty\_opt}, \\ \text{subprogram\_type} & : & \texttt{subtype} \\ \text{recurse\_limit} & : & \texttt{limit\_expr} \\ \text{builtin} & : & \texttt{builtin} \\ \text{override} & : & \texttt{override} \end{array} \right\}$$

$$\textit{rename\_locals\_func}(\texttt{f}) \xrightarrow{\text{ast}} \texttt{f\_new}$$

**ASTRule.RenameLocalsArgs**

The helper function

$$\textit{rename\_locals\_args}(\overbrace{((\text{identifier} \times \text{ty})^*)}^{\texttt{args}}) \longrightarrow \overbrace{(\text{identifier} \times \text{ty})^*}^{\texttt{new\_args}}$$

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**

EMPTY

$$\mathit{rename\_locals\_args}(\overbrace{[\,]}^{\texttt{args}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{new\_args}}$$

NON_EMPTY

$$\frac{\mathit{rename\_locals\_name}(\texttt{name}) \xrightarrow{\text{ast}} \texttt{name'} \quad \mathit{rename\_locals\_ty}(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t'} \quad \mathit{rename\_locals\_args}(\texttt{args1}) \xrightarrow{\text{ast}} \texttt{args1'}}{\mathit{rename\_locals\_args}(\overbrace{(\texttt{name}, \texttt{t}) + \texttt{args1}}^{\texttt{args}}) \xrightarrow{\text{ast}} \overbrace{[(\texttt{name'}, \texttt{t'})] + \texttt{args1'}}^{\texttt{new\_args}}}$$

**ASTRule.RenameLocalsNamedArgs**

The helper function

$$\mathit{rename\_locals\_named\_args}(\overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\texttt{params}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\texttt{new\_params}}$$

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**

EMPTY

$$\mathit{rename\_locals\_named\_args}(\overbrace{[\,]}^{\texttt{params}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{new\_params}}$$

NON_EMPTY

$$\frac{\mathit{rename\_locals\_name}(\texttt{name}) \xrightarrow{\text{ast}} \texttt{name'} \quad \mathsf{optional}[\mathit{rename\_locals\_ty}](\texttt{ty\_opt}) \xrightarrow{\text{ast}} \texttt{ty\_opt'} \quad \mathit{rename\_locals\_named\_args}(\texttt{params1}) \xrightarrow{\text{ast}} \texttt{params1'}}{\mathit{rename\_locals\_named\_args}(\overbrace{(\texttt{name}, \texttt{ty\_opt}) + \texttt{args1}}^{\texttt{params}}) \xrightarrow{\text{ast}} \overbrace{[(\texttt{name'}, \texttt{ty\_opt'})] + \texttt{params1'}}^{\texttt{new\_params}}}$$

**ASTRule.RenameLocalsTy**

The helper function

$$\mathit{rename\_locals\_ty}(\overbrace{\mathsf{ty}}^{\texttt{ty}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{new\_ty}}$$

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 (TARRAY):
  - `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**

UNCHANGED

$$\frac{\mathit{ast\_label}(\texttt{ty}) \in \{\mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Bool}, \mathsf{T\_Enum}, \mathsf{T\_Named}\} \lor (\texttt{ty} = \mathsf{T\_Int}(\texttt{c}) \land \mathit{ast\_label}(\texttt{c}) \in \{\mathsf{Unconstrained}, \mathsf{PendingConstrained}\})}{\mathit{rename\_locals\_ty}(\texttt{ty}) \xrightarrow{\text{ast}} \overbrace{\texttt{ty}}^{\texttt{new\_ty}}}$$

T_INT_PARAMETERIZED

$$\frac{\mathit{ast\_label}(\texttt{c}) = \mathsf{Parameterized}}{\mathit{rename\_locals\_ty}(\overbrace{\mathsf{T\_Int}(\texttt{c})}^{\texttt{ty}}) \xrightarrow{\text{ast}} \texttt{unimplemented}}$$

T_INT_WELLCONSTRAINED

$$\frac{\texttt{new\_cs} := [\texttt{c} \in \texttt{vcs} : \mathit{rename\_locals\_constraint}(\texttt{vcs})]}{\mathit{rename\_locals\_ty}(\overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{vcs}))}^{\texttt{ty}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{new\_cs}))}^{\texttt{new\_ty}}}$$

T_BITS

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e'}}{\mathit{rename\_locals\_ty}(\overbrace{\mathsf{T\_Bits}(\texttt{e}, \texttt{bitfields})}^{\texttt{ty}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Bits}(\texttt{e'}, \texttt{bitfields})}^{\texttt{new\_ty}}}$$

T_TUPLE

$$\frac{\texttt{new\_li} := [\texttt{t} \in \texttt{li} : \mathit{rename\_locals\_ty}(\texttt{t})]}{\mathit{rename\_locals\_ty}(\overbrace{\mathsf{T\_Tuple}(\texttt{li})}^{\texttt{ty}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Tuple}(\texttt{new\_li})}^{\texttt{new\_ty}}}$$

T_ARRAY

$$\frac{\mathit{ast\_label}(\texttt{ty}) = \mathsf{T\_Array}}{\mathit{rename\_locals\_ty}(\texttt{ty}) \xrightarrow{\text{ast}} \texttt{unimplemented}}$$

STRUCTURED

$$\frac{L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}, \mathsf{T\_Collection}\} \quad \texttt{new\_li} := [(\texttt{name}, \texttt{t}) \in \texttt{li} : (\texttt{name}, \mathit{rename\_locals\_ty}(\texttt{t}))]}{\mathit{rename\_locals\_ty}(\overbrace{L(\texttt{li})}^{\texttt{ty}}) \xrightarrow{\text{ast}} \overbrace{L(\texttt{new\_li})}^{\texttt{new\_ty}}}$$

### ASTRule.RenameLocalsStmt

The helper function

$$\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{stmt}}^{\texttt{s}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

renames the local storage elements appearing in the statement `s`, yielding the statement `new_s`.

### Prose

- All of the following apply (S_PASS):
  - `s` is a pass statement;
  - define `new_s` as `s`.
- All of the following apply (S_SEQ):
  - `s` is a sequencing statement for `s1` and `s2`;
  - renaming the local storage elements in the statement `s1` yields the statement `s1'`;
  - renaming the local storage elements in the statement `s2` yields the statement `s2'`;

    * define `new_s` as a sequencing statement for `s1'` and `s2'`.

- All of the following apply (S_DECL):
    * `s` is a declaration statement for the local declaration keyword `ldk`, local declaration item `ldi`, optional type annotation `ty`, and optional initializing expression `e`;
    * renaming the local storage elements in the local declaration item `ldi` yields the local declaration item `ldi'`;
    * applying *rename_locals_ty* to the optional value `ty` via optional yields `ty'`;
    * applying *rename_locals_expr* to the optional value `e` via optional yields `e'`;
    * define `new_s` as a declaration statement for the local declaration keyword `ldk`, local declaration item `ldi'`, optional type annotation `ty'`, and optional initializing expression `e'`.
- All of the following apply (S_ASSIGN):
    * `s` is 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**

S_PASS

$$\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Pass}}^{\mathtt{s}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Pass}}^{\mathtt{new\_s}}$$

S_SEQ

$$\frac{\mathit{rename\_locals\_stmt}(\mathtt{s1}) \xrightarrow{\mathsf{ast}} \mathtt{s1'} \qquad \mathit{rename\_locals\_stmt}(\mathtt{s2}) \xrightarrow{\mathsf{ast}} \mathtt{s2'}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Seq}(\mathtt{s1}, \mathtt{s2})}^{\mathtt{s}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Seq}(\mathtt{s1'}, \mathtt{s2'})}^{\mathtt{new\_s}}}$$

S_DECL

$$\frac{\begin{array}{c}\mathit{rename\_locals\_ldi}(\mathtt{ldi}) \xrightarrow{\mathsf{ast}} \mathtt{ldi'} \\ \mathsf{optional}[\mathit{rename\_locals\_ty}](\mathtt{ty}) \xrightarrow{\mathsf{ast}} \mathtt{ty'} \qquad \mathsf{optional}[\mathit{rename\_locals\_expr}](\mathtt{e}) \xrightarrow{\mathsf{ast}} \mathtt{e'}\end{array}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Decl}(\mathtt{ldk}, \mathtt{ldi}, \mathtt{ty}, \mathtt{e})}^{\mathtt{s}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Decl}(\mathtt{ldk}, \mathtt{ldi'}, \mathtt{ty'}, \mathtt{e'})}^{\mathtt{new\_s}}}$$

S_ASSIGN

$$\frac{\mathit{rename\_locals\_lexpr}(\mathtt{le}) \xrightarrow{\mathsf{ast}} \mathtt{le'} \qquad \mathit{rename\_locals\_expr}(\mathtt{e}) \xrightarrow{\mathsf{ast}} \mathtt{e'}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Assign}(\mathtt{le}, \mathtt{e})}^{\mathtt{s}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Assign}(\mathtt{le'}, \mathtt{e'})}^{\mathtt{new\_s}}}$$

S_CALL

$$\frac{\begin{array}{c}\mathit{rename\_locals\_args}(\mathtt{args}) \xrightarrow{\mathsf{ast}} \mathtt{new\_args} \\ \mathit{rename\_locals\_named\_args}(\mathtt{params}) \xrightarrow{\mathsf{ast}} \mathtt{new\_params}\end{array}}{\mathit{rename\_locals\_stmt}\left(\overbrace{\mathsf{S\_Call}\left(\left\{\begin{array}{rcl}\mathrm{name} & : & \mathtt{name}, \\ \mathrm{params} & : & \mathtt{params}, \\ \mathrm{args} & : & \mathtt{args}, \\ \mathrm{call\_type} & : & \mathtt{call\_type}\end{array}\right\}\right)}^{\mathtt{s}}\right) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Call}\left(\left\{\begin{array}{rcl}\mathrm{name} & : & \mathtt{name}, \\ \mathrm{params} & : & \mathtt{new\_params}, \\ \mathrm{args} & : & \mathtt{new\_args}, \\ \mathrm{call\_type} & : & \mathtt{call\_type}\end{array}\right\}\right)}^{\mathtt{new\_s}}$$

S_RETURN

$$\frac{\mathsf{optional}[\mathit{rename\_locals}](\texttt{e}) \xrightarrow{\text{ast}} \texttt{e'}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Return}(\texttt{e})}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Return}(\texttt{e'})}^{\texttt{new\_s}}}$$

S_COND

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e'} \quad \mathit{rename\_locals\_stmt}(\texttt{s1}) \xrightarrow{\text{ast}} \texttt{s1'} \quad \mathit{rename\_locals\_stmt}(\texttt{s2}) \xrightarrow{\text{ast}} \texttt{s2'}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Cond}(\texttt{e},\texttt{s1},\texttt{s2})}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Cond}(\texttt{e'},\texttt{s1'},\texttt{s2'})}^{\texttt{new\_s}}}$$

S_ASSERT

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e'}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Assert}(\texttt{e})}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Assert}(\texttt{e'})}^{\texttt{new\_s}}}$$

S_FOR

$$\frac{\begin{array}{c}\mathit{rename\_locals\_expr}(\texttt{e\_start}) \xrightarrow{\text{ast}} \texttt{e\_start'} \\ \mathit{rename\_locals\_expr}(\texttt{e\_end}) \xrightarrow{\text{ast}} \texttt{e\_end'} \quad \mathit{rename\_locals\_stmt}(\texttt{body}) \xrightarrow{\text{ast}} \texttt{body'} \\ \mathsf{optional}[\mathit{rename\_locals\_expr}](\texttt{e\_limit\_opt}) \xrightarrow{\text{ast}} \texttt{e\_limit\_opt'}\end{array}}{\mathit{rename\_locals\_stmt}\left(\mathsf{S\_For}\overbrace{\left\{\begin{array}{rcl}\text{index\_name} & : & \texttt{id},\\ \text{start\_e} & : & \texttt{e\_start},\\ \text{dir} & : & \texttt{dir},\\ \text{end\_e} & : & \texttt{e\_end},\\ \text{body} & : & \texttt{body},\\ \text{limit} & : & \texttt{e\_limit\_opt}\end{array}\right\}}^{\texttt{s}}\right) \xrightarrow{\text{ast}} \mathsf{S\_For}\overbrace{\left\{\begin{array}{rcl}\text{index\_name} & : & \texttt{id},\\ \text{start\_e} & : & \texttt{e\_start'},\\ \text{dir} & : & \texttt{dir},\\ \text{end\_e} & : & \texttt{e\_end'},\\ \text{body} & : & \texttt{body'},\\ \text{limit} & : & \texttt{e\_limit\_opt'}\end{array}\right\}}^{\texttt{new\_s}}$$

S_WHILE

$$\frac{\begin{array}{c}\mathit{rename\_locals\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e'} \quad \mathsf{optional}[\mathit{rename\_locals\_expr}](\texttt{limit}) \xrightarrow{\text{ast}} \texttt{limit'} \\ \mathit{rename\_locals\_stmt}(\texttt{body}) \xrightarrow{\text{ast}} \texttt{body'}\end{array}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_While}(\texttt{e},\texttt{limit},\texttt{body})}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_While}(\texttt{e'},\texttt{limit'},\texttt{body'})}^{\texttt{new\_s}}}$$

S_REPEAT

$$\frac{\mathit{rename\_locals\_stmt}(\texttt{s}) \xrightarrow{\text{ast}} \texttt{s'} \qquad \mathit{rename\_locals\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e'} \qquad \mathsf{optional}[\mathit{rename\_locals\_expr}](\texttt{limit}) \xrightarrow{\text{ast}} \texttt{limit'}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Repeat}(\texttt{s},\texttt{e},\texttt{limit})}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Repeat}(\texttt{s'},\texttt{e'},\texttt{limit'})}^{\texttt{new\_s}}}$$

S_THROW

$$\frac{\mathsf{optional}[\mathit{rename\_locals\_expr}](\texttt{e\_opt}) \xrightarrow{\text{ast}} \texttt{e\_opt'}}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Throw}(\texttt{e\_opt})}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Throw}(\texttt{e\_opt'})}^{\texttt{new\_s}}}$$

S_TRY

$$\frac{\mathit{ast\_label}(\texttt{s}) = \mathsf{S\_Try}}{\mathit{rename\_locals\_stmt}(\texttt{s}) \xrightarrow{\text{ast}} \texttt{not implemented yet}}$$

S_PRINT

$$\frac{\texttt{new\_args} = [\texttt{e} \in \texttt{args} : \mathit{rename\_locals\_expr}(\texttt{e})]}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Print}(\texttt{args},\texttt{newline})}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Print}(\texttt{new\_args},\texttt{newline})}^{\texttt{new\_s}}}$$

S_UNREACHABLE

$$\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Unreachable}}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Unreachable}}^{\texttt{new\_s}}$$

S_PRAGMA

$$\frac{\texttt{new\_args} := [\texttt{e} \in \texttt{args} : \mathit{rename\_locals\_expr}(\texttt{e})]}{\mathit{rename\_locals\_stmt}(\overbrace{\mathsf{S\_Pragma}(\texttt{name},\texttt{args})}^{\texttt{s}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pragma}(\texttt{name},\texttt{new\_args})}^{\texttt{new\_s}}}$$

### ASTRule.RenameLocalsExpr

The helper function

$$\mathit{rename\_locals\_expr}(\overbrace{\mathsf{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{new\_e}}$$

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 `t` yields the type `t'`;
  - define `new_e` as the record construction expression for the record type `t'` and list of field initializers `li`.

- All of the following apply (E_TUPLE):
  - `e` is a tuple expression for the list of expressions `li`;
  - define `li'` as the list obtained by the application of *rename_locals_ty* to each type in `li`;
  - define `new_e` as the tuple expression for the list of expressions `li'`.

- All of the following apply (E_PATTERN):
  - `e` is a pattern expression;
  - this case is not implemented.

**Formally**

E_LITERAL

$$\frac{\mathit{ast\_label}(\texttt{e}) = \mathsf{E\_Literal}}{\mathit{rename\_locals\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \overbrace{\texttt{e}}^{\texttt{new\_e}}}$$

E_VAR

$$\frac{\mathit{rename\_locals\_name}(\texttt{x}) \xrightarrow{\text{ast}} \texttt{x'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Var}(\texttt{x})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Var}(\texttt{x'})}^{\texttt{new\_e}}}$$

E_ARBITRARY

$$\frac{\mathit{rename\_locals\_ty}(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Arbitrary}(\texttt{t})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Arbitrary}(\texttt{t'})}^{\texttt{new\_e}}}$$

E_ATC

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'} \qquad \mathit{rename\_locals\_ty}(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_ATC}(\texttt{e1}, \texttt{t})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_ATC}(\texttt{e1'}, \texttt{t'})}^{\texttt{new\_e}}}$$

E_BINOP

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'} \qquad \mathit{rename\_locals\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Binop}(\texttt{op}, \texttt{e1'}, \texttt{e2'})}^{\texttt{new\_e}}}$$

E_UNOP

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Unop}(\texttt{op}, \texttt{e1})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Unop}(\texttt{op}, \texttt{e1'})}^{\texttt{new\_e}}}$$

E_CALL

$$\frac{\begin{array}{r}\texttt{new\_args} := [\texttt{e} \in \texttt{args} : \mathit{rename\_locals\_expr}(\texttt{e})] \\ \texttt{new\_params} := [\texttt{e} \in \texttt{params} : \mathit{rename\_locals\_expr}(\texttt{e})]\end{array}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Call}\left(\left\{\begin{array}{rcl}\text{name} & : & \texttt{name}, \\ \text{params} & : & \texttt{params}, \\ \text{args} & : & \texttt{args}, \\ \text{call\_type} & : & \texttt{call\_type}\end{array}\right\}\right)}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Call}\left(\left\{\begin{array}{rcl}\text{name} & : & \texttt{name}, \\ \text{params} & : & \texttt{new\_params}, \\ \text{args} & : & \texttt{new\_args}, \\ \text{call\_type} & : & \texttt{call\_type}\end{array}\right\}\right)}^{\texttt{new\_e}}}$$

E_SLICE

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'} \qquad \texttt{slices'} := [\texttt{s} \in \texttt{slices} : \mathit{rename\_locals\_slice}(\texttt{s})]}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Slice}(\texttt{e1}, \texttt{slices})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Slice}(\texttt{e1'}, \texttt{slices'})}^{\texttt{new\_e}}}$$

E_COND

$$\frac{\begin{array}{c}\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'} \\ \mathit{rename\_locals\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2'} \qquad \mathit{rename\_locals\_expr}(\texttt{e3}) \xrightarrow{\text{ast}} \texttt{e3'}\end{array}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Cond}(\texttt{e1}, \texttt{e2}, \texttt{e3})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Cond}(\texttt{e1'}, \texttt{e2'}, \texttt{e3'})}^{\texttt{new\_e}}}$$

E_GETARRAY

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'} \qquad \mathit{rename\_locals\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_GetArray}(\texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_GetArray}(\texttt{e1'}, \texttt{e2'})}^{\texttt{new\_e}}}$$

E_GETENUMARRAY

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'} \qquad \mathit{rename\_locals\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_GetEnumArray}(\texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_GetEnumArray}(\texttt{e1'}, \texttt{e2'})}^{\texttt{new\_e}}}$$

E_GETFIELD

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_GetField}(\texttt{e1}, \texttt{f})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_GetField}(\texttt{e1}, \texttt{f})}^{\texttt{new\_e}}}$$

E_GETFIELDS

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_GetFields}(\texttt{e1}, \texttt{li})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_GetFields}(\texttt{e1'}, \texttt{li})}^{\texttt{new\_e}}}$$

E_GETITEM

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_GetItem}(\texttt{e1}, \texttt{i})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_GetItem}(\texttt{e1'}, \texttt{i})}^{\texttt{new\_e}}}$$

E_RECORD

$$\frac{\mathit{rename\_locals\_ty}(\texttt{y}) \xrightarrow{\text{ast}} \texttt{t'}}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Record}(\texttt{t}, \texttt{li})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Record}(\texttt{t'}, \texttt{li})}^{\texttt{new\_e}}}$$

E_TUPLE

$$\frac{\texttt{li'} := [\texttt{e} \in \texttt{li} : \mathit{rename\_locals\_expr}(\texttt{e})]}{\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Tuple}(\texttt{li})}^{\texttt{e}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Tuple}(\texttt{li'})}^{\texttt{new\_e}}}$$

E_PATTERN

$$\mathit{rename\_locals\_expr}(\overbrace{\mathsf{E\_Pattern}(\_, \_)}^{\texttt{e}}) \xrightarrow{\text{ast}} \texttt{not implemented yet}$$

### ASTRule.RenameLocalsLexpr

The helper function

$$\mathit{rename\_locals\_lexpr}(\overbrace{\mathsf{lexpr}}^{\texttt{le}}) \longrightarrow \overbrace{\mathsf{lexpr}}^{\texttt{new\_le}}$$

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 `fl`;
  * 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 `fl`.

- All of the following apply (LE_DESTRUCTURING):
  * `le` is an assignable multi-expression for the list of assignable expressions `les`;
  * define `les'` as the list obtained by applying *rename_locals_lexpr* to each assignable expression in `les`;
  * define `new_le` as the assignable multi-expression for the list of assignable expressions `les'`.

**Formally**

LE_DISCARD

$$\overbrace{\mathit{rename\_locals\_lexpr}(\mathsf{LE\_Discard})}^{\texttt{le}} \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Discard}}^{\texttt{new\_le}}$$

LE_VAR

$$\frac{\mathit{rename\_locals\_name}(\texttt{x}) \xrightarrow{\text{ast}} \texttt{x'}}{\overbrace{\mathit{rename\_locals\_lexpr}(\mathsf{LE\_Var}(\texttt{x}))}^{\texttt{le}} \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Var}(\texttt{x'})}^{\texttt{new\_le}}}$$

LE_SLICE

$$\frac{\begin{array}{c}\mathit{rename\_locals\_lexpr}(\texttt{le1}) \xrightarrow{\text{ast}} \texttt{le1'} \\ \texttt{slices'} := [\texttt{s} \in \texttt{slices} : \mathit{rename\_locals\_slice}(\texttt{s})]\end{array}}{\overbrace{\mathit{rename\_locals\_lexpr}(\mathsf{LE\_Slice}(\texttt{le1}, \texttt{slices}))}^{\texttt{le}} \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Slice}(\texttt{le'}, \texttt{slices'})}^{\texttt{new\_le}}}$$

LE_SETARRAY

$$\frac{\mathit{rename\_locals\_lexpr}(\texttt{le1}) \xrightarrow{\text{ast}} \texttt{le1'} \qquad \mathit{rename\_locals\_expr}(\texttt{i}) \xrightarrow{\text{ast}} \texttt{i'}}{\overbrace{\mathit{rename\_locals\_lexpr}(\mathsf{LE\_SetArray}(\texttt{le1}, \texttt{i}))}^{\texttt{le}} \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_SetArray}(\texttt{le1'}, \texttt{i'})}^{\texttt{new\_le}}}$$

LE_SETFIELD

$$\frac{\mathit{rename\_locals\_lexpr}(\texttt{le1}) \xrightarrow{\text{ast}} \texttt{le1'}}{\overbrace{\mathit{rename\_locals\_lexpr}(\mathsf{LE\_SetField}(\texttt{le1}, \texttt{f}))}^{\texttt{le}} \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_SetField}(\texttt{le1'}, \texttt{f})}^{\texttt{new\_le}}}$$

LE_SETFIELDS

$$\frac{\mathit{rename\_locals\_lexpr}(\texttt{le1}) \xrightarrow{\text{ast}} \texttt{le1'}}{\mathit{rename\_locals\_lexpr}(\overbrace{\mathsf{LE\_SetFields}(\texttt{le1}, \texttt{fl})}^{\texttt{le}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_SetFields}(\texttt{le1'}, \texttt{fl})}^{\texttt{new\_le}}}$$

LE_DESTRUCTURING

$$\frac{\texttt{les'} := [\texttt{l} \in \texttt{les} : \mathit{rename\_locals\_lexpr}(\texttt{l})]}{\mathit{rename\_locals\_lexpr}(\overbrace{\mathsf{LE\_Destructuring}(\texttt{les})}^{\texttt{le}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Destructuring}(\texttt{les'})}^{\texttt{new\_le}}}$$

### ASTRule.RenameLocalsLDI

The helper function

$$\mathit{rename\_locals\_ldi}(\overbrace{\mathsf{local\_decl\_item}}^{\texttt{ldi}}) \longrightarrow \overbrace{\mathsf{local\_decl\_item}}^{\texttt{new\_ldi}}$$

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

VAR

$$\mathit{rename\_locals\_ldi}(\overbrace{\mathsf{LDI\_Var}(\texttt{x})}^{\texttt{ldi}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDI\_Var}(\texttt{x'})}^{\texttt{new\_ldi}}$$

TUPLE

$$\frac{\texttt{names'} := [\texttt{name} \in \texttt{names} : \mathit{rename\_locals\_name}(\texttt{name})]}{\mathit{rename\_locals\_ldi}(\overbrace{\mathsf{LDI\_Tuple}(\texttt{names})}^{\texttt{ldi}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDI\_Tuple}(\texttt{names'})}^{\texttt{new\_ldi}}}$$

### ASTRule.RenameLocalsConstraint

The helper function

$$\mathit{rename\_locals\_constraint}(\overbrace{\mathsf{int\_constraint}}^{\mathtt{c}}) \longrightarrow \overbrace{\mathsf{int\_constraint}}^{\mathtt{new\_c}}$$

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

$$\frac{\mathit{rename\_locals\_expr}(\mathtt{e}) \xrightarrow{\mathsf{ast}} \mathtt{e'}}{\mathit{rename\_locals\_constraint}(\overbrace{\mathsf{Constraint\_Exact}(\mathtt{e})}^{\mathtt{c}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{Constraint\_Exact}(\mathtt{e'})}^{\mathtt{new\_c}}}$$

RANGE

$$\frac{\mathit{rename\_locals\_expr}(\mathtt{e1}) \xrightarrow{\mathsf{ast}} \mathtt{e1'} \qquad \mathit{rename\_locals\_expr}(\mathtt{e2}) \xrightarrow{\mathsf{ast}} \mathtt{e2'}}{\mathit{rename\_locals\_constraint}(\overbrace{\mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{c}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{Constraint\_Range}(\mathtt{e1'}, \mathtt{e2'})}^{\mathtt{new\_c}}}$$

### ASTRule.RenameLocalsSlice

The helper function

$$\mathit{rename\_locals\_slice}(\overbrace{\mathsf{slice}}^{\texttt{slice}}) \longrightarrow \overbrace{\mathsf{slice}}^{\texttt{new\_slice}}$$

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 $S \in \{\mathsf{Slice\_Length}, \mathsf{Slice\_Range}, \mathsf{Slice\_Star}\}$;
  - 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

SINGLE

$$\frac{\mathit{rename\_locals\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e'}}{\mathit{rename\_locals\_slice}(\overbrace{\mathsf{Slice\_Single}(\texttt{e})}^{\texttt{slice}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{Slice\_Single}(\texttt{e'})}^{\texttt{new\_slice}}}$$

NON_SINGLE

$$\frac{S \in \{\mathsf{Slice\_Length}, \mathsf{Slice\_Range}, \mathsf{Slice\_Star}\} \quad \mathit{rename\_locals\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1'} \quad \mathit{rename\_locals\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2'}}{\mathit{rename\_locals\_slice}(\overbrace{S(\texttt{e1}, \texttt{e2})}^{\texttt{slice}}) \xrightarrow{\text{ast}} \overbrace{S(\texttt{e1'}, \texttt{e2'})}^{\texttt{new\_slice}}}$$

#### ASTRule.RenameLocalsName

The helper function

$$\mathit{rename\_locals\_name}(\overbrace{\mathsf{identifier}}^{\texttt{name}}) \longrightarrow \overbrace{\mathsf{identifier}}^{\texttt{new\_name}}$$

renames the local identifier `name`, yielding the identifier `new_name`.

#### Prose

define `new_name` as the string concatenation of `__stdlib_local_` and `name`.

#### Formally

$$\mathit{rename\_locals\_name}(\texttt{name}) \xrightarrow{\text{ast}} \overbrace{\texttt{\_\_stdlib\_local\_} + \texttt{name}}^{\texttt{new\_name}}$$

## 29.3 Marking Standard Library Functions

#### ASTRule.SetBuiltin

The helper function

$$\mathit{set\_builtin}(\overbrace{\mathsf{decl}}^{\texttt{decl}}) \longrightarrow \overbrace{\mathsf{decl}}^{\texttt{decl'}} \cup \overbrace{\mathsf{TBuildError}}^{\#\mathsf{BE}}$$

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 $\mathtt{TRUE}^{/\!/\mathtt{BE\_BD}}$;
- view `decl` as `D_Func`(`func_sig`);
- define `func_sig'` as `func_sig` with its builtin flag set to `TRUE`;
- define `decl'` as `D_Func`(`func_sig'`).

#### Formally

$$\frac{\mathit{check}(\mathit{ast\_label}(\texttt{decl}) = \texttt{D\_Func}, \texttt{BE\_BD}) \xrightarrow{\text{type}} \texttt{TRUE} /\!/ \ \#\texttt{BE} \qquad \texttt{decl} \overset{\text{is}}{=} \texttt{D\_Func}(\texttt{func\_sig}) \qquad \texttt{func\_sig'} := \texttt{func\_sig}[\text{builtin} \mapsto \texttt{TRUE}]}{\mathit{set\_builtin}(\texttt{decl}) \xrightarrow{\text{ast}} \texttt{D\_Func}(\texttt{func\_sig'})}$$

# Chapter 30

# Side Effects

This chapter defines a static *side effect analysis*. The analysis aims to answer which expressions are *pure* or symbolically evaluable. For purity, the analysis is *sound*: it proves that a sufficient condition for purity holds.

Intuitively, a pure expression is one whose evaluation does not affect the evaluation of any further expressions. In other words, a pure 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 Time Frames

We divide side effects by *time frames*, which indicate the phase where a side effect occurs:

**Constant** Contains effects that take place during static evaluation (see Chapter 31). That is, during typechecking.

**Execution** Contains effects that take place during semantic evaluation.

Formally, time frames are totally ordered via $<_{\text{time}}$ as follows:

$$\mathsf{TimeFrame} \triangleq \{\mathsf{Constant} <_{\text{time}} \mathsf{Execution}\}$$

Additionally, we define the less-than-or-equal ordering as follows:

$$f \leq_{\text{time}} f' \triangleq f <_{\text{time}} f' \vee f = f' \ .$$

We now define some helper functions for constructing time frames.

### TypingRule.TimeFrameLDK

The function

$$\mathit{time\_frame\_ldk}(\overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}) \longrightarrow \overbrace{\mathsf{TimeFrame}}^{\texttt{t}}$$

constructs a time frame `t` from a local declaration keyword `ldk`.

### Example: The Time Frame of Local Storage Declarations

The specification in Listing 30.1 shows examples of local storage declarations and their corresponding time frames (in comments).

Listing 30.1: The time frame of local storage declarations

```
func main() => integer
begin
    // local storage declaration    Time frame
    constant c = 10;                // Constant
    let l = 20;                     // Execution
    var v = 30;                     // Execution
    return 0;
end;
```

#### Prose

Define `t` as Constant if `ldk` is LDK_Constant, and Execution otherwise.

#### Formally

$$\mathit{time\_frame\_ldk}(\texttt{ldk}) \xrightarrow{\text{type}} \begin{cases} \text{Constant} & \text{if } \texttt{ldk} = \text{LDK\_Constant} \\ \text{Execution} & \text{if } \texttt{ldk} = \text{LDK\_Let} \\ \text{Execution} & \text{if } \texttt{ldk} = \text{LDK\_Var} \end{cases}$$

#### TypingRule.TimeFrameGDK

The function

$$\mathit{time\_frame\_gdk}(\overbrace{\text{global\_decl\_keyword}}^{\texttt{gdk}}) \longrightarrow \overbrace{\text{TimeFrame}}^{\texttt{t}}$$

constructs a time frame `t` from a global declaration keyword `gdk`.

### Example: The Time Frame of Global Storage Declarations

The specification in Listing 30.2 shows examples of global storage declarations and their corresponding time frames (in comments).

Listing 30.2: The time frame of global storage declarations

```
// Global storage declarations  Time frame
constant c = 10;                // Constant
config cfg : integer = 20;      // Execution
let l = 30;                     // Execution
var v = 40;                     // Execution
```

#### Prose

Define `t` as Constant if `gdk` is GDK_Constant, Execution if `gdk` is GDK_Config, Execution if `gdk` is GDK_Let, and Execution if `gdk` is GDK_Var.

**Formally**

$$\mathit{time\_frame\_gdk}(\texttt{gdk}) \xrightarrow{\text{type}} \begin{cases} \text{Constant} & \text{if } \texttt{gdk} = \text{GDK\_Constant} \\ \text{Execution} & \text{if } \texttt{gdk} = \text{GDK\_Config} \\ \text{Execution} & \text{if } \texttt{gdk} = \text{GDK\_Let} \\ \text{Execution} & \text{if } \texttt{gdk} = \text{GDK\_Var} \end{cases}$$

## 30.2 Side Effect Descriptors

We now define side effect descriptors, which are configurations used to describe side effects, as explained below:

$$\begin{aligned} \text{TSideEffect} \triangleq \{ & \\ & \text{ReadLocal}(\overbrace{\text{identifier}}^{\texttt{x}}, \overbrace{\text{TimeFrame}}^{\texttt{t}}, \overbrace{\mathbb{B}}^{\texttt{immutable}}) \cup \\ & \text{WriteLocal}(\overbrace{\text{identifier}}^{\texttt{x}}) \cup \\ & \text{ReadGlobal}(\overbrace{\text{identifier}}^{\texttt{x}}, \overbrace{\text{TimeFrame}}^{\texttt{t}}, \overbrace{\mathbb{B}}^{\texttt{immutable}}) \cup \\ & \text{WriteGlobal}(\overbrace{\text{identifier}}^{\texttt{x}}) \cup \\ & \text{ThrowException}(\overbrace{\text{identifier}}^{\texttt{x}}) \cup \\ & \text{RecursiveCall}(\overbrace{\text{identifier}}^{\texttt{f}}) \cup \\ & \text{PerformsAssertions} \cup \\ & \text{NonDeterministic} \\ \} & \end{aligned}$$

**ReadLocal** a local read side effect descriptor describes an evaluation of a construct that leads to reading the value of the local storage element `x` at the time frame `t` where `immutable` is `TRUE` if and only if `x` was declared as an immutable local storage element (that is, `constant` or `let`);

**WriteLocal** a local write side effect descriptor describes an evaluation of a construct that leads to modifying the value of the local storage element `x`;

**ReadGlobal** a global read side effect descriptor describes an evaluation of a construct that leads to reading the value of the global storage element `x` at the time frame `t` where `immutable` is `TRUE` if and only if `x` was declared as an immutable local storage element (that is, `constant`, `config`, or `let`);

**WriteGlobal** a global write side effect descriptor describes an evaluation of a construct that leads to modifying the value of the global storage element `x`;

**ThrowException** an exception side effect descriptor describes an evaluation of a construct that leads to raising an exception whose type is named `x`;

**RecursiveCall** a recursive call side effect descriptor describes an evaluation of a construct that leads to calling the recursive function `f`;

**PerformsAssertions** an assertion side effect descriptor describes an evaluation of a construct that leads to evaluating an `assert` statement;

**NonDeterministic** a non-determinism side effect descriptor describes an evaluation of a construct that leads to evaluating a non-deterministic expression (either `ARBITRARY` or a library function call known to be non-deterministic).

We now define a few helper functions over time frames.

### TypingRule.TimeFrame

The function

$$time\_frame(\overbrace{\mathsf{TSideEffect}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathsf{TimeFrame}}^{\mathtt{t}}$$

retrieves the time frame `t` from a side effect descriptor `s`.

### Example: The Time Frame of Side Effect Descriptors

The specification in Listing 30.3 shows examples of expressions and statements and, in comments, their corresponding sets of side effect descriptors and time frames.

Listing 30.3: The time frame of side effect descriptors

```
constant cg = 20;
let lg = 30;
config cfg : integer{0..100} = 40;
var vg : integer = 50;

type MyException of exception;

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 effect for RHS expression   Time frame
    var x : integer;
    - = x; // ReadLocal(x, Execution, FALSE)    Execution
    constant cl = 10;
    let ll : integer = 20;
    - = cl; // ReadLocal(cl, Constant, TRUE)    Constant
    - = ll; // ReadLocal(ll, Execution, TRUE)   Constant
    - = cg; // ReadGlobal(cg, Constant, TRUE)   Constant
    - = lg; // ReadGlobal(lg, Execution, TRUE)  Execution
    - = cg; // ReadGlobal(cg, Execution, TRUE)  Execution
    - = vg; // ReadGlobal(vg, Execution, FALSE) Execution

            // NonDeterministic                 Execution
    - = ARBITRARY: integer;

            // ReadLocal(x, Execution, FALSE)   Execution
            // PerformsAssertions               Constant
    - = x as integer{0};
```

```
            // RecursiveCall                    Execution
    - = factorial(10);

            // Side effect for LHS expression  Time frame
    vg = 60; // WriteGlobal(vg)                 Execution
    x = 70;  // WriteLocal(x)                   Execution

            // Statement                        Time frame
            // ThrowException                   Execution
    throw MyException{-};
    assert 1 == 1; // PerformsAssertions       Constant

    return 0;
end;
```

### Prose

One of the following applies:

- All of the following apply (READ_LOCAL):
  - `s` is a local read side effect descriptor for the time frame `t`.
- All of the following apply (READ_GLOBAL):
  - `s` is a global read side effect descriptor for the time frame `t`.
- All of the following apply (PERFORMS_ASSERTIONS):
  - `s` is an assertion side effect descriptor;
  - define `t` as Constant.
- All of the following apply (OTHER):
  - `s` is either a local write side effect descriptor, a global write side effect descriptor, a non-determinism side effect descriptor, a recursive call side effect descriptor, or a exception side effect descriptor;
  - define `t` as Execution.

### Formally

READ_LOCAL

$$time\_frame(\overbrace{\mathsf{ReadLocal}(\_, \mathtt{t}, \_)}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \mathtt{t}$$

READ_GLOBAL

$$time\_frame(\overbrace{\mathsf{ReadGlobal}(\_, \mathtt{t}, \_)}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \mathtt{t}$$

PERFORMS_ASSERTIONS

$$time\_frame(\overbrace{\mathsf{PerformsAssertions}}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Constant}}^{\mathtt{t}}$$

$$\frac{\text{OTHER} \quad \texttt{config\_dom}(\texttt{s}) \in \left\{ \begin{array}{c} \mathsf{WriteLocal}, \\ \mathsf{WriteGlobal}, \\ \mathsf{NonDeterministic}, \\ \mathsf{RecursiveCall}, \\ \mathsf{ThrowException} \end{array} \right\}}{\mathit{time\_frame}(\texttt{s}) \xrightarrow{\text{type}} \overbrace{\mathsf{Execution}}^{\texttt{t}}}$$

### TypingRule.SideEffectIsPure

$$\mathit{side\_effect\_is\_pure}(\overbrace{\mathsf{TSideEffect}}^{\texttt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

defines whether a side effect descriptors `s` is considered *pure*, yielding the result in `b`. Intuitively, a *pure* side effect descriptor helps to establish that an expression evaluates without modifying values of storage elements.

### Example: Pure and Symbolically Evaluable Side Effect Descriptors

The specification in Listing 30.4 shows examples of expressions and statements, and their corresponding side effect descriptors and whether those are pure or not and whether they are symbolically evaluable or not (in comments).

Notice that the call to `factorial(10)` is pure, even though it is recursive. The side-effect analysis adds RecursiveCall in TypingRule.DeclareOneFunc, while annotating a strongly-connected component of (mutually-recursive) subprograms, until finally removing it in TypingRule.TypecheckDecl. In this example, `factorial` is annotated before `main`, which contains the call to `factorial(10)`, which is why the analysis is able to determine that the call expression is pure and the sets of side effect descriptors is empty.

Listing 30.4: Side effect descriptors and whether they are pure and symbolically evaluable

```
constant cg = 20;
let lg = 30;
config cfg : integer{0..100} = 40;
var vg : integer = 50;

type MyException of exception;

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 effect for RHS expression   Pure?   Symbolically Evaluable?
    var x : integer;
    - = x; // ReadLocal(x, Execution, FALSE)    TRUE    FALSE
    constant cl = 10;
    let ll : integer = 20;
    - = cl; // ReadLocal(cl, Constant, TRUE)    TRUE    TRUE
```

```
    - = ll; // ReadLocal(ll, Execution, TRUE)    TRUE    TRUE
    - = cg; // ReadGlobal(cg, Constant, TRUE)    TRUE    TRUE
    - = lg; // ReadGlobal(lg, Execution, TRUE)   TRUE    TRUE
    - = cg; // ReadGlobal(cg, Execution, TRUE)   TRUE    TRUE
    - = vg; // ReadGlobal(vg, Execution, FALSE)  TRUE    TRUE

            // NonDeterministic                  TRUE    FALSE
    - = ARBITRARY: integer;

            // ReadLocal(x, Execution, FALSE)    TRUE    FALSE
            // PerformsAssertions                TRUE    FALSE
    - = x as integer{0};

            // None                              TRUE   FALSE
    - = factorial(10);

             // Side effect for LHS expression   Pure?
    vg = 60; // WriteGlobal(vg)                  FALSE   FALSE
    x = 70;  // WriteLocal(x)                    FALSE   FALSE

             // Statement                        Pure?
             // ThrowException                   FALSE   FALSE
    throw MyException{-};
    assert 1 == 1; // PerformsAssertions         TRUE    FALSE

    return 0;
end;
```

### Prose

Define `b` as `TRUE` if and only if `s` is either a local read side effect descriptor, a global read side effect descriptor, a non-determinism side effect descriptor, or an assertion side effect descriptor.

### Formally

$$\frac{\mathtt{b} := \mathtt{config\_dom}(\mathtt{s}) \in \{\mathsf{ReadLocal}, \mathsf{ReadGlobal}, \mathsf{NonDeterministic}, \mathsf{PerformsAssertions}\}}{\mathit{side\_effect\_is\_pure}(\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

### TypingRule.SideEffectIsSymbolicallyEvaluable

$$\mathit{side\_effect\_symbolically\_evaluable}(\overbrace{\mathsf{TSideEffect}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

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 failing assertions, without modifying any storage element, and always yielding the same result, that is, deterministically.

See Example: Pure and Symbolically Evaluable Side Effect Descriptors.

### Prose

Define `b` as `TRUE` if and only if `s` is either a local read side effect descriptor associated with an immutable storage element, or a global read side effect descriptor associated with an immutable storage element.

**Formally**

$$\frac{\texttt{b} := \texttt{s} = \mathsf{ReadLocal}(\_, \_, \texttt{TRUE}) \vee \texttt{s} = \mathsf{ReadGlobal}(\_, \_, \texttt{TRUE})}{\mathit{side\_effect\_symbolically\_evaluable}(\texttt{s}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.LDKIsImmutable

The function

$$\mathit{ldk\_is\_immutable}(\overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}) \xrightarrow{\text{type}} \overbrace{\texttt{TRUE}}^{\texttt{b}}$$

tests whether the local declaration keyword `ldk` relates to an immutable storage element, yielding the result in `b`.

### Example: Immutability of Local Declarations

The following table shows local declarations appearing in Listing 30.3 and whether their corresponding local declaration keywords are considered immutable or not.

| Local Declaration | Immutable? |
|---|---|
| `var x : integer;` | `FALSE` |
| `constant cl = 10;` | `TRUE` |
| `let ll : integer = 20;` | `TRUE` |

**Prose**

Define `b` as `TRUE` if and only if `ldk` corresponds to either the keyword `constant` or the keyword `let`.

**Formally**

$$\mathit{ldk\_is\_immutable}(\texttt{ldk}) \xrightarrow{\text{type}} \overbrace{\texttt{ldk} \in \{\mathsf{LDK\_Constant}, \mathsf{LDK\_Let}\}}^{\texttt{b}}$$

### TypingRule.GDKIsImmutable

The function

$$\mathit{gdk\_is\_immutable}(\overbrace{\mathsf{global\_decl\_keyword}}^{\texttt{gdk}}) \xrightarrow{\text{type}} \overbrace{\texttt{TRUE}}^{\texttt{b}}$$

tests whether the global declaration keyword `gdk` relates to an immutable storage element, yielding the result in `b`.

### Example: Immutability of Global Declarations

The following table shows global declarations appearing in Listing 30.3 and whether their corresponding global declaration keywords are considered immutable or not.

| Global Declaration | Immutable? |
|---|---|
| `constant cg = 20;` | TRUE |
| `let lg = 30;` | TRUE |
| `config cfg : integer{0..100} = 40;` | TRUE |
| `var vg : integer = 50;` | FALSE |

### Prose

Define `b` as TRUE if and only if `gdk` corresponds to either the keyword `constant`, the keyword `config`, or the keyword `let`.

### Formally

$$\mathit{gdk\_is\_immutable}(\texttt{gdk}) \xrightarrow{\text{type}} \overbrace{\texttt{gdk} \in \{\mathsf{GDK\_Constant}, \mathsf{GDK\_Config}, \mathsf{GDK\_Let}\}}^{\texttt{b}}$$

## 30.3 Side Effect Sets

### TypingRule.MaxTimeFrame

The function

$$\mathit{max\_time\_frame}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \longrightarrow \overbrace{\mathsf{TimeFrame}}^{\texttt{f}}$$

defines the maximal time frame for a set of side effect descriptors `ses`, which is returned in `f`. (If `ses` is empty, the result is Constant.)

We define the maximum of a set of time frames $\max_{\text{time}} : \mathcal{P}(\mathsf{TimeFrame}) \to \mathsf{TimeFrame}$ as follows:

$$\max_{\text{time}}(T) \triangleq t \text{ such that } t \in T \land \forall t' \in T.\ t' \leq_{\text{time}} t \ .$$

### Example: Maximal Time Frame

In Listing 30.5, the time frame of the expression `g` is Constant, whereas the time frame of the expression `gl` is Execution. Therefore, the time frame of the expression `g + gl` is Execution.

Listing 30.5: Maximal time frame

```
constant g = 5;
let gl : integer{0..1000} = 500;

type Data of bits(g * 2) {
    [0] LSB
};
```

```
// Legal as there are no bitfields.
type Data2 of bits(g + gl) {
};

// The next declaration in comment is illegal, since the maximal time frame
// for `g + gl` is Execution time.
// type Data3 of bits(g + gl) {
//     [0] LSB
// };
```

### Prose

All of the following apply:

- define `reads` as the subset of `ses` that contains only side effect descriptors that are either local read side effect descriptor or global read side effect descriptor;
- One of the following applies:
  - All of the following apply (EXECUTION):
    - `reads` is not equal to `ses`;
    - define `f` as Execution.
  - All of the following apply (READS):
    - `reads` is equal to `ses`;
    - define `time_frames` as the time frames appearing in the side effect descriptors in `reads`;
    - define `f` as the greatest time frame in the union of `time_frames` and the singleton set for Constant, where $\leq_{\text{time}}$ is used to compare any two time frames.

### Formally

EXECUTION

$$\frac{\texttt{reads} := \{\texttt{s} \in \texttt{ses} \mid \texttt{config\_dom}(\texttt{s}) \in \{\mathsf{ReadLocal}, \mathsf{ReadGlobal}\}\} \qquad \texttt{ses} \neq \texttt{reads}}{\mathit{max\_time\_frame}(\texttt{ses}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Execution}}^{\texttt{f}}}$$

READS

$$\frac{\begin{array}{r}\texttt{reads} := \{\texttt{s} \in \texttt{ses} \mid \texttt{config\_dom}(\texttt{s}) \in \{\mathsf{ReadLocal}, \mathsf{ReadGlobal}\}\} \qquad \texttt{ses} = \texttt{reads} \\ \texttt{time\_frames} := \{\mathit{time\_frame}(\texttt{f'}) \mid \texttt{f'} \in \texttt{reads}\} \\ \texttt{f} := \max_{\text{time}}(\texttt{time\_frames} \cup \{\mathsf{Constant}\})\end{array}}{\mathit{max\_time\_frame}(\texttt{ses}) \xrightarrow{\mathsf{type}} \texttt{f}}$$

### TypingRule.SESIsSymbolicallyEvaluable

The function

$$\mathit{is\_symbolically\_evaluable}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{bv}}$$

tests whether a set of side effect descriptors `ses` are all symbolically evaluable, yielding the result in `b`.

See Example: Checking Whether Expressions are Symbolically Evaluable.

### Prose

Define `b` as `TRUE` if and only if every side effect descriptor `s` in `ses` is symbolically evaluable.

### Formally

$$\frac{\texttt{b} := \bigwedge_{\texttt{s} \in \texttt{ses}} \mathit{side\_effect\_symbolically\_evaluable}(\texttt{s})}{\mathit{is\_symbolically\_evaluable}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.CheckSymbolicallyEvaluable

The function

$$\mathit{check\_symbolically\_evaluable}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \longrightarrow \{\texttt{TRUE}\} \cup \mathsf{TTypeError}$$

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 30.6, 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 30.6: Symbolically evaluable expressions

```
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 `e` in `tenv` yields `b`;
- the result is `TRUE` if `b` is `TRUE`, otherwise it is a type error indicating that the expression is not symbolically evaluable.

### Formally

$$\frac{\mathit{is\_\ symbolically\_\ evaluable}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{b} \qquad \mathit{check}(\texttt{b}, \texttt{NotSymbolicallyEvaluable}) \longrightarrow \texttt{TRUE} \ /\!/\ \texttt{\#TE}}{\mathit{check\_\ symbolically\_\ evaluable}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

### TypingRule.SESIsPure

The function

$$\mathit{ses\_\ is\_\ pure}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\text{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

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: Pure Expressions**

In Listing 30.7, the following expressions are checked for purity:

- `y > x` inside the assertion statement;
- `ARBITRARY : integer{1..1000} > g` inside the assertion statement, since `ARBITRARY` expressions are pure;
- `x` and `y` in the for statement.

Listing 30.7: Pure 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 30.8 and Listing 30.9, are ill-typed due to expressions that are not pure where pure expressions are expected (as explained in the comments).

Listing 30.8: Impure 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 pure.
    assert y > write_side_effecting();
    return 0;
end;
```

Listing 30.9: Impure 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 pure.
    for i = x to write_side_effecting() do
        g = g + 1;
    end;
    return 0;
end;
```

**Formally**

$$\frac{\bigwedge_{\mathtt{s} \in \mathtt{ses}} \mathit{side\_effect\_is\_pure}(\mathtt{s})}{\mathit{ses\_is\_pure}(\mathtt{ses}) \xrightarrow{\text{type}} \mathtt{TRUE}}$$

### TypingRule.SESIsDeterministic

The function

$$\mathit{ses\_is\_deterministic}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{bv}}$$

tests whether the NonDeterministic side effect descriptor is not included in `ses`, yielding the result in `b`.

### Example: Deterministic and Non-deterministic Expressions

The expressions `x` and `y` that are used as the start and end expressions for the for statement in Listing 30.10 are deterministic, which is why that statement is well-typed.

Listing 30.10: Deterministic expressions

```
var g : integer = 0;

func main() => integer
begin
    var x = 15;
    var y = x + 9;

    for i = x to y do
        g = g + 1;
    end;
    return 0;
end;
```

In contrast, the expression `ARBITRARY : integer{1..1000}` used as the end expression for the for statement in Listing 30.11 is non-deterministic, which is why that statement is ill-typed.

Listing 30.11: A non-deterministic expressions

```
var g : integer = 0;

func main() => integer
begin
    var x = 15;
    var y = x + 9;

    // The following statement is illegal, since
    // the expression ‘ARBITRARY : integer{1..1000}‘ is not deterministic.
    for i = x to ARBITRARY : integer{1..1000} do
        g = g + 1;
    end;
    return 0;
end;
```

### Prose

Define `b` as `TRUE` if and only if NonDeterministic is not included in `ses`.

### Formally

$$\mathit{ses\_is\_deterministic}(\texttt{ses}) \xrightarrow{\text{type}} \overbrace{\mathsf{NonDeterministic} \notin \texttt{ses}}^{\texttt{b}}$$

### TypingRule.SESIsBefore

The function

$$\mathit{ses\_is\_before}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}, \overbrace{\mathsf{TimeFrame}}^{\texttt{t}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

tests whether the time frames of side effect descriptors in `ses` are all less than or equal to the time frame `t`, yielding the result in `b`.

### Example: Time Frame Upper Bounds

The specification in Listing 30.12 is well-typed. Specifically, all expressions match their expected time frames (see comments).

Listing 30.12: Time frame upper bounds

```
constant WORD_SIZE = 32;

// The time frame of a bitvector width for a bitvector
// type with bitfields must be Constant.
type Data of bits(WORD_SIZE * 2) {
    [0] LSB
};

var g_execution_time : integer{0..10} = 5;
```

```
config c : integer{0..WORD_SIZE * 8} = WORD_SIZE * 4;

let gl : integer{0..10} = g_execution_time;

func main() => integer
begin
    // The time frame of an expression initializing
    // a local storage constant must be Constant.
    constant ADDRESS_SPACE_SIZE = 2 ^ WORD_SIZE;
    return 0;
end;
```

The specification in Listing 30.13 is ill-typed, since expressions appearing in constraints must be constant time expressions.

Listing 30.13: Time frame violation

```
let g : integer{0..10} = 1;

// Illegal as ‘g‘ is in the Execution time frame.
type Data of bits(g * 2) {
    [0] LSB
};
```

See also Example: Declaring Global Storage.

### Prose

Define `b` as `TRUE` if and only if the maximal time frame of all side effect descriptors in `ses` is less than or equal to `t` with respect to $\leq_{\text{time}}$.

### Formally

$$\mathit{ses\_is\_before}(\texttt{ses}, \texttt{t}) \xrightarrow{\text{type}} \overbrace{\mathit{max\_time\_frame}(\texttt{ses}) \leq_{\text{time}} \texttt{t}}^{\texttt{b}}$$

# Chapter 31

# Static Evaluation

In this chapter, we define how to statically evaluate an expression to yield a literal value.

### TypingRule.StaticEval

The function

$$\mathit{static\_eval}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \overbrace{\mathsf{literal}}^{\mathsf{v}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 $\mathit{static\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{v}$ 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 $\mathsf{L\_Int}(32)$) in order to determine that the 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

NORMAL_LITERAL

$$\frac{\mathit{static\_env\_to\_env}(\mathsf{tenv}) \xrightarrow{\text{type}} \mathtt{env} \qquad \mathit{eval\_expr}(\mathtt{env}, \mathtt{e}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{NV\_Literal}(\mathtt{v}), \_)}{\mathit{static\_eval}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} \mathtt{v}}$$

NORMAL_NON_LITERAL

$$\frac{\mathit{static\_env\_to\_env}(\mathsf{tenv}) \xrightarrow{\text{type}} \mathtt{env} \qquad \mathit{eval\_expr}(\mathtt{env}, \mathtt{e}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{x}, \_) \qquad \mathtt{x} \neq \mathtt{NV\_Literal}(\_)}{\mathit{static\_eval}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_SEF})}$$

ABNORMAL

$$\frac{\mathit{static\_env\_to\_env}(\mathsf{tenv}) \xrightarrow{\mathsf{type}} \mathtt{env} \qquad \mathit{eval\_expr}(\mathtt{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} C \qquad \mathtt{config\_dom}(C) \in \{\mathsf{Throwing}, \mathsf{DynError}\}}{\mathit{static\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_SEF})}$$

**TypingRule.StaticEnvToEnv**

The function

$$\mathit{static\_env\_to\_env}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}) \xrightarrow{\mathsf{type}} \overbrace{\mathbb{E}}^{\mathsf{env}}$$

transforms the constants defined in the static environment `tenv` into an environment `env`.

**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:

$$\{\mathsf{constant\_values} \mapsto \{\mathtt{HALF\_WORD\_SIZE} \mapsto \mathsf{L\_Int}(32)\}, \ldots\}$$

is transformed into the dynamic environment `env`, defined as follows:

$$(\overbrace{\{\mathtt{HALF\_WORD\_SIZE} \mapsto \mathtt{Int}(32)\}}^{G^{\mathsf{env}}}, \overbrace{\emptyset_\lambda}^{L^{\mathsf{env}}}) \ .$$

**Prose**

All of the following apply:

- define the global dynamic environment `global` as the map that bind each `id` in the domain of $G^{\mathsf{tenv}}$.constant_values to `NV_Literal`(`l`) if $G^{\mathsf{tenv}}$.constant_values(`id`) = `l`;
- define the local dynamic environment `local` as the map that bind each `id` in the domain of $L^{\mathsf{tenv}}$.constant_values to `NV_Literal`(`l`) if $L^{\mathsf{tenv}}$.constant_values(`id`) = `l`;
- define the environment `env` to have the static component `tenv` and the dynamic environment (`global`, `local`);

**Formally**

$$\frac{\mathtt{global} := [\mathtt{id} \mapsto \mathtt{NV\_Literal}(\mathtt{l}) \mid G^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{id}) = \mathtt{l}] \qquad \mathtt{local} := [\mathtt{id} \mapsto \mathtt{NV\_Literal}(\mathtt{l}) \mid L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{id}) = \mathtt{l}]}{\mathit{static\_env\_to\_env}(\mathsf{tenv}) \xrightarrow{\mathsf{type}} \overbrace{(\mathsf{tenv}, (\mathtt{global}, \mathtt{local}))}^{\mathsf{env}}}$$

# 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* $\mathtt{t}$ *and* $\mathtt{s}$ *and static environment* $\mathsf{tenv}$*, we say that* $\mathtt{t}$ *is a* domain subset *of* $\mathtt{s}$ *in* $\mathsf{tenv}$*, if the following condition holds:*

$$\begin{array}{l} \mathit{domain\_subset}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \triangleq \\ \quad \forall \mathsf{denv} \in \mathbb{DE}.\ \mathit{dyn\_dom}((\mathsf{tenv}, \mathsf{denv}), \mathtt{t}) \subseteq \mathit{dyn\_dom}((\mathsf{tenv}, \mathsf{denv}), \mathtt{s}) \ . \end{array} \tag{32.1}$$

For example, consider the assignment

$$\texttt{var x : } \overbrace{\texttt{integer\{1,2,3\}}}^{\texttt{s}} \texttt{ = ARBITRARY : } \overbrace{\texttt{integer\{1,2\}}}^{\texttt{t}}\texttt{;}$$

It is well-typed, since the dynamic domain of `integer{1,2,3}` is $\{\texttt{Int}(1), \texttt{Int}(2), \texttt{Int}(3)\}$ in every dynamic environment, which is a superset of the dynamic domain of `integer{1,2}`, which is $\{\texttt{Int}(1), \texttt{Int}(2)\}$ 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 $\mathit{symdom\_subset\_test}(\mathsf{tenv}, \mathtt{t}, \mathtt{s})$.

**Definition 46 (Sound Domain Subset Test)** *A predicate*

$$\mathit{symdom\_subset\_test}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathit{ty}}^{\mathtt{t}}, \overbrace{\mathit{ty}}^{\mathtt{s}}) \longrightarrow \mathbb{B}$$

*is* sound *if the following condition holds:*

$$\begin{array}{l} \forall \mathtt{t}, \mathtt{s} \in \mathit{ty}.\ \mathsf{tenv} \in \mathbb{SE}. \\ \quad \mathit{symdom\_subset\_test}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathit{type}} \mathit{TRUE} \implies \mathit{domain\_subset}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \ . \end{array} \tag{32.2}$$

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

**True Negative:** indeed, `t` is not a domain subset of `s` in the static environment `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:

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

$$\begin{array}{rcl} \mathsf{symdom} & \longrightarrow & \mathsf{Finite}(\mathcal{P}_{\mathrm{fin}}(\mathbb{Z}) \setminus \emptyset) \\ & | & \mathsf{ConstrainedDom}(\mathsf{int\_constraint}) \\ \mathsf{symdom\_or\_top} & \longrightarrow & \mathsf{Top} \\ & | & \mathsf{Subdomains}(\mathsf{symdom}^{+}) \end{array}$$

- 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 `vcs`; 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

$$symdom\_subset\_unions(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{symdom\_or\_top}}^{\texttt{sd1}}, \overbrace{\mathsf{symdom\_or\_top}}^{\texttt{sd2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

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 *symdom_normalize* to `symdoms1` yields `symdoms1_norm`;
  - applying *symdom_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 *symdom_subset* holds for `s1` and `s2` in tenv.

### Formally

RIGHT_TOP

$$\mathit{symdom\_subset\_unions}(\mathsf{tenv}, \mathtt{sd1}, \overbrace{\mathsf{Top}}^{\mathtt{sd2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{TRUE}}^{\mathtt{b}}$$

LEFT_TOP_RIGHT_NOT_TOP

$$\frac{\mathtt{sd1} \neq \mathsf{Top}}{\mathit{symdom\_subset\_unions}(\mathsf{tenv}, \overbrace{\mathsf{Top}}^{\mathtt{sd1}}, \mathtt{sd2}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

$$\frac{\begin{array}{l}\mathit{symdom\_normalize}(\mathtt{symdoms1}) \xrightarrow{\mathsf{type}} \mathtt{symdoms1\_norm} \\ \mathit{symdom\_normalize}(\mathtt{symdoms1}) \xrightarrow{\mathsf{type}} \mathtt{symdoms2\_norm} \\ \mathtt{b} \;:=\; \forall \mathtt{s1} \in \mathtt{symdoms1\_norm}.\ \exists \mathtt{s2} \in \mathtt{symdoms2\_norm}. \\ \qquad \mathit{symdom\_subset}(\mathsf{tenv}, \mathtt{s1}, \mathtt{s2})\end{array}}{\mathit{symdom\_subset\_unions}(\mathsf{tenv}, \overbrace{\mathsf{Subdomains}(\mathtt{symdoms1})}^{\mathtt{sd1}}, \overbrace{\mathsf{Subdomains}(\mathtt{symdoms1})}^{\mathtt{sd2}}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

### TypingRule.SymdomNormalize

The function

$$\mathit{symdom\_normalize}(\overbrace{\mathsf{symdom}^+}^{\texttt{symdoms}}) \longrightarrow \overbrace{\mathsf{symdom}^+}^{\texttt{new\_symdoms}}$$

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` $\mathsf{Finite}(\texttt{xs})$ and `tail` `others`, if `xs` is non-empty, and `others`, otherwise.

### Formally

$$\frac{\begin{array}{c}\texttt{others} = [\texttt{s} \in \texttt{symdoms} \mid \texttt{config\_dom}(\texttt{s}) \neq \mathsf{Finite}] \\ \texttt{finite\_domains} = [\texttt{s} \in \texttt{symdoms} \mid \texttt{config\_dom}(\texttt{s}) = \mathsf{Finite}] \\ \texttt{xs} := \bigcup_{\mathsf{Finite}(\texttt{s}) \in \texttt{finite\_domains}} \texttt{s} \\ \texttt{new\_symdoms} := \mathsf{choice}(\texttt{xs} \neq \emptyset, [\mathsf{Finite}(\texttt{xs})] + \texttt{others}, \texttt{others})\end{array}}{\mathit{symdom\_normalize}(\texttt{symdoms}) \xrightarrow{\mathsf{type}} \texttt{new\_symdomsnew\_symdoms}}$$

### TypingRule.SymdomOfType

The function

$$\mathit{symdom\_of\_type}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{t}}) \longrightarrow \overbrace{\mathsf{symdom\_or\_top}}^{\texttt{d}}$$

transforms a type `t` in a static environment $\mathsf{tenv}$ into a symbolic domain `d`. It assumes its input type has an underlying type which is an integer.

### Prose

One of the following applies:

- All of the following apply (INT_UNCONSTRAINED):
  - `t` is the unconstrained integer type;
  - define `d` as Top, which intuitively represents the entire set of integers.
- All of the following apply (INT_PARAMETERIZED):
  - `t` is the parameterized integer type for the identifier `id`;
  - define `d` as the singleton list consisting of the symbolic constrained integer domain with a single constraint for the variable expression for `id`, that is, ConstrainedDom(Constraint_Exact(E_Var(`id`))).
- All of the following apply (INT_WELL_CONSTRAINED):
  - `t` is the well-constrained integer type for the list of constraints `vcs`;
  - applying *symdom_of_constraint* to `tenv` and $\texttt{c}_\texttt{i}$, for every index `i` in the list of indices for `vcs`, yields $\texttt{d}_\texttt{i}$;
  - define `d` as the list symbolic integer domains consisting of $\texttt{d}_\texttt{i}$, for every index `i` in the list of indices for `vcs`.
- All of the following apply (T_NAMED):
  - `t` is the named type for identifier `id`;
  - applying *make_anonymous* to `t` in `tenv` yields `t1`;
  - applying *symdom_of_type* to `t1` in `tenv` yields `d`.

### Formally

INT_UNCONSTRAINED

$$\mathit{symdom\_of\_type}(\texttt{tenv}, \overbrace{\mathsf{unconstrained\_integer}}^{\texttt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Top}}^{\texttt{d}}$$

INT_PARAMETERIZED

$$\mathit{symdom\_of\_type}(\texttt{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{id}))}^{\texttt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Subdomains}([\mathsf{ConstrainedDom}(\mathsf{Constraint\_Exact}(\mathsf{E\_Var}(\texttt{id})))])}^{\texttt{d}}$$

INT_WELL_CONSTRAINED

$$\frac{\texttt{i} \in \mathit{indices}(\texttt{vcs}) : \mathit{symdom\_of\_constraint}(\texttt{tenv}, \texttt{vcs}[\texttt{i}]) \xrightarrow{\mathsf{type}} \texttt{d}_\texttt{i} \qquad \texttt{c} := \texttt{i} \in \mathit{indices}(\texttt{vcs})\texttt{d}_\texttt{i}}{\mathit{symdom\_of\_type}(\texttt{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{vcs}))}^{\texttt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Subdomains}(\texttt{vis})}^{\texttt{d}}}$$

T_NAMED

$$\frac{\texttt{t} = \mathsf{T\_Named}(\texttt{id}) \qquad \mathit{make\_anonymous}(\texttt{t}) \xrightarrow{\text{type}} \texttt{t1} \qquad \mathit{symdom\_of\_type}(\text{tenv}, \texttt{t1}) \xrightarrow{\text{type}} \texttt{d}}{\mathit{symdom\_of\_type}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{d}}$$

### TypingRule.SymdomOfWidthExpr

The function

$$\mathit{symdom\_of\_width\_expr}(\overbrace{\mathsf{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\mathsf{symdom\_or\_top}}^{\texttt{d}}$$

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:

$$\mathsf{Subdomains}([\ \overbrace{\overbrace{\overbrace{2}^{\mathsf{E\_Literal(L\_Int)}}}^{\mathsf{E\_Binop}}}^{\mathsf{Constraint\_Exact}} * \overbrace{x}^{\mathsf{E\_Var}}\ ])\ .$$

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

$$\mathit{symdom\_of\_width\_expr}(\texttt{e}) \xrightarrow{\text{type}} \overbrace{\mathsf{Subdomains}([\ \mathsf{ConstrainedDom}(\mathsf{Constraint\_Exact}(\texttt{e}))\ ])}^{\texttt{d}}$$

### TypingRule.SymdomOfConstraint

The function

$$\mathit{symdom\_of\_constraint}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\mathsf{c}}) \longrightarrow \overbrace{\mathsf{symdom}}^{\mathsf{d}}$$

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($\{n \mid \mathtt{v1} \leq n \leq \mathtt{v2}\}$)).

### Formally

EXACT

$$\frac{\mathit{symdom\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{v} \qquad \mathtt{d} := \mathsf{choice}(\mathtt{v} = \mathsf{Top}, \mathsf{ConstrainedDom}(\mathtt{c}), \mathsf{Finite}(\{\mathtt{v}\}))}{\mathit{symdom\_of\_constraint}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\mathsf{type}} \mathtt{d}}$$

RANGE

$$\frac{\mathit{symdom\_eval}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathtt{v1} \qquad \mathit{symdom\_eval}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{v2} \qquad \mathtt{d} := \mathsf{choice}(\mathtt{v1} = \mathsf{Top} \vee \mathtt{v2} = \mathsf{Top}, \mathsf{ConstrainedDom}(\mathtt{c}), \mathsf{Finite}(\{n \mid \mathtt{v1} \leq n \leq \mathtt{v2}\}))}{\mathit{symdom\_of\_constraint}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) \xrightarrow{\mathsf{type}} \mathtt{d}}$$

### TypingRule.SymdomEval

The function

$$\mathit{symdom\_eval}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \overbrace{\mathbb{Z}}^{n} \cup \{\mathsf{Top}\}$$

symbolically simplifies the integer-typed expression `e` and returns the resulting integer or Top if the result of the simplification is not an integer.

We assume that `e` has been annotated as it is part of the constraint for an integer type, and therefore applying *normalize* to it does not yield a type error.

### Prose

One of the following applies:

- All of the following apply (INTEGER):
  - applying *normalize* to `e` in `tenv` yields the expression `e1`;
  - applying *static_eval* to `e1` in `tenv` yields the integer literal for $n$.
- All of the following apply (TOP):
  - applying *normalize* to `e` in `tenv` yields the expression `e1`;
  - applying *static_eval* to `e1` in `tenv` yields $\top$.
  - the result is Top.

### Formally

INTEGER

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{e1} \qquad \mathit{static\_eval}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(n)}{\mathit{symdom\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} n}$$

TOP

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{e1} \qquad \mathit{static\_eval}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \top}{\mathit{symdom\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{Top}}$$

### TypingRule.SymdomSubset

The function

$$\mathit{symdom\_subset}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{symdom}}^{\mathsf{cd1}}, \overbrace{\mathsf{symdom}}^{\mathsf{cd2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathsf{b}}$$

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_FINITE):
  - `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**

FINITE_FINITE

$$\mathit{symdom\_subset}(\mathsf{tenv}, \overbrace{\mathsf{Finite}(\mathtt{s1})}^{\mathtt{cd1}}, \overbrace{\mathsf{Finite}(\mathtt{s2})}^{\mathtt{cd2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{s1} \subseteq \mathtt{s2}}^{\mathtt{b}}$$

CONSTRAINED_CONSTRAINED_EQUAL

$$\frac{\mathit{constraint\_equal}(\mathsf{tenv}, \mathtt{c1}, \mathtt{c2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}{\mathit{symdom\_subset}(\mathsf{tenv}, \overbrace{\mathsf{ConstrainedDom}(\mathtt{c1})}^{\mathtt{cd1}}, \overbrace{\mathsf{ConstrainedDom}(\mathtt{c2})}^{\mathtt{cd2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{TRUE}}^{\mathtt{b}}}$$

CONSTRAINED_CONSTRAINED_NON_EQUAL

$$\frac{\begin{array}{c}\mathit{constraint\_equal}(\mathsf{tenv}, \mathtt{c1}, \mathtt{c2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{approx\_constraint}(\mathsf{tenv}, \mathsf{Over}, \mathtt{c1}) \xrightarrow{\mathsf{type}} \mathtt{s1} \\ \mathit{approx\_constraint}(\mathsf{tenv}, \mathsf{Under}, \mathtt{c2}) \xrightarrow{\mathsf{type}} \mathtt{s2}\end{array}}{\mathit{symdom\_subset}(\mathsf{tenv}, \overbrace{\mathsf{ConstrainedDom}(\mathtt{c1})}^{\mathtt{cd1}}, \overbrace{\mathsf{ConstrainedDom}(\mathtt{c2})}^{\mathtt{cd2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{s1} \subseteq \mathtt{s2}}^{\mathtt{b}}}$$

FINITE_CONSTRAINED

$$\frac{\mathit{approx\_constraints}(\mathsf{tenv}, \mathsf{Under}, \mathtt{c2}) \xrightarrow{\mathsf{type}} \mathtt{s2}}{\mathit{symdom\_subset}(\mathsf{tenv}, \overbrace{\mathsf{Finite}(\mathtt{cd1})}^{\mathtt{s1}}, \overbrace{\mathsf{ConstrainedDom}(\mathtt{c2})}^{\mathtt{cd2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{s1} \subseteq \mathtt{s2}}^{\mathtt{b}}}$$

CONSTRAINED_FINITE

$$\frac{\mathit{approx\_constraints}(\mathsf{tenv}, \mathsf{Over}, \mathtt{c1}) \xrightarrow{\mathsf{type}} \mathtt{s1}}{\mathit{symdom\_subset}(\mathsf{tenv}, \overbrace{\mathsf{ConstrainedDom}(\mathtt{c1})}^{\mathtt{is1}}, \overbrace{\mathsf{Finite}(\mathtt{s2})}^{\mathtt{is2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{s1} \subseteq \mathtt{s2}}^{\mathtt{b}}}$$

### TypingRule.ApproxConstraints

The function

$$\mathit{approx\_constraints}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\{\mathsf{Over}, \mathsf{Under}\}}^{\mathsf{approx}}, \overbrace{\mathsf{int\_constraint}^{+}}^{\mathtt{cs}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^{\mathtt{s}}$$

conservatively approximates the non-empty list of constraints `cs` by a set of integers `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 $\mathtt{s}_\mathtt{c}$, for every constraint `c` in $\mathtt{cs}^{/\!/\mathbb{Z}}$;
  - define `s` as the union of all sets $\mathtt{s}_\mathtt{c}$, 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 $\mathtt{s}_\mathtt{c}$, for every constraint `c` in `cs`;
  - define `s` as the intersection of all sets $\mathtt{s}_\mathtt{c}$, for every constraint `c` in `cs`.

### Formally

OVER

$$\frac{\mathtt{c} \in \mathtt{cs} : \mathit{approx\_constraint}(\mathsf{tenv}, \mathsf{Over}, \mathtt{c}) \xrightarrow{\mathsf{type}} \mathtt{s}_\mathtt{c} \;/\!/\; \mathbb{Z}}{\mathit{approx\_constraints}(\mathsf{tenv}, \overbrace{\mathsf{Over}}^{\mathsf{approx}}, \mathtt{cs}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{c} \in \mathtt{cs}} \mathtt{s}_\mathtt{c}}^{\mathtt{s}}}$$

UNDER

$$\frac{\mathtt{c} \in \mathtt{cs} : \mathit{approx\_constraint}(\mathsf{tenv}, \mathtt{approx}, \mathtt{c}) \xrightarrow{\mathsf{type}} \mathtt{s}_\mathtt{c}}{\mathit{approx\_constraints}(\mathsf{tenv}, \overbrace{\mathsf{Under}}^{\mathsf{approx}}, \mathtt{cs}) \xrightarrow{\mathsf{type}} \overbrace{\bigcap_{\mathtt{c} \in \mathtt{cs}} \mathtt{s}_\mathtt{c}}^{\mathtt{s}}}$$

### TypingRule.ApproxConstraint

The function

$$approx\_constraint(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\{\mathsf{Over}, \mathsf{Under}\}}^{\text{approx}}, \overbrace{\mathsf{int\_constraint}}^{\text{c}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^{\text{s}}$$

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 $\mathtt{s}^{/\!/\mathbb{Z}}$.
- All of the following apply (RANGE_OVER):
  - `c` is a range constraint for the lower end expression `e1` and upper end expression `e2`;

  - approximating the minimal integer in the set of integers represented by the expression `e1` in the static environment `tenv` yields $\texttt{z1}^{/\!/\mathbb{Z}}$;
  - approximating the maximal integer in the set of integers represented by the expression `e2` in the static environment `tenv` yields $\texttt{z2}^{/\!/\mathbb{Z}}$;
  - 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**

EXACT

$$\frac{\mathit{approx\_expr}(\mathsf{tenv}, \mathtt{approx}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{s}^{/\!/\mathbb{Z}}}{\mathit{approx\_constraint}(\mathsf{tenv}, \mathtt{approx}, \overbrace{\mathsf{Constraint\_Exact}(\mathtt{e})}^{\mathtt{c}}) \xrightarrow{\mathsf{type}} \mathtt{s}}$$

RANGE_OVER

$$\frac{\begin{array}{c}\mathtt{approx} = \mathsf{Over} \qquad \mathit{approx\_expr\_min}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{z1}^{/\!/\mathbb{Z}} \\ \mathit{approx\_expr\_max}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{z2}^{/\!/\mathbb{Z}} \\ \mathtt{s\_interval} := \{\mathtt{z} \mid \mathtt{z1} \leq \mathtt{z} \leq \mathtt{z2}\} \qquad \mathit{approx\_bottom\_top}(\mathtt{approx}) \xrightarrow{\mathsf{type}} \mathtt{s\_bottom\_top} \\ \mathtt{s} := \mathsf{choice}(\mathtt{z1} \leq \mathtt{z2}, \mathtt{s\_interval}, \mathtt{s\_bottom\_top})\end{array}}{\mathit{approx\_constraint}(\mathsf{tenv}, \mathtt{approx}, \overbrace{\mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{c}}) \xrightarrow{\mathsf{type}} \mathtt{s}}$$

RANGE_UNDER

$$\frac{\begin{array}{c}\texttt{approx} = \mathsf{Under} \\ \mathit{approx\_expr\_max}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{z1} \qquad \mathit{approx\_expr\_min}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{z2} \\ \mathtt{s\_interval} := \{\mathtt{z} \mid \mathtt{z1} \leq \mathtt{z} \leq \mathtt{z2}\} \qquad \mathit{approx\_bottom\_top}(\texttt{approx}) \xrightarrow{\mathsf{type}} \mathtt{s\_bottom\_top} \\ \mathtt{s} := \mathsf{choice}(\mathtt{z1} \leq \mathtt{z2}, \mathtt{s\_interval}, \mathtt{s\_bottom\_top})\end{array}}{\mathit{approx\_constraint}(\mathsf{tenv}, \texttt{approx}, \overbrace{\mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{c}}) \xrightarrow{\mathsf{type}} \mathtt{s}}$$

### TypingRule.ApproxExprMin

The function

$$\mathit{approx\_expr\_min}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\mathbb{Z} \cup \{-\infty\}}^{\mathtt{z}}$$

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 `z` 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.

$$\frac{\mathit{approx\_expr}(\mathsf{tenv}, \mathsf{Over}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{s}}{\mathit{approx\_expr\_min}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \overbrace{\min_{-\infty}(\mathtt{s})}^{\mathtt{z}}}$$

### TypingRule.ApproxExprMax

The function

$$\mathit{approx\_expr\_max}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\mathbb{Z} \cup \{+\infty\}}^{\mathtt{z}}$$

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

$$\frac{\mathit{approx\_expr}(\mathsf{tenv}, \mathsf{Over}, \mathsf{e}) \xrightarrow{\text{type}} \mathtt{s}}{\mathit{approx\_expr\_max}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \overbrace{\max_{+\infty}(\mathtt{s})}^{\mathtt{z}}}$$

### TypingRule.ApproxBottomTop

The function

$$\mathit{approx\_bottom\_top}(\overbrace{\{\mathsf{Under}, \mathsf{Over}\}}^{\text{approx}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^{\mathtt{s}}$$

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**

$$\mathit{approx\_bottom\_top}(\texttt{approx}) \xrightarrow{\text{type}} \overbrace{\mathsf{choice}(\texttt{approx} = \mathsf{Under}, \emptyset, \mathbb{Z})}^{\texttt{s}}$$

### TypingRule.IntsetToConstraints

The function

$$\mathit{intset\_to\_constraints}(\overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^{\texttt{s}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs}}$$

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 $\overbrace{\mathsf{E\_Literal(L\_Int)}\ a\ ..\ \mathsf{E\_Literal(L\_Int)}\ b}^{\text{Constraint\_Range}}$, otherwise.

**Formally**

$$\texttt{intervals} := \{a..b \mid a..b \in \texttt{s} \land \forall (c..d) \in \texttt{s}.\ a..b \not\subseteq c..d\}$$

$$\texttt{cs} := [a..b \in \texttt{intervals} : \mathsf{choice}(a = b, \overbrace{\mathsf{E\_Literal(L\_Int)}\ a}^{\text{Constraint\_Exact}}, \overbrace{\mathsf{E\_Literal(L\_Int)}\ a\ ..\ \mathsf{E\_Literal(L\_Int)}\ b}^{\text{Constraint\_Range}})]$$

$$\mathit{intset\_to\_constraints}(\texttt{s}) \xrightarrow{\text{type}} \texttt{cs}$$

### TypingRule.ApproxExpr

The function

$$\mathit{approx\_expr}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\{\mathsf{Over}, \mathsf{Under}\}}^{\texttt{approx}}, \overbrace{\mathsf{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^{\texttt{s}}$$

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 $\texttt{s'}^{/\!/\mathbb{Z}}$;
    * 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 $\texttt{s1}^{/\!/\mathbb{Z}}$;
    * approximating the set integers represented by the expression `e2` in the static environment `tenv` with the symbol `approx` yields the set of integers $\texttt{s2}^{/\!/\mathbb{Z}}$;
    * 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 $\texttt{s2}^{/\!/\mathbb{Z}}$;
  - approximating the set integers represented by the expression `e3` in the static environment `tenv` with the symbol `approx` yields the set of integers $\texttt{s3}^{/\!/\mathbb{Z}}$;
  - 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**

LITERAL_INT

$$\mathit{approx\_expr}(\mathsf{tenv}, \mathtt{approx}, \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(\mathtt{z}))}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\{\mathtt{z}\}}^{\mathtt{s}}$$

LITERAL_NON_INT

$$\frac{\mathit{ast\_label}(\mathtt{l}) \neq \mathsf{L\_Int} \qquad \mathit{approx\_bottom\_top}(\mathtt{approx}) \xrightarrow{\mathsf{type}} \mathtt{s}}{\mathit{approx\_expr}(\mathsf{tenv}, \mathtt{approx}, \overbrace{\mathsf{E\_Literal}(\mathtt{l})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{s}}$$

In the following inference rule, the application of *type_of* is guaranteed not to result in a type error, since *symdom_subset* is only applied to annotated types.

VAR_OVER

$$\frac{\mathit{type\_of}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \mathtt{t} \qquad \mathit{approx\_type}(\mathsf{tenv}, \mathsf{Over}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{s}}{\mathit{approx\_expr}(\mathsf{tenv}, \overbrace{\mathsf{Over}}^{\mathtt{approx}}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{s}}$$

VAR_UNDER

$$\mathit{approx\_expr}(\mathsf{tenv}, \overbrace{\mathsf{Under}}^{\mathtt{approx}}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{s}}$$

UNOP

$$\frac{\begin{array}{l}\mathit{approx\_expr}(\mathsf{tenv}, \mathtt{approx}, \mathtt{e'}) \xrightarrow{\mathsf{type}} \mathtt{s'}^{/\!/\mathbb{Z}} \\ \mathtt{s} := \{\mathit{unop\_literals}(\mathtt{op}, \mathsf{L\_Int}(\mathtt{z})) \mid \mathtt{z} \in \mathtt{s'}\}\end{array}}{\mathit{approx\_expr}(\mathsf{tenv}, \mathtt{approx}, \overbrace{\mathsf{E\_Unop}(\mathtt{op}, \mathtt{e'})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{s}}$$

BINOP

$$\frac{\begin{array}{c}\mathit{approx\_expr}(\mathsf{tenv}, \mathsf{approx}, \mathsf{e1}) \xrightarrow{\text{type}} \mathsf{s1} \mathbin{//} \mathbb{Z} \\ \mathit{approx\_expr}(\mathsf{tenv}, \mathsf{approx}, \mathsf{e2}) \xrightarrow{\text{type}} \mathsf{s2} \mathbin{//} \mathbb{Z} \\ \mathit{annotate\_constraint\_binop}(\mathsf{tenv}, \mathsf{s1}, \mathsf{s2}) \xrightarrow{\text{type}} (\mathsf{s'}, \mathsf{plf}) \\ \mathit{approx\_constraints}(\mathsf{tenv}, \mathsf{approx}, \mathsf{s'}) \xrightarrow{\text{type}} \mathsf{s\_approx} \\ \mathsf{s} := \begin{cases} \mathsf{s\_approx} & \text{if } \mathsf{plf} = \mathsf{Precision\_Full} \vee (\mathsf{plf} = \mathsf{Precision\_Lost} \wedge \mathsf{approx} = \mathsf{Under}) \\ \mathbb{Z} & \text{if } \mathsf{plf} = \mathsf{Precision\_Lost} \wedge \mathsf{approx} = \mathsf{Over} \end{cases}\end{array}}{\mathit{approx\_expr}(\mathsf{tenv}, \mathsf{approx}, \overbrace{\mathsf{E\_Binop}(\mathsf{op}, \mathsf{e1}, \mathsf{e2})}^{\mathsf{e}}) \xrightarrow{\text{type}} \mathsf{s}}$$

COND

$$\frac{\begin{array}{c}\mathit{approx\_expr}(\mathsf{tenv}, \mathsf{approx}, \mathsf{e2}) \xrightarrow{\text{type}} \mathsf{s2} \mathbin{//} \mathbb{Z} \\ \mathit{approx\_expr}(\mathsf{tenv}, \mathsf{approx}, \mathsf{e3}) \xrightarrow{\text{type}} \mathsf{s3} \mathbin{//} \mathbb{Z} \\ \mathsf{s} := \mathsf{choice}(\mathsf{approx} = \mathsf{Over}, \mathsf{s2} \cup \mathsf{s3}, \mathsf{s2} \cap \mathsf{s3})\end{array}}{\mathit{approx\_expr}(\mathsf{tenv}, \mathsf{approx}, \overbrace{\mathsf{E\_Cond}(\_, \mathsf{e2}, \mathsf{e3})}^{\mathsf{e}}) \xrightarrow{\text{type}} \mathsf{s}}$$

OTHER

$$\frac{\mathit{ast\_label}(\mathsf{e}) \notin \{\mathsf{E\_Literal}, \mathsf{E\_Var}, \mathsf{E\_Unop}, \mathsf{E\_Binop}, \mathsf{E\_Cond}\} \quad \mathit{approx\_bottom\_top}(\mathsf{approx}) \xrightarrow{\text{type}} \mathsf{s}}{\mathit{approx\_expr}(\mathsf{tenv}, \mathsf{approx}, \mathsf{e}) \xrightarrow{\text{type}} \mathsf{s}}$$

### TypingRule.ApproxType

The function

$$\mathit{approx\_type}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\{\mathsf{Over}, \mathsf{Under}\}}^{\mathsf{approx}}, \overbrace{\mathsf{ty}}^{\mathsf{t}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^{\mathsf{s}}$$

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**

NAMED

$$\frac{\mathit{is\_named}(\texttt{t}) \qquad \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{t'} \qquad \mathit{approx\_type}(\texttt{tenv}, \texttt{approx}, \texttt{t'}) \xrightarrow{\text{type}} \texttt{s}}{\mathit{approx\_type}(\texttt{tenv}, \texttt{approx}, \texttt{t}) \xrightarrow{\text{type}} \texttt{s}}$$

INT_WELLCONSTRAINED

$$\frac{\texttt{t} = \mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{cs})) \qquad \mathit{approx\_constraints}(\texttt{tenv}, \texttt{approx}, \texttt{cs}) \xrightarrow{\text{type}} \texttt{s}}{\mathit{approx\_type}(\texttt{tenv}, \texttt{approx}, \texttt{t}) \xrightarrow{\text{type}} \texttt{s}}$$

OTHER

$$\frac{\neg \mathit{is\_named}(\texttt{t}) \wedge \neg \mathsf{is\_well\_constrained\_integer}(\texttt{t}) \qquad \mathit{approx\_bottom\_top}(\texttt{approx}) \xrightarrow{\text{type}} \texttt{s}}{\mathit{approx\_type}(\texttt{tenv}, \texttt{approx}, \texttt{t}) \xrightarrow{\text{type}} \texttt{s}}$$

**TypingRule.ConstraintBinop**

The function

$$\mathit{constraint\_binop}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs1}}, \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs2}}) \longrightarrow \overbrace{\mathsf{constraint\_kind}}^{\texttt{new\_cs}}$$

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` ∈ *indices*(`cs1`) and `j` ∈ *indices*(`cs2`), yields $\texttt{c}_{\texttt{i,j}}$;
  - define `new_cs` as the list of constraints $\texttt{c}_{\texttt{i,j}}$, for every `i` ∈ *indices*(`cs1`) and `j` ∈ *indices*(`cs2`).
- All of the following apply (MOD):
  - `op` is `MOD`;
  - applying *constraint_mod* to `cs2[j]`, for every `j` ∈ *indices*(`cs2`), yields $\texttt{c}_{\texttt{j}}$;
  - define `new_cs` as $\texttt{c}_{\texttt{j}}$, for every `j` ∈ *indices*(`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` ∈ *indices*(`cs1`) and `j` ∈ *indices*(`cs2`), yields $\texttt{c}_{\texttt{i,j}}$;
  - define `new_cs` as the list of constraints $\texttt{c}_{\texttt{i,j}}$, for every `i` ∈ *indices*(`cs1`) and `j` ∈ *indices*(`cs2`).

### Formally

EXTREMITIES

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{DIV}, \texttt{DIVRM}, \texttt{MUL}, \texttt{PLUS}, \texttt{MINUS}, \texttt{SHR}, \texttt{SHL}\} \\ \texttt{i} \in \mathit{indices}(\texttt{cs1}) : \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \\ \mathit{apply\_binop\_extremities}(\texttt{cs1[i]}, \texttt{cs2[j]}) \xrightarrow{\text{type}} \texttt{c}_{\texttt{i,j}} \\ \texttt{new\_cs} := [\texttt{i} \in \mathit{indices}(\texttt{cs1}) : \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \texttt{c}_{\texttt{i,j}}]\end{array}}{\mathit{constraint\_binop}(\texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

MOD

$$\frac{\texttt{op} = \texttt{MOD} \qquad \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \mathit{constraint\_mod}(\texttt{cs2[j]}) \xrightarrow{\text{type}} \texttt{c}_{\texttt{j}} \qquad \texttt{new\_cs} = [\texttt{j} \in \mathit{indices}(\texttt{cs2}) : \texttt{c}_{\texttt{j}}]}{\mathit{constraint\_binop}(\texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

POW

$$\frac{\begin{array}{c}\texttt{op} = \texttt{POW} \qquad \texttt{i} \in \mathit{indices}(\texttt{cs1}) : \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \\ \mathit{constraint\_pow}(\texttt{cs1[i]}, \texttt{cs2[j]}) \xrightarrow{\text{type}} \texttt{c}_{\texttt{i,j}} \\ \texttt{new\_cs} := [\texttt{i} \in \mathit{indices}(\texttt{cs1}) : \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \texttt{c}_{\texttt{i,j}}]\end{array}}{\mathit{constraint\_binop}(\texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

### TypingRule.ApplyBinopExtremities

The function

$$\mathit{apply\_binop\_extremities}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c1}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c2}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^{*}}^{\texttt{new\_cs}}$$

yields a list of constraints `new_cs` for the constraints `c1` and `c2`, which are needed to include range constraints for cases where the binary operation `op` yields a dynamic error.

### 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 $\overbrace{\texttt{a op c}}^{\text{E\_Binop}}$.

- 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 $\overbrace{\overbrace{\texttt{a' op c}}^{\text{E\_Binop}} .. \overbrace{\texttt{b' op c}}^{\text{E\_Binop}}}^{\text{Constraint\_Range}}$ 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 $\overbrace{\overbrace{\texttt{a op c'}}^{\text{E\_Binop}} .. \overbrace{\texttt{a op d'}}^{\text{E\_Binop}}}^{\text{Constraint\_Range}}$ 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 $\overbrace{\overbrace{\texttt{a' op c'}}^{\text{E\_Binop}} .. \overbrace{\texttt{b' op d'}}^{\text{E\_Binop}}}^{\text{Constraint\_Range}}$ 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

$$\textit{apply\_binop\_extremities}(\texttt{op}, \overbrace{\mathsf{Constraint\_Exact}(\texttt{a})}^{\texttt{c1}}, \overbrace{\mathsf{Constraint\_Exact}(\texttt{c})}^{\texttt{c2}}) \xrightarrow{\text{type}} \overbrace{[\mathsf{Constraint\_Exact}(\overbrace{\texttt{a op c}}^{\text{E\_Binop}})]}^{\texttt{new\_cs}}$$

RANGE_EXACT

$$\frac{\mathit{possible\_extremities\_left}(\texttt{op}, \texttt{a}, \texttt{b}) \xrightarrow{\text{type}} \texttt{extpairs} \qquad \texttt{new\_cs} := [(\texttt{a'}, \texttt{b'}) \in \texttt{extpairs} : \overbrace{\overbrace{\texttt{a' op c}}^{\text{E\_Binop}} .. \overbrace{\texttt{b' op c}}^{\text{E\_Binop}}}^{\text{Constraint\_Range}}]}{\mathit{apply\_binop\_extremities}(\texttt{op}, \overbrace{\text{Constraint\_Range}(\texttt{a}, \texttt{b})}^{\texttt{c1}}, \overbrace{\text{Constraint\_Exact}(\texttt{c})}^{\texttt{c2}}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

EXACT_RANGE

$$\frac{\mathit{possible\_extremities\_right}(\texttt{op}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \texttt{extpairs} \qquad \texttt{new\_cs} := [(\texttt{c'}, \texttt{d'}) \in \texttt{extpairs} : \overbrace{\overbrace{\texttt{a op c'}}^{\text{E\_Binop}} .. \overbrace{\texttt{a op d'}}^{\text{E\_Binop}}}^{\text{Constraint\_Range}}]}{\mathit{apply\_binop\_extremities}(\texttt{op}, \overbrace{\text{Constraint\_Exact}(\texttt{a})}^{\texttt{c1}}, \overbrace{\text{Constraint\_Range}(\texttt{c}, \texttt{d})}^{\texttt{c2}}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

RANGE_RANGE

$$\frac{\begin{array}{c}\mathit{possible\_extremities\_left}(\texttt{op}, \texttt{a}, \texttt{b}) \xrightarrow{\text{type}} \texttt{extpairs\_a\_b} \\ \mathit{possible\_extremities\_right}(\texttt{op}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \texttt{extpairs\_c\_d} \\ \texttt{new\_cs} := [(\texttt{a'}, \texttt{b'}) \in \texttt{extpairs\_a\_b}, (\texttt{c'}, \texttt{d'}) \in \texttt{extpairs\_c\_d} : \overbrace{\overbrace{\texttt{a' op c'}}^{\text{E\_Binop}} .. \overbrace{\texttt{b' op d'}}^{\text{E\_Binop}}}^{\text{Constraint\_Range}}]\end{array}}{\mathit{apply\_binop\_extremities}(\texttt{op}, \overbrace{\text{Constraint\_Range}(\texttt{a}, \texttt{b})}^{\texttt{c1}}, \overbrace{\text{Constraint\_Range}(\texttt{c}, \texttt{d})}^{\texttt{c2}}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

### TypingRule.PossibleExtremitiesLeft

The function

$$\mathit{possible\_extremities\_left}(\overbrace{\text{binop}}^{\texttt{op}}, \overbrace{\text{expr}}^{\texttt{a}}, \overbrace{\text{expr}}^{\texttt{b}}) \longrightarrow \overbrace{(\text{expr} \times \text{expr})^*}^{\texttt{extpairs}}$$

yields a list of pairs of expressions `extpairs` given the binary operation `op` and pair of expressions `a` and `b`, which are needed to form constraints for cases where applying `op` to `a` and `b` would lead to a dynamic error.

### 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

$$\mathit{possible\_extremities\_left}(\overbrace{\texttt{MUL}}^{\texttt{op}}, \texttt{a}, \texttt{b}) \xrightarrow{\text{type}} \overbrace{[(\texttt{a},\texttt{a}), (\texttt{a},\texttt{b}), (\texttt{b},\texttt{a}), (\texttt{b},\texttt{b})]}^{\texttt{extpairs}}$$

OTHER

$$\frac{\texttt{op} \in \{\texttt{DIV}, \texttt{DIVRM}, \texttt{SHR}, \texttt{SHL}, \texttt{PLUS}, \texttt{MINUS}\}}{\mathit{possible\_extremities\_left}(\texttt{op}, \texttt{a}, \texttt{b}) \xrightarrow{\text{type}} \overbrace{[(\texttt{a},\texttt{b})]}^{\texttt{extpairs}}}$$

### TypingRule.PossibleExtremitiesRight

The function

$$\mathit{possible\_extremities\_right}(\overbrace{\texttt{binop}}^{\texttt{op}}, \overbrace{\texttt{expr}}^{\texttt{c}}, \overbrace{\texttt{expr}}^{\texttt{d}}) \longrightarrow \overbrace{(\texttt{expr} \times \texttt{expr})^*}^{\texttt{extpairs}}$$

yields a list of pairs of expressions `extpairs` given the binary operation `op` and pair of expressions `c` and `d`, which are needed to form constraints for cases where applying `op` to `c` and `d` would lead to a dynamic error.

### 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`, E_Literal(L_Int) 0 ) and ( E_Literal(L_Int) 0 , `d`).
- All of the following apply (DIV_DIVRM):

* `op` is either `DIV` or `DIVRM`;
* define `extpairs` as the list consisting of (`d`, $\overbrace{1}^{\text{E\_Literal(L\_Int)}}$) and ($\overbrace{1}^{\text{E\_Literal(L\_Int)}}$, `d`).

**Formally**

PLUS

$$\mathit{possible\_extremities\_right}(\overbrace{\texttt{PLUS}}^{\texttt{op}}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{c}, \texttt{d})]}^{\texttt{extpairs}}$$

MINUS

$$\mathit{possible\_extremities\_right}(\overbrace{\texttt{MINUS}}^{\texttt{op}}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{d}, \texttt{c})]}^{\texttt{extpairs}}$$

MUL

$$\mathit{possible\_extremities\_right}(\overbrace{\texttt{MUL}}^{\texttt{op}}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{c}, \texttt{c}), (\texttt{c}, \texttt{d}), (\texttt{d}, \texttt{c}), (\texttt{d}, \texttt{d})]}^{\texttt{extpairs}}$$

SHL_SHR

$$\frac{\texttt{op} \in \{\texttt{SHL}, \texttt{SHR}\}}{\mathit{possible\_extremities\_right}(\texttt{op}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{d}, \overbrace{0}^{\text{E\_Literal(L\_Int)}}), (\overbrace{0}^{\text{E\_Literal(L\_Int)}}, \texttt{d})]}^{\texttt{extpairs}}}$$

DIV_DIVRM

$$\frac{\texttt{op} \in \{\texttt{DIV}, \texttt{DIVRM}\}}{\mathit{possible\_extremities\_right}(\texttt{op}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{d}, \overbrace{1}^{\text{E\_Literal(L\_Int)}}), (\overbrace{1}^{\text{E\_Literal(L\_Int)}}, \texttt{d})]}^{\texttt{extpairs}}}$$

### TypingRule.ConstraintMod

The function

$$\mathit{constraint\_mod}(\overbrace{\text{int\_constraint}}^{\texttt{c}}) \longrightarrow \overbrace{\text{int\_constraint}}^{\texttt{new\_c}}$$

yields a range constraint `new_c` from 0 to the expression in `c` that is maximal. This is needed to apply the modulus operation to a pair of constraints.

### 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

$$\texttt{e\_upper} := \begin{cases} \texttt{e} & \texttt{c} = \mathsf{Constraint\_Exact}(\texttt{e}) \\ \texttt{b} & \texttt{c} = \mathsf{Constraint\_Range}(\_, \texttt{b}) \end{cases}$$

$$\frac{\texttt{e\_minus\_1} := \mathsf{E\_Binop}(\texttt{MINUS}, \texttt{e\_upper}, \overbrace{1}^{\mathsf{E\_Literal(L\_Int)}})}{\mathit{constraint\_mod}(\texttt{c}) \xrightarrow{\text{type}} \mathsf{Constraint\_Range}(\overbrace{0}^{\mathsf{E\_Literal(L\_Int)}}, \overbrace{\texttt{e\_minus\_1}}^{\texttt{new\_c}})}$$

### TypingRule.ConstraintPow

The function

$$\mathit{constraint\_pow}(\overbrace{\mathsf{int\_constraint}}^{\texttt{c1}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c2}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^{+}}^{\texttt{new\_cs}}$$

yields a list of range constraints `new_cs` that are needed to calculate the result of applying a POW operation to the constraints `c1` and `c2`.

### 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

$$\mathit{constraint\_pow}(\overbrace{\mathsf{Constraint\_Exact}(\mathtt{a})}^{\mathtt{c1}}, \overbrace{\mathsf{Constraint\_Exact}(\mathtt{c})}^{\mathtt{c2}}) \xrightarrow{\text{type}} \overbrace{[\ \mathsf{Constraint\_Exact}(\mathsf{E\_Binop}(\mathtt{POW}, \mathtt{a}, \mathtt{c}))\ ]}^{\mathtt{new\_cs}}$$

RANGE_EXACT

$$\frac{\mathtt{mac} := \mathsf{E\_Binop}(\mathtt{POW}, \mathsf{E\_Unop}(\mathtt{NEG}, \mathtt{a}), \mathtt{c})}{\mathit{constraint\_pow}(\overbrace{\mathsf{Constraint\_Range}(\mathtt{a}, \mathtt{b})}^{\mathtt{c1}}, \overbrace{\mathsf{Constraint\_Exact}(\mathtt{c})}^{\mathtt{c2}}) \xrightarrow{\text{type}} \overbrace{[\ \overbrace{\mathsf{E\_Literal(L\_Int)}}^{\mathsf{Constraint\_Range}}\ 0\ ..\ \overbrace{\mathtt{b}\ \mathtt{POW}\ \mathtt{c}}^{\mathsf{E\_Binop}},\ \overbrace{\mathsf{E\_Unop}(\mathtt{NEG}, \mathtt{mac})..\mathtt{mac}}^{\mathsf{Constraint\_Range}}\ ]}^{\mathtt{new\_cs}}}$$

EXACT_RANGE

$$\texttt{mad} := \mathsf{E\_Binop}(\texttt{POW}, \mathsf{E\_Unop}(\texttt{NEG}, \texttt{a}), \texttt{d})$$

$$\overbrace{\mathit{constraint\_pow}(\overbrace{\mathsf{Constraint\_Exact}(\texttt{a})}^{\texttt{c1}}, \overbrace{\mathsf{Constraint\_Range}(\_, \texttt{d})}^{\texttt{c2}})}^{\texttt{new\_cs}} \xrightarrow{\text{type}}$$

$$\overbrace{\underbrace{\mathsf{E\_Literal(L\_Int)}}_{\mathsf{Constraint\_Range}}\ \underbrace{[\ 0\ ..\ \texttt{a POW d}}_{\mathsf{E\_Binop}},\ \underbrace{\mathsf{E\_Unop}(\texttt{NEG}, \texttt{mad})..\texttt{mad}}_{\mathsf{Constraint\_Range}},\ \underbrace{1\ ]}_{\mathsf{Constraint\_Exact}}}^{\mathsf{Constraint\_Range}}$$

RANGE_RANGE

$$\texttt{mad} := \mathsf{E\_Binop}(\texttt{POW}, \mathsf{E\_Unop}(\texttt{NEG}, \texttt{a}), \texttt{d})$$

$$\overbrace{\mathit{constraint\_pow}(\overbrace{\mathsf{Constraint\_Range}(\texttt{a}, \texttt{b})}^{\texttt{c1}}, \overbrace{\mathsf{Constraint\_Range}(\_, \texttt{d})}^{\texttt{c2}})}^{\texttt{new\_cs}} \xrightarrow{\text{type}}$$

$$\overbrace{\underbrace{\mathsf{E\_Literal(L\_Int)}}_{\mathsf{Constraint\_Range}}\ \underbrace{[\ 0\ ..\ \texttt{b POW d}}_{\mathsf{E\_Binop}},\ \underbrace{\mathsf{E\_Unop}(\texttt{NEG}, \texttt{mad})..\texttt{mad}}_{\mathsf{Constraint\_Range}},\ \underbrace{1\ ]}_{\mathsf{Constraint\_Exact}}}^{\mathsf{Constraint\_Range}}$$

# Chapter 33

# Symbolic Reduction and Equivalence Testing

In this chapter, we define two forms of symbolic reasoning — *symbolic reduction* and *symbolic equivalence testing*. Symbolic reduction simplifies expressions into *equivalent* expressions that are simpler to reason about. In our context, equivalence means that we can substitute one expression for another without affecting the semantics of the overall specification. Symbolic equivalence is a *conservative* test. By conservative, we mean that if a test for equivalence returns `TRUE` then the expressions being compared are indeed equivalent, but if the test returns `FALSE` then there are two possibilities:

- the expressions are not equivalent;
- the expressions are equivalent, but the reasoning power of our rules is not enough to prove it, and so we conservatively answer negatively.

In proof-theoretic terms, we can say that our equivalence tests are *sound* but *incomplete*.

Notice that for a conservative test, it is always correct to return `FALSE`.

In this chapter, we use the special value $\top$ to represent a failure in transforming an expression into a desired form (the specific desired form varies according to the functions utilizing this value).

We first define symbolic expressions and operations over symbolic expressions in Section 33.1 and then define the rules for symbolic reduction and equivalence testing in Section 33.2.

## 33.1 Symbolic Expressions

Our symbolic reduction and equivalence testing rules use *symbolic expressions*, defined below:

$$\begin{aligned} \mathsf{polynomial} &\triangleq \mathsf{unitary\_monomial} \rightharpoonup \mathbb{Q} \setminus \{0\} \\ \mathsf{unitary\_monomial} &\triangleq \mathbb{I} \rightharpoonup \mathbb{N}^+ \end{aligned}$$

We now explain each component of a symbolic expression and how it can be interpreted as a mathematical formula via the interpretation function $\alpha$. We also define operations over symbolic expressions.

**Definition 47 (Unitary Monomial)** *A* Unitary Monomial *is a partial function from identifiers to positive integers*[1].

*A non-empty unitary monomial,* $m \in \mathit{unitary\_monomial}$ *where* $m \neq \emptyset_\lambda$*, can be interpreted as follows:*

$$\alpha(m) \triangleq \prod_{x \in \mathrm{dom}(m)} x^{m(x)} \ .$$

*An empty unitary monomial is interpreted as the constant* 1*:*

$$\alpha(\emptyset_\lambda) \triangleq 1 \ .$$

For example,

$$\alpha(\ \{\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 2\}\ ) = x^3 \cdot y \cdot z^2 \ .$$

The function

$$\mathit{mul\_monomials}(\overbrace{\mathsf{unitary\_monomial}}^{\mathtt{m1}}, \overbrace{\mathsf{unitary\_monomial}}^{\mathtt{m2}}) \to \overbrace{\mathsf{unitary\_monomial}}^{\mathtt{m}}$$

multiplies two unitary monomials and returns a unitary monomial

$$\frac{\mathtt{f} := \lambda \mathtt{x} \in \mathsf{identifier}.\ \begin{cases} \mathtt{f1}(\mathtt{x}) & \text{if } \mathtt{x} \in \mathrm{dom}(\mathtt{f1}) \setminus \mathrm{dom}(\mathtt{f2}) \\ \mathtt{f2}(\mathtt{x}) & \text{if } \mathtt{x} \in \mathrm{dom}(\mathtt{f2}) \setminus \mathrm{dom}(\mathtt{f1}) \\ \mathtt{f1}(\mathtt{x}) + \mathtt{f2}(\mathtt{x}) & \text{else } \mathtt{x} \in \mathrm{dom}(\mathtt{f1}) \cap \mathrm{dom}(\mathtt{f2}) \end{cases}}{\mathit{mul\_monomials}(\overbrace{\mathtt{f1}}^{\mathtt{m1}}, \overbrace{\mathtt{f2}}^{\mathtt{m2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{f}}^{\mathtt{m}}}$$

For example,

$$\mathit{mul\_monomials}(\quad \{\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 2\}, \{\mathtt{x} \mapsto 1, \mathtt{w} \mapsto 2\}\ ) = \\ \{\mathtt{x} \mapsto 4, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 2, \mathtt{w} \mapsto 2\}$$

**Definition 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:*

$$\alpha(p) \triangleq \sum_{m \in \mathrm{dom}(p)} p(m) \cdot \alpha(m)$$

For example,

$$\left(\left\{\begin{array}{lll} \{\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 2\} & \mapsto & -1, \\ \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 1\} & \mapsto & \frac{3}{4} \end{array}\right\}\right) = -1 \cdot x^3 \cdot y \cdot z^2 + \frac{3}{4} \cdot \mathtt{x}^2 \cdot \mathtt{y} \ .$$

[1] A unitary monomial has a unit factor, for example $x^3$, whereas a non-unitary monomial has a non-unit factor, for example, $2x^3$.

The function

$$add\_polynomials : \mathsf{polynomial} \times \mathsf{polynomial} \to \mathsf{polynomial}$$

adds two polynomials:

$$\frac{\mathtt{f} := \lambda \mathtt{m} \in \mathsf{unitary\_monomial}. \begin{cases} \mathtt{f1}(\mathtt{m}) & \text{if } \mathtt{m} \in \mathrm{dom}(\mathtt{f1}) \setminus \mathrm{dom}(\mathtt{f2}) \\ \mathtt{f2}(\mathtt{m}) & \text{if } \mathtt{m} \in \mathrm{dom}(\mathtt{f2}) \setminus \mathrm{dom}(\mathtt{f1}) \\ \bot & \text{if } \mathtt{m} \in \mathrm{dom}(\mathtt{f1}) \cap \mathrm{dom}(\mathtt{f2}) \text{ and } \mathtt{f1}(\mathtt{m}) + \mathtt{f2}(\mathtt{m}) = 0 \\ \mathtt{f1}(\mathtt{m}) + \mathtt{f2}(\mathtt{m}) & \text{else } \mathtt{m} \in \mathrm{dom}(\mathtt{f1}) \cap \mathrm{dom}(\mathtt{f2}) \text{ and } \mathtt{f1}(\mathtt{m}) + \mathtt{f2}(\mathtt{m}) \neq 0 \end{cases}}{add\_polynomials(\overbrace{\mathtt{f1}}^{\mathtt{p1}}, \overbrace{\mathtt{f2}}^{\mathtt{p2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{f}}^{\mathtt{p}}}$$

The overloaded function

$$add\_polynomials : \mathsf{polynomial}^* \to \mathsf{polynomial}$$

adds a list of polynomials:

$$\frac{}{add\_polynomials([\,]) \xrightarrow{\mathsf{type}} \emptyset_\lambda} \text{EMPTY} \qquad \frac{}{add\_polynomials([\mathtt{p}]) \xrightarrow{\mathsf{type}} \mathtt{p}} \text{ONE}$$

$$\frac{add\_polynomials(\mathtt{p}_{2..k}) \xrightarrow{\mathsf{type}} \mathtt{p'} \qquad add\_polynomials(\mathtt{p}_1, \mathtt{p'}) \xrightarrow{\mathsf{type}} \mathtt{p}}{add\_polynomials(\mathtt{p}_{1..k}) \xrightarrow{\mathsf{type}} \mathtt{p}} \text{TWO\_OR\_MORE}$$

The function

$$mul\_polynomials : \overbrace{\mathsf{polynomial}}^{\mathtt{p1}} \times \overbrace{\mathsf{polynomial}}^{\mathtt{p2}} \to \overbrace{\mathsf{polynomial}}^{\mathtt{p}}$$

multiplies two polynomials.

$$\frac{\mathtt{ps} := \left\{ \begin{array}{l} \{mul\_monomials(\mathtt{m1}, \mathtt{m2}) \mapsto \mathtt{f1}(\mathtt{m1}) \times \mathtt{f2}(\mathtt{m2})\} \mid \\ \mathtt{m1} \in \mathrm{dom}(\mathtt{f1}) \ \wedge \ \mathtt{m2} \in \mathrm{dom}(\mathtt{f2}) \end{array} \right\} \qquad \begin{array}{l} \mathtt{ordered\_ps} := list\_set(\mathtt{ps}) \\ add\_polynomials(i = 1..k : \mathtt{ordered\_ps}) \xrightarrow{\mathsf{type}} \mathtt{p} \end{array}}{mul\_polynomials(\overbrace{\mathtt{f1}}^{\mathtt{p1}}, \overbrace{\mathtt{f2}}^{\mathtt{p2}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

## 33.2 Typing Rules

We employ the following rules:

- TypingRule.Normalize
- 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

$$\mathit{normalize}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \overbrace{\mathsf{expr}}^{\mathsf{new\_e}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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

```
func main() => integer
begin
                                // normalized rhs expression
    constant ONE = 1;           // 1
    var o = ONE;                // 1
    - = 5.0;                    // 5.0

    let x = 3;                  // 3
    - = 3 * x;                  // 9

    var y : integer{1, 2, 3};
    var z : integer{4, 5, 6};
    var p1 = (y + 5) - z;        // (y + 5) - z
    var p2 = (z DIV 2) * y;      // (z DIV 2) * y
```

```
    var p3 = z * (y DIV 2);       // z * (y DIV 2)
    var p4 = z * (z * y);         // z * (z * y)
    var p5 = z as integer{4, 5}; // z
    var p6 = z ^ y;               // z ^ y

    return 0;
end;
```

### Prose

One of the following applies:

- All of the following apply (NORMALIZABLE):
  - applying *to_ir* to `e` in `tenv` to obtain a symbolic expression yields a symbolic expression $\texttt{p1}^{/\!/\texttt{\#TE}}$;
  - applying *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

NORMALIZABLE

$$\frac{\texttt{p1} \neq \top \qquad \begin{array}{c} \mathit{to\_ir}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \texttt{p1} \;/\!/\; \texttt{\#TE} \\ \mathit{polynomial\_to\_expr}(\texttt{p1}) \xrightarrow{\mathsf{type}} \texttt{new\_e} \end{array}}{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \texttt{new\_e}}$$

NOT_NORMALIZABLE

$$\frac{\mathit{to\_ir}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \top}{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{e}}^{\texttt{new\_e}}}$$

### TypingRule.ReduceConstraint

The function

$$\mathit{reduce\_constraint}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\mathsf{c}}) \longrightarrow \overbrace{\mathsf{int\_constraint}}^{\texttt{new\_c}}$$

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 an range integer constraint for `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *normalize* to `e1` in `tenv` yields `e1'`;
  - applying *normalize* to `e2` in `tenv` yields `e2'`;
  - define `new_c` as the exact integer constraint for `e'`, that is, Constraint_Range(`e1'`, `e2'`).

### Formally

EXACT

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \mathsf{e'}}{\mathit{reduce\_constraint}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e'})}^{\mathtt{new\_c}}}$$

RANGE

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} \mathsf{e1'} \qquad \mathit{normalize}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} \mathsf{e2'}}{\mathit{reduce\_constraint}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1'}, \mathsf{e2'})}^{\mathtt{new\_c}}}$$

### TypingRule.ReduceConstraints

The function

$$\mathit{reduce\_constraints}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\texttt{new\_cs}}$$

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 $\texttt{c}_\texttt{i}$;
- define `new_cs` as the list containing $\texttt{c}_\texttt{i}$ for every `i` in *indices*(`cs`).

### Formally

$$\frac{\texttt{i} \in \mathit{indices}(\texttt{cs}) : \mathit{reduce\_constraint}(\texttt{tenv}, \texttt{cs}[\texttt{i}]) \xrightarrow{\text{type}} \texttt{c}_\texttt{i} \qquad \texttt{new\_cs} := [\texttt{i} \in \mathit{indices}(\texttt{cs}) : \texttt{c}_\texttt{i}]}{\mathit{reduce\_constraints}(\texttt{tenv}, \texttt{cs}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

### TypingRule.ToIR

The function

$$\mathit{to\_ir}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{expr}}^{\text{e}}) \longrightarrow \overbrace{\mathsf{polynomial}}^{\text{p}} \cup \{\top\} \cup \overbrace{\mathsf{TTypeError}}^{\text{\#TE}}$$

transforms a subset of ASL expressions into symbolic expressions. If an ASL expression cannot be represented by a symbolic expression (because, for example, it contains operations that are not available in symbolic expressions), the special value $\top$ is returned.

### 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
                                // to_ir of rhs expression
    constant ONE = 1;           // 1
    var o = ONE;                // 1
```

```
    - = 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 $\top$.
- 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 $\{\emptyset_\lambda \mapsto \texttt{i}\}$ if `v` is an integer literal for `i` and $\top$, 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 $\bot$;
  - looking up `s` in tenv for an associated immutable expression via *lookup_immutable_expr* yields the expression `e'`;
  - applying *to_ir* to `e'` in tenv yields `p`.
- All of the following apply (EVAR_EXACT_CONSTRAINT):

  - `e` is a variable expression with identifier `s`, that is, E_Var(`s`);
  - looking up the constant associated with `s` in `tenv` via *lookup_constant* yields $\bot$;
  - looking up `s` in `tenv` for an associated immutable expression via *lookup_immutable_expr* yields $\bot$;
  - determining the type of `s` yields $\texttt{t}^{/\!/\texttt{\#TE}}$;
  - the underlying type of `t` is $\texttt{ty1}^{/\!/\texttt{\#TE}}$;
  - `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 $\texttt{p}^{/\!/\top}$.

- 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 $\bot$;
  - determining the type of `s` yields $\texttt{t}^{/\!/\texttt{\#TE}}$;
  - the underlying type of `t` is $\texttt{ty1}^{/\!/\texttt{\#TE}}$;
  - `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, $\{\{\texttt{s} \mapsto 1\} \mapsto 1\}$.

- 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 $\bot$;
  - looking up the constant associated with `s` in `tenv` yields $\bot$;
  - determining the type of `s` yields $\texttt{t}^{/\!/\texttt{\#TE}}$;
  - the underlying type of `t` is $\texttt{ty1}^{/\!/\texttt{\#TE}}$;
  - `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 $\texttt{ir1}^{/\!/\top,\texttt{\#TE}}$;
  - converting `e2` to a symbolic expression in `tenv` yields $\texttt{ir2}^{/\!/\top,\texttt{\#TE}}$;
  - `p` is the symbolic expression adding up `ir1` and `ir2`.

- All of the following apply (EBINOP_MINUS):
  - `e` is a binary subtraction expression with operands `e1` and `e2`, that is, `E_Binop(MINUS, e1, e2)`;
  - `e'` is the addition expression with operands `e1` and the negation of `e2`, that is, `E_Binop(PLUS, e1, E_Unop(MINUS, e2))`;
  - converting `e'` into a symbolic expression in `tenv` yields $\mathtt{p}^{/\!/\top,\#\mathtt{TE}}$.
- 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 $\mathtt{p}^{/\!/\top,\#\mathtt{TE}}$.
- 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 $\mathtt{p}^{/\!/\top,\#\mathtt{TE}}$.
- 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 $\mathtt{ir1}^{/\!/\top,\#\mathtt{TE}}$;
  - converting `e2` to a symbolic expression in `tenv` yields $\mathtt{ir2}^{/\!/\top,\#\mathtt{TE}}$;
  - `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 $\mathtt{ir1}^{/\!/\top,\#\mathtt{TE}}$;
  - `f2` is $\frac{1}{\mathtt{i2}}$ (testing against $\mathtt{i2} = 0$ is done dynamically);
  - `p` is the polynomial `ir1` with each monomial multiplied by `f2`.

- All of the following apply (EBINOP_DIV_MONOMIAL_DENOMINATOR):
  - `e` is a binary division expression with operands `e1` and `e2`, that is, E_Binop(DIV, `e1`, `e2`);
  - converting `e1` to a symbolic expression in `tenv` yields $\texttt{ir1}^{/\!/\top,\#\text{TE}}$;
  - converting `e2` to a symbolic expression in `tenv` yields $\texttt{ir2}^{/\!/\top,\#\text{TE}}$;
  - `ir2` consists of a single binding between the monomial `mono` and the factor `v_factor`;
  - applying `polynomial_divide_by_term` to `ir1`, `v_factor`, and `mono` yields $\texttt{p}^{/\!/\top}$.
- All of the following apply (EBINOP_DIV_NON_MONOMIAL_DENOMINATOR):
  - `e` is a binary division expression with operands `e1` and `e2`, that is, E_Binop(DIV, `e1`, `e2`);
  - converting `e1` to a symbolic expression in `tenv` yields $\texttt{ir1}^{/\!/\top,\#\text{TE}}$;
  - converting `e2` to a symbolic expression in `tenv` yields $\texttt{ir2}^{/\!/\top,\#\text{TE}}$;
  - `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 $\texttt{ir1}^{/\!/\top,\#\text{TE}}$;
  - `i2` is non-negative;
  - `f2` is $2^{\texttt{i2}}$;

  - `p` is the polynomial `ir1` with each monomial multiplied by `f2`.

- All of the following apply (EBINOP_OTHER_NON_LITERALS):
  - `e` is a binary expression with an operator `op` that is other than `PLUS`, `MINUS`, `MUL`, or `SHL`, applied to the operand expressions `e1` and `e2`;
  - at least one of `e1` and `e2` is not a literal expression;
  - `p` is $\top$.
- All of the following apply (EBINOP_OTHER_LITERALS_NON_INT_RESULT):
  - `e` is a binary expression with an operator `op` that is other than `PLUS`, `MINUS`, `MUL`, `DIV`, or `SHL`, applied to the operand expressions `e1` and `e2`;
  - `e1` is the literal expression for literal `l1`;
  - `e2` is the literal expression for literal `l2`;
  - statically applying `op` to `l1` and `l2` yields the literal `l`, which is not an integer literal;
  - `p` is $\top$.
- All of the following apply (EBINOP_OTHER_LITERALS_INT_RESULT):
  - `e` is a binary expression with an operator `op` that is other than `PLUS`, `MINUS`, `MUL`, or `SHL`, applied to the operand expressions `e1` and `e2`;
  - `e1` is the literal expression for literal `l1`;
  - `e2` is the literal expression for literal `l2`;
  - statically applying `op` to `l1` and `l2` yields the integer literal for $k$;
  - `p` is the symbolic expression for the integer $k$, that is, $\{\emptyset_\lambda \mapsto k\}$.
- All of the following apply (EUNOP_NEG):
  - `e` is a unary expression with the negation operator `NEG` and operand `e1`;
  - converting the binary expression with operator `MUL` and left-hand-side operand for the integer literal $-1$ and right-hand-side operand `e1` in `tenv` yields $\mathtt{p}^{/\!/\top,\#\mathtt{TE}}$.
- All of the following apply (EUNOP_OTHER):
  - `e` is a unary expression with an operator other than `NEG`;
  - `p` is $\top$.
- All of the following apply (ATC):
  - `e` is an asserting type conversion for the subexpression `e'`, that is, E_ATC(`e'`, _);
  - applying *to_ir* to `e'` yields $\mathtt{p}^{/\!/\top,\#\mathtt{TE}}$.

- All of the following apply (OTHER):
  - `e` is an expression with a label other than E_Literal, E_Var, E_Binop, E_Unop, and E_ATC;
  - `p` is $\top$.

**Formally**

LITERAL_INT

$$\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(\mathtt{i}))}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\{\emptyset_\lambda \mapsto \mathtt{i}\}}^{\mathtt{p}}$$

LITERAL_OTHER

$$\frac{\mathit{ast\_label}(\mathtt{v}) \neq \mathsf{L\_Int}}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Literal}(\mathtt{v})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \top}$$

EVAR_CONSTANT

$$\frac{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{E\_Literal}(\mathtt{v}) \qquad \mathtt{p} := \mathsf{choice}(\mathtt{v} = \mathsf{L\_Int}(\mathtt{i}), \{\emptyset_\lambda \mapsto \mathtt{i}\}, \top)}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EVAR_IMMUTABLE_EXPR

$$\frac{\begin{array}{c}\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \\ \mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{e'} \qquad \mathit{to\_ir}(\mathtt{e'}) \xrightarrow{\mathsf{type}} \mathtt{p}\end{array}}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EVAR_EXACT_CONSTRAINT

$$\frac{\begin{array}{c}\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \\ \mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \qquad \mathit{type\_of}(\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{t} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{ty1} \;/\!/\; \mathtt{\#TE} \\ \mathit{ast\_label}(\mathtt{ty1}) = \mathsf{T\_Int} \\ \mathtt{ty1} = \mathsf{T\_Int}(\mathsf{WellConstrained}([\mathsf{Constraint\_Exact}(\mathtt{e})])) \qquad \mathit{to\_ir}(\mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{p} \;/\!/\; \top\end{array}}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EVAR_INT

$$\frac{\begin{array}{c}\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \qquad \mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \\ \mathit{type\_of}(\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{t} \qquad \mathit{make\_anonymous}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{ty1} \\ \mathit{ast\_label}(\mathtt{ty1}) = \mathsf{T\_Int} \qquad \mathtt{ty1} \neq \mathsf{T\_Int}(\mathsf{WellConstrained}([\mathsf{Constraint\_Exact}(\_)]))\end{array}}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\{\{\mathtt{s} \mapsto 1\} \mapsto 1\}}^{\mathtt{p}}}$$

EVAR_NON_INT

$$\frac{\mathit{lookup\_constant}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \quad \mathit{lookup\_immutable\_expr}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \quad \mathit{type\_of}(\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{t} \quad \mathit{make\_anonymous}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{ty1} \quad \mathit{ast\_label}(\mathtt{ty1}) \neq \mathsf{T\_Int}}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \top}$$

EBINOP_PLUS

$$\frac{\mathit{to\_ir}(\mathtt{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{ir1} \;/\!/\; \mathtt{\#TE}, \top \quad \mathit{to\_ir}(\mathtt{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{ir2} \;/\!/\; \mathtt{\#TE}, \top \quad \mathtt{p} := \mathit{add\_polynomials}(\mathtt{ir1}, \mathtt{ir2})}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{PLUS}, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_MINUS

$$\frac{\mathtt{e'} := \mathsf{E\_Binop}(\mathtt{PLUS}, \mathtt{e1}, \mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{e2})) \quad \mathit{to\_ir}(\mathtt{tenv}, \mathtt{e'}) \xrightarrow{\mathsf{type}} \mathtt{p} \;/\!/\; \mathtt{\#TE}, \top}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{MINUS}, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_MUL_DIV_LEFT

$$\frac{\mathit{to\_ir}(\mathtt{tenv}, \mathsf{E\_Binop}(\mathtt{DIV}, \mathsf{E\_Binop}(\mathtt{MUL}, \mathtt{e1}, \mathtt{e3}), \mathtt{e2})) \xrightarrow{\mathsf{type}} \mathtt{p} \;/\!/\; \mathtt{\#TE}, \top}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{MUL}, \mathsf{E\_Binop}(\mathtt{DIV}, \mathtt{e1}, \mathtt{e2}), \mathtt{e3})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_MUL_DIV_RIGHT

$$\frac{\mathtt{e1} \neq \mathsf{E\_Binop}(\mathtt{DIV}, \_, \_) \quad \mathit{to\_ir}(\mathtt{tenv}, \mathsf{E\_Binop}(\mathtt{DIV}, \mathsf{E\_Binop}(\mathtt{MUL}, \mathtt{e1}, \mathtt{e2}), \mathtt{e3})) \xrightarrow{\mathsf{type}} \mathtt{p} \;/\!/\; \mathtt{\#TE}, \top}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{MUL}, \mathtt{e1}, \mathsf{E\_Binop}(\mathtt{DIV}, \mathtt{e2}, \mathtt{e3}))}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_MUL

$$\frac{\mathtt{e1} \neq \mathsf{E\_Binop}(\mathtt{DIV}, \_, \_) \quad \mathtt{e2} \neq \mathsf{E\_Binop}(\mathtt{DIV}, \_, \_) \quad \mathit{to\_ir}(\mathtt{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{ir1} \;/\!/\; \mathtt{\#TE}, \top \quad \mathit{to\_ir}(\mathtt{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{ir2} \;/\!/\; \mathtt{\#TE}, \top \quad \mathtt{p} := \mathit{mul\_polynomials}(\mathtt{ir1}, \mathtt{ir2})}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{MUL}, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_DIV_INT_DENOMINATOR

$$\frac{\begin{array}{c} to\_ir(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathtt{ir1} \;/\!/\; \mathtt{\#TE}, \top \\ \mathtt{f2} := \frac{1}{\mathtt{i2}} \qquad \mathtt{ir1} \overset{\mathsf{is}}{=} [i = 1..k : \mathtt{m_i} \mapsto \mathtt{c_i}] \qquad \mathtt{p} := [i = 1..k : \mathtt{m_i} \mapsto \mathtt{c_i} \times \mathtt{f2}] \end{array}}{to\_ir(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{DIV}, \mathsf{e1}, \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(\mathtt{i2}))}^{\mathsf{e2}})}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_DIV_MONOMIAL_DENOMINATOR

$$\frac{\begin{array}{c} to\_ir(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathtt{ir1} \;/\!/\; \mathtt{\#TE}, \top \\ to\_ir(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{ir2} \;/\!/\; \mathtt{\#TE}, \top \\ \mathtt{ir2} = [\mathtt{mono} \mapsto \mathtt{v\_factor}] \\ \mathtt{polynomial\_divide\_by\_term}(\mathtt{ir1}, \mathtt{v\_factor}, \mathtt{mono}) \xrightarrow{\mathsf{type}} \mathtt{p} \;/\!/\; \top \end{array}}{to\_ir(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{DIV}, \mathsf{e1}, \mathsf{e2})}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_DIV_NON_MONOMIAL_DENOMINATOR

$$\frac{\begin{array}{c} to\_ir(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathtt{ir1} \;/\!/\; \mathtt{\#TE}, \top \\ to\_ir(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{ir2} \;/\!/\; \mathtt{\#TE}, \top \\ \mathtt{ir2} \neq [\mathtt{mono} \mapsto \mathtt{v\_factor}] \end{array}}{to\_ir(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{DIV}, \mathsf{e1}, \mathsf{e2})}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \top}$$

EBINOP_SHL_NON_LINT_EXPONENT

$$\frac{\mathsf{e2} \neq \mathsf{E\_Literal}(\mathsf{L\_Int}(\_))}{to\_ir(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{SHL}, \_, \mathsf{e2})}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \top}$$

EBINOP_SHL_NEG_SHIFT

$$\frac{\mathtt{i2} < 0}{to\_ir(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{SHL}, \mathsf{e1}, \mathsf{E\_Literal}(\mathsf{L\_Int}(\mathtt{i2})))}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \top}$$

EBINOP_SHL_OKAY

$$\frac{\begin{array}{c} to\_ir(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathtt{ir1} \;/\!/\; \mathtt{\#TE}, \top \\ \mathtt{i2} \geq 0 \\ \mathtt{f2} := 2^{\mathtt{i2}} \qquad \mathtt{ir1} \overset{\mathsf{is}}{=} [i = 1..k : \mathtt{m_i} \mapsto \mathtt{c_i}] \qquad \mathtt{p} := [i = 1..k : \mathtt{m_i} \mapsto \mathtt{c_i} \times \mathtt{f2}] \end{array}}{to\_ir(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{SHL}, \mathsf{e1}, \mathsf{E\_Literal}(\mathsf{L\_Int}(\mathtt{i2})))}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_OTHER_NON_LITERALS

$$\frac{\mathtt{op} \notin \{\mathtt{PLUS}, \mathtt{MINUS}, \mathtt{MUL}, \mathtt{DIV}, \mathtt{SHL}\} \qquad (\mathtt{e1} \neq \mathsf{E\_Literal}(\_) \vee \mathtt{e2} \neq \mathsf{E\_Literal}(\_))}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{op}, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \top}$$

EBINOP_OTHER_LITERALS_NON_INT_RESULT

$$\frac{\mathtt{op} \notin \{\mathtt{PLUS}, \mathtt{MINUS}, \mathtt{MUL}, \mathtt{SHL}\} \qquad \mathit{binop\_literals}(\mathtt{op}, \mathtt{l1}, \mathtt{l2}) \xrightarrow{\mathsf{type}} \mathtt{l} \qquad \mathtt{l} \neq \mathsf{L\_Int}(\_)}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{op}, \mathsf{E\_Literal}(\mathtt{l1}), \mathsf{E\_Literal}(\mathtt{l2}))}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \top}$$

EBINOP_OTHER_LITERALS_INT_RESULT

$$\frac{\mathtt{op} \notin \{\mathtt{PLUS}, \mathtt{MINUS}, \mathtt{MUL}, \mathtt{SHL}\} \qquad \mathit{binop\_literals}(\mathtt{op}, \mathtt{l1}, \mathtt{l2}) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(k) \qquad \mathtt{p} := \{\emptyset_\lambda \mapsto k\}}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{op}, \mathsf{E\_Literal}(\mathtt{l1}), \mathsf{E\_Literal}(\mathtt{l2}))}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EUNOP_NEG

$$\frac{\mathit{to\_ir}(\mathsf{tenv}, \mathsf{E\_Binop}(\mathtt{MUL}, \mathsf{E\_Literal}(\mathsf{L\_Int}(-1)), \mathtt{e1})) \xrightarrow{\mathsf{type}} \mathtt{p} \;/\!/\; \mathtt{\#TE}, \top}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Unop}(\mathtt{NEG}, \mathtt{e1})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EUNOP_OTHER

$$\frac{\mathtt{op} \neq \mathtt{NEG}}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Unop}(\mathtt{op}, \_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \top}$$

ATC

$$\frac{\mathit{to\_ir}(\mathsf{tenv}, \mathtt{e'}) \xrightarrow{\mathsf{type}} \mathtt{p} \;/\!/\; \mathtt{\#TE}, \top}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_ATC}(\mathtt{e'}, \_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

OTHER

$$\frac{\mathit{ast\_label}(\mathtt{e}) \notin \{\mathsf{E\_Literal}, \mathsf{E\_Var}, \mathsf{E\_Binop}, \mathsf{E\_Unop}, \mathsf{E\_ATC}\}}{\mathit{to\_ir}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \top}$$

### TypingRule.ExprEqual

The function

$$\mathit{expr\_equal}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathtt{e1}}, \overbrace{\mathsf{expr}}^{\mathtt{e2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

conservatively checks whether the expression `e1` is equivalent to the expression `e2` in environment `tenv`. The result is given in `b` or a type error, if one is detected.

### 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

```
func f(x: integer) => integer
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`$^{\#\text{TE}}$;
  - `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`$^{\#\text{TE}}$.

**Formally**

$$\frac{\textit{expr\_equal\_norm}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}}{\textit{expr\_equal}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}\ \textsc{norm\_true}$$

$$\frac{\textit{expr\_equal\_norm}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \quad \textit{expr\_equal\_case}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathsf{b} \;/\!/\; \mathtt{\#TE}}{\textit{expr\_equal}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathsf{b}}\ \textsc{norm\_false}$$

**TypingRule.ExprEqualNorm**

The helper function

$$\textit{expr\_equal\_norm}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e1}}, \overbrace{\mathsf{expr}}^{\mathsf{e2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathsf{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

conservatively tests whether the expression `e1` is equivalent to the expression `e2` in environment `tenv` by attempting to transform both expressions to their symbolic expression form and, if successful, comparing the resulting normal forms for equality. The result is given in `b` or a type error, if one is detected.

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 $\mathtt{ir1}^{/\!/\mathtt{\#TE}}$;
  - transforming `e2` into a symbolic expression in `tenv` yields $\mathtt{ir2}^{/\!/\mathtt{\#TE}}$;
  - `b` is the result of equating `ir1` and `ir2`.
- All of the following apply (UNSUPPORTED1):
  - transforming `e1` into a symbolic expression in `tenv` yields $\top$;
  - `b` is `FALSE`;
- All of the following apply (UNSUPPORTED2):
  - transforming `e1` into a symbolic expression in `tenv` yields `ir1`;
  - transforming `e2` into a symbolic expression in `tenv` yields $\top$;
  - `b` is `FALSE`;

**Formally**

$$\textsc{all\_supported}\ \frac{\mathit{to\_ir}(\texttt{e1}) \xrightarrow{\text{type}} \texttt{ir1} \;/\!/\; \texttt{\#TE} \qquad \mathit{to\_ir}(\texttt{e2}) \xrightarrow{\text{type}} \texttt{ir2} \;/\!/\; \texttt{\#TE}}{\mathit{expr\_equal\_norm}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \overbrace{\texttt{ir1} = \texttt{ir2}}^{\texttt{b}}}$$

$$\textsc{unsupported1}\ \frac{\mathit{to\_ir}(\texttt{e1}) \xrightarrow{\text{type}} \top}{\mathit{expr\_equal\_norm}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \overbrace{\texttt{FALSE}}^{\texttt{b}}}$$

$$\textsc{unsupported2}\ \frac{\mathit{to\_ir}(\texttt{e1}) \xrightarrow{\text{type}} \texttt{ir1} \qquad \mathit{to\_ir}(\texttt{e2}) \xrightarrow{\text{type}} \top}{\mathit{expr\_equal\_norm}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \overbrace{\texttt{FALSE}}^{\texttt{b}}}$$

### TypingRule.ExprEqualCase

The helper function

$$\mathit{expr\_equal\_case}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{expr}}^{\texttt{e1}}, \overbrace{\texttt{expr}}^{\texttt{e2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

specializes the equivalence test for expressions `e1` and `e2` in `tenv` for the different types of expressions. The result is given in `b` or a type error, if one is detected.

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 $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing the equivalence of `e1_2` and `e2_2` in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;

  - `b` is `TRUE` if and only if `op1` is equal to `op2` and both `b1` and `b2` are `TRUE`.

- All of the following apply (E_CALL):
  - `e1` is a call expression with subprogram name `name1` and list of arguments `args1`, that is, E_Call(`name1`, `args1`, _);
  - `e2` is a call expression with subprogram name `name2` and list of arguments `args2`, that is, E_Call(`name2`, `args2`, _);
  - checking whether `name1` is equal to `name2` either yields `TRUE` or `FALSE`, which short-circuits the entire rule;
  - checking whether the lists of arguments `args1` and `args2` have the same length yields `TRUE` or `FALSE`, which short-circuits the entire rule;
  - for each index $i$ in the list of indices for `args1`, testing whether `args1`$[i]$ is equivalent to `args2`$[i]$ in `tenv` yields $\texttt{b}_i{}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if $\texttt{b}_i$ is `TRUE` for each index $i$ in the list of indices for `args1`.

- All of the following apply (E_COND):
  - `e1` is a conditional expression with expressions `e1_1`, `e1_2`, and `e1_3`, that is, E_Cond(`e1_1`, `e1_2`, `e1_3`);
  - `e2` is a conditional expression with expressions `e2_1`, `e2_2`, and `e2_3`, that is, E_Cond(`e2_1`, `e2_2`, `e2_3`);
  - testing whether `e1_1` is equivalent to `e2_1` yields $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing whether `e1_2` is equivalent to `e2_2` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - testing whether `e1_3` is equivalent to `e2_3` yields $\texttt{b3}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if all of `b1`, `b2`, and `b3` are `TRUE`.

- All of the following apply (E_SLICE):
  - `e1` is a slicing expression with expression `e1_1` and list of slices `slices1`, that is, E_Slice(`e1_1`, `slices1`);
  - `e1` is a slicing expression with expression `e2_1` and list of slices `slices2`, that is, E_Slice(`e2_1`, `slices2`);
  - testing whether `e1_1` is equivalent to `e2_1` yields $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing whether the lists of slices `slices1` and `slices2` are equivalent in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only both `b1` and `b2` are `TRUE`.

- All of the following apply (E_GETARRAY):
  - `e1` is an array access expression with array expression `e1_1` and position expression `e1_2`, that is, E_GetArray(`e1_1`, `e1_2`);

  - `e2` is an array access expression with array expression `e2_1` and position expression `e2_2`, that is, E_GetArray(`e2_1`, `e2_2`);
  - testing whether `e1_1` is equivalent to `e2_1` yields `b1`$^{\#TE}$;
  - testing whether `e1_2` is equivalent to `e2_2` yields `b2`$^{\#TE}$;
  - `b` is TRUE if and only both `b1` and `b2` are TRUE.

- All of the following apply (E_GETFIELD):
  - `e1` is a field access expression with subexpression `e1_1` and field name `field1`, that is, E_GetField(`e1_1`, `field1`);
  - `e2` is a field access expression with subexpression `e2_1` and field name `field2`, that is, E_GetField(`e2_1`, `field2`);
  - `b1` is TRUE if and only if `field1` is equal to `field2`;
  - testing whether `e1_1` is equivalent to `e2_1` yields `b2`$^{\#TE}$;
  - `b` is TRUE if and only both `b1` and `b2` are TRUE.

- All of the following apply (E_GETFIELDS):
  - `e1` is a fields access expression with subexpression `e1_1` and list of field names `fields1`, that is, E_GetFields(`e1_1`, `fields1`);
  - `e2` is a fields access expression with subexpression `e2_1` and list of field names `fields2`, that is, E_GetFields(`e2_1`, `fields2`);
  - `b1` is TRUE if and only if `fields1` is equal to `fields2`;
  - testing whether `e1_1` is equivalent to `e2_1` yields `b2`$^{\#TE}$;
  - `b` is TRUE if and only both `b1` and `b2` are TRUE.

- All of the following apply (E_GETITEM):
  - `e1` is a tuple access expression with subexpression `e1_1` and position `i1`, that is, E_GetItem(`e1_1`, `i1`);
  - `e2` is a tuple access expression with subexpression `e2_1` and position `i2`, that is, E_GetItem(`e2_1`, `i2`);
  - `b1` is TRUE if and only if `i1` is equal to `i2`;
  - testing whether `e1_1` is equivalent to `e2_1` yields `b2`$^{\#TE}$;
  - `b` is TRUE if and only both `b1` and `b2` are TRUE.

- All of the following apply (E_LITERAL):
  - `e1` is the literal expression with literal `v1`;
  - `e2` is the literal expression with literal `v2`;
  - `b` is TRUE if and only if `v1` is 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 $\texttt{b}_i$ $/\!/^{\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if $\texttt{b}_i$ is `TRUE` for each index $i$ in the list of indices for `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 $\texttt{b1}$ $/\!/^{\texttt{\#TE}}$;
  - applying *expr_equal* to `v1` and `v2` in `tenv` yields $\texttt{b1}$ $/\!/^{\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

- All of the following apply (E_UNOP):
  - `e1` is a unary operator expression with operator `op1` and operand expressions `e1_1`, that is, E_Unop(`op1`, `e1_1`);
  - `e2` is a unary operator expression with operator `op2` and operand expressions `e2_1`, that is, E_Unop(`op2`, `e2_1`);
  - testing whether `e1_1` is equivalent to `e2_1` in `tenv` yields `b1`;
  - `b` is `TRUE` if and only if `op1` is equal to `op2` and `b1` is `TRUE`.

- All of the following apply (E_ARBITRARY):
  - both `e1` and `e2` are `ARBITRARY` expressions;
  - `b` is `FALSE`.

- All of the following apply (E_ATC):

  - `e1` is a type assertion with subexpression with operator `e1_1` and type `t1`, that is, E_ATC(`e1_1`, `t1`);
  - `e2` is a type assertion with subexpression with operator `e2_1` and type `t2`, that is, E_ATC(`e2_1`, `t2`);
  - testing whether `e1_1` is equivalent to `e2_1` in `tenv` yields `b1`;
  - testing whether `t1` is equivalent to `t2` in `tenv` yields `b2`;
  - `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

- All of the following apply (E_VAR):
  - `e1` is a variable expression with identifier `name1`, that is, E_Var(`name1`);
  - `e2` is a variable expression with identifier `name2`, that is, E_Var(`name2`);
  - `b` is `TRUE` if and only if both `name1` is equal to `name2`.

**Formally**

DIFFERENT_LABELS

$$\frac{\mathit{ast\_label}(\texttt{e1}) \neq \mathit{ast\_label}(\texttt{e2})}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

E_BINOP

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_Binop}(\texttt{op1}, \texttt{e1\_1}, \texttt{e1\_2}) \\ \texttt{e2} = \texttt{E\_Binop}(\texttt{op2}, \texttt{e2\_1}, \texttt{e2\_2}) \qquad \mathit{expr\_equal}(\texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{e1\_2}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := (\texttt{op1} = \texttt{op2}) \wedge \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

(Recall that a conjunction over an empty set equals `TRUE`.)

E_CALL

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_Call}(\texttt{name1}, \texttt{args1}, \_) \qquad \texttt{e2} = \texttt{E\_Call}(\texttt{name2}, \texttt{args2}, \_) \\ \mathit{bool\_transition}(\texttt{name1} = \texttt{name2}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{FALSE} \\ \mathit{equal\_length}(\texttt{args1}, \texttt{args2}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{FALSE} \\ i \in \mathit{indices}(\texttt{args1}) : \mathit{expr\_equal}(\texttt{tenv}, \texttt{args1}[i], \texttt{args2}[i]) \xrightarrow{\text{type}} \texttt{b}_i \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{args1})} \texttt{b}_i\end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_COND

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_Cond}(\texttt{e1\_1}, \texttt{e1\_2}, \texttt{e1\_3}) \qquad \texttt{e2} = \texttt{E\_Cond}(\texttt{e2\_1}, \texttt{e2\_2}, \texttt{e2\_3}) \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_2}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_3}, \texttt{e2\_3}) \xrightarrow{\text{type}} \texttt{b3} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3}\end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

E_SLICE

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_Slice}(\texttt{e1\_1}, \texttt{slices1}) \qquad \texttt{e2} = \texttt{E\_Slice}(\texttt{e2\_1}, \texttt{slices2}) \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b2} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_GETARRAY

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_GetArray}(\texttt{e1\_1}, \texttt{e1\_2}) \qquad \texttt{e2} = \texttt{E\_GetArray}(\texttt{e2\_1}, \texttt{e2\_2}) \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_2}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_GETFIELD

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_GetField}(\texttt{e1\_1}, \texttt{field1}) \qquad \texttt{e2} = \texttt{E\_GetField}(\texttt{e2\_1}, \texttt{field2}) \\ \texttt{b1} := \texttt{field1} = \texttt{field2} \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b2} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_GETFIELDS

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_GetFields}(\texttt{e1\_1}, \texttt{fields1}) \qquad \texttt{e2} = \texttt{E\_GetFields}(\texttt{e2\_1}, \texttt{fields2}) \\ \texttt{b1} := \texttt{fields1} = \texttt{fields2} \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b2} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_GETITEM

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_GetItem}(\texttt{e1\_1}, \texttt{i1}) \qquad \texttt{e2} = \texttt{E\_GetItem}(\texttt{e2\_1}, \texttt{i2}) \\ \texttt{b1} := \texttt{i1} = \texttt{i2} \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b2} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_LITERAL

$$\frac{\mathtt{e1} = \mathsf{E\_Literal}(\mathtt{v1}) \qquad \mathtt{e2} = \mathsf{E\_Literal}(\mathtt{v2}) \qquad \mathtt{b} := \mathtt{v1} = \mathtt{v2}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

E_PATTERN

$$\frac{\mathit{ast\_label}(\mathtt{e1}) = \mathsf{E\_Pattern} \wedge \mathit{ast\_label}(\mathtt{e2}) = \mathsf{E\_Pattern}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}$$

E_RECORD

$$\frac{\mathit{ast\_label}(\mathtt{e1}) = \mathsf{E\_Record} \wedge \mathit{ast\_label}(\mathtt{e2}) = \mathsf{E\_Record}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}$$

E_TUPLE

$$\frac{\begin{array}{c}\mathtt{e1} = \mathsf{E\_Tuple}(\mathtt{l1}) \qquad \mathtt{e2} = \mathsf{E\_Tuple}(\mathtt{l2}) \qquad \mathit{equal\_length}(\mathtt{l1}, \mathtt{l2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \mathbin{/\!/} \mathtt{FALSE} \\ i \in \mathit{indices}(\mathtt{l1}) : \mathit{expr\_equal}(\mathtt{tenv}, \mathtt{l1}[i], \mathtt{l2}[i]) \xrightarrow{\mathsf{type}} \mathtt{b}_i \mathbin{/\!/} \mathtt{\#TE} \\ \mathtt{b} := \bigwedge_{i \in \mathit{indices}(\mathtt{l1})} \mathtt{b}_i \end{array}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

E_ARRAY

$$\frac{\begin{array}{c}\mathtt{e1} = \mathsf{E\_Array}\{\text{length} : \mathtt{l1}, \text{value} : \mathtt{v1}\} \\ \mathtt{e2} = \mathsf{E\_Array}\{\text{length} : \mathtt{l2}, \text{value} : \mathtt{v2}\} \qquad \mathit{expr\_equal}(\mathtt{tenv}, \mathtt{l1}, \mathtt{l2}) \xrightarrow{\mathsf{type}} \mathtt{b1} \mathbin{/\!/} \mathtt{\#TE} \\ \mathit{expr\_equal}(\mathtt{tenv}, \mathtt{v1}, \mathtt{v2}) \xrightarrow{\mathsf{type}} \mathtt{b1} \mathbin{/\!/} \mathtt{\#TE} \\ \mathtt{b} := \mathtt{b1} \wedge \mathtt{b2} \end{array}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

E_UNOP

$$\frac{\mathtt{e1} = \mathsf{E\_Unop}(\mathtt{op1}, \mathtt{e1\_1}) \qquad \mathtt{e2} = \mathsf{E\_Unop}(\mathtt{op2}, \mathtt{e2\_1}) \qquad \mathit{expr\_equal}(\mathtt{tenv}, \mathtt{e1\_1}, \mathtt{e2\_1}) \xrightarrow{\mathsf{type}} \mathtt{b1} \mathbin{/\!/} \mathtt{\#TE} \qquad \mathtt{b} := (\mathtt{op1} = \mathtt{op2}) \wedge \mathtt{b1}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

E_ARBITRARY

$$\frac{(\mathit{ast\_label}(\mathtt{e1}) = \mathsf{E\_Arbitrary} \wedge \mathit{ast\_label}(\mathtt{e2}) = \mathsf{E\_Arbitrary})}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}$$

E_ATC

$$\frac{\begin{array}{c}\mathtt{e1} = \mathsf{E\_ATC}(\mathtt{e1\_1}, \mathtt{t1}) \\ \mathtt{e2} = \mathsf{E\_ATC}(\mathtt{e2\_1}, \mathtt{t2}) \qquad \mathit{expr\_equal}(\mathtt{tenv}, \mathtt{e1\_1}, \mathtt{e2\_1}) \xrightarrow{\mathsf{type}} \mathtt{b1} \;/\!/\; \mathtt{\#TE} \\ \mathit{type\_equal}(\mathtt{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{b2} \;/\!/\; \mathtt{\#TE} \\ \mathtt{b} := \mathtt{b1} \wedge \mathtt{b2}\end{array}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

E_VAR

$$\frac{\mathtt{e1} = \mathsf{E\_Var}(\mathtt{name1}) \qquad \mathtt{e2} = \mathsf{E\_Var}(\mathtt{name2}) \\ \mathtt{b} := \mathtt{name1} = \mathtt{name2}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

### TypingRule.TypeEqual

The function

$$\mathit{type\_equal}(\overbrace{\mathsf{ty}}^{\mathtt{t1}}, \overbrace{\mathsf{ty}}^{\mathtt{t2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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 $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

- All of the following apply (TARRAY):
  - `t1` is an array type with index `l1` and element type `t1`, that is, `T_Array(l1,t1)`;
  - `t2` is an array type with index `l2` and element type `t2`, that is, `T_Array(l2,t2)`;
  - testing whether `l1` is equivalent to `l2` in `tenv` yields $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing whether `t1` is equivalent to `t2` in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

- All of the following apply (TNAMED):
  - `t1` is a named type with identifier `s1`, that is `T_Named(s1)`;
  - `t2` is a named type with identifier `s2`, that is `T_Named(s2)`;
  - `b` is `TRUE` if and only if `s1` is equal to `s2`.

- All of the following apply (TENUM):
  - `t1` is an enumeration type with identifier `l1`, that is `T_Enum(l1)`;
  - `t2` is an enumeration type with identifier `l2`, that is `T_Enum(l2)`;
  - `b` is `TRUE` if and only if `l1` is equal to `l2`.

- All of the following apply (TSTRUCTURED):
  - $L$ is either `T_Record` or `T_Exception`;
  - `t1` is a structured type with list of fields `fields1`, that is $L$(`fields1`);
  - `t2` is a structured type with list of fields `fields2`, that is $L$(`fields2`);
  - checking whether the set of field names in `fields1` is equal to the set of field names in `fields2` yields `TRUE` or `FALSE`, which short-circuits the entire rule;
  - for each field `f` in the set of fields of `fields1`, testing whether the type associated with `f` in `fields1` is equivalent to the type associated with `f` in `fields2` in `tenv` yields $\texttt{b}_\texttt{f}{}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if $\texttt{b}_\texttt{f}$ 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 $\texttt{b}_i{}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if $\texttt{b}_i$ is `TRUE` for each index $i$ in the list `ts1`.

**Formally**

DIFFERENT_LABELS

$$\frac{\mathit{ast\_label}(\texttt{t1}) \neq \mathit{ast\_label}(\texttt{t2})}{\mathit{type\_equal}(\texttt{tenv}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

TBOOL_TREAL_TSTRING

$$\frac{\mathit{ast\_label}(\texttt{t1}) = \mathit{ast\_label}(\texttt{t2}) \qquad \mathit{ast\_label}(\texttt{t1}) \in \{\texttt{T\_Bool}, \texttt{T\_Real}, \texttt{T\_String}\}}{\mathit{type\_equal}(\texttt{tenv}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

TINT_UNCONSTRAINED

$$\mathit{type\_equal}(\texttt{tenv}, \texttt{unconstrained\_integer}, \texttt{unconstrained\_integer}) \xrightarrow{\text{type}} \texttt{TRUE}$$

TINT_PARAMETERIZED

$$\frac{\texttt{b} := \texttt{i1} = \texttt{i2}}{\mathit{type\_equal}(\texttt{tenv}, \texttt{T\_Int}(\texttt{Parameterized}(\texttt{i1})), \texttt{T\_Int}(\texttt{Parameterized}(\texttt{i2}))) \xrightarrow{\text{type}} \texttt{b}}$$

TINT_WELLCONSTRAINED

$$\frac{\mathit{constraints\_equal}(\texttt{tenv}, \texttt{c1}, \texttt{c2}) \xrightarrow{\text{type}} \texttt{b} \;\; /\!/ \;\; \texttt{\#TE}}{\mathit{type\_equal}(\texttt{tenv}, \texttt{T\_Int}(\texttt{WellConstrained}(\texttt{c1})), \texttt{T\_Int}(\texttt{WellConstrained}(\texttt{c2}))) \xrightarrow{\text{type}} \texttt{b}}$$

TBITS

$$\frac{\begin{array}{c}\mathit{bitwidth\_equal}(\texttt{tenv}, \texttt{w1}, \texttt{w2}) \xrightarrow{\text{type}} \texttt{b1} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{bitfields\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b2} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{type\_equal}(\texttt{tenv}, \texttt{T\_Bits}(\texttt{w1}, \texttt{bf1}), \texttt{T\_Bits}(\texttt{w2}, \texttt{bf2})) \xrightarrow{\text{type}} \texttt{b}}$$

TARRAY

$$\frac{\begin{array}{c}\mathit{expr\_equal}(\texttt{tenv}, \texttt{l1}, \texttt{l2}) \xrightarrow{\text{type}} \texttt{b1} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{type\_equal}(\texttt{tenv}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{b2} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{type\_equal}(\texttt{tenv}, \texttt{T\_Array}(\texttt{l1}, \texttt{t1}), \texttt{T\_Array}(\texttt{l2}, \texttt{t2})) \xrightarrow{\text{type}} \texttt{b}}$$

TNAMED

$$\frac{\texttt{b} := \texttt{s1} = \texttt{s2}}{\mathit{type\_equal}(\texttt{tenv}, \texttt{T\_Named}(\texttt{s1}), \texttt{T\_Named}(\texttt{s2})) \xrightarrow{\text{type}} \texttt{b}}$$

TENUM

$$\frac{\texttt{b} := \texttt{l1} = \texttt{l2}}{\mathit{type\_equal}(\texttt{tenv}, \texttt{T\_Enum}(\texttt{l1}), \texttt{T\_Enum}(\texttt{l2})) \xrightarrow{\text{type}} \texttt{b}}$$

TSTRUCTURED

$$\frac{\begin{array}{c} L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \\ \mathit{bool\_transition}(\mathit{field\_names}(\texttt{fields1}) = \mathit{field\_names}(\texttt{fields2})) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \\ \quad \texttt{f} \in \mathit{field\_names}(\texttt{fields1}) : \\ \quad \mathit{type\_equal}(\mathsf{tenv}, \mathit{field\_type}(\texttt{fields1}, \texttt{f}), \mathit{field\_type}(\texttt{fields2}, \texttt{f})) \xrightarrow{\mathsf{type}} \mathtt{b_f} \;/\!/\; \mathtt{\#TE} \\ \texttt{b} := \bigwedge_{\texttt{f} \in \mathit{field\_names}(\texttt{fields1})} \mathtt{b_f} \end{array}}{\mathit{type\_equal}(\mathsf{tenv}, L(\texttt{fields1}), L(\texttt{fields2})) \xrightarrow{\mathsf{type}} \texttt{b}}$$

TTUPLE

$$\frac{\begin{array}{c} \mathit{equal\_length}(\texttt{ts1}, \texttt{ts2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \\ i \in \mathit{indices}(\texttt{ts1}) : \mathit{type\_equal}(\mathsf{tenv}, \texttt{ts1}[i], \texttt{ts2}[i]) \xrightarrow{\mathsf{type}} \mathtt{b}_i \;/\!/\; \mathtt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{ts1})} \mathtt{b}_i \end{array}}{\mathit{type\_equal}(\mathsf{tenv}, \mathsf{T\_Tuple}(\texttt{ts1}), \mathsf{T\_Tuple}(\texttt{ts2})) \xrightarrow{\mathsf{type}} \texttt{b}}$$

### TypingRule.BitwidthEqual

The function

$$\mathit{bitwidth\_equal}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathtt{w1}}, \overbrace{\mathsf{expr}}^{\mathtt{w2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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`$^{/\!/\texttt{\#TE}}$.

### Formally

$$\frac{\mathit{expr\_equal}(\texttt{tenv}, \texttt{w1}, \texttt{w2}) \xrightarrow{\text{type}} \texttt{b} \;/\!/\; \texttt{\#TE}}{\mathit{bitwidth\_equal}(\texttt{tenv}, \texttt{w1}, \texttt{w2}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.BitFieldsEqual

The function

$$\mathit{bitfields\_equal}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bf1}}, \overbrace{\text{bitfield}^*}^{\text{bf2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

conservatively tests whether the list of bitfields `bf1` is equivalent to the list of bitfields `bf2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

### 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 $\mathtt{b}_i$ // #TE;
  - `b` is TRUE if and only if $\mathtt{b}_i$ is TRUE for every index of `bf1`.

**Formally**

$$\frac{\textsc{different\_lengths}\quad \mathit{equal\_length}(\mathtt{bf1},\mathtt{bf2}) \xrightarrow{\text{type}} \mathtt{FALSE}}{\mathit{bitfields\_equal}(\mathtt{tenv},\mathtt{bf1},\mathtt{bf2}) \xrightarrow{\text{type}} \mathtt{FALSE}}$$

$$\frac{\begin{array}{c}\textsc{same\_lengths}\quad \mathit{equal\_length}(\mathtt{bf1},\mathtt{bf2}) \xrightarrow{\text{type}} \mathtt{TRUE} \\ i \in \mathit{indices}(\mathtt{bf1}) : \mathit{bitfield\_equal}(\mathtt{tenv},\mathtt{bf1}[i],\mathtt{bf2}[i]) \xrightarrow{\text{type}} \mathtt{b}_i \\ \mathtt{b} := \bigwedge_{i \in \mathit{indices}(\mathtt{bf1})} \mathtt{b}_i\end{array}}{\mathit{bitfields\_equal}(\mathtt{tenv},\mathtt{bf1},\mathtt{bf2}) \xrightarrow{\text{type}} \mathtt{b}}$$

### TypingRule.BitFieldEqual

The function

$$\mathit{bitfield\_equal}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{bitfield}}^{\mathtt{bf1}}, \overbrace{\mathsf{bitfield}}^{\mathtt{bf2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

conservatively tests whether the bitfield `bf1` is equivalent to the bitfield `bf2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

### 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 $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - testing whether the bitfields `bf1_1` and `bf2_1` are equivalent in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `b` is TRUE if and only if both `b1` and `b2` are TRUE.

- All of the following apply (BITFIELD_TYPED):
  - `bf1` is a typed bitfield with name `name1`, list of slices `slices1`, and type `t1`, that is, BitField_Type(`name1`, `slices1`, `t1`);
  - `bf2` is a typed bitfield with name `name2`, list of slices `slices2`, and type `t2`, that is, BitField_Type(`name2`, `slices2`, `t2`);
  - checking whether `name1` is equal to `name2` yields $\texttt{TRUE}^{/\!/\texttt{FALSE}}$;
  - testing whether `slices1` and `slices2` are equivalent in `tenv` yields $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing whether the types `t1` and `t2` are equivalent in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `b` is TRUE if and only if both `b1` and `b2` are TRUE.

**Formally**

$$\frac{ast\_label(\texttt{bf1}) \neq ast\_label(\texttt{bf2})}{bitfield\_equal(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{FALSE}}\ \text{DIFFERENT\_LABELS}$$

$$\frac{\begin{array}{c}\texttt{bf1} = \texttt{BitField\_Simple}(\texttt{name1}, \texttt{slices1}) \\ \texttt{bf2} = \texttt{BitField\_Simple}(\texttt{name2}, \texttt{slices2}) \qquad bool\_transition(\texttt{name1} = \texttt{name2}) \longrightarrow \texttt{b1} \\ slices\_equal(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b2} \ /\!/\ \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{bitfield\_equal(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}\ \text{BITFIELD\_SIMPLE}$$

$$\frac{\begin{array}{c}\texttt{bf1} = \texttt{BitField\_Nested}(\texttt{name1}, \texttt{slices1}, \texttt{bf1\_1}) \\ \texttt{bf2} = \texttt{BitField\_Nested}(\texttt{name2}, \texttt{slices2}, \texttt{bf2\_1}) \\ bool\_transition(\texttt{name1} = \texttt{name2}) \longrightarrow \texttt{TRUE} \ /\!/\ \texttt{FALSE} \\ slices\_equal(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b1} \ /\!/\ \texttt{\#TE}, \\ bitfields\_equal(\texttt{tenv}, \texttt{bf1\_1}, \texttt{bf2\_1}) \xrightarrow{\text{type}} \texttt{b2}\end{array}}{bitfield\_equal(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}\ \text{BITFIELD\_NESTED}$$

$$\frac{\begin{array}{c}\texttt{bf1} = \texttt{BitField\_Type}(\texttt{name1}, \texttt{slices1}, \texttt{t1}) \qquad \texttt{bf2} = \texttt{BitField\_Type}(\texttt{name2}, \texttt{slices2}, \texttt{t2}) \\ bool\_transition(\texttt{name1} = \texttt{name2}) \longrightarrow \texttt{TRUE} \ /\!/\ \texttt{FALSE} \\ slices\_equal(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b1} \ /\!/\ \texttt{\#TE} \\ type\_equal(\texttt{tenv}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{b2} \ /\!/\ \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{bitfield\_equal(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}\ \text{BITFIELD\_TYPED}$$

### TypingRule.ConstraintsEqual

The function

$$constraints\_equal(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{int\_constraint}^*}^{\texttt{cs1}}, \overbrace{\texttt{int\_constraint}^*}^{\texttt{cs2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

conservatively tests whether the constraint list `cs1` is equivalent to the constraint list `cs2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

### 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 $\texttt{TRUE}^{/\!/\texttt{FALSE}}$;
- testing whether the constraint $\texttt{cs1}[i]$ is equivalent to the constraint $\texttt{cs2}[i]$ in `tenv` yields $\texttt{b}_i$ for each index in the index in the indices for `cs1` ($i \in \mathit{indices}(\texttt{cs1})$)$^{/\!/\texttt{\#TE}}$;
- `b` is `TRUE` if and only if all $\texttt{b}_i$ are `TRUE` for each index in the indices for `cs1`.

### Formally

$$\frac{\begin{array}{c}\mathit{equal\_length}(\texttt{cs1},\texttt{cs2}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{FALSE} \\ i \in \mathit{indices}(\texttt{cs1}) : \mathit{constraint\_equal}(\texttt{tenv},\texttt{cs1}[i],\texttt{cs2}[i]) \xrightarrow{\text{type}} \texttt{b}_i \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{cs1})} \texttt{b}_i\end{array}}{\mathit{constraints\_equal}(\texttt{tenv},\texttt{cs1},\texttt{cs2}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.ConstraintEqual

The function

$$\mathit{constraint\_equal}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\text{c1}}, \overbrace{\mathsf{int\_constraint}}^{\text{s2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\mathsf{TTypeError}}^{\text{\#TE}}$$

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**

$$\textsc{different\_labels}\ \frac{\mathit{ast\_label}(\texttt{c1}) \neq \mathit{ast\_label}(\texttt{c2})}{\mathit{constraint\_equal}(\texttt{tenv}, \texttt{c1}, \texttt{c2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

$$\textsc{constraint\_exact}\ \frac{\begin{array}{c}\texttt{c1} = \mathsf{Constraint\_Exact}(\texttt{e1}) \\ \texttt{c2} = \mathsf{Constraint\_Exact}(\texttt{e2}) \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b} \ /\!/\ \texttt{\#TE}\end{array}}{\mathit{constraint\_equal}(\texttt{tenv}, \texttt{c1}, \texttt{c2}) \xrightarrow{\text{type}} \texttt{b}}$$

$$\textsc{constraint\_range}\ \frac{\begin{array}{c}\texttt{bf1} = \mathsf{Constraint\_Range}(\texttt{e1\_1}, \texttt{e1\_2}) \\ \texttt{bf2} = \mathsf{Constraint\_Range}(\texttt{e2\_1}, \texttt{e2\_2}) \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \ /\!/\ \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_2}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \ /\!/\ \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{constraint\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.SlicesEqual

The function

$$\mathit{slices\_equal}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices1}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

conservatively tests whether the list of slices `slices1` is equivalent to the list of slices `slices2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

### 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 $\texttt{slices1}[i]$ is equivalent to $\texttt{slices2}[i]$ in `tenv` for each index in the indices for `slices1` ($i \in \mathit{indices}(\texttt{slices1})$), yields $\texttt{b}_i$ // `#TE`;
  - `b` is `TRUE` if and only if all $\texttt{b}_i$ are `TRUE` for each index in the indices for `slices1`.

**Formally**

DIFFERENT_LENGTHS

$$\frac{\mathit{equal\_length}(\texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{FALSE}}{\mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

SAME_LENGTHS

$$\frac{\begin{array}{c}\mathit{equal\_length}(\texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{TRUE} \\ i \in \mathit{indices}(\texttt{slices1}) : \mathit{slice\_equal}(\texttt{tenv}, \texttt{slices1}[i], \texttt{slices2}[i]) \xrightarrow{\text{type}} \texttt{b}_i \;//\; \texttt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{slices1})} \texttt{b}_i\end{array}}{\mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.SliceEqual

The function

$$\mathit{slice\_equal}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}}^{\texttt{slice1}}, \overbrace{\mathsf{slice}}^{\texttt{slice2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

conservatively tests whether the slice `slice1` is equivalent to the slice `slice2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

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`$^{\#\text{TE}}$;
- 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`$^{\#\text{TE}}$;
  - testing `e1_2` and `e2_2` for equivalence yields `b2`$^{\#\text{TE}}$;
  - 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**

$$\frac{\textit{expr\_equal}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathsf{b} \;/\!/\; \#\mathsf{TE}}{\textit{slices\_equal}(\mathsf{tenv}, \overbrace{\mathsf{Slice\_Single}(\mathsf{e1})}^{\mathsf{slice1}}, \overbrace{\mathsf{Slice\_Single}(\mathsf{e2})}^{\mathsf{slice2}}) \xrightarrow{\mathsf{type}} \mathsf{b}} \;\text{SINGLE\_EXPR}$$

$$\frac{\begin{array}{c}\mathsf{L1}, \mathsf{L2} \in \{\mathsf{Slice\_Length}, \mathsf{Slice\_Range}, \mathsf{Slice\_Star}\} \\ \mathsf{L1} = \mathsf{L2} \qquad \textit{expr\_equal}(\mathsf{tenv}, \mathsf{e1\_1}, \mathsf{e1\_2}) \xrightarrow{\mathsf{type}} \mathsf{b1} \;/\!/\; \#\mathsf{TE} \\ \textit{expr\_equal}(\mathsf{tenv}, \mathsf{e2\_1}, \mathsf{e2\_2}) \xrightarrow{\mathsf{type}} \mathsf{b2} \;/\!/\; \#\mathsf{TE} \\ \mathsf{b} := \mathsf{b1} \wedge \mathsf{b2}\end{array}}{\textit{slices\_equal}(\mathsf{tenv}, \overbrace{\mathsf{L1}(\mathsf{e1\_1}, \mathsf{e2\_1})}^{\mathsf{slice1}}, \overbrace{\mathsf{L2}(\mathsf{e1\_2}, \mathsf{e2\_2})}^{\mathsf{slice2}}) \xrightarrow{\mathsf{type}} \mathsf{b}} \;\text{TWO\_EXPRS}$$

$$\frac{\mathsf{config\_dom}(\mathsf{slice1}) \neq \mathsf{config\_dom}(\mathsf{slice2})}{\textit{slices\_equal}(\mathsf{tenv}, \mathsf{slice1}, \mathsf{slice2}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{FALSE}}^{\mathsf{b}}} \;\text{DIFFERENT\_LABELS}$$

### TypingRule.ArrayLengthEqual

The function

$$\textit{array\_length\_equal}(\overbrace{\mathsf{array\_index}}^{\mathsf{l1}}, \overbrace{\mathsf{array\_index}}^{\mathsf{l2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathsf{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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):
  - `l1` and `l2` have different AST labels;
  - `b` is `FALSE`.
- All of the following apply (EXPR_EXPR):
  - `l1` is an integer type length expression with subexpression `e1_1`, that is, ArrayLength_Expr(`e1_1`);
  - `l2` is an integer type length expression with subexpression `e2_1`, that is, ArrayLength_Expr(`e2_1`);
  - testing whether `e1_1` and `e2_1` are equivalent in `tenv` yields $\mathtt{b}^{/\!/\mathtt{\#TE}}$.
- All of the following apply (ENUM_ENUM):
  - `l1` is an enumeration type index for the identifier `enum1` and a list of labels, that is, ArrayLength_Enum(`enum1`, _);
  - `l2` is an enumeration type index for the identifier `enum2` and a list of labels, that is, ArrayLength_Enum(`enum2`, _);
  - `b` is `TRUE` if and only if `enum1` is equal to `enum2`.

**Formally**

$$\textsc{different\_labels}\ \frac{\mathit{ast\_label}(\mathtt{l1}) \neq \mathit{ast\_label}(\mathtt{l2})}{\mathit{array\_length\_equal}(\mathtt{l1},\mathtt{l2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}$$

$$\textsc{expr\_expr}\ \frac{\mathit{expr\_equal}(\mathtt{e1\_1},\mathtt{e2\_1}) \xrightarrow{\mathsf{type}} \mathtt{b}\ /\!/\ \mathtt{\#TE}}{\mathit{array\_length\_equal}(\mathsf{ArrayLength\_Expr}(\mathtt{e1\_1}),\mathsf{ArrayLength\_Expr}(\mathtt{e2\_1})) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

$$\textsc{enum\_enum}\ \frac{\mathtt{b} := (\mathtt{enum1} = \mathtt{enum2})}{\mathit{array\_length\_equal}(\mathsf{ArrayLength\_Enum}(\mathtt{enum1},\_),\mathsf{ArrayLength\_Enum}(\mathtt{enum2},\_)) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

**TypingRule.PolynomialToExpr**

The function

$$\mathit{polynomial\_to\_expr}(\overbrace{\mathsf{polynomial}}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{expr}}^{\mathtt{e}}$$

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, $\emptyset_\lambda$;
  - define `e` as the literal expression for 0.
- All of the following apply (NON_EMPTY):
  - `p` is the polynomial $f$;
  - sorting (see *sort* for details) the graph of $f$ (see `func_graph` for details) yields `monoms` — a list consisting of pairs of unitary monomials and rationals. In principle, any total order of the graph of $f$ is acceptable for sorting. The function *compare_monomial_bindings* provides one such way of ordering the graph of $f$;
  - transforming `monoms` to an expression and sign via *monomials_to_expr* yields the expression `e1` and sign `s1`;
  - define `e` as `e1` if `s1` is 1, the integer literal expression for 0 if `s1` is 0, and the unary expression negating `e1`, that is, $\mathsf{E\_Unop}(\mathtt{NEG}, \mathtt{e1})$, if `s1` is $-1$.

### Formally

EMPTY

$$\mathit{polynomial\_to\_expr}(\overbrace{\emptyset_\lambda}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(0))}^{\mathtt{e}}$$

NON_EMPTY

$$\frac{\mathit{sort}(\mathtt{func\_graph}(f), \mathit{compare\_monomial\_bindings}) = \mathtt{monoms} \qquad \mathit{monomials\_to\_expr}(\mathtt{monoms}) \xrightarrow{\mathsf{type}} (\mathtt{e1}, \mathtt{s1}) \qquad \mathtt{e} := \begin{cases} \mathsf{E\_Literal}(\mathsf{L\_Int}(0)) & \text{if } \mathtt{s1} = 0 \\ \mathtt{e1} & \text{if } \mathtt{s1} = 1 \\ \mathsf{E\_Unop}(\mathtt{NEG}, \mathtt{e1}) & \text{if } \mathtt{s1} = -1 \end{cases}}{\mathit{polynomial\_to\_expr}(\overbrace{f}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \mathtt{e}}$$

### TypingRule.CompareMonomialBindings

The function

$$\mathit{compare\_monomial\_bindings}((\overbrace{\mathsf{monomial}}^{\mathtt{m1}} \times \overbrace{\mathbb{Q}}^{\mathtt{q1}}), (\overbrace{\mathsf{monomial}}^{\mathtt{m2}} \times \overbrace{\mathbb{Q}}^{\mathtt{q2}})) \longrightarrow \overbrace{\{-1, 0, 1\}}^{\mathtt{s}}$$

compares two monomial bindings given by $(\mathtt{m1}, \mathtt{q1})$ and $(\mathtt{m2}, \mathtt{q2})$ 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 $(\texttt{x}, -1)$ to $(\texttt{x}, -1)$ yields 0.

Comparing $(\texttt{x}, 1)$ to $(\texttt{y}, 1)$ 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 to $g$;
  - `ids` is the list obtained by taking the set of identifiers in the domain of $f$ and in the domain of $g$, and sorting them according to the lexical order for identifiers (ASCII string order);
  - `v` is the first identifier in `ids` for which $f$ and $g$ behave differently (either one of them is defined for `v` and the other is not, or they both bind `v` to a different value);
  - $s$ is determined as follows: 1 if `v` is not in the domain of $f$ and is in the domain of $g$; $-1$ if `v` is not in the domain of $f$ and is in the domain of $g$; otherwise it is the sign of $g(\texttt{v}) - f(\texttt{v})$.

### Formally

The function *compare_identifier* compares two identifiers, which are lists of ASCII characters, via the lexicographic ordering.

EQUAL_MONOMIALS

$$\frac{f = g \qquad \texttt{s} := \texttt{sign}(\texttt{q2} - \texttt{q1})}{\mathit{compare\_monomial\_bindings}((\overbrace{f}^{\texttt{m1}}, \texttt{q1}), (\overbrace{g}^{\texttt{m2}}, \texttt{q2})) \xrightarrow{\text{type}} \texttt{s}}$$

DIFFERENT_MONOMIALS

$$\frac{\begin{array}{c} f \neq g \qquad \texttt{ids} := \mathit{sort}(\mathrm{dom}(f) \cup \mathrm{dom}(g), \mathit{compare\_identifier}) \\ \texttt{ids} \overset{\text{is}}{=} \texttt{ids1} + \texttt{ids2} \qquad \texttt{i} \in \mathit{indices}(\texttt{ids1}) : f(\texttt{ids1}[\texttt{i}]) = g(\texttt{ids1}[\texttt{i}]) \\ \texttt{v} := \texttt{ids2}[1] \qquad \texttt{s} := \begin{cases} 1 & f(\texttt{v}) = \bot \wedge g(\texttt{v}) \neq \bot \\ -1 & f(\texttt{v}) \neq \bot \wedge g(\texttt{v}) = \bot \\ \texttt{sign}(g(\texttt{v}) - f(\texttt{v})) & f(\texttt{v}) \neq \bot \wedge g(\texttt{v}) \neq \bot \end{cases} \end{array}}{\mathit{compare\_monomial\_bindings}((\overbrace{f}^{\texttt{m1}}, \texttt{q1}), (\overbrace{g}^{\texttt{m2}}, \texttt{q2})) \xrightarrow{\text{type}} \texttt{s}}$$

### TypingRule.MonomialsToExpr

The function

$$\mathit{monomials\_to\_expr}(\overbrace{(\overbrace{\text{unitary\_monomial}}^{\texttt{m}} \times \overbrace{\mathbb{Q}}^{q})^*}^{\texttt{monoms}}) \longrightarrow (\overbrace{\text{expr}}^{\texttt{e}}, \overbrace{\{-1, 0, 1\}}^{\texttt{s}})$$

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 $(\texttt{x} \times \texttt{y}^2, 2)$, $(\texttt{x} \times \texttt{y}^2, 3)$ and $(\texttt{x} \times \texttt{y}, -5)$ 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 $(\texttt{m}, q)$ as its `head` and `monoms1` as its `tail`;
  - transforming the unitary monomial `m` to an expression via *unitary_monomials_to_expr* yields `e1'`;

* transforming `e1'` and $q$ via *monomial_to_expr* yields the expression `e1` and sign `s1`;
* transforming `monoms` to an expression and sign via *monomials_to_expr* yields $(\texttt{e2}, \texttt{s2})$;
* symbolically adding $\texttt{e1}, \texttt{s1}, \texttt{e2}, \texttt{s2}$ via *sym_add_expr* yields $(\texttt{e}, \texttt{s})$.

**Formally**

$$\textsc{EMPTY}\quad \textit{monomials\_to\_expr}(\overbrace{[\,]}^{\texttt{monoms}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(0))}^{\texttt{e}}, \overbrace{0}^{\texttt{s}})$$

$$\textsc{NON\_EMPTY}\quad \frac{\textit{unitary\_monomials\_to\_expr}(\texttt{m}) \xrightarrow{\text{type}} \texttt{e1'} \qquad \textit{monomial\_to\_expr}(\texttt{e1'}, q) \xrightarrow{\text{type}} (\texttt{e1}, \texttt{s1}) \qquad \textit{monomials\_to\_expr}(\texttt{monoms}) \xrightarrow{\text{type}} (\texttt{e2}, \texttt{s2}) \qquad \textit{sym\_add\_expr}(\texttt{e1}, \texttt{s1}, \texttt{e2}, \texttt{s2}) \xrightarrow{\text{type}} (\texttt{e}, \texttt{s})}{\textit{monomials\_to\_expr}(\overbrace{[(\texttt{m}, q)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} (\texttt{e}, \texttt{s})}$$

**TypingRule.MonomialToExpr**

The function

$$\textit{monomial\_to\_expr}(\overbrace{\mathsf{expr}}^{\texttt{e}}, \overbrace{\mathbb{Q}}^{q}) \longrightarrow (\overbrace{\mathsf{expr}}^{\texttt{new\_e}} \times \overbrace{\{-1, 0, 1\}}^{\texttt{s}})$$

transforms an expression `e` and rational $q$ into the expression `new_e`, which represents the absolute value of `e` multiplied by $q$, and the sign of $q$ as `s`.

**Example: Transforming Expressions and Rationals**

The following are examples of inputs and outputs of *monomial_to_expr*:

$$\textit{monomial\_to\_expr}(\mathsf{E\_Var}(\texttt{x}), 0) \xrightarrow{\text{type}} (\mathsf{E\_Literal}(\mathsf{L\_Int}(0)), 0)$$

$$\textit{monomial\_to\_expr}(\mathsf{E\_Var}(\texttt{x}), 5) \xrightarrow{\text{type}} (\overbrace{\underbrace{\mathsf{E\_Literal}(\mathsf{L\_Int})}_{5} * \underbrace{\mathsf{E\_Var}}_{\texttt{x}}}^{\mathsf{E\_Binop}}, 1)$$

$$\textit{monomial\_to\_expr}(\mathsf{E\_Var}(\texttt{x}), 9/12) \xrightarrow{\text{type}} (\overbrace{(\overbrace{\underbrace{\mathsf{E\_Var}}_{\texttt{x}} * 3}^{\mathsf{E\_Binop}}) / \underbrace{\mathsf{E\_Literal}(\mathsf{L\_Int})}_{4}}^{\mathsf{E\_Binop}}, 1)$$

$$\textit{monomial\_to\_expr}(\mathsf{E\_Var}(\texttt{x}), -1) \xrightarrow{\text{type}} (\mathsf{E\_Unop}(\texttt{NEG}, \underbrace{\mathsf{E\_Var}}_{\texttt{x}}), -1)$$

**Prose**

One of the following applies:

- All of the following apply (Q_ZERO):
  - $q$ is 0;
  - `new_e` is the literal expression for 0;
  - `s` is 0.
- All of the following apply (Q_NATURAL):
  - $q$ a strictly positive;
  - symbolically multiplying the literal expression for $q$ and `e` via *sym_mul_expr* yields `new_e`;
  - `s` is 1.
- All of the following apply (Q_POSITIVE_FRACTION):
  - $q$ a strictly positive fraction, that is, not an integer;
  - the reduced representation of the fraction $q$ is $\frac{d}{n}$;
  - symbolically multiplying the literal expression for $q$ and `e` via *sym_mul_expr* yields `e2`;
  - `e` is the binary expression with operator `DIV` and operands `e2` and the literal expression for $n$;
  - `s` is 1.
- All of the following apply (Q_NEGATIVE):
  - $q$ a strictly negative;
  - transforming `e` with $-q$ to an expression and a sign via *monomial_to_expr* yields $(\texttt{new\_e}, 1)$;
  - `s` is $-1$.

**Formally**

Q_ZERO

$$\frac{q = 0}{\mathit{monomial\_to\_expr}(\mathtt{e}, q) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(0))}^{\mathtt{new\_e}}, \overbrace{0}^{\mathtt{s}})}$$

Q_NATURAL

$$\frac{q > 0 \qquad q \in \mathbb{N} \qquad \mathit{sym\_mul\_expr}(\mathsf{E\_Literal}(\mathsf{L\_Int}(q)), \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}{\mathit{monomial\_to\_expr}(\mathtt{e}, q) \xrightarrow{\mathsf{type}} (\mathtt{new\_e}, \overbrace{1}^{\mathtt{s}})}$$

Q_POSITIVE_FRACTION

$$\frac{\begin{array}{c} q > 0 \qquad q \notin \mathbb{N} \\ q \overset{\mathsf{is}}{=} \frac{d}{n} \quad \text{is the reduced fraction for } q \\ \mathit{sym\_mul\_expr}(\mathsf{E\_Literal}(\mathsf{L\_Int}(d)), \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{e2} \\ \mathtt{new\_e} := \mathsf{E\_Binop}(\mathtt{DIV}, \mathtt{e2}, \mathsf{E\_Literal}(\mathsf{L\_Int}(n))) \end{array}}{\mathit{monomial\_to\_expr}(\mathtt{e}, q) \xrightarrow{\mathsf{type}} (\mathtt{new\_e}, \overbrace{1}^{\mathtt{s}})}$$

Q_NEGATIVE

$$\frac{q < 0 \qquad \mathit{monomial\_to\_expr}(\mathtt{e}, -q) \xrightarrow{\mathsf{type}} (\mathtt{new\_e}, 1)}{\mathit{monomial\_to\_expr}(\mathtt{e}, q) \xrightarrow{\mathsf{type}} (\mathtt{new\_e}, \overbrace{-1}^{\mathtt{s}})}$$

### TypingRule.SymAddExpr

The function

$$\mathit{sym\_add\_expr}(\overbrace{\mathsf{expr}}^{\mathtt{e1}}, \overbrace{\{-1, 0, 1\}}^{\mathtt{s1}}, \overbrace{\mathsf{expr}}^{\mathtt{e2}}, \overbrace{\{-1, 0, 1\}}^{\mathtt{s2}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{expr}}^{\mathtt{e}}, \overbrace{\{-1, 0, 1\}}^{\mathtt{s}})$$

symbolically sums the expressions `e1` and `e2` with respective signs `s1` and `s2` yielding the expression `e` and sign `s`.

The effect of the function can be summarized by the following table:

| | s1 | | |
|---|---|---|---|
| s2 | -1 | 0 | 1 |
| -1 | (e1 + e2, s1) | (e2,s2) | (e1 − e2, s1) |
| 0 | (e1, s1) | (e1,s1) | (e1, s1) |
| 1 | (e1 − e2, s1) | (e2,s2) | (e1 + e2, s1) |

### 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**

ZERO

$$\frac{(\texttt{s1} = 0 \lor \texttt{s2} = 0) \qquad (\texttt{e}, \texttt{s}) := \mathsf{choice}(\texttt{s1} = 0, (\texttt{e2}, \texttt{s2}), (\texttt{e1}, \texttt{s1}))}{\mathit{sym\_add\_expr}(\texttt{e1}, \texttt{s1}, \texttt{e2}, \texttt{s2}) \xrightarrow{\mathsf{type}} (\texttt{e}, \texttt{s})}$$

SAME_SIGN

$$\frac{\texttt{s1} \neq 0 \land \texttt{s2} \neq 0 \qquad \texttt{s1} = \texttt{s2}}{\mathit{sym\_add\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{E\_Binop}(\texttt{PLUS}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}, \overbrace{\texttt{s1}}^{\texttt{s}})}$$

DIFFERENT_SIGNS

$$\frac{\texttt{s1} \neq 0 \land \texttt{s2} \neq 0 \qquad \texttt{s1} \neq \texttt{s2}}{\mathit{sym\_add\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{E\_Binop}(\texttt{MINUS}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}, \overbrace{\texttt{s1}}^{\texttt{s}})}$$

### TypingRule.UnitaryMonomialsToExpr

The function

$$\mathit{unitary\_monomials\_to\_expr}(\overbrace{(\mathsf{identifier} \times \mathbb{N})^*}^{\texttt{monoms}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{e}}$$

transforms a list of single-variable unitary monomials `monoms` into an expression `e`. Intuitively, `monoms` represented a multiplication of the single-variable unitary monomials.

### 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 $(\texttt{v}, 0)$ and its tail is `monoms`;
  - transforming `monoms1` to an expression yields `e`.
- All of the following apply (EXP_ONE):
  - `monoms` is a list where the first element is $(\texttt{v}, 1)$ and its tail is `monoms`;
  - `e1` is the variable expression for `v`;
  - transforming `monoms1` to an expression yields `e2`;
  - symbolically multiplying `e1` and `e2` via *sym_mul_expr* yields `e`.

- All of the following apply (EXP_TWO):
  - `monoms` is a list where the first element is $(\texttt{v}, 2)$ and its tail is `monoms`;
  - `e1` is the binary expression with operator `MUL` and operands E_Var(v) and E_Var(v) (that is, v squared);
  - transforming `monoms1` to an expression yields `e2`;
  - symbolically multiplying `e1` and `e2` via *sym_mul_expr* yields `e`.
- All of the following apply (EXP_GT_TWO):
  - `monoms` is a list where the first element is $(\texttt{v}, n)$ and its tail is `monoms`;
  - $n$ is greater than 1;
  - `e1` is the binary expression with operator `POW` and base operand being the variable expression for `v` and the exponent operand being the variable expression for $n$;
  - transforming `monoms1` to an expression yields `e2`;
  - symbolically multiplying `e1` and `e2` via *sym_mul_expr* yields `e`.

**Formally**

EMPTY

$$\textit{unitary\_monomials\_to\_expr}(\overbrace{[\,]}^{\texttt{monoms}}) \xrightarrow{\text{type}} \overbrace{\texttt{E\_Literal}(\texttt{L\_Int}(1))}^{\texttt{e}}$$

EXP_ZERO

$$\frac{\textit{unitary\_monomials\_to\_expr}(\texttt{monoms1}) \xrightarrow{\text{type}} \texttt{e}}{\textit{unitary\_monomials\_to\_expr}(\overbrace{[(\texttt{v}, 0)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} \texttt{e}}$$

EXP_ONE

$$\frac{\texttt{e1} := \texttt{E\_Var}(\texttt{v}) \qquad \textit{unitary\_monomials\_to\_expr}(\texttt{monoms1}) \xrightarrow{\text{type}} \texttt{e2} \qquad \textit{sym\_mul\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{e}}{\textit{unitary\_monomials\_to\_expr}(\overbrace{[(\texttt{v}, 1)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} \texttt{e}}$$

EXP_TWO

$$\frac{\texttt{e1} := \overbrace{\texttt{E\_Var}(\texttt{v})\ \texttt{MUL}\ \texttt{E\_Var}(\texttt{v})}^{\texttt{E\_Binop}} \qquad \textit{unitary\_monomials\_to\_expr}(\texttt{monoms1}) \xrightarrow{\text{type}} \texttt{e2} \qquad \textit{sym\_mul\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{e}}{\textit{unitary\_monomials\_to\_expr}(\overbrace{[(\texttt{v}, 2)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} \texttt{e}}$$

EXP_GT_TWO

$$\frac{n \geq 2 \qquad \texttt{e1} := \overbrace{\texttt{E\_Var}(\texttt{v})\ \texttt{POW}\ \texttt{E\_Literal}(n)}^{\texttt{E\_Binop}} \qquad \mathit{unitary\_monomials\_to\_expr}(\texttt{monoms1}) \xrightarrow{\texttt{type}} \texttt{e2} \qquad \mathit{sym\_mul\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{e}}{\mathit{unitary\_monomials\_to\_expr}(\overbrace{[(\texttt{v}, n)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\texttt{type}} \texttt{e}}$$

### TypingRule.SymMulExpr

The function $\mathit{sym\_mul\_expr}(\overbrace{\texttt{expr}}^{\texttt{e1}}, \overbrace{\texttt{expr}}^{\texttt{e2}}) \xrightarrow{\texttt{type}} \overbrace{\texttt{expr}}^{\texttt{e}}$ 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

ONE_OPERAND

$$\frac{(\texttt{e1} = \texttt{E\_Literal}(\texttt{L\_Int}(1)) \vee \texttt{e2} = \texttt{E\_Literal}(\texttt{L\_Int}(1))) \qquad \texttt{e} := \texttt{choice}(\texttt{e1} = \texttt{E\_Literal}(\texttt{L\_Int}(1)), \texttt{e2}, \texttt{e1})}{\mathit{sym\_mul\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{e}}$$

$$\frac{\text{NO\_ONE\_OPERAND}\quad (\texttt{e1} \neq \mathsf{E\_Literal}(\mathsf{L\_Int}(1)) \wedge \texttt{e2} \neq \mathsf{E\_Literal}(\mathsf{L\_Int}(1)))}{\mathit{sym\_mul\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Binop}(\texttt{MUL}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}}$$

### TypingRule.TypeOf

The function

$$\mathit{type\_of}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{s}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{ty}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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**

$$\text{LOCAL}\quad \frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{s}) = \mathtt{ty}}{\mathit{type\_of}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}$$

$$\text{GLOBAL}\quad \frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{s}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{s}) = \mathtt{ty}}{\mathit{type\_of}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}$$

$$\text{ERROR}\quad \frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{s}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{s}) = \bot}{\mathit{type\_of}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UI})}$$

### TypingRule.NormalizeOpt

The helper function

$$\mathit{normalize\_opt}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\langle \mathsf{expr} \rangle}^{\mathtt{new\_e\_opt}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

is similar to *normalize*, except that it returns `None` when `e` is not an expression that can be symbolically simplified. Otherwise, the result is a type error.

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 $\top$)$^{\not/\!/\,\mathtt{\#TE}}$;
  - applying *normalize* to `e` in `tenv` yields `new_e`;
  - define `new_e_opt` as $\langle$`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

$$\text{NORMALIZABLE}\quad \frac{\mathit{to\_ir}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{p1} \;/\!/\; \mathtt{\#TE} \qquad \mathtt{p1} \neq \top \qquad \mathit{normalize}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}{\mathit{normalize}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \overbrace{\langle \mathtt{new\_e} \rangle}^{\mathtt{new\_e\_opt}}}$$

$$\text{NOT\_NORMALIZABLE}\quad \frac{\mathit{to\_ir}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \top}{\mathit{normalize}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{None}}^{\mathtt{new\_e\_opt}}}$$

# Chapter 34

# Type System Utility Rules

### 34.0.1 Checked Transitions

We define the following rules to allow us asserting that a condition holds, returning a type error otherwise:

CHECK_TRANS_TRUE

$$\mathit{check}(\texttt{TRUE}, \texttt{code}) \longrightarrow \texttt{TRUE}$$

CHECK_TRANS_FALSE

$$\mathit{check}(\texttt{FALSE}, \texttt{code}) \longrightarrow \mathsf{TypeError}(\texttt{code})$$

### 34.0.2 Converting a List of Pairs to a Map

The parametric function

$$\mathit{pairs\_to\_map}(\overbrace{(\mathsf{identifier} \times T)^*}^{\texttt{pairs}}) \longrightarrow \overbrace{(\mathsf{identifier} \rightharpoonup T)}^{f} \cup \mathsf{TTypeError}$$

converts a list of pairs — `pairs` — where each pair consists of an identifier and a value of type $T$ into a function mapping each identifier to its respective value in the list. If a duplicate identifier exists in `pairs` then a type error is returned.

#### Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `pairs` is empty;
  - $f$ is the empty function.
- All of the following apply (ERROR):

  - there exist two different positions in the list where the identifier is the same;
  - the result is a type error indicating the existence of a duplicate identifier.

- All of the following apply (OKAY):
  - all identifiers occurring in the list are unique;
  - $f$ is a function that associates to each identifier the value appearing with it in `pairs`.

**Formally**

$$\frac{}{\mathit{pairs\_to\_map}([\,]) \xrightarrow{\mathsf{type}} \emptyset_\lambda}\ \text{EMPTY}$$

$$\frac{i, j \in 1..k \qquad i \neq j \qquad \mathtt{id}_i = \mathtt{id}_j}{\mathit{pairs\_to\_map}([i = 1..k : (\mathtt{id}_i, t_i)]) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_IAD})}\ \text{ERROR}$$

$$\frac{\forall i, j \in 1..k . \mathtt{id}_i \neq \mathtt{id}_j \qquad f := \lambda \mathtt{id}.\ \begin{cases} t_i & \text{if } i \in 1..k \wedge \mathtt{id} = \mathtt{id}_i \\ \bot & \text{otherwise} \end{cases}}{\mathit{pairs\_to\_map}([i = 1..k : (\mathtt{id}_i, t_i)]) \xrightarrow{\mathsf{type}} f}\ \text{OKAY}$$

### TypingRule.CheckNoDuplicates

The function

$$\mathit{check\_no\_duplicates}(\overbrace{\mathsf{identifier}^*}^{\mathtt{id}_{1..k}}) \longrightarrow \{\mathtt{TRUE}\} \cup \mathsf{TTypeError}$$

checks whether a non-empty list of identifiers contains a duplicate identifier. If it does not, the result is `TRUE` and otherwise the result is a type error.

### Example: Checking for Absence of Duplicates in an Identifier List

In Listing 13.19, 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 13.26, 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 13.27, 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

$$\text{OKAY}\quad \frac{|\{\texttt{id}_{1..k}\}| = k}{\mathit{check\_no\_duplicates}(\texttt{id}_{1..k}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

$$\text{ERROR}\quad \frac{i, j \in 1..k \qquad i \neq j \qquad \texttt{id}_i = \texttt{id}_j}{\mathit{check\_no\_duplicates}(\texttt{id}_{1..k}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_IAD})}$$

### TypingRule.Sort

The parametric function

$$\mathit{sort}(\overbrace{T^*}^{\texttt{l1}}, \overbrace{(T \times T) \to \{-1, 0, 1\}}^{\texttt{compare}}) \xrightarrow{\text{type}} \overbrace{T^*}^{\texttt{l2}}$$

sorts a list of elements of type $T$ — `l1` — using the comparison function `compare`, resulting in the sorted list `l2`. `compare`$(a, b)$ returns 1 to mean that $a$ should be ordered before $b$, 0 to mean that $a$ and $b$ can be ordered in any way, and $-1$ to mean that $b$ should be ordered before $a$.

### Example: Sorting Identifiers

The following is an example of sorting lists of identifiers using lexicographic order:
$\mathit{sort}([\texttt{y}, \texttt{x}], \mathit{compare\_identifier}) \xrightarrow{\text{type}} [\texttt{x}, \texttt{y}]$.

The following is an example of sorting monomial bindings" (see TypingRule.PolynomialToExpr):
$\mathit{sort}([(x, 1), (y, 1), (x, -1)], \mathit{compare\_monomial\_bindings}) \xrightarrow{\text{type}} [(x, -1), (x, 1), (y, 1)]$.

### Prose

One of the following applies:

- All of the following apply (EMPTY_OR_SINGLE):
  - `l1` is either empty or contains a single element;
  - `l2` is `l1`.
- All of the following apply (TWO_OR_MORE):
  - `l1` contains at least two elements;
  - $f$ is a permutation of $1..n$;
  - `l2` is the application of the permutation $f$ to `l1`;
  - applying `compare` to every pair of consecutive elements in `l2` yields either 0 or 1.

### Formally

EMPTY_OR_SINGLE

$$\frac{|\texttt{l1}| = n \qquad n < 2}{\mathit{sort}(\texttt{l1}, \texttt{compare}) \xrightarrow{\text{type}} \overbrace{\texttt{l1}}^{\texttt{l2}}}$$

TWO_OR_MORE

$$\frac{\begin{array}{c}|\texttt{l1}| = n \qquad f : 1..n \to 1..n \text{ is a bijection} \\ \texttt{l2} := [\ i = 1..n : \texttt{l1}[f(i)]\ ] \qquad i = 1..n-1 : \texttt{compare}(\texttt{l2}[i], \texttt{l2}[i+1]) \geq 0\end{array}}{\mathit{sort}(\texttt{l1}, \texttt{compare}) \xrightarrow{\text{type}} \texttt{l2}}$$

### TypingRule.FindBitfieldOpt

The function

$$\mathit{find\_bitfield\_opt}(\overbrace{\text{identifier}}^{\texttt{name}}, \overbrace{\text{bitfield}^*}^{\texttt{bitfields}}) \longrightarrow \overbrace{\langle \text{bitfield} \rangle}^{\texttt{r}}$$

returns the bitfield associated with the name `name` in the list of bitfields `bitfields`, if there is one. Otherwise, the result is `None`.

### Example: Finding Bitfields

In Listing 15.22, 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 15.21, 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

MATCH

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name}}{\mathit{find\_bitfield\_opt}(\texttt{name}, \overbrace{\texttt{bf} + \texttt{bitfields'}}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \overbrace{\langle \texttt{bf} \rangle}^{\texttt{r}}}$$

TAIL

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name'} \qquad \texttt{name} \neq \texttt{name'} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bitfields'}) \xrightarrow{\text{type}} \texttt{r}}{\mathit{find\_bitfield\_opt}(\texttt{name}, \overbrace{\texttt{bf} + \texttt{bitfields'}}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \texttt{r}}$$

EMPTY

$$\mathit{find\_bitfield\_opt}(\texttt{name}, \overbrace{[\,]}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \texttt{None}$$

### TypingRule.TypeOfArrayLength

The function

$$\mathit{type\_of\_array\_length}(\overbrace{\mathsf{array\_index}}^{\texttt{size}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{t}}$$

returns the type for the array length `size` in `t`.

### Example: Retrieving the Type of an Array from an Array Index

In Listing 13.22, annotating the expression `int_arr[[3]]`, yields
ArrayLength_Expr( $\overset{\text{E\_Literal(L\_Int)}}{\boxed{3}}$ ), and

$$\mathit{type\_of\_array\_length}(\text{ArrayLength\_Expr}(\overset{\text{E\_Literal(L\_Int)}}{\boxed{3}})) \xrightarrow{\text{type}} \text{unconstrained\_integer}$$

Annotating the expression `big_little_arr[[LITTLE]]` yields
ArrayLength_Enum(`BitsArray`, [**BIG**, **LITTLE**]), and

$$\mathit{type\_of\_array\_length}(\text{ArrayLength\_Enum}(\texttt{BitsArray}, [\textbf{BIG}, \textbf{LITTLE}])) \xrightarrow{\text{type}} \text{T\_Named}(\texttt{BitsArray}) \ .$$

### 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

$$\mathit{type\_of\_array\_length}(\text{ArrayLength\_Enum}(\texttt{s}, \_)) \xrightarrow{\text{type}} \text{T\_Named}(\texttt{s})$$

EXPR

$$\mathit{type\_of\_array\_length}(\text{ArrayLength\_Expr}(\_)) \xrightarrow{\text{type}} \text{T\_Int}(\text{Unconstrained})$$

### TypingRule.AssocOpt

The function

$$\mathit{assoc\_opt}(\overbrace{(\text{identifier} \times T)^*}^{\texttt{li}}, \overbrace{\text{identifier}}^{\texttt{id}}) \xrightarrow{\text{type}} \langle \overbrace{T}^{\texttt{v}} \rangle$$

returns the value `v` associated with the identifier `id` in the list of pairs `li` or `None`, if no such association exists.

**Example: Finding a Value Associated with an Identifier**

In Listing 15.19, 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:

$$\mathit{assoc\_opt}([(\texttt{i}, \mathsf{unconstrained\_integer}), (\texttt{b}, \mathsf{T\_Bool})], \texttt{i}) \xrightarrow{\text{type}} \langle \mathsf{unconstrained\_integer} \rangle$$
$$\mathit{assoc\_opt}([(\texttt{i}, \mathsf{unconstrained\_integer}), (\texttt{b}, \mathsf{T\_Bool})], \texttt{b}) \xrightarrow{\text{type}} \langle \mathsf{T\_Bool} \rangle$$

In Listing 15.20, however, annotating the expression `my_record.undeclared_identifier` fails:

$$\mathit{assoc\_opt}([(\texttt{i}, \mathsf{unconstrained\_integer}), (\texttt{b}, \mathsf{T\_Bool})], \texttt{undeclared\_identifier}) \xrightarrow{\text{type}} \texttt{None}$$

**Prose**

One of the following applies:

- All of the following apply (MEMBER):
  - a pair $(\texttt{id}, \texttt{v})$ exists in the list `li`;
  - the result is $\langle \texttt{v} \rangle$.
- All of the following apply (NOT_MEMBER):
  - every pair $(\texttt{x}, \_)$ in the list `li` has $\texttt{x} \neq \texttt{id}$;
  - the result is `None`.

**Formally**

$$\frac{\text{NOT\_MEMBER} \quad (\texttt{x}, \texttt{v}) \in \texttt{li} : \texttt{x} \neq \texttt{id}}{\mathit{assoc\_opt}(\texttt{li}, \texttt{id}) \xrightarrow{\text{type}} \texttt{None}} \qquad \frac{\text{MEMBER} \quad (\texttt{id}, \texttt{v}) \in \texttt{li}}{\mathit{assoc\_opt}(\texttt{li}, \texttt{id}) \xrightarrow{\text{type}} \langle \texttt{v} \rangle}$$

## 34.1 Static Environment Utilities

**TypingRule.WithEmptyLocal**

The function

$$\mathit{with\_empty\_local}(\overbrace{\mathbb{GSE}}^{\text{genv}}) \longrightarrow \overbrace{\mathbb{SE}}^{\text{tenv}}$$

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 $\emptyset_{\mathbb{SE}}$.

#### Formally

$$\mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} (\texttt{genv}, L^{\emptyset_{\mathbb{SE}}})$$

### TypingRule.CheckVarNotInEnv

The function

$$\mathit{check\_var\_not\_in\_env}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathbb{S}}^{\text{id}}) \longrightarrow \{\texttt{TRUE}\} \cup \mathsf{TTypeError}$$

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 $\texttt{TRUE}^{/\!/\texttt{TE\_IAD}}$.

### Formally

$$\frac{\mathit{is\_undefined}(\texttt{tenv}, \texttt{id}) \xrightarrow{\text{type}} \texttt{b} \qquad \mathit{check}(\texttt{b}, \texttt{TE\_IAD}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_var\_not\_in\_env}(\texttt{tenv}, \texttt{id}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

### TypingRule.CheckVarNotInGEnv

The function

$$\mathit{check\_var\_not\_in\_genv}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathbb{S}}^{\texttt{x}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

checks whether `id` is already declared in the global static environment `genv`. If it is, the result is a type error, and otherwise the result is `TRUE`.

### 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 $\texttt{TRUE}^{/\!/\texttt{TE\_IAD}}$.

### Formally

$$\frac{\mathit{is\_global\_undefined}(\texttt{genv}, \texttt{id}) \xrightarrow{\text{type}} \texttt{b} \qquad \mathit{check}(\texttt{b}, \texttt{TE\_IAD}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_var\_not\_in\_genv}(\texttt{genv}, \texttt{id}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

### TypingRule.AddLocal

The function

$$\mathit{add\_local}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{id}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}, \overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

adds the identifier `id` as a local storage element with type `ty` and local declaration keyword `ldk` to the local environment of `tenv`, resulting in the static environment `new_tenv`.

### 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, $(\texttt{ty}, \texttt{ldk})$;
- `new_tenv` is defined by updating the local environment with the binding of local_storage_types to `new_local_storagetypes`.

### Formally

$$\frac{\begin{array}{l}\texttt{new\_local\_storagetypes} := L^{\texttt{tenv}}.\text{local\_storage\_types}[\texttt{id} \mapsto (\texttt{ty}, \texttt{ldk})] \\ \texttt{new\_tenv} := (G^{\texttt{tenv}}, L^{\texttt{tenv}}[\text{local\_storage\_types} \mapsto \texttt{new\_local\_storagetypes}])\end{array}}{\mathit{add\_local}(\texttt{tenv}, \texttt{id}, \texttt{ty}, \texttt{ldk}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

### TypingRule.IsUndefined

The function

$$\mathit{is\_undefined}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\texttt{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

checks whether the identifier `x` is defined as a storage element in the static environment `tenv`.

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

$$\frac{\mathit{is\_global\_undefined}(G^{\texttt{tenv}}, \texttt{x}) \xrightarrow{\text{type}} \texttt{b1} \qquad \mathit{is\_local\_undefined}(L^{\texttt{tenv}}, \texttt{x}) \xrightarrow{\text{type}} \texttt{b2}}{\mathit{is\_undefined}(\texttt{tenv}, \texttt{x}) \xrightarrow{\text{type}} \overbrace{\texttt{b1} \wedge \texttt{b2}}^{\texttt{b}}}$$

### TypingRule.IsGlobalUndefined

The function

$$\mathit{is\_global\_undefined}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{identifier}}^{\texttt{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

checks whether the identifier `x` is defined in the global static environment `genv` when subprogram definitions are ignored (see Guide.GlobalNamespace), yielding the result in `b`.

### Example: Checking Whether an Identifier is Associated with a Global Storage Element

Listing 34.6 shows a specification which defines a variable `X` and a subprogram `Y`. In the typing environment obtained by typechecking this specification, *is_global_undefined* will yield `FALSE` for `X` and `TRUE` for `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 `b` as `TRUE` if and only if `x` is not bound in any of the following maps of `genv`: global_storage_types, and declared_types.

### Formally

$$\frac{\begin{array}{rlcl} \texttt{b} := & \texttt{genv}.\mathsf{global\_storage\_types}(\texttt{x}) & = & \bot \wedge \\ & \texttt{genv}.\mathsf{declared\_types}(\texttt{x}) & = & \bot \end{array}}{\mathit{is\_global\_undefined}(\texttt{genv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.IsLocalUndefined

The function

$$\mathit{is\_local\_undefined}(\overbrace{\mathbb{LSE}}^{\texttt{lenv}}, \overbrace{\mathsf{identifier}}^{\texttt{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

checks whether `x` is declared as a local storage element in the static local environment `lenv`, yielding the result in `b`.

### 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

$$\mathit{is\_local\_undefined}(\texttt{lenv}, \texttt{x}) \xrightarrow{\text{type}} \overbrace{L^{\text{tenv}}.\text{local\_storage\_types}(\texttt{x}) = \bot}^{\text{b}}$$

### TypingRule.LookupConstant

The function

$$\mathit{lookup\_constant}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{s}}) \longrightarrow \overbrace{\text{literal}}^{\text{v}} \cup \{\bot\}$$

looks up the environment tenv for a constant `v` associated with an identifier `s`. The result is $\bot$ if `s` is not associated with any constant.

### Example: Looking Up Constants

The specification in Listing 34.7 shows examples where global constants, local 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.

    constant HALF_WORD_SIZE = 32;
    var bv2: bits(HALF_WORD_SIZE); // Requires looking up HALF_WORD_SIZE.

    return 0;
end;
```

### Prose

One of the following applies:

- All of the following apply (LOCAL):
  - `s` is associated with a constant `v` in the local environment of tenv;
- All of the following apply (GLOBAL):
  - `s` is not associated with a constant in the local environment of tenv;
  - `s` is associated with a constant `v` in the global environment of tenv;
- All of the following apply (NOT_FOUND):
  - `s` is not associated with a constant in the local environment of tenv;
  - `s` is not associated with a constant in the global environment of tenv;
  - the result is $\bot$.

### Formally

$$\frac{L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \mathtt{v}}{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{v}}\ \text{LOCAL}$$

$$\frac{L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \mathtt{v}}{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{v}}\ \text{GLOBAL}$$

$$\frac{L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \bot}{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot}\ \text{NOT\_FOUND}$$

### TypingRule.AddGlobalConstant

The function

$$\mathit{add\_global\_constant}(\overbrace{\mathbb{GSE}}^{\mathtt{genv}}, \overbrace{\mathsf{identifier}}^{\mathtt{name}}, \overbrace{\mathsf{literal}}^{\mathtt{v}}) \xrightarrow{\mathsf{type}} \overbrace{\mathbb{GSE}}^{\mathtt{new\_genv}}$$

binds the identifier `name` to the literal `v` in the global static environment `genv`, yielding the updated global static environment `new_genv`.

### 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.

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

$$\mathit{add\_global\_constant}(\texttt{genv}, \texttt{name}, \texttt{v}) \xrightarrow{\text{type}} \overbrace{\texttt{genv.constant\_values}[\texttt{name} \mapsto \texttt{v}]}^{\texttt{new\_genv}}$$

### TypingRule.AddLocalConstant

The function

$$\mathit{add\_local\_constant}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{literal}}^{\texttt{v}}) \xrightarrow{\text{type}} \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

binds the identifier `name` to the literal `v` in the local static environment component of the static environment `tenv`, yielding the updated static environment `new_tenv`.

### Example: Binding Local Storage Elements to Constants

The specification in Listing 34.9 is well-typed. Specifically, the type system binds `z` to `32` upon annotating `constant z: integer {0..100} = foo(x);`, which enables it to infer that the statement `let bv: bits(32) = Zeros{z};` is well-typed.

Listing 34.9: Binding constants to local storage elements

```
func foo(x: integer {0..100}) => integer {0..100}
begin
  return x;
end;

func main() => integer
begin
  constant x = 32;
  constant z: integer {0..100} = foo(x);
```

```
  // The static environment binds z to 32.

  let bv: bits(32) = Zeros{z};
  return 0;
end;
```

### Prose

Define `new_tenv` as tenv with the global component updated such that its constant_values map is updated to bind `name` to `v`.

### Formally

$$\mathit{add\_local\_constant}(\mathsf{tenv}, \mathtt{name}, \mathtt{v}) \xrightarrow{\text{type}} \overbrace{(G^{\mathsf{tenv}}.\mathsf{constant\_values}[\mathtt{name} \mapsto \mathtt{v}], L^{\mathsf{tenv}})}^{\mathtt{new\_tenv}}$$

### TypingRule.LookupImmutableExpr

The function

$$\mathit{lookup\_immutable\_expr}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{identifier}}^{\mathtt{x}}) \longrightarrow \overbrace{\mathsf{expr}}^{\mathtt{e}} \cup \{\bot\}$$

looks up the static environment tenv for an immutable expression associated with the identifier `x`, returning $\bot$ 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 $\bot$;
  - applying expr_equiv to `x` in the global component of tenv, yields `e`.
- All of the following apply (NONE):
  - applying expr_equiv to `x` in the local component of tenv, yields $\bot$;
  - applying expr_equiv to `x` in the global component of tenv, yields $\bot$;
  - `e` is $\bot$.

**Formally**

$$\text{LOCAL}\ \frac{L^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \mathsf{e}}{\mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \mathsf{e}}$$

$$\text{GLOBAL}\ \frac{L^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \mathsf{e}}{\mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \mathsf{e}}$$

$$\text{NONE}\ \frac{L^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \bot}{\mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \bot}$$

### TypingRule.AddGlobalImmutableExpr

The function

$$\mathit{add\_global\_immutable\_expr}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{identifier}}^{\mathtt{x}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}}$$

binds the identifier `x`, which is assumed to name a global storage element, to the expression `e`, which is assumed to be symbolically evaluable, in the static environment `tenv`, resulting in the updated environment `new_tenv`.

### Example: Remembering Global Immutable Expressions

The specification in Listing 34.10 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 34.10: 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**

$$\mathit{add\_global\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{(G^{\mathsf{tenv}}.\mathsf{expr\_equiv}[\mathtt{x} \mapsto \mathsf{e}], L^{\mathsf{tenv}})}^{\mathtt{new\_tenv}}$$

### TypingRule.AddLocalImmutableExpr

The function

$$\mathit{add\_local\_immutable\_expr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{x}}, \overbrace{\mathsf{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

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

$$\mathit{add\_local\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}, \mathtt{e}) \xrightarrow{\mathsf{type}} \overbrace{(G^{\mathsf{tenv}}, L^{\mathsf{tenv}}.\mathsf{expr\_equiv}[\mathtt{x} \mapsto \mathtt{e}])}^{\texttt{new\_tenv}}$$

### TypingRule.ShouldRememberImmutableExpression

The helper function

$$\mathit{should\_remember\_immutable\_expr}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \xrightarrow{\mathsf{type}} \overbrace{\mathbb{B}}^{\texttt{b}}$$

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 34.11 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 34.11: Remembering vs. not remembering immutable expressions

```
constant N = 15;

func main() => integer
begin
```

```
    var l = 1;
    let x: integer{1..2 * N} = l;
    // 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` with assertion side effect descriptor removed from it, yields `TRUE`.

### Formally

$$\frac{\mathit{is\_symbolically\_evaluable}(\texttt{ses} \setminus \{\mathsf{PerformsAssertions}\}) \xrightarrow{\text{type}} \texttt{b}}{\mathit{should\_remember\_immutable\_expr}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{b}}$$

### TypingRule.AddImmutableExpr

The function

$$\mathit{add\_immutable\_expr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}, \overbrace{\langle \overbrace{\mathsf{expr}}^{\texttt{e}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_e}} \rangle}^{\texttt{e\_opt}}, \overbrace{\mathsf{identifier}}^{\texttt{x}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \cup \mathsf{TTypeError}$$

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 34.12, 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 34.12: 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 either LDK_Constant or LDK_Let;
  - applying *should_remember_immutable_expr* to `ses_e` yields `TRUE`;
  - applying *normalize* to `e` in `tenv` yields $\texttt{e'}^{\texttt{// \#TE}}$;
  - applying *add_local_immutable_expr* to `x` and `e` yields `new_tenv`.
- All of the following apply (FAIL):
  - One of the following applies:
    - `e'` is `None`;
    - `ldk` is neither LDK_Constant nor LDK_Let;
    - `e'` contains the expression `e` and set of side effect descriptors `ses_e` and applying *should_remember_immutable_expr* to `ses_e` yields `TRUE`;
  - define `new_tenv` as `tenv`.

### Formally

OK

$$\frac{\begin{array}{c}\texttt{ldk} \in \{\texttt{LDK\_Constant}, \texttt{LDK\_Let}\} \\ \textit{should\_remember\_immutable\_expr}(\texttt{ses\_e}) \xrightarrow{\text{type}} \texttt{TRUE} \\ \textit{normalize}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} \texttt{e' // \#TE} \\ \textit{add\_local\_immutable\_expr}(\texttt{x}, \texttt{e'}) \xrightarrow{\text{type}} \texttt{new\_tenv}\end{array}}{\textit{add\_immutable\_expr}(\texttt{tenv}, \texttt{ldk}, \overbrace{\langle \texttt{e}, \texttt{ses\_e} \rangle}^{\texttt{e\_opt}}, \texttt{x}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

$$\frac{\begin{array}{l}\text{FAIL}\\ \texttt{ldk} \notin \{\text{LDK\_Constant}, \text{LDK\_Let}\} \quad \vee\\ \texttt{e\_opt} = \text{None} \quad \vee\\ \texttt{e\_opt} = \langle \texttt{e}, \texttt{ses\_e} \rangle \wedge \mathit{should\_remember\_immutable\_expr}(\texttt{ses\_e}) \xrightarrow{\text{type}} \texttt{FALSE}\end{array}}{\mathit{add\_immutable\_expr}(\texttt{tenv}, \texttt{ldk}, \texttt{e\_opt}, \texttt{x}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}}$$

### TypingRule.AddSubprogram

The function

$$\mathit{add\_subprogram}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{S}}^{\texttt{name}}, \overbrace{\text{func}}^{\texttt{func\_def}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\texttt{s}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

updates the global environment of `tenv` by mapping the (unique) subprogram identifier `name` to the function definition `func_def` and side effect descriptors `s` in `tenv`, resulting in a new static environment `new_tenv`.

### Example: Updating a Static Environment for a Subprogram Declaration

Consider Listing 28.6 and assume that the integer type version of `increment` is annotated before the real type version. Then, annotating the integer type version of `increment` applies *add_subprogram* to the empty environment, `increment` as a name, the subprogram definition node

$$\left\{\begin{array}{rcl}\text{name} & : & \texttt{increment},\\ \text{parameters} & : & [\,],\\ \text{args} & : & [(\texttt{x}, \text{unconstrained\_integer})],\\ \text{body} & : & \text{S\_Return}(\overbrace{\overbrace{\texttt{x}}^{\text{E\_Var}} + \overbrace{1}^{\text{E\_Literal(L\_Int)}}}^{\text{E\_Binop}}),\\ \text{return\_type} & : & \langle \text{unconstrained\_integer} \rangle,\\ \text{subprogram\_type} & : & \texttt{ST\_Function}\\ \text{recurse\_limit} & : & \text{None}\\ \text{builtin} & : & \texttt{FALSE}\\ \text{override} & : & \text{None}\end{array}\right\}$$

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

$$\left\{\begin{array}{rcl} \text{name} & : & \texttt{increment-1}, \\ \text{parameters} & : & [\,], \\ \text{args} & : & [(\texttt{x}, \mathsf{T\_Real})], \\ \text{body} & : & \mathsf{S\_Return}(\overbrace{\overbrace{\texttt{x}}^{\mathsf{E\_Var}} + \mathsf{E\_Literal}(\mathsf{L\_Real}(1/1))}^{\mathsf{E\_Binop}}), \\ \text{return\_type} & : & \langle \mathsf{T\_Real} \rangle, \\ \text{subprogram\_type} & : & \texttt{ST\_Function} \\ \text{recurse\_limit} & : & \texttt{None} \\ \text{builtin} & : & \texttt{FALSE} \\ \text{override} & : & \texttt{None} \end{array}\right\}$$

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

$$\frac{\texttt{new\_tenv} := (G^{\text{tenv}}.\text{subprograms}[\texttt{name} \mapsto (\texttt{func\_def}, \texttt{s})], L^{\text{tenv}})}{\mathit{add\_subprogram}(\texttt{tenv}, \texttt{name}, \texttt{func\_def}, \texttt{s}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

### TypingRule.AddType

The function

$$\mathit{add\_type}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{identifier}}^{\texttt{name}}, \overbrace{\text{ty}}^{\texttt{ty}}, \overbrace{\text{TimeFrame}}^{\texttt{f}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

binds the type `ty` and time frame `f` to the identifier `name` in the static environment `tenv`, yielding the modified static environment `new_tenv`.

### 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 Constant as the time frame, binding `Color` to the pair (T_Enum(`RED, GREEN, BLUE`), Constant).

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 Execution as the time frame (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

$$\mathit{add\_type}(\mathsf{tenv}, \mathtt{name}, \mathtt{ty}, \mathtt{f}) \xrightarrow{\mathsf{type}} \overbrace{(G^{\mathsf{tenv}}.\mathsf{declared\_types}[\mathtt{name} \mapsto (\mathtt{ty}, \mathtt{f})], L^{\mathsf{tenv}})}^{\mathtt{new\_tenv}}$$

# 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.CreateBitvector;
- SemanticsRule.ConcatBitvectors;
- SemanticsRule.ReadFromBitvector;
- SemanticsRule.WriteToBitvector;
- SemanticsRule.GetIndex;
- SemanticsRule.SetIndex;
- SemanticsRule.GetField;
- SemanticsRule.SetField;
- SemanticsRule.DeclareLocalIdentifier;
- SemanticsRule.DeclareLocalIdentifierM;
- SemanticsRule.DeclareLocalIdentifierMM;

**SemanticsRule.GetStackSize**

The function

$$\mathit{get\_statck\_size}(\overbrace{\mathsf{denv}}^{\mathbb{DE}}, \overbrace{\mathtt{name}}^{\text{identifier}}) \longrightarrow \overbrace{\mathtt{s}}^{\mathbb{N}}$$

retrieves the value associated with `name` in denv.stack_size or 0 if no value is associated with it.

**Prose**

define `s` is 0 if no value is associated with `name` in denv.stack_size and the value bound to `name` in denv.stack_size otherwise.

**Formally**

$$\frac{\mathtt{s} := \mathsf{choice}(\mathtt{name} \in \mathrm{dom}(\mathsf{denv.stack\_size}), \mathsf{denv.stack\_size}(\mathtt{name}), 0)}{\mathit{get\_statck\_size}(\mathsf{denv}, \mathtt{name}) \xrightarrow{\mathsf{eval}} \mathtt{s}}$$

**SemanticsRule.SetStackSize**

The function

$$\mathit{set\_statck\_size}(\overbrace{\mathtt{genv}}^{\mathbb{GDE}}, \overbrace{\mathtt{name}}^{\text{identifier}}, \overbrace{\mathtt{v}}^{\mathbb{N}}) \longrightarrow \overbrace{\mathtt{new\_genv}}^{\mathbb{DE}}$$

updates the value bound to `name` in `genv`.storage to `v`, yielding the new global dynamic environment `new_genv`.

**Prose**

define `new_denv` as `genv` updated to bind `name` to `v` in `genv`.stack_size.

**Formally**

$$\mathit{set\_statck\_size}(\mathtt{genv}, \mathtt{name}, \mathtt{v}) \xrightarrow{\mathsf{eval}} \overbrace{\mathtt{genv.stack\_size}[\mathtt{name} \mapsto \mathtt{v}]}^{\mathtt{new\_genv}}$$

**SemanticsRule.IncrStackSize**

The function

$$\mathit{incr\_statck\_size}(\overbrace{\mathtt{genv}}^{\mathbb{GDE}}, \overbrace{\mathtt{name}}^{\text{identifier}}) \longrightarrow \overbrace{\mathtt{new\_genv}}^{\mathbb{GDE}}$$

increments the value associated with `name` in `genv`.stack_size, yielding the updated global dynamic environment `new_genv`.

**Prose**

All of the following apply:

- applying *get_statck_size* to `name` in $(\texttt{genv}, \emptyset_\lambda)$ yields `prev`;
- applying *set_statck_size* to `name` and $\texttt{prev} + 1$ in `genv` yields `new_genv`.

**Formally**

$$\frac{\mathit{get\_statck\_size}((\texttt{genv}, \emptyset_\lambda)v, \texttt{name}) \xrightarrow{\text{eval}} \texttt{prev} \quad \mathit{set\_statck\_size}(\texttt{genv}, \texttt{name}, \texttt{prev} + 1) \xrightarrow{\text{eval}} \texttt{new\_genv}}{\mathit{incr\_statck\_size}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{eval}} \texttt{new\_genv}}$$

**SemanticsRule.DecrStackSize**

The function

$$\mathit{decr\_statck\_size}(\overbrace{\texttt{genv}}^{\mathbb{GDE}}, \overbrace{\texttt{name}}^{\text{identifier}}) \longrightarrow \overbrace{\texttt{new\_genv}}^{\mathbb{GDE}} \cup \overbrace{\texttt{new\_denv}}^{\mathbb{DE}}$$

decrements the value associated with `name` in `genv`.stack_size, yielding the updated global dynamic environment `new_genv`. It is assumed that $\mathit{get\_statck\_size}((\texttt{genv}, \emptyset_\lambda), \texttt{name})$ yields a positive value.

**Prose**

All of the following apply:

- applying *get_statck_size* to `name` in $(\texttt{genv}, \emptyset_\lambda)$ yields `prev`;
- applying *set_statck_size* to `name` and $\texttt{prev} - 1$ in `genv` yields `new_genv`.

**Formally**

$$\frac{\mathit{get\_statck\_size}((\texttt{genv}, \emptyset_\lambda), \texttt{name}) \xrightarrow{\text{eval}} \texttt{prev} \quad \mathit{set\_statck\_size}(\texttt{genv}, \texttt{name}, \texttt{prev} - 1) \xrightarrow{\text{eval}} \texttt{new\_genv}}{\mathit{decr\_statck\_size}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{eval}} \texttt{new\_genv}}$$

**SemanticsRule.RemoveLocal**

**Prose**

The relation

$$\mathit{remove\_local}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}) \times \overbrace{\mathbb{E}}^{\texttt{new\_env}}$$

removes the binding of the identifier `name` from the local storage of the environment `env`, yielding the environment `new_env`.

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 — $G^{\mathsf{denv}}$, and local component $L^{\mathsf{denv}}$, with the identifier `name` removed from its domain.

**Formally**

(Recall that $[\texttt{name} \mapsto \bot]$ means that `name` is not in the domain of the resulting function.)

$$\frac{\texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}[\texttt{name} \mapsto \bot]))}{\mathit{remove\_local}(\texttt{env}, \texttt{name}) \xrightarrow{\mathsf{eval}} \texttt{new\_env}}$$

### SemanticsRule.ReadIdentifier

**Prose**

The relation

$$\mathit{read\_identifier}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes \mathcal{G}$$

reads a value `v` into a storage element given by an identifier `name`. The result is an execution graph containing a single Read Effect, which denotes reading from `name`.

**Formally**

$$\mathit{read\_identifier}(\texttt{name}, \texttt{v}) \xrightarrow{\mathsf{eval}} \mathsf{ReadEffect}(\texttt{name})$$

### SemanticsRule.WriteIdentifier

**Prose**

The relation

$$\mathit{write\_identifier}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes \mathcal{G}$$

writes the value `v` into a storage element given by an identifier `name`. The result is an execution graph containing a single Write Effect, which denotes writing into `name`.

**Formally**

$$\mathit{write\_identifier}(\texttt{name}, \texttt{v}) \xrightarrow{\mathsf{eval}} \mathsf{WriteEffect}(\texttt{name})$$

### SemanticsRule.CreateBitvector

#### Prose

The relation

$$\mathit{create\_bitvector}(\overbrace{\mathbb{V}^*}^{\texttt{vs}}) \times \mathcal{BV}$$

creates a native vector value bitvector from a sequence of values `vs`.

#### Formally

$$\mathit{create\_bitvector}(\texttt{vs}) \xrightarrow{\text{eval}} \texttt{Bitvector}\,\texttt{vs}$$

### SemanticsRule.ConcatBitvectors

The relation

$$\mathit{concat\_bitvectors}(\overbrace{\mathcal{BV}^*}^{\texttt{vs}}) \times \overbrace{\mathcal{BV}}^{\texttt{new\_vs}}$$

transforms a (possibly empty) list of bitvector native values `vs` into a single bitvector `new_vs`.

#### 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` $\texttt{vs}'$;
  - view `v` as the native value bitvector for the sequence of bits `bv`;
  - applying *concat_bitvectors* to $\texttt{vs}'$ yields the native value bitvector for the sequence of bits `bv`;
  - define `res` as the concatenation of `bv` and $\texttt{bv}'$;
  - 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**

$$\text{EMPTY}\quad \mathit{concat\_bitvectors}(\overbrace{[\,]}^{\texttt{vs}}) \xrightarrow{\text{eval}} \overbrace{\texttt{Bitvector}([\,])}^{\texttt{new\_vs}}$$

$$\text{NON\_EMPTY}\quad \frac{\texttt{v} \overset{\text{is}}{=} \texttt{Bitvector}(\texttt{bv}) \qquad \texttt{vs} = [\texttt{v}] + \texttt{vs}' \qquad \mathit{concat\_bitvectors}(\texttt{vs}') \xrightarrow{\text{eval}} \texttt{Bitvector}(\texttt{bv}') \qquad \texttt{res} := \texttt{bv} + \texttt{bv}'}{\mathit{concat\_bitvectors}(\texttt{vs}) \xrightarrow{\text{eval}} \texttt{Bitvector}(\texttt{res})}$$

**SemanticsRule.SlicesToPositions**

The relation

$$\mathit{slices\_to\_positions}(\overbrace{\mathbb{N}}^{\texttt{n}}, (\overbrace{\mathcal{Z}}^{\texttt{s}_i} \times \overbrace{\mathcal{Z}}^{\texttt{l}_i})^{+}) \bigtimes (\overbrace{\mathbb{N}^{*}}^{\texttt{positions}} \cup \mathsf{TDynError})$$

returns the list of positions (indices) specified by the slices `slices` and the bitvector width `n`, if all slices are within the range 0 to $\texttt{n} - 1$. Otherwise, the result is a dynamic error.

The helper predicate $\mathit{position\_in\_range}(\texttt{s}, \texttt{l}, n)$ checks whether the indices starting at index `s` and up to $\texttt{s} + \texttt{l}$, inclusive, would refer to actual indices of a bitvector of length $n$:

$$\mathit{position\_in\_range}(\texttt{s}, \texttt{l}, \texttt{n}) \triangleq (\texttt{s} \geq 0) \wedge (\texttt{l} \geq 0) \wedge (\texttt{s} + \texttt{l} < \texttt{n}) \ .$$

**Prose**

All of the following apply:

- *slices* is the list of pairs $(\texttt{s}_i, \texttt{l}_i)$, for $i = 1..k$;
- One of the following applies:
  - All of the following apply (InRange):
    - the predicate *position_in_range* holds for `n` and every $\texttt{s}_i$, $\texttt{l}_i$, for every $i = 1..k$;
    - define `positions` as the concatenation of lists starting from $\texttt{s}_i$ up to and including $\texttt{s}_i + vl_i$, for every $i = 1..k$.
  - All of the following apply (OutOfRange):
    - there exists $j \in 1..k$ such that *position_in_range* does not hold for `n` and $\texttt{s}_j$, $\texttt{l}_j$;
    - the result is a dynamic error (DE_BI).

**Formally**

InRange

$$\frac{\texttt{slices} \overset{\text{is}}{=} [i = 1..k : (\texttt{Int}(\texttt{s}_i), \texttt{Int}(\texttt{l}_i))] \qquad i = 1..k : \mathit{position\_in\_range}(\texttt{s}_i, \texttt{l}_i, \texttt{n}) \quad \texttt{positions} := [\texttt{s}_1, \ldots, \texttt{s}_1 + \texttt{l}_1] + \ldots + [\texttt{s}_k, \ldots, \texttt{s}_k + \texttt{l}_k]}{\mathit{slices\_to\_positions}(\texttt{n}, \texttt{slices}) \xrightarrow{\text{eval}} \texttt{positions}}$$

OutOfRange

$$\frac{\texttt{slices} \overset{\text{is}}{=} [i = 1..k : (\texttt{Int}(\texttt{s}_i), \texttt{Int}(\texttt{l}_i))] \qquad j \in 1..k : \neg \mathit{position\_in\_range}(\texttt{s}_j, \texttt{l}_j, \texttt{n})}{\mathit{slices\_to\_positions}(\texttt{n}, \texttt{slices}) \xrightarrow{\text{eval}} \text{DynError}(\texttt{DE\_BI})}$$

**SemanticsRule.AsBitvector**

The function

$$\mathit{as\_bitvector} : (\overbrace{\mathcal{BV} \cup \mathcal{Z}}^{\texttt{v}}) \to \overbrace{\{0,1\}^*}^{\texttt{bits}}$$

transforms a native value `v`, which either represents an integer or a bitvectors into a sequence of binary values `bits`.

**Prose**

One of the following applies:

- All of the following apply (BITS):
  - `v` is a native bitvector for the sequence of bits `bits`.
- All of the following apply (INT):
  - `v` is a native integer for the integer $n$;
  - define `bits` as the two's complement representation of $n$.

**Formally**

BITS

$$\mathit{as\_bitvector}(\overbrace{\texttt{Bitvector}(\texttt{bv})}^{\texttt{v}}) \xrightarrow{\text{eval}} \overbrace{\texttt{bv}}^{\texttt{bits}}$$

INT

$$\frac{\texttt{bits} := \text{ two's complement representation of } n}{\mathit{as\_bitvector}(\overbrace{\texttt{Int}(n)}^{\texttt{v}}) \xrightarrow{\text{eval}} \texttt{bv}}$$

**SemanticsRule.ReadFromBitvector**

The relation

$$\mathit{read\_from\_bitvector}(\overbrace{\mathcal{BV}}^{\texttt{bv}}, \overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\texttt{slices}}) \bigtimes \overbrace{\mathcal{BV}}^{\texttt{v}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

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.

**Prose**

One of the following applies:

- All of the following apply (EMPTY):
  - `slices` is the empty list;
  - define `v` as the native bitvector for the empty list of bits.
- All of the following apply (NON_EMPTY):
  - `slices` is not the empty list;
  - applying *as_bitvector* to `bv` yields the list of bits $\mathtt{b}_n \ldots \mathtt{b}_1$;
  - applying *slices_to_positions* to $n$ and `slices` yields the list of positions $j_{1..m}$
  - define `v` as the native bitvector for the list of bits from $\mathtt{b}_n \ldots \mathtt{b}_1$ indicated by the positions $j_{1..m}$, that is, $\mathtt{b}_{j_m+1} \ldots \mathtt{b}_{j_1+1}$.

**Formally**

EMPTY

$$\mathit{read\_from\_bitvector}(\mathtt{bv}, \overbrace{[\,]}^{\texttt{slices}}) \xrightarrow{\mathsf{eval}} \overbrace{\mathtt{Bitvector}([\,])}^{\texttt{v}}$$

NON_EMPTY

$$\frac{\begin{array}{c}\mathtt{slices} \neq [\,];\ \mathit{as\_bitvector}(\mathtt{bv}) := \mathtt{b}_n \ldots \mathtt{b}_1 \\ \mathit{slices\_to\_positions}(n, \mathtt{slices}) \xrightarrow{\mathsf{eval}} [j_{1..m}] \ /\!/\ \texttt{\#DE} \\ \mathtt{v} := \mathtt{Bitvector}(\mathtt{b}_{j_m+1} \ldots \mathtt{b}_{j_1+1})\end{array}}{\mathit{read\_from\_bitvector}(\mathtt{bv}, \mathtt{slices}) \xrightarrow{\mathsf{eval}} \mathtt{v}}$$

**SemanticsRule.WriteToBitvector**

The relation

$$\mathit{write\_to\_bitvector}(\overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\texttt{slices}}, \overbrace{\mathcal{BV}}^{\texttt{src}}, \overbrace{\mathcal{BV}}^{\texttt{dst}}) \bigtimes \overbrace{\mathcal{BV}}^{\texttt{v}} \cup \overbrace{\mathsf{TDynError}}^{\#\mathsf{DE}}$$

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:

- applying *as_bitvector* to `src` yields the list of bits $\texttt{s}_m \ldots \texttt{s}_0$;
- applying *as_bitvector* to `dst` yields the list of bits $\texttt{d}_n \ldots \texttt{d}_0$;
- applying *slices_to_positions* to $n$ and `slices` yields the list of indices `positions`;
- view `positions` as the list $I_m \ldots I_0$;
- define the function *bit* as mapping an index $i$ in 0 to $n$ to $\texttt{s}_j$, if there exists an index $I_j$ in `positions` such that $I_j$ is equal to $i$, and $\texttt{d}_i$, otherwise.
- define `bits` as the list of bits defined as $\mathit{bit}(n) \ldots \mathit{bit}(0)$;
- define `v` as the native bitvector for `bits`.

**Formally**

$$\frac{\begin{array}{c}\texttt{s}_m \ldots \texttt{s}_0 := \mathit{as\_bitvector}(\texttt{src}) \qquad \texttt{d}_n \ldots \texttt{d}_0 := \mathit{as\_bitvector}(\texttt{dst}) \\ \mathit{slices\_to\_positions}(n, \texttt{slices}) \xrightarrow{\mathsf{eval}} \texttt{positions} \;/\!/\; \#\mathsf{DE} \\ \texttt{positions} \overset{\mathsf{is}}{=} I_m \ldots I_0 \qquad \mathit{bit} = \lambda i \in 0..n. \begin{cases} \texttt{s}_j & \exists j \in 1..m.\ i = I_j \\ \texttt{d}_i & \text{otherwise} \end{cases} \\ \texttt{bits} := [i = n..0 : \mathit{bit}(i)]\end{array}}{\mathit{write\_to\_bitvector}(\texttt{slices}, \texttt{src}, \texttt{dst}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Bitvector}(\texttt{bits})}^{\texttt{v}}}$$

**Example: Writing to a Bitvector**

In reference to Listing 18.13, we have the following application of the current rule:

$$\mathit{as\_bitvector}(\texttt{Bitvector}(000000)) = \overbrace{0}^{\texttt{s}_5}\ \overbrace{0}^{\texttt{s}_4}\ \overbrace{0}^{\texttt{s}_3}\ \overbrace{0}^{\texttt{s}_2}\ \overbrace{0}^{\texttt{s}_1}\ \overbrace{0}^{\texttt{s}_0}$$

$$\mathit{as\_bitvector}(\texttt{Bitvector}(11111111)) = \overbrace{1}^{\texttt{d}_7}\ \overbrace{1}^{\texttt{d}_6}\ \overbrace{1}^{\texttt{d}_5}\ \overbrace{1}^{\texttt{d}_4}\ \overbrace{1}^{\texttt{d}_3}\ \overbrace{1}^{\texttt{d}_2}\ \overbrace{1}^{\texttt{d}_1}\ \overbrace{1}^{\texttt{d}_0}$$

$$\mathit{slices\_to\_positions}(8, [\overbrace{(0,4)}^{\texttt{3:0}}, \overbrace{(6,2)}^{\texttt{7:6}}]) \xrightarrow{\text{eval}} [3,2,1,0,7,6]$$

$$\texttt{positions} := [\ \overbrace{3}^{I_5}, \overbrace{2}^{I_4}, \overbrace{1}^{I_3}, \overbrace{0}^{I_2}, \overbrace{7}^{I_1}, \overbrace{6}^{I_0}\ ]$$

$$\mathit{bit} = \lambda i \in 0..7. \begin{cases} \texttt{s}_j & \exists j \in 1..5.\ i = I_j \\ \texttt{d}_i & \text{otherwise} \end{cases}$$

$$\texttt{bits} := \mathit{bit}(7)\ \mathit{bit}(6)\ \mathit{bit}(5)\ \mathit{bit}(4)\ \mathit{bit}(3)\ \mathit{bit}(2)\ \mathit{bit}(1)\ \mathit{bit}(0)$$

---

$$\mathit{write\_to\_bitvector}([\overbrace{(0,4)}^{\texttt{3:0}}, \overbrace{(6,2)}^{\texttt{7:6}}], \texttt{Bitvector}(000000), \texttt{Bitvector}(11111111)) \xrightarrow{\text{eval}}$$

$$\texttt{Bitvector}(\overbrace{0}^{\texttt{s}_1}\ \overbrace{0}^{\texttt{s}_0}\ \overbrace{1}^{\texttt{d}_5}\ \overbrace{1}^{\texttt{d}_4}\ \overbrace{0}^{\texttt{s}_5}\ \overbrace{0}^{\texttt{s}_4}\ \overbrace{0}^{\texttt{s}_3}\ \overbrace{0}^{\texttt{s}_2})$$

**SemanticsRule.GetIndex**

**Prose**

The relation

$$\mathit{get\_index}(\overbrace{\mathbb{N}}^{\texttt{i}}, \overbrace{\mathcal{VEC}}^{\texttt{vec}}) \times \overbrace{\mathcal{VEC}}^{\texttt{v}_\texttt{i}}$$

reads the value $\texttt{v}_i$ from the vector of values `vec` at the index `i`.

**Formally**

$$\frac{\texttt{vec} \overset{\text{is}}{=} \texttt{v}_{0..k} \qquad \texttt{i} \leq k}{\mathit{get\_index}(\texttt{i}, \texttt{vec}) \xrightarrow{\text{eval}} \texttt{v}_\texttt{i}}$$

Notice that there is no rule to handle the case where the index is out of range — this is guaranteed by the typechecker not to happen. Specifically,

- TypingRule.EGetArray ensures that an index is within the bounds of the array being accessed via a check that the type of the index satisfies the type of the array size.
- Typing rules TypingRule.LEDestructuring, TypingRule.PTuple, and TypingRule.LDTuple use the same index sequences for the tuples involved and the corresponding lists of expressions.

If the rules listed above do not hold the typechecker fails.

### SemanticsRule.SetIndex

#### Prose

The relation

$$\mathit{set\_index}(\overbrace{\mathbb{N}}^{\texttt{i}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathcal{VEC}}^{\texttt{vec}}) \bigtimes \overbrace{\mathcal{VEC}}^{\texttt{res}}$$

overwrites the value at the given index `i` in a vector of values `vec` with the new value `v`.

#### Formally

$$\frac{\texttt{vec} \overset{\text{is}}{=} \texttt{u}_{0..k} \qquad \texttt{i} \leq k \qquad \texttt{res} \overset{\text{is}}{=} \texttt{w}_{0..k} \qquad \texttt{v} := \texttt{w}_{\texttt{i}} \qquad j \in \{0..k\} \setminus \{\texttt{i}\}.\ \texttt{w}_j = \texttt{u}_j}{\mathit{set\_index}(\texttt{i}, \texttt{v}, \texttt{vec}) \xrightarrow{\text{eval}} \texttt{res}}$$

Similar to *get_index*, there is no need to handle the out-of-range index case.

### SemanticsRule.GetField

#### Prose

The relation

$$\mathit{get\_field}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathcal{REC}}^{\texttt{record}}) \bigtimes \mathbb{V}$$

retrieves the value corresponding to the field name `name` from the record value `record`.

#### Formally

$$\frac{\texttt{record} \overset{\text{is}}{=} \texttt{NV\_Record}(\texttt{field\_map})}{\mathit{get\_field}(\texttt{name}, \texttt{record}) \xrightarrow{\text{eval}} \texttt{field\_map}(\texttt{name})}$$

The typechecker ensures, via TypingRule.EGetRecordField, that the field `name` exists in `record`.

### SemanticsRule.SetField

#### Prose

The function

$$\mathit{set\_field}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathcal{REC}}^{\texttt{record}}) \longrightarrow \mathcal{REC}$$

overwrites the value corresponding to the field name `name` in the record value `record` with the value `v`.

#### Formally

$$\frac{\texttt{record} \overset{\text{is}}{=} \texttt{NV\_Record}(\texttt{field\_map}) \qquad \texttt{field\_map'} := \texttt{field\_map}[\texttt{name} \mapsto \texttt{v}]}{\mathit{set\_field}(\texttt{name}, \texttt{v}, \texttt{record}) \xrightarrow{\text{eval}} \texttt{NV\_Record}(\texttt{field\_map'})}$$

The typechecker ensures that the field `name` exists in `record`.

### SemanticsRule.DeclareLocalIdentifier

#### Prose

The relation

$$\mathit{declare\_local\_identifier}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{g}})$$

associates `v` to `name` as a local storage element in the environment `env` and returns the updated environment `new_env` with the execution graph consisting of a Write Effect to `name`.

#### Formally

$$\frac{\texttt{g} := \mathsf{WriteEffect}(\texttt{name}) \qquad \texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}[\texttt{name} \mapsto \texttt{v}]))}{\mathit{declare\_local\_identifier}(\texttt{env}, \texttt{name}, \texttt{v}) \xrightarrow{\mathsf{eval}} (\texttt{new\_env}, \texttt{g})}$$

### SemanticsRule.DeclareLocalIdentifierM

#### Prose

The relation

$$\mathit{declare\_local\_identifier\_m}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{x}}, \overbrace{\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}}^{\texttt{m}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

declares the local identifier `x` in the environment `env`, in the context of the value-graph pair `(v, g)`, yielding a pair consisting of the environment `new_env` and execution graph `new_g`.

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

$$\frac{\texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g}) \qquad \mathit{declare\_local\_identifier}(\texttt{env}, \texttt{x}, \texttt{v}) \xrightarrow{\mathsf{eval}} (\texttt{new\_env}, \texttt{g1}) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \texttt{g1}}{\mathit{declare\_local\_identifier\_m}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\mathsf{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

**SemanticsRule.DeclareLocalIdentifierMM**

**Prose**

The relation

$$\mathit{declare\_local\_identifier\_mm}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{x}}, \overbrace{(\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}})}^{\texttt{m}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

declares the local identifier `x` in the environment `env`, in the context of the value-graph pair $(\texttt{v}, \texttt{g})$, yielding a pair consisting of an environment `new_env` and an execution graph `g2`.

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**

$$\frac{\mathit{declare\_local\_identifier\_m}(\texttt{env}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g1}) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_po}} \texttt{g1}}{\mathit{declare\_local\_identifier\_mm}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

# 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 20.45 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 20.23 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 20.46 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 22:
       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.EGetTupleItem.

`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 `Unreachable()` (TypingRule.CheckStmtReturnsOrThrows).

`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/ASLRefALP3.1/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.

Note that `print(...)` and `println(...)` are print statements, and `Unreachable()` is an unreachable statement. They are not standard subprograms.

# Bibliography


[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat, 2016.

[3] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Transactions on Programming Languages and Systems*, 43(2):8:1–8:54, 2021.

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2):7:1–7:74, 2014.

[5] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[6] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., USA, 1992.

[7] François Pottie and Yann Régis-Gianas. *Menhir Reference Manual*.

# Appendix A

# Not Implemented by ASLRef

This chapter describes what is not yet present in the executable version of ASLRef (Build from Dec 12, 2024).

## A.1 Syntax

### A.1.1 Declaring Multiple Identifiers Without Initialization

The following simultaneous declaration of three global variables does not currently parse with ASLRef.

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

The same line does parse and correctly handled inside a subprogram.

### A.1.2 Guards

Guards are used on `case` and `catch` statements, to restrict matching on the evaluation of a boolean expression. They are not yet implemented in ASLRef.

## A.2 Semantics

### A.2.1 Non-`main` Entry Point

Currently ASLRef only supports `main` as an entry point.

## A.3 Typing

### A.3.1 Throwing Exceptions without Braces

In the following example, the commented out `throw` statement should typecheck, but it currently fails.

```
type except of exception;

func main() => integer
begin
  // throw except; // Should typecheck
  throw except{}; // Okay

  return 0;
end
```

### A.3.2 Side-effect-free Subprograms with respect to dynamic errors

ASLRef performs a side effect analysis (see Chapter 30). The analysis currently ignores dynamic errors that are not due to assertions.

### A.3.3 Restriction on Use of Parameterized Integer Types

#### As storage types

Restrictions on the use of parameterized integer types as storage element types are not implemented.

#### `as` Expression With a Constrained Type

Restriction on the use of parameterized integer types as left-hand-side of an Asserted Typed Conversion is not implemented in ASLRef. For example, the following will not raise a type error:

```
func foo {N} (x: bits(N)) => integer {0..2*N}
begin
  return N as integer {0..2*N};
end;
```

# Appendix B

# Issues Not Yet Addressed by the Reference

## B.1 Semantics

## B.2 Typing

### B.2.1 Checking Type Annotations for Absence of Side Effects

Type annotations that contain expressions that may fail dynamically are not checked for.