ASL Semantics Reference
DDI 0621

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

The semantics of ASL are embodied in a function that takes as input an ASL program and returns all possible behaviours of that program.

3.1 Environments

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

3.2 Variables

Variables have two different possible scopes: global to the whole program which come with an initial value and cannot be declared inside a subprogram, or local to a subprogram. Global variables are initialised at the start of the evaluation of the program.

3.3 Functions

Functions are declared statically: they are declared before the start of the program and are not values, thus a subprogram call uses directly a function name.

Formally, an environment $E \in \mathcal{E}$ is a pair of a binding from global variable names to their value, and a binding from local variable names to their values:

$$\mathcal{E} = (\mathcal{X} \hookrightarrow \mathcal{V}) \times (\mathcal{X} \hookrightarrow \mathcal{V})$$

We define some notations, for an environment $E = (G, F, L)$, a variable x

and a value v:

$$E.\text{globals} \triangleq G \qquad E.\text{locals} \triangleq L \qquad \text{Env}(G, L) \triangleq E$$

$$E[x \mapsto v] \triangleq \begin{cases} (G[x \mapsto v], L) & \text{if } x \in \text{dom}(G) \\ (G, L[x \mapsto v]) & \text{otherwise} \end{cases}$$

The notation E.globals refers to the global part of the environment, and E.locals to the local variable mappings, and $\text{Env}(G, L)$ is the environment formed with the global mappings in G and the local from L. The notation $E[x \mapsto v]$ is the environment E modified so x is bound to v. Furthermore, $v?$ refers to v or $\perp$, and the usage of v without a question mark implies that $v \neq \perp$. In particular, the over-writing of a variable in an environment depends on whether the variable is a global or local variable.

3.4 Evaluation

Evaluating a program is evaluating its "main" subprogram. Constructively, evaluating a program requires following its Abstract Syntax Tree and evaluating each of its components.

The semantics of a program are given by applying a set of `eval_<label>` functions. Each `eval_<label>` function describes how to evaluate a specific label, as follows.

- `eval_expr` evaluates expressions: it takes an environment and an expression and returns a value and a new environment (see Chapter 5);
- `eval_lexpr` evaluates left-hand sides of assignments: it takes an environment, the left-hand side of an assignment and a value to be written, and returns a environment updated with the new value (see Chapter 6);
- `eval_slices` evaluates slices (see Chapter 7);
- `eval_patterns` evaluates patterns (see Chapter 8);
- `eval_local_decl` evaluates local declarations (see Chapter 9);
- `eval_stmt` evaluates statements: it takes an environment and a statement and returns a new environment, viz, the environment updated with the side-effects of the statement (see Chapter 10);
- `eval_block` evaluates blocks (see Chapter 11);
- `eval_loop` evaluates both `while` and `repeat` loops (see Section 12.1);
- `eval_for` evaluates `for` loops (see Section 12.2);
- `eval_catchers` evaluates catchers (see Chapter 13);
- `eval_func` evaluates subprograms: it takes an environment, a subprogram name and its arguments, and returns a list of the return values of the subprogram (see Chapter 14).

Chapter 4

Reading guide

The definition of each `eval_<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: SemanticsRule.Lit in Section 5.1, Semantics.Binop in Section 5.8 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/ASLSemanticsReference.t`
- a Code paragraph which gives a verbatim of the corresponding implementation in the interpreter of ASLRef `~/herdtools7/asllib/Interpreter.ml`;
- a Formal paragraph for the sequential case, which gives a formal definition of the rule: the sequential case essentially gives the same information as the Prose paragraph;
- a Formal paragraph for the concurrent case: the concurrent case augments the information given by the sequential case with a set of `herd7` execution graphs. This enables building, for each AArch64 instruction, the Intrinsic dependencies used by the AArch64 memory model from this instruction's ASL code.

Formally, the semantics of different language constructs are given by the functions $[|\cdot|]_F$ that map an environment to a set of pair of values and environments, where F is a given set of ASL subprograms. In the following, for simplicity, we will omit the indices F from the semantics definitions, apart when defined, as they are constant in the whole evaluation. Furthermore, the variable names inside set definitions are implicitly quantified. For example the two following definitions are equivalent:

$$X + Y \triangleq \{n \mid n_1 \in X \text{ and } n_2 \in Y \text{and } n = n_1 + n_2\}$$
$$X + Y \triangleq \{n \mid \exists n_1 \in X, \exists n_2 \in Y, n = n_1 + n_2\}$$

Chapter 5

Evaluation of Expressions

`eval_expr` specifies how to evaluate an expression `e` in an environment `env`.

Evaluation of the expression `e` under an environment `env` is either a value `v` together with a potentially updated environment `new_env`, or an error, and one of the following applies:

- SemanticsRule.Lit (see Section 5.1);
- SemanticsRule.ELocalVar (see Section 5.2)
- SemanticsRule.EGlobalVar (see Section 5.3)
- SemanticsRule.EUndefIdent (see Section 5.4)
- SemanticsRule.BinopAnd (see Section 5.5)
- SemanticsRule.BinopOr (see Section 5.6)
- SemanticsRule.BinopImpl (see Section 5.7)
- SemanticsRule.Binop (see Section 5.8)
- SemanticsRule.Unop (see Section 5.9)
- SemanticsRule.ECond (see Section 5.10)
- SemanticsRule.ESlice (see Section 5.11)
- SemanticsRule.ECall (see Section 5.12)
- SemanticsRule.EGetArray (see Section 5.13)
- SemanticsRule.ERecord (see Section 5.14)
- SemanticsRule.EGetField (see Section 5.15)
- SemanticsRule.EConcat (see Section 5.16)

- SemanticsRule.ETuple (see Section 5.17)
- SemanticsRule.EUnknown (see Section 5.18)
- SemanticsRule.EPattern (see Section 5.19)
- SemanticsRule.CTC (see Section 5.20)

5.1 SemanticsRule.Lit

5.1.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a Literal `l`;
- `res` is a value `v`;
- `v` is the value of `l`;
- `new_env` is `env`.

5.1.2 Example

In the program:

```
func main () => integer
begin

  assert 3 == 3;
  return 0;

end
```

the expression `3` evaluates to the value `3`.

5.1.3 Code

```
  | E_Literal v -> return_normal (B.v_of_literal v, env) |: SemanticsRule.Lit
```

5.1.4 Formally: sequential case

The evaluation of a literal value `v` associates any environment `env` to itself coupled with `v`:

$$[|\mathrm{v}|](\mathtt{env}) \triangleq \{(\mathrm{v}, \mathtt{env})\} \tag{5.1}$$

5.1.5 Formally: concurrent case

$$[|\mathtt{v}|](\mathtt{env}) \triangleq \{(\mathtt{v}, \mathtt{env}, \emptyset)\} \tag{5.2}$$

5.1.6 Comments

5.2 SemanticsRule.ELocalVar

5.2.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a variable `x` which is bound locally in `env`;
- `res` is a value `v`;
- `v` is the value of `x` in `env`;
- `new_env` is `env`.

5.2.2 Example: SemanticsRuleELocalVar.asl

In the program:

```
func main () => integer
begin

  var x: integer = 3;
  assert x == 3;

  return 0;
end
```

the evaluation of `x` within `assert x == 3;` uses SemanticsRule.ELocalVar.

5.2.3 Code

```
| Local v ->
    let* () = B.on_read_identifier x (IEnv.get_scope env) v in
    return_normal (v, env) |: SemanticsRule.ELocalVar
```

5.2.4 Formally: sequential case

The evaluation of a variable `x` in an environment `env` that maps `x` to `v` is the tuple $(\mathtt{v}, \mathtt{env})$:

$$[|\mathrm{x}|](\mathtt{env}) \triangleq \{(\mathtt{env}[\mathrm{x}], \mathtt{env}))\} \tag{5.3}$$

5.2.5 Formally: concurrent case

$$[|x|](\texttt{env}) \triangleq \{(\texttt{env}[\texttt{x}], \texttt{env}, \texttt{R(x)})\} \tag{5.4}$$

5.2.6 Comments

When there exists a global variable called `x` then an ASL program cannot use the name `x` as a local variable. ASLRef checks that this property, sometimes called "banning shadowing", is true at type-checking.

5.3 SemanticsRule.EGlobalVar

5.3.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a variable `x` which is bound globally in `env`;
- `res` is a value `v`;
- `v` is the value of `x` in `env`;
- `new_env` is `env`.

5.3.2 Example: SemanticsRuleEGlobalVar.asl

In the program:

```
var global_x: integer = 3;

func main () => integer
  begin

    assert global_x == 3;
    return 0;

  end
```

the evaluation of `global_x` within `assert global_x == 3;` uses SemanticsRule.EGlobalVar.

5.3.3 Code

```
| Global v ->
    let* () = B.on_read_identifier x Scope_Global v in
    return_normal (v, env) |: SemanticsRule.EGlobalVar
```

5.3.4 Formally: sequential case

$$[|\mathtt{x}|](\mathtt{env}) \triangleq \{(\mathtt{env}[\mathtt{x}], \mathtt{env}))\} \tag{5.5}$$

5.3.5 Formally: concurrent case

$$[|\mathtt{x}|](\mathtt{env}) \triangleq \{(\mathtt{env}[\mathtt{x}], \mathtt{env}, \mathtt{R(x)})\} \tag{5.6}$$

5.3.6 Comments

5.4 SemanticsRule.EUndefIdent

5.4.1 Prose

Evaluation of the expression `e` under environment `env` is `res` and all of the following apply:

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

5.4.2 Example: SemanticsRule.EUndefIdent.asl

The program:

```
func main () => integer
begin

  let x = 42;
  assert y;

  return 0;
end
```

raises an `Undefined Identifier` error: the variable `y` is undefined in the environment where `x` is bound to `42`.

5.4.3 Code

```
| NotFound ->
    fatal_from e @@ Error.UndefinedIdentifier x
    |: SemanticsRule.EUndefIdent)
```

5.4.4 Formally: sequential case

5.4.5 Formally: concurrent case

5.4.6 Comments

5.5 SemanticsRule.BinopAnd

5.5.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a conjunction over two expressions `e1` and `e2`;
- `(res,new_env)` is the result of the evaluation of `if e1 then e2 else false` (see Section 5.10).

5.5.2 Example: SemanticsRule.EBinopAndFalse.asl

```
func fail() => boolean
begin
  assert FALSE;
  return TRUE;
end

func main () => integer
begin
  let b = FALSE && fail();
  assert b == FALSE;
  return 0;
end
```

the expression `FALSE && fail()` evaluates to the value `FALSE`. Notice that the function `fail` is not called.

5.5.3 Code

```
    | E_Binop (BAND, e1, e2) ->
        (* if e1 then e2 else false *)
        E_Cond (e1, e2, false')
        |> add_pos_from e |> eval_expr env |: SemanticsRule.BinopAnd
```

5.5.4 Formally: sequential case

5.5.5 Formally: concurrent case

5.5.6 Comments

This is related to $\texttt{R}_{\texttt{BKNT}}$, $\texttt{R}_{\texttt{XKGC}}$ and $\texttt{I}_{\texttt{QRXP}}$.

5.6 SemanticsRule.BinopOr

5.6.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a disjunction over two expressions `e1` and `e2`;
- `(res,new_env)` is the result of the evaluation of `if e1 then true else e2` (see Section 5.10).

5.6.2 Example: SemanticsRule.EBinopOrTrue.asl

```
func main () => integer
begin
  let b = (0 == 1) || (1 == 1);
  assert b;
  return 0;
end
```

the expression `(0 == 1)` ||`&&` `(1 == 1)` evaluates to the value `TRUE`.

5.6.3 Code

```
    | E_Binop (BOR, e1, e2) ->
        (* if e1 then true else e2 *)
        E_Cond (e1, true', e2)
        |> add_pos_from e |> eval_expr env |: SemanticsRule.BinopOr
```

5.6.4 Formally: sequential case

5.6.5 Formally: concurrent case

5.6.6 Comments

This is related to $\texttt{R}_{\texttt{BKNT}}$, $\texttt{R}_{\texttt{XKGC}}$ and $\texttt{I}_{\texttt{QRXP}}$.

5.7 SemanticsRule.BinopImpl

5.7.1 Prose

All of the following apply:

- `e` denotes an implication over two expressions `e1` and `e2`;
- `e` is evaluated as `if e1 then e2 else true`.

5.7.2 Example: SemanticsRule.EBinopImplExFalso.asl

```
func main () => integer
begin
  let b = (0 == 1) --> (1 == 0);
  assert b;
  return 0;
end
```

the expression `(0 == 1) --> (1 == 0)` evaluates to the value `TRUE`, according to the definition of implication.

5.7.3 Code

```
| E_Binop (IMPL, e1, e2) ->
    (* if e1 then e2 else true *)
    E_Cond (e1, e2, true')
    |> add_pos_from e |> eval_expr env |: SemanticsRule.BinopImpl
```

5.7.4 Formally: sequential case

5.7.5 Formally: concurrent case

5.7.6 Comments

This is related to $\texttt{R}_{\texttt{BKNT}}$, and $\texttt{I}_{\texttt{QRXP}}$.

5.8 SemanticsRule.Binop

5.8.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a Binary Operator `op` over two expressions `e1` and `e2`;
- The evaluation of the expression `e1` under `env` is `v1`;

- env' is the result of modifying env after evaluation of the expression e1 under env;
- The evaluation of the expression e2 under env is v2;
- new_env is the result of modifying env' after evaluation of the expression e2 under env';
- v is the result of applying the Binary Operator op to v1 and v2;
- res is v;
- new_env is env.

5.8.2 Example: SemanticsRule.EBinopPlusAssert.asl

In this program:

```
func main () => integer
begin

  let x = 3 + 2;
  assert x==5;

  return 0;
end
```

the expression 3 + 2 evaluates to the value 5.

5.8.3 Example: SemanticsRule.EDIVBackendDefinedError.asl

In the program:

```
func main () => integer
begin

  let x = 3 DIV 0;

  return 0;
end
```

- in ASLv0, the expression 3 DIV 0 raises a backend-defined error, e.g. ASL Execution error: Illegal application of operator DIV for values 3 and 0.
- in ASLv1, the expression 3 DIV 0 raises a type error.

5.8.4 Code

```
| E_Binop (op, e1, e2) ->
    let*^ m1, env' = eval_expr env e1 in
    let*^ m2, new_env = eval_expr env' e2 in
    let* v1 = m1 and* v2 = m2 in
    let* v = B.binop op v1 v2 in
    return_normal (v, new_env) |: SemanticsRule.Binop
```

5.8.5 Formally: sequential case

$$[|\texttt{e1} + \texttt{e2}|](\texttt{env}) \triangleq \left\{ (v, \texttt{new\_env}) \;\middle|\; \begin{array}{l} \texttt{v1} + \texttt{v2} = \texttt{v} \\ \text{and } (\texttt{v1}, \texttt{env'}) \in [|\texttt{e1}|](E) \\ \text{and } (\texttt{v2}, \texttt{new\_env}) \in [|\texttt{e2}|](\texttt{env'}) \end{array} \right\} \tag{5.7}$$

5.8.6 Formally: concurrent case

For the ordering constraints, the different arguments of a n-ary subprogram are considered computed in parallel:

$$[|\texttt{e1} + \texttt{e2}|](\texttt{env}) \triangleq \left\{ (\texttt{v}, \texttt{new\_env}, S_1 \parallel S_2) \;\middle|\; \begin{array}{l} \texttt{v1} + \texttt{v2} = \texttt{v} \\ \text{and } (\texttt{v1}, \texttt{env'}, S_1) \in [|\texttt{e1}|](\texttt{env}) \\ \text{and } (\texttt{v2}, \texttt{new\_env}, S_2) \in \; < [|\texttt{e2}|](\texttt{env'}) \end{array} \right\} \tag{5.8}$$

5.8.7 Comments

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

5.9 SemanticsRule.Unop

5.9.1 Prose

All of the following apply:

- `e` denotes a Unary Operator `op` over an expression `e'` in an environment `env`;
- The evaluation of the expression `e'` under `env` is `v',new_env`;
- `v, new_env` is the result of applying the Unary Operator `op` to `v'`.

5.9.2 Example: SemanticsRule.EUnopAssert.asl

In the program:

```
func main () => integer
begin

  let x = NOT '1010';
  assert x=='0101';

  return 0;
end
```

the expression `NOT '1010'` evaluates to the value `'0101'`.

5.9.3 Code

```
    | E_Unop (op, e') ->
        let** v', env' = eval_expr env e' in
        let* v = B.unop op v' in
        return_normal (v, env') |: SemanticsRule.Unop
```

5.9.4 Formally: sequential case

5.9.5 Formally: concurrent case

5.9.6 Comments

5.10 SemanticsRule.ECond

5.10.1 Prose

All of the following apply:

- `e` denotes a conditional expression `e_cond` with two options `e1` and `e2`;
- The evaluation of the conditional expression `e_cond` under `env` is `m_cond`;
- The evaluation of `e1` or `e2`, depending on `m_cond`, is `v`.

5.10.2 Example: SemanticsRule.ECondFalse.asl

In the program:

```
func Return42() => integer
begin
  return 42;
end
```

```
func main () => integer
begin

  let x = if FALSE then Return42() else 3;
  assert x==3;

  return 0;
end
```

the expression `if FALSE then Return42() else 3` evaluates to the value `3`.

5.10.3 Example: SemanticsRule.ECondUNKNOWN3or42.asl

In the program:

```
func Return42() => integer
begin
  return 42;
end

func main () => integer
begin

  let x = if UNKNOWN: boolean then 3 else Return42();
  assert x==3;

  return 0;
end
```

the expression `if UNKNOWN: boolean then 3 else Return42()` will evaluate either `3` or `Return42()` depending on how `UNKNOWN` is implemented.

5.10.4 Code

```
| E_Cond (e_cond, e1, e2) ->
    let*^ m_cond, env1 = eval_expr env e_cond in
    if is_simple_expr e1 && is_simple_expr e2 then
      let*= v_cond = m_cond in
      let* v =
        B.ternary v_cond
          (fun () -> eval_expr_sef env1 e1)
          (fun () -> eval_expr_sef env1 e2)
      in
      return_normal (v, env) |: SemanticsRule.ECondSimple
    else choice m_cond e1 e2 >>*= eval_expr env1 |: SemanticsRule.ECond
```

5.10.5 Formally: sequential case

5.10.6 Formally: concurrent case

5.10.7 Comments

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

5.11 SemanticsRule.ESlice

5.11.1 Prose

All of the following apply:

- `e` denotes an expression `e_bv` sliced as per `slices`;
- The evaluation of `e_bv` under `env` is `v_bv`;
- The evaluation of `slices` under `env` is `positions`;
- `v` is the value read in `v_bv` from `positions`.

5.11.2 Example: SemanticsRule.ESlice.asl

In the program:

```
func main () => integer
begin

  let x = ['11110000'[6:3]];
  assert x == '1110';

  return 0;
end
```

the expression `'11110000'[5:2]` evaluates to the value `'1100'`.

5.11.3 Code

```
  | E_Slice (e_bv, slices) ->
      let*^ m_bv, env1 = eval_expr env e_bv in
      let*^ m_positions, env' = eval_slices env1 slices in
      let* v_bv = m_bv and* positions = m_positions in
      let* v = B.read_from_bitvector positions v_bv in
      return_normal (v, env') |: SemanticsRule.ESlice
```

5.11.4 Formally: sequential case

5.11.5 Formally: concurrent case

5.11.6 Comments

5.12 SemanticsRule.ECall

5.12.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a subprogram call `(name, params, actual_args)`;
- The evaluation of that subprogram call under `env` is `ms`;
- `res` is a value `v`;
- `v` is the value read from `ms`;
- `new_env` is `env`.

5.12.2 Example: SemanticsRule.ECall.asl

In the program:

```
func Return42() => integer
begin
  return 42;
end

func main () => integer
begin

  let x = Return42();
  assert x == 42;

  return 0;
end
```

the expression `Return42()` evaluates to the value `42` because the subprogram `Return42()` is implemented to return the value `42`.

5.12.3 Code

```
    | E_Call (name, actual_args, params) ->
        let**| ms, env = eval_call (to_pos e) name env actual_args params in
        let* v =
```

```
      match ms with
      | [ m ] -> m
      | _ ->
          let* vs = sync_list ms in
          B.create_vector vs
    in
    return_normal (v, env) |: SemanticsRule.ECall
```

5.12.4 Formally: sequential case

5.12.5 Formally: concurrent case

5.12.6 Comments

5.13 SemanticsRule.EGetArray

5.13.1 Prose

Evaluation of the expression `e` under environment `env` is such that all of the following apply:

- `e` denotes an array `e_array` and an index `e_index`;
- The evaluation of `e_array` under `env` is `v_array`;
- The evaluation of `e_index` under `env` is `v_index`;
- One of the following applies:
  * All of the following apply:
    - the result of evaluation of `e` under `env` is `(res, env)`;
    - `res` is a value `v`;
    - `v` is the value found at the index `v_index` of `v_array`;
    - `new_env` is `env`;
  * `res` is a typing error.

5.13.2 Example: SemanticsRule.EGetArray.asl

In the program:

```
type MyArrayType of array [3] of integer;

var my_array : MyArrayType;

func main () => integer
begin

  my_array[2]=42;
```

```
  assert my_array[2]==42;

  return 0;
end
```

the expression `my_array[2]` evaluates to the value 42 since the element indexed by 2 in `my_array` is 42.

5.13.3 Example: SemanticsRule.EGetArrayTooSmall.asl

The program:

```
type MyArrayType of array [3] of integer;

var my_array : MyArrayType;

func main () => integer
begin

  my_array[3]=42;
  assert my_array[3]==42;

  return 0;
end
```

raises a typing error since we are trying to access index 3 of an array which has indexes 0, 1 and 2 only.

5.13.4 Code

```
    | E_GetArray (e_array, e_index) -> (
        let*^ m_array, env1 = eval_expr env e_array in
        let*^ m_index, env' = eval_expr env1 e_index in
        let* v_array = m_array and* v_index = m_index in
        match B.v_to_int v_index with
        | None ->
            (* TODO: create a proper runtime error for this.
               It should be caught at type-checking, but still. *)
            fatal_from e (Error.UnsupportedExpr e_index)
        | Some i ->
            let* v = B.get_index i v_array in
            return_normal (v, env') |: SemanticsRule.EGetArray)
```

5.13.5 Formally: sequential case

5.13.6 Formally: concurrent case

5.13.7 Comments

5.14 SemanticsRule.ERecord

5.14.1 Prose

Evaluation of the expression e under environment env is (res,new_env) and all of the following apply:

- e denotes a record value names and associated expressions fields;
- fields evaluates in env to v_fields;
- res is a value v;
- v is the record built by associating the names names to v_fields.
- new_env is env.

5.14.2 Example: SemanticsRule.ERecord.asl

In the program:

```
type MyRecordType of record {a: integer, b: integer};

func main () => integer
begin

  let my_record = MyRecordType{a=3, b=42};
  assert my_record.a == 3;

  return 0;
end
```

the expression MyRecordType a=3, b=42 evaluates to the value a: 3, b: 42.

5.14.3 Code

```
  | E_Record (_, e_fields) ->
      let names, fields = List.split e_fields in
      let** v_fields, env' = eval_expr_list env fields in
      let* v = B.create_record (List.combine names v_fields) in
      return_normal (v, env') |: SemanticsRule.ERecord
```

5.14.4 Formally: sequential case

5.14.5 Formally: concurrent case

5.14.6 Comments

5.15 SemanticsRule.EGetField

5.15.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a field name `field_name` in a record value `e_vec`;
- the evaluation of `e_vec` in `env` is `v_vec`;
- `res` is a value `v`;
- `v` is the value mapped by `field_name` in `v_vec`;
- `new_env` is `env`.

5.15.2 Example: SemanticsRule.ERecord.asl

In the program:

```
type MyRecordType of record {a: integer, b: integer};

func main () => integer
begin

  let my_record = MyRecordType{a=3, b=42};
  assert my_record.a == 3;

  return 0;
end
```

the expression `my_record.a` evaluates to the value `3`.

5.15.3 Code

```
| E_GetField (e_record, field_name) ->
    let** v_record, env' = eval_expr env e_record in
    let* v = B.get_field field_name v_record in
    return_normal (v, env') |: SemanticsRule.EGetBitField
```

5.15.4 Formally: sequential case

5.15.5 Formally: concurrent case

5.15.6 Comments

5.16 SemanticsRule.EConcat

5.16.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a list of bitvector expressions `e_list`;
- the evaluation of `e_list` in `env` is `v_list`;
- `res` is a value `v`;
- `v` is the concatenation of `v_list`;
- `new_env` is `env`.

5.16.2 Example: SemanticsRule.EConcat

In the program:

```
func main () => integer
begin

  let x = [['10','11']];
  assert x=='1011';

  return 0;
end
```

the expression `['10', '11']` evaluates to the value `'1011'`.

5.16.3 Code

```
    | E_Concat e_list ->
        let** v_list, env' = eval_expr_list env e_list in
        let* v = B.concat_bitvectors v_list in
        return_normal (v, env') |: SemanticsRule.EConcat
```

5.16.4 Formally: sequential case

5.16.5 Formally: concurrent case

5.16.6 Comments

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

5.17 SemanticsRule.ETuple

5.17.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a list of expression `e_list`;
- the evaluation of `e_list` in `env` is `v_list`;
- `res` is a value `v`;
- `v` is the tuple built from `v_list`;
- `new_env` is `env`.

5.17.2 Example: SemanticsRule.ETuple.asl

In the program:

```
func Return42() => integer
begin
  return 42;
end

func main () => integer
begin

  let (x,y) = (3, Return42());
  assert x == 3;
  assert y == 42;

  return 0;
end
```

the expression `(3, Return42())` evaluates to the value `(3, 42)`.

5.17.3 Code

```
  | E_Tuple e_list ->
      let** v_list, env' = eval_expr_list env e_list in
      let* v = B.create_vector v_list in
      return_normal (v, env') |: SemanticsRule.ETuple
```

5.17.4 Formally: sequential case

5.17.5 Formally: concurrent case

5.17.6 Comments

5.18 SemanticsRule.EUnknown

5.18.1 Domain of a type

The domain of a type t, written $D(t)$, is defined as follows:

$$D(t) \triangleq \begin{cases} \{\text{true}, \text{false}\} & \text{if } t \text{ is boolean} \\ \mathbb{Z} & \text{if } t \text{ is integer} \\ \{0,1\}^n & \text{if } t \text{ is bits}(n) \end{cases}$$

5.18.2 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a type `t`;
- `res` is a value `v`;
- `v` is a value in the domain of `t`;
- `new_env` is `env`.

5.18.3 Example: SemanticsRule.EUnknownInteger3.asl

In the program:

```
func main () => integer
begin

  let x = UNKNOWN:integer;
  assert x==3;

  return 0;
end
```

the expression `[UNKNOWN : integer]` evaluates to an integer value.

5.18.4 Example: SemanticsRule.EUnknownIntegerRange3-42-3.asl

In the program:

```
func main () => integer
begin

  let x = UNKNOWN:integer {3, 42};
  assert x==3;

  return 0;
end
```

the expression `UNKNOWN : integer {3, 42}` evaluates to either the value `3` or the value `42`.

5.18.5 Code

```
| E_Unknown t ->
    let v = B.v_unknown_of_type t in
    return_normal (v, env) |: SemanticsRule.EUnknown
```

5.18.6 Formally: sequential case

$$[|\texttt{unknown} : t|](E) \triangleq \{(v, E) \mid v \in D(t)\} \tag{5.9}$$

5.18.7 Formally: concurrent case

$$[|\texttt{unknown} : t|](E) \triangleq \{(v, E, \varnothing) \mid v \in D(t)\} \tag{5.10}$$

5.18.8 Comments

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

5.19 SemanticsRule.EPattern

5.19.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `e` denotes a pattern `e,p`;
- `res` is a value `v`;
- `v` is the boolean determining whether the evaluation of `e` in `env` matches `p`;
- `new_env` is `env`.

5.19.2 Example: SemanticsRule.EPatternFALSE.asl

In the program:

```
func main () => integer
begin

  let x = 42 IN {0..3, -4};
  assert x == FALSE;

  return 0;
end
```

the expression `42 IN {0..3, -4}` evaluates to the value `FALSE`.

5.19.3 Example: SemanticsRule.EPatternTRUE.asl

In the program:

```
func main () => integer
begin

  let x = 42 IN {0..3, 42};
  assert x == TRUE;

  return 0;
end
```

the expression `42 IN {0..3, 42}` evaluates to `TRUE`.

5.19.4 Code

```
    | E_Pattern (e, p) ->
        let** v, env' = eval_expr env e in
        let* v = eval_pattern env e v p in
        return_normal (v, env') |: SemanticsRule.EPattern
```

5.19.5 Formally: sequential case

5.19.6 Formally: concurrent case

5.19.7 Comments

5.20 SemanticsRule.CTC

5.20.1 Prose

Evaluation of the expression `e` under environment `env` is `(res,new_env)` and all of the following apply:

- `(e,_t)` denotes an expression `e` and a type `t`;
- `v` is the result of evaluating `e` in `env`;
- `new_env` is `env` modified after evaluating `e` in `env`;
- `b` is `true` or `false` depending on whether `v` is of type `t` in `env`;
- One of the following applies:
  - ∗ All of the following apply:
    - `b` is `true`;
    - `res` is `v`;
    - `new_env` is `new_env`
  - ∗ All of the following apply:
    - `b` is `false`;
    - a type error is raised.

5.20.2 Example: SemanticsRule.CTCValue.asl

```
func main () => integer
begin

  let my_ctc = 3 as integer;
  assert my_ctc == 3;

  return 0;
end
```

5.20.3 Example: SemanticsRule.CTCError.asl

```
func main () => integer
begin

  let my_ctc = (3 as integer {3..5});

  return 0;
end
```

5.20.4 Code

```
    | E_CTC (e, t) ->
        let** v, new_env = eval_expr env e in
        let* b = is_val_of_type e env v t in
```

```
(if b then return_normal (v, new_env)
 else fatal_from e (Error.MismatchType (B.debug_value v, [ t.desc ])))
|: SemanticsRule.CTC
```

5.20.5 Formally: sequential case

5.20.6 Formally: concurrent case

5.20.7 Comments

This is related to $\mathtt{R}_{\mathtt{WZVX}}$, $\mathtt{I}_{\mathtt{VQLX}}$, $\mathtt{R}_{\mathtt{YCPX}}$, $\mathtt{I}_{\mathtt{TCST}}$, $\mathtt{I}_{\mathtt{CGRH}}$.

Chapter 6

Evaluation of Left-Hand-Side Expressions

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is either `new_env` or an error is raised and one of the following applies:

- SemanticsRule.LEDiscard (see Section 6.1);
- SemanticsRule.LELocalVar (see Section 6.2);
- SemanticsRule.LEGlobalVar (see Section 6.3);
- SemanticsRule.LEUndefIdentV0 (see Section 6.4);
- SemanticsRule.LEUndefIdentV1 (see Section 6.5);
- SemanticsRule.LESlice (see Section 6.6);
- SemanticsRule.LESetArray (see Section 6.7);
- SemanticsRule.LESetField (see Section 6.8);
- SemanticsRule.LEDestructuring (see Section 6.9).

6.1 SemanticsRule.LEDiscard

6.1.1 Prose

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is `new_env` and all of the following apply:

- `le` can be discarded;
- `new_env` is `env`.

6.1.2 Example: SemanticsRule.LEDiscard.asl

In the program:

```
func main () => integer
begin

  - = 42;
  assert TRUE;

  return 0;
end
```

`- = 42;` does not affect the environment.

6.1.3 Code

```
  | LE_Discard -> return_normal env |: SemanticsRule.LEDiscard
```

6.1.4 Formally: sequential case

6.1.5 Formally: concurrent case

6.1.6 Comments

6.2 SemanticsRule.LELocalVar

6.2.1 Prose

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is `new_env` and all of the following apply:

- `le` denotes a variable `x`;
- `x` is locally bound in `env`;
- `new_env` is `env` where `x` is bound to `v`.

6.2.2 Example: SemanticsRule.LELocalVar.asl

In the program:

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

```
  var x: integer = 3;
  x = 42;
  assert x == 42;

  return 0;
end
```

the evaluation of the left-hand-side expression `x` within `x = 42;` uses SemanticsRule.LELocalVar.

6.2.3 Code

```
| Local env ->
    let* () = B.on_write_identifier x (IEnv.get_scope env) v in
    return_normal env |: SemanticsRule.LELocalVar
```

6.2.4 Formally: sequential case

6.2.5 Formally: concurrent case

6.2.6 Comments

6.3 SemanticsRule.LEGlobalVar

6.3.1 Prose

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is `new_env` and all of the following apply:

- `le` denotes a variable `x`;
- `x` is globally bound in `env`;
- `new_env` is `env` where `x` is bound to `v`.

6.3.2 Example: SemanticsRule.LEGlobalVar.asl

In the program:

```
var x: integer = 3;

func main () => integer
begin

  x = 42;
  assert x==42;

  return 0;
end
```

the evaluation of the left-hand-side expression `x` within `x = 42;` uses SemanticsRule.LEGlobalVar.

6.3.3 Code

```
| Global env ->
    let* () = B.on_write_identifier x Scope_Global v in
    return_normal env |: SemanticsRule.LEGlobalVar
```

6.3.4 Formally: sequential case

6.3.5 Formally: concurrent case

6.3.6 Comments

6.4 SemanticsRule.LEUndefIdentV0

6.4.1 Prose

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is `new_env` and all of the following apply:

- `le` denotes a variable `x` which is not bound in `env`;
- the ASL language version is `V0`;
- `new_env` is `env` where `x` has been declared a local variable bound to the value `v`.

6.4.2 Example: SemanticsRule.LEUndefIdentV0.asl

```
integer main ()
  let x = 42;
  y = 3;
  assert y == 3 && x == 42;
  return 0;
```

6.4.3 Code

```
| V0 ->
    (* V0 first assignments promoted to local declarations *)
    declare_local_identifier env x v
    >>= return_normal |: SemanticsRule.LEUndefIdentV0))
```

6.4.4 Formally: sequential case

6.4.5 Formally: concurrent case

6.4.6 Comments

6.5 SemanticsRule.LEUndefIdentV1

6.5.1 Prose

All of the following apply:

- `le` denotes a variable `x` which is not bound in `env`;
- the ASL language version is `V1`;
- an `UndefinedIdentifier` error is raised.

6.5.2 Example: SemanticsRule.LEUndefIdentV1.asl

In the program:

```
func main () => integer
begin

  let x = 42;
  y = 3;

  return 0;
end
```

the evaluation of the left-hand-side expression `y` within `y = 3;` raises an "Undefined identifier" error in the environment where `x` is bound to `42`.

6.5.3 Code

```
| V1 ->
    fatal_from le @@ Error.UndefinedIdentifier x
    |: SemanticsRule.LEUndefIdentV1
```

6.5.4 Formally: sequential case

6.5.5 Formally: concurrent case

6.5.6 Comments

6.6 SemanticsRule.LESlice

6.6.1 Prose

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is `new_env` and all of the following apply:

- `le` denotes a left-hand-side expression sliced as per `slices`;
- The right-hand-side expression corresponding to `le` is `re_bv`;
- The evaluation of `re_bv` under `env` is `rv_bv`;
- The evaluation of `slices` under `env` is `positions`;
- `new_m_bv` is `rv_bv` where the positions `positions` have been updated to `v`;
- `new_env` is `env` where `le` is bound to `new_m_bv`.

6.6.2 Example: SemanticsRule.LESlice.asl

In the program:

```
func main () => integer
begin

  var x = '11111111';
  x[3:0] = '0000';
  assert x == '11110000';

  return 0;
end
```

`x[3:0] = '0000'` binds `x` to `'11110000'` in the environment where `x` is bound to `'11111111'`.

6.6.3 Code

```
  | LE_Slice (re_bv, slices) ->
      let*^ rm_bv, env = expr_of_lexpr re_bv |> eval_expr env in
      let*^ m_positions, env = eval_slices env slices in
      let new_m_bv =
        let* v = m and* positions = m_positions and* rv_bv = rm_bv in
```

```
    B.write_to_bitvector positions v rv_bv
  in
  eval_lexpr ver re_bv env new_m_bv |: SemanticsRule.LESlice
```

6.6.4 Formally: sequential case

6.6.5 Formally: concurrent case

6.6.6 Comments

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

6.7 SemanticsRule.LESetArray

6.7.1 Prose

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is `new_env` and all of the following apply:

- `le` denotes an array `le_array` and an index `e_index`;
- The right-hand-side expression corresponding to `le_array` is `re_array`;
- The evaluation of `re_array` under `env` is `rv_array`;
- The evaluation of `e_index` under `env` is `v_index`;
- `new_v_array` is `rv_array` where the value at index `v_index` has been updated to `v`;
- `new_env` is `env` where `le_array` is bound to `new_v_array`.

6.7.2 Example: SemanticsRule.LESetArray.asl

The program:

```
func main () => integer
begin

  var my_array: array [42] of integer;
  my_array[3] = 53;
  assert my_array[3] == 53;

  return 0;
end
```

binds the third element of `my_array` to the value `53`.

6.7.3 Code

```
| LE_SetArray (re_array, e_index) ->
    let*^ rm_array, env = expr_of_lexpr re_array |> eval_expr env in
    let*^ m_index, env = eval_expr env e_index in
    let m' =
      let* v = m and* v_index = m_index and* rv_array = rm_array in
      match B.v_to_int v_index with
      | None -> fatal_from le (Error.UnsupportedExpr e_index)
      | Some i -> B.set_index i v rv_array
    in
    eval_lexpr ver re_array env m' |: SemanticsRule.LESetArray
```

6.7.4 Formally: sequential case

6.7.5 Formally: concurrent case

6.7.6 Comments

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

6.8 SemanticsRule.LESetField

6.8.1 Prose

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is `new_env` and all of the following apply:

- `le` denotes a field name `field_name` in a record `le_record`;
- The right-hand-side expression corresponding to `le_record` is `re_record`;
- The evaluation of `re_record` under `env` is `rv_record`;
- `new_v_record` is `rv_record` where the field `field_name` has been updated to `v`;
- `new_env` is `env` where `le_record` is bound to `new_v_record`.

6.8.2 Example: SemanticsRule.LESetField.asl

In the program:

```
type MyRecordType of record { a: integer, b: integer };

func main () => integer
begin

  var my_record = MyRecordType { a = 3, b = 42 };
```

```
  my_record.a = 42;
  assert my_record.a == 42 && my_record.b == 42;

  return 0;
end
```

`my_record.a = 42;` binds `my_record` to {`a: 42, b: 42`} in the environment where `my_record` is bound to {`a: 3, b: 42`}.

6.8.3 Code

```
  | LE_SetField (re_record, field_name) ->
      let*^ rm_record, env = expr_of_lexpr re_record |> eval_expr env in
      let m' =
        let* v = m and* rv_record = rm_record in
        B.set_field field_name v rv_record
      in
      eval_lexpr ver re_record env m' |: SemanticsRule.LESetField
```

6.8.4 Formally: sequential case

6.8.5 Formally: concurrent case

6.8.6 Comments

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

6.9 SemanticsRule.LEDestructuring

6.9.1 Prose

Evaluation of the left-hand-side expression `le` associated with a value `v` under an environment `env` is `new_env` and all of the following apply:

- `le` denotes a list of left-hand-side expressions `le_list`;
- `new_env` is `env` where each left-hand-side expression in `le_list` has been assigned the value at the corresponding index in `v`.

6.9.2 Example: SemanticsRule.LEDestructuring.asl

In the program:

```
func main () => integer
begin

  var x: integer = 42;
  var y: integer = 3;
```

```
  (x, y) = (3, 42);

  assert x == 3 && y == 42;

  return 0;
end
```

`(x, y) = (3, 42)` binds `x` to `3` and `y` to `42` in the environment where `x` is bound to `42` and `y` is bound to `3`.

6.9.3 Code

```
    | LE_Destructuring le_list ->
        (* The index-out-of-bound on the vector are done either in typing,
           either in [B.get_index]. *)
        let n = List.length le_list in
        let nmonads = List.init n (fun i -> m >>= B.get_index i) in
        multi_assign ver env le_list nmonads |: SemanticsRule.LEDestructuring
```

6.9.4 Formally: sequential case

6.9.5 Formally: concurrent case

6.9.6 Comments

Chapter 7

Evaluation of Slices

`eval_slices env slices` is the list of pair `(start_n, length_n)` that corresponds to the start (included) and the length of each slice in `slices`.

Evaluation of the slice `s` under environment `env` is `((start, length), new_env)`, or an error, and one of the following applies:

- SemanticsRule.SliceSingle (see Section 7.1),
- SemanticsRule.SliceLength (see Section 7.2),
- SemanticsRule.SliceRange (see Section 7.3),
- SemanticsRule.SliceStar (see Section 7.4).

7.1 SemanticsRule.SliceSingle

7.1.1 Prose

The result of evaluation is `((start, length), new_env)` and all of the following apply:

- `s` is the single expression `e`;
- `start` is the result of evaluation of the expression `e` in the environment `env`;
- `new_env` is the environment `env` modified after evaluation of the expression `e`;
- `length` is the integer value 1.

7.1.2 Example: SemanticsRule.SliceSingle.asl

In the program:

```
func main () => integer
begin
  let x = '00000100';

  assert x[2] == '1';

  return 0;
end
```

the slice `[2]` evaluates to `(2, 1)`, i. e. the slice of length 1 starting at index 2.

7.1.3 Code

```
| Slice_Single e ->
    let** start, new_env = eval_expr env e in
    return_normal ((start, one), new_env) |: SemanticsRule.SliceSingle
```

7.1.4 Formally: sequential case

7.1.5 Formally: concurrent case

7.1.6 Comments

7.2 SemanticsRule.SliceLength

7.2.1 Prose

The result of evaluation is `((start, length), new_env)` and all of the following apply:

- `s` is the slice which starts at expression `e_start` with length `e_length`;
- `start` is the result of evaluation of the expression `e_start` in the environment `env`;
- `env_1` is the environment `env` modified after evaluation of the expression `e_start`;
- `length` is the result of evaluation of the expression `e_length` in the environment `env_1`;
- `new_env` is the environment `env_1` modified after evaluation of the expression `e_length`.

7.2.2 Example: SemanticsRule.SliceLength.asl

In the program:

```
func main () => integer
begin
  let x = '00011100';

  assert x[2+:3] == '111';

  return 0;
end
```

`2+:3` evaluates to `(2, 3)`.

7.2.3 Code

```
    | Slice_Length (e_start, e_length) ->
        let*^ start, env1 = eval_expr env e_start in
        let*^ length, new_env = eval_expr env1 e_length in
        let* start = start and* length = length in
        return_normal ((start, length), new_env) |: SemanticsRule.SliceLength
```

7.2.4 Formally: sequential case

7.2.5 Formally: concurrent case

7.2.6 Comments

7.3 SemanticsRule.SliceRange

7.3.1 Prose

The result of evaluation is `((start, length), new_env)` and all of the following apply:

- `s` is the slice range between the expressions `e_start` and `e_top`;
- `v_top` is the result of evaluation of the expression `e_top` in the environment `env`;
- `env_1` is the environment `env` modified after evaluation of the expression `e_top`;
- `start` is the result of evaluation of the expression `e_start` in the environment `env_1`;
- `new_env` is the environment `env_1` modified after evaluation of the expression `e_start`;

- length is the integer value (vtop - start) + 1;

7.3.2 Example: SemanticsRule.SliceRange.asl

In the program:

```
func main () => integer
begin

  let x = '00011100';

  assert x[4:2] == '111';

  return 0;
end
```

4:2 evaluates to (2, 3).

7.3.3 Code

```
| Slice_Range (e_top, e_start) ->
    let*^ v_top, env1 = eval_expr env e_top in
    let*^ start, new_env = eval_expr env1 e_start in
    let* v_top = v_top and* start = start in
    let* length = B.binop MINUS v_top start >>= B.binop PLUS one in
    return_normal ((start, length), new_env) |: SemanticsRule.SliceRange
```

7.3.4 Formally: sequential case

7.3.5 Formally: concurrent case

7.3.6 Comments

7.4 SemanticsRule.SliceStar

7.4.1 Prose

The result of evaluation is ((start, length), new_env) and all of the following apply:

- s is the slice with factor given by the expression e_factor and length given by the expression e_length;
- v_factor is the result of evaluation of the expression e_factor in the environment env;
- env_1 is the environment env modified after evaluation of the expression e_factor;

- `length` is the result of evaluation of the expression `e_length` in the environment `env_1`;
- `new_env` is the environment `env_1` modified after evaluation of the expression `e_length`;
- `start` is the integer value `v_factor` × `length`.

7.4.2 Example: SemanticsRule.SliceStar.asl

In the program:

```
func main () => integer
begin
  let x = '11000000';

  assert x[3*:2] == '11';

  return 0;
end
```

`x[2*:3]` evaluates to `(6, 2)`.

7.4.3 Code

```
| Slice_Star (e_factor, e_length) ->
    let*^ v_factor, env1 = eval_expr env e_factor in
    let*^ length, new_env = eval_expr env1 e_length in
    let* v_factor = v_factor and* length = length in
    let* start = B.binop MUL v_factor length in
    return_normal ((start, length), new_env) |: SemanticsRule.SliceStar
```

7.4.4 Formally: sequential case

7.4.5 Formally: concurrent case

7.4.6 Comments

Chapter 8

Evaluation of Patterns

`eval_pattern env pos v p` determines if `v` matches the pattern `p`.

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b`, or an error, and one of the following applies:

- SemanticsRule.PAll (see Section 8.1)
- SemanticsRule.PAny (see Section 8.2)
- SemanticsRule.PGeq (see Section 8.3)
- SemanticsRule.PLeq (see Section 8.4)
- SemanticsRule.PNot (see Section 8.5)
- SemanticsRule.PRange (see Section 8.6)
- SemanticsRule.PSingle (see Section 8.7)
- SemanticsRule.PMask (see Section 8.8)
- SemanticsRule.PTuple (see Section 8.9)

8.1 SemanticsRule.PAll

8.1.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` is the pattern which matches everything, and therefore matches `v`;
- `b` is the boolean value `true`.

8.1.2 Example: SemanticsRule.PAll.asl

```
func main () => integer
begin

  let match_me = 42 IN { - };
  assert match_me == TRUE;

  return 0;
end
```

8.1.3 Code

```
    | Pattern_All -> true_ |: SemanticsRule.PAll
```

8.1.4 Formally: sequential case

8.1.5 Formally: concurrent case

8.1.6 Comments

8.2 SemanticsRule.PAny

8.2.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` is a list of patterns `ps`;
- `bs` is the list resulting of the evaluation of the patterns in `ps` under environment `env` with respect to value `v`;
- `b` is the disjunction of the values in `bs`.

8.2.2 Example: SemanticsRule.PAnyTRUE.asl

```
func main () => integer
begin

  let match_me = 42 IN { 3, 42 };
  assert match_me == TRUE;

  return 0;
end
```

8.2.3 Example: SemanticsRule.PAnyFALSE.asl

```
func main () => integer
begin

  let match_me = 42 IN { 3, 4 };
  assert match_me == FALSE;

  return 0;
end
```

8.2.4 Code

```
    | Pattern_Any ps ->
        let bs = List.map (eval_pattern env pos v) ps in
        disjunction bs |: SemanticsRule.PAny
```

8.2.5 Formally: sequential case

8.2.6 Formally: concurrent case

8.2.7 Comments

8.3 SemanticsRule.PGeq

8.3.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` is the condition corresponding to being greater or equal than the side-effect-free expression `e`;
- `v'` is the side-effect-free evaluation of `e` in `env`;
- `b` is the boolean value corresponding to whether `v` is greater or equal to `v'`.

8.3.2 Example: SemanticsRule.PGeqTRUE.asl

```
func main () => integer
begin

  let match_me = 42 IN { >= 3 };
  assert match_me == TRUE;

  return 0;
end
```

8.3.3 Example: SemanticsRule.PGeqFALSE.asl

```
func main () => integer
begin

  let match_me = 3 IN { >= 42 };
  assert match_me == FALSE;

  return 0;
end
```

8.3.4 Code

```
    | Pattern_Geq e ->
        let* v' = eval_expr_sef env e in
        B.binop GEQ v v' |: SemanticsRule.PGeq
```

8.3.5 Formally: sequential case

8.3.6 Formally: concurrent case

8.3.7 Comments

8.4 SemanticsRule.PLeq

8.4.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` is the condition corresponding to being less or equal than the side-effect-free expression `e`;
- `v'` is the side-effect-free evaluation of `e` in `env`;
- `b` is the boolean value corresponding to whether `v` is less or equal to `v'`.

8.4.2 Example: SemanticsRule.PLeqTRUE.asl

```
func main () => integer
begin

  let match_me = 3 IN { <= 42 };
  assert match_me == TRUE;

  return 0;
end
```

8.4.3 Example: SemanticsRule.PLeqFALSE.asl

```
func main () => integer
begin

  let match_me = 42 IN { <= 3 };
  assert match_me == FALSE;

  return 0;
end
```

8.4.4 Code

```
    | Pattern_Leq e ->
        let* v' = eval_expr_sef env e in
        B.binop LEQ v v' |: SemanticsRule.PLeq
```

8.4.5 Formally: sequential case

8.4.6 Formally: concurrent case

8.4.7 Comments

8.5 SemanticsRule.PNot

8.5.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` is the negation of the pattern `p'`;
- `b'` is the result of the evaluation of the pattern `p'` under environment `env` with respect to the value `v`;
- `b` is the boolean negation of `b'`.

8.5.2 Example: SemanticsRule.PNotTRUE.asl

```
func main () => integer
begin

  let match_me = 42 IN !{ 3 };
  assert match_me == TRUE;

  return 0;
end
```

8.5.3 Example: SemanticsRule.PNotFALSE.asl

```
func main () => integer
begin

  let match_me = 42 IN !{ 42 };
  assert match_me == FALSE;

  return 0;
end
```

8.5.4 Code

```
  | Pattern_Not p' ->
      let* b' = eval_pattern env pos v p' in
      B.unop BNOT b' |: SemanticsRule.PNot
```

8.5.5 Formally: sequential case

8.5.6 Formally: concurrent case

8.5.7 Comments

8.6 SemanticsRule.PRange

8.6.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` is the condition corresponding to being greater or equal to `e1`, and lesser or equal to `e2`;
- `e1` and `e2` are side-effect-free expressions;
- `v1` is the side-effect-free evaluation of `e1` in `env`;
- `v2` is the side-effect-free evaluation of `e2` in `env`;
- `b1` is the boolean value corresponding to whether `v` is greater or equal to `v1`.
- `b2` is the boolean value corresponding to whether `v` is less or equal to `v2`.
- `b` is the boolean conjunction of `b1` and `b2`.

8.6.2 Example: SemanticsRule.PRangeTRUE.asl

```
func main () => integer
begin

  let match_me = 42 IN {3..42};
  assert match_me == TRUE;

  return 0;
end
```

8.6.3 Example: SemanticsRule.PRangeFALSE.asl

```
func main () => integer
begin

  let match_me = 1 IN {3..42};
  assert match_me == FALSE;

  return 0;
end
```

8.6.4 Code

```
    | Pattern_Range (e1, e2) ->
        let* b1 =
          let* v1 = eval_expr_sef env e1 in
          B.binop GEQ v v1
        and* b2 =
          let* v2 = eval_expr_sef env e2 in
          B.binop LEQ v v2
        in
        B.binop BAND b1 b2 |: SemanticsRule.PRange
```

8.6.5 Formally: sequential case

8.6.6 Formally: concurrent case

8.6.7 Comments

8.7 SemanticsRule.PSingle

8.7.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` is the condition corresponding to being equal to the side-effect-free expression `e`;
- `v'` is the side-effect-free evaluation of `e` in environment `env`;
- `b` is the boolean value corresponding to whether `v` is equal to `v'`.

8.7.2 Example: SemanticsRule.PSingleTRUE.asl

```
func main () => integer
begin

  let match_me = 42 IN { 42 };
  assert match_me == TRUE;

  return 0;
end
```

8.7.3 Example: SemanticsRule.PSingleFALSE.asl

```
func main () => integer
begin

  let match_me = 42 IN { 3 };
  assert match_me == FALSE;

  return 0;
end
```

8.7.4 Code

```
    | Pattern_Single e ->
        let* v' = eval_expr_sef env e in
        B.binop EQ_OP v v' |: SemanticsRule.PSingle
```

8.7.5 Formally: sequential case

8.7.6 Formally: concurrent case

8.7.7 Comments

8.8 SemanticsRule.PMask

8.8.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` is a mask `m` of length n (with spaces removed);
- `m_set` is the bitvector value of length n with bit set at index i if the mask requires a bit set at index i, i. e. $\texttt{m}[i] =' 1'$;
- `m_unset` is the bitvector value of length n with bit set at index i if the mask requires a bit unset at index i, i. e. $\texttt{m}[i] =' 0'$;
- `m_specified` is the bitwise disjunction of `m_set` and `m_unset`;
- `nv` is the bitwise negation of `v`;
- `v_set` is the bitwise conjunction of `m_set` and `v`;
- `v_unset` is the bitwise conjunction of `m_unset` and `nv`;
- `v_set_or_unset` is the bitwise disjunction of `v_set` and `v_unset`;
- `b` is the boolean value of the bitwise equality of `v_set_or_unset` and `m_specified`.

8.8.2 Example: SemanticsRule.PMaskTRUE.asl

```
func main () => integer
begin

  let match_me = '101010' IN 'xx1010';
  assert match_me == TRUE;

  return 0;
end
```

8.8.3 Example: SemanticsRule.PMaskFALSE.asl

```
func main () => integer
begin

  let match_me = '101010' IN '0x1010';
  assert match_me == FALSE;

  return 0;
end
```

8.8.4 Code

```
    | Pattern_Mask m ->
        let bv bv = L_BitVector bv |> B.v_of_literal in
        let m_set = Bitvector.mask_set m
        and m_unset = Bitvector.mask_unset m in
```

```
        let m_specified = Bitvector.logor m_set m_unset in
        let* nv = B.unop NOT v in
        let* v_set = B.binop AND (bv m_set) v
        and* v_unset = B.binop AND (bv m_unset) nv in
        let* v_set_or_unset = B.binop OR v_set v_unset in
        B.binop EQ_OP v_set_or_unset (bv m_specified) |: SemanticsRule.PMask
```

8.8.5 Formally: sequential case

8.8.6 Formally: concurrent case

8.8.7 Comments

8.9 SemanticsRule.PTuple

8.9.1 Prose

Evaluation of the pattern `p` under environment `env` with respect to value `v` is `b` and all of the following apply:

- `p` gives a list of patterns `ps` of length n;
- `v` gives a tuple of values `vs` of length n;
- for all $1 \leq i \leq n$, $\texttt{b}_i$ is the evaluation result of $\texttt{p}_i$ with respect to the value $\texttt{v}_i$ in environment `env`;
- `bs` is the list of all $\texttt{b}_i$ for $1 \leq i \leq n$;
- `b` is the conjunction of the boolean values of `bs`.

8.9.2 Example: SemanticsRule.PTupleTRUE.asl

```
func main () => integer
begin

  let match_me = (3, '101010') IN {( <= 42, 'xx1010')};
  assert match_me == TRUE;

  return 0;
end
```

8.9.3 Example: SemanticsRule.PTupleFALSE.asl

```
func main () => integer
begin

  let match_me = (3, '101010') IN {( >= 42, 'xx1010')};
```

```
  assert match_me == FALSE;

  return 0;
end
```

8.9.4 Code

```
    | Pattern_Tuple ps ->
        let n = List.length ps in
        let* vs = List.init n (fun i -> B.get_index i v) |> sync_list in
        let bs = List.map2 (eval_pattern env pos) vs ps in
        conjunction bs |: SemanticsRule.PTuple
```

8.9.5 Formally: sequential case

8.9.6 Formally: concurrent case

8.9.7 Comments

Chapter 9

Evaluation of Local Declarations

`eval_local_decl ldi env m_init_opt` declares local variables `ldi` in `env` with an optional initialisation value `m_init_opt`. Evaluation of the local variables `ldi` under an environment `env` is either `new_env` or raises an error and one of the following applies:

- SemanticsRule.LDDiscard (see Section 9.1,
- SemanticsRule.LDVar (see Section 9.2,
- SemanticsRule.LDTypedVar (see Section 9.3,
- SemanticsRule.LDTuple (see Section 9.4,
- SemanticsRule.LDTypedTuple (see Section 9.5,
- SemanticsRule.LDUninitialisedTuple (see Section 9.6.

9.1 SemanticsRule.LDDiscard

9.1.1 Prose

Evaluation of the local variables `ldi` under the environment `env` is `new_env` and all of the following apply:

- `ldi` indicates that the initialisation value will be discarded;
- `new_env` is `env`.

9.1.2 Example: SemanticsRule.LDDiscard.asl

In the program:

```
func main () => integer
begin

  var - : integer;
  assert TRUE;

  return 0;
end
```

`var - : integer;` does not modify the environment.

9.1.3 Code

```
| LDI_Discard _ty, _ -> return_normal env |: SemanticsRule.LDDiscard
```

9.1.4 Formally: sequential case

9.1.5 Formally: concurrent case

9.1.6 Comments

9.2 SemanticsRule.LDVar

9.2.1 Prose

Evaluation of the local variables `ldi` under the environment `env` is `new_env` and all of the following apply:

- `ldi` is a variable `x`;
- `m_init_opt` is a value `m`;
- `new_env` is `env` modified to declare `x` as a local variable bound to value `m`.

9.2.2 Example: SemanticsRule.LDVar0.asl

In the program:

```
func main () => integer
begin

  var x = 3;

  assert x == 3;
```

```
  return 0;
end
```

`var x = 3;` binds `x` to the evaluation of `3` in `env`.

9.2.3 Example: SemanticsRule.LDVar1.asl

In the program:

```
func main () => integer
begin

  var x : integer = 3;

  assert x == 3;

  return 0;
end
```

`var x : integer = 3;` binds `x` to the evaluation of `3` in `env`, without type consideration at runtime.

9.2.4 Code

```
    | LDI_Var (x, _ty), Some m ->
        m
        >>= declare_local_identifier env x
        >>= return_normal |: SemanticsRule.LDVar
```

9.2.5 Formally: sequential case

9.2.6 Formally: concurrent case

9.2.7 Comments

9.3 SemanticsRule.LDTypedVar

9.3.1 Prose

Evaluation of the local variables `ldi` under the environment `env` is `new_env` and all of the following apply:

- `ldi` is a variable `x` of type `ty`;
- `m_init_opt` is `None`;
- `new_env` is `env` modified to declare `x` as a local variable bound to the base value of `ty`.

9.3.2 Example: SemanticsRule.LDTypedVar.asl

In the program:

```
func main () => integer
begin

  var x: integer;

  assert x == 0;

  return 0;
end
```

`var x : integer;` binds `x` in `env` to the base value of `integer`.

9.3.3 Code

```
  | LDI_Var (x, Some ty), None ->
      base_value env ty
      >>= declare_local_identifier env x
      >>= return_normal |: SemanticsRule.LDTypedVar
```

9.3.4 Formally: sequential case

9.3.5 Formally: concurrent case

9.3.6 Comments

9.4 SemanticsRule.LDTuple

9.4.1 Prose

Evaluation of the local variables `ldi` under the environment `env` is `new_env` and all of the following apply:

- `ldi` gives a list of local variables `ldis`;
- `m_init_opt` is a list of values `liv`;
- `new_env` is `env` modified to declare each element of `ldis` to be bound to the corresponding value in `liv`.

9.4.2 Example: SemanticsRule.LDTuple.asl

In the program:

```
func main () => integer
begin

  var (x, y, z) = (1, 2, 3);

  assert x == 1 && y == 2 && z == 3;

  return 0;
end
```

`var (x,y,z) = (1,2,3);` binds `x` (resp. `y`, `z`) to the evaluation of `1` (resp. `2`, `3`) in `env`.

9.4.3 Code

```
| LDI_Tuple (ldis, _ty), Some m ->
    let n = List.length ldis in
    let liv = List.init n (fun i -> m >>= B.get_index i) in
    let folder envm ldi' vm =
      let**| env = envm in
      eval_local_decl s ldi' env (Some vm)
    in
    List.fold_left2 folder (return_normal env) ldis liv
    |: SemanticsRule.LDTuple
```

9.4.4 Formally: sequential case

9.4.5 Formally: concurrent case

9.4.6 Comments

9.5 SemanticsRule.LDTypedTuple

9.5.1 Prose

Evaluation of the local variables `ldi` under the environment `env` is `new_env` and all of the following apply:

- `ldi` gives a list of local variables `ldis` and a type `ty`;
- `m_init_opt` is `None`;
- `new_env` is `env` modified to declare each element of `ldis` with type `ty`.

9.5.2 Example: SemanticsRule.LDTypedTuple.asl

In the program:

```
func main () => integer
begin

  var (x,y,z) : integer;

  assert x == 0 && y == 0 && z == 0;

  return 0;
end
```

`var (x,y,z) : integer;` binds `x`, `y` and `z` in `env` to the base value of `integer`.

9.5.3 Code

```
    | LDI_Tuple (_ldis, Some ty), None ->
        let m = base_value env ty in
        eval_local_decl s ldi env (Some m) |: SemanticsRule.LDTypedTuple
```

9.5.4 Formally: sequential case

9.5.5 Formally: concurrent case

9.5.6 Comments

9.6 SemanticsRule.LDUninitialisedTuple

9.6.1 Prose

Evaluation of the local variables `ldi` under the environment `env` is `new_env` and all of the following apply:

- `ldi` gives a list of local variables `ldis`;
- `new_env` is `env` modified to declare each element of `ldis`.

9.6.2 Example: SemanticsRule.LDUninitalisedTuple.asl

In the program:

```
func main () => integer
begin

  var (x : integer, y : boolean);

  assert x == 0 && y == TRUE;

  return 0;
end
```

`var (x : integer, y : boolean);` binds `x` to the base value of `integer` and `y` to the base value of `boolean`.

9.6.3 Code

```
| LDI_Tuple (ldis, None), None ->
    let folder envm ldi' =
      let**| env = envm in
      eval_local_decl s ldi' env None
    in
    List.fold_left folder (return_normal env) ldis
    |: SemanticsRule.LDUninitialisedTuple
```

9.6.4 Formally: sequential case

9.6.5 Formally: concurrent case

9.6.6 Comments

Chapter 10

Evaluation of Statements

Evaluation `eval_stmt env s` of a statement `s` under environment `env` is either a `Throwing`, an interruption `Returning vs` or a new environment `new_env`. Formally, one of the following applies:

- SemanticsRule.SPass (see Section 10.1),
- SemanticsRule.SAssign (see Section 10.2),
- SemanticsRule.SAssignCall (see Section 10.3),
- SemanticsRule.SAssignTuple (see Section 10.4),
- SemanticsRule.SReturnNone (see Section 10.5),
- SemanticsRule.SReturnOne (see Section 10.6),
- SemanticsRule.SReturnSome (see Section 10.7),
- SemanticsRule.SSeq (see Section 10.8),
- SemanticsRule.SCall (see Section 10.9),
- SemanticsRule.SCond (see Section 10.10),
- SemanticsRule.SCase (see Section 10.11),
- SemanticsRule.SAssert (see Section 10.12),
- SemanticsRule.SWhile (see Section 10.13),
- SemanticsRule.SRepeat (see Section 10.14),
- SemanticsRule.SFor (see Section 10.15),
- SemanticsRule.SThrowNone (see Section 10.16),
- SemanticsRule.SThrowSomeTyped (see Section 10.17),

- SemanticsRule.STry (see Section 10.18),
- SemanticsRule.SDeclSome (see Section 10.19),
- SemanticsRule.SDeclNone (see Section 10.20).

10.1 SemanticsRule.SPass

10.1.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

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

10.1.2 Example: SemanticsRule.SPass.asl

In the program:

```
func main () => integer
begin

  pass;

  return 0;
end
```

`pass;` does nothing.

10.1.3 Code

```
| S_Pass -> return_continue env |: SemanticsRule.SPass
```

10.1.4 Formally: sequential case

$$[|\mathsf{pass}|](\mathtt{env}) \triangleq \{(\bot, \mathtt{env})\} \tag{10.1}$$

10.1.5 Formally: concurrent case

$$[|\mathsf{pass}|](\mathtt{env}) \triangleq \{(\bot, \mathtt{env}, \varnothing)\} \tag{10.2}$$

10.1.6 Comments

10.2 SemanticsRule.SAssign

10.2.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` is an assignment `le = re`;
- `v` is the evaluation of the expression `re` under `env` as per Chapter 5;
- `r_env` is `env` modified after evaluation of the expression `re` under `env` as per Chapter 5;
- `new_env` is `r_env` modified after evaluation of `le` under `r_env` with `v`, as per Chapter 6.

10.2.2 Example: SemanticsRule.SAssign.asl

In the program:

```
func main () => integer
begin
  var x : integer = 42;

  x = 3;

  assert x == 3;

  return 0;
end
```

`x = 3;` binds `x` to `3` in the environment where `x` is bound to `42`, and `new_env` is such that `x` is bound to `3`.

10.2.3 Code

```
  | S_Assign (le, re, ver) ->
      let*^ v, env' = eval_expr env re in
      let**| new_env = eval_lexpr ver le env' v in
      return_continue new_env |: SemanticsRule.SAssign
```

10.2.4 Formally: sequential case

$$[|\texttt{le = re}|](\texttt{env}) \triangleq \{(\bot, \texttt{env'}[\texttt{le} \mapsto \texttt{v}]) \mid (\texttt{v}, \texttt{env'}) \in [|\texttt{re}|](\texttt{env})\} \qquad (10.3)$$

10.2.5 Formally: concurrent case

$$[|\texttt{le = re}|](\texttt{env}) \triangleq \left\{ \left(\bot, \texttt{env'}[\texttt{le} \mapsto \texttt{v}], S \xrightarrow{\texttt{asl\_data}} \texttt{W(le)} \right) \;\middle|\; (\texttt{v}, \texttt{env'}, S) \in [|\texttt{re}|](\texttt{E}) \right\} \tag{10.4}$$

10.2.6 Comments

10.3 SemanticsRule.SAssignCall

10.3.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` gives a left-hand-side tuple expression `les` and a subprogram call `(name, args, named_args)`;
- `vs` is the list of values resulting from the evaluation of the subprogram call;
- `env'` is the environment resulting from modifying `env` after the evaluation of the subprogram call;
- `new_env` is the result of modifying `env'` after assigning each values in `vs` to the elements of the tuple `les`.

10.3.2 Example: SemanticsRule.SAssignCall.asl

```
func f(x:integer) => (integer, integer)
begin
  return (x,x+1);
end

func main() => integer
begin
  var a,b : integer;

  (a,b) = f(1);

  assert (a+b == 3);
  return 0;
end
```

given that the function call `f(1)` returns the pair of values `(1,2)`, statement `(a,b) = f(1)` assigns the value `1` to the mutable variable `a` and the value `2` to the mutable variable `b`.

10.3.3 Code

```
  | S_Assign
      ( { desc = LE_Destructuring les; _ },
        { desc = E_Call (name, args, named_args); _ },
        ver )
    when List.for_all lexpr_is_var les ->
      let**| vs, env' = eval_call (to_pos s) name env args named_args in
      let**| new_env = protected_multi_assign ver env' s les vs in
      return_continue new_env |: SemanticsRule.SAssignCall
```

10.3.4 Formally: sequential case

10.3.5 Formally: concurrent case

10.3.6 Comments

10.4 SemanticsRule.SAssignTuple

10.4.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` gives a left-hand-side tuple expression `les` and a tuple expression `exprs`;
- `vs` is the list of values resulting from the evaluation of `exprs`;
- `env'` is the environment resulting from modifying `env` after the evaluation of `exprs`;
- `new_env` is the result of modifying `env'` after assigning each values in `vs` to the elements of the tuple `les`.

10.4.2 Example: SemanticsRule.SAssignTuple.asl

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

  (b,x) = (TRUE,42);

  assert (b && x == 42);
  return 0;
end
```

statement `(b,x)` assigns the value `TRUE` to the mutable variable `b` and the value `42` to the mutable variable `x`.

10.4.3 Code

```
  | S_Assign
      ({ desc = LE_Destructuring les; _ }, { desc = E_Tuple exprs; _ }, ver)
    when List.for_all lexpr_is_var les ->
      let**| vs, env' = eval_expr_list_m env exprs in
      let**| new_env = protected_multi_assign ver env' s les vs in
      return_continue new_env |: SemanticsRule.SAssignTuple
```

10.4.4 Formally: sequential case

10.4.5 Formally: concurrent case

10.4.6 Comments

10.5 SemanticsRule.SReturnNone

10.5.1 Prose

Evaluation of the statement `s` under environment `env` is `Returning vs` and all of the following apply:

- `s` is a `return` statement;
- `vs` is `[]`;
- `new_env` is `env`.

10.5.2 Example: SReturnNoneReturn.asl

The program:

```
func print_me ()
begin

  for i = 0 to 42 do
    if i >= 3 then
      return;
    end
  end
  assert FALSE;

end

func main () => integer
begin

    print_me ();
```

```
    return 0;
end
```

exits the current procedure.

10.5.3 Code

```
    | S_Return None -> return_return env [] |: SemanticsRule.SReturnNone
```

10.5.4 Formally: sequential case

10.5.5 Formally: concurrent case

10.5.6 Comments

10.6 SemanticsRule.SReturnOne

10.6.1 Prose

Evaluation of the statement `s` under environment `env` is `Returning vs` and all of the following apply:

- `s` is a `return` statement;
- `s` gives an expression `e`;
- `v` is the evaluation of `e` under `env`;
- `vs` is `[v]`;
- `new_env` is `env` modified after evaluation of the expression `e` under `env` as per Chapter 5.

10.6.2 Example: SemanticsRule.SReturnOne.asl

In the program:

```
func f () => integer
begin
  var x : integer = 0;
  for i = 0 to 5 do
    x = x + 1;
    assert x == 1; // Only the first loop is executed
    return 3;
  end
end

func main () => integer
```

```
begin

  assert f () == 3;

  return 0;
end
```

`return 3;` exits the current subprogram with value `3`.

10.6.3 Code

```
  | S_Return (Some e) ->
      let** v, env' = eval_expr env e in
      let* () =
        B.on_write_identifier (return_identifier 0) (IEnv.get_scope env') v
      in
      return_return env' [ v ] |: SemanticsRule.SReturnOne
```

10.6.4 Formally: sequential case

10.6.5 Formally: concurrent case

10.6.6 Comments

10.7 SemanticsRule.SReturnSome

10.7.1 Prose

Evaluation of the statement `s` under environment `env` is `Returning vs` and all of the following apply:

- `s` is a `return` statement;
- `s` gives a list of expressions `es`;
- `vs` is the result of the evaluation of each element of the list `es` under `env` as per Chapter 5;
- `new_env` is `env` modified after the evaluation of each element of the list `es` under `env` as per Chapter 5.

10.7.2 Example: SemanticsRule.SReturnSome.asl

In the program:

```
func f () => (integer, integer)
begin
  var x: integer = 0;
```

```
  for i = 0 to 5 do
    x = x + 1;
    assert x == 1; // Only the first loop is executed
    return (3, 42);
  end
end

func main () => integer
begin

  let (x, y) = f ();
  assert x == 3 && y == 42;

  return 0;
end
```

`return (3, 42);` exits the current subprogram with value `(3, 42)`.

10.7.3 Code

```
  | S_Return (Some { desc = E_Tuple es; _ }) ->
      let**| ms, env = eval_expr_list_m env es in
      let scope = IEnv.get_scope env in
      let folder acc m =
        let*| i, vs = acc in
        let* v = m in
        let* () = B.on_write_identifier (return_identifier i) scope v in
        return (i + 1, v :: vs)
      in
      let*| _i, vs = List.fold_left folder (return (0, [])) ms in
      return_return env (List.rev vs) |: SemanticsRule.SReturnSome
```

10.7.4 Formally: sequential case

$$[|\texttt{return } e|](E) \triangleq [|e|](E) \tag{10.5}$$

10.7.5 Formally: concurrent case

$$[|\texttt{return } e|](E) \triangleq [|e|](E) \tag{10.6}$$

10.7.6 Comments

10.8 SemanticsRule.SSeq

10.8.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` is a sequence statement `s1; s2`;
- `env'` is `env` modified after evaluation of `s1`;
- `new_env` is `env'` modified after evaluation of `s2`.

10.8.2 Example: SemanticsRule.SSeq.asl

In the program:

```
func main () => integer
begin

  let x = 3;
  let y = x + 1;

  assert x == 3 && y == 4;

  return 0;
end
```

`let x = 3; let y = x + 1` evaluates `let x = 3` then `let y = x + 1`.

10.8.3 Code

```
  | S_Seq (s1, s2) ->
      let*> env' = eval_stmt env s1 in
      eval_stmt env' s2 |: SemanticsRule.SSeq
```

10.8.4 Formally: sequential case

The semantics of `s1; s2` is the semantics of `s2` applied to the results of the semantics of `s1` if they do not perform an early return, in which case it is the semantics of `s1`.

$$[|\texttt{s1; s2}|](\texttt{env}) \triangleq \left\{ (\texttt{v?}, \texttt{new\_env}) \;\middle|\; \begin{array}{c} (\texttt{v}, \texttt{new\_env}) \in [|\texttt{s1}|](\texttt{env}) \\ \text{or} \left(\begin{array}{c} (\bot, \texttt{env'}) \in [|\texttt{s1}|](\texttt{env}) \\ \text{and } (\texttt{v?}, \texttt{new\_env}) \in [|\texttt{s2}|](\texttt{env'}) \end{array} \right) \end{array} \right\} \tag{10.7}$$

10.8.5 Formally: concurrent case

The evaluation of two statements introduces an asl_po arrow between the two graphs produced by their interpretations:

$$[|\texttt{s1; s2}|](\texttt{env}) \triangleq \left\{ (\texttt{v?}, \texttt{new\_env}, S) \;\middle|\; \begin{array}{c} (\texttt{v}, \texttt{new\_env}, S) \in [|\texttt{s1}|](\texttt{env}) \\ \text{or} \left(\begin{array}{c} (\bot, \texttt{env'}, S') \in [|\texttt{s1}|](\texttt{env}) \\ \text{and } (v, \texttt{new\_env}, S'') \in [|\texttt{s2}|](\texttt{env'}) \\ \text{and } S = S' \xrightarrow{\texttt{asl\_po}} S'' \end{array} \right) \end{array} \right\} \tag{10.8}$$

10.8.6 Comments

10.9 SemanticsRule.SCall

10.9.1 Prose

Evaluation of the statement s under environment env is new_env and all of the following apply:

- s is a call statement;
- s gives a subprogram name name with actual arguments actual_args;
- env' is env modified after evaluation of the subprogram call;
- new_env is env'.

10.9.2 Example: SemanticsRule.SCall.asl

In the program:

```
func main () => integer
begin

  assert Zeros(3) == '000';

  return 0;
end
```

Zeros(3) evaluates to '000'.

10.9.3 Code

```
    | S_Call (name, args, named_args) ->
        let**| returned, env' = eval_call (to_pos s) name env args named_args in
        let () = assert (returned = []) in
        return_continue env' |: SemanticsRule.SCall
```

10.9.4 Formally: sequential case

10.9.5 Formally: concurrent case

10.9.6 Comments

10.10 SemanticsRule.SCond

10.10.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` gives a condition `cond` and two conditional blocks `s1` and `s2`;
- `v_cond` is the evaluation of `cond`;
- `new_env` is `env` modified after evaluation of `s1` or `s2` depending on `v_cond`.

10.10.2 Example: SemanticsRule.SCond.asl

The program:

```
func main () => integer
begin

  if TRUE
    then assert TRUE;
    else assert FALSE;
  end

  return 0;
end
```

does not raise any Assertion Error.

10.10.3 Code

```
| S_Cond (e, s1, s2) ->
    let*^ v, env’ = eval_expr env e in
    let*= s’ = choice v s1 s2 in
    eval_block env’ s’ |: SemanticsRule.SCond
```

10.10.4 Formally: sequential case

The semantics of a conditional statement if e then s_1 else s_2 end chooses between the semantics of s_1 or s_2 depending on the evaluation of e:

$$\left[\left|\begin{array}{l}\texttt{if } e\\ \texttt{then } s_1\\ \texttt{else } s_2 \texttt{ end}\end{array}\right|\right](E) \triangleq \left\{ (v?, E'') \;\middle|\; \begin{array}{l} \qquad (b, E') \in [|e|](E) \\ \text{and } \left(\begin{array}{l} \quad (b = \texttt{true} \text{ and } s' = s_1) \\ \text{or } (b = \texttt{false} \text{ and } s' = s_2) \end{array}\right) \\ \text{and } (v?, E'') \in [|s'|](E') \end{array} \right\} \tag{10.9}$$

10.10.5 Formally: concurrent case

A conditional statement introduces control dependencies `asl_ctrl` between its condition and its body:

$$\left[\left|\begin{array}{l}\texttt{if } e\\ \texttt{then } s_1\\ \texttt{else } s_2 \texttt{ end}\end{array}\right|\right](E) \triangleq \left\{ (v, E'', S'') \;\middle|\; \begin{array}{l} \qquad (b, E', S) \in [|e|](E) \\ \text{and } \left(\begin{array}{l} \quad (b = \texttt{true} \text{ and } s' = s_1) \\ \text{or } (b = \texttt{false} \text{ and } s' = s_2) \end{array}\right) \\ \text{and } (v, E'', S') \in [|s'|](E') \\ \text{and } S'' = S \xrightarrow{\texttt{asl\_ctrl}} S' \end{array} \right\} \tag{10.10}$$

10.10.6 Comments

10.11 SemanticsRule.SCase

10.11.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` gives a condition `cond` and a number of statements `s_1`, $\cdots$, `s_n`;
- `v_cond` is the evaluation of `cond`;
- `new_env` is `env'` modified after evaluation of one of the statements `s_i` depending on `v_cond`.

10.11.2 Example: SemanticsRule.SCase.asl

The program:

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

```
  case 3 of
    when 42: assert FALSE;
    when <= 42: assert TRUE;
    otherwise: assert FALSE;
  end

  return 0;
end
```

uses the second `when` clause because `3` is less than `42`.

10.11.3 Code

```
    | S_Case _ -> case_to_conds s |> eval_stmt env |: SemanticsRule.SCase
```

10.11.4 Formally: sequential case

10.11.5 Formally: concurrent case

10.11.6 Comments

10.12 SemanticsRule.SAssert

10.12.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` or an error and all of the following apply: All of the following apply:

- `s` is an `assert` statement;
- `s` gives an expression `e`;
- `v` is the evaluation of the expression `e` as per Chapter 5;
- One of the following applies:
  - `v` is `true` and `new_env` is `env`,
  - an "AssertionFailed" error is raised.

10.12.2 Example: SemanticsRule.SAssertOk.asl

In the program:

```
func main () => integer
begin

  assert (42 != 3);

  return 0;
end
```

`assert (42 != 3);` ensures that `3` is not equal to `42`.

10.12.3 Example: SemanticsRule.SAssertNo.asl

In the program:

```
func main () => integer
begin

  assert (42 == 3);

  return 0;
end
```

`assert (42 == 3);` raises an "`AssertionFailed`" error.

10.12.4 Code

```
    | S_Assert e ->
        let*^ v, env' = eval_expr env e in
        let*= b = choice v true false in
        if b then return_continue env'
        else fatal_from e @@ Error.AssertionFailed e |: SemanticsRule.SAssert
```

10.12.5 Formally: sequential case

10.12.6 Formally: concurrent case

10.12.7 Comments

10.13 SemanticsRule.SWhile

10.13.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` is a `while` statement;
- `s` gives an expression `e` and a loop body `body`;
- `new_env` is `env` modified after evaluation of the loop `(e,body)` as per Section 12.1.

10.13.2 Example: SemanticsRule.SWhile.asl

The program:

```
func main () => integer
begin

var i: integer = 0;
  while i <= 3 do
    assert i <= 3;
    i = i + 1;
   end

  return 0;
end
```

prints “0123”.

10.13.3 Code

```
    | S_While (e, body) ->
        let env = IEnv.tick_push env in
        eval_loop true env e body |: SemanticsRule.SWhile
```

10.13.4 Formally: sequential case

10.13.5 Formally: concurrent case

10.13.6 Comments

10.14 SemanticsRule.SRepeat

10.14.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` is a `repeat` statement;
- `s` gives an expression `e` and a loop body `body`;
- `new_env` is `env` modified after evaluation of the loop `(e,body)` as per Section 12.1.

10.14.2 Example: SemanticsRule.SRepeat.asl

The program:

```
func main () => integer
begin

  var i: integer = 0;
```

```
  repeat
    assert i <= 3;
    i = i + 1;
  until i > 3;

  return 0;
end
```

prints "0123".

10.14.3 Code

```
    | S_Repeat (body, e) ->
        let*> env = eval_block env body in
        let env = IEnv.tick_push_bis env in
        eval_loop false env e body |: SemanticsRule.SRepeat
```

10.14.4 Formally: sequential case

10.14.5 Formally: concurrent case

10.14.6 Comments

10.15 SemanticsRule.SFor

10.15.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` is a `for` statement;
- `s` gives `(id,e1,dir,e2,s)`;
- `new_env` is `env` modified after evaluation of the `for` loop `(id,e1,dir,e2,s)` as per Section 12.2.

10.15.2 Example: SemanticsRule.SFor.asl

The program:

```
func main () => integer
begin

  for i = 0 to 3 do
    assert i <= 3;
  end
```

```
  return 0;
end
```

prints "0123".

10.15.3 Code

```
| S_For (id, e1, dir, e2, s) ->
    let* v1 = eval_expr_sef env e1 and* v2 = eval_expr_sef env e2 in
    (* By typing *)
    let undet = B.is_undetermined v1 || B.is_undetermined v2 in
    let*| env = declare_local_identifier env id v1 in
    let env = if undet then IEnv.tick_push_bis env else env in
    let*> env = eval_for undet env id v1 dir v2 s in
    let env = if undet then IEnv.tick_pop env else env in
    IEnv.remove_local id env |> return_continue |: SemanticsRule.SFor
```

10.15.4 Formally: sequential case

10.15.5 Formally: concurrent case

10.15.6 Comments

10.16 SemanticsRule.SThrowNone

10.16.1 Prose

All of the following apply:

- s is a throw statement which gives no expression;
- new_env is env;
- an exception is thrown with new_env.

10.16.2 Example: SemanticsRule.SThrowNone.asl

The program:

```
type MyExceptionType of exception{ a: integer };

func main () => integer
begin

  try
    try
      throw MyExceptionType { a = 42 };
    catch
```

```
      when MyExceptionType => throw;
      otherwise => assert FALSE;
    end
    assert FALSE;

  catch
    when exn: MyExceptionType =>
      assert exn.a == 42;
    otherwise => assert FALSE;
  end

  return 0;
end
```

throws a "`MyException`" exception.

10.16.3 Code

```
| S_Throw None -> return (Throwing (None, env)) |: SemanticsRule.SThrowNone
```

10.16.4 Formally: sequential case

10.16.5 Formally: concurrent case

10.16.6 Comments

10.17 SemanticsRule.SThrowSomeTyped

10.17.1 Prose

All of the following apply:

- `s` is a `throw` statement which gives an expression `e` and a type `t`;
- `v` is the result of evaluating the expression `e` in `env`;
- `new_env` is the environment modified after evaluating the expression `e` in `env`;
- an exception is thrown with `v` and `new_env`.

10.17.2 Example: SemanticsRule.SThrowSomeTyped.asl

The program:

```
type MyExceptionType of exception{ a: integer };

func main () => integer
begin
```

```
  try
    throw MyExceptionType { a = 42 };
  catch
    when exn: MyExceptionType =>
      assert exn.a == 42;
    otherwise => assert FALSE;
  end

  return 0;
end
```

throws a "`MyException {a: 3, b: 42}`" exception.

10.17.3 Code

```
| S_Throw (Some (e, Some t)) ->
    let** v, new_env = eval_expr env e in
    let name = throw_identifier () and scope = Scope_Global in
    let* () = B.on_write_identifier name scope v in
    return (Throwing (Some ((v, name, scope), t), new_env))
    |: SemanticsRule.SThrowSomeTyped
```

10.17.4 Formally: sequential case

10.17.5 Formally: concurrent case

10.17.6 Comments

10.18 SemanticsRule.STry

10.18.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` is a `try` statement `[(s', catchers, otherwise_opt)`;
- `s_m` is the evaluation of the block `s'` under `env`;
- `new_env` is `env` modified after evaluation of the catchers `(catchers otherwise_opt s_m)` as per Chapter 13.

10.18.2 Example: SemanticsRule.STry.asl

The program:

```
type MyExceptionType of exception{ a: integer };

func main () => integer
begin

  try
    throw MyExceptionType { a = 42 };

  catch
    when MyExceptionType => assert TRUE;
    otherwise => assert FALSE;
  end

  return 0;
end
```

does not raise any Assertion error, and the program terminates with the exit code 0.

10.18.3 Code

```
| S_Try (s, catchers, otherwise_opt) ->
    let s_m = eval_block env s in
    eval_catchers env catchers otherwise_opt s_m |: SemanticsRule.STry
```

10.18.4 Formally: sequential case

10.18.5 Formally: concurrent case

10.18.6 Comments

10.19 SemanticsRule.SDeclSome

10.19.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` is a declaration `(ldi, Some e)`;
- `new_env` is `env` modified after evaluation of the local declaration `ldi env (Some m)` as per Chapter 9.

10.19.2 Example: SemanticsRule.SDeclSome.asl

The program:

```
func main () => integer
begin

  let x = 3;

  assert x == 3;

  return 0;
end
```

`let x = 3;` binds `x` to `3` in the empty environment.

10.19.3 Code

```
| S_Decl (_ldk, ldi, Some e) ->
    let*^ m, env1 = eval_expr env e in
    let**| env' = eval_local_decl s ldi env1 (Some m) in
    return_continue env' |: SemanticsRule.SDeclSome
```

10.19.4 Formally: sequential case

10.19.5 Formally: concurrent case

10.19.6 Comments

10.20 SemanticsRule.SDeclNone

10.20.1 Prose

Evaluation of the statement `s` under environment `env` is `new_env` and all of the following apply:

- `s` is a declaration `(ldi, None)`;
- `new_env` is `env` modified after evaluation of the local declaration `ldi env None` as per Chapter 9.

10.20.2 Example: SemanticsRule.SDeclNone.asl

In the program:

```
func main () => integer
begin

  var x: integer;

  assert x == 0;
```

```
  return 0;
end
```

`var x : integer;` binds `x` in `env` to the base value of `integer`.

10.20.3 Code

```
    | S_Decl (_dlk, ldi, None) ->
        let**| env' = eval_local_decl s ldi env None in
        return_continue env' |: SemanticsRule.SDeclNone
```

10.20.4 Formally: sequential case

10.20.5 Formally: concurrent case

10.20.6 Comments

Chapter 11

Evaluation of Blocks

11.1 SemanticsRule.Block

11.1.1 Prose

`eval_block env stm` is `new_env` and all of the following applies:

- `block_env'` is `env` modified after the evaluation of the statement `stm` as per Chapter 10;
- `new_env` is `block_env'` after restoring the variable bindings of `env` with the updated values of `block_env'`.

11.1.2 Example: SemanticsRule.Block.asl

In the program:

```
func main() => integer
begin
  var x : integer = 1;

  if TRUE then x = 2; let y = 2;  else pass; end
  let y = 1;
  assert (x == 2 && y == 1);

  return 0;
end
```

the conditional statement `if TRUE then... end;` defines a block structure. Thus, the scope of the declaration `let y = 2;` is limited to its declaring block—or the binding for `y` no longer exists once the block is exited. As a consequence, the subsequent declaration `let y = 1` is valid. By contrast, the assigment of the mutable variable `x` persists after block end. However, observe that `x` is defined before the block and hence still exists after the block.

11.1.3 Code

```
and eval_block env stm =
  let block_env = IEnv.push_scope env in
  let*> block_env' = eval_stmt block_env stm in
  IEnv.pop_scope env block_env' |> return_continue |: SemanticsRule.Block
```

11.1.4 Formally: sequential case

11.1.5 Formally: concurrent case

11.1.6 Comments

Chapter 12

Evaluation of Loops

The evaluation of loop is a common part of the evaluation of multiple loop statements. For example, the semantic rule *Loop* is used by the semantic rule *SWhile* at Section 10.13 and the semantic rule *SRepeat* at Section 10.14. The semantic rule *For* is only used by the semantic rule *SFor* at Section 10.15.

12.1 SemanticsRule.Loop

`eval_loop is_while env e_cond body` evaluates `body` in `env`: this is either an interruption `Returning vs`, a `Throwing` or a new environment `new_env`.

12.1.1 Prose

`cond_m` evaluates to `e_cond` or `not e_cond` as determined by `is_while` and one of the following applies:

- All of the following apply:
  - * `cond_m` evaluates to `false`;
  - * `new_env` is `env`—the loop is exited.
- All of the following apply:
  - * `cond_m` evaluates to `true`;
  - * `env1` is `env` modified after the evaluation of the statement `body`—this step might affect the value of `cond_m` eventually leading to exiting the loop;
  - * `new_env` is `env1` modified after the evaluation of `eval_loop is_while env e_cond body`.

12.1.2 Example: SemanticsRule.Loop.asl

The program:

```
func main () => integer
begin

  var i: integer = 0;

  while i <= 3 do
    assert i <= 3;
    i = i + 1;
  end

  return 0;
end
```

does not raise any Assertion Error and the program terminates with exit code 0.

12.1.3 Code

```
  and eval_loop is_while env e_cond body : stmt_eval_type =
    (* Name for warn messages. *)
    let loop_name = if is_while then "While loop" else "Repeat loop" in
    (* Continuation in the positive case. *)
    let loop env =
      let*> env1 = eval_block env body in
      eval_loop is_while env1 e_cond body
    in
    (* First we evaluate the condition *)
    let*^ cond_m, env = eval_expr env e_cond in
    (* Depending if we are in a while or a repeat, we invert that condition. *)
    let cond_m = if is_while then cond_m else cond_m >>= B.unop BNOT in
    (* If needs be, we tick the unrolling stack before looping. *)
    B.delay cond_m @@ fun cond cond_m ->
    let binder = bind_maybe_unroll loop_name (B.is_undetermined cond) in
    (* Real logic: if condition is validated, we loop, otherwise we continue to
       the next statement. *)
    choice cond_m loop return_continue
    >>*= binder (return_continue env)
    |: SemanticsRule.Loop
```

12.1.4 Formally: sequential case

12.1.5 Formally: concurrent case

12.1.6 Comments

12.2 SemanticsRule.For

`eval_for undet env index_name v_start dir v_end body` evaluates `body` in `env`: this is either an interruption `Returning vs` or a new environment `new_env`.

12.2.1 Prose

`cond_m` evaluates to `leq v_end v_start` or `geq v_end v_start` as determined by `dir` and one of the following applies:

- All of the following apply:
  - `cond_m` evaluates to `true`;
  - `new_env` is `env` as the loop is exited.
- All of the following apply:
  - `cond_m` evaluates to `false`;
  - `env1` is `env` modified after the evaluation of the statement `body`;
  - `env2` is `env1` modified such that `index_name` is bound to `v_step`;
  - `v_step` evaluates to `v_start+1` or `v_start-1` as determined by `dir`;
  - `new_env` is `env2` modified after the evaluation of `eval_for undet env index_name v_step dir v_end body`.

12.2.2 Example: SemanticsRule.For.asl

The program:

```
func main () => integer
begin

  for i = 0 to 3 do
    assert i <= 3;
  end

  return 0;
end
```

does not raise any assertion error, and the program terminates with exit-code 0.

12.2.3 Code

```
and eval_for undet (env : env) index_name v_start dir v_end body :
    stmt_eval_type =
  (* Evaluate the condition: "Is the for loop terminated?" *)
  let cond_m =
    let op = match dir with Up -> LT | Down -> GT in
    let* () = B.on_read_identifier index_name (IEnv.get_scope env) v_start in
    B.binop op v_end v_start
  in
  (* Increase the loop counter *)
  let step env index_name v_start dir =
    let op = match dir with Up -> PLUS | Down -> MINUS in
    let* () = B.on_read_identifier index_name (IEnv.get_scope env) v_start in
    let* v_step = B.binop op v_start one in
    let* env = assign_local_identifier env index_name v_step in
    return (v_step, env)
  in
  (* Continuation in the positive case. *)
  let loop env =
    bind_maybe_unroll "For loop" undet (eval_block env body) @@ fun env1 ->
    let*| v_step, env2 = step env1 index_name v_start dir in
    eval_for undet env2 index_name v_step dir v_end body
  in
  (* Real logic: if condition is validated, we continue to the next
     statement, otherwise we loop. *)
  choice cond_m return_continue loop >>*= fun kont ->
  kont env |: SemanticsRule.For
```

12.2.4 Formally: sequential case

12.2.5 Formally: concurrent case

12.2.6 Comments

Chapter 13

Evaluation of Catchers

`eval_catchers env catchers otherwise_opt s_m`, given the result `s_m` of the evaluation of a statement under environment `env` is `res` which is either a `Throwing (v, v_ty, env_throw)`, an interruption `Returning vs` or a new environment `new_env`. Formally, one of the following applies:

- SemanticsRule.Catch (see Section 13.1),
- SemanticsRule.CatchNamed (see Section 13.2),
- SemanticsRule.CatchOtherwise (see Section 13.3),
- SemanticsRule.CatchNone (see Section 13.4),
- SemanticsRule.CatchNoThrow (see Section 13.5).

13.1 SemanticsRule.Catch

13.1.1 Prose

All of the following apply:

- `s_m` is `Throwing (v, v_ty, env_throw)`;
- `catcher` is the first catcher in `catchers` that matches `v_ty`;
- `catcher` does not declare a name;
- `catcher` gives a statement `s`;
- One of the following applies:
  * `env_throw` and `env` have the same scope, and `env1` is `env_throw`;
  * `env1` is the environment formed with the global part of `env_throw` and the local part of `env`;

- One of the following applies:

  * `Throwing (None, None, env_throw1)` is the result of the evaluation of the block `s` in `env1`, and `res` is `Throwing (v, v_ty, env_throw1)`;

  * `res` is the result of the evaluation of the block `s` in `env1`.

13.1.2 Example: SemanticsRule.Catch.asl

The program:

```
type MyExceptionType of exception{};

func main () => integer
begin

    try
      throw MyExceptionType {};
      assert FALSE;
    catch
      when MyExceptionType =>
        assert TRUE;
      otherwise =>
        assert FALSE;
    end

  return 0;
end
```

prints “`MyException`”.

13.1.3 Code

```
match catcher with
| None, _e_ty, s ->
    eval_block env1 s
    |> rethrow_implicit (v, v_ty)
    |: SemanticsRule.Catch
```

13.1.4 Formally: sequential case

13.1.5 Formally: concurrent case

13.1.6 Comments

13.2 SemanticsRule.CatchNamed

13.2.1 Prose

All of the following apply:

- `s_m` is `Throwing (v, v_ty, env_throw)`;
- `catcher` is the first catcher in `catchers` that matches `v_ty`;
- `catcher` declares a name `name`;
- `catcher` gives a statement `s`;
- One of the following applies:
  - ∗ `env_throw` and `env` have the same scope, and `env1` is `env_throw`;
  - ∗ `env1` is the environment formed with the global part of `env_throw` and the local part of `env`;
  - ∗ `env2` is `env1` modified after binding locally `name` to the exception `v` raised by `s_m`;
- One of the following applies:
  - ∗ `Throwing (None, None, env_throw1)` is the result of the evaluation of the block `s` in `env2`, and `res` is `Throwing (v, v_ty, env_throw1)`;
  - ∗ `env3` is `env2` modified after the evaluation of the block `s` in `env2`, and `new_env` is `env3` modified after unbinding `name` from `env3`.
  - ∗ `res` is the result of the evaluation of the block `s` in `env2`.

13.2.2 Example: SemanticsRule.CatchNamed.asl

The program:

```
type MyExceptionType of exception{ msg: integer };

func main () => integer
begin

    try
      throw MyExceptionType { msg=42 };
    catch
      when exn: MyExceptionType =>
```

```
        assert exn.msg == 42;
    otherwise =>
      assert FALSE;
    end

  return 0;
end
```

prints “`My exception with my message`”.

13.2.3 Code

```
| Some name, _e_ty, s ->
    (* If the exception is declared to be used in the catcher, we
       update the environment before executing [s]. *)
    let*| env2 =
      read_value_from v |> declare_local_identifier_m env1 name
    in
    (let*> env3 = eval_block env2 s in
     IEnv.remove_local name env3 |> return_continue)
    |> rethrow_implicit (v, v_ty)
    |: SemanticsRule.CatchNamed)
```

13.2.4 Formally: sequential case

13.2.5 Formally: concurrent case

13.2.6 Comments

13.3 SemanticsRule.CatchOtherwise

13.3.1 Prose

All of the following apply:

- `s_m` is `Throwing (v, v_ty, env_throw)`;
- `otherwise_opt` is `Some s`;
- no catcher matches `v_ty`;
- One of the following applies:
  - ∗ `env_throw` and `env` have the same scope, and `env1` is `env_throw`;
  - ∗ `env1` is the environment formed with the global part of `env_throw` and the local part of `env`;
- One of the following applies:

  * `Throwing (None, None, env_throw1)` is the result of the evaluation of the block `s` in `env1`, and `res` is `Throwing (v, v_ty, env_throw1)`;
  * `res` is the result of the evaluation of the block `s` in `env1`.

13.3.2 Example: SemanticsRule.CatchOtherwise.asl

The program:

```
type MyExceptionType1 of exception{};
type MyExceptionType2 of exception{};

func main () => integer
begin

     try
       throw MyExceptionType1 {};
       assert FALSE;
     catch
       when MyExceptionType2 =>
         assert FALSE;
       otherwise =>
         assert TRUE;
     end

  return 0;
end
```

prints “`Another exception`”.

13.3.3 Code

```
      | Some s ->
          eval_block env1 s
          |> rethrow_implicit (v, v_ty)
          |: SemanticsRule.CatchOtherwise
```

13.3.4 Formally: sequential case

13.3.5 Formally: concurrent case

13.3.6 Comments

13.4 SemanticsRule.CatchNone

13.4.1 Prose

All of the following apply:

- s_m is Throwing (v, v_ty, env_throw);
- otherwise_opt is None;
- no catcher matches v_ty;
- new_env is env.

13.4.2 Example: SemanticsRule.CatchNone.asl

The program:

```
type MyExceptionType1 of exception{};
type MyExceptionType2 of exception{};

func main () => integer
begin

  try
     try
       throw MyExceptionType1 {};
       assert FALSE;
     catch
       when MyExceptionType2 =>
         assert FALSE;
     end
  catch MyExceptionType1;
    assert TRUE;
  end

  return 0;
end
```

does not print anything.

13.4.3 Code

```
            | None -> s_m |: SemanticsRule.CatchNone))
```

13.4.4 Formally: sequential case

13.4.5 Formally: concurrent case

13.4.6 Comments

13.5 SemanticsRule.CatchNoThrow

13.5.1 Prose

All of the following apply:

- s_m is not Throwing;
- res is s_m.

13.5.2 Example: SemanticsRule.CatchNoThrow.asl

The program:

```
type MyExceptionType of exception{};

func main () => integer
begin

    try
      assert TRUE;
    catch
      when MyExceptionType =>
        assert FALSE;
      otherwise =>
        assert FALSE;
    end

  return 0;
end
```

prints “No exception raised”.

13.5.3 Code

```
    | Normal _ | Throwing (None, _) -> s_m |: SemanticsRule.CatchNoThrow
```

13.5.4 Comments

Chapter 14

Evaluation of Functions

`eval_func genv name pos actual_args params` evaluates the subprogram named `name` in the global environment `genv`, with `actual_args` the list of actual arguments, and `params` the list of arguments deduced by type equality. This is a new global environment `new_genv` and a list of values `vs`, or an error is raised. One of the following applies:

- SemanticsRule.FUndefIdent (see Section 14.1),
- SemanticsRule.FPrimitive (see Section 14.2),
- SemanticsRule.FCall (see Section 14.3).

14.1 SemanticsRule.FUndefIdent

14.1.1 Prose

All of the following apply:

- `name` is undeclared in `genv`;
- an `UndefinedIdentifier` error is raised.

14.1.2 Example: SemanticsRule.FUndefIdent.asl

The program:

```
func main () => integer
begin

     foo ();

  return 0;
end
```

raises an `UndefinedIdentifier "Foo"` error.

14.1.3 Code

```
  | None ->
      fatal_from pos @@ Error.UndefinedIdentifier name
      |: SemanticsRule.FUndefIdent
```

14.1.4 Formally: sequential case

14.1.5 Formally: concurrent case

14.1.6 Comments

14.2 SemanticsRule.FPrimitive

14.2.1 Prose

All of the following apply:

- `name` is bound in `genv` to a primitive subprogram with a body `body`;
- `new_genv` is `genv`;
- `vs` is the application of `body` on `actual_args`.

14.2.2 Example

In the program:

```
func main () => integer
begin

  print("Hello, world!");

  return 0;
end
```

`print ("Hello, world!");` calls the primitive `print` on the evaluation of `"Hello, world!"`.

14.2.3 Code

```
  | Some (r, { body = SB_Primitive body; _ }) ->
      let scope = Scope_Local (name, !r) in
      let () = incr r in
      let* ms = body actual_args in
      let _, vsm =
        List.fold_right
          (fun m (i, acc) ->
```

```
        let x = return_identifier i in
        let m' =
          let*| v =
            let* v = m in
            let* () = B.on_write_identifier x scope v in
            return (v, x, scope)
          and* vs = acc in
          return (v :: vs)
        in
        (i + 1, m'))
      ms
      (0, return [])
  in
  let*| vs = vsm in
  return_normal (vs, genv) |: SemanticsRule.FPrimitive
```

14.2.4 Formally: sequential case

14.2.5 Formally: concurrent case

14.2.6 Comments

14.3 SemanticsRule.FCall

14.3.1 Prose

All of the following apply:

- `name` is bound in `genv` to a subprogram with a list of formal arguments `arg_decls` and a body statement `body`;
- `env1` is the environment made of `genv` and the empty local environment,
- `env2` is `env1` modified so that each formal argument in `arg_decls` is locally bound to the corresponding actual argument in `actual_args`;
- `env3` is `env2` modified so that each parameter in `params` is declared;
- `res` is the evaluation of `body` in `env3` and one of the following applies:
  - ∗ `res` is an environment `env4` and `new_genv` is the global environment given by `env4`—e.g. where the subprogram called is either a setter or a procedure;
  - ∗ `res` is an interruption `Returning(xs,ret_genv)` and `new_genv` is `ret_genv`—this is the general case.

14.3.2 Example: SemanticsRule.FCall.asl

The program:

```
func foo (x : integer) => integer
begin

  return x + 1;

end

func bar (x : integer)
begin

  assert x == 3;

end

func main () => integer
begin

  assert foo(2) == 3;
  bar(3);

  return 0;
end
```

calls the function `foo` and the procedure `bar`.

14.3.3 Code

```
| Some (r, { body = SB_ASL body; args = arg_decls; _ }) ->
    (let () = if false then Format.eprintf "Evaluating %s.@." name in
     let scope = Scope_Local (name, !r) in
     let () = incr r in
     let env1 = IEnv.{ global = genv; local = empty_scoped scope } in
     let one_arg envm (x, _) m = declare_local_identifier_mm envm x m in
     let env2 =
       List.fold_left2 one_arg (return env1) arg_decls actual_args
     in
     let one_narg envm (x, m) =
       let*| env = envm in
       if IEnv.mem x env then return env
       else declare_local_identifier_m env x m
     in
     let*| env3 = List.fold_left one_narg env2 params in
     let**| res = eval_stmt env3 body in
```

```
let () =
  if false then Format.eprintf "Finished evaluating %s.@." name
in
match res with
| Continuing env4 -> return_normal ([], env4.global)
| Returning (xs, ret_genv) ->
    let vs =
      List.mapi (fun i v -> (v, return_identifier i, scope)) xs
    in
    return_normal (vs, ret_genv))
|: SemanticsRule.FCall
```

14.3.4 Formally: sequential case

The evaluation of a n-ary subprogram evaluates the arguments in order then calls the subprogram:

$$[|f(e_1, \dots e_n)|](E_0) \triangleq \left\{ (v, E') \;\middle|\; \begin{array}{c} \forall i \in [|1, n|], (v_i, E_i) \in [|e_i|](E_{i-1}) \\ \text{and } (v, E') \in [| \langle f, v_1, \dots v_n \rangle |] \, (E_n) \end{array} \right\} \tag{14.1}$$

14.3.5 Formally: concurrent case

For i ranging implicitly from 1 to n included, a call to a subprogram f is interpreted as the interpretation of the subprogram call (see to f after evaluating every argument in order:

$$[|f(e_i)|](E_0) \triangleq \left\{ (v, E', S) \;\middle|\; \begin{array}{c} (v_i, E_i, S_i) \in [|e_i|](E_{i-1}) \\ \text{and } (v, E', S) \in [| \langle f, (v_i, S_i) \rangle |] \, (E_n) \end{array} \right\} \tag{14.2}$$

14.3.6 Comments

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

Chapter 15

Evaluation of Programs

15.1 SemanticsRule.TopLevel

15.1.1 Prose

The evaluation of a program `ast` is `res` or an error and all of the following apply:

- `ast'` is `ast` modified to add the standard library;
- `ast_typed, static_env` is the result of typing `ast'`;
- `genv` is the global environment built using `static_env` and evaluating the global constants in `ast_typed` following the Directed Acyclic Graph of their definitions;
- `res` is the result of evaluating the function "`main`, without any argument, in `genv`;
- One of the following applies:
    - ∗ `res` is a value `v`, or
    - ∗ All of the following apply:
        - – `res` is an implicitly thrown exception;
        - – An error "`Uncaught exception:  implicitly thrown out of a try-catch`" is raised;
    - ∗ All of the following apply:
        - – `res` is an exception `exn` with an associated type `ty`;
        - – An error "`Uncaught exception:  {ty} {exn}`" is raised.

15.1.2 Example

15.1.3 Code

```
let run_typed_env env (ast : B.ast) (static_env : StaticEnv.env) : B.value m =
  let*| env = build_genv env eval_expr_sef base_value static_env ast in
  let*| res = eval_func env "main" dummy_annotated [] [] in
  match res with
  | Normal ([ v ], _genv) -> read_value_from v
  | Normal _ -> Error.(fatal_unknown_pos (MismatchedReturnValue "main"))
  | Throwing (v_opt, _genv) ->
      let msg =
        match v_opt with
        | None -> "implicitely thrown out of a try-catch."
        | Some ((v, _, _scope), ty) ->
            Format.asprintf "%a %s" PP.pp_ty ty (B.debug_value v)
      in
      Error.fatal_unknown_pos (Error.UncaughtException msg)

let run_typed ast env = run_typed_env [] ast env

let run_env (env : (AST.identifier * B.value) list) (ast : B.ast) : B.value m
    =
  let ast = Builder.with_stdlib ast in
  let ast, static_env =
    Typing.type_check_ast C.type_checking_strictness ast StaticEnv.empty
  in
  let () =
    if false then Format.eprintf "@[<v 2>Typed AST:@ %a@]@." PP.pp_t ast
  in
  run_typed_env env ast static_env

let run ast = run_env [] ast |: SemanticsRule.TopLevel
```

15.1.4 Formally: sequential case

15.1.5 Formally: concurrent case

15.1.6 Comments

Chapter 16

Side-effects

Side-effects can arise due to:

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