ASL Typing Reference
DDI 0622

Arm Architecture Technology Group

January 23, 2024

Contents

Chapter 1

Non-Confidential Proprietary Notice

Chapter 2

Disclaimer

This document is part of the ASLRef material. It is a snapshot of: `https://github.com/herd/herdtools7/commit/6dd15fe7833fea24eb94933486d0858038f0c2e8`

This material covers both ASLv0 (viz, the existing ASL pseudocode language which appears in the Arm Architecture Reference Manual) and ASLv1, a new, experimental, and as yet unreleased version of ASL.

The development version of ASLRef can be found here `~/herdtools7/asllib`.

A list of open items being worked on can be found here `~/herdtools7/asllib/doc/ASLRefProgress.tex`.

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

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

Chapter 3

Preamble

3.1 Abstract Syntax

An *abstract syntax tree* (AST, for short) represents an ASL program as a labelled tree. The ASL Abstract Syntax is given in [1].

3.2 Environments

An environment is what the typing operates over: a structure which amongst other things associates types to variables. Intuitively, the typing of a program makes an initial environment evolve, with new types as given by the variable declarations of the program. Environments map names to variables and subprograms.

More precisely, a *typing environment* (environment, for short) maps AST nodes to types.

A *type judgement* associates a node in an abstract syntax tree (which intuitively corresponds to an ASL construct) with a certain type. We then say that the node is *annotated* with that type.

Defining the type annotation requires inferring various auxiliary attributes for AST nodes. We consider these auxiliary attributes as part of the environment as well. Technically, we partition the environment into two distinct components: the global environment G—pertaining to AST nodes appearing outside of a given subprogram, and the local environment L—pertaining to AST nodes appearing inside a given subprogram.

3.3 Type System

A *typing rule* specifies a conjunction of conditions that must hold in order to annotate an AST node with a given type. These conditions typically inspect the label of node and the type annotations on the children of the node (in a given environment).

Not all conditions are type judgements. Some conditions are expressed in terms of the auxiliary attributes mentioned above. We define these in the next chapter.

A *typing system* is a set of typing rules. More than one rule can be associated with a given node label, but they are exclusive—at most one rule holds in a given environment.

This is related to $\mathtt{D}_{\mathtt{JRXM}}$, $\mathtt{I}_{\mathtt{ZTMQ}}$, $\mathtt{D}_{\mathtt{HBCP}}$, $\mathtt{I}_{\mathtt{SMMH}}$, $\mathtt{I}_{\mathtt{DFML}}$, $\mathtt{R}_{\mathtt{WMFV}}$.

3.4 Annotation

Typing a program consists of annotating the root of its AST. This is typically done by traversing the AST bottom-up. To annotate a node, the typing algorithm finds a rule that matches the node—that is a rule whose conditions are satisfied. If one such rule is found, the node is annotated by the result type specified by the rule, essentially adding the type judgement to the environment. If no such rule is found, it is considered a *typing error* and the algorithm exits.

Sometimes it is to necessary to define *error rules*—rules that result in an error and provide extra information to help understand the reason for the error.

We implement the process described above via a set of `annotate_<label>` functions. Each `annotate_<label>` function describes how to annotate an AST node, given its label, as follows:

- `annotate_expr` annotates expressions;
- `annotate_slices` annotates slices;
- `annotate_pattern` annotates pattern;
- `annotate_local_decl_item` annotates local declarations;
- `annotate_lexpr` annotates left-hand sides of assignments;
- `annotate_stmt` annotates statements;
- `annotate_block` annotates blocks;
- `annotate_catcher` annotates catchers;
- `annotate_call` annotates functions calls;
- `annotate_func` annotates functions.

This is related to $\mathtt{R}_{\mathtt{VDPC}}$.

Chapter 4

Reading guide

The definition of each `annotation_<label>` function is given by a number of rules, which follow the possible shapes the `label` can have. For example, an expression can be a literal, or a binary operator, amongst other things. Each of those has its own evaluation rule: TypingRule.Lit in Section 7.1, Typing.Binop in Section 7.7 respectively.

Each rule is presented using the following template:

- a Prose paragraph gives the rule in English, and corresponds as much as possible to the code of the reference implementation ASLRef given at `~/herdtools7/asllib`;
- one or several Example, which as much as possible are also given as regression tests in `~/herdtools7/asllib/tests/ASLTypingReference.t`
- a Code paragraph which gives a verbatim of the corresponding implementation in the type-checker of ASLRef `~/herdtools7/asllib/Typing.ml`;
- Formal paragraphs which give formal definitions of the rule.

Chapter 5

Type Algebra

5.1 TypingRule.BuiltinSingularType

5.1.1 Prose

The *builtin singular types* are:

- integer;
- real;
- string;
- boolean;
- bits;
- enumeration.

5.1.2 Example: TypingRule.BuiltinSingularTypes.asl

```
let i : integer = 0;
let r : real = 0.0;
let s : string = "0.0";
let b : boolean = TRUE;
let z4 : bits(4) = '0000';
let o2 : bits(2) = '11';
```

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

5.1.3 Example: TypingRule.EnumerationType.asl

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

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

The type `color` consists in two different constants `RED` and `BLACK`.

5.1.4 Code

```
let is_builtin_singular ty =
  match ty.desc with
  | T_Real | T_String | T_Bool | T_Bits _ | T_Enum _ | T_Int _ -> true
  | _ -> false |: TypingRule.BuiltinSingularType
```

5.1.5 Formally

$$
\begin{array}{rl}
 & \texttt{is\_builtin\_singular}(\texttt{T\_Real}) \\
 & \texttt{is\_builtin\_singular}(\texttt{T\_String}) \\
 & \texttt{is\_builtin\_singular}(\texttt{T\_Bool}) \\
fl \in \textsf{bit\_field}^* \vdash & \texttt{is\_builtin\_singular}(\texttt{T\_Bits}(fl)) \\
il \in \texttt{<identifier>}^* \vdash & \texttt{is\_builtin\_singular}(\texttt{T\_Enum}(il)) \\
c \in \textsf{int\_constraints}? \vdash & \texttt{is\_builtin\_singular}(\texttt{T\_Int}(c))
\end{array}
$$

5.1.6 Comments

This is related to $\texttt{D}_{\texttt{PQCK}}$ and $\texttt{D}_{\texttt{NZWT}}$.

5.2 TypingRule.BuiltinAggregateType

5.2.1 Prose

The builtin aggregate types are:

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

5.2.2 Example: TypingRule.BuiltinAggregateTypes.asl

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

type T of array [3] of real;
type coord of enumeration { X, Y, Z };
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;
  assert (t1[0] == t2[X]);

  let o = pointRecord { x=0.0, y=0.0, z=0.0 };
  assert (t2[Z] == o.z);

  return 0;
end
```

Type `pair` is the type of integer and booleans pairs. Notice that the syntax of types and expressions are similar.

Arrays are indexed either by integers from 0 to (array size minus 1) as specified in type declaration, as illustrated by the type `T`, or by the elements of an enumeration type, as illustrated by type `pointCoord`.

The type `pointRecord` is defined as a record type with three fields `x`, `y` and `z`.

5.2.3 Example: TypingRule.BuiltinExceptionType.asl

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

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

  return 0;
end
```

Two exception types are defined: exceptions `Not_found` carry no values, while exceptions `Error` carry a messsage. Notice the similarity with record types and that the empty field list {} can be omitted in type declarations, as it is the case for `Not_found`.

5.2.4 Code

```
let is_builtin_aggregate ty =
  match ty.desc with
  | T_Tuple _ | T_Array _ | T_Record _ | T_Exception _ -> true
  | _ -> false |: TypingRule.BuiltinAggregateType
```

5.2.5 Formally

$$
\begin{array}{rl}
tl \in \mathsf{ty}^* \vdash & \texttt{is\_builtin\_aggregate}(\texttt{T\_Tuple}(tl)) \\
t \in \mathsf{ty} \vdash & \texttt{is\_builtin\_aggregate}(\texttt{T\_Array}(t)) \\
fl \in \mathsf{field}^* \vdash & \texttt{is\_builtin\_aggregate}(\texttt{T\_Record}(fl)) \\
fl \in \mathsf{field}^* \vdash & \texttt{is\_builtin\_aggregate}(\texttt{T\_Exception}(fl))
\end{array}
$$

5.2.6 Comments

This is related to $\texttt{D}_{\texttt{PQCK}}$ and $\texttt{D}_{\texttt{KNBD}}$.

5.3 TypingRule.BuiltinSingularOrAggregate

5.3.1 Prose

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

- `t` is singular;
- `t` is aggregate.

5.3.2 Example

5.3.3 Code

```
let is_builtin ty =
  is_builtin_singular ty
  || is_builtin_aggregate ty |: TypingRule.BuiltinSingularOrAggregate
```

5.3.4 Formally

$$
\begin{array}{rl}
t \in \mathsf{ty}\texttt{is\_builtin\_singular}(t) \vdash & \texttt{is\_builtin}(t) \\
t \in \mathsf{ty}\texttt{is\_builtin\_aggregate}(t) \vdash & \texttt{is\_builtin}(t)
\end{array}
$$

5.3.5 Comments

5.4 TypingRule.NamedType

5.4.1 Prose

A named type is a type which is declared using the `type` syntax.

5.4.2 Example

5.4.3 Code

```
let is_named ty =
  match ty.desc with T_Named _ -> true | _ -> false |: TypingRule.NamedType
```

5.4.4 Formally

$$i \in \texttt{<identifier>} \vdash \quad \texttt{is\_named}(\texttt{T\_Named}(i))$$

5.4.5 Comments

This is related to $\texttt{D}_{\texttt{VMZX}}$.

5.5 TypingRule.AnonymousType

5.5.1 Prose

An anonymous type is a type which is not declared using the type syntax.

5.5.2 Example

5.5.3 Code

```
let is_anonymous ty = (not (is_named ty)) |: TypingRule.AnonymousType
```

5.5.4 Formally

$$t \in \texttt{ty} \;\; \neg\texttt{is\_named}(t) \vdash \quad \texttt{is\_anonymous}(t)$$

5.5.5 Comments

This is related to $\texttt{D}_{\texttt{VMZX}}$.

5.6 TypingRule.SingularType

5.6.1 Prose

A type `t` is singular if one of the following applies:

- `t` is a builtin singular type;
- All of the following apply:
  - `t` is a named type;
  - `t_struct` is the structure of `t`;
  - `t_struct` is a builtin singular.

5.6.2 Example

5.6.3 Code

```
let is_singular env ty =
  is_builtin_singular ty
  || (is_named ty && get_structure env ty |> is_builtin_singular)
     |: TypingRule.SingularType
```

5.6.4 Formally

$$t \in \texttt{ty} \;\; \texttt{is\_builtin}(t) \vdash \;\; \texttt{is\_singular}(t)$$
$$t \in \texttt{ty} \;\; \texttt{is\_named}(t) \;\; \texttt{is\_builtin\_aggregate}(\texttt{t\_struct}(t)) \vdash \;\; \texttt{is\_singular}(t)$$

5.6.5 Comments

This is related to $\texttt{R}_{\texttt{GVZK}}$.

5.7 TypingRule.AggregateType

5.7.1 Prose

A type `t` is aggregate if one of the following applies:

- `t` is a builtin aggregate type;
- All of the following apply:
  - `t` is a named type;
  - `t_struct` is the structure of `t`;
  - `t_struct` is a builtin aggregate.

5.7.2 Example

5.7.3 Code

```
let is_aggregate env ty =
  is_builtin_aggregate ty
  || (is_named ty && get_structure env ty |> is_builtin_aggregate)
     |: TypingRule.AggregateType
```

5.7.4 Formally

$$t \in \texttt{ty} \;\; \texttt{is\_builtin\_aggregate}(t) \vdash \;\; \texttt{is\_aggregate}(t)$$
$$t \in \texttt{ty} \;\; \texttt{is\_named}(t) \;\; \texttt{is\_builtin\_aggregate}(\texttt{t\_struct}(t)) \vdash \;\; \texttt{is\_aggregate}(t)$$

5.7.5 Comments

This is related to $\texttt{R}_{\texttt{GVZK}}$.

5.8 TypingRule.NonPrimitiveType

5.8.1 Prose

A type `t` is non-primitive if one of the following applies:

- `t` is a named type;
- All of the following apply:
  - `t` is a tuple `li`;
  - there exists a non-primitive type in `li`;
- All of the following apply:
  - `t` is an array of type `ty`
  - `ty` is non-primitive;
- All of the following apply:
  - `t` is a record with fields `fields`;
  - there exists a non-primitive type in `fields`;
- All of the following apply:
  - `t` is an exception with fields `fields`;
  - there exists a non-primitive type in `fields`;

5.8.2 Example

5.8.3 Code

```
let rec is_non_primitive ty =
  match ty.desc with
  | T_Real | T_String | T_Bool | T_Bits _ | T_Enum _ | T_Int _ -> false
  | T_Named _ -> true
  | T_Tuple li -> List.exists is_non_primitive li
  | T_Array (_, ty) -> is_non_primitive ty
  | T_Record fields | T_Exception fields ->
      List.exists (fun (_, ty) -> is_non_primitive ty) fields
      |: TypingRule.NonPrimitiveType
```

5.8.4 Formally

5.8.5 Comments

This is related to $\mathtt{D}_{\mathtt{GWXK}}$.

5.9 TypingRule.PrimitiveType

5.9.1 Prose

A type `t` is primitive if it is not non-primitive.

5.9.2 Example

5.9.3 Code

```
let is_primitive ty = (not (is_non_primitive ty)) |: TypingRule.PrimitiveType
```

5.9.4 Formally

5.9.5 Comments

This is related to $\mathtt{D}_{\mathtt{GWXK}}$.

5.10 TypingRule.Structure

5.10.1 Prose

`ty` is a type and its structure is `t_struct` and one of the following applies:

- All of the following apply:
  - ∗ `ty` is a named type `x`;
  - ∗ One of the following applies:

  - All of the following apply:
    - ▷ `x` is not declared in the global environment;
    - ▷ an error "`Undefined Identifier`" is raised;
  - All of the following apply:
    - ▷ `x` is declared in the global environment as some type `ty'`;
    - ▷ `t_struct` is the structure of `ty'`;

- All of the following apply:
  - ∗ `t` is a builtin singular type;
  - ∗ `t_struct` is `ty`;
- All of the following apply:
  - ∗ `ty` is a tuple type with `tys`;
  - ∗ `t_struct` is a tuple type with the structure of each element in `tys`;
- All of the following apply:
  - ∗ `ty` is an array type of length `e` with element type `t`;
  - ∗ `t_struct` is an array type with of length `e` with element type the structure of `t`;
- All of the following apply:
  - ∗ `ty` is a record type with fields `fields`;
  - ∗ `fields` associates a name x to a type t_x;
  - ∗ `fields'` associates to each name x of `fields` to the structure of t_x;
  - ∗ `t_struct` is a record type with fields `fields'`.
- All of the following apply:
  - ∗ `ty` is an exception type with fields `fields`;
  - ∗ `fields` associates a name x to a type t_x;
  - ∗ `fields'` associates to each name x of `fields` to the structure of t_x;
  - ∗ `t_struct` is an exception type with fields `fields'`.

5.10.2 Example

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

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

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

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

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

5.10.3 Code

```
let rec get_structure (env : env) (ty : ty) : ty =
  let () =
    if false then Format.eprintf "@[Getting structure of %a.@]@." PP.pp_ty ty
  in
  let with_pos = add_pos_from ty in
  (match ty.desc with
  | T_Named x -> (
      match IMap.find_opt x env.global.declared_types with
      | None -> undefined_identifier ty x
      | Some ty' -> get_structure env ty')
  | T_Int _ | T_Real | T_String | T_Bool | T_Bits _ | T_Enum _ -> ty
  | T_Tuple tys -> T_Tuple (List.map (get_structure env) tys) |> with_pos
  | T_Array (e, t) -> T_Array (e, (get_structure env) t) |> with_pos
  | T_Record fields ->
      let fields' = assoc_map (get_structure env) fields |> canonical_fields in
      T_Record fields' |> with_pos
  | T_Exception fields ->
      let fields' = assoc_map (get_structure env) fields |> canonical_fields in
      T_Exception fields' |> with_pos)
  |: TypingRule.Structure
```

5.10.4 Formally

5.10.5 Comments

The structure of a type is the primitive type it is equivalent to such that it can hold the same values.

This is related to $\texttt{D}_{\texttt{FXQV}}$.

5.11 TypingRule.Domain

5.11.1 Prose

The domain of a type is the set of values which storage elements of that type may hold.

5.11.2 Example

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

The domain of `bits(1)` is the set '1', '0'.
The domain of `integer 2,16` is the set containing the integers 2 and 16.

5.11.3 Code

```
type t =
  | D_Bool
  | D_String
  | D_Real  (** The domain of an enum is a set of symbols *)
  | D_Symbols of ISet.t
  | D_Int of int_set  (** The domain of a bitvector is given by its width. *)
  | D_Bits of int_set
(* |: TypingRule.Domain *)
```

5.11.4 Formally

5.11.5 Comments

This is related to D_{BMGM}, R_{PHRL}, R_{PZNR}, R_{RLQP}, R_{LYDS}, R_{SVDJ}, I_{WLPJ}, R_{FWMM}, I_{WPWL}, I_{CDVY}, I_{KFCR}, I_{BBQR}, R_{ZWGH}, R_{DKGQ}, R_{DHZT}, I_{HSWR}.

5.12 Constrained Types

5.12.1 Prose

- A constrained type is a type whose definition depends on an expression, e.g. certain integers and bitvectors.

- A type which is not constrained is unconstrained.

- A constrained type with a non-empty constraint is well-constrained.

- An under-constrained integer type is an implicit type of a subprogram parameter.

5.12.2 Example

Bitvector storage element's widths are constrained integers.

5.12.3 Code

```
| T_Int of int_constraints option
| T_Bits of expr * bitfield list
```

5.12.4 Formally

5.12.5 Comments

This is related to $\mathtt{D}_{\mathtt{ZTPP}}$, $\mathtt{R}_{\mathtt{WJYH}}$, $\mathtt{R}_{\mathtt{HJPN}}$, $\mathtt{R}_{\mathtt{CZTX}}$, $\mathtt{R}_{\mathtt{TPHR}}$

Chapter 6

Type Satisfaction and Related Notions

6.1 TypingRule.Subtype

6.1.1 Prose

A type `t1` subtypes a type `t2` in the environment `env` if and only if one of the following applies:

- All of the following apply:
  * `t1` is a named type;
  * `t2` is a named type;
  * `t1` and `t2` are the same type.
- All of the following apply:
  * `t1` is a named type;
  * `t2` is a named type;
  * `t1` is declared as a subtype of `t1'` in `env`;
  * `t1'` is a subtype of `t2` in `env`.

The subtype relation is a partial order.

6.1.2 Example

6.1.3 Code

```
let rec subtypes_names env s1 s2 =
  if String.equal s1 s2 then true
  else
```

```
    match IMap.find_opt s1 env.SEnv.global.subtypes with
    | None -> false
    | Some s1' -> subtypes_names env s1' s2

let subtypes env t1 t2 =
  (match (t1.desc, t2.desc) with
  | T_Named s1, T_Named s2 -> subtypes_names env s1 s2
  | _ -> false)
  |: TypingRule.Subtype
```

6.1.4 Formally

To model subtyping, we define the function

$$super : \texttt{<identifier>} \rightarrow \texttt{<identifier>} \cup \perp \ ,$$

which maps an identifier to the name of its super-type or $\perp$ if it does not have one.

We define the subtyping relation between two named types in a given environment as follows:

$$\begin{array}{rl} a, b \in \texttt{<identifier>} & (G, L) \vdash \texttt{T\_Named(a)} \sqsubseteq \texttt{T\_Named(a)} \\ a, b \in \texttt{<identifier>} \quad G.super(a) = b & (G, L) \vdash \texttt{T\_Named(a)} \sqsubseteq \texttt{T\_Named(b)} \end{array}$$

6.1.5 Comments

Since the subtype relation is a partial order, it is reflexive, viz, every type is also a subtype of itself.

Since the subtype relation is a partial order, it is transitive, viz, if A is a subtype of B and B is a subtype of C then A is a subtype of C.

As a consequence, there is no need to declare the reflexive and transitive subtype relations explicitly. All other subtype relations must be explicitly declared.

Since the subtype relation is a partial order, it is antisymmetric. Therefore it is an error if all of the following apply:

- `id1` is a subtype of `id2`;
- `id2` is a subtype of `id1`.

This is related to $\texttt{R}_{\texttt{NXRX}}$, $\texttt{I}_{\texttt{KGKS}}$, $\texttt{I}_{\texttt{MTML}}$, $\texttt{I}_{\texttt{JVRM}}$, $\texttt{I}_{\texttt{CHMP}}$.

6.2 TypingRule.Supertype

6.2.1 Prose

`T` is a supertype of `S` if and only if `S` is a subtype of `T`.

6.2.2 Example

6.2.3 Code

6.2.4 Formally

6.2.5 Comments

Since the subtype relation is a partial order, it is reflexive. Therefore the supertype relation also is reflexive, viz, every type is also a supertype of itself.

This is related to $\texttt{I}_{\texttt{KGKS}}$.

6.3 TypingRule.StructuralSubtypeSatisfaction

6.3.1 Prose

`T` structural-subtype-satisfies `S` if one of the following applies:

- All of the following apply:
  - `S` has the structure of an integer type;
  - `T` has the structure of an integer type.
- All of the following apply:
  - `S` has the structure of a real type;
  - `T` has the structure of a real type.
- All of the following apply:
  - `S` has the structure of a string type;
  - `T` has the structure of a string type.
- All of the following apply:
  - `S` has the structure of a boolean type;
  - `T` has the structure of a boolean type.
- All of the following apply:
  - `S` has the structure of an enumeration type;
  - `T` has the structure of an enumeration type;
  - `S` and `T` have the same enumeration literals.
- All of the following apply:
  - `S` has the structure of a bitvector type with determined width `w`;
  - One of the following applies:
    - `T` has the structure of a bitvector type of determined width `w`;

– `T` has the structure of a bitvector type of undetermined width.

- All of the following apply:
  - ∗ `S` has the structure of a bitvector type with undetermined width;
  - ∗ `T` has the structure of a bitvector type.
- All of the following apply:
  - ∗ `S` has the structure of a bitvector type with bitfields `bitfields` and width `width`;
  - ∗ `T` has the structure of a bitvector type with width `width`;
  - ∗ for every bitfield `f` in `bitfields` there is a bitfield `f'` in `T` and all of the following apply:
    - – `f'` has the same name, width and offset as `f`;
    - – `f'` type-satisfies `f`.
- All of the following apply:
  - ∗ `S` has the structure of an array type with elements of type `E`;
  - ∗ `T` has the structure of an array type with elements of type `E`;
  - ∗ `T` has the same element indices as `S`.
- All of the following apply:
  - ∗ `S` has the structure of a tuple type;
  - ∗ `T` has the structure of a tuple type;
  - ∗ `T` has the same number of elements as `S`;
  - ∗ for each element `e` in `S` there is an element `e'` in `T` and `e'` type-satisfies `e`.
- All of the following apply:
  - ∗ `S` has the structure of a record type;
  - ∗ `T` has the structure of a record type;
  - ∗ for each field `f` in `S` there is an element `f'` in `T` and `f'` has the same type as `f`.
- All of the following apply:
  - ∗ `S` has the structure of an exception type;
  - ∗ `T` has the structure of an exception type;
  - ∗ for each field `f` in `S` there is an element `f'` in `T` and `f'` has the same type as `f`.

6.3.2 Example

6.3.3 Code

```
and structural_subtype_satisfies env t s =
  (* A type T subtype-satisfies type S if and only if all of the following
     conditions hold: *)
  match ((make_anonymous env s).desc, (make_anonymous env t).desc) with
  (* If S has the structure of an integer type then T must have the structure
     of an integer type. *)
  | T_Int _, T_Int _ -> true
  | T_Int _, _ -> false
  (* If S has the structure of a real type then T must have the structure of a
     real type. *)
  | T_Real, T_Real -> true
  | T_Real, _ -> false
  (* If S has the structure of a string type then T must have the structure of
     a string type. *)
  | T_String, T_String -> true
  | T_String, _ -> false
  (* If S has the structure of a boolean type then T must have the structure of
     a boolean type. *)
  | T_Bool, T_Bool -> true
  | T_Bool, _ -> false
  (* If S has the structure of an enumeration type then T must have the
     structure of an enumeration type with exactly the same enumeration
     literals. *)
  | T_Enum li_s, T_Enum li_t -> list_equal String.equal li_s li_t
  | T_Enum _, _ -> false
  (*
    • If S has the structure of a bitvector type with determined width then
      either T must have the structure of a bitvector type of the same
      determined width or T must have the structure of a bitvector type with
      undetermined width.
    • If S has the structure of a bitvector type with undetermined width then T
      must have the structure of a bitvector type.
    • If S has the structure of a bitvector type which has bitfields then T
      must have the structure of a bitvector type of the same width and for
      every bitfield in S there must be a bitfield in T of the same name, width
      and offset, whose type type-satisfies the bitfield in S.
  *)
  | T_Bits (w_s, bf_s), T_Bits (w_t, bf_t) -> (
      (* Interpreting the first two condition as just a condition on domains. *)
      match (bf_s, bf_t) with
      | [], _ -> true
      | _, [] -> false
```

```
    | bfs_s, bfs_t ->
        bitwidth_equal env w_s w_t && bitfields_included env bfs_s bfs_t)
| T_Bits _, _ -> false
(* If S has the structure of an array type with elements of type E then T
   must have the structure of an array type with elements of type E, and T
   must have the same element indices as S. *)
| T_Array (length_s, ty_s), T_Array (length_t, ty_t) ->
    expr_equal env length_s length_t && type_equal env ty_s ty_t
| T_Array _, _ -> false
(* If S has the structure of a tuple type then T must have the structure of
   a tuple type with same number of elements as S, and each element in T
   must type-satisfy the corresponding element in S.*)
| T_Tuple li_s, T_Tuple li_t ->
    List.compare_lengths li_s li_t = 0
    && List.for_all2 (type_satisfies env) li_t li_s
| T_Tuple _, _ -> false
(* If S has the structure of an exception type then T must have the structure
   of an exception type with at least the same fields (each with the same
   type) as S.
   If S has the structure of a record type then T must have the structure of
   a record type with at least the same fields (each with the same type) as
   S.
   TODO: order of fields? *)
| T_Exception fields_s, T_Exception fields_t
| T_Record fields_s, T_Record fields_t ->
    List.for_all
      (fun (name_s, ty_s) ->
        List.exists
          (fun (name_t, ty_t) ->
            String.equal name_s name_t && type_equal env ty_s ty_t)
          fields_t)
      fields_s
| T_Exception _, _ | T_Record _, _ -> false (* A structure cannot be a name *)
| T_Named _, _ -> assert false |: TypingRule.StructuralSubtypeSatisfaction
```

6.3.4 Formally

6.3.5 Comments

This is related to $\mathtt{D}_{\mathtt{TRVR}}$, $\mathtt{I}_{\mathtt{SJDC}}$, $\mathtt{I}_{\mathtt{MHYB}}$, $\mathtt{I}_{\mathtt{TWTZ}}$, $\mathtt{I}_{\mathtt{GYSK}}$, $\mathtt{I}_{\mathtt{KXSD}}$.

6.4 TypingRule.DomainSubtypeSatisfaction

6.4.1 Prose

`T` domain-subtype-satisfies `S` if one of the following applies:

- All of the following apply:
  - ∗ `S` does not have the structure of an aggregate type or bitvector type;
  - ∗ the domain of `T` is a subset of the domain of `S`.
- All of the following apply:
- One of the following applies:
  - ∗ `S` has the structure of a bitvector type with undetermined width;
  - ∗ `T` has the structure of a bitvector type with undetermined width;
- the domain of `T` is a subset of the domain of `S`.

6.4.2 Example

6.4.3 Code

```
and domain_subtype_satisfies env t s =
  let s_struct = get_structure env s in
  match s_struct.desc with
  | T_Named _ ->
      (* Cannot happen *)
      assert false
      (* If S does not have the structure of an aggregate type or bitvector type
         then the domain of T must be a subset of the domain of S. *)
  | T_Tuple _ | T_Array _ | T_Record _ | T_Exception _ -> true
  | T_Real | T_String | T_Bool | T_Enum _ | T_Int _ ->
      let d_s = Domain.of_type env s_struct
      and d_t = get_structure env t |> Domain.of_type env in
      Domain.is_subset d_t d_s
  | T_Bits _ ->
      ((*
        • If either S or T have the structure of a bitvector type with
          undetermined width then the domain of T must be a subset of the domain
          of S.
         *)
       (* Implicitly, T must have the structure of a bitvector. *)
       let t_struct = get_structure env t in
       let t_domain = Domain.of_type env t_struct
       and s_domain = Domain.of_type env s_struct in
       let () =
```

```
      if false then
        Format.eprintf "Is %a included in %a?@." Domain.pp t_domain Domain.pp
          s_domain
    in
    match
      ( Domain.get_width_singleton_opt s_domain,
        Domain.get_width_singleton_opt t_domain )
    with
    | Some w_s, Some w_t -> Z.equal w_s w_t
    | _ -> Domain.is_subset t_domain s_domain)
   |: TypingRule.DomainSubtypeSatisfaction
```

6.4.4 Formally

6.4.5 Comments

This is related to $\texttt{D}_{\texttt{TRVR}}$.

6.5 TypingRule.SubtypeSatisfaction

6.5.1 Prose

`T` subtype-satisfies `S` if all of the following apply:

- `T` structural-subtype-satisfies `S`;
- `T` domain-subtype-satisfies `S`.

6.5.2 Example

6.5.3 Code

```
and subtype_satisfies env t s =
  let () =
    if false then
      let b1 = structural_subtype_satisfies env t s in
      let b2 = domain_subtype_satisfies env t s in
      Format.eprintf "%a subtypes %a ? struct: %B -- domain: %B@." PP.pp_ty t
        PP.pp_ty s b1 b2
  in
  structural_subtype_satisfies env t s
  && domain_subtype_satisfies env t s |: TypingRule.SubtypeSatisfaction
```

6.5.4 Formally

6.5.5 Comments

This is related to $\texttt{D}_{\texttt{TRVR}}$, $\texttt{I}_{\texttt{KNXJ}}$.

6.6 TypingRule.TypeSatisfaction

6.6.1 Prose

`T` type-satisfies `S` if one of the following applies:

- `T` is a subtype of `S`;
- All of the following apply:
  - `T` subtype-satisfies `S`;
  - One of the following applies:
    - `S` is an anonymous type;
    - `T` is an anonymous type;
- All of the following apply:
  - `T` is an anonymous bitvector with no bitfields;
  - `S` has the structure of a bitvector (with or without bitfields);
  - `S` has the same width as `T`.

6.6.2 Example: TypingRule.TypeSatisfaction1.asl

In the program:

```
type T1 of integer;          // the named type ‘T1‘ whose structure is integer
type T2 of integer;          // the named type ‘T2‘ whose structure is integer
type pairT of (integer, T1); // the named type ‘pairT‘ whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1, dataT1);
  // legal since the right hand side has anonymous, non-primitive type (integer, T1)
return 0;
end
```

`var pair: pairT = (1, dataT1)` is legal since the right-hand-side has anonymous, non-primitive type `(integer, T1)`.

6.6.3 Example: TypingRule.TypeSatisfaction2.asl

In the program:

```
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, in

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1,dataT1);

  let dataAsInt: integer = dataT1;
  pair = (1, dataAsInt);
  // legal since the right-hand-side has anonymous, primitive type (integer, integer)
  return 0;
end
```

`pair = (1, dataAsInt);` is legal since the right-hand-side has anonymous, primitive type `(integer, integer)`.

6.6.4 Example: TypingRule.TypeSatisfaction3.asl

In the program:

```
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, in

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1,dataT1);

  let dataT2: T2 = 10;
  pair = (1, dataT2);
  // illegal since the right-hand-side has anonymous, non-primitive type (integer, T2)
  // which does not subtype-satisfy named type pairT
  return 0;
end
```

`pair = (1, dataT2);` is illegal since the right-hand-side has anonymous, non-primitive type `(integer, T2)` which does not subtype-satisfy named type `pairT`.

6.6.5 Code

```
and type_satisfies env t s =
  (* Type T type-satisfies type S if and only if at least one of the following
```

```
   conditions holds: *)
(* T is a subtype of S *)
subtypes env t s
(* T subtype-satisfies S and at least one of S or T is an anonymous type *)
|| ((is_anonymous t || is_anonymous s) && subtype_satisfies env t s)
||
(* T is an anonymous bitvector with no bitfields and S has the structure of a
   bitvector (with or without bitfields) of the same width as T. *)
(* Here I interprete "same width" as statically the same width, otherwise
   it's strange. *)
match (t.desc, (get_structure env s).desc) with
| T_Bits (width_t, []), T_Bits (width_s, _) ->
    bitwidth_equal env width_t width_s
| _ -> false |: TypingRule.TypeSatisfaction
```

6.6.6 Formally

6.6.7 Comments

Since the subtype relation is a partial order, it is reflexive. Therefore every type T is a subtype of itself, and as a consequence, every type T type-satisfies itself.

This is related to $\mathtt{R}_{\mathtt{FMXK}}$ and $\mathtt{I}_{\mathtt{NLFD}}$.

6.7 TypingRule.CanAssignTo

Prose

S can be assigned to T if and only if all of the following apply:

- neither S nor T has the structure of the under-constrained integer type;
- T type-satisfies S.

6.7.1 Example

6.7.2 Code

```
let can_assign_to env s t =
  let s_struct = Types.get_structure env s
  and t_struct = Types.get_structure env t in
  match (s_struct.desc, t_struct.desc) with
  | T_Int (Some []), T_Int (Some []) -> false
  | _ -> Types.type_satisfies env t s |: TypingRule.CanAssignTo
```

6.7.3 Formally

6.7.4 Comments

This is related to $\texttt{R}_{\texttt{GNTS}}$, $\texttt{I}_{\texttt{MMKF}}$, $\texttt{I}_{\texttt{DGWJ}}$, $\texttt{I}_{\texttt{KKCC}}$ and $\texttt{R}_{\texttt{LXQZ}}$.

6.8 TypingRule.TypeClash

6.8.1 Prose

T type-clashes with S if one of the following applies:

- S and T both have the structure of integers;
- S and T both have the structure of reals;
- S and T both have the structure of strings;
- S and T both have the structure of enumeration types with the same enumeration literals;
- S and T both have the structure of bitvectors;
- S and T both have the structure of arrays whose element types type-clash;
- S and T both have the structure of tuples of the same length whose corresponding element types type-clash;
- S is a subtype of T;
- S is a supertype of T.

6.8.2 Example

6.8.3 Code

```
let rec type_clashes env t s =
  (*
   Definition VPZZ:
   A type T type-clashes with S if any of the following hold:
      • they both have the structure of integers
      • they both have the structure of reals
      • they both have the structure of strings
      • they both have the structure of enumeration types with the same
        enumeration literals
      • they both have the structure of bit vectors
      • they both have the structure of arrays whose element types type-clash
      • they both have the structure of tuples of the same length whose
        corresponding element types type-clash
      • S is either a subtype or a supertype of T *)
```

```
(* We will add a rule for boolean and boolean. *)
(subtypes env s t || subtypes env t s)
||
let s_struct = get_structure env s and t_struct = get_structure env t in
match (s_struct.desc, t_struct.desc) with
| T_Int _, T_Int _
| T_Real, T_Real
| T_String, T_String
| T_Bits _, T_Bits _
| T_Bool, T_Bool ->
    true
| T_Enum li_s, T_Enum li_t -> list_equal String.equal li_s li_t
| T_Array (_, ty_s), T_Array (_, ty_t) -> type_clashes env ty_s ty_t
| T_Tuple li_s, T_Tuple li_t ->
    List.compare_lengths li_s li_t = 0
    && List.for_all2 (type_clashes env) li_s li_t
| _ -> false |: TypingRule.TypeClash
```

6.8.4 Formally

6.8.5 Comments

Note that if `T` subtype-satisfies `S` then `T` and `S` type-clash, but not the other way around.

Note that type-clashing is an equivalence relation. Therefore if `T` type-clashes with `A` and `B` then it is also the case that `A` and `B` type-clash.

This is related to $\texttt{D}_{\texttt{VPZZ}}$, $\texttt{I}_{\texttt{PQCT}}$ and $\texttt{I}_{\texttt{WZKM}}$.

6.9 TypingRule.LowestCommonAncestor

6.9.1 Prose

The lowest common ancestor of types `S` and `T` is `ty` and one of the following applies:

- All of the following apply:
  - `S` and `T` are the same type;
  - `ty` is `S`.
- All of the following apply:
  - `S` and `T` are both named types;
  - `ty` is a common supertype of `S` and `T`;
  - `ty` is a subtype of all other common supertypes of `S` and `T`.
- All of the following apply:

- ∗ `S` and `T` both have the structure of array types with the same index type and the same element types;
- ∗ One of the following applies:
  - – All of the following apply:
    - ▷ `S` is a named type;
    - ▷ `T` is an anonymous type;
    - ▷ `ty` is `S`.
  - – All of the following apply:
    - ▷ `S` is an anonymous type;
    - ▷ `T` is a named type;
    - ▷ `ty` is `T`.

- All of the following apply:
  - ∗ `S` and `T` both have the structure of tuple types with the same number of elements;
  - ∗ The types of the elements of `S` type-satisfy the types of the elements of `T`;
  - ∗ The types of the elements of `T` type-satisfy the types of the elements of `S`;
  - ∗ One of the following applies:
  - ∗ All of the following apply:
    - – `S` is a named type;
    - – `T` is an anonymous type;
    - – `ty` is `S`.
  - ∗ All of the following apply:
    - – `S` is an anonymous type;
    - – `T` is a named type;
    - – `ty` is `T`.
  - ∗ All of the following apply:
    - – `S` is an anonymous type;
    - – `T` is an anonymous type;
    - – `ty` is the tuple type where the type of each element is the lowest common ancestor of the types of the corresponding elements of `S` and `T`.

- All of the following apply:
  - ∗ `S` and `T` both have the structure of well-constrained integer types;
  - ∗ One of the following applies:
    - – All of the following apply:

  - ▷ `S` is a named type;
  - ▷ `T` is an anonymous type;
  - ▷ `ty` is `S`.
- – All of the following apply:
  - ▷ `S` is an anonymous type;
  - ▷ `T` is a named type;
  - ▷ `ty` is `T`.
- – All of the following apply:
  - ▷ `S` is an anonymous type;
  - ▷ `T` is an anonymous type;
  - ▷ `ty` is the well-constrained integer type whose domain is the union of the domains of `S` and `T`.

- All of the following apply:
  - ∗ Either `S` or `T` have the structure of an unconstrained integer type;
  - ∗ One of the following applies:
  - ∗ All of the following apply:
    - – `S` is a named type;
    - – `S` has the structure of an unconstrained integer type;
    - – `T` is an anonymous type;
    - – `ty` is `S`.
  - ∗ All of the following apply:
    - – S is an anonymous type;
    - – T is a named type;
    - – T has the structure of an unconstrained integer type;
    - – ty is T.
  - ∗ All of the following apply:
    - – `S` is an anonymous type;
    - – `T` is an anonymous type;
    - – `ty` is the unconstrained integer type.
- All of the following apply:
  - ∗ Either `S` or `T` have the structure of an under-constrained integer type;
  - ∗ `ty` is the under-constrained integer type.
- `ty` is undefined.

6.9.2 Example

6.9.3 Code

```
let rec lowest_common_ancestor env s t =
  (* The lowest common ancestor of types S and T is: *)
  (* • If S and T are the same type: S (or T). *)
  if type_equal env s t then Some s
  else
    match (s.desc, t.desc) with
    | T_Named name_s, T_Named name_t -> (
        (* If S and T are both named types: the (unique) common supertype of S
           and T that is a subtype of all other common supertypes of S and T. *)
        match find_named_lowest_common_supertype env name_s name_t with
        | None -> None
        | Some name -> Some (T_Named name |> add_dummy_pos))
    | _ -> (
        let struct_s = get_structure env s and struct_t = get_structure env t in
        match (struct_s.desc, struct_t.desc) with
        | T_Array (l_s, t_s), T_Array (l_t, t_t)
          when type_equal env t_s t_t && expr_equal env l_s l_t -> (
            (* If S and T both have the structure of array types with the same
               index type and the same element types:
                { If S is a named type and T is an anonymous type: S
                { If S is an anonymous type and T is a named type: T *)
            match (s.desc, t.desc) with
            | T_Named _, T_Named _ -> assert false
            | T_Named _, _ -> Some s
            | _, T_Named _ -> Some t
            | _ -> assert false)
        | T_Tuple li_s, T_Tuple li_t
          when List.compare_lengths li_s li_t = 0
               && List.for_all2 (type_satisfies env) li_s li_t
               && List.for_all2 (type_satisfies env) li_t li_s -> (
            (* If S and T both have the structure of tuple types with the same
               number of elements and the types of elements of S type-satisfy the
               types of the elements of T and vice-versa:
                { If S is a named type and T is an anonymous type: S
                { If S is an anonymous type and T is a named type: T
                { If S and T are both anonymous types: the tuple type with the
                  type of each element the lowest common ancestor of the types of
                  the corresponding elements of S and T. *)
            match (s.desc, t.desc) with
            | T_Named _, T_Named _ -> assert false
            | T_Named _, _ -> Some s
            | _, T_Named _ -> Some t
```

```
  | _ ->
      let maybe_ancestors =
        List.map2 (lowest_common_ancestor env) li_s li_t
      in
      let ancestors = List.filter_map Fun.id maybe_ancestors in
      if List.compare_lengths ancestors li_s = 0 then
        Some (add_dummy_pos (T_Tuple ancestors))
      else None)
| T_Int (Some []), _ ->
    (* TODO: revisit? *)
    (* If either S or T have the structure of an under-constrained
       integer type: the under-constrained integer type. *)
    Some s
| _, T_Int (Some []) ->
    (* TODO: revisit? *)
    (* If either S or T have the structure of an under-constrained
       integer type: the under-constrained integer type. *)
    Some t
| T_Int (Some cs_s), T_Int (Some cs_t) -> (
    (* Implicit: cs_s and cs_t are non-empty, see patterns above. *)
    (* If S and T both have the structure of well-constrained integer
       types:
       { If S is a named type and T is an anonymous type: S
       { If T is an anonymous type and S is a named type: T
       { If S and T are both anonymous types: the well-constrained
         integer type with domain the union of the domains of S and T.
    *)
    match (s.desc, t.desc) with
    | T_Named _, T_Named _ -> assert false
    | T_Named _, _ -> Some s
    | _, T_Named _ -> Some t
    | _ ->
        (* TODO: simplify domains ? If domains use a form of diets,
           this could be more efficient. *)
        Some (add_dummy_pos (T_Int (Some (cs_s @ cs_t)))))
| T_Int None, _ -> (
    (* Here S has the structure of an unconstrained integer type. *)
    (* TODO: revisit? *)
    (* TODO: typo corrected here, on point 2 S and T have
       been swapped. *)
    (* If either S or T have the structure of an unconstrained integer
       type:
       { If S is a named type with the structure of an unconstrained
         integer type and T is an anonymous type: S
       { If T is an anonymous type and S is a named type with the
         structure of an unconstrained integer type: T
```

```
        { If S and T are both anonymous types: the unconstrained integer
          type. *)
      match (s.desc, t.desc) with
      | T_Named _, T_Named _ -> assert false
      | T_Named _, _ -> Some s
      | _, T_Named _ -> assert false
      | _, _ -> Some (add_dummy_pos (T_Int None)))
  | _, T_Int None -> (
      (* Here T has the structure of an unconstrained integer type. *)
      (* TODO: revisit? *)
      (* TODO: typo corrected here, on point 2 S and T have
         been swapped. *)
      (* If either S or T have the structure of an unconstrained integer
         type:
         { If S is a named type with the structure of an unconstrained
           integer type and T is an anonymous type: S
         { If T is an anonymous type and S is a named type with the
           structure of an unconstrained integer type: T
         { If S and T are both anonymous types: the unconstrained integer
           type. *)
      match (s.desc, t.desc) with
      | T_Named _, T_Named _ -> assert false
      | T_Named _, _ -> assert false
      | _, T_Named _ -> Some t
      | _, _ -> Some (add_dummy_pos (T_Int None)))
  | _ -> None |: TypingRule.LowestCommonAncestor)
```

6.9.4 Formally

6.9.5 Comments

This is related to $\mathtt{R}_{\mathtt{YZHM}}$.

6.10 TypingRule.CheckUnop

6.10.1 Goal

Checking compatibility of an unary operator with the type of its argument.

6.10.2 Prose

`t` is the result of checking compatibility of a unary operator `op` with type `t1` and one of the following applies:

- All of the following apply:
  - `op` is `BNOT`;

  * t1 type-satisfies boolean;
  * t is boolean;

- All of the following apply:
  * op is NEG;
  * One of the following applies:
    - t1 type-satisfies integer;
    - t1 type-satisfies real;
  * One of the following applies:
    - All of the following apply:
      ▷ t1 has the structure of an unconstrained integer;
      ▷ t is an unconstrained integer;
    - All of the following apply:
      ▷ t1 has the structure of a constrained integer;
      ▷ t is a constrained integer whose constraint is ;

- All of the following apply:
  * op is NOT;
  * t1 has the structure of a bitvector;
  * t is t1.

6.10.3 Example

6.10.4 Code

```
let check_unop loc env op t1 =
  match op with
  | BNOT ->
      let+ () = check_type_satisfies loc env t1 t_bool in
      T_Bool |> add_pos_from loc
  | NEG -> (
      let+ () =
        either
          (check_type_satisfies loc env t1 t_int)
          (check_type_satisfies loc env t1 t_real)
      in
      match (Types.get_structure env t1).desc with
      | T_Int None -> T_Int None |> add_pos_from loc
      | T_Int (Some cs) ->
          let neg e = E_Unop (NEG, e) |> add_pos_from e in
          let constraint_minus = function
            | Constraint_Exact e -> Constraint_Exact (neg e)
```

```
          | Constraint_Range (top, bot) ->
              Constraint_Range (neg bot, neg top)
        in
        T_Int (Some (List.map constraint_minus cs)) |> add_pos_from loc
    | _ -> (* fail case *) t1)
| NOT ->
    let+ () = check_structure_bits loc env t1 in
    t1 |: TypingRule.CheckUnop
```

6.10.5 Formally

6.10.6 Comments

6.11 TypingRule.CheckBinop

6.11.1 Goal

Checking compatibility of a binary operator with the types of its arguments.

6.11.2 Prose

`t` is the result of checking compatibility of a binary operator `op` with types `t1` and `t2` and one of the following applies:

- All of the following apply:
  - ∗ `op` is `AND`, `OR`, `EQ` or `IMPL`;
  - ∗ `t1` type-satisfies `boolean`;
  - ∗ `t2` type-satisfies `boolean`;
  - ∗ `t` is `boolean`.
- All of the following apply:
  - ∗ `op` is `AND`, `OR`, or `EOR`;
  - ∗ `t1` has the structure of a bitvector;
  - ∗ `t2` has the structure of a bitvector;
  - ∗ `t1` and `t2` have the same bitvector width `w`;
  - ∗ `t` is the bitvector type of width `w`.
- All of the following apply:
  - ∗ `op` is `PLUS` or `MINUS`;
  - ∗ `t1` has the structure of a bitvector;
  - ∗ `t2` has the structure of a bitvector;
  - ∗ `t1` and `t2` have the same bitvector width `w`;

  - `t2` type-satisfies `integer`;
  - `t` is the bitvector type of width `w`.

- All of the following apply:
  - `op` is `EQ_OP` or `NEQ`;
  - One of the following applies:
    - `t1` is equal to `t2`;
    - All of the following apply:
      - ▷ `t1` type-satisfies `integer`;
      - ▷ `t2` type-satisfies `integer`;
    - All of the following apply:
      - ▷ `t1` has the structure of a bitvector;
      - ▷ `t2` has the structure of a bitvector;
      - ▷ `t1` and `t2` have the same bitvector width;
    - All of the following apply:
      - ▷ `t1` type-satisfies `boolean`;
      - ▷ `t2` type-satisfies `boolean`;
    - All of the following apply:
      - ▷ `t1` enumerates local declarations `li1`;
      - ▷ `t2` enumerates local declarations `li2`;
      - ▷ `li1` equals `li2`;
  - `t` is `boolean`.

- All of the following apply:
  - `op` is `LEQ`, `GEQ`, `GT` or `LT`;
  - One of the following applies:
    - All of the following apply:
      - ▷ `t1` type-satisfies `integer`;
      - ▷ `t2` type-satisfies `integer`;
    - All of the following apply:
      - ▷ `t1` type-satisfies `real`;
      - ▷ `t2` type-satisfies `real`;
  - `t` is boolean.

- All of the following apply:
  - `op` is `MUL`, `DIV`, `DIVRM`, `MOD`, `SHL`, `SHR`, `POW`, `PLUS` or `MINUS`;
  - `struct1` is the structure of `t1`;
  - `struct2` is the structure of `t2`;
  - One of the following applies:

- All of the following apply:
  - ▷ `t1` has the structure of an unconstrained integer;
  - ▷ `t2` has the structure of an integer;
  - ▷ `t` is an unconstrained integer;
- All of the following apply:
  - ▷ `t1` has the structure of an integer;
  - ▷ `t2` has the structure of an unconstrained integer;
  - ▷ `t` is an unconstrained integer;
- One of the following applies:
  - ▷ All of the following apply:
    - + `t1` has the structure of an under-constrained integer;
    - + `t2` has the structure of a constrained integer;
    - + `t` is an under-constrained integer;
  - ▷ All of the following apply:
    - + `t1` has the structure of a constrained integer;
    - + `t2` has the structure of an under-constrained integer;
    - + `t` is an under-constrained integer;
- One of the following applies:
  - ▷ All of the following apply:
    - + `t1` has the structure of a well-constrained integer;
    - + `t2` has the structure of a well-constrained integer;
    - + `t` is a constrained integer whose constraint is calculated by applying the operation to all possible value pairs;
  - ▷ All of the following apply:
    - + `t1` has the structure of a well-constrained integer;
    - + `t2` has the structure of an well-constrained integer;
    - + `t` is a constrained integer whose constraint is calculated by applying the operation to all possible value pairs;
- All of the following apply:
  - ▷ `t1` has the structure of `real`;
  - ▷ `t2` has the structure of `real`;
  - ▷ `op` is `PLUS`, `MINUS` or `MUL`;
  - ▷ `t` is `real`;
- All of the following apply:
  - ▷ `t1` has the structure of `real`;
  - ▷ `t2` has the structure of `integer`;
  - ▷ `op` is `POW`;
  - ▷ `t` is `real`;

- All of the following apply:

  * op is RDIV;
  * t1 type-satisfies real;
  * t is real.

6.11.3 Example

6.11.4 Code

```
let check_unop loc env op t1 =
  match op with
  | BNOT ->
      let+ () = check_type_satisfies loc env t1 t_bool in
      T_Bool |> add_pos_from loc
  | NEG -> (
      let+ () =
        either
          (check_type_satisfies loc env t1 t_int)
          (check_type_satisfies loc env t1 t_real)
      in
      match (Types.get_structure env t1).desc with
      | T_Int None -> T_Int None |> add_pos_from loc
      | T_Int (Some cs) ->
          let neg e = E_Unop (NEG, e) |> add_pos_from e in
          let constraint_minus = function
            | Constraint_Exact e -> Constraint_Exact (neg e)
            | Constraint_Range (top, bot) ->
                Constraint_Range (neg bot, neg top)
          in
          T_Int (Some (List.map constraint_minus cs)) |> add_pos_from loc
      | _ -> (* fail case *) t1)
  | NOT ->
      let+ () = check_structure_bits loc env t1 in
      t1 |: TypingRule.CheckUnop
```

6.11.5 Formally

$$\frac{op \in \texttt{binop\_boolean} \qquad (G,L) \vdash e_1 : \texttt{T\_Bool} \qquad (G,L) \vdash e_2 : \texttt{T\_Bool}}{(G,L) \vdash \texttt{E\_Binop}(op, e_1, e_2) : \texttt{T\_Bool1}}$$

6.11.6 Comments

This is related to $\mathrm{R}_{\mathtt{BKNT}}$, $\mathrm{R}_{\mathtt{ZYWY}}$, $\mathrm{R}_{\mathtt{BZKW}}$, $\mathrm{R}_{\mathtt{KFYS}}$, $\mathrm{R}_{\mathtt{KXMR}}$, $\mathrm{R}_{\mathtt{SQXN}}$, $\mathrm{R}_{\mathtt{MRHT}}$, $\mathrm{R}_{\mathtt{JGWF}}$, $\mathrm{R}_{\mathtt{TTGQ}}$, $\mathrm{I}_{\mathtt{YHML}}$, $\mathrm{I}_{\mathtt{YHRP}}$, $\mathrm{I}_{\mathtt{VMZF}}$, $\mathrm{I}_{\mathtt{YXSY}}$, $\mathrm{I}_{\mathtt{LGHJ}}$, $\mathrm{I}_{\mathtt{RXLG}}$.

Chapter 7

Typing of Expressions

`annotate_expr` specifies how to annotate an expression `e` in an environment `env`. Formally, the result of annotating the expression `e` in `env` is `t,new_env` where `t` is a type and `new_env` is an environment, or an error, and one of the following applies:

- TypingRule.Lit (see Section 7.1);
- TypingRule.ELocalVarConstant (see Section 7.2)
- TypingRule.ELocalVar (see Section 7.3)
- TypingRule.EGlobalVarConstant (see Section 7.4)
- TypingRule.EGlobalVar (see Section 7.5)
- TypingRule.EUndefIdent (see Section 7.6)
- TypingRule.Binop (see Section 7.7)
- TypingRule.Unop (see Section 7.8)
- TypingRule.ECond (see Section 7.9)
- TypingRule.ESlice (see Section 7.10)
- TypingRule.ECall (see Section 7.11)
- TypingRule.EGetArray (see Section 7.12)
- TypingRule.EStructuredNotStructured (see Section 7.13)
- TypingRule.EStructuredMissingField (see Section 7.14)
- TypingRule.ERecord (see Section 7.15)
- TypingRule.EGetRecordField (see Section 7.16)

- TypingRule.EGetBadRecordField (see Section 7.17)
- TypingRule.EGetBadBitField (see Section 7.18)
- TypingRule.EGetBadField (see Section 7.19)
- TypingRule.EGetBitField (see Section 7.20)
- TypingRule.EGetBitFieldNested (see Section 7.21)
- TypingRule.EGetBitFieldTyped (see Section 7.22)
- TypingRule.EConcatEmpty (see Section 7.23)
- TypingRule.EConcat (see Section 7.24)
- TypingRule.ETuple (see Section 7.25)
- TypingRule.EUnknown (see Section 7.26)
- TypingRule.EPattern (see Section 7.27)
- TypingRule.CTC (see Section 7.28)

7.1 TypingRule.Lit

7.1.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` is a Literal `v`;
- `t` is the type of `v`;
- `new_env` is `e`.

7.1.2 Example

7.1.3 Code

```
| E_Literal v -> (infer_value v |> here, e) |: TypingRule.Lit
```

7.1.4 Formally

7.1.5 Comments

7.2 TypingRule.ELocalVarConstant

7.2.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes a variable `x`;
- `x` is bound to a local constant `v` of type `ty` in the local environment given by `env`;
- `t` is `ty`;
- `new_env` is the Literal `v`.

7.2.2 Example

7.2.3 Code

```
| ty, LDK_Constant ->
    let v = IMap.find x env.local.constants_values in
    let e = E_Literal v |> here in
    (ty, e) |: TypingRule.ELocalVarConstant
```

7.2.4 Formally

7.2.5 Comments

7.3 TypingRule.ELocalVar

7.3.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes a variable `x`;
- `x` is not bound to a local constant;
- `x` has type `ty` in the local environment given by `env`;
- `t` is `ty`;
- `new_env` is `e`.

7.3.2 Example

7.3.3 Code

```
| ty, _ -> (ty, e) |: TypingRule.ELocalVar
```

7.3.4 Formally

7.3.5 Comments

7.4 TypingRule.EGlobalVarConstantVal

7.4.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes a variable `x`;
- `x` is bound to a global constant `v` of type `ty` in the global environment given by `env`;
- `t` is `ty`;
- `new_env` is the Literal `v`.

7.4.2 Example

7.4.3 Code

```
| ty, GDK_Constant -> (
    match IMap.find_opt x env.global.constants_values with
    | Some v ->
        (ty, E_Literal v |> here)
        |: TypingRule.EGlobalVarConstantVal
```

7.4.4 Formally

7.4.5 Comments

7.5 TypingRule.EGlobalVar

7.5.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes a variable `x`;
- `x` is not bound to a global constant;
- `x` has type `ty` in the global environment given by `env`;
- `t` is `ty`;
- `new_env` is `e`.

7.5.2 Example

7.5.3 Code

```
| ty, _ -> (ty, e) |: TypingRule.EGlobalVar
```

7.5.4 Formally

7.5.5 Comments

7.6 TypingRule.EUndefIdent

7.6.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` is a variable `x`;
- `x` is not bound in `env`;
- an error "`Undefined Identifier`" is raised.

7.6.2 Example

7.6.3 Code

```
with Not_found ->
  let () =
    if false then
      Format.eprintf "@[Cannot find %s in env@ %a.@]@." x pp_env env
  in
  undefined_identifier e x |: TypingRule.EUndefIdent))
```

7.6.4 Formally

7.6.5 Comments

7.7 TypingRule.Binop

7.7.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes a binary operation `op` over two expressions `e1` and `e2`;
- `t1,e1'` is the result of annotating `e1` in `env`;
- `t2,e2'` is the result of annotating `e2` in `env`;

- t is the result of checking compatibility of op with t1 and t2 as per Section 6.11;
- new_env denotes op over e1' and e2'.

7.7.2 Example

7.7.3 Code

```
| E_Binop (op, e1, e2) ->
    let t1, e1' = annotate_expr env e1 in
    let t2, e2' = annotate_expr env e2 in
    let t = check_binop e env op t1 t2 in
    (t, E_Binop (op, e1', e2') |> here) |: TypingRule.Binop
```

7.7.4 Formally

7.7.5 Comments

7.8 TypingRule.Unop

7.8.1 Prose

The result of annotating the expression e in env is t,new_env and all of the following apply:

- e denotes a unary operation op over an expression e';
- t'',e'' is the result of annotating e' in env;
- t is the result of checking compatibility of op with t'' as per Section 6.10;
- new_env denotes op over e''.

7.8.2 Example

7.8.3 Code

```
| E_Unop (op, e') ->
    let t'', e'' = annotate_expr env e' in
    let t = check_unop e env op t'' in
    (t, E_Unop (op, e'') |> here) |: TypingRule.Unop
```

7.8.4 Formally

7.8.5 Comments

7.9 TypingRule.ECond

7.9.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes a conditional expression with condition `e_cond` with two options `e_true` and `e_false`;
- `t_cond, e'_cond` is the result of annotating `e_cond` in `env`;
- `t_true, e'_true` is the result of annotating `e_true` in `env`;
- `t_false, e'_false` is the result of annotating `e_false` in `env`;
- One of the following applies:
  * All of the following apply:
    - `t` is the lowest common ancestor of `t_true` and `t_false`;
    - `new_env` is the condition `e'_cond` with two options `e'_true` and `e'_false`.
  * All of the following apply:
    - there is no lowest common ancestor of `t_true` and `t_false`;
    - an error "`Unreconciliable Types`" is raised.

7.9.2 Example

7.9.3 Code

```
| E_Cond (e_cond, e_true, e_false) ->
    let t_cond, e'_cond = annotate_expr env e_cond in
    let+ () = check_structure_boolean e env t_cond in
    let t_true, e'_true = annotate_expr env e_true
    and t_false, e'_false = annotate_expr env e_false in
    let t =
      best_effort t_true (fun _ ->
          match Types.lowest_common_ancestor env t_true t_false with
          | None ->
              fatal_from e (Error.UnreconciliableTypes (t_true, t_false))
          | Some t -> t)
    in
    (t, E_Cond (e'_cond, e'_true, e'_false) |> here) |: TypingRule.ECond
```

7.9.4 Formally

7.9.5 Comments

This is related to $\texttt{R}_{\texttt{XZVT}}$.

7.10 TypingRule.ESlice

7.10.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and 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 `env`;
- an error "`Conflicting Types`" is raised or `t_e'` has the structure of an integer or a bitvector and all of the following apply:
- `w` is the width of `slices`;
- `slices'` is the result of annotating `slices` in `env`;
- `t` is the bitvector type of width `w`;
- `new_env` is the slicing of expression `e'` by the slices `slices'`.

7.10.2 Example

7.10.3 Code

```
| T_Int _ | T_Bits _ ->
    let w = slices_width env slices in
    (* TODO: check that:
       - Rule SNQJ: An expression or subexpression which may result in
         a zero-length bitvector must not be side-effecting.
    *)
    let slices' = best_effort slices (annotate_slices env) in
    (T_Bits (w, []) |> here, E_Slice (e', slices') |> here)
    |: TypingRule.ESlice
```

7.10.4 Formally

7.10.5 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 initialiser expression.

This is related to $\texttt{I}_{\texttt{MJWM}}$.

7.11 TypingRule.ECall

7.11.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes a call to a subprogram named `name` with arguments `args` and parameters `eqs`;
- `name', args', eqs', ty` is the result of annotating the call of that subprogram in `env`;
- `t` is `ty`;
- `new_env` is the call to the subprogram named `name'` with arguments `args'` and parameters `eqs'`.

7.11.2 Example

7.11.3 Code

```
| E_Call (name, args, eqs) ->
    let name', args', eqs', ty_opt =
      annotate_call (to_pos e) env name args eqs ST_Function
    in
    let t = match ty_opt with Some ty -> ty | None -> assert false in
    (t, E_Call (name', args', eqs') |> here) |: TypingRule.ECall
```

7.11.4 Formally

7.11.5 Comments

This is related to $\texttt{D}_{\texttt{CFYP}}$, $\texttt{R}_{\texttt{BQJG}}$.

7.12 TypingRule.EGetArray

7.12.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` or an error and 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 `env`;
- `t_e'` has the structure of an array with index type `wanted_t_index` and element type `t`;

- an error "Conflicting Types" is raised or `slices` is a single expression `e_index` and all of the following apply:
- `t_index', e_index'` is the result of annotating `e_index` in `env`;
- an error "`Conflicting Types`" or `t_index'` type-satisfies `wanted_t_index` as per Section 6.6 and all of the following apply:
- `new_e` is an access to array `e'` at index `e_index'`.

7.12.2 Example

7.12.3 Code

```
| T_Array (size, ty') -> (
    let wanted_t_index =
      let t_int =
        T_Int
          (Some [ Constraint_Range (!$0, binop MINUS size !$1) ])
        |> here
      in
      match size.desc with
      | E_Var name -> (
          match IMap.find_opt name env.global.declared_types with
          | Some t -> t (* TODO check that this is an enum *)
          | None -> t_int)
      | _ -> t_int
    in
    match slices with
    | [ Slice_Single e_index ] ->
        let t_index', e_index' = annotate_expr env e_index in
        let+ () =
          check_type_satisfies e env t_index' wanted_t_index
        in
        (ty', E_GetArray (e', e_index') |> here)
    | _ -> conflict e [ T_Int None; default_t_bits ] t_e')
| _ -> conflict e [ T_Int None; default_t_bits ] t_e' |: TypingRule.EGetAr
```

7.12.4 Formally

7.12.5 Comments

7.13 TypingRule.EStructuredNotStructured

7.13.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the record expression or an exception expression of type `ty` with fields `fields`;
- `ty` is neither a record nor an exception type;
- an error "`Conflicting Types`" is raised.

7.13.2 Example

7.13.3 Code

```
match (Types.get_structure env ty).desc with
| T_Exception fields | T_Record fields -> fields
| _ -> conflict e [ T_Record [] ] ty |: TypingRule.EStructuredNotStructured
```

7.13.4 Formally

7.13.5 Comments

This is related to $\mathtt{R}_{\mathtt{WBCQ}}$.

7.14 TypingRule.EStructuredMissingField

7.14.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the record expression or an exception expression of type `ty` with fields `fields`;
- `ty` is the name of a record or exception type with fields `field_types`;
- one field in `field_types` is not initialised by `fields`;
- an error "`Missing Field`" is raised.

7.14.2 Example

7.14.3 Code

```
fatal_from e (Error.MissingField (List.map fst fields, ty))
|: TypingRule.EStructuredMissingField
```

7.14.4 Formally

7.14.5 Comments

This is related to $\mathtt{R}_{\mathtt{WBCQ}}$.

7.15 TypingRule.ERecord

7.15.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the record expression of type `ty` with fields `fields`;
- `ty` is the name of a record type with fields `field_types`;
- For each field named `name` associated with the expression `e'` in `field_types`, all of the following apply:
  - `t',e''` is the result of annotating `e'` in `env`;
  - `t_spec'` is the type associated to `name` in `field_types`;
  - `t'` type-satisfies `t_spec'` as per Section 6.6;
  - `fields'` associates `name` to `e''`;
- `t` is `ty`;
- `new_env` is the record expression of type `ty` with fields `fields'`.

7.15.2 Example

7.15.3 Code

```
          List.map
            (fun (name, e') ->
              let t', e'' = annotate_expr env e' in
              let t_spec' =
                match List.assoc_opt name field_types with
                | None -> fatal_from e (Error.BadField (name, ty))
                | Some t_spec' -> t_spec'
              in
              (* TODO:
                 Rule LXQZ: A storage element of type S, where S is any
                 type that does not have the structure of the
                 under-constrained integer type, may only be assigned
                 or initialized with a value of type T if T
                 type-satisfies S. *)
              let+ () = check_type_satisfies e env t' t_spec' in
              (name, e''))
            fields)
    in
    (ty, E_Record (ty, fields') |> here) |: TypingRule.ERecord
```

7.15.4 Formally

7.15.5 Comments

This is related to $\texttt{R}_{\texttt{WBCQ}}$.

7.16 TypingRule.EGetRecordField

7.16.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the access of field `field_name` on expression `e1`;
- `t_e1, e2` is the result of annotating `e1` in `env`;
- `t_e1` has the structure of an exception or record type with fields `fields`;
- `t_e2` has the structure of an exception or record type with fields `fields`;
- `field_name` is declared in `fields`;
- `t` is the type corresponding to `field_name` in `fields`;
- `new_env` is the access of field `field_name` on expression `e2`.

7.16.2 Example

7.16.3 Code

```
| Some t ->
    (t, E_GetField (e2, field_name) |> here)
    |: TypingRule.EGetRecordField)
```

7.16.4 Formally

7.16.5 Comments

7.17 TypingRule.EGetBadRecordField

7.17.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the access of field `field_name` on expression `e1`;
- `t_e1, e2` is the result of annotating `e1` in `env`;
- `t_e1` has the structure of an exception or record type with fields `fields`;

- `t_e2` has the structure of an exception or record type with fields `fields`;
- `t_e2` is an Exception or a Record type with fields `fields`;
- `field_name` is not declared in `fields`;
- an error "`Bad Field`" is raised.

7.17.2 Example

7.17.3 Code

```
| None ->
    fatal_from e (Error.BadField (field_name, t_e2))
    |: TypingRule.EGetBadRecordField
```

7.17.4 Formally

7.17.5 Comments

7.18 TypingRule.EGetBadBitField

7.18.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the access of field `field_name` on expression `e1`;
- `t_e1` has the structure a bitvector type with bitfields `bitfields`;
- `t_e2` has the structure a bitvector type with bitfields `bitfields`;
- `field_name` is not declared in `bitfields`;
- an error "`Bad Field`" is raised.

7.18.2 Example

7.18.3 Code

```
| None ->
    fatal_from e (Error.BadField (field_name, t_e2))
    |: TypingRule.EGetBadBitField
```

7.18.4 Formally

7.18.5 Comments

7.19 TypingRule.EGetBadField

7.19.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the access of field `field_name` on expression `e1`;
- `t_e1, e2` is the result of annotating `e1` in `env`;
- `t_e1` does not have the structure of a record or an exception or a bitvector type;
- an error "`Conflicting Types`" is raised.

7.19.2 Example

7.19.3 Code

```
| _ ->
    conflict e [ default_t_bits; T_Record []; T_Exception [] ] t_e1
    |: TypingRule.EGetBadField)
```

7.19.4 Formally

7.19.5 Comments

7.20 TypingRule.EGetBitField

7.20.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the access of field `field_name` on expression `e1`;
- `t_e1, e2` is the result of annotating `e1` in `env`;
- `t_e1` has the structure of a bitvector type with bitfields `bitfields`;
- `t_e2` has the structure of a bitvector type with bitfields `bitfields`;
- `field_name` is declared in `bitfields`;
- `slices` gives the slices corresponding to the bitfield `field_name` in `bitfields`;
- `e3` denotes the slicing of the expression `e2` by the slices `slices`;
- `t,new_env` is the result of annotating `e3`.

7.20.2 Example

7.20.3 Code

```
| Some (BitField_Simple (_field, slices)) ->
    let e3 = E_Slice (e1, slices) |> here in
    annotate_expr env e3 |: TypingRule.EGetBitField
```

7.20.4 Formally

7.20.5 Comments

7.21 TypingRule.EGetBitFieldNested

7.21.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the access of field `field_name` on expression `e1`;
- `t_e1, e2` is the result of annotating `e1` in `env`;
- `t_e1` has the structure of a bitvector type with bitfields `bitfields`;
- `t_e2` has the structure of a bitvector type with bitfields `bitfields`;
- `field_name` is declared in `bitfields`;
- `slices` gives the slices corresponding to the bitfield `field_name` in `bitfields`;
- `e3` denotes the slicing of the expression `e2` by the slices `slices`;
- `t4, e4` is the result of annotating `e3` in `env`;
- `bitfields'` gives the bitfields corresponding to the bitfield `field_name` in `bitfields`;
- `t` is the bitvector type with the width of `t4` and the bitfields `bitfields'`
- `new_env` is `e4`.

7.21.2 Example

7.21.3 Code

```
| Some (BitField_Nested (_field, slices, bitfields')) ->
    let t_e3, e3 =
      E_Slice (e2, slices) |> here |> annotate_expr env
    in
    let t_e4 =
```

```
          match t_e3.desc with
          | T_Bits (width, _bitfields) ->
              T_Bits (width, bitfields') |> add_pos_from t_e2
          | _ -> assert false
        in
        (t_e4, e3) |: TypingRule.EGetBitFieldNested
```

7.21.4 Formally

7.21.5 Comments

7.22 TypingRule.EGetBitFieldTyped

7.22.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes `e1, field_name`;
- `t_e1, e2` is the result of annotating `e1` in `env`;
- `t_e1` has the structure of a bitvector type with bitfields `bitfields`;
- `t_e2` has the structure of a bitvector type with bitfields `bitfields`;
- `field_name` is declared in `bitfields`;
- `slices` gives the slices corresponding to the bitfield `field_name` in `bitfields`;
- `t_e3,e3` is the result of annotating `e2,slices` in `env`;
- `t` gives the type corresponding to the bitfield `field_name` in `bitfields`;
- `t_e3` type-satisfies `t` in `env`;
- `new_env` is `e3`.

7.22.2 Example

7.22.3 Code

```
      | Some (BitField_Type (_field, slices, t)) ->
          let t_e3, e3 =
            E_Slice (e2, slices) |> here |> annotate_expr env
          in
          let+ () = check_type_satisfies e3 env t_e3 t in
          (t, e3) |: TypingRule.EGetBitFieldTyped)
```

7.22.4 Formally

7.22.5 Comments

7.23 TypingRule.EConcatEmpty

7.23.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the empty concatenation;
- `t` is `bits(0)`;
- `new_env` is `e`.

7.23.2 Example

7.23.3 Code

```
| E_Concat [] ->
    (T_Bits (expr_of_int 0, []) |> here, e) |: TypingRule.EConcatEmpty
```

7.23.4 Formally

7.23.5 Comments

7.24 TypingRule.EConcat

7.24.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes the concatenation of a non-empty list of expressions `li`;
- `ts, es` is the result of annotating `li` in `env`;
- all elements of `ts` have the structure of a bitvector type;
- `w` is the sum of the widths of the bitvector types `ts`;
- `t` is `bits(w)`;
- `new_env` is `es`.

7.24.2 Example

7.24.3 Code

```
| E_Concat (_ :: _ as li) ->
    let ts, es = List.map (annotate_expr env) li |> List.split in
    let w =
      let widths = List.map (get_bitvector_width e env) ts in
      let wh = List.hd widths and wts = List.tl widths in
      List.fold_left (width_plus env) wh wts
    in
    (T_Bits (w, []) |> here, E_Concat es |> here) |: TypingRule.EConcat
```

7.24.4 Formally

7.24.5 Comments

This is related to $\mathrm{R}_{\texttt{NYNK}}$ and $\mathrm{R}_{\texttt{KCZS}}$.

The sum of the widths of the bitvector types `ts` might be a symbolic expression that is unresolvable to an integer. For example:

```
func foo{N}(x: bits(N)) => bit
begin
    return x[0];
end

config LIMIT1: integer = 2;
config LIMIT2: integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10} = 7;

func bar() => integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
begin
    var ret: integer = 0;
    while ret < LIMIT1 do
        ret = ret + ret * 2;
    end
    return ret as integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
end

func main() => integer
begin
    let N = bar();
    let M = LIMIT2;
    let x = Zeros(N);
    let y = Zeros(M);
    let z = foo([x, y]);
    return 0;
end
```

7.25 TypingRule.ETuple

7.25.1 Prose

The result of annotating the expression e in env is t,new_env and all of the following apply:

- e denotes a tuple li;
- ts, es is the result of annotating in env each expression in li;
- t is ts;
- new_env is es.

7.25.2 Example

7.25.3 Code

```
| E_Tuple li ->
    let ts, es = List.map (annotate_expr env) li |> List.split in
    (T_Tuple ts |> here, E_Tuple es |> here) |: TypingRule.ETuple
```

7.25.4 Formally

7.25.5 Comments

7.26 TypingRule.EUnknown

7.26.1 Prose

The result of annotating the expression e in env is t,new_env and all of the following apply:

- e denotes an expression UNKNOWN of type ty;
- ty' is the structure of ty in env;
- t is ty;
- new_env is an expression UNKNOWN of type ty'.

7.26.2 Example

7.26.3 Code

```
| E_Unknown ty ->
    let ty' = Types.get_structure env ty in
    (ty, E_Unknown ty' |> here) |: TypingRule.EUnknown
```

7.26.4 Formally

7.26.5 Comments

7.27 TypingRule.EPattern

7.27.1 Prose

The result of annotating the expression `e` in `env` is `t,new_env` and all of the following apply:

- `e` denotes whether the expression `e'` matches `patterns`;
- `t_e', e''` is the result of annotating `e'` in `env`;
- `patterns'` is the result of annotating `patterns, t_e'` in `env`;
- `t` is `boolean`;
- `new_env` denotes whether the expression `e''` matches `patterns'`.

7.27.2 Example

7.27.3 Code

```
| E_Pattern (e', patterns) ->
    (*
     Rule ZNDL states that

        The IN operator is equivalent to testing its first operand for
        equality against each value in the (possibly infinite) set denoted
        by the second operand, and taking the logical OR of the result.
        Values denoted by a bitmask_lit comprise all bitvectors that could
        match the bit-mask. It is not an error if any or all of the values
        denoted by the first operand can be statically determined to never
        compare equal with the second operand.

      e IN pattern            is sugar for
        "-"                       ->         TRUE
      | e1=expr                   ->         e == e1
      | bitmask_lit               ->         not yet implemented
      | e1=expr ".." e2=expr      ->         e1 <= e && e <= e2
      | "<=" e1=expr              ->         e <= e1
      | ">=" e1=expr              ->         e >= e1
      |  { p0 , ... pN }          ->         e IN p0 || ... e IN pN
      | !{ p0 , ... pN }          ->         not (e IN p0) && ... e IN pN

     We cannot reduce them here (as otherwise e might be evaluated a bad
     number of times), but we will apply the same typing rules as for those
```

```
     desugared expressions.
     *)
    let t_e', e'' = annotate_expr env e' in
    let patterns' = best_effort patterns (annotate_pattern e env t_e') in
    (T_Bool |> here, E_Pattern (e'', patterns') |> here)
    |: TypingRule.EPattern
```

7.27.4 Formally

7.27.5 Comments

7.28 TypingRule.CTC

7.28.1 Prose

The result of annotating the expression `e` in `env` is `t,new_e` and all of the following apply:

- `e` denotes an expression `e'` and a type `t'`;
- `t'',new_e'` is the result of annotating `e'` in `env`;
- One of the following applies:
  - All of the following apply:
    - `t''` is a structural subtype of `t` in `env`;
    - `t''` is a domain subtype of `t` in `env`;
    - `new_e` is `new_e'`.
  - All of the following apply:
    - `t''` is a structural subtype of `t'` in `env`;
    - `t''` is not a domain subtype of `t'` in `env`;
    - an execution-time check that the expression evaluates to a value in the domain of the required type is required.
    - `new_e` is the expression denoting a Checked Type Conversion of `new_e'` in the type `t'`.
  - All of the following apply:
    - `t''` is not a structural subtype of `t'` in `env`;
    - a “`ConflictingTypes`” error is raised.

7.28.2 Example

7.28.3 Code

```
| E_CTC (e', t') ->
    let t'', e'' = annotate_expr env e' in
```

```
(* - If type-checking determines that the expression type-satisfies
     the required type, then no further check is required.
   - If the expression only fails to type-satisfy the required type
     because the domain of its type is not a subset of the domain of
     the required type, an execution-time check that the expression
     evaluates to a value in the domain of the required type is
     required. *)
best_effort
  (t', E_CTC (e'', t') |> here)
  (fun res ->
    let env' = env in
    if Types.structural_subtype_satisfies env' t'' t' then
      if Types.domain_subtype_satisfies env' t'' t' then
        (* I am disabling the opmitization here as long as the type
           system is not sound. *)
        (* (t', e'') *)
        res
      else res
    else conflict e [ t'.desc ] t'')
|: TypingRule.CTC
```

7.28.4 Formally

7.28.5 Comments

This is related to R_{VBLL}, I_{KRLL}, G_{PFRQ}, I_{XVBG}, R_{GYJZ}, I_{SZVF}, R_{PZZJ}, R_{YCPX}, I_{ZLBW}, I_{TCST}, I_{CGRH}, I_{YJBB}.

Chapter 8

Typing of Left-Hand-Side Expressions

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and one of the following applies:

- TypingRule.LEDiscard (see Section 8.1),
- TypingRule.LELocalVar (see Section 8.2),
- TypingRule.LEGlobalVar (see Section 8.3),
- TypingRule.LEDestructuring (see Section 8.4),
- TypingRule.LESlice (see Section 8.5),
- TypingRule.LESetArray (see Section 8.6),
- TypingRule.LESetBadStructuredField (see Section 8.7),
- TypingRule.LESetStructuredField (see Section 8.8),
- TypingRule.LESetBadBitField (see Section 8.9),
- TypingRule.LESetBitField (see Section 8.10),
- TypingRule.LESetBitFieldNested (see Section 8.11),
- TypingRule.LESetBitFieldTyped (see Section 8.12),
- TypingRule.LESetBadField (see Section 8.13),
- TypingRule.LEConcat (see Section 8.14).

8.1 TypingRule.LEDiscard

8.1.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes an expression which can be discarded;
- `new_le` is `le`.

8.1.2 Example

8.1.3 Code

```
| LE_Discard -> le |: TypingRule.LEDiscard
```

8.1.4 Formally

8.1.5 Comments

8.2 TypingRule.LELocalVar

8.2.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes a local variable `x`;
- `x` is locally declared as a mutable variable of type `ty` in `env`;
- `t_e` can be assigned to `ty`;
- `new_le` is `le`.

8.2.2 Example

8.2.3 Code

```
match IMap.find_opt x env.local.storage_types with
| Some (ty, LDK_Var) -> ty |: TypingRule.LELocalVar
```

8.2.4 Formally

8.2.5 Comments

This is related to $\mathtt{R}_{\mathtt{WDGQ}}$.

8.3 TypingRule.LEGlobalVar

8.3.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes a local variable `x`;
- `x` is globally declared as a variable of type `ty` in `env`;
- `t_e` can be assigned to `ty`;
- `new_le` is `le`.

8.3.2 Example

8.3.3 Code

```
match IMap.find_opt x env.global.storage_types with
| Some (ty, _) ->
    (* TODO: check that the keyword is a variable. *)
    ty |: TypingRule.LEGlobalVar
```

8.3.4 Formally

8.3.5 Comments

This is related to $\mathtt{R}_{\mathtt{WDGQ}}$.

8.4 TypingRule.LEDestructuring

8.4.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes a tuple `les`;
- `t_e` has the structure of a tuple type `sub_tys`;
- the elements of `sub_tys` can be assigned to the type of the elements of `les`;
- One of the following applies:
  - ∗ All of the following apply:
    - – `les` and `sub_tys` have the same length;

    - new_le is the result of annotating les with sub_tys in env
  * All of the following apply:
    - les and sub_tys do not have the same length;
    - an error "Bad Arity LEDestructuring" is raised.

8.4.2 Example

8.4.3 Code

```
| LE_Destructuring les ->
    (match t_e.desc with
    | T_Tuple sub_tys ->
        if List.compare_lengths sub_tys les != 0 then
          Error.fatal_from le
            (Error.BadArity
               ("LEDestructuring", List.length sub_tys, List.length les))
        else
          let les' = List.map2 (annotate_lexpr env) les sub_tys in
          LE_Destructuring les' |> here
    | _ -> conflict le [ T_Tuple [] ] t_e)
    |: TypingRule.LEDestructuring
```

8.4.4 Formally

8.4.5 Comments

8.5 TypingRule.LESlice

8.5.1 Prose

Annotating le in an environment env, assuming t_e to be the type of the corresponding right-hand-side (annotate_lexpr version env le t_e), results in new_le and all of the following apply:

- le denotes the slicing of a left-hand-side expression le1 by the slices slices;

- t_le1 is the type result of annotating the right-hand-side expression corresponding to le1 in env;

- t_le1 has the structure of a bitvector type;

- le2 is the result of annotating le1 in env;

- width is the width of the slices slices in env;

- t is the bitvector type of width width;

- `te` can be assigned to `t`;
- `slices2` is the result of annotating `slices` in `env`;
- `new_le` is the slicing of `le2` by `slices2`.

8.5.2 Example

8.5.3 Code

```
match struct_t_le1.desc with
| T_Bits _ ->
    let le2 = annotate_lexpr env le1 t_le1 in
    let+ () =
     fun () ->
      let width = slices_width env slices |> reduce_expr env in
      let t = T_Bits (width, []) |> here in
      check_can_assign_to le env t t_e ()
    in
    let slices2 = best_effort slices (annotate_slices env) in
    LE_Slice (le2, slices2) |> here |: TypingRule.LESlice
```

8.5.4 Formally

8.5.5 Comments

8.6 TypingRule.LESetArray

8.6.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes the slicing of a left-hand-side expression `le1` by the slices `slices`;
- `t_le1` is the type result of annotating the right-hand-side expression corresponding to `le1` in `env`;
- `t_le1` has the structure of an array type of size `size` and item type `t`;
- `te` can be assigned to `t`;
- `le2` is the result of annotating `le1` in `env`;
- One of the following applies:
  - ∗ `wanted_t_index` is an enumeration type of name `size`;

  * `wanted_t_index` is the type `integer 0..size-1`;

- `slices` is a single expression `e_index`;
- `t_index', e_index'` is the result of annotating `e_index` in `env`;
- `wanted_t_index` can be assigned to `t_index'`;
- `new_le` is an access to array `le2` at index `e_index'`.

8.6.2 Example

8.6.3 Code

```
| T_Array (size, t) -> (
    let le2 = annotate_lexpr env le1 t_le1 in
    let+ () = check_can_assign_to le2 env t t_e in
    let wanted_t_index =
      let t_int =
        T_Int (Some [ Constraint_Range (!$0, binop MINUS size !$1) ])
        |> here
      in
      match size.desc with
      | E_Var name -> (
          match IMap.find_opt name env.global.declared_types with
          | Some t -> t
          | None -> t_int)
      | _ -> t_int
    in
    match slices with
    | [ Slice_Single e_index ] ->
        let t_index', e_index' = annotate_expr env e_index in
        let+ () =
          check_type_satisfies le2 env t_index' wanted_t_index
        in
        LE_SetArray (le2, e_index') |> here |: TypingRule.LESetArray
```

8.6.4 Formally

8.6.5 Comments

8.7 TypingRule.LESetBadStructuredField

8.7.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- `t_le1` is the type result of annotating the right-hand-side expression corresponding to `le1` in `env`;
- `le2` is the result of annotating `le1` in `env`;
- `t_le1` has the structure of an exception or a record type with fields `fields`;
- `field` is not declared in `fields`;
- an error "`Bad Field`" is raised.

8.7.2 Example

8.7.3 Code

```
| None ->
    fatal_from le (Error.BadField (field, t_le1))
    |: TypingRule.LESetBadStructuredField
```

8.7.4 Formally

8.7.5 Comments

8.8 TypingRule.LESetStructuredField

8.8.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- `t_le1` is the type result of annotating the right-hand-side expression corresponding to `le1` in `env`;
- `le2` is the result of annotating `le1` in `env`;
- `t_le1` has the structure of an exception or a record type with fields `fields`;
- `field` is bound to type `t` in `fields`;
- `t` can be assigned to `t_e`;
- `new_le` is the access to the field `field` in `le2`.

8.8.2 Example

8.8.3 Code

```
  | Some t -> t
in
let+ () = check_can_assign_to le env t t_e in
LE_SetField (le2, field) |> here |: TypingRule.LESetStructuredField
```

8.8.4 Formally

8.8.5 Comments

8.9 TypingRule.LESetBadBitField

8.9.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- `t_le1` is the type result of annotating the right-hand-side expression corresponding to `le1` in `env`;
- `le2` is the result of annotating `le1` in `env`;
- `t_le1` has the structure of a bitvector with bitfields `bitfields`;
- `field` is not declared in `bitfields`;
- an error "`Bad Field`" is raised.

8.9.2 Example

8.9.3 Code

```
| None ->
    fatal_from le1 (Error.BadField (field, t_le1_struct))
    |: TypingRule.LESetBadBitField
```

8.9.4 Formally

8.9.5 Comments

8.10 TypingRule.LESetBitField

8.10.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- `t_le1` is the type result of annotating the right-hand-side expression corresponding to `le1` in `env`;
- `le2` is the result of annotating `le1` in `env`;
- `t_le1` has the structure of a bitvector with bitfields `bitfields`;
- `field` is declared in `bitfields`;
- `slices` gives the slices corresponding to the bitfield `field` in `bitfields`;
- `w` is the width of `slices`;
- `t` is the bitvector type of width `w`;
- `t` can be assigned to `t_e`;
- `le2` is the slicing of `le1` by `slices`;
- `new_le` is the result of annotating `le2` in `env`.

8.10.2 Example

8.10.3 Code

```
| Some (BitField_Simple (_field, slices)) ->
    (bits slices [], slices) |: TypingRule.LESetBitField
```

8.10.4 Formally

8.10.5 Comments

8.11 TypingRule.LESetBitFieldNested

8.11.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- `t_le1` is the type result of annotating the right-hand-side expression corresponding to `le1` in `env`;
- `le2` is the result of annotating `le1` in `env`;
- `t_le1` has the structure of a bitvector with bitfields `bitfields`;
- `slices` gives the slices corresponding to the bitfield `field` in `bitfields`;
- `w` is the width of `slices`;
- `bitfields'` gives the bitfields corresponding to `field` in `bitfields`;
- `t` is the bitvector type of width `w` and bitfields `bitfields'`;
- `t` can be assigned to `t_e`;
- `le2` is the slicing of `le1` by `slices`;
- `new_le` is the result of annotating `le2` in `env`.

8.11.2 Example

8.11.3 Code

```
| Some (BitField_Nested (_field, slices, bitfields')) ->
    (bits slices bitfields', slices)
    |: TypingRule.LESetBitFieldNested
```

8.11.4 Formally

8.11.5 Comments

8.12 TypingRule.LESetBitFieldTyped

8.12.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- `t_le1` is the type result of annotating the right-hand-side expression corresponding to `le1` in `env`;
- `le2` is the result of annotating `le1` in `env`;
- `t_le1` has the structure of a bitvector with bitfields `bitfields`;

- `slices` gives the slices corresponding to the bitfield `field` in `bitfields`;
- `w` is the width of `slices`;
- `t'` is the bitvector type of width `w`;
- `t` gives the type corresponding to the bitfield `field` in `bitfields`;
- `t` can be assigned to `t'`;
- `t` can be assigned to `t_e`;
- `le2` is the slicing of `le1` by `slices`;
- `new_le` is the result of annotating `le2` in `env`.

8.12.2 Example

8.12.3 Code

```
| Some (BitField_Type (_field, slices, t)) ->
    let t' = bits slices [] in
    let+ () = check_type_satisfies le env t' t in
    (t, slices) |: TypingRule.LESetBitFieldTyped
```

8.12.4 Formally

8.12.5 Comments

8.13 TypingRule.LESetBadField

8.13.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- `t_le1` is the type result of annotating the right-hand-side expression corresponding to `le1` in `env`;
- `le2` is the result of annotating `le1` in `env`;
- `t_le1` does not have the structure of a record, or an exception or a bitvector type;
- an error "`Conflicting Types`" is raised.

8.13.2 Example

8.13.3 Code

```
| _ -> conflict le1 [ default_t_bits; T_Record []; T_Exception [] ] t_e)
|: TypingRule.LESetBadField
```

8.13.4 Formally

8.13.5 Comments

8.14 TypingRule.LEConcat

8.14.1 Prose

Annotating `le` in an environment `env`, assuming `t_e` to be the type of the corresponding right-hand-side (`annotate_lexpr version env le t_e`), results in `new_le` and all of the following apply:

8.14.2 Example

8.14.3 Code

```
| LE_Concat (les, _) ->
    let e_eq = expr_of_lexpr le in
    let t_e_eq, _e_eq = annotate_expr env e_eq in
    let+ () = check_bits_equal_width' env t_e_eq t_e in
    let bv_length t =
      let e_width = get_bitvector_width le env t in
      match reduce_constants env e_width with
      | L_Int z -> Z.to_int z
      | _ ->
          fatal_from le @@ MismatchType ("bitvector width", [ T_Int None ])
    in
    let annotate_one (les, widths, sum) le =
      let e = expr_of_lexpr le in
      let t_e, _e = annotate_expr env e in
      let width = bv_length t_e in
      let t_e' = T_Bits (expr_of_int width, []) |> add_pos_from le in
      let le = annotate_lexpr env le t_e' in
      (le :: les, width :: widths, sum + width)
    in
    let rev_les, rev_widths, _real_width =
      List.fold_left annotate_one ([], [], 0) les
    in
    (* Here as the first check, we have _real_width == bv_length t_e *)
```

```
let les = List.rev rev_les and widths = List.rev rev_widths in
LE_Concat (les, Some widths) |> add_pos_from le |: TypingRule.LEConcat
```

8.14.4 Formally

8.14.5 Comments

Chapter 9

Typing of Slices

Annotating slices `slices` in an environment `env` (`annotate_slices env slices`) results in the pair `(offset, length)` and one of the following applies:

- TypingRule.SliceSingle (see Section 9.1),
- TypingRule.SliceLength (see Section 9.2),
- TypingRule.SliceRange (see Section 9.3),
- TypingRule.SliceStar (see Section 9.4).

9.1 TypingRule.SliceSingle

9.1.1 Prose

Annotating slices `slices` in an environment `env` (`annotate_slices env slices`) results in the pair `(offset, length)` and all of the following apply:

- `slices` gives an index `i`;
- `(offset, length)` is the result of applying TypingRule.SliceLength to `i, i+:1`.

9.1.2 Example

9.1.3 Code

```
| Slice_Single i ->
   (* LRM R_GXKG:
      The notation b[i] is syntactic sugar for b[i +: 1].
   *)
   tr_one (Slice_Length (i, !$1)) |: TypingRule.SliceSingle
```

9.1.4 Formally

9.1.5 Comments

R`GXKG`: The notation b`i` is syntactic sugar for b`i +: 1`.

9.2 TypingRule.SliceLength

9.2.1 Prose

Annotating slices `slices` in an environment `env` (`annotate_slices env slices`) results in the pair `(offset, length)` and all of the following apply:

- `slices` gives `offset` and `length`;
- `t_offset, offset'` is the result of annotating `offset` in `env`;
- `t_length, length'` is the result of annotating `length` in `env`;
- `t_offset` has the structure of an integer type;
- `t_length` has the structure of an integer type;
- `length` is statically evaluable.

9.2.2 Example

9.2.3 Code

```
| Slice_Length (offset, length) ->
    let t_offset, offset' = annotate_expr env offset
    and t_length, length' = annotate_expr env length in
    let+ () = check_structure_integer offset' env t_offset in
    let+ () = check_structure_integer length' env t_length in
    let+ () = check_statically_evaluable env length in
    (* TODO: if offset is statically evaluable, check that it is less
       than sliced expression width. *)
    Slice_Length (offset', length') |: TypingRule.SliceLength
```

9.2.4 Formally

9.2.5 Comments

9.3 TypingRule.SliceRange

9.3.1 Prose

Annotating slices `slices` in an environment `env` (`annotate_slices env slices`) results in the pair `(offset, length)` and all of the following apply:

- slices gives a range (j, i);
- pre_length is i +: j-i+1;
- offset, length is the result of applying TypingRule.SliceLength to i,pre_length.

9.3.2 Example

9.3.3 Code

```
| Slice_Range (j, i) ->
    (* LRM R_GXKG:
       The notation b[j:i] is syntactic sugar for b[i +: j-i+1].
    *)
    let pre_length = binop MINUS j i |> binop PLUS !$1 in
    tr_one (Slice_Length (i, pre_length)) |: TypingRule.SliceRange
```

9.3.4 Formally

9.3.5 Comments

R_{GXKG}: The notation bj:i is syntactic sugar for bi +: j-i+1.

9.4 TypingRule.SliceStar

9.4.1 Prose

Annotating slices slices in an environment env (annotate_slices env slices) results in the pair (offset, length) and all of the following apply:

- slices gives (factor, pre_length);
- pre_offset is factor * pre_length;
- offset, length is the result of applying TypingRule.SliceLength to (pre_offset, pre_length).

9.4.2 Example

9.4.3 Code

```
| Slice_Star (factor, pre_length) ->
    (* LRM R_GXQG:
       The notation b[i *: n] is syntactic sugar for b[i*n +: n]
    *)
    let pre_offset = binop MUL factor pre_length in
    tr_one (Slice_Length (pre_offset, pre_length)) |: TypingRule.SliceStar
```

9.4.4 Formally

9.4.5 Comments

$\mathtt{R}_{\mathtt{GXQG}}$: The notation b`i *: n` is syntactic sugar for b`i*n +: n`

Chapter 10

Typing of Patterns

Annotating a pattern t in an environment env given a type t (annotate_pattern) results in a pattern new_p and one of the following applies:

- TypingRule.PAll (see Section 10.1),
- TypingRule.PAny (see Section 10.2),
- TypingRule.PGeq (see Section 10.3),
- TypingRule.PLeq (see Section 10.4),
- TypingRule.PNot (see Section 10.5),
- TypingRule.PRange (see Section 10.6),
- TypingRule.PSingle (see Section 10.7),
- TypingRule.PMask (see Section 10.8),
- TypingRule.PTupleBadArity (see Section 10.9),
- TypingRule.PTuple (see Section 10.10),
- TypingRule.PTupleConflict (see Section 10.11),

10.1 TypingRule.PAll

10.1.1 Prose

Annotating a pattern t in an environment env given a type t (annotate_pattern) results in a pattern new_p and all of the following apply:

- p is the pattern matching everything;
- new_p is p.

10.1.2 Example

10.1.3 Code

```
| Pattern_All as p -> p |: TypingRule.PAll
```

10.1.4 Formally

10.1.5 Comments

10.2 TypingRule.PAny

10.2.1 Prose

Annotating a pattern `t` in an environment `env` given a type `t` (`annotate_pattern`) results in a pattern `new_p` and all of the following apply:

- `p` is the pattern which matches anything in a list `li`;
- `new_li` is the result of mapping the result of annotating `p` in `env` onto `li`;
- `new_p` is the pattern which matches anything in `new_li`.

10.2.2 Example

10.2.3 Code

```
| Pattern_Any li ->
    let new_li = List.map (annotate_pattern loc env t) li in
    Pattern_Any new_li |: TypingRule.PAny
```

10.2.4 Formally

10.2.5 Comments

10.3 TypingRule.PGeq

10.3.1 Prose

Annotating a pattern `t` in an environment `env` given a type `t` (`annotate_pattern`) results in a pattern `new_p` and all of the following apply:

- `p` is the pattern which matches anything greater than or equal to an expression `e`;
- `t_e, e'` is the result of annotating `e` in `env`;
- `e'` is a compile-time constant expression;
- One of the following applies:

* All of the following apply:
  - both t and t_e have the structure of an integer;
  - new_p is the pattern which matches anything greater than or equal to e'.
* All of the following apply:
  - both t and t_e have the structure of a real;
  - new_p is the pattern which matches anything greater than or equal to e'.
* an error "Conflicting Types" is raised.

10.3.2 Example

10.3.3 Code

```
| Pattern_Geq e ->
    let t_e, e' = annotate_expr env e in
    let+ () = check_statically_evaluable env e' in
    let+ () =
      both (* TODO: case where they are both real *)
        (check_structure_integer loc env t)
        (check_structure_integer loc env t_e)
    in
    Pattern_Geq e' |: TypingRule.PGeq
```

10.3.4 Formally

10.3.5 Comments

10.4 TypingRule.PLeq

10.4.1 Prose

Annotating a pattern t in an environment env given a type t (annotate_pattern) results in a pattern new_p and all of the following apply:

- p is the pattern which matches anything lesser than or equal to an expression e;
- t_e, e' is the result of annotating e in env;
- e' is a compile-time constant expression;
- One of the following applies:
  * All of the following apply:
    - both t and t_e have the structure of an integer;

    - new_p is the pattern which matches anything lesser than or equal to e'.
  * All of the following apply:
    - both t and t_e have the structure of a real;
    - new_p is the pattern which matches anything lesser than or equal to e'.
  * an error "Conflicting Types" is raised.

10.4.2 Example

10.4.3 Code

```
| Pattern_Leq e ->
    let t_e, e' = annotate_expr env e in
    let+ () = check_statically_evaluable env e' in
    let+ () =
      both  (* TODO: case where they are both real *)
        (check_structure_integer loc env t)
        (check_structure_integer loc env t_e)
    in
    Pattern_Leq e' |: TypingRule.PLeq
```

10.4.4 Formally

10.4.5 Comments

10.5 TypingRule.PNot

10.5.1 Prose

Annotating a pattern t in an environment env given a type t (annotate_pattern) results in a pattern new_p and all of the following apply:

- p is the pattern which matches the negation of a pattern q;
- new_q is the result of annotating q in env;
- new_p is pattern which matches the negation of new_q.

10.5.2 Example

10.5.3 Code

```
| Pattern_Not q ->
    let new_q = annotate_pattern loc env t q in
    Pattern_Not new_q |: TypingRule.PNot
```

10.5.4 Formally

10.5.5 Comments

10.6 TypingRule.PRange

10.6.1 Prose

Annotating a pattern `t` in an environment `env` given a type `t` (`annotate_pattern`) results in a pattern `new_p` and all of the following apply:

- `p` is the pattern which matches anything within the range given by expressions `e1` and `e2`;
- `t_e1, e1'` is the result of annotating `e1` in `env`;
- `t_e2, e2'` is the result of annotating `e2` in `env`;
- e1' and e2' are compile-time constant expressions;
- One of the following applies:
  - ∗ All of the following apply:
    - – both `t_e1` and `t_e2` have the structure of an integer;
    - – `new_p` is the pattern which matches anything within the range given by expressions `e1'` and `e2'`.
  - ∗ All of the following apply:
    - – both `t_e1` and `t_e2` have the structure of a real;
    - – `new_p` is the pattern which matches anything within the range given by expressions `e1'` and `e2'`.
  - ∗ an error "`Conflicting Types`" is raised.

10.6.2 Example

10.6.3 Code

```
| Pattern_Range (e1, e2) ->
    let t_e1, e1' = annotate_expr env e1
    and t_e2, e2' = annotate_expr env e2 in
    let+ () =
      both (* TODO: case where they are both real *)
        (check_structure_integer loc env t)
        (both
           (check_structure_integer loc env t_e1)
           (check_structure_integer loc env t_e2))
    in
    Pattern_Range (e1', e2') |: TypingRule.PRange
```

10.6.4 Formally

10.6.5 Comments

10.7 TypingRule.PSingle

10.7.1 Prose

Annotating a pattern t in an environment env given a type t (annotate_pattern) results in a pattern new_p and all of the following apply:

- p is the pattern that matches the expression e;
- t_e, e' is the result of annotating the expression e in env;
- One of the following applies:
  - * All of the following apply:
    - – t_e has the structure of the real type;
    - – t has the structure of the real type;
  - * All of the following apply:
    - – t_e has the structure of the boolean type;
    - – t has the structure of the boolean type;
  - * All of the following apply:
    - – t_e has the structure of an integer type;
    - – t has the structure of an integer type;
  - * All of the following apply:
    - – t_e has the structure of a bitvector type;
    - – t has the structure of a bitvector type;
    - – the bitvector types t_e and t have the same length;
  - * All of the following apply:
    - – t_e has the structure of an enumeration type;
    - – t has the structure of an enumeration type;
    - – the enumeration types t_e and t have the same literals;
- new_p is p;

10.7.2 Example

10.7.3 Code

```
| Pattern_Single e ->
    let t_e, e = annotate_expr env e in
    let+ () =
     fun () ->
```

```
    let t_struct = Types.get_structure env t
    and t_e_struct = Types.get_structure env t_e in
    match (t_struct.desc, t_e_struct.desc) with
    | T_Bool, T_Bool | T_Real, T_Real -> ()
    | T_Int _, T_Int _ -> ()
    | T_Bits _, T_Bits _ ->
        check_bits_equal_width loc env t_struct t_e_struct ()
    (* TODO: Multiple discriminants can be matched at once by forming
       a tuple of discriminants and a tuple used in the pattern_set.
       Both tuples must have the same number of elements. A
       successful pattern match occurs when each discriminant term
       matches the respective term of the pattern tuple. *)
    | T_Enum li1, T_Enum li2 when list_equal String.equal li1 li2 -> ()
    | _ -> fatal_from loc (Error.BadTypesForBinop (EQ_OP, t, t_e))
  in
  Pattern_Single e |: TypingRule.PSingle
```

10.7.4 Formally

10.7.5 Comments

10.8 TypingRule.PMask

10.8.1 Prose

Annotating a pattern `t` in an environment `env` given a type `t` (`annotate_pattern`) results in a pattern `new_p` and all of the following apply:

- `p` is the pattern which matches a mask `m`;
- `t` has the structure of a bitvector type;
- `n` is the length of mask `m`;
- `t_m` is the bitvector type of width `n`;
- One of the following applies:
- All of the following apply:
  - `t` type-satisfies `t_m`;
  - `new_p` is `p`.
- an error "`Conflicting Types`" is raised.

10.8.2 Example

10.8.3 Code

```
| Pattern_Mask m as p ->
    let+ () = check_structure_bits loc env t in
    let+ () =
      let n = !$(Bitvector.mask_length m) in
      let t_m = T_Bits (n, []) |> add_pos_from loc in
      check_type_satisfies loc env t t_m
    in
    p |: TypingRule.PMask
```

10.8.4 Formally

10.8.5 Comments

This is related to $\texttt{I}_{\texttt{VMKF}}$.

10.9 TypingRule.PTupleBadArity

10.9.1 Prose

Annotating a pattern `t` in an environment `env` given a type `t` (`annotate_pattern`) results in a pattern `new_p` and all of the following apply:

- `p` is the pattern which matches a tuple `li`;
- `t` has the type structure of a tuple type `ts`;
- `ts` is a list of different size to the size of `li`;
- an error "`Bad Arity`" is raised.

10.9.2 Example

10.9.3 Code

```
| T_Tuple ts when List.compare_lengths li ts != 0 ->
    Error.fatal_from loc
      (Error.BadArity
         ("pattern matching on tuples", List.length li, List.length ts))
    |: TypingRule.PTupleBadArity
```

10.9.4 Formally

10.9.5 Comments

10.10 TypingRule.PTuple

10.10.1 Prose

Annotating a pattern `t` in an environment `env` given a type `t` (`annotate_pattern`) results in a pattern `new_p` and all of the following apply:

- `p` is the pattern which matches a tuple `li`;
- `t` has the type structure of a tuple type `ts`;
- `ts` is a list of the same size as `li`;
- `new_li` is the result of annotating `li` with `ts`;
- `new_p` is the pattern which matches the tuple `new_li`.

10.10.2 Example

10.10.3 Code

```
| T_Tuple ts ->
    let new_li = List.map2 (annotate_pattern loc env) ts li in
    Pattern_Tuple new_li |: TypingRule.PTuple
```

10.10.4 Formally

10.10.5 Comments

10.11 TypingRule.PTupleConflict

10.11.1 Prose

Annotating a pattern `t` in an environment `env` given a type `t` (`annotate_pattern`) results in a pattern `new_p` and all of the following apply:

- `p` is the pattern which matches a tuple `li`;
- `t` has the type structure of a tuple type `ts`;
- `t_struct` is not a tuple type;
- an error "`Conflicting Types`" is raised.

10.11.2 Example

10.11.3 Code

```
| _ -> conflict loc [ T_Tuple [] ] t |: TypingRule.PTupleConflict
```

10.11.4 Formally

10.11.5 Comments

Chapter 11

Typing of Local Declarations

Annotating a local declaration `ldi`, given a type `ty`, in an environment `env` results in `new_env, new_ldi` (`annotate_local_decl_item`) and one of the following applies:

- TypingRule.LDDiscardNone (see Section 11.1),
- TypingRule.LDDiscardSome (see Section 11.2),
- TypingRule.LDVar (see Section 11.3),
- TypingRule.LDTuple (see Section 11.4).

This is related to $\texttt{R}_{\texttt{YSPM}}$.

11.1 TypingRule.LDDiscardNone

11.1.1 Prose

Annotating a local declaration `ldi`, given a type `ty`, in an environment `env` results in `new_env, new_ldi` and all of the following apply:

- `ldi` is a local declaration which can be discarded;
- `ldi` does not specify a type;
- `new_env` is `env`;
- `new_ldi` is `ldi`.

11.1.2 Example

11.1.3 Code

```
| LDI_Discard None -> (env, ldi) |: TypingRule.LDDiscardNone
```

11.1.4 Formally

11.1.5 Comments

11.2 TypingRule.LDDiscardSome

11.2.1 Prose

Annotating a local declaration `ldi`, given a type `ty`, in an environment `env` results in `new_env, new_ldi` and all of the following apply:

- `ldi` is a local declaration which can be discarded;
- `ldi` specifies a type `t`;
- One of the following applies:
  - ∗ All of the following apply:
    - – `t` can be initialised with `ty` in `env`;
    - – `new_env` is `env`;
    - – `new_ldi` is `ldi`.
  - ∗ All of the following apply:
    - – `t` cannot be initialised with `ty` in `env`;
    - – an error "`Conflicting Types`" is raised.

11.2.2 Example

11.2.3 Code

```
| LDI_Discard (Some t) ->
    let+ () = check_can_be_initialized_with loc env t ty in
    (env, ldi) |: TypingRule.LDDiscardSome
```

11.2.4 Formally

11.2.5 Comments

11.3 TypingRule.LDVar

11.3.1 Prose

Annotating a local declaration `ldi`, given a type `ty`, in an environment `env` results in `new_env, new_ldi` and all of the following apply:

- `ldi` denotes a variable `x` with an optional type `ty_opt`;
- `x` is not declared in `env`;
- One of the following applies:
  - ∗ All of the following apply:
    - `ty_opt` is `None`;
    - `t` is `ty`
  - ∗ All of the following apply:
    - `ty_opt` is `Some t`;
    - `t` can be initialized with `ty` in `env`;
- `new_env` is `env` modified so that `x` is locally declared of type `t`;
- `new_ldi` is the declaration of variable `x` with type `t`.

11.3.2 Example

11.3.3 Code

```
| LDI_Var (x, ty_opt) ->
    let t =
      best_effort ty @@ fun _ ->
      match ty_opt with
      | None -> ty
      | Some t ->
          let+ () = check_can_be_initialized_with loc env t ty in
          t
    in
    (* Rule LCFD: A local declaration shall not declare an identifier
       which is already in scope at the point of declaration. *)
    let+ () = check_var_not_in_env loc env x in
    let new_env = add_local x t ldk env in
    (new_env, LDI_Var (x, Some t)) |: TypingRule.LDVar
| LDI_Tuple ([ ld ], None) ->
  (* TODO: this is prohibited *)
    annotate_local_decl_item loc env ty ldk ld
    |: TypingRule.LDUninitialisedTypedTuple
```

11.3.4 Formally

11.3.5 Comments

This is related to $\texttt{R}_{\texttt{YSPM}}$, $\texttt{D}_{\texttt{FXST}}$.

11.4 TypingRule.LDTuple

11.4.1 Prose

Annotating a local declaration ldi, given a type ty, in an environment env results in new_env, new_ldi and all of the following apply:

- ldi denotes a list ldis;
- ldi does not specify a type;
- ty has the structure of a tuple type of the same length as ldis;
- new_env is env modified so that each element in ldis is annotated with the corresponding type in ty;
- new_ldi is ldis where each element is declared with the corresponding type in ty.

11.4.2 Example

11.4.3 Code

```
| LDI_Tuple (ldis, None) ->
   let tys =
     match (Types.get_structure env ty).desc with
     | T_Tuple tys when List.compare_lengths tys ldis = 0 -> tys
     | T_Tuple tys ->
         fatal_from loc
           (Error.BadArity
              ("tuple initialization", List.length tys, List.length ldis))
     | _ -> conflict loc [ T_Tuple [] ] ty
   in
   let new_env, new_ldi =
     List.fold_right2
       (fun ty' ldi' (env', les) ->
         let env', le = annotate_local_decl_item loc env' ty' ldk ldi' in
         (env', le :: les))
       tys ldis (env, [])
   in
   (new_env, LDI_Tuple (new_ldi, None)) |: TypingRule.LDTuple
```

11.4.4 Formally

11.4.5 Comments

Chapter 12

Typing of Statements

Annotating a statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and one of the following applies:

- TypingRule.SPass (see Section 12.1),
- TypingRule.SAssign (see Section 12.2),
- TypingRule.SReturnNone (see Section 12.3),
- TypingRule.SReturnOne (see Section 12.4),
- TypingRule.SReturnSome (see Section 12.5),
- TypingRule.SSeq (see Section 12.6),
- TypingRule.SCall (see Section 12.7),
- TypingRule.SCond (see Section 12.8),
- TypingRule.SCase (see Section 12.9),
- TypingRule.SAssert (see Section 12.10),
- TypingRule.SWhile (see Section 12.11),
- TypingRule.SRepeat (see Section 12.12),
- TypingRule.SFor (see Section 12.13),
- TypingRule.SThrowNone (see Section 12.14),
- TypingRule.SThrowSome (see Section 12.15),
- TypingRule.STry (see Section 12.16).
- TypingRule.SDeclSome (see Section 12.17),
- TypingRule.SDeclNone (see Section 12.18),

12.1 TypingRule.SPass

12.1.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a pass statement;
- `new_s` is `s`;
- `new_env` is `env`.

12.1.2 Example

12.1.3 Code

```
| S_Pass -> (s, env) |: TypingRule.SPass
```

12.1.4 Formally

12.1.5 Comments

12.2 TypingRule.SAssign

12.2.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is an assignment `le = re` under language version `ver`;
- `t_e, e1` is the result of annotating `re` in `env`;
- `reduced` is the result of inlining a setter call in `le`;
- One of the following applies:
  - ∗ All of the following apply:
    - – `reduced` gives a statement `s`;
    - – `new_s` is `s`;
    - – `new_env` is `env`.
  - ∗ All of the following apply:
    - – `reduced` does not give a statement `s`;
    - – One of the following applies:
      - ▷ All of the following apply:
        - \+ `ver` is ASLv1;
        - \+ `env1` is `env`;

      ▷ All of the following apply:
        + `ver` is ASLv0;
        + `env1` is the result of annotating undeclared variables by using the first assignments to such variables as declarations;
  – `le1` is the result of annotating `le` with `t_e` in `env1`;
  – `new_s` is the assignment `le1 = e1`;
  – `new_env` is `env1`.

12.2.2 Example

12.2.3 Code

```
| S_Assign (le, re, ver) ->
    (let () =
       if false then
         Format.eprintf "@[<3>Annotating assignment@ @[%a@]@]@." PP.pp_stmt
           s
     in
     let t_e, e1 = annotate_expr env re in
     let () =
       if false then Format.eprintf "@[Type: @[%a@]@]@." PP.pp_ty t_e
     in
     let reduced = setter_should_reduce_to_call_s env le e1 in
     match reduced with
     | Some s -> (s, env)
     | None ->
         let env1 =
           match ver with
           | V1 -> env
           | V0 -> (
               (*
                * In version V0, variable declaration is optional,
                * As a result typing will be partial and some
                * function calls may lack extra parameters.
                * Fix this by typing first assignments of
                * undeclared variables as declarations.
                *)
               match ASTUtils.lid_of_lexpr le with
               | None -> env
               | Some ldi ->
                   let rec undefined = function
                     | LDI_Discard _ -> true
                     | LDI_Var (x, _) -> StaticEnv.is_undefined x env
                     | LDI_Tuple (ldis, _) -> List.for_all undefined ldis
                   in
```

```
                    if undefined ldi then
                      let () =
                        if false then
                          Format.eprintf
                            "@[<3>Assignment@ @[%a@] as declaration@]@."
                            PP.pp_stmt s
                      in
                      let ldk = LDK_Var in
                      let env2, _ldi =
                        annotate_local_decl_item s env t_e ldk ldi
                      in
                      env2
                    else env)
          in
          let le1 = annotate_lexpr env1 le t_e in
          (S_Assign (le1, e1, ver) |> here, env1))
     |: TypingRule.SAssign
```

12.2.4 Formally

12.2.5 Comments

12.3 TypingRule.SReturnNone

12.3.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a `return` statement with no value and no return type;
- `new_s` is a `return` statement with no value;
- the enclosing subprogram does not have a `return` type (it is either a setter or a procedure);
- `new_env` is `env`.

12.3.2 Example

12.3.3 Code

```
     | None, None -> (S_Return None |> here, env) |: TypingRule.SReturnNone
```

12.3.4 Formally

12.3.5 Comments

This is related to $\mathrm{R}_{\mathtt{FTPK}}$.

12.4 TypingRule.SReturnOne

12.4.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- One of the following applies:
  - ∗ All of the following apply:
    - – `s` is a `return` statement with some value;
    - – the enclosing subprogram does not have a return type;
  - ∗ All of the following apply:
    - – `s` is a `return` statement with no value;
    - – the enclosing subprogram has a `return` type;
- an error "`Bad Return Statement`" is raised.

12.4.2 Example

12.4.3 Code

```
| None, Some _ | Some _, None ->
    fatal_from s (Error.BadReturnStmt env.local.return_type)
    |: TypingRule.SReturnOne
```

12.4.4 Formally

12.4.5 Comments

This is related to $\mathtt{R}_{\mathtt{FTPK}}$.

12.5 TypingRule.SReturnSome

12.5.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a `return` statement with some value `e`;
- the enclosing subprogram has a return type `t`;
- `t_e',e'` is the result of annotating `e` in `env`;
- One of the following applies:
  - ∗ All of the following apply:

  - t_e' type-satisfies t;
  - new_s is a return statement with value e';
  - new_env is env.

* an error "Conflicting Types" is raised.

12.5.2 Example

12.5.3 Code

```
| Some t, Some e ->
    let t_e', e' = annotate_expr env e in
    let () =
      if false then
        Format.eprintf
          "Can I return %a(of type %a) when return_type = %a?@."
          PP.pp_expr e PP.pp_ty t_e' PP.pp_ty t
    in
    let+ () = check_type_satisfies s env t_e' t in
    (S_Return (Some e') |> here, env))
|: TypingRule.SReturnSome
```

12.5.4 Formally

12.5.5 Comments

This is related to $\mathrm{R}_{\mathrm{FTPK}}$.

12.6 TypingRule.SSeq

12.6.1 Prose

Annotating statement s in an environment env (annotate_stmt env s) results in a statement new_s and an environment new_env and all of the following apply:

- s is a statement s1; s2;
- new_s1, env1 is the result of annotating s1 in env;
- new_s2, env2 is the result of annotating s2 in env1;
- new_s is a then statement over two statements new_s1 and new_s2;
- new_env is env2.

12.6.2 Example

12.6.3 Code

```
| S_Seq (s1, s2) ->
    let new_s1, env1 = try_annotate_stmt env s1 in
    let new_s2, env2 = try_annotate_stmt env1 s2 in
    (S_Seq (new_s1, new_s2) |> here, env2) |: TypingRule.SSeq
```

12.6.4 Formally

12.6.5 Comments

12.7 TypingRule.SCall

12.7.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a call to a subprogram named `name` with arguments `args` and parameters `eqs`;
- `new_name, new_args, new_eqs` is the result of annotating the call to the procedure `name` with arguments `args` and parameters `eqs`;
- `new_s` is the call to a subprogram named `new_name` with arguments `new_args` and parameters `new_eqs`;
- `new_env` is `env`.

12.7.2 Example

12.7.3 Code

```
| S_Call (name, args, eqs) ->
    let new_name, new_args, new_eqs, ty =
      annotate_call (to_pos s) env name args eqs ST_Procedure
    in
    let () = assert (ty = None) in
    (* TODO: check that call does not returns anything. *)
    (S_Call (new_name, new_args, new_eqs) |> here, env) |: TypingRule.SCall
```

12.7.4 Formally

12.7.5 Comments

This is related to $\texttt{D}_{\texttt{VXKM}}$.

12.8 TypingRule.SCond

12.8.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a condition `e` with two statements `s1` and `s2`;
- `t_cond, e_cond` is the result of annotating `e` in `env`;
- One of the following applies:
  - ∗ All of the following apply:
    - – `t_cond` type-satisfies `t_bool`;
    - – `s1'` is the result of annotating `s1` in `env`;
    - – `s2'` is the result of annotating `s2` in `env`;
    - – `new_s` is the condition `e_cond` with two statements `s1'` and `s2'`;
    - – `new_env` is `env`.
  - ∗ an error "`Conflicting Types`" is raised.

12.8.2 Example

12.8.3 Code

```
| S_Cond (e, s1, s2) ->
    let t_cond, e_cond = annotate_expr env e in
    let+ () = check_type_satisfies e_cond env t_cond t_bool in
    let s1' = try_annotate_block env s1 in
    let s2' = try_annotate_block env s2 in
    (S_Cond (e_cond, s1', s2') |> here, env) |: TypingRule.SCond
```

12.8.4 Formally

12.8.5 Comments

This is related to $\texttt{R}_{\texttt{NBDJ}}$.

12.9 TypingRule.SCase

12.9.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a case statement with expression `e` and cases `cases`;
- `t_e, e1` is the result of annotating `e` in `env`;

- cases1, env1 is the result of annotating each case in cases given t_e;
- new_s is a case statement with expression e1 and cases cases1;
- new_env is env1.

12.9.2 Example

12.9.3 Code

```
| S_Case (e, cases) ->
    let t_e, e1 = annotate_expr env e in
    let annotate_case (acc, env) case =
      let p, s = case.desc in
      let p1 = annotate_pattern e1 env t_e p in
      let s1 = try_annotate_block env s in
      (add_pos_from_st case (p1, s1) :: acc, env)
    in
    let cases1, env1 = List.fold_left annotate_case ([], env) cases in
    (S_Case (e1, List.rev cases1) |> here, env1) |: TypingRule.SCase
```

12.9.4 Formally

12.9.5 Comments

This is related to R_{WGSY}.

12.10 TypingRule.SAssert

12.10.1 Prose

Annotating statement s in an environment env (annotate_stmt env s) results in a statement new_s and an environment new_env and all of the following apply:

- s is an assert statement with expression e;
- t_e',e' is the result of annotating e in env;
- One of the following applies:
- All of the following apply:
  - t_e' type-satisfies t_bool;
  - new_s is an assert statement with expression e';
  - new_env is env.
- an error "Conflicting Types" is raised.

12.10.2 Example

12.10.3 Code

```
| S_Assert e ->
    let t_e', e' = annotate_expr env e in
    let+ () = check_type_satisfies s env t_e' t_bool in
    (S_Assert e' |> here, env) |: TypingRule.SAssert
```

12.10.4 Formally

12.10.5 Comments

This is related to $\texttt{R}_{\texttt{JQYF}}$.

12.11 TypingRule.SWhile

12.11.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a `while` statement with expression `e1` and statement block `s1`;
- `t, e2` is the result of annotating `e1` in `env`;
- One of the following applies:
- All of the following apply:
  - `t` type-satisfies `t_bool`;
  - `s2` is the result of annotating `s1` in `env`;
  - `new_s` is a `while` statement with expression `e2` and statement block `s2`;
  - `new_env` is `env`.
- an error "`Conflicting Types`" is raised.

12.11.2 Example

12.11.3 Code

```
| S_While (e1, s1) ->
    let t, e2 = annotate_expr env e1 in
    let+ () = check_type_satisfies e2 env t t_bool in
    let s2 = try_annotate_block env s1 in
    (S_While (e2, s2) |> here, env) |: TypingRule.SWhile
```

12.11.4 Formally

12.11.5 Comments

This is related to $\texttt{R}_{\texttt{FTVN}}$.

12.12 TypingRule.SRepeat

12.12.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a `repeat` statement with expression `e1` and statement block `s1`;
- `s2` is the result of annotating `s1` in `env`;
- `t, e2` is the result of annotating `e1` in `env`;
- One of the following applies:
  * All of the following apply:
    - `t` type-satisfies `t_bool`;
    - `new_s` is a `repeat` statement with expression `e2` and statement block `s2`;
    - `new_env` is `env`.
  * an error "`Conflicting Types`" is raised.

12.12.2 Example

12.12.3 Code

```
| S_Repeat (s1, e1) ->
    let s2 = try_annotate_block env s1 in
    let t, e2 = annotate_expr env e1 in
    let+ () = check_type_satisfies e2 env t t_bool in
    (S_Repeat (s2, e2) |> here, env) |: TypingRule.SRepeat
```

12.12.4 Formally

12.12.5 Comments

This is related to $\texttt{R}_{\texttt{FTVN}}$.

12.13 TypingRule.SFor

12.13.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a `for` statement with index `id`, direction `dir`, two expressions `e1` and `e2` and a statement block `s'`;
- `t1,e1'` is the result of annotating `e1` in `env`;
- `t2,e2'` is the result of annotating `e2` in `env`;
- an error is raised: "`ASL Typing Error : A subtype of integer was expected, t1 was provided`" or `t1` has the structure of an integer type and all of the following apply:
- an error is raised: "`ASL Typing Error : A subtype of integer was expected, t2 was provided`" or `t2` has the structure of an integer type and all of the following apply:
- One of the following applies:
  - ∗ All of the following applies:
    - – `t1` has the structure of an unconstrained integer type;
    - – `ty` is the unconstrained integer type;
  - ∗ All of the following applies:
    - – `t2` has the structure of an unconstrained integer type;
    - – `ty` is the unconstrained integer type;
  - ∗ All of the following applies:
    - – `t1` has the structure of a constrained integer type with constraint `cs1`;
    - – `t2` has the structure of a constrained integer type with constraint `cs2`;
    - – One of the following applies:
      - ▷ All of the following applies:
        - + `dir` is `to`;
        - + `bot_cs` is `cs1`;
        - + `top_cs` is `cs2`;
      - ▷ All of the following applies:
        - + `dir` is `down to`;
        - + `bot_cs` is `cs2`;
        - + `top_cs` is `cs1`;
    - – One of the following applies:

  - ▷ All of the following applies:
    - + bot_cs contains a an expression that is not evaluable at compile-time;
    - + cs is the empty constraint;
  - ▷ All of the following apply:
    - + top_cs contains a an expression that is not evaluable at compile-time;
    - + cs is the empty constraint;
  - ▷ All of the following apply:
    - + bot is the minimum of the constraints bot_cs;
    - + top is the maximum of the constraints top_cs;
    - + bot is less or equal than top;
    - + cs is the constraint bot .. top;
  - ▷ All of the following apply:
    - + bot is the minimum of the constraints bot_cs;
    - + top is the maximum of the constraints top_cs;
    - + top is strictly less than bot
    - + cs is cs1;
- – ty is the constrained integer type with constraint cs;

- an error is raised "ASL Typing Error: cannot declare already declared element \id"." or id is not bound in env and all of the following apply:
- env' is env modified so that id is locally declared of type ty;
- s'' is the result of annotating s' in env';
- new_s is a for statement with index id, direction dir, two expressions e1' and e2' and statement s'';
- new_env is env.

12.13.2 Example

12.13.3 Code

```
| S_For (id, e1, dir, e2, s') ->
    let t1, e1' = annotate_expr env e1 and t2, e2' = annotate_expr env e2 in
    let+ () = check_structure_integer s' env t1 in
    let+ () = check_structure_integer s' env t2 in
    let cs =
      match
        ( (Types.get_structure env t1).desc,
          (Types.get_structure env t2).desc )
      with
```

```
      | T_Int None, T_Int _ | T_Int _, T_Int None -> None
      | T_Int (Some cs1), T_Int (Some cs2) -> (
          try
            let bot_cs, top_cs =
              match dir with Up -> (cs1, cs2) | Down -> (cs2, cs1)
            in
            let bot = min_constraints env bot_cs
            and top = max_constraints env top_cs in
            if bot <= top then
              Some [ Constraint_Range (expr_of_z bot, expr_of_z top) ]
            else Some cs1
          with ConstraintMinMaxTop ->
            (* TODO: this case is not specified by the LRM. *)
            Some [])
      | _ -> None
      (* only happens in relaxed type-checking mode because of check_structure_inte
    in
    let ty = T_Int cs |> here in
    let s'' =
      let+ () = check_var_not_in_env s' env id in
      let env' = add_local id ty LDK_Let env in
      try_annotate_block env' s'
    in
    (S_For (id, e1', dir, e2', s'') |> here, env) |: TypingRule.SFor
```

12.13.4 Formally

12.13.5 Comments

This is related to $\texttt{R}_{\texttt{VTJW}}$.

12.14 TypingRule.SThrowNone

12.14.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a throw statement with no expression;
- `new_s` is `s`;
- `new_env` is `env`.

12.14.2 Example

12.14.3 Code

```
| S_Throw None ->
    (* TODO: verify that this is allowed? *)
    (s, env) |: TypingRule.SThrowNone
```

12.14.4 Formally

12.14.5 Comments

Note that $R_{\texttt{BRCJ}}$ is done in [2, SemanticsRule.TopLevel].

12.15 TypingRule.SThrowSome

12.15.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a throw statement with expression `e`;
- `t_e,e'` is the result of annotating `e` in `env`;
- `t_e` has the structure of an exception type;
- `new_s` is a throw statement with expression `e'` and type `t_e`;
- `new_env` is `env`.

12.15.2 Example

12.15.3 Code

```
| S_Throw (Some (e, _)) ->
    let t_e, e' = annotate_expr env e in
    let+ () = check_structure_exception s env t_e in
    (S_Throw (Some (e', Some t_e)) |> here, env) |: TypingRule.SThrowSome
```

12.15.4 Formally

12.15.5 Comments

This is related to $R_{\texttt{NXRC}}$.

12.16 TypingRule.STry

12.16.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a try statement with statement `s'`, catchers `catchers` and block `otherwise`;
- `s''` is the result of annotating `s'` in `env`;
- `otherwise'` is the result of annotating `otherwise` in `env`;
- `catchers'` is the result of annotating `catchers` in `env`;
- `new_s` is a try statement with statement `s''`, catchers `catchers'` and block `otherwise'`;
- `new_env` is `env`.

12.16.2 Example

12.16.3 Code

```
| S_Try (s', catchers, otherwise) ->
    let s'' = try_annotate_block env s' in
    let otherwise' = Option.map (try_annotate_block env) otherwise in
    let catchers' = List.map (annotate_catcher env) catchers in
    (S_Try (s'', catchers', otherwise') |> here, env) |: TypingRule.STry
```

12.16.4 Formally

12.16.5 Comments

This is related to $\texttt{R}_{\texttt{WVXS}}$.

12.17 TypingRule.SDeclSome

12.17.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a declaration with local identifiers `ldi` and an expression `e`;
- `t_e,e'` is the result of annotating `e` in `env`;
- `env', ldi'` is the result of declaring the local identifiers of `ldi` in `env`;
- `new_s` is a declaration with `ldk`, `ldi'` and an expression `e'`;
- `new_env` is `env'`.

12.17.2 Example

12.17.3 Code

```
| _, Some e ->
    let t_e, e' = annotate_expr env e in
    let env', ldi' =
      if ldk = LDK_Constant then
        let v = reduce_constants env e in
        declare_local_constant s env t_e v ldi
      else annotate_local_decl_item s env t_e ldk ldi
    in
    (S_Decl (ldk, ldi', Some e') |> here, env') |: TypingRule.SDeclSome
```

12.17.4 Formally

12.17.5 Comments

This is related to $\texttt{R}_{\texttt{YSPM}}$.

12.18 TypingRule.SDeclNone

12.18.1 Prose

Annotating statement `s` in an environment `env` (`annotate_stmt env s`) results in a statement `new_s` and an environment `new_env` and all of the following apply:

- `s` is a declaration statement with local identifiers `ldi` and no initial expression;
- `env', s'` is the result of annotating uninitialised local declarations `ldi` in `env`;
- `new_s` is `s'`;
- `new_env` is `env'`.

12.18.2 Example

12.18.3 Code

```
| LDK_Var, None ->
    let env', s' = annotate_local_decl_item_uninit s env ldi in
    (s', env') |: TypingRule.SDeclNone
| (LDK_Constant | LDK_Let), None ->
    (* by construction in Parser. *)
    assert false)
```

12.18.4 Formally

12.18.5 Comments

Chapter 13

Typing of Blocks

13.1 TypingRule.Block

13.1.1 Prose

Annotating a block `s` in an environment `env`, given a type `return_type` (`annotate_block env return_type s`, is the result of annotating the statement `s` in `env`.

13.1.2 Example: TypingRule.Block0.asl

```
func main () => integer
begin
  if UNKNOWN: boolean then
     let i = 3;
     print (i);
  end
  let i = "Some text";
  print (i);
  return 0;
end
```

13.1.3 Code

```
and try_annotate_block env s =
  (*
     See rule JFRD:
        A local identifier declared with var, let or constant is in scope
        from the point immediately after its declaration until the end of the
        immediately enclosing block.

     From that follows that we can discard the environment at the end of an
     enclosing block.
```

```
*)
best_effort s (fun _ -> annotate_stmt env s |> fst) |: TypingRule.Block
```

13.1.4 Formally

13.1.5 Comments

A local identifier declared with var, let or constant is in scope from the point immediately after its declaration until the end of the immediately enclosing block.

From that follows that we can discard the environment at the end of an enclosing block.

This is related to $\texttt{R}_{\texttt{JBXQ}}$.

Chapter 14

Typing of Catchers

Annotating catchers `(name_opt, ty, stmt)` in an environment `env` given a type `return_type` (`annotate_catchers env return_type (name_opt, ty, stmt)`) results in `(name_opt, ty, new_stmt)` and one of the following applies:

- TypingRule.CatcherNone (see Section 14.1),
- TypingRule.CatcherSome (see Section 14.2).

14.1 TypingRule.CatcherNone

14.1.1 Prose

Annotating catchers `(name_opt, ty, stmt)` in an environment `env` given a type `return_type` (`annotate_catchers env return_type (name_opt, ty, stmt)`) results in `(name_opt, ty, new_stmt)` and all of the following apply:

- `ty` has the structure of an exception type;
- `name_opt` gives no name;
- `env'` is `env`;
- `new_stmt` is the result of annotating `stmt` in `env'` with `return_type`.

14.1.2 Example

14.1.3 Code

```
| None -> env |: TypingRule.CatcherNone
```

14.1.4 Formally

14.1.5 Comments

This is related to $\texttt{R}_{\texttt{SDJK}}$.

14.2 TypingRule.CatcherSome

14.2.1 Prose

Annotating catchers `(name_opt, ty, stmt)` in an environment `env` given a type `return_type` (`annotate_catchers env return_type (name_opt, ty, stmt)`) results in `(name_opt, ty, new_stmt)` and all of the following apply:

- `ty` has the structure of an exception type;
- `name_opt` gives a name `name`;
- `name` is not already declared in `env`;
- `name` has type `ty` in `env`;
- `env'` is `env` modified to have `name` locally declared as immutable of type `ty`;
- `new_stmt` is the result of annotating `stmt` in `env'` with `return_type`.

14.2.2 Example

14.2.3 Code

```
| Some name ->
    let+ () = check_var_not_in_env stmt env name in
    add_local name ty LDK_Let env |: TypingRule.CatcherSome
```

14.2.4 Formally

14.2.5 Comments

This is related to R_{SDJK}, R_{WVXS}, I_{FCGK}.

Chapter 15

Typing of Subprogram Calls

Annotating the call to subprogram `name` with arguments `args`, parameters `eqs`, and call type `call_type` (`annotate_call`) results in `(name1, args, eqs2, ret_ty1)` or an error is raised and one of the following applies:

- TypingRule.FCallBadArity (see Section 15.1),
- TypingRule.FCallGetter (see Section 15.2),
- TypingRule.FCallSetter (see Section 15.3),
- TypingRule.FCallMismatch (see Section 15.4).

15.1 TypingRule.FCallBadArity

15.1.1 Prose

Annotating the call to subprogram `name` with arguments `args` and parameters `eqs` (`annotate_call`) results in `(name1, args, eqs2, ret_ty1)` or an error is raised and all of the following apply:

- `name` is bound in `env` to a function with argument types `callee_arg_types`;
- the lists `callee_arg_types` and `args` do not have the same length;
- an error "`Bad Arity`" is raised.

15.1.2 Example

15.1.3 Code

```
if List.compare_lengths callee_arg_types args1 != 0 then
  fatal_from loc
  @@ Error.BadArity (name, List.length callee_arg_types, List.length args1)
  |: TypingRule.FCallBadArity
```

15.1.4 Formally

15.1.5 Comments

15.2 TypingRule.FCallGetter

15.2.1 Prose

Annotating the call to subprogram `name` with arguments `args`, parameters `eqs`, and call-type `call_type` (`annotate_call`) results in `(name1, args, eqs2, ret_ty1)` or an error is raised and all of the following apply:

- `caller_arg_types, arg1` is the result of annotating `args` in `env`;
- `name` is bound in `env` to a subprogram with argument types `callee_arg_types`;
- `eqs2` is `eqs1` appended with the equations deduced by using the types of the actual arguments `caller_arg_types` to defined parameters in `callee_arg_types`;
- `call_type` is either a function or a getter type;
- `ret_ty1` is the result of renaming `ty` in `eqs2`.

15.2.2 Example

15.2.3 Code

```
| (ST_Function | ST_Getter), Some ty ->
    Some (rename_ty_eqs eqs2 ty) |: TypingRule.FCallGetter
```

15.2.4 Formally

15.2.5 Comments

This is related to $\mathrm{I}_{\mathtt{VFDP}}$, $\mathrm{D}_{\mathtt{TRFW}}$, $\mathrm{R}_{\mathtt{KMDB}}$, $\mathrm{I}_{\mathtt{YMHX}}$, $\mathrm{R}_{\mathtt{CCVD}}$, $\mathrm{R}_{\mathtt{QYBH}}$, $\mathrm{R}_{\mathtt{PFWQ}}$, $\mathrm{R}_{\mathtt{ZLWD}}$, $\mathrm{I}_{\mathtt{FLKF}}$, $\mathrm{D}_{\mathtt{PMBL}}$, $\mathrm{R}_{\mathtt{MWBN}}$, $\mathrm{R}_{\mathtt{TZSP}}$, $\mathrm{R}_{\mathtt{SBWR}}$, $\mathrm{I}_{\mathtt{CMLP}}$, $\mathrm{R}_{\mathtt{BQJG}}$, $\mathrm{R}_{\mathtt{RTCF}}$.

15.3 TypingRule.FCallSetter

15.3.1 Prose

Annotating the call to subprogram `name` with arguments `args`, parameters `eqs`, and call-type `call_type` (`annotate_call`) results in `(name1, args, eqs2, ret_ty1)` or an error is raised and all of the following apply:

- `caller_arg_types, arg1` is the result of annotating `args` in `env`;
- `name` is bound in `env` to a subprogram with a unique name `name1` whose argument types `callee_arg_types` type-clash `caller_arg_types` and whose return type is `ret_ty`;

- `eqs1` is the list made of both `eqs` and `extra_nargs`;
- `eqs2` is `eqs1` appended with the equations deduced by using the types of the actual arguments `caller_arg_types` to defined parameters in `callee_arg_types`;
- `call_type` is a setter or procedure type;
- `ret_ty` is None;
- `ret_ty1` is None.

15.3.2 Example

15.3.3 Code

```
| (ST_Setter | ST_Procedure), None -> None |: TypingRule.FCallSetter
```

15.3.4 Formally

15.3.5 Comments

This is related to I_{VFDP}, D_{TRFW}, R_{KMDB}, I_{YMHX}, R_{CCVD}, R_{QYBH}, R_{PFWQ}, R_{ZLWD}, I_{FLKF}, D_{PMBL}, R_{MWBN}, R_{TZSP}, R_{SBWR}, I_{CMLP}, R_{RTCF}.

15.4 TypingRule.FCallMismatch

15.4.1 Prose

Annotating the call to subprogram `name` with call type `call_type` (`annotate_call`) results in an error and one of the following apply:

- All of the following apply:
  - `call_type` is a function or a getter;
  - `name` is bound in `env` a subprogram without a return-type;
  - A “Mismatched return value” error is raised.
- All of the following apply:
  - `call_type` is a procedure or a setter;
  - `name` is bound in `env` a subprogram with a return type;
  - A “Mismatched return value” error is raised.

15.4.2 Example

15.4.3 Code

```
| _ ->
    fatal_from loc @@ Error.MismatchedReturnValue name
    |: TypingRule.FCallMismatch
```

15.4.4 Formally

15.4.5 Comments

Chapter 16

Typing of Subprograms

Annotating a subprogram `f` in an environment `env` (`annotate_func`) results in `f`, `new_body` and `name`.

16.1 TypingRule.Subprogram

16.1.1 Prose

Annotating a subprogram `f` in an environment `env` (`annotate_func`) results in `f`, `new_body` and all of the following apply:

- `env1` is `env` modified to have an empty local environment and a return type given by `f`;
- `env2` is `env1` with every formal argument given by `f` declared as immutable with its type;
- `env3` is `env2` modified to add explicit parameters given by `f`;
- `env4` is `env3` modified to resolve dependently typed identifiers in the arguments given by `f`;
- `env5` is `env4` modified to resolve dependently typed identifiers in the result type given by `f`;
- `body` is the body given by `f`;
- `new_body` is the result of annotating `body` in `env5`.

16.1.2 Example

16.1.3 Code

```
let annotate_subprogram loc (env : env) (f : 'p AST.func) : 'p AST.func =
  let () = if false then Format.eprintf "Annotating %s.@." f.name in
```

```
(* Build typing local environment. *)
let env1 = { env with local = empty_local_return_type f.return_type } in
let env2 =
  let one_arg env1 (x, ty) =
    let+ () = check_var_not_in_env loc env1 x in
    add_local x ty LDK_Let env1
  in
  List.fold_left one_arg env1 f.args
in
(* Add explicit parameters *)
let env3 =
  let one_param env2 (x, ty_opt) =
    let ty =
      match ty_opt with
      | Some ty -> ty
      | None -> ASTUtils.underconstrained_integer
    in
    let+ () = check_var_not_in_env loc env2 x in
    add_local x ty LDK_Let env2
  in
  List.fold_left one_param env2 f.parameters
in
(* Add dependently typed identifiers. *)
let add_dependently_typed_from_ty env'' ty =
  match ty.desc with
  | T_Bits ({ desc = E_Var x; _ }, _) -> (
      match StaticEnv.type_of_opt env x with
      | Some { desc = T_Int None; _ } ->
          add_local x ASTUtils.underconstrained_integer LDK_Let env''
      | Some _ -> env''
      | None -> add_local x ASTUtils.underconstrained_integer LDK_Let env'')
  | _ -> env''
in
(* Resolve dependently typed identifiers in the arguments. *)
let env4 =
  let one_arg env3 (_, ty) = add_dependently_typed_from_ty env3 ty in
  List.fold_left one_arg env3 f.args
in
(* Resolve dependently typed identifiers in the result type. *)
let env5 =
  match f.return_type with
  | None -> env4
  | Some { desc = T_Bits ({ desc = E_Var x; _ }, _); _ } -> (
      match StaticEnv.type_of_opt env x with
      | Some { desc = T_Int None; _ } ->
          add_local x ASTUtils.underconstrained_integer LDK_Let env4
```

```
      | _ -> env4)
  | _ -> env4
in
(* Annotate body *)
let body =
  match f.body with SB_ASL body -> body | SB_Primitive _ -> assert false
in
let new_body = try_annotate_block env5 body in
(* Optionnally rename the function if needs be *)
let name =
  let args = List.map snd f.args in
  let _, name, _, _ = FunctionRenaming.try_find_name loc env5 f.name args in
  name
in
{ f with body = SB_ASL new_body; name } |: TypingRule.Subprogram
```

16.1.4 Formally

16.1.5 Comments

This is related to $\texttt{I}_{\texttt{GHGK}}$, $\texttt{R}_{\texttt{HWTV}}$, $\texttt{R}_{\texttt{SCHV}}$, $\texttt{R}_{\texttt{VDPC}}$, $\texttt{R}_{\texttt{TJKQ}}$, $\texttt{I}_{\texttt{LFJZ}}$, $\texttt{I}_{\texttt{BZVB}}$, $\texttt{I}_{\texttt{RQQB}}$.

Chapter 17

Side-effects

Side-effects can arise due to:

- Reads and Writes to global variables;
- Exceptions;
- Calls to primitives.

Bibliography

[1] Arm Architecture Technology Group. *ASL Abstract Syntax Reference*. 2023.

[2] Arm Architecture Technology Group. *ASL Semantics Reference*. 2023.

[3] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.