ASL Semantics Reference
DDI 0621

Arm Architecture Technology Group

September 25, 2024

Contents

Chapter 1

Non-Confidential Proprietary Notice

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

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

Chapter 2

Disclaimer

This document is part of the ASLRef material.

This material covers ASLv1, a new, experimental, and as yet unreleased version of ASL.

The development version of ASLRef can be found here:
https://github.com/herd/herdtools7.

A list of open items being worked on can be found here:
https://github.com/herd/herdtools7/blob/master/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

Introduction

The semantics of ASL define all valid behaviors of a given ASL specification. More precisely, an ASL specification is first parsed into an *abstract syntax tree*, or AST, for short. Second, a type checker analyzes the *untyped AST* to determine whether it is well-typed and, if successful, returns a *static environment* and a *typed AST*. Otherwise, it returns a type error.

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

Related documents:

- The Syntax Reference [5] defines the concrete syntax, the abstract syntax, the untyped AST, and the typed AST.
- The ASL Typing Reference [6] defines the set of untyped ASTs that are considered statically valid and how to produce the corresponding typed ASTs and static environments.

Understanding the Semantics Formalization: We assume basic familiarity with the ASL language constructs. The ASL semantics is defined in terms of its AST, and as a consequence familiarity with the AST is required to understand the semantics. The few components of the type system needed to understand the ASL semantics are explained in context. The mathematical background needed to understand the mathematical formalization of the ASL semantics appears in Chapter 4 and Chapter 5.

3.1 When Do ASL Specifications Have Meaning

The ASL semantics defined here assign meaning only to *well-typed specifications*. That is, specifications for which the type-checker produces a static environment rather than a

type error. Specifications that are not well-typed have no defined semantics. In the rest of this document, we assume well-typed specifications.

ASL admits non-determinism, for example via the `UNKNOWN` expression. This means that a given specification might have (potentially infinitely) many derivation trees.

An ASL specification is *terminating* when all of its derivation trees are finite.

Although ASL does not require specifications to terminate, the semantics defined in this document assign meaning only to terminating specifications. A future version of this document, will assign meaning to non-terminating specifications.

3.2 Basic Semantic Concepts

The ASL semantics are given by relations between *semantic configurations*, or *configurations* [7], for short. We refer to relations between semantic configurations as *semantic relations*. Configurations encapsulate information needed to transition into other configurations, such as:

- a *dynamic environment*, which binds variables to values;
- the typed AST node that needs to be evaluated;
- a *concurrent execution graph*, as per a given memory model; and
- values resulting from evaluating expressions.

The semantic relations are constructively defined via *semantic rules*. These semantic rules are defined by induction over the typed AST.

Execution: A valid execution of an ASL specification transitions from an *initial configuration*, which consists of the given specification and the standard library specification, to an output configuration consisting of an output value and a concurrent execution graph.

Primitive Subprograms: The semantics of ASL are parameterized by a set of primitive subprograms — subprograms whose implementation is not defined by ASL statements and whose effect on the dynamic environment is defined externally. Critically, access to memory is given by primitive subprograms.

We define two types of semantics — *sequential semantics* and *concurrent semantics*.

Concurrent Semantics: The concurrent semantics operate over concurrent execution graphs. Intuitively, these graphs define Read Effects and Write Effects to variables and constraints over those effects. Together with the constraints that define a given memory model (such as the ARM memory model [3]), these graphs axiomatically define the valid interactions of shared variables of a given specification.

Sequential Semantics: The sequential semantics correspond to executing an ASL specification in the context of a single thread of execution; notice that ASL does not contain any concurrency constructs. Technically, the sequential semantics are defined by omitting the concurrent execution graph components from all configurations.

Reading guide: The semantic rules are organized into chapters, which roughly group the rules by their AST node type. The set of rules in each chapter is further split according to additional syntactic and semantic predicates over the AST node. For example, expressions are split into literal expressions, binary operation expressions, etc. Each such subset of rules is named accordingly and appears in a section with the same name. For example, the rule for literal expressions is named SemanticsRule.Lit and defined in Section 6.3.

Sometimes rules are split into cases. We then drop the SemanticsRule prefix from each case, for succinctness. The same convention applies to helper rules.

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 paragraphs, which, as much as possible, are also given as regression tests in /herdtools7/asllib/tests/ASLSemanticsReference.t;
- a Formal paragraph, which provides semantic rules that essentially give the same information as the Prose paragraph, for both the sequential semantics and the concurrent semantics, in the form of concurrent execution graphs. The latter enables defining, for each AArch64 instruction, the Intrinsic dependencies used by the AArch64 memory model from the ASL code of the instruction [1, B2.3.2].
- Comments paragraphs, which provide additional details.

Chapter 4

Formal System

In this chapter, we define the mathematical concepts and notations used throughout. This chapter appears in all of the ASL references as its content it used in all of them. We start by defining general mathematical concepts and then describe how sets of rules formally define functions and relations.

4.1 Mathematical Definitions and Notations

We use $\triangleq$ to define mathematical concepts.

We define the following sets:

- $\mathbb{N}$ is the set of natural numbers, including 0.
- $\mathbb{N}^+$ is the set of natural numbers, excluding 0.
- $\mathbb{Z}$ is the set of integers.
- $\mathbb{Q}$ is the set of rationals.
- $\mathbb{B}$ is the set of ASL Boolean literals, which consists of `TRUE` and `FALSE`. We employ these literals to represent the corresponding mathematical truth values, which are used to denote whether logical assertions hold or not. We also employ the mathematical meaning of logical conjunction $\wedge$, logical disjunction $\vee$, and logical negation $\neg$, given next. For a set of Boolean values A:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Definition 7 (Partial Function) *A* partial function*, denoted* $f : A \rightharpoonup B$*, is a function from a subset of* A *to* B*. The* domain *of a partial function* f*, denoted* $\mathrm{dom}(f)$*, is the subset of* A *for which it is defined. We write* $f(x) = \perp$ *to denote that* x *is not in the domain of* f*, that is,* $x \notin \mathrm{dom}(f)$*.*

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

Definition 8 (Finite-domain Function) *The notation* $\to_{fin}$ *stands for a function whose domain is finite.*

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

Definition 10 (Function Update) *The function denoted as* $f[x \mapsto v]$ *is a function identical to* f*, except that* x *is bound to* v*. That is, if* $g = f[x \mapsto v]$ *then*

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

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

Definition 11 (Function Restriction) *The* restriction *of a function* $f : X \to Y$ *to a subset of its domain* $A \subseteq \mathrm{dom}(f)$*, denoted as* $f|_A$*, is defined in terms of the set of input-output pairs:*

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

Definition 12 (Function Graph) *The* graph *of a finite-domain function* $f : X \to_{fin} Y$ *is the list of input-output pairs for* f*, given in any order:*

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

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

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

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

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

Definition 13 (Choice) *The parametric function* $\mathsf{choice} : \overbrace{\mathbb{B}}^{b} \times \overbrace{T}^{x} \times \overbrace{T}^{y} \to \overbrace{T}^{z}$*, is defined as follows:*

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

4.1.1 Lists

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

A list of elements is either empty, denoted by [ ], or non-empty. A non-empty list is either denoted by listing the elements in sequence, $v_1 \ldots v_k$, or in bracketed form,

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

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

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

Definition 14 (List Length) *The* length *of a list is the number of elements in that list:* $|[\,]| \triangleq 0$ *and* $|v_1, \ldots, v_k| = k$.

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

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

Definition 15 (List Concatenation) *The parametric function* $+ : T^* \times T^* \to T^*$ *concatenates two lists:*

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

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

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

compares the length of two lists:

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

Definition 17 (Indices of a List) *The parametric function* ***indices*** $: T^* \to \mathbb{N}^*$ *returns the (1-based) list of indices for a given list:*

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

Definition 18 (Unzipping a List of Pairs) *The parametric function*

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

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

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

4.1.2 OCaml-style Notations

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

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

Definition 19 (Optional) *The notation $\langle\cdot\rangle$ stands for either an empty set or a singleton set, where* `None` $\triangleq \langle\rangle$ *denotes an empty set and $\langle v\rangle$ denotes a set containing the single element v. The notation $\langle T\rangle$, where T denotes a mathematical type, stands for $\{\langle\rangle\} \cup \{\langle v\rangle \mid v \in T\}$.*

We refer to $\langle T\rangle$ as an optional.

4.2 How we use Rules

An *inference rule* (rule, for short) is an implication between a set of logical assertions, called the *premises* of the rule, and a *conclusion* assertion. The conclusion holds when the conjunction of its premises holds.

We use the following rule notation, where $P_{1..k}$ are the rule premises and C is the conclusion:

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

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

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

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

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

The free variables appearing in the premises and conclusion are interpreted universally. That is, the rules apply to any values (of the appropriate types) assigned to their free variables. For example, the rule TypingRule.Binop applies to any choice of values for the free variables `tenv` (a static environment), `e1`, `e2`, `e1'`, `e2'` (expressions), `t`, `t1`, and `t2` (types).

Definition 20 (Grounding) *Assertions can be* grounded *by substituting their free variables with values. A* ground rule *is a rule with all its assertions (premises and conclusion) grounded.*

For example, the following is a grounding of TypingRule.Binop

$$\frac{\begin{array}{c}\texttt{annotate\_expr}(\emptyset_{\text{tenv}}, \text{E\_Literal}(\text{L\_Int}(2))) \xrightarrow{\text{type}} (\text{T\_Int}, \text{E\_Literal}(\text{L\_Int}(2))) \\ \texttt{annotate\_expr}(\emptyset_{\text{tenv}}, \text{E\_Literal}(\text{L\_Int}(3))) \xrightarrow{\text{type}} (\text{T\_Int}, \text{E\_Literal}(\text{L\_Int}(3))) \\ \texttt{check\_binop}(\emptyset_{\text{tenv}}, \texttt{MUL}, \text{T\_Int}, \text{T\_Int}) \xrightarrow{\text{type}} \text{T\_Int}\end{array}}{\begin{array}{c}\texttt{annotate\_expr}(\emptyset_{\text{tenv}}, \text{E\_Binop}(\texttt{MUL}, \text{E\_Literal}(\text{L\_Int}(2)), \text{E\_Literal}(\text{L\_Int}(3)))) \xrightarrow{\text{type}} \\ (\text{T\_Int}, \text{E\_Binop}(\texttt{MUL}, \text{E\_Literal}(\text{L\_Int}(2)), \text{E\_Literal}(\text{L\_Int}(3))))\end{array}}$$

obtained by the following substitutions:

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

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

Definition 21 (Axiom) *An* axiom *is a rule with an empty set of premises. An axiom is denoted by simply stating its conclusion.*

An example of an axiom in the ASL type system is TypingRule.SPass:

$$\texttt{annotate\_stmt}(\text{tenv}, \text{S\_Pass}) \xrightarrow{\text{type}} (\text{S\_Pass}, \text{tenv})$$

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

$$\texttt{eval\_pattern}(\texttt{env}, \_, \text{Pattern\_All}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{Bool}(\texttt{TRUE}), \emptyset_{\text{g}})$$

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

Definition 22 (Derivation Tree) *A* derivation tree *is a tree whose vertices correspond to ground assertions. More specifically, the leaves of a derivation tree correspond to ground axioms, and an internal vertex corresponds to a ground conclusion of a rule with its children corresponding to the ground premises of the same rule.*

4.2.1 Transitions

We use rules as a structured way for defining relations (and therefore functions, as a special case).

To define a relation $R \subseteq X \times Y$, we use assertions of the form $tx \xrightarrow{R} ty$ where tx and ty are logical terms denoting sets of elements from X and Y, respectively. We call such assertions *transitions*. A set of rules M with transition assertions defines the relation

$$R = \{(x, y) \mid x \xrightarrow{R} y \text{ can be derived from rules in } M\} \ .$$

For example, the rule TypingRule.ELit defines a relation between the infinite set of elements of the form `annotate_expr`(tenv, E_Literal(v)) (for the infinite choice of values for the free variables tenv and v) to the infinite set of pairs of the form (t, E_Literal(v)), such that the premise holds.

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

4.2.2 Configurations

Our relations range over compound values. That is, values that often nest tuples and lists inside other tuples and lists. We refer to such values as *configurations*. To make it easier to distinguish between different configurations, we will sometimes attach labels to tuples using the OCaml-style notation discussed earlier. We refer to those labels as *configuration domains*. The domain of a configuration $C = L(\ldots)$, denoted `config_domain`(C), is the label L.

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

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

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

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

Definition 23 (Fresh Element) *Premises of the form $x \in T$ `is fresh` mean that in any instantiation in a derivation tree, the value of x is unique. That is, different from all other values instantiated for any other variable.*

Definition 24 (Ignore Variable) *To keep rules succinct, we write _ for a mathematical variable whose name is irrelevant for understanding the rule, and can thus be omitted. Each occurrence of _ represents a variable whose name is different from any other free variable in the rule.*

For example, the rule SemanticsRule.PAll, shown above, uses an ignore variable to stand for the value being matched by a `-` pattern. Since the rule does not need to refer to the value, we do not name it and use an ignore variable instead.

4.2.3 Flavors of Equality In Rules

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

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

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

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

is the list of premises

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

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

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

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

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

4.2.4 AST-related Notations

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

For example[1], the func non-terminal is of a record type and has the following fields: name, parameters, args, body, return_type, and subprogram_type. The notation {body : SB_ASL(body), args : arg_decls, ...} allows us to deconstruct a given func node by matching only the body and args fields.

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

4.2.5 How to Parse Rules Efficiently

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

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

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

Now, scan the premises in order to see where env, op, e1, and e2 are used and how premises assign values to v, g, and new_env. In this case, v is assigned as the result of the transition assertion *binop*(op, v1, v2) $\xrightarrow{\text{eval}}$ v, g is assigned the expression g1 $\parallel$ g2, and new_env is assigned as the result of the transition assertion *eval_expr*(env1, e2) $\xrightarrow{\text{eval}}$ Normal(m2, new_env). Notice that to assign values to the variables v, g, and new_env, intermediate values have to be assigned first. For example, *eval_expr*(env, e1) $\xrightarrow{\text{eval}}$ Normal(m1, env1) assigned values to env1, which is then used by the transition *eval_expr*(env1, e2) $\xrightarrow{\text{eval}}$ Normal(m2, new_env). Similarly, g is requires first assigning values to g1 and g2, which are components of the previously assigned variables m1 and m2.

[1] This example is from SemanticsRule.FCall.

4.2.6 Short-Circuit Rule Macros

Short-circuit rule macros, or *rule macros*, for short, allow us to succinctly define sets of rules. Specifically, they allow us to capture situations where transitions have two alternative output configurations. If the transition results in the first of the alternative output configurations, the following premises are considered. However, if the result is the second, short-circuit output configuration, then the following premises are ignored and the conclusion transitions into the short-circuit output configuration. These short-circuit output configurations are typically, but not always, due to (type or dynamic) errors.

In the following, XP and XQ stand for, possibly empty, sequences of premises. A rule macro includes the special premise form $C \xrightarrow{R} C' /\!\!/ E$, which introduces alternative output configurations C' and short-circuit E:

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

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

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

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

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

We allow more than one premise to include short-circuiting alternatives and also a single premise to include several alternatives. That is, a rule macro of the form

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

Stands for the set of rule macros

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

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

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

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

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

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

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

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

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

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

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

tively:

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

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

Expanding the last transition assertion, gives us the case:

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

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

4.2.7 Boolean Transition Assertions

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

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

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

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

4.2.8 Rule Naming

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

For example, the ASL Semantics Reference defines the rule SemanticsRule.BaseValue using 11 cases of which two are the following:

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

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

The full name of the first case is then SemanticsRule.BaseValue.BOOL and the full name of the second case is SemanticsRule.BaseValue.REAL.

When explaining rules in English prose, we include the name of the case rules in parenthesis to make it easier to relate the prose to the corresponding mathematical definitions (see, for example, the Prose paragraph of SemanticsRule.BaseValue or that of TypingRule.CheckUnop).

4.2.9 Generic Notations

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

Chapter 5

Semantics Building Blocks

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

5.1 Native Values

The semantics of an ASL specification associates *native values* to variables. The set of native values $\mathbb{V}$ is the minimal set defined by the following recursive rules (`NV` stands for Native Value):

5.1.1 Prose

The set of native values $\mathbb{V}$ is the minimal set satisfying all of the following rules:

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

5.1.2 Formally

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

$$\frac{\texttt{v} \in \text{literal}}{\texttt{NV\_Literal}(\texttt{v}) \in \mathbb{V}}$$

(TUPLE VALUES AND ARRAY VALUES)

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

(RECORD VALUES)

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

We define the following shorthands for native value literals:

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

We define the following types of native values:

$$\begin{aligned}
\mathcal{Z} &\triangleq \{\texttt{Int}(z) \mid z \in \mathbb{Z}\} \\
\mathcal{B} &\triangleq \{\texttt{Bool}(\texttt{TRUE}), \texttt{Bool}(\texttt{FALSE})\} \\
\mathcal{R} &\triangleq \{\texttt{Real}(r) \mid r \in \mathbb{Q}\} \\
\mathcal{STR} &\triangleq \{\texttt{String}(s) \mid \texttt{"}s\texttt{"} \in \texttt{<string>}\} \\
\mathcal{BV} &\triangleq \{\texttt{Bitvector}(\mathit{bits}) \mid \mathit{bits} \in \{0,1\}^*\} \\
\mathcal{VEC} &\triangleq \{\texttt{NV_Vector}(\mathit{vals}) \mid \mathit{vals} \in \mathbb{V}^*\} \\
\mathcal{REC} &\triangleq \{\texttt{NV_Record}(\mathit{field_map}) \mid \mathit{field_map} \in \mathbb{I} \to \mathbb{V}\}
\end{aligned}$$

5.2 Semantic Configurations

In the remainder of this document, we use the term configurations to refer to semantic configurations.

Configurations express intermediate states related by *semantic relations*. More precisely, semantic relations relate two distinct sets of configurations — *input configurations* and *output configurations*. Input configurations consist of an environment and an AST node. Output configurations consist of an output environment, values, and concurrent execution graphs. Configurations wrap together elements such as environments and AST nodes and associate them with a *configuration domain*. Input configuration domains determine the semantic relation they pertain to, while output configuration domains distinguish between conceptually different kinds of outputs, for example ones where an exception was raised, ones when a dynamic error occurred, etc.

We now explain the components over which configurations are defined:

- Static Environments (Definition 26) consist of the information inferred by the typechecker for the specification.
- Dynamic Environments (Definition 25) associate native values to variables.
- Concurrent Execution Graphs (Section 5.2.1) track Read and Write Effects over variables.

Definition 25 (Dynamic Environments) *A* sequential dynamic environment*, or* dynamic environment*, for short, is a structure which, associates native values to variables. Formally, a sequential environment* $\mathsf{denv} \in \mathbb{DE}$ *is a pair consisting of a partial function (see Definition 7) from global variable names to their native value, and a partial function from local variable names to their native values:*

$$\begin{array}{rcl} \mathbb{DE} & \triangleq & \mathbb{G} \times \mathbb{L} \\ \mathbb{G} & \triangleq & (\mathbb{I} \rightharpoonup \mathbb{V}) \\ \mathbb{L} & \triangleq & (\mathbb{I} \rightharpoonup \mathbb{V}) \end{array}$$

Definition 26 (Static Environment) *A* static environment *[6]* $\mathsf{tenv} \in \mathbb{SE}$ *(also referred to as a* type environment*) is produced by the type-checker from the untyped AST.*

We assume that the static environment supports the following functions:

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

The partial function find_func returns the typed AST of the subprogram for a given identifier. (Recall that ASL allows subprogram overloading so a name does not uniquely identify a specific subprogram. However, the type-checker renames each function uniquely so that it can be accessed based on its name alone.) The function type_satisfies$(\mathtt{t}, \mathtt{s})$ *returns true if the type* `t` type-satisfies *the type* `s` *(see TypingRule.TypeSatisfaction [6]). This is used in matching a raised exception to a corresponding catch clause.*

Definition 27 (Environments) *Environments pair static environments with dynamic environments:* $\mathbb{E} = \mathbb{SE} \times \mathbb{DE}$.

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

5.2.1 Concurrent Execution Graphs

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

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

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

Nodes represent unique Read and Write Effects. Formally, a node $(u, l, \mathtt{id}) \in \mathcal{N}$ associates a unique instance counter u to an *ordering label* l, which specifies whether

it represents a Read Effect of a Write Effect to a variable named `id`. We say that en Effect `E1` is l-*before* another Effect `E2`, for $l \in \mathcal{L}$ and a given execution graph g, when $(\texttt{E1}, l, \texttt{E2}) \in E^g$.

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

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

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

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

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

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

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

Definition 28 (Read/Write Effects)

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

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

Definition 29 (Unordered Graph Composition) *Given two execution graphs $S_1 = (N_1, E_1, O_1)$ and $S_1 = (N_2, E_2, O_2)$ their unordered composition, denoted $S_1 \parallel S_2$ is defined as follows:*

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

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

Definition 30 (Ordered Graph Composition) *Given two execution graphs, $S_1 = (N_1, E_1, O_1)$ and $S_1 = (N_2, E_2, O_2)$ and an ordering label l, the ordered composition $S_1 \xrightarrow{l} S_2$ is defined as follows:*

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

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

5.2.2 Kinds of Semantic Configurations

Recall that the ASL semantics defines a relation between input configurations and output configurations (Section 5.2). Input configuration domains are unique to the semantic relation that employs them. For that reason, we name semantic relations by the name of the corresponding input configuration domain. For example, the semantic relation that employs input configurations with the domain `eval_expr` is named `eval_expr`. We will often use the prefix `eval` for semantic relations with the intuition being that their input configurations should be semantically evaluated, yielding an output configuration.

ASL semantics mainly utilizes the following types of output configurations:

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

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

Exceptions. Configurations in

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

represent thrown exceptions.

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

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

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

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

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

We define the following shorthands for types of output configurations:

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

We will say that a semantic transition *terminates*:

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

We introduce the following shorthands for configurations where all variables appearing are fresh:

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

5.2.3 Extracting and Substituting Elements of Configurations

Given a configuration C, we define the graph component of the configuration, *graph*(C), and the environment of the configuration, *environ*(C), as follows:

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

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

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

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

Configuration	$C(\mathit{environ} \mapsto \texttt{env'})$
Normal(v, g)	undefined
Normal((v, g), env)	Normal((v, g), env')
Normal($(i = 1..k : (\texttt{va}_i, \texttt{vb}), \texttt{g}), \texttt{env}$)	Normal($(i = 1..k : (\texttt{va}_i, \texttt{vb}), \texttt{g}), \texttt{env'}$)
Normal(g, env)	Normal(g, env')
Normal($i = 1..k : \texttt{v}_i, \texttt{g}$)	Normal($i = 1..k : \texttt{v}_i, \texttt{g}$)
Normal($[i = 1..k : (\texttt{v}_i, \texttt{g}_i)], \texttt{env}$)	Normal($[i = 1..k : (\texttt{v}_i, \texttt{g}_i)], \texttt{env'}$)
Throwing((value_read_from(x, v), g), env)	Throwing((value_read_from(x, v), g), env')
Returning($(i = 1..k : \texttt{v}_i, \texttt{g}), \texttt{env}$)	Returning($(i = 1..k : \texttt{v}_i, \texttt{g}), \texttt{env'}$)
Continuing(g, env)	Continuing(g, env')

5.3 Semantic Evaluation

The semantics of ASL is given by the relations[1] eval and primitive. The relation eval is defined as the disjoint union of the relations defined in this document. The relation primitive provides the semantics of primitive subprograms and is not otherwise defined constructively.

5.3.1 Natural Operational Semantics

We define the ASL semantics in the style of *natural operational semantics* [7] (also known as *big step semantics*). Natural operational semantics evaluates the AST inductively. That is, it concludes transitions for configurations starting from non-leaf AST nodes by concluding transitions from configurations starting from their children nodes.

[1] The reason that relations, rather than functions, are used is due to the potential non-determinism in the primitive subprograms and the non-determinism inherent in the `UNKNOWN` expression.

No Undefined Behaviors

When an input configuration does not satisfy any semantic rule, there is no output configuration for it to transition to. We say that the configuration is *stuck* and the ASL semantics is undefined for that input configuration.

The ASL semantics is defined for well-typed ASL specifications and gets stuck only in cases of non-terminating specifications (due to non-terminating loops, or infinite recursion). Otherwise, for every input configuration there is at least one rule that can be used to take a semantic transition.

Evaluation Example

The following example shows how to utilize the rules for expression literals and binary operator expressions to derive a transition from an input configuration with the expression $(1 + 2) * (4 + 5)$, given by the AST

$$\mathsf{E\_Binop}(\texttt{MUL}, \overbrace{\mathsf{E\_Binop}(\texttt{PLUS}, \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(1))}^{\texttt{e1}}, \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(2))}^{\texttt{e2}})}^{\texttt{e12}}, \overbrace{\mathsf{E\_Binop}(\texttt{PLUS}, \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(4))}^{\texttt{e4}}, \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(5))}^{\texttt{e5}})}^{\texttt{e45}})$$

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

We annotate subexpressions to allow referring to them.

We write $\emptyset_{\mathsf{env}}$ to stand for a trivial environment (that is, one where all functions are empty).

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

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

$$\frac{\begin{array}{l} \mathit{eval\_expr}(\emptyset_{\mathsf{env}}, \mathtt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(1), \emptyset_{\mathsf{env}}) \ 6.3 \\ \mathit{eval\_expr}(\emptyset_{\mathsf{env}}, \mathtt{e2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(2), \emptyset_{\mathsf{env}}) \ 6.3 \\ \mathit{binop}(\mathtt{PLUS}, \mathtt{Int}(1), \mathtt{Int}(2)) \xrightarrow{\mathsf{eval}} \mathtt{Int}(3) \ 17.19 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathsf{env}}, \mathtt{e12}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(3), \emptyset_{\mathsf{env}})} \ 6.9$$

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\emptyset_{\mathtt{env}}, \mathsf{e4}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(4), \emptyset_{\mathtt{env}}) \; 6.3 \\ \mathit{eval\_expr}(\emptyset_{\mathtt{env}}, \mathsf{e5}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(5), \emptyset_{\mathtt{env}}) \; 6.3 \\ \mathit{binop}(\mathtt{PLUS}, \mathtt{Int}(4), \mathtt{Int}(5)) \xrightarrow{\mathsf{eval}} \mathtt{Int}(9) \; 17.19 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathtt{env}}, \mathsf{e45}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(9), \emptyset_{\mathtt{env}})} \; 6.9$$

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\emptyset_{\mathtt{env}}, \mathsf{e12}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(3), \emptyset_{\mathtt{env}}) \\ \mathit{eval\_expr}(\emptyset_{\mathtt{env}}, \mathsf{e45}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(9), \emptyset_{\mathtt{env}}) \\ \mathit{binop}(\mathtt{MUL}, \mathtt{Int}(3), \mathtt{Int}(9)) \xrightarrow{\mathsf{eval}} \mathtt{Int}(27) \; 17.19 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathtt{env}}, \mathsf{E\_Binop}(\mathtt{MUL}, \mathsf{e12}, \mathsf{e45})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(27), \emptyset_{\mathtt{env}})} \; 6.9$$

Chapter 6

Evaluation of Expressions

6.1 Informal Preamble

6.1.1 Execution-time expressions

Expressions in ASL are either execution-time expressions or non-execution-time expressions.

An expression is an execution-time expression if either:

- it contains an execution-time storage element identifier
- it contains an execution-time function or getter invocation

6.1.2 Compile-time-constant expressions

Expressions in ASL are either compile-time-constant expressions or non-compile-time-constant expressions.

An expression is a compile-time-constant expression if each one of its atomic expressions is one of:

- a literal constant
- a compile-time-constant storage element identifier
- an immutable storage element identifier with a compile-time-constant initializer expression.
- compile-time-constant function or getter invocations

6.2 Formal Preamble

The relation

$$\mathit{eval\_expr}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{expr}}^{\texttt{e}}) \bigtimes \mathsf{Normal}((\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

evaluates the expression `e` in an environment `env` and one of the following applies:

- the evaluation terminates normally, returning a native value `v`, a concurrent execution graph `g`, and a modified environment `new_env`;
- the evaluation terminates abnormally.

The evaluation of an expression is specialized by the shape of the expression `e`, and one of the following applies:

- SemanticsRule.Lit (see Section 6.3);
- SemanticsRule.ELocalVar (see Section 6.4)
- SemanticsRule.EGlobalVar (see Section 6.5)
- SemanticsRule.BinopAnd (see Section 6.6)
- SemanticsRule.BinopOr (see Section 6.7)
- SemanticsRule.BinopImpl (see Section 6.8)
- SemanticsRule.Binop (see Section 6.9)
- SemanticsRule.Unop (see Section 6.10)
- SemanticsRule.ECond (see Section 6.11)
- SemanticsRule.ESlice (see Section 6.12)
- SemanticsRule.ECall (see Section 6.13)
- SemanticsRule.EGetArray (see Section 6.14)
- SemanticsRule.ERecord (see Section 6.15)
- SemanticsRule.EGetField (see Section 6.16)
- SemanticsRule.EConcat (see Section 6.17)
- SemanticsRule.ETuple (see Section 6.18)
- SemanticsRule.EUnknown (see Section 6.19)
- SemanticsRule.EPattern (see Section 6.20)
- SemanticsRule.ATC (see Section 6.21)

We also define the following helper relations:

- SemanticsRule.EExprList (see Section 6.22);
- SemanticsRule.EExprListM (see Section 6.23);
- SemanticsRule.ESideEffectFreeExpr (see Section 6.24);

6.3 SemanticsRule.Lit

6.3.1 Prose

All of the following apply:

- `e` is the literal expression for `l`, that is, E_Literal(`l`)
- `v` is the native value corresponding to `l`;
- `g` is the empty graph, as literals do not yield any Read and Write Effects;
- `new_env` is `env`.

6.3.2 Example

In the specification:

```
func main () => integer
begin

  assert 3 == 3;
  return 0;

end
```

each of the expressions `3` evaluates to the native value `Int`(3).

6.3.3 Formally

$$\mathit{eval\_expr}(\texttt{env}, \overbrace{\mathsf{E\_Literal}(\texttt{l})}^{\texttt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\overbrace{\texttt{NV\_Literal}(\texttt{l})}^{\texttt{v}}, \overbrace{\emptyset_{\mathsf{g}}}^{\texttt{g}}), \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

6.4 SemanticsRule.ELocalVar

6.4.1 Prose

All of the following apply:

- `e` denotes a variable expression, E_Var(`x`), which is bound locally in `env`;
- `v` is the value of `x` in `env`;
- `new_env` is `env`.
- `g` is the graph containing a single Read Effect for `x`.

6.4.2 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

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

6.4.3 Formally

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

6.4.4 Comments

When there exists a global variable `x`, the type system forbids having `x` as a local variable. This is enforced by TypingRule.LDVar in the Chapter "Typing of Local Declarations", and TypingRule.DeclareGlobalStorage and TypingRule.DeclareOneFunc, both in the Chapter "Typing of Global Declarations".

6.5 SemanticsRule.EGlobalVar

6.5.1 Prose

All of the following apply:

- `e` denotes a variable expression, E_Var(`x`), which is bound globally in `env`;
- `v` is the value of `x` in `env`;
- `new_env` is `env`.
- `g` is the graph containing a single Read Effect for `x`.

6.5.2 Example

In the specification:

```
var global_x: integer = 3;

func main () => integer
  begin

    assert global_x == 3;
    return 0;

  end
```

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

6.5.3 Formally

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

6.6 SemanticsRule.BinopAnd

6.6.1 Prose

All of the following apply:

- `e` denotes a conjunction over two expressions, $\mathsf{E\_Binop}(\mathtt{BAND}, \mathtt{e1}, \mathtt{e2})$;
- C is the result of the evaluation of the expression `if e1 then e2 else false` (see Section 6.11).

6.6.2 Example

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

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

the expression `FALSE && fail()` evaluates to the value `FALSE`. Notice that the function `fail` is never called.

6.6.3 Formally

$$\frac{\texttt{false'} := \mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{FALSE})) \qquad \mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Cond}(\texttt{e1}, \texttt{e2}, \texttt{false'})) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{BAND}, \texttt{e1}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}$$

6.6.4 Comments

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to `TRUE` is `e2` evaluated.

It is an error for an expression's meaning to rely on evaluation order except that conditional expressions, and uses of the boolean operators `&&`, `||`,`-->`, are guaranteed to evaluate from left to right.

An implementation could enforce this rule by performing a global analysis of all functions to determine whether a function can throw an exception and the set of global variables read and written by a function.

Conditional expressions and the operations `&&`, `||`, `-->` provide a short-circuit evaluation mechanism:

- the first operand of `if` is always evaluated but only one of the remaining operands is evaluated;
- if the first operand of `and_bool` is `FALSE`, then the second operand is not evaluated;
- if the first operand of `or_bool` is `TRUE`, then the second operand is not evaluated; and,
- if the first operand of `implies_bool` is `FALSE`, then the second operand is not evaluated.

However, note that relying on this short-circuit evaluation can be confusing for readers of ASL specifications and as a consequence it is recommended that an if-statement is used to achieve the same effect.

6.7 SemanticsRule.BinopOr

6.7.1 Prose

All of the following apply:

- `e` denotes a disjunction of two expressions, $\mathsf{E\_Binop}(\texttt{BOR}, \texttt{e1}, \texttt{e2})$;
- C is the result of the evaluation of `if e1 then true else e2` (see Section 6.11).

6.7.2 Example

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

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

$$\frac{\texttt{true'} := \mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{TRUE})) \qquad \mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Cond}(\texttt{e1}, \texttt{true'}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{BOR}, \texttt{e1}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}$$

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to `FALSE`, is `e2` evaluated.

6.7.3 Comments

It is an error for an expression's meaning to rely on evaluation order except that conditional expressions, and uses of the boolean operators `&&`, `||`, `-->`, are guaranteed to evaluate from left to right.

An implementation could enforce this rule by performing a global analysis of all functions to determine whether a function can throw an exception and the set of global variables read and written by a function.

Conditional expressions and the operations `&&`, `||`, `-->` provide a short-circuit evaluation mechanism:

- the first operand of `if` is always evaluated but only one of the remaining operands is evaluated;
- if the first operand of `and_bool` is `FALSE`, then the second operand is not evaluated;
- if the first operand of `or_bool` is `TRUE`, then the second operand is not evaluated; and,
- if the first operand of `implies_bool` is `FALSE`, then the second operand is not evaluated.

However, note that relying on this short-circuit evaluation can be confusing for readers of ASL specifications and as a consequence it is recommended that an if-statement is used to achieve the same effect.

6.8 SemanticsRule.BinopImpl

6.8.1 Prose

All of the following apply:

- `e` denotes an implication over two expressions, E_Binop(`IMPL`, `e1`, `e2`);
- `e` is evaluated as `if e1 then e2 else true`.

6.8.2 Example

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

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

$$\frac{\texttt{true'} := \mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{TRUE})) \qquad \mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Cond}(\texttt{e1}, \texttt{e2}, \texttt{true'})) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{IMPL}, \texttt{e1}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}$$

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to `TRUE`, is `e2` evaluated.

Conditional expressions and the operations `&&`, `||`, `-->` provide a short-circuit evaluation mechanism:

- the first operand of `if` is always evaluated but only one of the remaining operands is evaluated;
- if the first operand of `and_bool` is `FALSE`, then the second operand is not evaluated;
- if the first operand of `or_bool` is `TRUE`, then the second operand is not evaluated; and,
- if the first operand of `implies_bool` is `FALSE`, then the second operand is not evaluated.

However, note that relying on this short-circuit evaluation can be confusing for readers of ASL specifications and as a consequence it is recommended that an if-statement is used to achieve the same effect.

6.9 SemanticsRule.Binop

6.9.1 Prose

All of the following apply:

- `e` denotes a Binary Operator `op` over two expressions, E_Binop(`op`, `e1`, `e2`);
- the operator `op` is not one of `BAND`, `BOR`, or `IMPL`. These operators are handled by rules SemanticsRule.BinopAnd (Section 6.6), SemanticsRule.BinopOr (Section 6.7), and SemanticsRule.BinopImpl (Section 6.8);
- the evaluation of the expression `e1` in `env` is the configuration Normal(`m1`, `env1`)$^{\#T,\#DE}$;
- the evaluation of the expression `e2` in `env1` is the configuration Normal(`m2`, `new_env`)$^{\#T,\#DE}$;
- `m1` consists of the value `v1` and the execution graph `g1`;
- `m2` consists of the value `v2` and the execution graph `g2`;
- applying the Binary Operator `op` to `v1` and `v2` results in `v`$^{\#DE}$;
- `g` is the parallel composition of `g1` and `g2`.

6.9.2 Example

In this specification:

```
func main () => integer
begin

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

  return 0;
end
```

the expression `3 + 2` evaluates to the value `5`.

6.9.3 Example

In the specification:

```
func main () => integer
begin

  let x = 3 DIV 0;

  return 0;
end
```

the expression `3 DIV 0` results in a type error.

6.9.4 Formally

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

The rule above applies to many binary operators, including `EQ_OP` (which is used for `<->` as well as `==`).

6.9.5 Comments

The semantics takes a semantic transition over the left subexpression before the right subexpression.

This is an arbitrary choice as the type-checker must ensure that either order of evaluation of the operands yields the same result.

In other words, it it is an error for an expression's meaning to rely on evaluation order except that conditional expressions, and uses of the boolean operators `&&`, `||`, `-->`, are guaranteed to evaluate from left to right.

An implementation could enforce this rule by performing a global analysis of all functions to determine whether a function can throw an exception and the set of global variables read and written by a function.

Notice that when one of the subexpressions terminates exceptionally, the other expression must be side effect-free and non-throwing.

In other words, for any function call `F (e1, ..., em)`, tuple `(e1, ..., em)`, or operation `e1 op e2` (with the exception of `&&`, `||` and `-->`), it is an error if the subexpressions conflict with each other by:

- both writing to the same variable.
- one writing to a variable and the other reading from that same variable
- one writing to a variable and the other throwing an exception
- both throwing exceptions

These conditions are sufficient but not necessary to ensure that evaluation order does not affect the result of an expression, including any side-effects.

6.10 SemanticsRule.Unop

6.10.1 Prose

All of the following apply:

- `e` denotes a unary operator `op` over an expression, E_Unop(`op`, `e1`);
- the evaluation of the expression `e1` in `env` yields Normal((`v1`, `g`), `new_env`)$^{\#T,\#DE}$;
- applying the unary operator `op` to `v1` is `v`.

6.10.2 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

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

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v1}, \texttt{g}), \texttt{new\_env}) \;/\!/\; \#T, \#DE \qquad \mathit{unop}(\texttt{op}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{v}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Unop}(\texttt{op}, \texttt{e1})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

6.11 SemanticsRule.ECond

6.11.1 Prose

All of the following apply:

- `e` denotes a conditional expression `e_cond` with two options `e1` and `e2`, that is, E_Cond(`e_cond`, `e1`, `e2`);
- the evaluation of the conditional expression `e_cond` in `env` yields Normal(`m_cond`, `env1`)$^{\#T,\#DE}$;
- `m_cond` consists of a native Boolean for `b` and execution graph `g1`;
- `e'` is `e1` if `b` is `TRUE` and `e2` otherwise;
- the evaluation of `e'` in `env1` yields Normal((`v2`, `g2`), `new_env`)$^{\#T,\#DE}$;
- `g` is the parallel composition of `g1` and `g2`.

6.11.2 Example

In the specification:

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

func main () => integer
begin

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

  return 0;
end
```

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

6.11.3 Example

In the specification:

```
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.

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

6.11.4 Comments

A conditional expression evaluates to its `then` expression if the condition expression evaluates to `TRUE`. If the condition expression evaluates to `FALSE` each `elsif` condition expression is evaluated sequentially until an `elsif` condition expression evaluates to `TRUE`; the conditional expression evaluates to the corresponding `elsif` expression. If no `elsif` expression evaluates to `TRUE` the conditional expression evaluates to the `else` expression.

6.12 SemanticsRule.ESlice

6.12.1 Prose

All of the following apply:

- `e` denotes a slicing expression, E_Slice(`e_bv`, `slices`);
- the evaluation of `e_bv` in `env` yields Normal(`m_bv`, `env1`)$^{//\#T,\#DE}$;
- the evaluation of `slices` in `env` yields Normal(`m_positions`, `new_env`)$^{//\#T,\#DE}$;
- `m_positions` consists of `positions` — all the indices that need to be added to the resulting bitvector — and the execution graph `g1`;
- reading from `v_bv` as a bitvector at the indices indicated by `positions` (see Section 17.6) results in the bitvector `v`, which concatenates all of the values from the indicates indices$^{//\#DE}$;
- `g` is the parallel composition of `g1` and `g2`.

6.12.2 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

the expression `'11110000'[6:3]` evaluates to the value `'1110'`.

6.12.3 Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\mathtt{env}, \mathtt{e\_bv}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_bv}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_bv} \overset{\mathsf{is}}{=} (\mathtt{v\_bv}, \mathtt{g1}) \\ \mathit{eval\_slices}(\mathtt{env1}, \mathtt{slices}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_positions}, \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_positions} \overset{\mathsf{is}}{=} (\mathtt{positions}, \mathtt{g2}) \\ \mathit{read\_from\_bitvector}(\mathtt{v\_bv}, \mathtt{positions}) \xrightarrow{\mathsf{eval}} \mathtt{v} \;/\!/\; \mathtt{\#DE} \\ \mathtt{g} := \mathtt{g1} \parallel \mathtt{g2} \end{array}}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_Slice}(\mathtt{e\_bv}, \mathtt{slices})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

6.13 SemanticsRule.ECall

All of the following apply:

- `e` denotes a subprogram call, E_Call(`name`, `actual_args`, `params`);
- the evaluation of that subprogram call in `env` is either Normal(`vms`, `new_env`)$^{/\!/\mathtt{\#T},\mathtt{\#DE}}$;
- one of the following applies:
  - all of the following apply (SINGLE_RETURNED_VALUE):
    - `vms` consists of a single returned value (`v`, `g`), which goes into the output configuration Normal((`v`, `g`), `new_env`).
  - all of the following apply (MULTIPLE_RETURNED_VALUES):
    - `vms` consists of a list of returned value ($\mathtt{v}_i$, $\mathtt{g}_i$), for $i = 1..k$;
    - `g` is the parallel composition of $\mathtt{g}_i$, for $i = 1..k$;
    - `v` is the native value vector of values $\mathtt{v}_i$, for $i = 1..k$;
    - the resulting configuration is Normal((`v`, `g`), `new_env`).

6.13.1 Example

In the specification:

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

func main () => integer
begin

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

  return 0;
end
```

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

6.13.2 Formally

SINGLE_RETURNED_VALUE

$$\frac{\mathit{eval\_call}(\texttt{env}, \texttt{name}, \texttt{actual\_args}, \texttt{params}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{vms} \overset{\text{is}}{=} [(\texttt{v}, \texttt{g})]}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Call}(\texttt{name}, \texttt{actual\_args}, \texttt{params})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

MULTIPLE_RETURNED_VALUES

$$\frac{\mathit{eval\_call}(\texttt{env}, \texttt{name}, \texttt{actual\_args}, \texttt{params}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{vms} \overset{\text{is}}{=} [i = 1..k : (\texttt{v}_i, \texttt{g}_i)] \quad \texttt{g} := \texttt{g}_1 \parallel \ldots \parallel \texttt{g}_k \quad \texttt{v} := \texttt{NV\_Vector}(\texttt{v}_{1..k})}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Call}(\texttt{name}, \texttt{actual\_args}, \texttt{params})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

6.14 SemanticsRule.EGetArray

6.14.1 Prose

All of the following apply:

- `e` denotes an array access expression, E_GetArray(`e_array`, `e_index`);
- the evaluation of `e_array` in env is Normal(`m_array`, `env1`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- the evaluation of `e_index` in env is Normal(`m_index`, `new_env`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$
- `m_array` consists of the native vector `v_array` and execution graph `g1`;
- `m_index` consists of the native integer `index` and execution graph `g2`;
- `index` is the native integer for `i`;
- evaluating the value at index `i` of `v_array` is `v`;
- `g` is the parallel composition of `g1` and `g2`.

6.14.2 Example

In the specification:

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

var my_array : MyArrayType;

func main () => integer
begin
```

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

  return 0;
end
```

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

6.14.3 Example

The specification:

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

var my_array : MyArrayType;

func main () => integer
begin
  print(my_array[3]);
  return 0;
end
```

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

6.14.4 Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\mathtt{env}, \mathtt{e\_array}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_array}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathit{eval\_expr}(\mathtt{env1}, \mathtt{e\_index}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_index}, \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_array} \overset{\mathsf{is}}{=} (\mathtt{v\_array}, \mathtt{g1}) \qquad \mathtt{m\_index} \overset{\mathsf{is}}{=} (\mathtt{index}, \mathtt{g2}) \\ \mathtt{index} \overset{\mathsf{is}}{=} \mathtt{Int}(\mathtt{i}) \qquad \mathit{get\_index}(\mathtt{i}, \mathtt{v\_array}) \xrightarrow{\mathsf{eval}} \mathtt{v} \qquad \mathtt{g} := \mathtt{g1} \parallel \mathtt{g2} \end{array}}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_GetArray}(\mathtt{e\_array}, \mathtt{e\_index})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

6.15 SemanticsRule.ERecord

6.15.1 Prose

All of the following apply:

- `e` denotes a record creation expression, E_Record(`names`, `e_fields`);
- the names of the fields are $\mathtt{id}_{1..k}$;
- the expressions associated with the fields are $\mathtt{e}_{1..k}$;
- evaluating the expressions of `fields` in order yields $\mathsf{Normal}((\mathtt{v\_fields}, \mathtt{g}), \mathtt{new\_env})^{/\!/\mathtt{\#T},\mathtt{\#DE}}$;

- `v_fields` is a list of native values $\mathtt{v}_{1..k}$;
- `v` is the native record that maps $\mathtt{id}_i$ to $\mathtt{v}_i$, for $i = 1..k$.

6.15.2 Example

In the specification:

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

func main () => integer
begin

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

  return 0;
end
```

the expression `MyRecordType{a=3, b=42}` evaluates to the native record value $\mathsf{NV\_Record}(\mathtt{a} \mapsto \mathsf{Int}(3), \mathtt{b} \mapsto \mathsf{Int}(42))$.

6.15.3 Formally

$$\frac{\mathtt{e\_fields} \overset{\text{is}}{=} [i = 1..k : (\mathtt{id}_i, \mathtt{e}_i)] \qquad \mathtt{names} := \mathtt{id}_{1..k} \qquad \mathtt{fields} := \mathtt{e}_{1..k} \qquad \mathit{eval\_expr\_list}(\mathtt{env}, \mathtt{fields}) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v\_fields}, \mathtt{g}), \mathtt{new\_env}) \;/\!/\; \mathsf{\#T}, \mathsf{\#DE} \qquad \mathtt{v\_fields} \overset{\text{is}}{=} \mathtt{v}_{1..k} \qquad \mathtt{v} := \mathsf{NV\_Record}(\{i = 1..k : \mathtt{id}_i \mapsto \mathtt{v}_i\})}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_Record}(\_, \mathtt{e\_fields})) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

6.16 SemanticsRule.EGetField

6.16.1 Prose

All of the following apply:

- `e` denotes a field access expression, $\mathsf{E\_GetField}(\mathsf{E\_Record}, \mathtt{field\_name})$;
- the evaluation of $\mathsf{E\_Record}$ in `env` is $\mathsf{Normal}((\mathtt{v\_record}, \mathtt{g}), \mathtt{new\_env})^{/\!/\mathsf{\#T},\mathsf{\#DE}}$;
- `v` is the value mapped by `field_name` in the native record `v_record`.

6.16.2 Example

In the specification:

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

func main () => integer
begin

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

  return 0;
end
```

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

6.16.3 Formally

$$\frac{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Record}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v\_record}, \texttt{g}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{get\_field}(\texttt{field\_name}, \texttt{v\_record}) \xrightarrow{\mathsf{eval}} \texttt{v}}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_GetField}(\mathsf{E\_Record}, \texttt{field\_name})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

6.17 SemanticsRule.EConcat

6.17.1 Prose

All of the following apply:

- `e` denotes a concatenation of bitvector expressions, E_Concat(`e_list`);
- the evaluation of `e_list` in `env` is $\mathsf{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `v` is the bitvector constructed from the concatenation of `v_list`.

6.17.2 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

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

6.17.3 Example

In the specification:

```
var T: boolean = [ '1111', '0000' ] == '11110000';

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

the expression `['1111', '0000']` evaluates to the value `'11110000'`.

6.17.4 Formally

$$\frac{\mathit{eval\_expr\_list}(\texttt{env}, \texttt{e\_list}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{concat\_bitvectors}(\texttt{v\_list}) \xrightarrow{\text{eval}} \texttt{v}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Concat}(\texttt{e\_list})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

6.17.5 Comments

Concatenation of multiple bitvectors is done using a comma separated list surrounded with square brackets.

6.18 SemanticsRule.ETuple

6.18.1 Prose

All of the following apply:

- `e` denotes a tuple expression, E_Tuple(`e_list`);
- the evaluation of `e_list` in `env` is $\text{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env})^{/\!/ \texttt{\#T},\texttt{\#DE}}$;
- `v` is the native vector constructed from the values in `v_list`.

6.18.2 Example

In the specification:

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

func main () => integer
begin
```

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

  return 0;
end
```

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

6.18.3 Formally

$$\frac{\mathit{eval\_expr\_list}(\texttt{env}, \texttt{e\_list}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{v} := \texttt{NV\_Vector}(\texttt{v\_list})}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Tuple}(\texttt{e\_list})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

6.19 SemanticsRule.EUnknown

Definition 31 (Domain of a Type) *The domain of a type* `t` *in an environment* $\mathit{env} = (\mathit{tenv}, \mathit{denv})$*, denoted by* $\mathit{dyn\text{-}dom}(\mathit{tenv}, \mathit{denv}, \texttt{t})$*, is defined by the type system [6] (TypingRule.Domain) as the set of values that* `t` *may store in* env*. The reason that the dynamic environment is needed to determine the domain is due to subprogram parameters, which constrain integer parameters to a singleton value domain.*

6.19.1 Prose

All of the following apply:

- `e` denotes the `UNKNOWN` expression annotated with type `t`;
- `v` is an arbitrary value in the domain of `t` in `env`;
- `new_env` is `env`.
- `g` is the empty execution graph.

6.19.2 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

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

6.19.3 Example

In the specification:

```
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 `Int`(3) or `Int`(42).

6.19.4 Formally

$$\frac{\texttt{env} \overset{\text{is}}{=} (\texttt{tenv}, \texttt{denv}) \qquad \texttt{v} \in \text{dyn-dom}(\texttt{tenv}, \texttt{env}, \texttt{t})}{\textit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_Unknown}(\texttt{t})}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \overbrace{\emptyset_{\texttt{g}}}^{\texttt{g}}), \overbrace{\texttt{env}}^{\texttt{new\_env}})}$$

Notice that this rule introduces non-determinism.

6.19.5 Comments

The expression `UNKNOWN: ty` evaluates to an arbitrary value in the domain of `ty`.

6.20 SemanticsRule.EPattern

6.20.1 Prose

All of the following apply:

- `e` denotes a pattern expression, E_Pattern(`e`, `p`);
- evaluating the expression `e` in an environment `env` results in Normal((`v1`, `g1`), `new_env`)$^{\#\texttt{T},\#\texttt{DE}}$;
- evaluating whether the pattern `p` matches the value `v1` in `env` results in Normal(`v`, `g2`) where is a native Boolean that determines whether the is indeed a match;
- `g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

6.20.2 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

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

6.20.3 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

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

6.20.4 Formally

$$\frac{\mathit{eval\_expr}(\mathsf{env}, \mathsf{e}) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v1}, \mathtt{g1}), \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \qquad \mathit{eval\_pattern}(\mathsf{env}, \mathtt{v1}, \mathsf{p}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{v}, \mathtt{g2}) \qquad \mathtt{g} := \mathtt{g1} \xrightarrow{\text{asl\_data}} \mathtt{g2}}{\mathit{eval\_expr}(\mathsf{env}, \mathsf{E\_Pattern}(\mathsf{e}, \mathsf{p})) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

6.21 SemanticsRule.ATC

The rule about domains in the definitions of subtype-satisfaction and type-satisfaction means that it is illegal to use the unconstrained integer where a constrained integer is expected. An asserting type conversion (ATC) can be used to overcome this.

An ATC allows code to explicitly mark places where uses of constrained types would otherwise be a static type-checking error. The intent is to reduce the incidence of unintended errors by making such uses fail type-checking unless the asserting type conversion is provided.

Note that ATCs are execution-time checks. An execution-time check is a condition that is evaluated during the evaluation of an execution-time initializer expression or subprogram. If the condition evaluates to `FALSE` it is a dynamic error.

6.21.1 Prose

All of the following apply:

- `e` denotes an asserted type conversion expression, E_ATC(`e1`, `t`);
- evaluating `e1` in `env` results in Normal((`v`, `g1`), `new_env`)$^{//\#T,\#DE}$;
- evaluating whether `v` has type `t` in `env` results in (`b`, `g2`)$^{//\#DE}$;
- one of the following applies:
  - all of the following apply (OKAY):
    - `b` is the native Boolean for `TRUE`;
    - `g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.
  - all of the following apply (ERROR):
    - `b` is the native Boolean for `TRUE`;
    - an asserted type conversion error is returned.

6.21.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

6.21.3 Example

```
func main () => integer
begin

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

  return 0;
end
```

6.21.4 Example

Dynamic error conditions only hold if the asserting type conversion is evaluated.

In the example below, the asserting type conversion on `y` is not a dynamic error if the invocation of `f1` returns `FALSE` when evaluated:

```
func f1()
begin
  return FALSE;
end

func checkY (y: integer)
begin
  if (f1() && f2(y as {2,4,8})) then pass; end
end
```

6.21.5 Example

The following excerpts indicates several points where various static and dynamic errors can occur:

```
func ErrorExample()
begin
  var a: integer{1, 2, 3} = 2 as integer{1, 2, 3}; // legal
  var b: integer{4, 5, 6} = 2;                     // static error
  var c: integer{4, 5, 6} = 2 as integer{4, 5, 6}; // dynamic error
  if FALSE then
      var d: integer{4, 5, 6} = 2;                     // static error
      var e: integer{4, 5, 6} = 2 as integer{4, 5, 6}; // not a dynamic error as will never be evaluate
  end
end
```

6.21.6 Formally

OKAY

$$\frac{\mathit{eval\_expr}(\mathsf{env}, \mathsf{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathsf{v}, \mathsf{g1}), \mathsf{new\_env}) \;/\!/\; \#\mathsf{T}, \#\mathsf{DE} \quad \mathit{is\_val\_of\_type}(\mathsf{env}, \mathsf{v}, \mathsf{t}) \xrightarrow{\mathsf{eval}} (\mathsf{b}, \mathsf{g2}) \;/\!/\; \#\mathsf{DE} \quad \mathsf{b} \overset{\mathsf{is}}{=} \mathsf{Bool}(\mathsf{TRUE}) \quad \mathsf{g} := \mathsf{g1} \xrightarrow{\mathsf{asl\_data}} \mathsf{g2}}{\mathit{eval\_expr}(\mathsf{env}, \mathsf{E\_ATC}(\mathsf{e1}, \mathsf{t})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathsf{v}, \mathsf{g}), \mathsf{new\_env})}$$

ERROR

$$\frac{\mathit{eval\_expr}(\mathsf{env}, \mathsf{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathsf{v}, \_), \_) \quad \neg \mathit{is\_val\_of\_type}(\mathsf{env}, \mathsf{v}, \mathsf{t}) \xrightarrow{\mathsf{eval}} (\mathsf{b}, \_) \quad \mathsf{b} \overset{\mathsf{is}}{=} \mathsf{Bool}(\mathsf{FALSE})}{\mathit{eval\_expr}(\mathsf{env}, \mathsf{E\_ATC}(\mathsf{e1}, \mathsf{t})) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\texttt{"ERROR[ATC\_TypeMismatch]"})}$$

6.22 SemanticsRule.EExprList

6.22.1 Prose

The relation

$$\mathit{eval\_expr\_list}(\overbrace{\mathbb{E}}^{\mathsf{env}}, \overbrace{\mathsf{expr}^*}^{\mathsf{le}}) \bigtimes \mathsf{Normal}((\overbrace{\mathbb{V}^*}^{\mathsf{v}} \times \overbrace{\mathcal{G}}^{\mathsf{g}}), \overbrace{\mathbb{E}}^{\mathsf{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\#\mathsf{T}} \cup \overbrace{\mathsf{TDynError}}^{\#\mathsf{DE}}$$

evaluates the list of expressions `le` in left-to-right order in the initial environment `env` and returns the resulting value `v`, the parallel composition of the execution graphs generated from evaluating each expression, and the new environment `new_env`. If the evaluation of any expression terminates abnormally then the abnormal configuration is returned.

6.22.2 Formally

$$\frac{}{\textit{eval\_expr\_list}(\texttt{env}, [\,]) \xrightarrow{\text{eval}} \text{Normal}((\texttt{[ ]}, \emptyset_{\texttt{g}}), \texttt{env})}\ \text{EMPTY}$$

$$\frac{\texttt{le} \overset{\text{is}}{=} [\texttt{e}] + \texttt{le1} \qquad \textit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v1}, \texttt{g1}), \texttt{env1}) \ /\!/\ \texttt{\#T}, \texttt{\#DE} \\ \textit{eval\_expr\_list}(\texttt{env1}, \texttt{le1}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{vs}, \texttt{g2}), \texttt{new\_env}) \ /\!/\ \texttt{\#T}, \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \parallel \texttt{g2} \qquad \texttt{v} := [\texttt{v1}] + \texttt{vs}}{\textit{eval\_expr\_list}(\texttt{env}, \texttt{le}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}\ \text{NONEMPTY}$$

6.23 SemanticsRule.EExprListM

The relation

$$\textit{eval\_expr\_list\_m}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\text{expr}^*}^{\texttt{es}}) \times \text{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\text{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

evaluates a list of expressions `es` in left-to-right in the initial environment `env` and returns the list of values associated with graphs `vms` and the new environment `new_env`. If the evaluation of any expression terminates abnormally then the abnormal configuration is returned.

6.23.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `es` is an empty list;
  - `vms` is then empty list.
- All of the following apply (NON_EMPTY):
  - `es` is a list with `head` `e` and `tail` `es1`;
  - evaluating `e` in `env` yields $\text{Normal}(\texttt{m1}, \texttt{env1})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - evaluating `es1` in `env1` via *eval_expr_list_m* yields $\text{Normal}(\texttt{vms1}, \texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - the result is the normal configuration with the list consisting of `m1` as its `head` and `vms1` as its `tail` and `new_env`.

6.23.2 Formally

EMPTY

$$\mathit{eval\_expr\_list\_m}(\texttt{env}, \overbrace{[\,]}^{\texttt{es}}) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{[\,]}^{\texttt{vms}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

NON_EMPTY

$$\frac{\texttt{es} \overset{\text{is}}{=} [\texttt{e}] + \texttt{es1} \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_expr\_list\_m}(\texttt{env1}, \texttt{es1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms1}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE}}{\mathit{eval\_expr\_list\_m}(\texttt{env}, \texttt{es}) \xrightarrow{\text{eval}} \text{Normal}([\texttt{m1}] + \texttt{vms1}, \texttt{new\_env})}$$

6.24 SemanticsRule.ESideEffectFreeExpr

6.24.1 Prose

The relation

$$\mathit{eval\_expr\_sef}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\texttt{expr}}^{\texttt{e}}) \times \text{Normal}(\overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

specializes the expression evaluation relation for side-effect-free expressions by omitting throwing configurations as possible output configurations.

6.24.2 Formally

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{env}) \;/\!/\; \texttt{\#DE}}{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v}, \texttt{g})}$$

Notice that the output configuration does not contain an environment, since side-effect-free expressions do not modify the environment.

Chapter 7

Evaluation of Left-Hand Side Expressions

The relation

$$\mathit{eval\_lexpr}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{lexpr}}^{\texttt{le}}, (\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}})) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\#\mathsf{T}} \cup \overbrace{\mathsf{TDynError}}^{\#\mathsf{DE}}$$

evaluates the assignment of a value `v` to the left-hand-side expression `le` in an environment `env`, resulting in either a configuration Normal(`new_g`, `env`) or an abnormal configuration.

One of the following applies:

- SemanticsRule.LEDiscard (see Section 7.1);
- SemanticsRule.LELocalVar (see Section 7.2);
- SemanticsRule.LEGlobalVar (see Section 7.3);
- SemanticsRule.LESlice (see Section 7.4);
- SemanticsRule.LESetArray (see Section 7.5);
- SemanticsRule.LESetField (see Section 7.6);
- SemanticsRule.LEDestructuring (see Section 7.7).

We also define the helper rule SemanticsRule.LEMultiAssign (7.8).

Some of the rules require viewing left-hand-side expressions as their corresponding right-hand side expressions. The correspondence is defined in the ASL Syntax Reference [5, Chapter 5] and given by the function rexpr : lexpr → expr. For example, SemanticsRule.LESetField needs to evaluate the record subexpression `re_record`, which is a left-hand-side expression. To achieve this, rexpr(`record`) is used to obtain a right-hand-side expression, which then allows using `eval_expr` to evaluate it.

Naming Convention: In this chapter, variables containing `m` range over $\mathbb{V} \times \mathcal{G}$ while variables where the `m` is replaced with `v` correspond to their value component. For example, `rm_array` $\overset{\text{is}}{=}$ (`rv_array`, `g2`) and `m_index` $\overset{\text{is}}{=}$ (`index`, `g1`).

7.1 SemanticsRule.LEDiscard

7.1.1 Prose

All of the following apply:

- `le` is a discarding expression, LE_Discard;
- `new_g` is `g`;
- `new_env` is `env`.

7.1.2 Example

In the specification:

```
func main () => integer
begin

  - = 42;
  assert TRUE;

  return 0;
end
```

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

7.1.3 Formally

$$\frac{\texttt{new\_g} := \texttt{g} \qquad \texttt{new\_env} := \texttt{env}}{\mathit{eval\_lexpr}(\texttt{env}, \mathsf{LE\_Discard}, (\texttt{v}, \texttt{g})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

7.2 SemanticsRule.LELocalVar

7.2.1 Prose

All of the following apply:

- `le` denotes a variable, LE_Var(`x`);
- `x` is locally bound in `env`;
- `new_g` is the ordered composition of `g` and a Write Effect for `x` with the `asl_data` edge;
- `new_env` is `env` where `x` is bound to `v` in the local component of the environment.

7.2.2 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

SemanticsRule.LELocalVar is (only) used to assign the value 42 to the left-hand-side expression `x` within `x = 42;`.

7.2.3 Formally

$$\frac{\texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \qquad \texttt{x} \in \mathrm{dom}(L^{\mathsf{denv}}) \qquad \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}[\texttt{x} \mapsto \texttt{v}])) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \mathsf{WriteEffect}(\texttt{x})}{\mathit{eval\_lexpr}(\texttt{env}, \mathsf{LE\_Var}(\texttt{x}), (\texttt{v}, \texttt{g})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

7.3 SemanticsRule.LEGlobalVar

7.3.1 Prose

All of the following apply:

- `le` denotes a variable `x`, LE_Var(`x`);
- `x` is globally bound in `env`;
- `new_g` is the ordered composition of `g` and a Write Effect for `x` with the `asl_data` edge;
- `new_env` is `env` where `x` is bound to `v` in the global component of the environment.

7.3.2 Example

In the specification:

```
var x: integer = 3;

func main () => integer
begin

  x = 42;
```

```
  assert x==42;

  return 0;
end
```

SemanticsRule.LEGlobalVar is (only) used to assign the value 42 to the left-hand-side expression `x` within `x = 42;`.

7.3.3 Formally

$$\frac{\mathsf{env} \overset{\text{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \qquad \mathtt{x} \in \mathrm{dom}(G^{\mathsf{denv}}) \qquad \mathtt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}[\mathtt{x} \mapsto \mathtt{v}], L^{\mathsf{denv}})) \qquad \mathtt{new\_g} := \mathtt{g} \xrightarrow{\mathtt{asl\_data}} \mathsf{WriteEffect}(\mathtt{x})}{\mathit{eval\_lexpr}(\mathtt{env}, \mathsf{LE\_Var}(\mathtt{x}), (\mathtt{v}, \mathtt{g})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{new\_g}, \mathtt{new\_env})}$$

7.4 SemanticsRule.LESlice

7.4.1 Prose

All of the following apply:

- `le` denotes a left-hand-side slicing expression, LE_Slice(`e_bv`, `slices`);
- evaluating the right-hand-side expression that corresponds to `e_bv` (given by applying rexpr to `e_bv`) in env is Normal(`m_bv`, `env1`)$^{\#\mathtt{T},\#\mathtt{DE}}$;
- evaluating `slices` in `env1` is Normal(`m_positions`, `env2`)$^{\#\mathtt{T},\#\mathtt{DE}}$;
- `m_positions` consists of the execution graph `g1` and the list of indices `positions`;
- `m_bv` consists of the native bitvector `v_bv` and the execution graph `g2`;
- writing to the bitvector `v_bv` at indices `positions` using the values from `v` results in the updated native bitvector `v1`$^{\#\mathtt{DE}}$;
- `g3` is the parallel composition of `g`, `g1`, and `g2`;
- `new_m_bv` is a pair consisting of `v1` and the execution graph `g3`;
- the steps so far computed the updated bitvector, but have not assigned it to the variable bound to the bitvector given by `e_bv`, which is achieved next. Evaluating the left-hand-side expression `e_bv` with `new_m_bv` in an environment `env2` is the output configuration C,

7.4.2 Example

In the specification:

```
func main () => integer
begin

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

  return 0;
end
```

The assignment `x[3:0] = '0000'` binds `x` to `Bitvector`(11110000) via the rule SemanticsRule.LESlice.asl in the environment where `x` is bound to `Bitvector`(11111111).

7.4.3 Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\mathtt{env}, \mathsf{rexpr}(\mathtt{e\_bv})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_bv}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathit{eval\_slices}(\mathtt{env1}, \mathtt{slices}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_positions}, \mathtt{env2}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_positions} \overset{\mathsf{is}}{=} (\mathtt{positions}, \mathtt{g1}) \qquad \mathtt{m\_bv} \overset{\mathsf{is}}{=} (\mathtt{v\_bv}, \mathtt{g2}) \\ \mathit{write\_to\_bitvector}(\mathtt{positions}, \mathtt{v}, \mathtt{v\_bv}) \xrightarrow{\mathsf{eval}} \mathtt{v1} \;/\!/\; \mathtt{\#DE} \\ \mathtt{g3} := \mathtt{g} \parallel \mathtt{g1} \parallel \mathtt{g2} \qquad \mathtt{new\_m\_bv} := (\mathtt{v1}, \mathtt{g3}) \qquad \mathit{eval\_lexpr}(\mathtt{env2}, \mathtt{e\_bv}, \mathtt{new\_m\_bv}) \xrightarrow{\mathsf{eval}} C \end{array}}{\mathit{eval\_lexpr}(\mathtt{env2}, \mathsf{LE\_Slice}(\mathtt{e\_bv}, \mathtt{slices}), (\mathtt{v}, \mathtt{g})) \xrightarrow{\mathsf{eval}} C}$$

7.4.4 Comments

If the declared type of a setter's RHS argument has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

7.5 SemanticsRule.LESetArray

7.5.1 Prose

All of the following apply:

- `le` denotes an array update expression, LE_SetArray(`re_array`, `e_index`);

- evaluating the right-hand-side expression corresponding to `re_array` in `env` is Normal(`rm_array`, `env1`)$^{/\!/\#T,\#DE}$;
- evaluating `e_index` in `env1` is Normal(`m_index`, `env2`)$^{/\!/\#T,\#DE}$;
- `m_index` consists of the native integer `index` and the execution graph `g1`;
- `index` is the native integer for `i`;
- `rm_array` consists of the native vector `rv_array` and the execution graph `g2`;
- setting the value `v` at index `i` of `rv_array` is the native vector `v1`;
- `m1` is the pair consisting of `v1` and the parallel composition of `g1` and `g2`;
- the steps so far computed the updated array, but have not assigned it to the variable bound to the array given by `re_array`, which is achieved next. Evaluating the left-hand-side expression `re_array` in an environment `env2` with `m1` is the output configuration C.

7.5.2 Example

The specification:

```
func main () => integer
begin

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

  return 0;
end
```

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

7.5.3 Formally

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathtt{env}, \mathsf{rexpr}(\mathtt{re\_array})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{rm\_array}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathit{eval\_expr}(\mathtt{env1}, \mathtt{e\_index}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_index}, \mathtt{env2}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_index} \overset{\mathsf{is}}{=} (\mathtt{index}, \mathtt{g1}) \\ \mathtt{index} \overset{\mathsf{is}}{=} \mathtt{Int}(\mathtt{i}) \qquad \mathtt{rm\_array} \overset{\mathsf{is}}{=} (\mathtt{rv\_array}, \mathtt{g2}) \qquad \mathit{set\_index}(\mathtt{i}, \mathtt{v}, \mathtt{rv\_array}) \xrightarrow{\mathsf{eval}} \mathtt{v1} \\ \mathtt{m1} := (\mathtt{v1}, \mathtt{g1} \parallel \mathtt{g2}) \qquad \mathit{eval\_lexpr}(\mathtt{env2}, \mathtt{re\_array}, \mathtt{m1}) \xrightarrow{\mathsf{eval}} C\end{array}}{\mathit{eval\_lexpr}(\mathtt{env}, \mathsf{LE\_SetArray}(\mathtt{re\_array}, \mathtt{e\_index}), (\mathtt{v}, \mathtt{g})) \xrightarrow{\mathsf{eval}} C}$$

7.5.4 Comments

If the declared type of a setter's RHS argument has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

We note that the index is guaranteed by the type-checker to be within the array bounds via TypingRule.LESetArray (see the ASL Typing Reference [6] chapter "Typing of Left-Hand-Side Expression").

7.6 SemanticsRule.LESetField

7.6.1 Prose

All of the following apply:

- `le` denotes a field update expression, LE_SetField(`re_record`, `field_name`);
- evaluating the right-hand-side expression corresponding to `re_record` in `env` is Normal(`rm_record`, `env1`)$^{\#T,\#DE}$;
- `rm_record` is a pair consisting of the native record `rv_record` and the execution graph `g1`;
- setting the field `field_name` in the native record `rv_record` to `v` is the updated native record `v1`;
- `m1` is the pair consisting of the native vector `v1` and the execution graph that is, the parallel composition of `g` and `g1`;
- the steps so far computed the updated record, but have not assigned it to the variable holding the record given by `record`, which is achieved next. Evaluating the left-hand-side expression `re_record` in an environment `env1` with `m1` is the output configuration C.

7.6.2 Example

In the specification:

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

func main () => integer
begin

  var my_record = MyRecordType { a = 3, b = 100 };
  my_record.a = 42;
  assert my_record.a == 42 && my_record.b == 100;

  return 0;
end
```

`my_record.a = 42;` binds `my_record` to {`a: 42, b: 100`} in the environment where `my_record` is bound to {`a: 3, b: 100`}.

7.6.3 Formally

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathsf{env}, \mathsf{rexpr}(\texttt{re\_record})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{rm\_record}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{rm\_record} \overset{\mathsf{is}}{=} (\texttt{rv\_record}, \texttt{g1}) \qquad \mathit{set\_field}(\texttt{field\_name}, \texttt{v}, \texttt{rv\_record}) \xrightarrow{\mathsf{eval}} \texttt{v1} \\ \texttt{m1} := (\texttt{v1}, \texttt{g} \parallel \texttt{g1}) \qquad \mathit{eval\_lexpr}(\mathsf{env}1, \texttt{re\_record}, \texttt{m1}) \xrightarrow{\mathsf{eval}} C\end{array}}{\mathit{eval\_lexpr}(\mathsf{env}, \mathsf{LE\_SetField}(\texttt{re\_record}, \texttt{field\_name}), (\texttt{v}, \texttt{g})) \xrightarrow{\mathsf{eval}} C}$$

7.6.4 Comments

We note that the type-checker guarantees that `field_name` exists in the record given by `record` via TypingRule.LESetStructuredField.

If the declared type of a setter's RHS argument has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

7.7 SemanticsRule.LEDestructuring

7.7.1 Prose

All of the following apply:

- `le` denotes a list of left-hand-side expressions, LE_Destructuring(`le_list`);
- `le_list` is the list of expressions $\texttt{le}_{1..k}$;

- getting the values from the native vector `v` at each index $i = 1..k$ results in $\mathtt{v}_{i=1..k}$;
- `nmonads` is the list of pairs consisting of $\mathtt{v}_i$ and `g` for $i = 1..k$;
- evaluating the multi-assignment between `le_list` and the list `nmonads` in `env` achieves the effects of assigning each value to the respective subexpressions, resulting in the output configuration C.

7.7.2 Example

In the specification:

```
func main () => integer
begin

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

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

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

  return 0;
end
```

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

7.7.3 Formally

$$\frac{\mathtt{le\_list} \overset{\text{is}}{=} [\mathtt{le}_{1..k}] \qquad i = 1..k : \mathit{get\_index}(i, \mathtt{v}) \xrightarrow{\text{eval}} \mathtt{v}_i \qquad \mathtt{nmonads} := [i = 1..k : (\mathtt{v}_i, \mathtt{g})] \qquad \mathit{multi\_assign}(\mathtt{env}, \mathtt{le\_list}, \mathtt{nmonads}) \xrightarrow{\text{eval}} C}{\mathit{eval\_lexpr}(\mathtt{env}, \mathsf{LE\_Destructuring}(\mathtt{le\_list}), (\mathtt{v}, \mathtt{g})) \xrightarrow{\text{eval}} C}$$

7.8 SemanticsRule.LEMultiAssign

7.8.1 Prose

The helper relation

$$\mathit{multi\_assign}(\overbrace{\mathbb{E}}^{\mathtt{env}}, \overbrace{\mathsf{expr}^*}^{\mathtt{le\_list}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\mathtt{vm\_list}}) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\mathtt{new\_g}}, \overbrace{\mathbb{E}}^{\mathtt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\#\mathtt{T}} \cup \overbrace{\mathsf{TDynError}}^{\#\mathtt{DE}}$$

evaluates multi-assignments. That is, the simultaneous assignment of the list of value-execution graph pairs `vm_list` to the corresponding list of left-hand side expressions `le_list`, in the environment `env`. The result is either the execution graph `g` and new environment `new_env` or an abnormal configuration.

7.8.2 Formally

$$\text{EMPTY}\quad \frac{}{\mathit{multi\_assign}(\texttt{env}, [\,], [\,]) \xrightarrow{\text{eval}} \text{Normal}(\emptyset_{\texttt{g}}, \texttt{env})}$$

$$\text{NONEMPTY}\quad \frac{\begin{array}{c}\texttt{le\_list} \overset{\text{is}}{=} [\texttt{le}] + \texttt{le\_list1} \\ \texttt{vm\_list} \overset{\text{is}}{=} [\texttt{m}] + \texttt{vm\_list1} \qquad \mathit{eval\_lexpr}(\texttt{env}, \texttt{le}, \texttt{m}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{env1}, \texttt{g1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{multi\_assign}(\texttt{env1}, \texttt{le\_list1}, \texttt{vm\_list1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_env}, \texttt{g2}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_po}} \texttt{g2}\end{array}}{\mathit{multi\_assign}(\texttt{env}, \texttt{le\_list}, \texttt{vm\_list}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

Notice that this rule is only defined when the lists `le_list` and `vm_list` have the same length. To see this, notice that to form a derivation tree, we must employ the NONEMPTY case, which ensures both lists have at least one element and shortens the lengths of both lists by one, until both lists become empty which is when the EMPTY axiom case is used.

Chapter 8

Evaluation of Slices

The rule for evaluating a list of slices is SemanticsRule.Slices (see Section 8.1).

The relation for evaluating a single slice is

$$\mathit{eval\_slice}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{slice}}^{\texttt{s}}) \times \mathsf{Normal}(((\overbrace{\mathbb{Z}}^{\texttt{v\_start}} \times \overbrace{\mathbb{Z}}^{\texttt{v\_length}}) \times \overbrace{\mathcal{G}}^{\texttt{new\_g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{Throwing}}^{\#\texttt{T}} \cup \overbrace{\mathsf{TDynError}}^{\#\texttt{DE}}$$

where a single slice `s` is evaluated in an environment `env` is, resulting either in Normal(((`v_start`, `v_length`), `g`), `new_env`) or an error configuration, and one of the following applies:

- SemanticsRule.SliceSingle (see Section 8.2),
- SemanticsRule.SliceLength (see Section 8.3),
- SemanticsRule.SliceRange (see Section 8.4),
- SemanticsRule.SliceStar (see Section 8.5).

8.1 SemanticsRule.Slices

8.1.1 Prose

The relation

$$\mathit{eval\_slices}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices}}) \times \mathsf{Normal}(((\overbrace{(\mathbb{V} \times \mathbb{V})^*}^{\texttt{ranges}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{Throwing}}^{\#\texttt{T}} \cup \overbrace{\mathsf{TDynError}}^{\#\texttt{DE}}$$

evaluates a list of slices `slices` in an environment `env`, resulting in either Normal((`ranges`, `new_g`), `new_env`) or an abnormal configuration.

One of the following applies:

- All of the following apply (EMPTY):
  - the list of slices is empty;
  - `ranges` is the empty list;
  - `new_g` is the empty graph;
  - `new_env` is `env`;
- All of the following apply (NONEMPTY):
  - the list of slices has slice as the head and `slices1` as the tail;
  - evaluating the slice `slice` in `env` results in $\mathsf{Normal}((\texttt{range}, \texttt{g1}), \texttt{env1})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - evaluating the tail list `slices1` in `env1` results in $\mathsf{Normal}((\texttt{ranges1}, \texttt{g2}), \texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - `ranges` is the concatenation of `range` to `ranges1`;
  - `new_g` is the parallel composition of `g1` and `g2`.

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

8.1.2 Formally

EMPTY

$$\mathit{eval\_slices}(\texttt{env}, [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(([\,], \emptyset_g), \texttt{env})$$

NONEMPTY

$$\frac{\begin{array}{c}\texttt{slices} \overset{\mathsf{is}}{=} [\texttt{slice}] + \texttt{slices1} \\ \mathit{eval\_slice}(\texttt{env}, \texttt{slice}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{range}, \texttt{g1}), \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_slices}(\texttt{env1}, \texttt{slices1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{ranges1}, \texttt{g2}), \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{ranges} := [\texttt{range}] + \texttt{ranges1} \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\mathit{eval\_slices}(\texttt{env}, \texttt{slices}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{ranges}, \texttt{new\_g}), \texttt{new\_env})}$$

8.2 SemanticsRule.SliceSingle

8.2.1 Prose

All of the following apply:

- `s` is a single value slicing expression, Slice_Single(`e`);
- evaluating `e` in `env` results in $\mathsf{Normal}((\texttt{v\_start}, \texttt{new\_g})\texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `v_length` is the integer value 1.

8.2.2 Example

In the specification:

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

  assert x[2] == '1';

  return 0;
end
```

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

8.2.3 Formally

$$\frac{\mathit{eval\_expr}(\mathsf{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v\_start}, \mathtt{new\_g}), \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \quad \mathtt{v\_length} := \mathsf{Int}(1)}{\mathit{eval\_slice}(\mathsf{env}, \mathsf{Slice\_Single}(\mathsf{e})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(((\mathtt{v\_start}, \mathtt{v\_length}), \mathtt{new\_g}), \mathtt{new\_env})}$$

8.3 SemanticsRule.SliceLength

8.3.1 Prose

All of the following apply:

- `s` is the slice which starts at expression `e_start` with length `length`, that is, Slice_Length(`e_start`, `length`);
- evaluating `e_start` in env is Normal(`m_start`, `env1`)$^{/\!/\mathtt{\#T,\#DE}}$;
- evaluating `length` in `env1` is Normal(`m_length`, `new_env`)$^{/\!/\mathtt{\#T,\#DE}}$;
- `m_start` is a pair consisting of the native integer `v_start` and execution graph `g1`;
- `m_length` is a pair consisting of the native integer `v_length` and execution graph `g2`;
- `new_g` is the parallel composition of `g1` and `g2`.

8.3.2 Example

In the specification:

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

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

  return 0;
end
```

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

8.3.3 Formally

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e\_start}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_start}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{length}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_length}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_start} \overset{\text{is}}{=} (\texttt{v\_start}, \texttt{g1}) \qquad \texttt{m\_length} \overset{\text{is}}{=} (\texttt{v\_length}, \texttt{g2}) \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\texttt{env}, \text{Slice\_Length}(\texttt{e\_start}, \texttt{length})) \xrightarrow{\text{eval}} \\ \text{Normal}(((\texttt{v\_start}, \texttt{v\_length}), \texttt{new\_g}), \texttt{new\_env})\end{array}}$$

8.4 SemanticsRule.SliceRange

8.4.1 Prose

All of the following apply:

- `s` is the slice range between the expressions `e_start` and `e_top`, that is, Slice_Range(`e_top`, `e_start`);
- evaluating `e_top` in `env` is Normal(`m_top`, `env1`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `m_top` is a pair consisting of the native integer `v_top` and execution graph `g1`;
- evaluating `e_start` in `env1` is Normal(`m_start`, `new_env`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `m_start` is a pair consisting of the native integer `v_start` and execution graph `g2`;
- `v_length` is the integer value `(v_top - v_start) + 1`;
- `new_g` is the parallel composition of `g1` and `g2`.

8.4.2 Example

In the specification:

```
func main () => integer
begin

  let x = '00011100';

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

  return 0;
end
```

`4:2` evaluates to `(2, 3)`.

8.4.3 Formally

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathtt{env}, \mathtt{e\_top}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{m\_top}, \mathtt{env1}) \;/\!/\; \#\mathtt{T}, \#\mathtt{DE} \\ \mathtt{m\_top} \overset{\text{is}}{=} (\mathtt{v\_top}, \mathtt{g1}) \\ \mathit{eval\_expr}(\mathtt{env1}, \mathtt{e\_start}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{m\_start}, \mathtt{new\_env}) \;/\!/\; \#\mathtt{T}, \#\mathtt{DE} \\ \mathtt{m\_start} \overset{\text{is}}{=} (\mathtt{v\_start}, \mathtt{g2}) \qquad \mathit{binop}(\mathtt{MINUS}, \mathtt{v\_top}, \mathtt{v\_start}) \xrightarrow{\text{eval}} \mathtt{v\_diff} \\ \mathit{binop}(\mathtt{PLUS}, \mathtt{Int}(1), \mathtt{v\_diff}) \xrightarrow{\text{eval}} \mathtt{v\_length} \qquad \mathtt{new\_g} := \mathtt{g1} \parallel \mathtt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\mathtt{env}, \text{Slice\_Range}(\mathtt{e\_top}, \mathtt{e\_start})) \xrightarrow{\text{eval}} \\ \text{Normal}(((\mathtt{v\_start}, \mathtt{v\_length}), \mathtt{new\_g}), \mathtt{new\_env})\end{array}}$$

8.5 SemanticsRule.SliceStar

8.5.1 Prose

All of the following apply:

- `s` is the slice with factor given by the expression `factor` and length given by the expression `length`, that is, Slice_Star(`factor`, `length`);
- evaluating `factor` in `env` is Normal(`m_factor`, `env1`)$^{/\!/\#\mathtt{T},\#\mathtt{DE}}$;
- `m_factor` is a pair consisting of the native integer `v_factor` and execution graph `g1`;
- evaluating `length` in `env` is Normal(`m_length`, `new_env`)$^{/\!/\#\mathtt{T},\#\mathtt{DE}}$;
- `m_length` is a pair consisting of the native integer `v_length` and execution graph `g2`;
- `v_start` is the native integer `v_factor` $\times$ `v_length`;
- `new_g` is the parallel composition of `g1` and `g2`.

8.5.2 Example

In the specification:

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

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

  return 0;
end
```

`x[3*:2]` evaluates to `'11'`.

8.5.3 Formally

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathsf{env}, \mathtt{factor}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_factor}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_factor} \overset{\mathsf{is}}{=} (\mathtt{v\_factor}, \mathtt{g1}) \\ \mathit{eval\_expr}(\mathtt{env1}, \mathtt{length}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m\_length}, \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_length} \overset{\mathsf{is}}{=} (\mathtt{v\_length}, \mathtt{g2}) \\ \mathit{binop}(\mathtt{MUL}, \mathtt{v\_factor}, \mathtt{v\_length}) \xrightarrow{\mathsf{eval}} \mathtt{v\_start} \qquad \mathtt{new\_g} := \mathtt{g1} \parallel \mathtt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\mathsf{env}, \mathsf{Slice\_Star}(\mathtt{factor}, \mathtt{length})) \xrightarrow{\mathsf{eval}} \\ \mathsf{Normal}(((\mathtt{v\_start}, \mathtt{v\_length}), \mathtt{new\_g}), \mathtt{new\_env})\end{array}}$$

Chapter 9

Evaluation of Patterns

The relation

$$\mathit{eval\_pattern}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathsf{pattern}}^{\texttt{p}}) \times \mathsf{Normal}(\overbrace{\mathcal{B}}^{\texttt{b}}, \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

determines whether a value `v` matches the pattern `p` in an environment `env` resulting in either Normal(`b`, `new_g`) or an abnormal configuration, and one of the following applies:

- SemanticsRule.PAll (see Section 9.1)
- SemanticsRule.PAny (see Section 9.2)
- SemanticsRule.PGeq (see Section 9.3)
- SemanticsRule.PLeq (see Section 9.4)
- SemanticsRule.PNot (see Section 9.5)
- SemanticsRule.PRange (see Section 9.6)
- SemanticsRule.PSingle (see Section 9.7)
- SemanticsRule.PMask (see Section 9.8)
- SemanticsRule.PTuple (see Section 9.9)

9.1 SemanticsRule.PAll

9.1.1 Prose

All of the following apply:

- `p` is the pattern which matches everything, Pattern_All, and therefore matches `v`;
- `b` is the native Boolean value `TRUE`;
- `new_g` is the empty graph.

9.1.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.1.3 Formally

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

9.2 SemanticsRule.PAny

9.2.1 Prose

All of the following apply:

- p is a list of patterns, Pattern_Any(ps);
- ps is $\mathtt{p}_{1..k}$;
- evaluating each pattern $\mathtt{p}_i$ in env results in $\mathsf{Normal}(\texttt{Bool}(\mathtt{b}_i), \mathtt{g}_i)^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- b is the native Boolean which is the disjunction of $\mathtt{b}_i$, for $i = 1..k$;
- new_g is the parallel composition of all execution graphs $\mathtt{g}_i$, for $i = 1..k$.

9.2.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.2.3 Example

```
func main () => integer
begin

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

```
  return 0;
end
```

9.2.4 Formally

$$\frac{\texttt{ps} \overset{\text{is}}{=} \texttt{p}_{1..k} \qquad i = 1..k : \mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \texttt{p}_i) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathsf{Bool}(\texttt{b}_i), \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \qquad \texttt{b} := \mathsf{Bool}(\bigvee_{i=1..k} \texttt{b}_i) \qquad \texttt{new\_g} := \texttt{g}_1 \parallel \ldots \parallel \texttt{g}_k}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \mathsf{Pattern\_Any}(\texttt{ps})) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{b}, \texttt{new\_g})}$$

9.3 SemanticsRule.PGeq

9.3.1 Prose

All of the following apply:

- p is the condition corresponding to being greater than or equal than the side-effect-free expression e, Pattern_Geq(e);
- the side-effect-free evaluation of e is either $\mathsf{Normal}(\texttt{v1}, \texttt{new\_g})^{/\!/\texttt{\#DE}}$;
- b is the Boolean value corresponding to whether v is greater than or equal to v1.

9.3.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.3.3 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.3.4 Formally

$$\frac{\mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{v1}, \mathtt{new\_g}) \;/\!/\; \mathtt{\#DE} \qquad \mathit{binop}(\mathtt{GEQ}, \mathtt{v}, \mathtt{v1}) \xrightarrow{\mathsf{eval}} \mathtt{b}}{\mathit{eval\_pattern}(\mathsf{env}, \mathtt{v}, \mathsf{Pattern\_Geq}(\mathsf{e})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{b}, \mathtt{new\_g})}$$

9.4 SemanticsRule.PLeq

9.4.1 Prose

All of the following apply:

- p is the condition corresponding to being less than or equal to the side-effect-free expression e, Pattern_Leq(e);
- the side-effect-free evaluation of e is either $\mathsf{Normal}(\mathtt{v1}, \mathtt{new\_g})^{/\!/\mathtt{\#DE}}$;
- b is the Boolean value corresponding to whether v is less than or equal to v1.

9.4.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.4.3 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.4.4 Formally

$$\frac{\mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{v1}, \mathtt{new\_g}) \;/\!/\; \mathtt{\#DE} \qquad \mathit{binop}(\mathtt{LEQ}, \mathtt{v}, \mathtt{v1}) \xrightarrow{\mathsf{eval}} \mathtt{b}}{\mathit{eval\_pattern}(\mathsf{env}, \mathtt{v}, \mathsf{Pattern\_Leq}(\mathsf{e})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{b}, \mathtt{new\_g})}$$

9.5 SemanticsRule.PNot

9.5.1 Prose

All of the following apply:

- p is a negation pattern, Pattern_Not(p1);
- evaluating that pattern p1 in an environment env is Normal(b1, new_g) // #DE;
- b is the Boolean negation of b1.

9.5.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.5.3 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.5.4 Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{p1}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{b1}, \mathtt{new\_g}) \;\; // \;\; \#\mathtt{DE} \\ \mathit{unop}(\mathtt{BNOT}, \mathtt{b1}) \xrightarrow{\text{eval}} \mathtt{b} \end{array}}{\mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \text{Pattern\_Not}(\mathtt{p1})) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{b}, \mathtt{new\_g})}$$

9.6 SemanticsRule.PRange

9.6.1 Prose

All of the following apply:

- `p` is the condition corresponding to being greater than or equal to `e1`, and lesser or equal to `e2`, that is, Pattern_Range(`e1`, `e2`);
- `e1` and `e2` are side-effect-free expressions;
- the side-effect-free evaluation of `e1` in env is Normal(`v1`, `g1`)$^{/\!/\texttt{\#DE}}$;
- the side-effect-free evaluation of `e2` in env is Normal(`v2`, `g2`)$^{/\!/\texttt{\#DE}}$;
- `b1` is the Boolean value corresponding to whether `v` is greater than or equal to `v1`;
- `b2` is the Boolean value corresponding to whether `v` is less than or equal to `v2`;
- `b` is the Boolean conjunction of `b1` and `b2`;
- `new_g` is the parallel composition of `g1` and `g2`.

9.6.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.6.3 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.6.4 Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{binop}(\texttt{GEQ}, \texttt{v}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b1} \qquad \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v2}, \texttt{g2}) \;/\!/\; \texttt{\#DE} \\ \mathit{binop}(\texttt{LEQ}, \texttt{v}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{b2} \qquad \mathit{binop}(\texttt{BAND}, \texttt{b1}, \texttt{b2}) \xrightarrow{\text{eval}} \texttt{b} \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2} \end{array}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \text{Pattern\_Range}(\texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

9.7 SemanticsRule.PSingle

9.7.1 Prose

All of the following apply:

- p is the condition corresponding to being equal to the side-effect-free expression e, Pattern_Single(e);
- the side-effect-free evaluation of e in environment env is Normal(v1, new_g) // #DE;
- b is the Boolean value corresponding to whether v is equal to v1.

9.7.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.7.3 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.7.4 Formally

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{new\_g}) \;//\; \#\text{DE} \qquad \mathit{binop}(\texttt{EQ\_OP}, \texttt{v1}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \text{Pattern\_Single}(\texttt{e})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

9.8 SemanticsRule.PMask

9.8.1 Prose

All of the following apply:

- `p` is a mask pattern, Pattern_Mask(`m`), of length n (with spaces removed);
- `v` is a native bitvector of bits $\mathtt{u}_{1..n}$;
- `b` is the native Boolean formed from the conjunction of Boolean values for each i, where the bit $\mathtt{u}_i$ is checked for matching the mask character $\mathtt{m}_i$;
- `new_g` is the empty graph.

9.8.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.8.3 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.8.4 Formally

The helper function mask_match : $\{0, 1, \mathtt{x}\} \times \{0, 1\} \to \mathbb{B}$, checks whether a bit value (second operand) matches a mask value (first operand), is defined by the following table:

mask_match	0	1	x
0	TRUE	FALSE	TRUE
1	FALSE	TRUE	TRUE

$$\frac{\mathtt{m} \overset{\text{is}}{=} \mathtt{m}_{1..n} \qquad \mathtt{v} \overset{\text{is}}{=} \mathtt{Bitvector}(\mathtt{u}_{1..n}) \qquad \mathtt{b} := \mathtt{Bool}(\bigwedge_{i=1..n} \text{mask\_match}(\mathtt{m}_i, \mathtt{u}_i))}{\mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \text{Pattern\_Mask}(\mathtt{m})) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{b}, \emptyset_{\mathtt{g}})}$$

9.9 SemanticsRule.PTuple

9.9.1 Prose

All of the following apply:

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

9.9.2 Example

```
func main () => integer
begin

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

  return 0;
end
```

9.9.3 Example

```
func main () => integer
begin

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

  return 0;
end
```

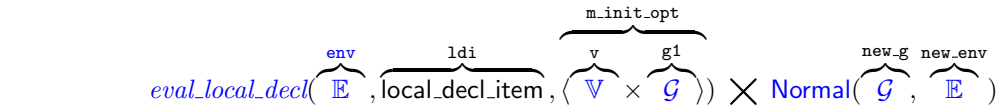
9.9.4 Formally

$$\frac{\begin{array}{c}\mathtt{ps} \overset{\text{is}}{=} \mathtt{p}_{1..k} \qquad i = 1..k : \mathit{get\_index}(i, \mathtt{v}) \xrightarrow{\text{eval}} \mathtt{vs}_i \\ i = 1..k : \mathit{eval\_pattern}(\mathtt{env}, \mathtt{vs}_i, \mathtt{p}_i) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{Bool}(\mathtt{bs}_i), \mathtt{g}_i) \;\; // \;\; \texttt{\#DE} \\ \mathtt{res} := \mathtt{Bool}(\bigwedge_{i=1..k} \mathtt{bs}_i) \qquad \mathtt{g} := \mathtt{g}_1 \parallel \ldots \parallel \mathtt{g}_k\end{array}}{\mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathsf{Pattern\_Tuple}(\mathtt{ps})) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{res}, \emptyset_{\mathtt{g}})}$$

Chapter 10

Evaluation of Local Declarations

The relation



evaluates a local declaration of one or more variables `ldi` in env with an optional initialisation value `m_init_opt`. That is, the right-hand side of the declaration, if it exists, has already been evaluated, yielding `m_init_opt` (see, for example, SemanticsRule.SDeclSome in Section 11.20). Evaluation of the local variables `ldi` in an environment env is either Normal(g, `new_env`) or an abnormal configuration and one of the following applies:

- SemanticsRule.LDDiscard (see Section 10.1),
- SemanticsRule.LDVar (see Section 10.2),
- SemanticsRule.LDTyped (see Section 10.3),
- SemanticsRule.LDTuple (see Section 10.4),
- SemanticsRule.LDUninitialisedTyped (see Section 10.5),

Recall that ASL has three different categories of variable declarations — constants, mutable variables (declared via `var`), and immutable variables (declared via `let`). From the perspective of evaluating the semantics of local declarations (and local declarations statements in Chapter 11), they are all treated the same way.

10.1 SemanticsRule.LDDiscard

10.1.1 Prose

All of the following apply:

- `ldi` indicates that the initialisation value will be discarded, LDI_Discard;
- `new_g` is the empty graph;
- `new_env` is `env`.

10.1.2 Example

In the specification:

```
func main () => integer
begin

  var - : integer;
  assert TRUE;

  return 0;
end
```

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

10.1.3 Formally

$$\mathit{eval\_local\_decl}(\texttt{env}, \overbrace{\mathsf{LDI\_Discard}}^{\texttt{ldi}}, \overbrace{\_}^{\texttt{m\_init\_opt}}) \xrightarrow{\text{eval}} \mathsf{Normal}(\overbrace{\emptyset_g}^{\texttt{new\_g}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

10.2 SemanticsRule.LDVar

10.2.1 Prose

All of the following apply:

- `ldi` is a variable declaration, LDI_Var(`x`);
- `m_init_opt` is `m`;
- `m` is a pair consisting of the value `v` and execution graph `g1`;
- declaring `x` in `env` is (`new_env`, `g2`);
- `new_g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

10.2.2 Example

In the specification:

```
func main () => integer
begin

  var x = 3;

  assert x == 3;

  return 0;
end
```

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

10.2.3 Example

In the specification:

```
func main () => integer
begin

  var x : integer = 3;

  assert x == 3;

  return 0;
end
```

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

10.2.4 Formally

$$\frac{\texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g1}) \qquad \textit{declare\_local\_identifier}(\texttt{env}, \texttt{x}, \texttt{v}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g2}) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2}}{\textit{eval\_local\_decl}(\texttt{env}, \mathsf{LDI\_Var}(\texttt{x}), \langle \texttt{m} \rangle) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

10.3 SemanticsRule.LDTyped

10.3.1 Prose

All of the following apply:

- `ldi` is a typed declaration, $\mathsf{LDI\_Typed}(\texttt{ldi1}, \texttt{t})$;
- `m_init_opt` is `m`;
- the resulting configuration is obtained via the evaluation of the local declaration `ldi1` in `env` with `m_init_opt` as `m`, that is, $\textit{eval\_local\_decl}(\texttt{env}, \texttt{ldi1}, \langle \texttt{m} \rangle)$.

10.3.2 Example

In the specification:

```
func main () => integer
begin

  var x: integer = 42;

  assert x == 42;

  return 0;
end
```

`var x : integer = 42;` binds `x` in `env` to `Int`(42).

10.3.3 Formally

$$\frac{\mathit{eval\_local\_decl}(\texttt{env}, \texttt{ldi1}, \langle \texttt{m} \rangle) \xrightarrow{\text{eval}} C}{\mathit{eval\_local\_decl}(\texttt{env}, \mathsf{LDI\_Typed}(\texttt{ldi1}, \_), \langle \texttt{m} \rangle) \xrightarrow{\text{eval}} C}$$

10.4 SemanticsRule.LDTuple

10.4.1 Prose

All of the following apply:

- `ldi` declares a list of local variables, LDI_Tuple(`ldis`);
- `m_init_opt` is `m`;
- `m` is a pair consisting of the native vector `v` and execution graph `g`;
- `ldis` is a list of local declaration items $\texttt{ldi}_{1..k}$;
- the value at each index of `v` is $\texttt{v}_i$, for $i = 1..k$;
- `liv` is the list of pairs $(\texttt{v}_i, \texttt{g})$, for $i = 1..k$;
- the output configuration is obtained by declare each local declaration item $\texttt{ldi}_i$ with the corresponding value (`m_init_opt` component) $(\texttt{v}_i, \texttt{g})$.

10.4.2 Example

In the specification:

```
func main () => integer
begin

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

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

  return 0;
end
```

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

10.4.3 Formally

We first define the helper semantic relation

$$\mathit{ldi\_tuple\_folder}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{local\_decl\_item}^*}^{\texttt{ldis}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{liv}}) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}})$$

via the following rules:

$$\mathit{ldi\_tuple\_folder}(\texttt{env}, [\,], [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\emptyset_{\texttt{g}}, \texttt{env})$$

$$\frac{\begin{array}{c}\texttt{ldis} \overset{\mathsf{is}}{=} [\texttt{ldi}] + \texttt{ldis}' \\ \texttt{liv} \overset{\mathsf{is}}{=} [\texttt{m}] + \texttt{liv}' \qquad \texttt{m} \overset{\mathsf{is}}{=} (\texttt{v}, \texttt{g1}) \qquad \mathit{eval\_local\_decl}(\texttt{env}, \texttt{ldi}, \langle \texttt{m} \rangle) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{g1}, \texttt{env1}) \\ \mathit{ldi\_tuple\_folder}(\texttt{env1}, \texttt{ldis}', \texttt{liv}') \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{g2}, \texttt{new\_env}) \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\mathit{ldi\_tuple\_folder}(\texttt{env}, \texttt{ldis}, \texttt{liv}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

We now use the helper rules to define the rule for local declaration item tuples:

$$\frac{\begin{array}{c}\texttt{m} \overset{\mathsf{is}}{=} (\texttt{v}, \texttt{g}) \qquad \texttt{ldis} \overset{\mathsf{is}}{=} \texttt{ldi}_{1..k} \qquad i = 1..k : \mathit{get\_index}(i, \texttt{v}) \xrightarrow{\mathsf{eval}} \texttt{v}_i \\ \texttt{liv} \overset{\mathsf{is}}{=} [i = 1..k : (\texttt{v}_i, \texttt{g})] \qquad \mathit{ldi\_tuple\_folder}(\texttt{env}, \texttt{ldis}, \texttt{liv}) \xrightarrow{\mathsf{eval}} C\end{array}}{\mathit{eval\_local\_decl}(\texttt{env}, \mathsf{LDI\_Tuple}(\texttt{ldis}), \langle \texttt{m} \rangle) \xrightarrow{\mathsf{eval}} C}$$

10.5 SemanticsRule.LDUninitialisedTyped

10.5.1 Prose

All of the following apply:

- `ldi` gives a local declaration with a type, but no initial value, LDI_Typed(`ldi1`, `t`);
- `m_init_opt` is `None`;
- the base value of `t` is $\texttt{m}^{/\!/\texttt{\#DE}}$;
- evaluating the local declaration `ldi1` with `m` as the `m_init_opt` component yields the output configuration.

10.5.2 Example

In the specification:

```
func main () => integer
begin
  var x: integer {3..42};

  assert x == 3;

  return 0;
end
```

`var x : integer{3..43};` binds `x` in `env` to the base value of `integer{3..43}`, which is `Int`(3).

10.5.3 Formally

$$\frac{\begin{array}{c}\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} \texttt{m} \;/\!/\; \texttt{\#DE} \\ \mathit{eval\_local\_decl}(\texttt{env}, \texttt{ldi1}, \langle \texttt{m} \rangle) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_local\_decl}(\texttt{env}, \mathsf{LDI\_Typed}(\texttt{ldi1}, \texttt{t}), \texttt{None}) \xrightarrow{\text{eval}} C}$$

Chapter 11

Evaluation of Statements

11.1 Syntax

11.2 Abstract Syntax

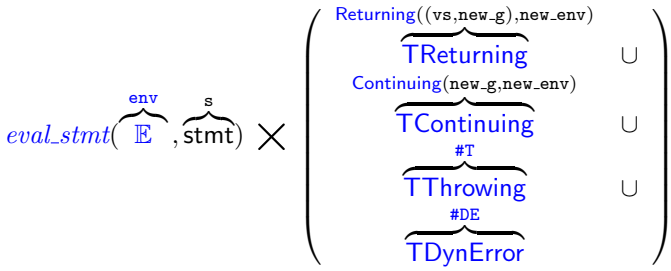
11.3 Informal preamble

Statements consist of:

- Pass statements, which do nothing (see Section 11.5),
- Assignment statements (see Section 11.6 and Section 11.7),
- Sequencing statements (see Section 11.9),
- Return statements (see Section , Section 11.8.3, Section 11.8.4 and Section 11.8.7),
- Procedure invocation statements (see Section 11.10),
- Case statements (see Section 11.12),
- Assertion statements (see Section **??**),
- Repetitive statements (see Section 11.14, Section **??**, Section 13.2),
- Throw statements, which throw exceptions (see Section **??** and Section 11.18),
- Conditional statements (see Section 11.11),
- Exception handling (see Section 11.19),
- Declarations of variables, let values and constants (see Section 11.20 and Section 11.21).

11.4 Formal preamble

The relation



evaluates a statement `s` in an environment `env`, resulting in one of four types of configurations (see more details in Section 5.2.2):

- returning configurations with values `vs`, execution graph `new_g`, and a modified environment `new_env`;
- continuing configurations with an execution graph `new_g` and modified environment `new_env`;
- throwing configurations;
- error configurations.

In evaluating a statement `s`, one of the following applies:

- SemanticsRule.SPass (see Section 11.5),
- SemanticsRule.SAssign (see Section 11.6),
- SemanticsRule.SAssignCall (see Section 11.7),
- SemanticsRule.SReturnNone (see Section 11.8.3),
- SemanticsRule.SReturnOne (see Section 11.8.4),
- SemanticsRule.SReturnSome (see Section 11.8.7),
- SemanticsRule.SSeq (see Section 11.9),
- SemanticsRule.SCall (see Section 11.10),
- SemanticsRule.SCond (see Section 11.11),
- SemanticsRule.SCase (see Section 11.12),
- SemanticsRule.SAssert (see Section 11.13),
- SemanticsRule.SWhile (see Section 11.14),

- SemanticsRule.SRepeat (see Section 11.15),
- SemanticsRule.SFor (see Section 11.16),
- SemanticsRule.SThrowNone (see Section 11.17),
- SemanticsRule.SThrowSomeTyped (see Section 11.18),
- SemanticsRule.STry (see Section 11.19),
- SemanticsRule.SDeclSome (see Section 11.20),
- SemanticsRule.SDeclNone (see Section 11.21).

11.5 SemanticsRule.SPass

11.5.1 Syntax

11.5.2 AbstractSyntax

11.5.3 Example

In the specification:

```
func main () => integer
begin

  pass;

  return 0;
end
```

`pass;` does nothing.

11.5.4 Formally

All of the following apply:

- `s` is a `pass` statement, S_Pass;
- `new_g` is the empty graph;
- `new_env` is `env`.

$$\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Pass}) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\overbrace{\emptyset_g}^{\texttt{new\_g}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

11.6 SemanticsRule.SAssign

11.6.1 Syntax

11.6.2 AbstractSyntax

11.6.3 Example

In the specification:

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

  x = 3;

  assert x == 3;

  return 0;
end
```

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

11.6.4 Formally

All of the following apply:

- `s` is an assignment statement, S_Assign(`le`, `re`);
- `re` is not a call expression;
- evaluating the expression `re` in `env` as per Chapter 6 is Normal(`m`, `env1`) (here, `m` is a pair consisting of a value and an execution graph)$^{\#T,\#DE}$;
- evaluating the left-hand-side expression `le` with `m` in `env1`, as per Chapter 7, is Normal(`new_g`, `new_env`)$^{\#T,\#DE}$.

$$\frac{\mathit{ast\_label}(\texttt{re}) \neq \text{E\_Call} \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{re}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m}, \texttt{env1}) \;//\; \#T, \#DE \qquad \mathit{eval\_lexpr}(\texttt{env1}, \texttt{le}, \texttt{m}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env}) \;//\; \#T, \#DE}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Assign}(\texttt{le}, \texttt{re})) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

11.6.5 Comments

This rule covers all assignment statements, except the ones where the right-hand side expression is a function call, which is covered by SemanticsRule.SAssignCall (see Section 11.7). Although the sequential semantics of both statements is the same, SemanticsRule.SAssignCall generates a different execution graph.

Notice that this rule first produces a value for the right-hand side expression and then completes the update via an appropriate rule for evaluating the left-hand side expression, which in turn handles variables, tuples, bitvectors, etc.

11.7 SemanticsRule.SAssignCall

11.7.1 Syntax

11.7.2 AbstractSyntax

11.7.3 Example

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

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

  (a,b) = f(1);

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

given that the function call `f(1)` returns a pair of values — Int(1) and Int(2) (each with its own associated execution graph), the statement `(a,b) = f(1)` assigns the value Int(1) to the mutable variable `a` and the value Int(2) to the mutable variable `b`.

11.7.4 Formally

All of the following apply:

- `s` assigns a left-hand-side expression list from a subprogram call, S_Assign(LE_Destructuring(`les`), E_Call(`name`, `args`, `named_args`));
- `les` is a list of left-hand-side expressions, each of which is either a variable (LE_Var(_)) or a discarded variable (LE_Discard);
- evaluating the subprogram call as per Chapter 15 is Normal(`vms`, `env1`)$^{\#T,\#DE}$;
- assigning each value in `vms` to the respective element of the tuple `les` is Normal(g2, `new_g`)$^{\#T,\#DE}$.

We first define the syntactic predicate

$$\textit{lexpr\_is\_var}(\text{lexpr}) \longrightarrow \texttt{TRUE}$$

which holds when a left-hand side expression represents a variable:

$$\mathit{lexpr\_is\_var}(\mathsf{LE\_Var}(\_)) \xrightarrow{\mathsf{eval}} \mathtt{TRUE} \qquad \mathit{lexpr\_is\_var}(\mathsf{LE\_Discard}) \xrightarrow{\mathsf{eval}} \mathtt{FALSE}$$

We now define the evaluation of assigning from a subprogram call:

$$\frac{\begin{array}{c}\mathtt{les} := \mathtt{le}_{1..k} \qquad i = 1..k : \mathit{lexpr\_is\_var}(\mathtt{le}_i) \xrightarrow{\mathsf{eval}} \mathtt{TRUE} \\ \mathit{eval\_call}(\mathtt{env}, \mathtt{name}, \mathtt{args}, \mathtt{named\_args}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{vms}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathit{multi\_assign}(\mathtt{env1}, \mathtt{les}, \mathtt{vms}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{new\_g}, \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE}\end{array}}{\begin{array}{c}\mathit{eval\_stmt}(\mathtt{env}, \mathsf{S\_Assign}(\mathsf{LE\_Destructuring}(\mathtt{les}), \mathsf{E\_Call}(\mathtt{name}, \mathtt{args}, \mathtt{named\_args}))) \\ \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\mathtt{new\_g}, \mathtt{new\_env})\end{array}}$$

11.8 Return statements

11.8.1 Syntax

11.8.2 AbstractSyntax

A return statement returns the control flow to the caller of a subprogram.

A return statement appearing in a getter or function requires a return value expression that type-satisfies the return type of the subprogram.

A return statement appearing in a setter or procedure must have no return value expression.

11.8.3 SemanticsRule.SReturnNone

Example

The specification:

```
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.

Formally

All of the following apply:

- `s` is a `return` statement, S_Return(`None`);
- `vs` is the empty list, [ ];
- `new_g` is the empty graph;
- `new_env` is `env`.

$$\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Return}(\texttt{None})) \xrightarrow{\mathsf{eval}} \mathsf{Returning}((\texttt{[ ]}, \emptyset_{\texttt{g}}), \texttt{env})$$

11.8.4 SemanticsRule.SReturnOne

11.8.5 Syntax

11.8.6 AbstractSyntax

Example

In the specification:

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

Formally

All of the following apply:

- `s` is a `return` statement;
- `s` is a `return` statement for a single expression, S_Return(⟨e⟩);
- evaluating `e` in `env` is Normal((v, g1), new_env)$^{\#T,\#DE}$;
- `vs` is [v];
- `g2` is the result of adding a Write Effect for a fresh identifier and the value `v`;
- `new_g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g1}), \texttt{new\_env}) \;\; /\!/ \;\; \#T, \#DE \qquad \texttt{wid} \in \mathbb{I} \text{ is fresh} \qquad \mathit{write\_identifier}(\texttt{wid}, \texttt{v}) \xrightarrow{\text{eval}} \texttt{g2} \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Return}(\langle \texttt{e} \rangle)) \xrightarrow{\text{eval}} \text{Returning}((\texttt{[v]}, \texttt{new\_g}), \texttt{new\_env})}$$

11.8.7 SemanticsRule.SReturnSome

11.8.8 Syntax

11.8.9 AbstractSyntax

Example

In the specification:

```
func f () => (integer, integer)
begin
  var x: integer = 0;
  for i = 0 to 5 do
    x = x + 1;
    assert x == 1; // Only the first loop 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)`.

Formally

All of the following apply:

- `s` is a `return` statement for a list of expressions, S_Return(⟨E_Tuple(es)⟩);
- evaluating each expression in `es` separately as per Section 6.23 is Normal(ms, new_env)$^{\text{//\#T,\#DE}}$;
- writing the list of values in `vms` results in (vs, new_g).

We first define the helper relation

$$\texttt{write\_folder}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms}}) \bigtimes (\overbrace{\mathbb{V}^*}^{\texttt{vs}}, \overbrace{\mathcal{G}}^{\texttt{new\_g}}) \ ,$$

which concatenates the input values in `vms` and generates an execution graph by composing the graphs in `vms` with Write Effects for the respective values.

EMPTY

$$\texttt{write\_folder}([\,]) \xrightarrow{\text{eval}} ([\,], \emptyset_{\mathrm{g}})$$

NONEMPTY

$$\frac{\texttt{vms} \overset{\text{is}}{=} [\texttt{m}] + \texttt{vms1} \qquad \texttt{m} := (\texttt{v}, \texttt{g}) \qquad \texttt{wid} \in \mathbb{I} \text{ is fresh} \qquad \textit{write\_identifier}(\texttt{wid}, \texttt{v}) \xrightarrow{\text{eval}} \texttt{g1} \qquad \texttt{write\_folder}(\texttt{vms1}, \texttt{g1}) \xrightarrow{\text{eval}} (\texttt{vs1}, \texttt{g2}) \qquad \texttt{vs} := [\texttt{v}] + \texttt{vs1} \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2}}{\texttt{write\_folder}(\texttt{vms}) \xrightarrow{\text{eval}} (\texttt{vs}, \texttt{g} \xrightarrow{\texttt{asl\_po}} \texttt{new\_g})}$$

We now use the helper relation `write_folder` to define the rule for returning a tuple of values:

$$\frac{\textit{eval\_expr\_list\_m}(\texttt{env}, \texttt{es}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{ms}, \texttt{new\_env}) \ /\!/ \ \texttt{\#T}, \texttt{\#DE} \qquad \texttt{write\_folder}(\texttt{ms}) \xrightarrow{\text{eval}} (\texttt{vs}, \texttt{new\_g})}{\textit{eval\_stmt}(\texttt{env}, \text{S\_Return}(\langle \text{E\_Tuple}(\texttt{es}) \rangle)) \xrightarrow{\text{eval}} \text{Returning}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env})}$$

11.9 SemanticsRule.SSeq

11.9.1 Syntax

11.9.2 AbstractSyntax

11.9.3 Example

In the specification:

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

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

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

  return 0;
end
```

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

11.9.4 Formally

All of the following apply:

- `s` is a *sequencing statement* `s1; s2`, that is, S_Seq(`s1`, `s2`);
- evaluating `s1` in `env` is either Continuing(`g1`, `env1`) in which case the evaluation continues, or a returning configuration $(\text{Returning}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env}))^{/\!/ \texttt{\#T},\texttt{\#DE}}$;
- evaluating `s2` in `env1` yields a non-abnormal configuration (either Normal or Continuing) $C^{/\!/ \texttt{\#T},\texttt{\#DE}}$;
- `new_g` is the ordered composition of `g1` and the execution graph of C with the `asl_po` edge;
- D is the configuration C with the execution graph component replaced with `new_g`.

$$\frac{\begin{array}{c}\mathit{eval\_stmt}(\texttt{env}, \texttt{s1}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g1}, \texttt{env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_stmt}(\texttt{env1}, \texttt{s2}) \xrightarrow{\text{eval}} C \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_po}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{new\_g})\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Seq}(\texttt{s1}, \texttt{s2})) \xrightarrow{\text{eval}} D}$$

11.10 SemanticsRule.SCall

11.10.1 Syntax

11.10.2 AbstractSyntax

A procedure invocation statement calls a procedure subprogram using the given actual arguments. The subprogram must not have a return type.

11.10.3 Example

In the specification:

```
func main () => integer
begin

  assert Zeros(3) == '000';

  return 0;
end
```

`Zeros(3)` evaluates to `'000'`.

11.10.4 Formally

All of the following apply:

- `s` is a call statement;
- `s` is a call statement, S_Call(`name`, `args`, `named_args`);
- evaluating the subprogram call as per Chapter 15 is Normal(`new_g`, `new_env`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

$$\frac{\mathit{eval\_call}(\texttt{env}, \texttt{name}, \texttt{args}, \texttt{named\_args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Call}(\texttt{name}, \texttt{args}, \texttt{named\_args})) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

11.11 SemanticsRule.SCond

11.11.1 Syntax

11.11.2 AbstractSyntax

11.11.3 Informally

Conditional statements select which block to execute by testing condition expressions sequentially until a `TRUE` condition is found.

If no `TRUE` condition is found and there is an `else` block, the `else` block is executed.

If no `TRUE` condition is found and there is no `else` block, no block is executed.

11.11.4 Examples

The specification:

```
func main () => integer
begin

  if TRUE
```

```
    then assert TRUE;
    else assert FALSE;
  end

  return 0;
end
```

does not result in any Assertion Error.

The specification:

```
func main () => integer

begin
var x:integer;
var y:integer;

  if x > y then
      return 1;
  elsif x < y then
      return -1;
  else
      return 0;
  end

end
```

The specification:

```
func main () => integer
begin
  var d:integer;
  var n:integer;

  if d IN {13,15} || n IN {13,15} then
      UNPREDICTABLE();
  end
end
```

The specification:

```
func main () => integer
begin
  var size:bits(2);
  var esize:integer;
  var elements:integer;

  if size == '01' then
    esize = 16; elements = 4;
  end

return 0;
end
```

11.11.5 Formally

Evaluation of the statement `s` in an environment `env` is the output configuration D, and all of the following apply:

- `s` is a condition statement, S_Cond(`e`, `s1`, `s2`);
- evaluating `e` in `env` is Normal((`v`, `g1`))$^{\#T,\#DE}$;
- `v` is a native Boolean for `b`;
- the statement `s'` is `s1` is `b` is `TRUE` and `s2` otherwise (so that `s1` will be evaluated if the condition evaluates to `TRUE` and otherwise `s2` will be evaluated);
- evaluating `s'` in `env1` as per Chapter 12 is a non-abnormal configuration (either Normal or Continuing) $C^{\#T,\#DE}$;
- `g` is the ordered composition of `g1` and the execution graph of the configuration C;
- D is the configuration C with the execution graph component updated to be `g`.

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g1}), \texttt{env1}) \;\; /\!/ \;\; \#T, \#DE \\ \texttt{v} \overset{\text{is}}{=} \text{Bool}(\texttt{b}) \qquad \texttt{s'} := \text{choice}(\texttt{b}, \texttt{s1}, \texttt{s2}) \qquad \mathit{eval\_block}(\texttt{env1}, \texttt{s'}) \xrightarrow{\text{eval}} C \;\; /\!/ \;\; \#T, \#DE \\ \texttt{g} := \texttt{g1} \xrightarrow{\text{asl\_ctrl}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{g}) \end{array}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Cond}(\texttt{e}, \texttt{s1}, \texttt{s2})) \xrightarrow{\text{eval}} D}$$

11.12 SemanticsRule.SCase

11.12.1 Syntax

11.12.2 AbstractSyntax

11.12.3 Example

The specification:

```
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`.

11.12.4 Formally

Evaluation of the statement `s` in an environment `env` is a configuration C, and all of the following apply:

- `s` is a case statement, S_Case(e, case_list);
- desugaring `s` gives a statement `s1`, which assigns `e` to a fresh variable and then matches its value against a list of patterns corresponding to `case_list` nested conditions. In particular, this means that the cases are considered in order and only one of them is executed;
- evaluating `s1` in `env` results in the output configuration C.

A case statement is syntactic sugar for a condition ladder where each of the alternatives is a pattern. That is, a statement of the form `if e1 then s1; else if e2 then s2; else if e3 then s3; ... else pass;`

We define the relation

$$\mathit{case\_to\_conds}(\mathsf{expr}, (\mathsf{pattern} \times \mathsf{stmt})^*) \bigtimes \mathsf{stmt}$$

which performs this AST-to-AST transformation, effectively desugaring the case statement.

VAR-EMPTY

$$\mathit{case\_to\_conds}(\mathsf{E\_Var}(\mathtt{x}), [\,]) \xrightarrow{\mathsf{eval}} \mathsf{S\_Pass}$$

VAR-NON-EMPTY

$$\frac{\mathit{case\_to\_conds}(\mathsf{E\_Var}(\mathtt{x}), \mathtt{case\_list}) \xrightarrow{\mathsf{eval}} \mathtt{case\_list}'}{\mathit{case\_to\_conds}(\mathsf{E\_Var}(\mathtt{x}), [(\mathtt{p}, \mathtt{s})] + \mathtt{case\_list}) \xrightarrow{\mathsf{eval}} \mathsf{S\_Cond}(\mathsf{E\_Pattern}(\mathsf{E\_Var}(\mathtt{x}), \mathtt{p}), \mathtt{s}, \mathtt{case\_list}')}$$

NON-VAR

$$\frac{\mathit{ast\_label}(\mathtt{e}) \neq \mathsf{E\_Var} \qquad \mathtt{y} \in \mathbb{I} \text{ is fresh} \qquad \mathtt{vardecl} \overset{\mathsf{is}}{=} \mathsf{S\_Decl}(\mathsf{LDK\_Let}, \mathsf{LDI\_Typed}(\mathsf{LDI\_Var}(\mathtt{y}), \mathsf{T\_Int}(\mathsf{Unconstrained}))) \qquad \mathit{case\_to\_conds}(\mathsf{E\_Var}(\mathtt{y}), \mathtt{case\_list}) \xrightarrow{\mathsf{eval}} \mathtt{case\_cond}}{\mathit{case\_to\_conds}(\mathtt{e}, \mathtt{case\_list}) \xrightarrow{\mathsf{eval}} \mathsf{S\_Seq}(\mathtt{vardecl}, \mathtt{case\_cond})}$$

We now define the semantics of a case statement in terms of the desugared statement:

$$\frac{\mathit{case\_to\_conds}(\mathtt{e}, \mathtt{case\_list}) \xrightarrow{\mathsf{eval}} \mathtt{s1} \qquad \mathit{eval\_stmt}(\mathtt{env}, \mathtt{s1}) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_stmt}(\mathtt{env}, \mathsf{S\_Case}(\mathtt{e}, \mathtt{case\_list})) \xrightarrow{\mathsf{eval}} C}$$

11.13 SemanticsRule.SAssert

An assertion statement takes an expression that is asserted by the specification to be `TRUE` when the assertion statement is executed.

If an assertion expression is FALSE when the assertion statement is executed, it is a dynamic error. An implementation may throw an implementation-defined exception in this case, but is not required to.

The expression in an assertion statement must be side-effect-free.

11.13.1 Prose

All of the following apply:

- s is an assertion statement, S_Assert(e);
- one of the following holds:
  - all of the following hold (OKAY):
    - evaluating e in env is Normal((v, new_g), new_env)$^{\#T,\#DE}$;
    - v is a native Boolean value for TRUE;
    - the resulting configuration is Continuing(new_g, new_env).
  - all of the following hold (ERROR):
    - evaluating e in env is Normal((v, new_g), new_env);
    - v is a native Boolean value for FALSE;
    - an AssertionFailed error is returned.

11.13.2 Example

In the specification:

```
func main () => integer
begin

  assert (42 != 3);

  return 0;
end
```

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

11.13.3 Example

In the specification:

```
func main () => integer
begin

  assert (42 == 3);

  return 0;
end
```

`assert (42 == 3);` results in an `AssertionFailed` error.

11.13.4 Formally

OKAY

$$\frac{\mathit{eval\_expr}(\mathtt{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathsf{v}, \mathtt{new\_g}), \mathtt{new\_env}) \;\; /\!/ \;\; \mathtt{\#T}, \mathtt{\#DE} \qquad \mathsf{v} \overset{\mathsf{is}}{=} \mathsf{Bool}(\mathtt{TRUE})}{\mathit{eval\_stmt}(\mathtt{env}, \mathsf{S\_Assert}(\mathsf{e})) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\mathtt{new\_g}, \mathtt{new\_env})}$$

ERROR

$$\frac{\mathit{eval\_expr}(\mathtt{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathsf{v}, \_), \_) \qquad \mathsf{v} \overset{\mathsf{is}}{=} \mathsf{Bool}(\mathtt{FALSE})}{\mathit{eval\_stmt}(\mathtt{env}, \mathsf{S\_Assert}(\mathsf{e})) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{"ERROR[AssertionFailed]"})}$$

11.14 SemanticsRule.SWhile

11.14.1 Prose

Evaluation of the statement `s` in an environment `env` is the output configuration C and all of the following apply:

- `s` is a `while` statement, S_While(e, _, body);
- evaluating the loop as per Section 13.1 in an environment `env`, with the arguments `TRUE` (which conveys that this is a `while` statement), `e`, and `body` results in C.

11.14.2 Example

The specification:

```
func main () => integer
begin

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

  return 0;
end
```

prints `0123`.

11.14.3 Formally

$$\frac{\mathit{eval\_loop}(\mathtt{env}, \mathtt{TRUE}, \mathsf{e}, \mathsf{body}) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_stmt}(\mathtt{env}, \overbrace{\mathsf{S\_While}(\mathsf{e}, \_, \mathsf{body})}^{\mathsf{s}}) \xrightarrow{\mathsf{eval}} C}$$

11.15 SemanticsRule.SRepeat

11.15.1 Prose

Evaluation of the statement `s` in an environment `env` is either Returning((`vs`, `new_g`), `new_env`) or an output configuration D and all of the following apply:

- `s` is a `repeat` statement, S_Repeat(`e`, `body`, _);
- evaluating `body` in `env` as per Chapter 12 yields Continuing(`g1`, `env1`)$^{\#R,\#T,\#DE}$;
- evaluating the loop as per Section 13.1 in an environment `env1`, with the arguments `FALSE` (which conveys that this is a `repeat` statement), `e`, and `body` results in C;
- `g2` is the ordered composition of `g1` and `g2` with the `asl_po` edge;
- the output configuration D is the output configuration C with its execution graph substituted with `g2`.

11.15.2 Example

The specification:

```
func main () => integer
begin

  var i: integer = 0;
  repeat
    assert i <= 3;
    print(i);
    i = i + 1;
  until i > 3;

  return 0;
end
```

prints

```
0
1
2
3
```

11.15.3 Formally

$$\frac{\begin{array}{c}\mathit{eval\_block}(\texttt{env}, \texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g1}, \texttt{env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_loop}(\texttt{env1}, \texttt{FALSE}, \texttt{e}, \texttt{body}) \xrightarrow{\text{eval}} C \\ \texttt{g2} := \texttt{g1} \xrightarrow{\text{asl\_po}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{g2})\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \overbrace{\text{S\_Repeat}(\texttt{e}, \texttt{body}, \_)}^{\texttt{s}}) \xrightarrow{\text{eval}} D}$$

11.16 SemanticsRule.SFor

Evaluating a `for` statement involves introducing an index variable to the environment. The type system ensures, via TypingRule.SFor, that the index variable is not already declared in the scope of the subprogram containing the `for` statement.

11.16.1 Prose

All of the following apply:

- `s` is a `for` statement, S_For $\left\{ \begin{array}{rcl} \text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{direction} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \_ \end{array} \right\}$;
- evaluating the side-effect-free expression `e1` in `env` is either Normal(`v1`, `g1`)$^{\#DE}$;
- evaluating the side-effect-free expression `e2` in `env` is either Normal(`v2`, `g2`)$^{\#DE}$;
- declaring the local identifier `index_name` in `env` with value `v1` is (`g3`, `env1`);
- evaluating the `for` loop with arguments (`index_name`, `e1`, `dir`, `e2`, `s`) in `env1`, as per Section 13.2 is Normal(`g4`, `env2`)$^{\#T,\#DE}$;
- removing the local `index_name` from `env2` is `env3`;
- `new_g` is formed as follows: taking the parallel composition of `g1` and `g2`, then taking the ordered composition of the result with the `asl_data` edge, and finally taking the ordered composition of the result with the `asl_po` edges;
- `new_env` is `env3`.
- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

11.16.2 Example

The specification:

```
func main () => integer
begin

  for i = 0 to 3 do
    assert i <= 3;
    print(i);
  end

  return 0;
end
```

prints

```
0
1
2
3
```

11.16.3 Formally

Recall that the expressions for the `for` loop range are side-effect-free, which is why they are evaluated via the rule for evaluating side-effect-free expressions.

$$\frac{\begin{array}{c}\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{start\_e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{start\_v}, \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{end\_e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{end\_v}, \texttt{g2}) \;/\!/\; \texttt{\#DE} \\ \mathit{declare\_local\_identifier}(\texttt{env}, \texttt{index\_name}, \texttt{end\_v}) \xrightarrow{\text{eval}} (\texttt{g3}, \texttt{env1}) \\ \mathit{eval\_for}(\texttt{env1}, \texttt{index\_name}, \texttt{start\_v}, \texttt{dir}, \texttt{end\_v}, \texttt{body}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{g4}, \texttt{env2}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{remove\_local}(\texttt{env2}, \texttt{index\_name}) \xrightarrow{\text{eval}} \texttt{env3} \\ \texttt{new\_g} := (\texttt{g1} \parallel \texttt{g2}) \xrightarrow{\text{asl\_data}} \texttt{g3} \xrightarrow{\text{asl\_po}} \texttt{g4} \qquad \texttt{new\_env} := \texttt{env3}\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_For}\overbrace{\left\{\begin{array}{rcl}\text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{direction} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \_\end{array}\right\}}^{\texttt{s}}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

11.17 SemanticsRule.SThrowNone

11.17.1 Prose

All of the following apply:

- `s` is a `throw` statement that does not provide an expression, S_Throw(None);
- `new_env` is `env`;
- `ex` is None;
- `new_g` is the empty graph;
- an exception is thrown with `new_env`.

11.17.2 Example

The specification:

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

func main () => integer
begin

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

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

  return 0;
end
```

throws a `MyException` exception.

11.17.3 Formally

$$\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Throw}(\texttt{None})) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}((\texttt{None}, \emptyset_{\mathsf{g}}), \texttt{env})$$

11.18 SemanticsRule.SThrowSomeTyped

11.18.1 Prose

All of the following apply:

- `s` is a `throw` statement that provides an expression and a type, $\mathsf{S\_Throw}(\langle(\texttt{e}, \texttt{t})\rangle)$;
- evaluating `e` in `env` is $\mathsf{Normal}((\texttt{v}, \texttt{g1}), \texttt{new\_env})^{\mathit{//}\#\mathsf{T},\#\mathsf{DE}}$;
- `name` is a fresh identifier (which conceptually holds the exception value);
- `g2` is a Write Effect to `name`;
- `new_g` is the ordered composition of `g1` and `g2` with the `asl_data` edge;

- `ex` consists of the exception value `v`, the name of the variable holding it — `name`, and the type annotation for the exception — `t`;
- the result of the entire evaluation is Throwing((ex, new_g), env).

11.18.2 Example

The specification:

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

func main () => integer
begin

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

  return 0;
end
```

terminates successfully. That is, no dynamic error occurs.

11.18.3 Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\mathtt{env}, \mathtt{e}) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v}, \mathtt{g1}), \mathtt{new\_env}) \;\; /\!/ \;\; \#\text{T}, \#\text{DE} \\ \mathtt{name} \in \mathbb{I} \text{ is fresh} \qquad \mathtt{g2} := \text{WriteEffect}(\mathtt{name}) \\ \mathtt{new\_g} := \mathtt{g1} \xrightarrow{\text{asl\_data}} \mathtt{g2} \qquad \mathtt{ex} := \langle(\text{value\_read\_from}(\mathtt{v}, \mathtt{name}), \mathtt{t})\rangle \end{array}}{\mathit{eval\_stmt}(\mathtt{env}, \text{S\_Throw}(\langle(\mathtt{e}, \mathtt{t})\rangle)) \xrightarrow{\text{eval}} \text{Throwing}((\mathtt{ex}, \mathtt{new\_g}), \mathtt{new\_env})}$$

11.19 SemanticsRule.STry

11.19.1 Prose

All of the following apply:

- `s` is a `try` statement, S_Try(s, catchers, otherwise_opt);
- evaluating `s1` in env as per Chapter 12 is a non-abnormal (that is, either Normal or Continuing) configuration $\mathtt{s\_m}^{/\!/\#\text{T},\#\text{DE}}$;
- evaluating (catchers, otherwise_opt, s_m) as per Chapter 14 is C, which is the result of the entire evaluation.

11.19.2 Example

The specification:

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

func main () => integer
begin

  try
    throw MyExceptionType { a = 42 };

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

  return 0;
end
```

does not result in any Assertion error, and the specification terminates with the exit code 0.

11.19.3 Formally

$$\frac{\mathit{eval\_block}(\texttt{env}, \texttt{s1}) \xrightarrow{\text{eval}} \texttt{s\_m} \;/\!/\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\text{eval}} C}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Try}(\texttt{s1}, \texttt{catchers}, \texttt{otherwise\_opt})) \xrightarrow{\text{eval}} C}$$

11.20 SemanticsRule.SDeclSome

11.20.1 Prose

All of the following apply:

- `s` is a declaration with an initial value, S_Decl(ldk, `ldi`, ⟨`e`⟩);
- evaluating `e` in env is Normal(`m`, `env1`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- evaluating the local declaration `ldi` with ⟨`m`⟩ as the initializing value in `env1` as per Chapter 10 is Normal(`new_g`, `new_env`);
- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

11.20.2 Example

The specification:

```
func main () => integer
begin

  let x = 3;

  assert x == 3;

  return 0;
end
```

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

11.20.3 Formally

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_local\_decl}(\texttt{env1}, \texttt{ldi}, \langle \texttt{m} \rangle) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Decl}(\_, \texttt{ldi}, \langle \texttt{e} \rangle)) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

11.21 SemanticsRule.SDeclNone

11.21.1 Prose

All of the following apply:

- `s` is a declaration without an initial value, S_Decl(_, `ldi`, `None`);
- evaluating the local declaration (`ldi`, `None`) as per Chapter 10 is Normal(`new_g`, `new_env`);
- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

11.21.2 Example

In the specification:

```
func main () => integer
begin

  var x: integer;

  assert x == 0;

  return 0;
end
```

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

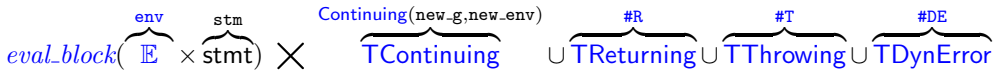
11.21.3 Formally

$$\frac{\mathit{eval\_local\_decl}(\texttt{env}, \texttt{s}, \texttt{ldi}, \texttt{None}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}{\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Decl}(\_, \texttt{ldi}, \texttt{None})) \xrightarrow{\text{eval}} \mathsf{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

Chapter 12

Evaluation of Blocks

The relation



evaluates a statement `stm` as a *block*. That is, `stm` is evaluated in a fresh local environment, which drops back to the original local environment of `env` when the evaluation terminates.

12.1 SemanticsRule.Block

12.1.1 Prose

All of the following apply:

- `block_env` is the environment `env` modified by replacing the local component (of the inner dynamic environment) by an empty one;
- evaluating `stm` in `block_env`, as per Chapter 11, is Continuing(`new_g`, `block_env1`)$^{/\!/\texttt{\#R}}$;
- `new_env` is formed from `block_env1` after restoring the variable bindings of `env` with the updated values of `block_env`. The effect is that of discarding the bindings for variables declared inside `stm`;
- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

12.1.2 Example

In the specification:

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

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

  return 0;
end
```

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

12.1.3 Formally

$$\frac{\begin{array}{c}\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \texttt{block\_env} := (\text{tenv}, (G^{\text{denv}}, \emptyset_\lambda)) \\ \textit{eval\_stmt}(\texttt{block\_env}, \texttt{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{block\_env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \texttt{block\_env1} \overset{\text{is}}{=} (\text{tenv}, \text{denv1}) \qquad \texttt{new\_env} := (\text{tenv}, (G^{\text{denv1}}, L^{\text{denv1}}|_{\text{dom}(L^{\text{denv}})}))\end{array}}{\textit{eval\_block}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

That is, evaluating a block discards the bindings for variables declared inside `stm`.

Chapter 13

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 11.14 and the semantic rule *SRepeat* at Section 11.15. The semantic rule *For* is only used by the semantic rule *SFor* at Section 11.16.

13.1 SemanticsRule.Loop

The relation

$$\mathit{eval\_loop}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{B}}^{\texttt{is\_while}}, \overbrace{\mathsf{expr}}^{\texttt{e\_cond}}, \overbrace{\mathsf{stmt}}^{\texttt{body}}) \times \begin{pmatrix} \mathsf{Continuing}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) & \cup \\ \overbrace{\mathsf{TReturning}}^{\#R} & \cup \\ \overbrace{\mathsf{TThrowing}}^{\#T} & \cup \\ \overbrace{\mathsf{TDynError}}^{\#DE} & \end{pmatrix}$$

evaluates `body` in `env` as long as `e_cond` holds when `is_while` is `TRUE` or until `e_cond` holds when `is_while` is `FALSE`. The result is either the continuing configuration Continuing(`new_g`, `new_env`), an early return configuration, or an abnormal configuration.

13.1.1 Prose

One of the following applies:

- all of the following apply (EXIT):
  - evaluating `e_cond` in `env` is Normal(`cond_m`, `new_env`)$^{\#T,\#DE}$;
  - `cond_m` consists of a native Boolean for `b` and an execution graph `new_g`;
  - `b` is not equal to `is_while`;

    * the result of the entire evaluation is Continuing(`new_g`, `new_env`) and the loop is exited.

- all of the following apply (CONTINUE):

    * evaluating `e_cond` in `env` is Normal(`cond_m`, `env1`);
    * `m_cond` consists of a native Boolean for `b` and an execution graph `g1`;
    * `b` is equal to `is_while`;
    * evaluating `body` in `env1` as per Chapter 12 is either Continuing(`g2`, `env2`)$^{\#R,\#T,\#DE}$;
    * evaluating (`is_while`, `e_cond`, `body`) in `env2` as a loop is Continuing(`g3`, `new_env`)$^{\#R,\#T,\#DE}$;
    * `new_g` is the ordered composition of `g1` and `g2` with the `asl_ctrl` label and then the ordered composition of the result and `g3` with the `asl_po` edge;
    * the result of the entire evaluation is Continuing(`new_g`, `new_env`).

13.1.2 Example

The specification:

```
func main () => integer
begin

  var i: integer = 0;

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

  return 0;
end
```

does not result in any Assertion Error and the specification terminates with exit code 0.

13.1.3 Formally

The premise `b` $\neq$ `is_while` is `TRUE` in the case of a `while` loop and the loop condition `e` not holding, which is exactly when we want the loop to exit. The opposite holds for a `repeat` loop. The negation of the condition is used to decide whether to continue the loop iteration.

EXIT

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{cond\_m}, \texttt{new\_env}) \;/\!/\; \#\text{T}, \#\text{DE} \qquad \texttt{cond\_m} \overset{\text{is}}{=} (\text{Bool}(\texttt{b}), \texttt{new\_g}) \qquad \texttt{b} \neq \texttt{is\_while}}{\mathit{eval\_loop}(\texttt{env}, \texttt{is\_while}, \texttt{e\_cond}, \texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

CONTINUE

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{cond\_m}, \texttt{env1}) \qquad \texttt{cond\_m} \overset{\text{is}}{=} (\text{Bool}(\texttt{b}), \texttt{g1}) \\ \texttt{b} = \texttt{is\_while} \qquad \mathit{eval\_block}(\texttt{env1}, \texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g2}, \texttt{env2}) \;/\!/\; \#\text{R}, \#\text{T}, \#\text{DE} \\ \mathit{eval\_loop}(\texttt{env2}, \texttt{is\_while}, \texttt{e\_cond}, \texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g3}, \texttt{new\_env}) \;/\!/\; \#\text{R}, \#\text{T}, \#\text{DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_ctrl}} \texttt{g2} \xrightarrow{\texttt{asl\_po}} \texttt{g3}\end{array}}{\mathit{eval\_loop}(\texttt{env}, \texttt{is\_while}, \texttt{e\_cond}, \texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

13.2 SemanticsRule.For

The relation

$$\mathit{eval\_for}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{index\_name}}, \overbrace{\mathcal{Z}}^{\texttt{v\_start}}, \overbrace{\{\texttt{Up}, \texttt{Down}\}}^{\texttt{dir}}, \overbrace{\mathcal{Z}}^{\texttt{v\_end}}, \overbrace{\texttt{stmt}}^{\texttt{body}}) \bigtimes \left(\begin{array}{ll} \overbrace{\text{TReturning}}^{\#\text{R}} & \cup \\ \overbrace{\text{TContinuing}}^{\#\text{C}} & \cup \\ \overbrace{\text{TThrowing}}^{\#\text{T}} & \cup \\ \overbrace{\text{TDynError}}^{\#\text{DE}} & \end{array} \right)$$

evaluates the `for` loop with the index variable `index_name` starting from the value `v_start` going in the direction given by `dir` until the value given by `v_end`, executing `body` on each iteration. The evaluation utilizes two helper relations: `eval_for_step` and `eval_for_loop`.

The helper relation

$$\texttt{eval\_for\_step}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{index\_name}}, \overbrace{\mathcal{Z}}^{\texttt{v\_start}}, \overbrace{\{\texttt{Up}, \texttt{Down}\}}^{\texttt{dir}}) \bigtimes ((\overbrace{\mathcal{Z}}^{\texttt{v\_step}} \times \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

either increments or decrements the index variable, returning the new value of the index variable, the modified environment, and the resulting execution graph.

The helper relation

$$\texttt{eval\_for\_loop}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{index\_name}}, \overbrace{\mathcal{Z}}^{\texttt{v\_start}}, \overbrace{\{\texttt{Up}, \texttt{Down}\}}^{\texttt{dir}}, \overbrace{\mathcal{Z}}^{\texttt{v\_end}}, \overbrace{\texttt{stmt}}^{\texttt{body}}) \times \begin{pmatrix} \overbrace{\mathsf{TContinuing}}^{\mathsf{Continuing}(\texttt{new\_g},\texttt{new\_env})} & \cup \\ \overbrace{\mathsf{TReturning}}^{\#\mathsf{R}} & \cup \\ \overbrace{\mathsf{TThrowing}}^{\#\mathsf{T}} & \cup \\ \overbrace{\mathsf{TDynError}}^{\#\mathsf{DE}} & \end{pmatrix}$$

executes one iteration of the loop body and then uses `eval_for` to execute the remaining iterations.

13.2.1 Prose

Stepping the Index Variable

All of the following apply:

- `op_for_dir` is either `PLUS` when `dir` is `Up` or `MINUS` when `dir` is `Down`;
- reading `v_start` into the identifier `index_name` gives `g1`;
- applying the binary operator `op_for_dir` to `v_start` and the native integer for 1 is `v_step`;
- the execution graph for writing `v_step` into the identifier `index_name` gives `g2`;
- updating the local component of the dynamic environment of `env` by binding `index_name` to `v_step` gives `new_env`;
- `new_g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

Running the Loop Body

All of the following apply:

- evaluating `body` as a block statement (see Chapter 12) in `env` is $\mathsf{Continuing}(\texttt{g1}, \texttt{env1})^{/\!/\#\mathsf{R},\#\mathsf{T},\#\mathsf{DE}}$;
- stepping the index `index_name` with `v_start` and the direction `dir` in `env1`, that is, $\texttt{eval\_for\_step}(\texttt{env1}, \texttt{index\_name}, \texttt{v\_start}, \texttt{dir})$ gives $((\texttt{v\_step}, \texttt{env2}), \texttt{g2})$;
- evaluating the `for` loop with $(\texttt{index\_name}, \texttt{v\_step}, \texttt{dir}, \texttt{v\_end}, \texttt{body})$ in `env2` results in a continuing configuration $\mathsf{Continuing}(\texttt{g3}, \texttt{new\_env})^{/\!/\#\mathsf{R},\#\mathsf{T},\#\mathsf{DE}}$;
- `new_g` is the ordered composition of `g1`, `g2`, and `g3` with the `asl_po` edge.

Overall Evaluation

Evaluating (index_name, v_start, dir, v_end, body) in env is either a continuing configuration Continuing(new_g, new_env) or a returning configuration (in case the body of the loop results in an early return) or an abnormal configuration, and All of the following apply:

- comp_for_dir is either LT when dir is Up or GT when dir is Down;
- reading v_start into the identifier index_name gives g1;
- One of the following applies:
  - All of the following apply (RETURN):
    - using comp_for_dir to compare v_end to v_start gives the native Boolean for TRUE;
    - new_g is g1;
    - new_env is env;
    - the result of the entire evaluation is Continuing(new_g, new_env).
  - All of the following apply (CONTINUE):
    - using comp_for_dir to compare v_end to v_start gives the native Boolean for FALSE;
    - evaluating the loop body via eval_for_loop with (index_name, v_start, dir, v_end, body) in env is Continuing(g2, new_env)$^{//\#R,\#T,\#DE}$;
    - new_g is the ordered composition of g1 and g2 with the asl_ctrl label.

13.2.2 Example

The specification:

```
func main () => integer
begin

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

  return 0;
end
```

does not result in any assertion error, and the specification terminates with exit-code 0.

13.2.3 Formally

Advancing the loop counter one step towards the end of its range is achieved via the following rule:

$$\frac{\begin{array}{c}\texttt{op\_for\_dir} := \mathsf{choice}(\texttt{dir} = \texttt{Up}, \texttt{PLUS}, \texttt{MINUS}) \\ \mathit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \texttt{g1} \\ \mathit{binop}(\texttt{op\_for\_dir}, \texttt{v\_start}, \mathsf{Int}(1)) \xrightarrow{\mathsf{eval}} \texttt{v\_step} \\ \mathit{write\_identifier}(\texttt{index}, \texttt{v\_step}) \xrightarrow{\mathsf{eval}} \texttt{g2} \qquad \texttt{env} \overset{\mathsf{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \\ \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}[\texttt{index\_name} \mapsto \texttt{v\_step}])) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2}\end{array}}{\texttt{eval\_for\_step}(\texttt{env}, \texttt{index\_name}, \texttt{v\_start}, \texttt{dir}) \xrightarrow{\mathsf{eval}} ((\texttt{v\_step}, \texttt{new\_env}), \texttt{new\_g})}$$

Running the loop body is achieved via the following rule:

$$\frac{\begin{array}{c}\mathit{eval\_block}(\texttt{env}, \texttt{body}) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{g1}, \texttt{env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \texttt{eval\_for\_step}(\texttt{env1}, \texttt{index\_name}, \texttt{v\_start}, \texttt{dir}) \xrightarrow{\mathsf{eval}} ((\texttt{v\_step}, \texttt{env2}), \texttt{g2}) \\ \mathit{eval\_for}(\texttt{env2}, \texttt{index\_name}, \texttt{v\_step}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{g3}, \texttt{new\_env}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_po}} \texttt{g2} \xrightarrow{\texttt{asl\_po}} \texttt{g3}\end{array}}{\texttt{eval\_for\_loop}(\texttt{env}, \texttt{index\_name}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

Finally, the rules for evaluating a `for` loop utilize both `eval_for_step` and `eval_for_loop` (the latter in a mutually recursive manner):

RETURN

$$\frac{\begin{array}{c}\texttt{comp\_for\_dir} := \mathsf{choice}(\texttt{dir} = \texttt{Up}, \texttt{LT}, \texttt{GT}) \\ \mathit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \texttt{g1} \\ \mathit{binop}(\texttt{comp\_for\_dir}, \texttt{v\_end}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \mathsf{Bool}(\texttt{TRUE}) \\ \texttt{new\_g} := \texttt{g1} \qquad \texttt{new\_env} = \texttt{env}\end{array}}{\mathit{eval\_for}(\texttt{env}, \texttt{index\_name}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

CONTINUE

$$\frac{\begin{array}{c}\texttt{comp\_for\_dir} := \mathsf{choice}(\texttt{dir} = \texttt{Up}, \texttt{LT}, \texttt{GT}) \\ \mathit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \texttt{g1} \\ \mathit{binop}(\texttt{comp\_for\_dir}, \texttt{v\_end}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \mathsf{Int}(\texttt{FALSE}) \\ \texttt{eval\_for\_loop}(\texttt{env}, \texttt{index\_name}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\mathsf{eval}} \\ \mathsf{Continuing}(\texttt{g2}, \texttt{new\_env}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_ctrl}} \texttt{g2}\end{array}}{\mathit{eval\_for}(\texttt{env}, \texttt{index\_name}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

Chapter 14

Evaluation of Catchers

The semantic relation for evaluating catchers employs an argument that is an output configuration. This argument corresponds to the result of evaluating a `try` statement and its type is defined as follows:

$$\mathsf{TOutConfig} \triangleq \mathsf{TNormal} \cup \mathsf{TThrowing} \cup \mathsf{TContinuing} \cup \mathsf{TReturning} \ .$$

The relation

$$\mathit{eval\_catchers}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{catcher}^*}^{\texttt{catchers}}, \overbrace{\langle \mathsf{stmt} \rangle}^{\texttt{otherwise\_opt}}, \overbrace{\mathsf{TOutConfig}}^{\texttt{s\_m}}) \times \begin{pmatrix} \mathsf{TReturning} & \cup \\ \mathsf{TContinuing} & \cup \\ \mathsf{TThrowing} & \cup \\ \mathsf{TDynError} & \end{pmatrix}$$

evaluates a list of `catch` clauses `catchers`, an `otherwise` clause, and a configuration `s_m` resulting from the evaluation of the throwing expression, in the environment `env`. The result is either a continuation configuration, an early return configuration, or an abnormal configuration.

When the statement in a `try` block, which we will refer to as the try-block statement, is evaluated, it may call a function that updates the global environment. If evaluation of the `try` block raises an exception that is caught, either by a `catch` clause or an `otherwise` clause, the statement associated with that clause, which we will refer to as the clause statement, is evaluated. It is important to evaluate the clause statement in an environment that includes any updates to the global environment made by evaluating the try-block statement. We demonstrate this with the following example.

Consider the following specification:

```
type MyExceptionType of exception{};
var g : integer = 0;

func update_and_throw()
begin
    var x = 5;
```

```
    g = 1;
    throw MyExceptionType{};
end

func main() => integer
begin
    var x = 2;
     try
       update_and_throw();
     catch
       when MyExceptionType =>
         print(x, g);
     end
  return 0;
end
```

Here, the try-block statement consists of the single statement `update_and_throw();`. Evaluating the call to `update_and_throw` employs an environment `env` where `g` is bound to 0. Notice that the call to `update_and_throw` binds `g` to 1 before raising an exception. Therefore, evaluating the call to `update_and_throw` returns a configuration of the form Throwing(_, `env_throw`) where `env_throw` binds `g` to 1. When the catch clause is evaluated the semantics takes the global environment from `env_throw` to account for the update to `g` and the local environment from `env` to account for the updates to the local environment in `main`, which binds `x` to 2, and use this environment to evaluate `print(x, g)`, resulting in the output `2 1`.

One of the following applies:

- SemanticsRule.Catch (see Section 14.1),
- SemanticsRule.CatchNamed (see Section 14.2),
- SemanticsRule.CatchOtherwise (see Section 14.3),
- SemanticsRule.CatchNone (see Section 14.4),
- SemanticsRule.CatchNoThrow (see Section 14.5).

We also define two helper relations:

- SemanticsRule.FindCatcher (see Section 14.6),
- SemanticsRule.RethrowImplicit (see Section 14.7).

14.1 SemanticsRule.Catch

14.1.1 Prose

All of the following apply:

- `s_m` is Throwing((⟨value_read_from(`v`, `e_id`), `v_ty`⟩, `s_g`), `env_throw`);

- env consists of the static environment tenv and dynamic environment denv;
- env_throw consists of the static environment tenv and dynamic environment denv_throw;
- env1 is defined by taking the static environment tenv, the global component of the dynamic environment from denv_throw and the local component of the dynamic environment from denv;
- finding the first catcher with the static environment tenv, the exception type v_ty, and the list of catchers catchers gives a catcher that does not declare a name (None) and gives a statement s;
- evaluating s in env1 as a block (Chapter 12) is not an error configuration $C^{\#\mathrm{DE}}$;
- editing potential implicit throwing configurations via *rethrow_implicit*(v, v_ty, C) gives the configuration D;
- new_g is the ordered composition of s_g and the graph of D;
- the result of the entire evaluation is D with its graph substituted with new_g.

14.1.2 Example

The specification:

```
type MyExceptionType of exception{};

func main () => integer
begin

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

  return 0;
end
```

terminates successfully. That is, no dynamic error occurs.

14.1.3 Formally

$$\frac{\begin{array}{c}\texttt{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \\ \mathsf{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \\ \texttt{env\_throw} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \qquad \texttt{env1} := (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\mathsf{denv}})) \\ \mathit{find\_catcher}(\mathsf{tenv}, \texttt{v\_ty}, \texttt{catchers}) \overset{\text{is}}{=} \langle(\mathtt{None}, \_, \texttt{s})\rangle \qquad \mathit{eval\_block}(\texttt{env1}, \texttt{s}) \xrightarrow{\mathsf{eval}} C \;/\!/\; \mathtt{\#DE} \\ D := \mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, C) \qquad \texttt{new\_g} := \texttt{s\_g} \xrightarrow{\mathtt{asl\_po}} \mathit{graph}(D)\end{array}}{\mathit{eval\_catchers}(\mathsf{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\mathsf{eval}} D(\mathit{graph} \mapsto \texttt{new\_g})}$$

14.2 SemanticsRule.CatchNamed

14.2.1 Prose

All of the following apply:

- `s_m` is $\mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw})$;
- env consists of the static environment tenv and dynamic environment denv;
- `env_throw` consists of the static environment tenv and dynamic environment `denv_throw`;
- `env1` is defined by taking the static environment tenv, the global component of the dynamic environment from `denv_throw` and the local component of the dynamic environment from denv;
- finding the first catcher with the static environment tenv, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives a statement `s`;
- `g1` is the execution graph resulting from reading `v` into the identifier `e_id`;
- declaring a local identifier `name` with (`e1`, `g1`) in `env1` gives (`env2`, `g2`);
- evaluating `s` in `env2` as a block (Chapter 12) is not an error configuration $C^{/\!/\mathtt{\#DE}}$;
- `env3` is the environment of the configuration C;
- removing the binding for `name` from the local component of the dynamic environment in `env3` gives `env4`;
- substituting the environment of C with `env4` gives D;
- editing potential implicit throwing configurations via $\mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, D)$ gives the configuration E;
- `new_g` is the ordered composition of `s_g`, `g1`, `g2`, and the graph of E, with the `asl_po` edges;
- the result of the entire evaluation is E with its graph substituted with `new_g`.

14.2.2 Example

The specification:

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

func main () => integer
begin

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

  return 0;
end
```

prints `My exception with my message`.

14.2.3 Formally

$$\texttt{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw})$$
$$\mathsf{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}))$$
$$\texttt{env\_throw} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \qquad \texttt{env1} := (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\mathsf{denv}}))$$
$$\mathit{find\_catcher}(\mathsf{tenv}, \texttt{v\_ty}, \texttt{catchers}) \overset{\text{is}}{=} \langle(\langle\texttt{name}\rangle, \_, \texttt{s})\rangle \qquad \texttt{g1} := \mathit{read\_identifier}(\texttt{e\_id}, \texttt{v})$$
$$\mathit{declare\_local\_identifier\_m}(\texttt{env1}, \texttt{name}, (\texttt{e1}, \texttt{g1})) \xrightarrow{\text{eval}} (\texttt{env2}, \texttt{g2})$$
$$\mathit{eval\_block}(\texttt{env2}, \texttt{s}) \xrightarrow{\text{eval}} C \;\; /\!/ \;\; \texttt{\#DE}$$
$$\texttt{env3} := \mathit{environ}(C)$$
$$\mathit{remove\_local}(\texttt{env3}, \texttt{name}) \xrightarrow{\text{eval}} \texttt{env4} \qquad D := C(\mathit{environ} \mapsto \texttt{env4})$$
$$E := \mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, D) \qquad \texttt{new\_g} := \texttt{s\_g} \xrightarrow{\text{asl\_po}} \texttt{g1} \xrightarrow{\text{asl\_po}} \texttt{g2} \xrightarrow{\text{asl\_po}} \mathit{graph}(E)$$
$$\overline{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\text{eval}} E(\mathit{graph} \mapsto \texttt{new\_g})}$$

14.3 SemanticsRule.CatchOtherwise

14.3.1 Prose

All of the following apply:

- `s_m` is $\mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw})$;
- `env` consists of the static environment tenv and dynamic environment denv;

- `env_throw` consists of the static environment `tenv` and dynamic environment `denv_throw`;
- `env1` is defined by taking the static environment `tenv`, the global component of the dynamic environment from `denv_throw` and the local component of the dynamic environment from `denv`;
- finding the first catcher with the static environment `tenv`, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives `None` (that is, neither of the `catch` clauses matches the raised exception);
- evaluating the `otherwise` statement `s` in `env2` as a block (Chapter 12) is not an error configuration $C^{\#\texttt{DE}}$;
- editing potential implicit throwing configurations via *rethrow_implicit*$(\texttt{v}, \texttt{v\_ty}, C)$ gives the configuration D;
- `new_g` is the ordered composition of `s_g` and the graph of D, with the `asl_po` edge;
- the result of the entire evaluation is D with its graph substituted with `new_g`.

14.3.2 Example

The specification:

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

func main () => integer
begin

     try
       throw MyExceptionType1 {};
       assert FALSE;
     catch
       when MyExceptionType2 =>
         assert FALSE;
       otherwise =>
         print("Otherwise");
     end

  return 0;
end
```

prints `Otherwise`.

14.3.3 Formally

$$\frac{\begin{array}{c}\texttt{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \\ \mathsf{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \\ \texttt{env\_throw} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \qquad \texttt{env1} := (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\mathsf{denv}})) \\ \mathit{find\_catcher}(\mathsf{tenv}, \texttt{v\_ty}, \texttt{catchers}) = \mathsf{None} \qquad \mathit{eval\_block}(\texttt{env1}, \texttt{s}) \xrightarrow{\text{eval}} C \ /\!/ \ \texttt{\#DE} \\ D := \mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, C) \qquad \texttt{g} := \texttt{s\_g} \xrightarrow{\texttt{asl\_po}} \mathit{graph}(D)\end{array}}{\mathit{eval\_catchers}(\mathsf{env}, \texttt{catchers}, \langle \texttt{s}\rangle, \texttt{s\_m}) \xrightarrow{\text{eval}} D(\mathit{graph} \mapsto \texttt{g})}$$

14.4 SemanticsRule.CatchNone

14.4.1 Prose

All of the following apply:

- `s_m` is Throwing((⟨value_read_from(v, e_id), v_ty⟩, s_g), env_throw);
- env consists of the static environment tenv and dynamic environment denv;
- `env_throw` consists of the static environment tenv and dynamic environment `denv_throw`;
- finding the first catcher with the static environment tenv, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives None (that is, neither of the `catch` clauses matches the raised exception);
- since there no `otherwise` clause, the result is `s_m`.

14.4.2 Example

The specification:

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

func main () => integer
begin

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

```
    assert TRUE;
  end

  return 0;
end
```

does not print anything.

14.4.3 Formally

$$\frac{\texttt{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\mathsf{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \quad \texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \qquad \mathit{find\_catcher}(\mathsf{tenv}, \texttt{v\_ty}, \texttt{catchers}) = \mathsf{None}}{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \mathsf{None}, \texttt{s\_m}) \xrightarrow{\mathsf{eval}} \texttt{s\_m}}$$

14.5 SemanticsRule.CatchNoThrow

14.5.1 Prose

all of the following apply:

- One of the following holds:
  - (IMPLICIT_THROW) `s_m` is Throwing((None, `s_g`), `env_throw`) (that is, an implicit throw);
  - (NON_THROWING) `s_m` is a normal configuration (that is, the domain of `s_m` is Normal);
- the result is `s_m`.

14.5.2 Example

The specification:

```
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`.

14.5.3 Formally

$$\frac{\texttt{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\mathsf{None}, \texttt{s\_g}), \texttt{env\_throw})}{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \_, \texttt{s\_m}) \xrightarrow{\text{eval}} \texttt{s\_m}} \text{IMPLICIT\_THROW}$$

$$\frac{\texttt{config\_domain}(\texttt{s\_m}) = \mathsf{Normal}}{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \_, \texttt{s\_m}) \xrightarrow{\text{eval}} \texttt{s\_m}} \text{NON\_THROWING}$$

14.6 SemanticsRule.FindCatcher

The (recursively-defined) helper relation

$$\mathit{find\_catcher}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{ty}}^{\text{v\_ty}}, \overbrace{\mathsf{catcher}^*}^{\text{catchers}}) \bigtimes \langle \mathsf{catcher} \rangle \ ,$$

returns the first catcher clause in `catchers` that matches the type `v_ty` (as a singleton set), or an empty set (`None`), by invoking *type_satisfies* with the static environment `tenv`.

14.6.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `catchers` is an empty list;
  - the result is `None`.
- All of the following apply (MATCH):
  - `catchers` has `c` as its head and `catchers1` as its tail;
  - `c` consists of (`name_opt`, `e_ty`, `s`);
  - `v_ty` type-satisfies `e_ty` in the static environment `tenv`;
  - the result is the singleton set for `c`.
- All of the following apply (NO_MATCH):
  - `catchers` has `c` as its head and `catchers1` as its tail;
  - `c` consists of (`name_opt`, `e_ty`, `s`);
  - `v_ty` does not type-satisfy `e_ty` in the static environment `tenv`;

* the result of finding a catcher for `v_ty` with the type environment `tenv` in the tail list `catchers1` is d;
* the result is d.

14.6.2 Formally

$$\frac{}{\textit{find\_catcher}(\texttt{tenv}, \texttt{v\_ty}, [\,]) \xrightarrow{\text{eval}} \texttt{None}}\ \text{EMPTY}$$

$$\frac{\texttt{catchers} \overset{\text{is}}{=} [\texttt{c}] + \texttt{catchers1} \qquad \texttt{c} \overset{\text{is}}{=} (\texttt{name\_opt}, \texttt{e\_ty}, \texttt{s}) \qquad \textit{type\_satisfies}(\texttt{tenv}, \texttt{v\_ty}, \texttt{e\_ty})}{\textit{find\_catcher}(\texttt{tenv}, \texttt{v\_ty}, \texttt{catchers}) \xrightarrow{\text{eval}} \langle \texttt{c} \rangle}\ \text{MATCH}$$

$$\frac{\texttt{catchers} \overset{\text{is}}{=} [\texttt{c}] + \texttt{catchers1} \qquad \texttt{c} \overset{\text{is}}{=} (\texttt{name\_opt}, \texttt{e\_ty}, \texttt{s}) \qquad \neg\textit{type\_satisfies}(\texttt{tenv}, \texttt{v\_ty}, \texttt{e\_ty}) \qquad d := \textit{find\_catcher}(\texttt{tenv}, \texttt{v\_ty}, \texttt{catchers1})}{\textit{find\_catcher}(\texttt{tenv}, \texttt{v\_ty}, \texttt{catchers}) \xrightarrow{\text{eval}} d}\ \text{NO\_MATCH}$$

14.6.3 Comments

When the `catch` of a `try` statement is executed, then the thrown exception is caught by the first catcher in that `catch` which it type-satisfies or the `otherwise_opt` in that catch if it exists.

14.7 SemanticsRule.RethrowImplicit

The helper relation

$$\textit{rethrow\_implicit}(\overbrace{\text{value\_read\_from}(\mathbb{V}, \mathbb{I})}^{\texttt{v}}, \overbrace{\texttt{ty}}^{\texttt{v\_ty}}, \overbrace{\text{TOutConfig}}^{\texttt{res}}) \bigtimes \text{TOutConfig}$$

changes *implicit throwing configurations* into *explicit throwing configurations*. That is, configurations of the form $\text{Throwing}((\texttt{None}, \texttt{g}), \texttt{env\_throw1}))$.

rethrow_implicit leaves non-throwing configurations, and *explicit throwing configurations*, which have the form $\text{Throwing}(\langle(\text{value\_read\_from}(\texttt{v}', \texttt{e\_id}), \texttt{v\_ty}')\rangle, \texttt{g})$, as is. Implicit throwing configurations are changed by substituting the optional value_read_from configuration-exception type pair with `v` and `v_ty`, respectively.

14.7.1 Prose

One of the following applies:

- All of the following apply (IMPLICIT_THROWING):

  - `res` is Throwing((None, g), env_throw1), which is an implicit throwing configuration;
  - the result is Throwing((⟨(v, v_ty)⟩, g), env_throw1).

- All of the following apply (EXPLICIT_THROWING):
  - `res` is Throwing(⟨(v′, v_ty′)⟩, g), which is an explicit throwing configuration (due to (v′, v_ty′));
  - the result is Throwing((⟨(v′, v_ty′)⟩, g), env_throw1).
    That is, the same throwing configuration is returned.

- All of the following apply (NON_THROWING):
  - the configuration, C, domain is non-throwing;
  - the result is C.

14.7.2 Formally

IMPLICIT_THROWING

$$\textit{rethrow\_implicit}(\mathtt{v}, \mathtt{v\_ty}, \mathsf{Throwing}((\mathsf{None}, \mathtt{g}), \mathtt{env\_throw1})) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}((\langle(\mathsf{value\_read\_from}(\mathtt{v}, \mathtt{e\_id}), \mathtt{v\_ty})\rangle, \mathtt{g}), \mathtt{env\_throw1})$$

EXPLICIT_THROWING

$$\textit{rethrow\_implicit}(\mathtt{v}, \mathtt{v\_ty}, \mathsf{Throwing}((\langle(\mathtt{v}', \mathtt{v\_ty}')\rangle, \mathtt{g}), \mathtt{env\_throw1})) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}((\langle(\mathtt{v}', \mathtt{v\_ty}')\rangle, \mathtt{g}), \mathtt{env\_throw1})$$

NON_THROWING

$$\frac{\mathtt{config\_domain}(C) \neq \mathsf{Throwing}}{\textit{rethrow\_implicit}(\_, \_, C, \_) \xrightarrow{\mathsf{eval}} C}$$

14.7.3 Comments

An expressionless `throw` statement causes the exception which the currently executing catcher caught to be thrown.

Chapter 15

Evaluation of Subprograms

15.1 Informal Preamble

An ASL specification describes behavior in terms of the execution of a single thread of control. Executing a function or procedure may modify global state and can result in one of the following situations:

- The function/procedure returns successfully (i.e., without throwing an ASL exception or detecting a dynamic error). In this case, the result is a possibly modified global state and, for functions, a return value.
- The function/procedure throws an ASL exception. In this case, the result is a possibly modified global state and an exception object (that could be caught by the caller in a try-catch block, if desired).
- The function/procedure detects a dynamic error condition. This indicates an error in the specification and the global state and any other behavior is meaningless.
- The function/procedure enters an infinite loop. This indicates an error in the specification and the global state and any other behavior is meaningless.

Note that Arm's implementation of ASL also supports some builtin functions for printing to the console, etc. These internal extensions are not used in Arm's published specifications but could be added to the description of the semantics.

15.1.1 Execution-time subprograms

Subprogram declarations in ASL are either execution-time subprogram declarations or non-execution-time subprogram declarations.

A subprogram declaration is an execution-time declaration if it makes use of any of the following:

- an execution-time storage element

- an execution-time expression
- an execution-time subprogram invocation

Subprogram invocations are also either execution-time or non-execution-time invocations.

A subprogram invocation is an execution-time invocation if the invoked subprogram has an execution-time declaration or if the invocation contains any of the following:

- an execution-time storage element
- an execution-time expression
- a bitvector whose width is an execution-time expression

This means that an execution-time subprogram declaration shall only have execution-time invocations.

15.1.2 Compile-time-constant subprograms

Subprogram declarations in ASL are either compile-time-constant subprogram declarations or non-compile-time- constant subprogram declarations.

A subprogram declaration is a compile-time-constant declaration if all of the following are true:

- the subprogram is side-effect-free
- all assignments in the subprogram are to the subprogram's local variables
- all expressions in the subprogram are compile-time-constant expressions
- any subprogram invocations made in the subprogram are compile-time-constant subprogram invocations

Standard (built-in) functions, as defined in the ASL standard library, are generally compile-time-constant, unless otherwise stated.

Subprogram invocations are also either compile-time-constant subprogram invocations or non-compile-time-constant subprogram invocations.

A subprogram invocation is a compile-time-constant invocation if all the following hold:

- the invoked subprogram is a compile-time-constant subprogram
- all of the actual arguments are compile-time-constant expressions
- all actual arguments which are bitvectors were declared with a constant expression width

Note:

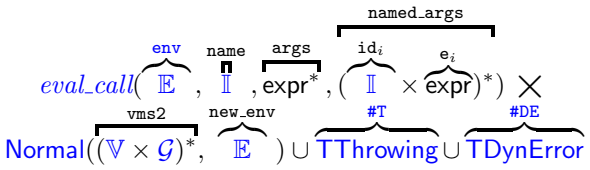
Subprogram declarations or invocations may be both non-compile-time-constant and non-execution-time.

For example, storage elements declared with config are non-execution-time and non-compile-time-constant so any use of them in a function or procedure declaration or invocation renders it non-compile-time-constant, but does not cause it to be execution-time.

Subprogram declarations or invocations may not be both compile-time-constant and execution-time since if the conditions for execution-time are met then the conditions for compile-time-constant cannot be met, and vice versa.

15.2 Formal Preamble

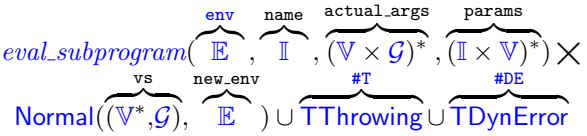
The relation



evaluates a call to the subprogram named `name` in the environment `env`, with the argument expressions `args`, and the parameter expressions `named_args`. The evaluation results in either a list of returned values, each one associated with an execution graph, and a new environment; or an abnormal configuration.

The evaluation first evaluates the expressions corresponding to the arguments and parameters and then passes their values in a resulting configuration to the helper relation `eval_subprogram`.

The relation



evaluates the subprogram named `name` in the environment `env`, with `actual_args` the list of actual arguments, and `params` the list of arguments deduced by type equality. The result is either a normal configuration or an abnormal configuration. In the case of a normal configuration, it consists of a list of pairs with a value and an identifier, and a new environment `new_env`. The values represent values returned by the subprogram call and the identifiers are used in generating execution graph constraints for the returned values.

The main subprogram call relation is given by SemanticsRule.Call (see Section 15.3). The different types of subprogram calls are evaluated via one of the following rules:

- SemanticsRule.FPrimitive (see Section 15.4)
- SemanticsRule.FCall (see Section 15.5)

We also define the following helper rules:

- SemanticsRule.ReadValueFrom (see Section 15.6)
- SemanticsRule.WriteRetVals (see Section 15.7)
- SemanticsRule.AssignArgs (see Section 15.8)
- SemanticsRule.AssignNamedArgs (see Section 15.9)
- SemanticsRule.MatchFuncRes (see Section 15.10)

15.3 SemanticsRule.Call

15.3.1 Prose

All of the following apply:

- `named_args` is a list of identifier-expression pairs $(\texttt{id}_i, \texttt{e}_i)$, for $i = 1..k$;
- `names` is the list of identifiers in `named_args`;
- `nargs1` is the list of argument expressions in `named_args`;
- evaluating each expression in `args` separately in `env` as per Section 6.23 is Normal(`args`, `env1`)$^{\#\texttt{T},\#\texttt{DE}}$;
- evaluating each expression in `nargs` separately in `env1` as per Section 6.23 is Normal(`nargs2`, `env2`)$^{\#\texttt{T},\#\texttt{DE}}$;
- `nargs2` is the list of value-execution graph pairs $\texttt{m}_i$, for $i = 1..k$;
- `nargs3` is the list of pairs $(\texttt{id}_i, \texttt{m}_i)$, for $i = 1..k$ (this is the format needed for `eval_subprogram`);
- `env2` consists of the static environment `tenv` and the dynamic environment `denv2`;
- the environment `env2′` is defined as the environment consisting of the static environment `tenv` and the dynamic environment with the global component of `denv2` and an empty local component (intuitively, this is because the called subprogram does not have access to the local environment of the caller);
- evaluating the subprogram named `name` with arguments `vargs` and parameters `nargs3` in `denv2′` is Normal(`vms`, (`global`, _)) (that is, we ignore the local environment of the callee)$^{\#\texttt{T},\#\texttt{DE}}$;
- the list `vms` consists of value-identifier pairs $(\texttt{v}_j, \texttt{rid}_j)$, for $i = 1..n$;
- applying the helper relation *read_value_from* to each $(\texttt{v}_j, \texttt{rid}_j)$ results in $\texttt{vms2}_j$, for $i = 1..n$;

- vms2 is defined as the list of $\texttt{vms2}_j$, for $i = 1..n$;
- new_env consists of the static environment tenv and the dynamic environment consisting of global as the global component and the local component of denv2 (that is, we restore the local environment to that of the caller and drop the local environment of the callee).
- the entire evaluation results in Normal(vms2, new_env).

15.3.2 Formally

$$\frac{\begin{array}{c}\texttt{named\_args} \overset{\text{is}}{=} [i = 1..k : (\texttt{id}_i, \texttt{e}_i)] \\ \texttt{names} := [i = 1..k : \texttt{id}_i] \qquad \texttt{nargs1} := [i = 1..k : \texttt{e}_i] \\ \textit{eval\_expr\_list\_m}(\texttt{env}, \texttt{args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vargs}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \textit{eval\_expr\_list\_m}(\texttt{env1}, \texttt{nargs1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{nargs2}, \texttt{env2}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{nargs2} \overset{\text{is}}{=} [i = 1..k : \texttt{m}_i] \\ \texttt{nargs3} := [i = 1...k : (\texttt{id}_i, \texttt{m}_i)] \qquad \texttt{env2} \overset{\text{is}}{=} (\texttt{tenv}, \texttt{denv2}) \qquad \texttt{env2}' := (\texttt{tenv}, (G^{\texttt{denv2}}, \emptyset_\lambda)) \\ \textit{eval\_subprogram}(\texttt{env2}', \texttt{name}, \texttt{vargs}, \texttt{nargs3}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms}, (\texttt{global}, \_)) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{vms} \overset{\text{is}}{=} [j = 1..n : (\texttt{v}_j, \texttt{rid}_j)] \qquad j = 1..n : \textit{read\_value\_from}(\texttt{v}_j, \texttt{rid}_j) \xrightarrow{\text{eval}} \texttt{vms2}_j \\ \texttt{vms2} := [j = 1..n : \texttt{vms2}_j] \qquad \texttt{new\_env} := (\texttt{tenv}, (\texttt{global}, L^{\texttt{denv2}})) \end{array}}{\textit{eval\_call}(\texttt{env}, \texttt{name}, \texttt{args}, \texttt{named\_args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms2}, \texttt{new\_env})}$$

15.4 SemanticsRule.FPrimitive

15.4.1 Prose

All of the following apply:

- env consists of the static environment tenv and the dynamic environment with genv as its global component and an empty local component;
- finding the function named name in the static environment tenv gives a func AST node with the body field SB_Primitive;
- evaluating the primitive subprogram name with the actual arguments actual_args is $\text{Normal}(\texttt{vms}, \texttt{g1})^{/\!/\texttt{\#DE}}$;
- writing the returned values vms as per Section 15.7 gives vsm;
- vsm is a pair consisting of the list of values vs and execution graph g2;
- new_g is the ordered composition of g1 and g2 with the asl_data label;
- new_env is the environment with tenv as its static environment component and the dynamic environment consisting of genv as its global component and an empty local component;
- the result of the entire evaluation is Normal((vs, new_g), new_env).

15.4.2 Example

In the specification:

```
func main () => integer
begin

  print("Hello, world!");

  return 0;
end
```

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

15.4.3 Formally

The following rule utilizes the transition relation

$$\mathit{eval\_primitive}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{actual\_args}}) \bigtimes \mathsf{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms}}, \overbrace{\mathcal{G}}^{\texttt{g1}}) \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}},$$

which parameterizes the ASL semantics and allows evaluating primitive subprograms. That is, it is not a part of $\xrightarrow{\mathsf{eval}}$ but rather a separate transition relation denoted $\overset{\mathsf{primitive}}{\rightsquigarrow}$.

$$\frac{\begin{array}{c}\mathsf{env} \overset{\mathsf{is}}{=} (\mathsf{tenv}, (\mathsf{genv}, \emptyset_\lambda)) \qquad \mathit{find\_func}(\mathsf{tenv}, \texttt{name}) = \{\texttt{body} = \mathsf{SB\_Primitive} \ldots\} \\ \mathit{eval\_primitive}(\texttt{name}, \texttt{actual\_args}) \overset{\mathsf{primitive}}{\rightsquigarrow} \mathsf{Normal}(\texttt{vms}, \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{write\_ret\_vals}(\texttt{vms}) \xrightarrow{\mathsf{eval}} \texttt{vsm} \\ \texttt{vsm} \overset{\mathsf{is}}{=} (\texttt{vs}, \texttt{g2}) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\mathsf{asl\_data}} \texttt{g2} \qquad \texttt{new\_env} := (\mathsf{tenv}, (\mathsf{genv}, \emptyset_\lambda))\end{array}}{\mathit{eval\_subprogram}(\mathsf{env}, \texttt{name}, \texttt{actual\_args}, \texttt{params}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env})}$$

15.5 SemanticsRule.FCall

15.5.1 Prose

All of the following apply:

- env consists of the static environment tenv and the dynamic environment with the global component genv and an empty local component;
- finding the function named name in tenv gives the AST func node with body SB_ASL(body) and arguments arg_decls;
- env1 is the environment consisting of the static environment tenv and the dynamic1 environment consisting of the dynamic component from denv and an empty local component;

- assigning the actual arguments with $((\texttt{env1}, \emptyset_{\texttt{g}}), \texttt{arg\_decls}, \texttt{actual\_args})$ as per Section 15.8 gives $(\texttt{env2}, \mathit{vgtwo})$ make sure that each formal argument in `arg_decls` is locally bound to the corresponding actual argument in `actual_args`;

- declaring and assigning the parameter values with $((\texttt{env2}, \texttt{g2}), \texttt{params})$ as per Section 15.9 gives $(\texttt{env3}, \texttt{g3})$;

- evaluating the body of the subprogram `body` as a statement in in `env3` is $\texttt{res}^{/\!/\texttt{\#T,\#DE}}$;

- matching the result `res` to obtain a normal configuration as per Section 15.10 gives C;

- `new_g` is the ordered composition of `g2` and `g3` with the `asl_po` edge;

- the result is C with its graph substituted for `new_g`.

15.5.2 Example

The specification:

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

  return x + 1;

end

func bar (x : integer)
begin

  assert x == 3;

end

func main () => integer
begin

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

  return 0;
end
```

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

15.5.3 Formally

$$\frac{\begin{array}{c}\texttt{env} \overset{\text{is}}{=} (\texttt{tenv}, \texttt{denv}) \\ \mathit{find\_func}(\texttt{tenv}, \texttt{name}) \overset{\text{is}}{=} \{\texttt{body} : \mathsf{SB\_ASL}(\texttt{body}), \texttt{args} : \texttt{arg\_decls}, \ldots\} \\ \texttt{env1} := (\texttt{tenv}, (G^{\texttt{denv}}, \emptyset_\lambda)) \\ \mathit{assign\_args}((\texttt{env1}, \emptyset_{\texttt{g}}), \texttt{arg\_decls}, \texttt{actual\_args}) \xrightarrow{\text{eval}} (\texttt{env2}, \texttt{g2}) \\ \mathit{assign\_named\_args}((\texttt{env2}, \texttt{g2}), \texttt{params}) \xrightarrow{\text{eval}} (\texttt{env3}, \texttt{g3}) \\ \mathit{eval\_stmt}(\texttt{env3}, \texttt{body}) \xrightarrow{\text{eval}} \texttt{res} \ /\!/ \ \texttt{\#T}, \texttt{\#DE} \\ \mathit{match\_func\_res}(\texttt{res}) \xrightarrow{\text{eval}} C \qquad \texttt{new\_g} := \texttt{g2} \xrightarrow{\texttt{asl\_po}} \texttt{g3}\end{array}}{\mathit{eval\_subprogram}(\texttt{env}, \texttt{name}, \texttt{actual\_args}, \texttt{params}) \xrightarrow{\text{eval}} C(\mathit{graph} \mapsto \texttt{new\_g})}$$

15.5.4 Comments

It is not an error for execution of a procedure or setter to end without a return statement.

15.6 SemanticsRule.ReadValueFrom

The helper relation

$$\mathit{read\_value\_from}(\mathbb{V},\mathbb{I}) \bigtimes (\mathbb{V} \times \mathcal{G})$$

generates an execution graph for reading the given value to a variable given by the identifier, and pairs it with the given value.

15.6.1 Prose

All of the following apply:

- reading the value `v` into the variable named `id` gives `new_g`;
- the result is (`v`, `new_g`).

15.6.2 Formally

$$\frac{\mathit{read\_identifier}(\texttt{v}, \texttt{id}) \xrightarrow{\text{eval}} \texttt{new\_g}}{\mathit{read\_value\_from}(\texttt{v}, \texttt{id}) \xrightarrow{\text{eval}} (\texttt{v}, \texttt{new\_g})}$$

15.7 SemanticsRule.WriteRetVals

The relation

$$\mathit{write\_ret\_vals}(\overbrace{(\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g1}})^*}^{\texttt{m}})^{\texttt{vsm}} \bigtimes (\overbrace{\mathbb{V}^*}^{\texttt{vs}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}}) \ .$$

generates Write Effects for the values returned by the evaluation of a primitive subprogram:

15.7.1 Prose

one of the following applies:

- All of the following apply (EMPTY):
  - the list of value-execution graphs `vsm` is empty;
  - the result is a pair consisting of an empty list and an empty graph.
- All of the following apply (NON_EMPTY):
  - the list of value-execution graphs `vsm` has `m` as its head and `vsm1` as its tail;
  - `x` is a fresh identifier;
  - `m` consists of the value `v` and execution graph `g1`;
  - the execution graph `g2` is generating by writing the value `v` for the variable named `x`;
  - writing the returned values in `vsm1` gives $(\texttt{vs1}, \texttt{g3})$;
  - `s` is defined as the list with `v` as its head and `vs1` as its tail;
  - `new_g` is defined by first taking the ordered composition of `g1` and `g2` with the `asl_data` edge and then composing the resulting execution graph in parallel with `g3`;
  - the result of the entire evaluation is $(\texttt{vs}, \texttt{new\_g})$.

15.7.2 Formally

EMPTY

$$\mathit{write\_ret\_vals}([\,]) \xrightarrow{\text{eval}} ([\,], \emptyset_{\text{g}})$$

NON_EMPTY

$$\frac{\begin{array}{c}\texttt{vsm} \overset{\text{is}}{=} [\texttt{m}] + \texttt{vsm1} \qquad \texttt{x} \in \mathbb{I} \text{ is fresh} \\ \texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g1}) \qquad \mathit{write\_identifier}(\texttt{x}, \texttt{v}) \xrightarrow{\text{eval}} \texttt{g2} \qquad \mathit{write\_ret\_vals}(\texttt{vsm1}) \xrightarrow{\text{eval}} (\texttt{vs1}, \texttt{g3}) \\ \texttt{vs} := [\texttt{v}] + \texttt{vs1} \qquad \texttt{new\_g} := (\texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2}) \parallel \texttt{g3}\end{array}}{\mathit{write\_ret\_vals}(\texttt{vsm}) \xrightarrow{\text{eval}} (\texttt{vs}, \texttt{new\_g})}$$

15.8 SemanticsRule.AssignArgs

The helper relation

$$\mathit{assign\_args}((\overbrace{\mathbb{E}}^{\texttt{env}} \times \overbrace{\mathcal{G}}^{\texttt{g1}}), \overbrace{(\mathbb{I} \times \mathsf{ty})^*}^{\texttt{arg\_decls}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{actual\_args}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

assigns the values of (the actual) arguments to the formal variables of a given subprogram.

15.8.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
  - both `arg_decls` and `actual_args` are empty lists;
  - the result is (`env`, `g1`).
- All of the following apply (NON_EMPTY):
  - `arg_decls` has (`x`, `_`) as its head and `arg_decls` as its tail, and `actual_args` has `m` as its head and `actual_args` as its tail;
  - declaring the local identifier `x` with `m` in `env` as per Section 17.14 gives (`env1`, `g2`).
  - assigning the remaining lists `arg_decls` and `actual_args` with the environment `env1` and the ordered composition of `g1` and `g2` with the `asl_po` edge gives (`new_env`, `new_g`).
  - the entire result of the evaluation is (`new_env`, `new_g`).

15.8.2 Formally

EMPTY

$$\mathit{assign\_args}((\texttt{env}, \texttt{g1}), [\,], [\,]) \xrightarrow{\text{eval}} (\texttt{env}, \texttt{g1})$$

NON_EMPTY

$$\frac{\mathit{declare\_local\_identifier\_mm}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{env1}, \texttt{g2}) \quad \mathit{assign\_args}((\texttt{env1}, \texttt{g1} \xrightarrow{\texttt{asl\_po}} \texttt{g2}), \texttt{arg\_decls}, \texttt{actual\_args}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g})}{\mathit{assign\_args}((\texttt{env}, \texttt{g1}), [(\texttt{x}, \_)] + \texttt{arg\_decls}, [\texttt{m}] + \texttt{actual\_args}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g})}$$

15.9 SemanticsRule.AssignNamedArgs

The helper relation

$$\mathit{assign\_named\_args}((\overbrace{\mathbb{E}}^{\texttt{env}} \times \overbrace{\mathcal{G}}^{\texttt{g1}}), \overbrace{(\overbrace{\mathbb{I}}^{\texttt{x}} \times (\overbrace{\mathbb{V} \times \mathcal{G}}^{\texttt{m}}))^*}^{\texttt{params}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

assigns values to the variables that correspond to the parameters of a given subprogram.

15.9.1 Prose

One of the following applies:

- All of the following apply (EMPTY):

  - `params` is an empty list;
  - the result is `env`, `g1`;

- All of the following apply (DECLARED):
  - `params` has (`x`, `m`) as its head and `params1` as its tail;
  - `env` consists of the static environment `tenv` and dynamic environment `denv`;
  - `x` is bound to a value in `denv`;
  - `acc` is defined as (`env`, `g1`);
  - assigning the named args with `acc` and `params1` gives (`new_env`, `g2`);
  - `new_g` is the ordered composition of `g1` and `g2` with the `asl_po` edge.
  - the result is (`new_env`, `new_g`).

- All of the following apply (NOT_DECLARED):
  - `params` has (`x`, `m`) as its head and `params1` as its tail;
  - `env` consists of the static environment `tenv` and dynamic environment `denv`;
  - `x` is not bound to a value in `denv`;
  - declaring the local identifier `x` with `m` in `env`, as per Section 17.13, gives (`env1`, `g2`);
  - `acc` is defined as (`env1`, `g2`);
  - assigning the named args with `acc` and `params1` gives (`new_env`, `g3`);
  - `new_g` is the ordered composition of `g1`, `g2`, and `g3` with the `asl_po` edge.
  - the result is (`new_env`, `new_g`).

15.9.2 Formally

We use the helper predicate

$$\texttt{is\_bound}(\mathsf{denv}, \texttt{x}) \triangleq G^{\mathsf{denv}}(\texttt{x}) \neq \bot \vee L^{\mathsf{denv}}(\texttt{x}) \neq \bot$$

to test whether the variable `x` is bound in the dynamic environment `denv`.

EMPTY

$$\mathit{assign\_named\_args}((\texttt{env}, \texttt{g1}), [\,]) \xrightarrow{\text{eval}} (\texttt{env}, \texttt{g1})$$

DECLARED

$$\frac{\begin{array}{c}\texttt{params} \overset{\text{is}}{=} [(\texttt{x}, \texttt{m})] + \texttt{params1} \\ \texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \qquad \texttt{is\_bound}(\mathsf{denv}, \texttt{x}) \qquad \texttt{acc} := (\texttt{env}, \texttt{g1}) \\ \mathit{assign\_named\_args}(\texttt{acc}, \texttt{params1}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g2}) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_po}} \texttt{g2}\end{array}}{\mathit{assign\_named\_args}((\texttt{env}, \texttt{g1}), \texttt{params}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

NOT_DECLARED

$$\frac{\begin{array}{c}\texttt{params} \overset{\text{is}}{=} [(\texttt{x}, \texttt{m})] + \texttt{params1} \qquad \texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \\ \neg\texttt{is\_bound}(\mathsf{denv}, \texttt{x}) \qquad \mathit{declare\_local\_identifier\_m}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{env1}, \texttt{g2}) \\ \texttt{acc} := (\texttt{env1}, \texttt{g2}) \qquad \mathit{assign\_named\_args}(\texttt{acc}, \texttt{params1}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g3}) \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_po}} \texttt{g2} \xrightarrow{\text{asl\_po}} \texttt{g3}\end{array}}{\mathit{assign\_named\_args}((\texttt{env}, \texttt{g1}), \texttt{params}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

15.10 SemanticsRule.MatchFuncRes

The helper relation

$$\mathit{match\_func\_res}(\mathsf{TContinuing} \cup \mathsf{TReturning}) \bigtimes \mathsf{Normal}(((\mathbb{I} \times \mathbb{V})^* \times \mathcal{G}), \mathbb{E})$$

converts normal and throwing configurations into corresponding normal configurations that can be returned by a subprogram evaluation.

15.10.1 Prose

One of the following applies:

- All of the following apply (CONTINUING):
  - the given configuration is $\mathsf{Continuing}(\texttt{g}, \texttt{env})$. This happens when, for example, the subprogram called is either a setter or a procedure;
  - the result is $\mathsf{Normal}(([\,], \texttt{g}), \texttt{env})$.
- All of the following apply (RETURNING):
  - the given configuration is $\mathsf{Returning}(\texttt{xs}, \texttt{ret\_env})$, which is the case of a function;
  - $\texttt{xs}$ is the list $\texttt{v}_i$, for $i = 1..k$;
  - define the list of fresh identifiers $\texttt{id}_i$, for $i = 1..k$;
  - define $\texttt{vs}$ to be $(\texttt{v}_i, \texttt{id}_i)$, for $i = 1..k$;
  - the result is $\mathsf{Normal}((\texttt{vs}, \emptyset_{\mathsf{g}}), \texttt{ret\_env})$.

15.10.2 Formally

CONTINUING

$$\mathit{match\_func\_res}(\mathsf{Continuing}(\mathtt{g}, \mathtt{env})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathsf{[\,]}, \mathtt{g}), \mathtt{env})$$

RETURNING

$$\frac{\mathtt{xs} \overset{\mathsf{is}}{=} [i = 1..k : \mathtt{v}_i] \qquad i = 1..k : \mathtt{id}_i \in \mathbb{I} \text{ is fresh} \qquad \mathtt{vs} := [i = 1..k : (\mathtt{v}_i, \mathtt{id}_i)]}{\mathit{match\_func\_res}(\mathsf{Returning}(\mathtt{xs}, \mathtt{ret\_env})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{vs}, \emptyset_{\mathtt{g}}), \mathtt{ret\_env})}$$

Chapter 16

Evaluation of Specifications

The rule SemanticsRule.TopLevel (see Section 16.1) evaluates entire specifications with the help of the following rules:

- SemanticsRule.EvalGlobals (see Section 16.2);
- SemanticsRule.BuildGlobalEnv (see Section 16.3).

16.1 SemanticsRule.TopLevel

The relation

$$\mathit{eval\_spec}(\ \overbrace{\mathsf{spec}}^{\texttt{parsed\_ast}}\ ,\ \overbrace{\mathsf{spec}}^{\texttt{parsed\_std}}\ ) \bigtimes ((\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}})$$

evaluates the `main` function in a given specification and standard library.

The function `annotate_spec`, which is defined in the typing reference [6], takes an initial typing environment and a untyped AST and returns a corresponding typed AST and typing environment.

16.1.1 Prose

All of the following apply:

- the AST for the parsed specification, `parsed_spec`, and AST for the parsed standard library, `parsed_std`, are concatenated to give `parsed_ast`;
- applying the type-checker to `parsed_ast` with an empty static environment yields $(\texttt{typed\_spec}, \mathsf{tenv})$;
- populating the environment with the declarations of the global storage elements is $(\texttt{env}, \texttt{g1})^{/\!/\texttt{\#DE}}$;

- One of the following applies:
  - All of the following apply (NORMAL):
    - evaluating the subprogram `main` with an empty list of actual arguments and empty list of parameters in `env` is $\mathsf{Normal}([(\mathtt{v}, \mathtt{g2})], \_)^{/\!/\,\mathtt{\#DE}}$;
    - `new_g` is the ordered composition of `g1` and `g2` with the `asl_po` edge;
    - the result of the entire evaluation is $(\mathtt{v}, \mathtt{new\_g})$.
  - All of the following apply (THROWING):
    - evaluating the subprogram `main` with an empty list of actual arguments and empty list of parameters in `env` is $\mathsf{Throwing}(\mathtt{v\_opt}, \_)$, which is an uncaught exception;
    - the result of the entire evaluation is an error indicating that an exception was not caught.

16.1.2 Example

16.1.3 Formally

NORMAL

$$\frac{\begin{array}{c}\mathtt{parsed\_ast} := \mathtt{parsed\_spec} + \mathtt{parsed\_std} \\ \mathtt{annotate\_spec}(\emptyset_{\mathsf{tenv}}, \mathtt{parsed\_ast}) \xrightarrow{\mathsf{type}} (\mathtt{typed\_spec}, \mathtt{tenv}) \\ \mathit{build\_genv}(\mathtt{typed\_spec}, (\mathtt{tenv}, (\emptyset_\lambda, \emptyset_\lambda))) \xrightarrow{\mathsf{eval}} (\mathtt{env}, \mathtt{g1}) \;\; /\!/ \;\; \mathtt{\#DE} \\ \mathit{eval\_subprogram}(\mathtt{env}, \mathtt{"main"}, [\,], [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Normal}([(\mathtt{v}, \mathtt{g2})], \_) \;\; /\!/ \;\; \mathtt{\#DE} \\ \mathtt{new\_g} := \mathtt{g1} \xrightarrow{\mathtt{asl\_po}} \mathtt{g2}\end{array}}{\mathit{eval\_spec}(\mathtt{parsed\_ast}, \mathtt{parsed\_std}) \xrightarrow{\mathsf{eval}} (\mathtt{v}, \mathtt{new\_g})}$$

THROWING

$$\frac{\begin{array}{c}\mathtt{parsed\_ast} := \mathtt{parsed\_spec} + \mathtt{parsed\_std} \\ \mathtt{annotate\_spec}(\emptyset_{\mathsf{tenv}}, \mathtt{parsed\_ast}) \xrightarrow{\mathsf{type}} (\mathtt{typed\_spec}, \mathtt{tenv}) \\ \mathit{build\_genv}(\mathtt{typed\_spec}, (\mathtt{tenv}, (\emptyset_\lambda, \emptyset_\lambda))) \xrightarrow{\mathsf{eval}} (\mathtt{env}, \mathtt{g1}) \\ \mathit{eval\_subprogram}(\mathtt{env}, \mathtt{"main"}, [\,], [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}(\mathtt{v\_opt}, \_)\end{array}}{\mathit{eval\_spec}(\mathtt{parsed\_spec}, \mathtt{parsed\_std}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{"ERROR[UncaughtException]"})}$$

Notice that when the type-checker fails due to a type error in the given specification, the corresponding premise in the rule above does not hold, and the semantics is undefined. Indeed, the ASL semantics is only defined for well-typed specifications.

16.2 SemanticsRule.EvalGlobals

The relation

$$\mathit{eval\_globals}(\overbrace{\mathsf{decl}^*}^{\mathtt{decls}}, (\overbrace{\overbrace{\mathbb{E}}^{\mathtt{env}} \times \overbrace{\mathcal{G}}^{\mathtt{g1}}}^{\mathtt{envm}})) \bigtimes \overbrace{(\mathbb{E} \times \mathcal{G})}^{C} \cup \overbrace{\mathsf{TDynError}}^{\mathtt{\#DE}}$$

updates the input environment and execution graph by initializing the global storage declarations, either from their initializing expression or from the base value defined for their type as per Section 17.16.

16.2.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
  - there are no declarations of global variables;
  - the result is `envm`.
- All of the following apply (WITH_INITIAL_VALUE):
  - `decls` has `d` as its head and `decls'` as its tail;
  - d is the AST node for declaring a global storage element with initial value `e`, name `name`, and type `t`;
  - `envm` is the environment-execution graph pair (`env`, `g1`);
  - evaluating the side-effect-free expression `e` in `env` as per Section 6.24 is (`v`, `g2`)$^{\#DE}$;
  - declaring the global `name` with value `v` in `env` gives `env2`;
  - evaluating the remaining global declarations `decls'` with the environment `env2` and the execution graph that is the ordered composition of `g1` and `g2` with the `asl_po` label gives C;
  - the result of the entire evaluation is C.
- All of the following apply (NO_INITIAL_VALUE):
  - `decls` has `d` as its head and `decls'` as its tail;
  - d is the AST node for declaring a global storage element with no initial value, name `name`, and type `t`;
  - `envm` is the environment-execution graph pair (`env`, `g1`);
  - the base value of type `t` in `env` is (`v`, `g2`)$^{\#DE}$;
  - declaring the global `name` with value `v` in `env` gives `env2`;
  - evaluating the remaining global declarations `decls'` with the environment `env2` and the execution graph that is the ordered composition of `g1` and `g2` with the `asl_po` label gives C;
  - the result of the entire evaluation is C.

16.2.2 Example

16.2.3 Formally

EMPTY

$$\frac{\texttt{decls} \overset{\text{is}}{=} [\,]}{\textit{eval\_globals}(\texttt{decls}, \texttt{envm}) \xrightarrow{\text{eval}} \texttt{envm}}$$

WITH_INITIAL_VALUE

$$\frac{\begin{array}{c}\texttt{decls} \overset{\text{is}}{=} [\texttt{d}] + \texttt{decls}' \\ \texttt{d} \overset{\text{is}}{=} \texttt{D\_GlobalStorage}(\{\text{initial\_value} = \langle \texttt{e} \rangle, \text{name} : \texttt{name}, \text{ty} : \texttt{t}, \ldots\}) \\ \texttt{envm} \overset{\text{is}}{=} (\texttt{env}, \texttt{g1}) \qquad \textit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} (\texttt{v}, \texttt{g2}) \;\; /\!\!/ \;\; \texttt{\#DE} \\ \textit{declare\_global}(\texttt{name}, \texttt{v}, \texttt{env}) \xrightarrow{\text{eval}} \texttt{env2} \\ \textit{eval\_globals}(\texttt{decls}', (\texttt{env2}, \texttt{g1} \xrightarrow{\text{asl\_po}} \texttt{g2})) \xrightarrow{\text{eval}} C\end{array}}{\textit{eval\_globals}(\texttt{decls}, \texttt{envm}) \xrightarrow{\text{eval}} C}$$

NO_INITIAL_VALUE

$$\frac{\begin{array}{c}\texttt{decls} \overset{\text{is}}{=} [\texttt{d}] + \texttt{decls}' \\ \texttt{d} \overset{\text{is}}{=} \texttt{D\_GlobalStorage}(\{\text{initial\_value} : \texttt{None}, \text{name} : \texttt{name}, \text{ty} : \texttt{t}, \ldots\}) \\ \texttt{envm} \overset{\text{is}}{=} (\texttt{env}, \texttt{g1}) \qquad \textit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{v}, \texttt{g2}) \;\; /\!\!/ \;\; \texttt{\#DE} \\ \textit{declare\_global}(\texttt{name}, \texttt{v}, \texttt{env}) \xrightarrow{\text{eval}} \texttt{env2} \\ \textit{eval\_globals}(\texttt{decls}', (\texttt{env2}, \texttt{g1} \xrightarrow{\text{asl\_po}} \texttt{g2})) \xrightarrow{\text{eval}} C\end{array}}{\textit{eval\_globals}(\texttt{decls}, \texttt{envm}) \xrightarrow{\text{eval}} C}$$

16.3 SemanticsRule.BuildGlobalEnv

The helper relation

$$\textit{build\_genv}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{spec}}^{\text{typed\_spec}}) \bigtimes (\overbrace{\mathbb{E}}^{\text{new\_env}} \times \overbrace{\mathcal{G}}^{\text{new\_g}}) \cup \overbrace{\text{TDynError}}^{\#DE}$$

populates the environment and output execution graph with the global storage declarations. This works by traversing the global storage declarations in *dependency order* and updating the environment accordingly. By dependency order, we mean that if a declaration b refers to an identifier declared in a then a is evaluated before b.

16.3.1 Prose

All of the following apply:

- sorting the declarations of the global storage elements in topological order with respect to the dependency order gives `decls`;
- evaluating the global storage declarations in `decls` in env with the empty execution graph is $(\texttt{new\_env}, \texttt{new\_g})^{/\!/ \texttt{\#DE}}$.

- the result of the entire evaluation is (`new_env`, `new_g`).

16.3.2 Example

16.3.3 Formally

The helper relation *topological_decls*($\overbrace{\mathsf{decl}^*}^{\texttt{parsed\_spec}}$, $\overbrace{\mathsf{decl}^*}^{\texttt{parsed\_std}}$) accepts a specification and returns the subset of global storage declarations ordered by dependency order.

$$\frac{\textit{topological\_decls}(\texttt{typed\_spec}) \xrightarrow{\mathsf{eval}} \texttt{decls} \qquad \textit{eval\_globals}(\texttt{decls}, (\texttt{env}, \emptyset_{\mathsf{g}})) \xrightarrow{\mathsf{eval}} (\texttt{new\_env}, \texttt{new\_g}) \;\; /\!/ \;\; \texttt{\#DE}}{\textit{build\_genv}(\texttt{env}, \texttt{typed\_spec}) \xrightarrow{\mathsf{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

Chapter 17

Basic Utility Relations

In this chapter, we define helper relations for operating on native values, environments, and operations involving values and types. In this chapter rules are presented without examples and the corresponding code.

We now define the following relations:

- SemanticsRule.RemoveLocal Section 17.1;
- SemanticsRule.ReadIdentifier Section 17.2;
- SemanticsRule.WriteIdentifier Section 17.3;
- SemanticsRule.CreateBitvector Section 17.4;
- SemanticsRule.ConcatBitvectors Section 17.5;
- SemanticsRule.ReadFromBitvector Section 17.6;
- SemanticsRule.WriteToBitvector Section 17.7;
- SemanticsRule.GetIndex Section 17.8;
- SemanticsRule.SetIndex Section 17.9;
- SemanticsRule.GetField Section 17.10;
- SemanticsRule.SetField Section 17.11;
- SemanticsRule.DeclareLocalIdentifier Section 17.12;
- SemanticsRule.DeclareLocalIdentifierMSection 17.13;
- SemanticsRule.DeclareLocalIdentifierMM Section 17.14;
- SemanticsRule.DeclareGlobal Section 17.15;
- SemanticsRule.BaseValue Section 17.16;

17.1 SemanticsRule.RemoveLocal

17.1.1 Prose

The relation

$$\mathit{remove\_local}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}) \bigtimes \overbrace{\mathbb{E}}^{\texttt{new\_env}}$$

removes the binding of the identifier `name` from the local storage of the environment `env`.

Removal of the identifier `name` from the local storage of the environment `env` is the environment `new_env` and all of the following apply:

- `env` consists of the static environment tenv and dynamic environment denv;
- `new_env` consists of the static environment tenv and the dynamic environment with the same global component as denv — G^{denv}, and local component L^{denv}, with the identifier `name` removed from its domain.

17.1.2 Formally

(Recall that $[\texttt{name} \mapsto \bot]$ means that `name` is not in the domain of the resulting function.)

$$\frac{\texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}[\texttt{name} \mapsto \bot]))}{\mathit{remove\_local}(\texttt{env}, \texttt{name}) \xrightarrow{\mathsf{eval}} \texttt{new\_env}}$$

17.2 SemanticsRule.ReadIdentifier

17.2.1 Prose

The relation

$$\mathit{read\_identifier}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes (\overbrace{\mathbb{V}}^{\texttt{v}} \times \mathcal{G})$$

reads a value `v` into a storage element given by an identifier `name`. The result is the value and an execution graph containing a single Read Effect, which denoting reading from `name`.

17.2.2 Formally

$$\mathit{read\_identifier}(\texttt{name}, \texttt{v}) \xrightarrow{\mathsf{eval}} (\texttt{v}, \mathsf{ReadEffect}(\texttt{name}))$$

17.3 SemanticsRule.WriteIdentifier

17.3.1 Prose

The relation

$$\mathit{write\_identifier}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes \mathcal{G}$$

writes the value v into a storage element given by an identifier name. The result is an execution graph containing a single Write Effect, which denotes writing into name.

17.3.2 Formally

$$\mathit{write\_identifier}(\texttt{name}, \texttt{v}) \xrightarrow{\text{eval}} \mathsf{WriteEffect}(\texttt{name})$$

17.4 SemanticsRule.CreateBitvector

17.4.1 Prose

The relation

$$\mathit{create\_bitvector}(\overbrace{\mathbb{V}^*}^{\texttt{vs}}) \bigtimes \mathcal{BV}$$

creates a native vector value bitvector from a sequence of values vs.

17.4.2 Formally

$$\mathit{create\_bitvector}(\texttt{vs}) \xrightarrow{\text{eval}} \texttt{Bitvector}\texttt{vs}$$

17.5 SemanticsRule.ConcatBitvectors

17.5.1 Prose

The relation

$$\mathit{concat\_bitvectors}(\overbrace{\mathcal{BV}^*}^{\texttt{vs}}) \bigtimes \mathcal{BV}$$

transforms a (possibly empty) list of bitvector native values vs into a single bitvector.

17.5.2 Formally

ConcatBitvector.Empty

$$\mathit{concat\_bitvectors}([\,]) \xrightarrow{\text{eval}} \texttt{Bitvector}([\,])$$

ConcatBitvector.NonEmpty

$$\frac{\texttt{v} \overset{\text{is}}{=} \texttt{Bitvector}(\texttt{bv}) \qquad \texttt{vs} \overset{\text{is}}{=} [\texttt{v}] + \texttt{vs}' \qquad \mathit{concat\_bitvectors}(\texttt{vs}') \xrightarrow{\text{eval}} \texttt{Bitvector}(\texttt{bv}') \qquad \texttt{res} := \texttt{bv} + \texttt{bv}'}{\mathit{concat\_bitvectors}(\texttt{vs}) \xrightarrow{\text{eval}} \texttt{Bitvector}(\texttt{res})}$$

17.6 SemanticsRule.ReadFromBitvector

The relation

$$\mathit{read\_from\_bitvector}(\overbrace{\mathcal{BV}}^{\texttt{bv}}, \overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\texttt{slices}}) \bigtimes \overbrace{\mathcal{BV}}^{\texttt{v}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

reads from a bitvector `bv`, or an integer seen as a bitvector, the indices specified by the list of slices `slices`, thereby concatenating their values.

17.6.1 Prose

One of the following applies:

- all indices are in range for `bv` and the returned bitvector consists of the concatenated bits specified by the slices.
- there exists an out-of-range index and an error is returned.

17.6.2 Formally

We start by introducing a few helper relations.

The predicate $\mathit{position\_in\_range}(\mathtt{s}, \mathtt{l}, n)$ checks whether the indices starting at index `s` and up to $\mathtt{s} + \mathtt{l}$, inclusive, would refer to actual indices of a bitvector of length n:

$$\mathit{position\_in\_range}(\mathtt{s}, \mathtt{l}, \mathtt{n}) \triangleq (\mathtt{s} \geq 0) \land (\mathtt{l} \geq 0) \land (\mathtt{s} + \mathtt{l} < \mathtt{n}) \ .$$

The relation

$$\mathit{slices\_to\_positions}(\overbrace{\mathbb{N}}^{\texttt{n}}, \overbrace{(\overbrace{\mathcal{Z}}^{\mathtt{s}_i} \times \overbrace{\mathcal{Z}}^{\mathtt{l}_i})^+}^{\texttt{slices}}) \bigtimes (\overbrace{\mathbb{N}^*}^{\texttt{positions}} \cup \mathsf{TDynError})$$

returns the list of positions (indices) specified by the slices `slices`, unless an index would be out of range for a bitvector of length `n`, in which case it returns an error configuration.

SlicesToPositionsOutOfRange

$$\frac{\mathtt{slices} \overset{\text{is}}{=} [i = 1..k : (\mathtt{Int}(\mathtt{s}_i), \mathtt{Int}(\mathtt{l}_i))] \qquad j \in 1..k : \neg \mathit{position\_in\_range}(\mathtt{s}_j, \mathtt{l}_j, \mathtt{n})}{\mathit{slices\_to\_positions}(\mathtt{n}, \mathtt{slices}) \xrightarrow{\text{eval}} \mathsf{DynError}(\texttt{"ERROR[Slice\_PositionOutOfRange]"})}$$

SlicesToPositionsInRange

$$\frac{\mathtt{slices} \overset{\text{is}}{=} [i = 1..k : (\mathtt{Int}(\mathtt{s}_i), \mathtt{Int}(\mathtt{l}_i))] \qquad i = 1..k : \mathit{position\_in\_range}(\mathtt{s}_i, \mathtt{l}_i, \mathtt{n}) \qquad \mathtt{positions} := [\mathtt{s}_1, \ldots, \mathtt{s}_1 + \mathtt{l}_1] + \ldots + [\mathtt{s}_k, \ldots, \mathtt{s}_k + \mathtt{l}_k]}{\mathit{slices\_to\_positions}(\mathtt{n}, \mathtt{slices}) \xrightarrow{\text{eval}} \mathtt{positions}}$$

The function $\mathit{as\_bitvector} : (\mathcal{BV} \cup \mathcal{Z}) \to \{0,1\}^*$ transforms native value integers and native value bitvectors into a sequence of binary values:

$$\frac{}{\mathit{as\_bitvector}(\texttt{Bitvector}(\texttt{bv})) \xrightarrow{\text{eval}} \texttt{bv}}\ \textsc{AsBitvectorBitvector}$$

$$\frac{\texttt{bv} := \text{ two's complement representation of } n}{\mathit{as\_bitvector}(\texttt{Int}(n)) \xrightarrow{\text{eval}} \texttt{bv}}\ \textsc{AsBitvectorInt}$$

Finally, the rules below distinguish between empty bitvectors and non-empty bitvectors.

$$\frac{}{\mathit{read\_from\_bitvector}(\texttt{bv}, [\,]) \xrightarrow{\text{eval}} \texttt{Bitvector}([\,])}\ \textsc{ReadFromBitvector.Empty}$$

$$\frac{\mathit{as\_bitvector}(\texttt{bv}) := \texttt{b}_n \ldots \texttt{b}_1 \qquad \mathit{slices\_to\_positions}(n, \texttt{slices}) \xrightarrow{\text{eval}} [j_{1..m}] \ /\!/\ \texttt{\#DE} \qquad \texttt{v} := \texttt{Bitvector}(\texttt{b}_{j_m+1} \ldots \texttt{b}_{j_1+1})}{\mathit{read\_from\_bitvector}(\texttt{bv}, \texttt{slices}) \xrightarrow{\text{eval}} \texttt{v}}\ \textsc{ReadFromBitvector.NonEmpty}$$

Notice that the bits of a bitvector go from the least significant bit being on the right to the most significant bit being on the left, which is reflected by how the rules list the bits. The effect of placing the bits in sequence is that of concatenating the results from all of the given slices. Also notice that bitvector bits are numbered from 1 and onwards, which is why we add 1 to the indices specified by the slices when accessing a bit.

17.7 SemanticsRule.WriteToBitvector

17.7.1 Prose

The relation

$$\mathit{write\_to\_bitvector}(\overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\texttt{slices}}, \overbrace{\mathcal{BV}}^{\texttt{src}}, \overbrace{\mathcal{BV}}^{\texttt{dst}}) \times \overbrace{\mathcal{BV}}^{\texttt{v}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

overwrites the bits of `dst` at the positions given by `slices` with the bits of `src` and one of the following applies:

- all positions specified by `slices` are within range for `dst` and the modified version of `dst` with the bits of `src` at the specified positions is returned;
- there exists a position in `slices` that is not in range for `dst` and an error is returned.

17.7.2 Formally

WriteToBitvector.Empty

$$\mathit{write\_to\_bitvector}([\,], \texttt{Bitvector}([\,]), \texttt{Bitvector}([\,])) \xrightarrow{\text{eval}} \texttt{Bitvector}([\,])$$

WriteToBitvector.NonEmpty

$$\frac{\begin{array}{c}\texttt{s}_n \ldots \texttt{s}_1 := \mathit{as\_bitvector}(\texttt{src}) \\ \texttt{d}_n \ldots \texttt{d}_1 := \mathit{as\_bitvector}(\texttt{dst}) \qquad \mathit{slices\_to\_positions}(n, \texttt{slices}) \xrightarrow{\text{eval}} \texttt{positions} \;/\!/\; \texttt{\#DE} \\ \mathit{bit} = \lambda i \in 1..n. \begin{cases} \texttt{s}_i & i \in \texttt{positions} \\ \texttt{d}_i & \text{otherwise} \end{cases} \qquad \texttt{v} := \texttt{Bitvector}(\mathit{bit}(n-1) \ldots \mathit{bit}(0))\end{array}}{\mathit{write\_to\_bitvector}(\texttt{slices}, \texttt{src}, \texttt{dst}) \xrightarrow{\text{eval}} \texttt{v}}$$

17.8 SemanticsRule.GetIndex

17.8.1 Prose

The relation

$$\mathit{get\_index}(\overbrace{\mathbb{N}}^{\texttt{i}}, \overbrace{\mathcal{VEC}}^{\texttt{vec}}) \bigtimes \overbrace{\mathcal{VEC}}^{\texttt{v}_\texttt{i}}$$

reads the value $\texttt{v}_i$ from the vector of values `vec` at the index `i`.

17.8.2 Formally

$$\frac{\texttt{vec} \overset{\text{is}}{=} \texttt{v}_{0..k} \qquad \texttt{i} \leq k}{\mathit{get\_index}(\texttt{i}, \texttt{vec}) \xrightarrow{\text{eval}} \texttt{v}_\texttt{i}}$$

Notice that there is no rule to handle the case where the index is out of range — this is guaranteed by the type-checker not to happen. Specifically,

- `TypingRule.EGetArray` ensures that an index is within the bounds of the array being accessed via a check that the type of the index satisfies the type of the array size.
- Typing rules `TypingRule.LEDestructuring`, `TypingRule.PTuple`, and `TypingRule.LDTuple` use the same index sequences for the tuples involved and the corresponding lists of expressions.

If the rules listed above do not hold the type checker fails.

17.9 SemanticsRule.SetIndex

17.9.1 Prose

The relation

$$\mathit{set\_index}(\overbrace{\mathbb{N}}^{\texttt{i}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathcal{VEC}}^{\texttt{vec}}) \bigtimes \overbrace{\mathcal{VEC}}^{\texttt{res}}$$

overwrites the value at the given index `i` in a vector of values `vec` with the new value `v`.

17.9.2 Formally

$$\frac{\texttt{vec} \overset{\text{is}}{=} \texttt{u}_{0..k} \qquad \texttt{i} \leq k \qquad \texttt{res} \overset{\text{is}}{=} \texttt{w}_{0..k} \qquad \texttt{v} := \texttt{w}_{\texttt{i}} \qquad j \in \{0..k\} \setminus \{\texttt{i}\}.\ \texttt{w}_j = \texttt{u}_j}{\mathit{set\_index}(\texttt{i}, \texttt{v}, \texttt{vec}) \xrightarrow{\text{eval}} \texttt{res}}$$

Similar to *get_index*, there is no need to handle the out-of-range index case.

17.10 SemanticsRule.GetField

17.10.1 Prose

The relation

$$\mathit{get\_field}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathcal{REC}}^{\texttt{record}}) \bigtimes \mathbb{V}$$

retrieves the value corresponding to the field name `name` from the record value `record`.

17.10.2 Formally

$$\frac{\texttt{record} \overset{\text{is}}{=} \texttt{NV\_Record}(\mathit{field\_map})}{\mathit{get\_field}(\texttt{name}, \texttt{record}) \xrightarrow{\text{eval}} \mathit{field\_map}(\texttt{name})}$$

The type-checker ensures, via TypingRule.EGetRecordField, that the field `name` exists in `record`.

17.11 SemanticsRule.SetField

17.11.1 Prose

The relation

$$\mathit{set\_field}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathcal{REC}}^{\texttt{record}}) \bigtimes \mathcal{REC}$$

overwrites the value corresponding to the field name `name` in the record value `record` with the value `v`.

17.11.2 Formally

$$\frac{\texttt{record} \overset{\text{is}}{=} \texttt{NV\_Record}(\mathit{field\_map}) \qquad \mathit{field\_map}' := \mathit{field\_map}[\texttt{name} \mapsto \texttt{v}]}{\mathit{set\_field}(\texttt{name}, \texttt{v}, \texttt{record}) \xrightarrow{\text{eval}} \texttt{NV\_Record}(\mathit{field\_map}')}$$

The type-checker ensures that the field `name` exists in `record`.

17.12 SemanticsRule.DeclareLocalIdentifier

17.12.1 Prose

The relation

$$\mathit{declare\_local\_identifier}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{g}})$$

associates `v` to `name` as a local storage element in the environment `env` and returns the updated environment `new_env` with the execution graph consisting of a Write Effect to `name`.

17.12.2 Formally

$$\frac{\texttt{g} := \mathsf{WriteEffect}(\texttt{name}) \qquad \texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}[\texttt{name} \mapsto \texttt{v}]))}{\mathit{declare\_local\_identifier}(\texttt{env}, \texttt{name}, \texttt{v}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g})}$$

17.13 SemanticsRule.DeclareLocalIdentifierM

17.13.1 Prose

The relation

$$\mathit{declare\_local\_identifier\_m}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{x}}, \overbrace{\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}}^{\texttt{m}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

declares the local identifier `x` in the environment `env`, in the context of the value-graph pair `(v, g)`, and all of the following apply:

- `new_env` is the environment `env` modified to declare the variable `x` as a local storage element;
- `g1` is the execution graph resulting from the declaration of `x`;
- `g2` is the execution graph resulting from the ordered composition of `g` and `g1` with the `asl_data` edge.

17.13.2 Formally

$$\frac{\texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g}) \qquad \mathit{declare\_local\_identifier}(\texttt{env}, \texttt{x}, \texttt{v}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g1}) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \texttt{g1}}{\mathit{declare\_local\_identifier\_m}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

17.14 SemanticsRule.DeclareLocalIdentifierMM

17.14.1 Prose

The relation

$$\mathit{declare\_local\_identifier\_mm}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{x}}, \overbrace{(\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}})}^{\texttt{m}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{g2}})$$

declares the local identifier `x` in the environment `env`, in the context of the value-graph pair `(v, g)`, and all of the following apply:

- `new_env` is the environment `env` modified to declare the variable `x` as a local storage element;
- `g1` is the execution graph resulting from the declaration of `x`;
- `g2` is the execution graph resulting from the ordered composition of `g` and `g1` with the `asl_po` edge.

17.14.2 Formally

$$\frac{\mathit{declare\_local\_identifier\_m}(\texttt{env}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g1}) \qquad \texttt{g2} := \texttt{g} \xrightarrow{\texttt{asl\_po}} \texttt{g1}}{\mathit{declare\_local\_identifier\_mm}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g2})}$$

17.15 SemanticsRule.DeclareGlobal

17.15.1 Prose

The relation

$$\mathit{declare\_global}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathbb{E}}^{\texttt{env}}) \bigtimes \overbrace{\mathbb{E}}^{\texttt{new\_env}}$$

updates the environment `env` by mapping `name` to `v` as a global storage element.

17.15.2 Formally

$$\frac{\texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}[\texttt{name} \mapsto \texttt{v}], L^{\mathsf{denv}}))}{\mathit{declare\_global}(\texttt{name}, \texttt{v}, \texttt{env}) \xrightarrow{\text{eval}} \texttt{new\_env}}$$

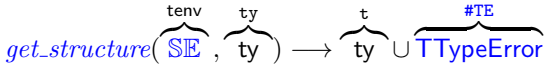
17.16 SemanticsRule.BaseValue

The relation

$$\mathit{base\_value}(\overbrace{(\overbrace{\mathbb{SE}}^{\mathsf{tenv}} \times \overbrace{\mathbb{DE}}^{\mathsf{denv}})}^{\texttt{env}}, \overbrace{\mathsf{ty}}^{\texttt{t}}) \bigtimes (\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

returns the *base value* of a type. The result is an error configuration if a dynamic error is detected.

Type Structure To obtain the base value of a type, we first obtain its *structure*, using the function



where `#TE` stands for a type error.

The structure of a type is the type that can hold the same set of values, but does not itself contain any other type names. This is essentially done by recursively replacing type names by their definition. For more information refer to TypingRule.Structure in ASL Typing Reference [6]. Since we assume the specification is well-typed (Section 3.1), *get_structure* returns a valid type.

17.16.1 Prose

The base value of the type `t` in the environment `env` is `v`, as well as the execution graph `g` that results from evaluating any of the side-effect-free expressions appearing in `t`, or an error, and one of the following applies:

- all of the following apply (BOOLEAN):
  - the structure of `t` is the Boolean type;
  - `v` is the native Boolean "true" value;
  - `g` is the empty graph.
- all of the following apply (REAL):
  - the structure of `t` is the real type;
  - `v` is the native real value 0;
  - `g` is the empty graph.
- all of the following apply (STRING):
  - the structure of `t` is the string type;
  - `v` is the native value for the empty string;
  - `g` is the empty graph.
- all of the following apply (BITVECTOR):
  - the structure of `t` is the bitvector with the length expression `e`;;
  - evaluating the side-effect-free expression `e` results in the native value `length` and execution graph $g^{/\!/\texttt{\#DE}}$;
  - `v` is the bitvector of length `length` where all bits are 0.

- all of the following apply (ENUM):
  - the structure of `t` is the enumeration type where the first identifier is $\texttt{id}_1$;
  - `l` is the literal associated with $\texttt{id}_1$ in the static environment;
  - `v` is the native value literal for `l`;
  - `g` is the empty graph.
- all of the following apply (UNCONSTRAINED_INTEGER):
  - the structure of `t` is that of the unconstrained integer;
  - `v` is the native value integer 0;
  - `g` is the empty graph.
- all of the following apply (WELL_CONSTRAINED_INTEGER):
  - the structure of `t` is that of the well-constrained integer where the first constraint is exact with the expression `e`;
  - $(\texttt{v}, \texttt{g})$ is the result of evaluating the side-effect-free expression `e`.
- all of the following apply (RECORD):
  - the structure of `t` is that of a record or an exception;
  - the base value of each field is obtained, and if any of the base values results in an error then the entire rule short-circuits with that error;
  - `v` is the native value record where each identifier in the record is mapped to its respective base value;
  - `g` is the parallel composition of the graphs resulting from the base value evaluation of all the fields.
- all of the following apply (TUPLE):
  - the structure of `t` is that of a tuple of types;
  - the base value of each type in the tuple is obtained, and if any of the base values results in an error then the entire rule short-circuits with that error;
  - `v` is the native value vector consisting of the base values in the order of the corresponding types of the tuple;
  - `g` is the parallel composition of the graphs resulting from the base value evaluation of all the tuple types.
- all of the following apply (ARRAY_LENGTH_GLOBAL_CONSTANT):
  - the structure of `t` is that of an array with length expression `length` and element type `v_ty`;
  - `length` is the value of a declared constant;

  - `length` is a variable expression with the variable name `x`;
  - the constant value for `x` in the static environment is the literal integer for n;
  - the base value of `v_ty` in `env` is $(\texttt{v\_elem}, \texttt{g})^{/\!/\texttt{\#DE}}$;
  - `v` is the native value vector of length n where each element is `v_elem`;

- all of the following apply (ARRAY_LENGTH_EXPRESSION):
  - the structure of `t` is that of an array with length expression `length` and element type `v_ty`;
  - `length` is not the value of a declared constant;
  - the base value of `v_ty` in `env` is $(\texttt{v\_elem}, \texttt{g})^{/\!/\texttt{\#DE}}$;
  - evaluating the side-effect-free expression `length` in the environment `env` results in $(\texttt{v\_length}, \texttt{g2})^{/\!/\texttt{\#DE}}$;
  - `v_length` is the native value integer for n;
  - `v` is the native value vector of length n where each element is `v_elem`;
  - `g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

17.16.2 Formally

Evaluating the inner expressions of the type `t` is done via the relation *eval_expr_sef*() 6.24. If evaluating an inner expression results in an error, there is no base value and an error configuration is returned.

BOOL

$$\frac{\mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bool}}{\mathit{base\_value}((\mathsf{tenv}, \mathsf{denv}), \mathtt{t}) \xrightarrow{\mathsf{eval}} (\overbrace{\texttt{Bool}(\texttt{TRUE})}^{\mathrm{v}}, \overbrace{\emptyset_{\mathsf{g}}}^{\mathrm{g}})}$$

REAL

$$\frac{\mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Real}}{\mathit{base\_value}((\mathsf{tenv}, \mathsf{denv}), \mathtt{t}) \xrightarrow{\mathsf{eval}} (\overbrace{\texttt{Real}(0)}^{\mathrm{v}}, \overbrace{\emptyset_{\mathsf{g}}}^{\mathrm{g}})}$$

STRING

$$\frac{\mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_String}}{\mathit{base\_value}((\mathsf{tenv}, \mathsf{denv}), \mathtt{t}) \xrightarrow{\mathsf{eval}} (\overbrace{\texttt{NV\_Literal}(\mathsf{L\_String}([\,]))}^{\mathrm{v}}, \overbrace{\emptyset_{\mathsf{g}}}^{\mathrm{g}})}$$

The base value of a bitvector is a bitvector native value consisting of a sequence of zeros of the length specified by the type (`e`). If the length is 0, the bitvector consists of

an empty sequence:

BITVECTOR

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{get\_structure}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \text{T\_Bits}(\texttt{e}, \_) \qquad \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} (\texttt{Int}(\texttt{length}), \texttt{g}) \;\; /\!/ \;\; \texttt{\#DE}}{\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\overbrace{\texttt{Bitvector}(\overbrace{0 \ldots 0}^{\texttt{length}})}^{\texttt{v}}, \texttt{g})}$$

The base value of an enumeration is obtained from its first declared literal. Accessing this literal is done via the constant_valuesmap in the global component of the static environment:

ENUM

$$\frac{\mathit{get\_structure}(\texttt{t}) \xrightarrow{\text{type}} \text{T\_Enum}(\texttt{id}_{1..k}) \qquad \texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \text{tenv} \overset{\text{is}}{=} (G^{\text{tenv}}, L^{\text{tenv}}) \qquad G^{\text{tenv}}.\text{constant\_values}(\texttt{id}_1) \overset{\text{is}}{=} \texttt{l}}{\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\overbrace{\texttt{NV\_Literal}(\texttt{l})}^{\texttt{v}}, \overbrace{\emptyset_{\texttt{g}}}^{\texttt{g}})}$$

INTEGER_UNCONSTRAINED

$$\frac{\mathit{get\_structure}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \text{T\_Int}(\text{Unconstrained})}{\mathit{base\_value}((\text{tenv}, \text{denv}), \texttt{t}) \xrightarrow{\text{eval}} (\overbrace{\texttt{Int}(0)}^{\texttt{v}}, \overbrace{\emptyset_{\texttt{g}}}^{\texttt{g}})}$$

INTEGER_CONSTRAINT_EXACT

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{get\_structure}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \text{T\_Int}(\text{WellConstrained}([\text{Constraint\_Exact}(\texttt{e})] + \_)) \qquad \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} C}{\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} C}$$

INTEGER_CONSTRAINT_RANGE

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{get\_structure}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \text{T\_Int}(\text{WellConstrained}([\text{Constraint\_Range}(\texttt{e}, \_)] + \_)) \qquad \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} C}{\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} C}$$

RECORD

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{get\_structure}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} L([i = 1..k : (\texttt{id}_i, \texttt{t}_i)]) \qquad L \in \{\text{T\_Record}, \text{T\_Exception}\} \qquad i = 1..k : \mathit{base\_value}(\texttt{env}, \texttt{t}_i) \xrightarrow{\text{eval}} (\texttt{v}_i, \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE}}{\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\overbrace{\texttt{NV\_Record}(\{i = 1..k : \texttt{id}_i \mapsto \texttt{v}_i\})}^{\texttt{v}}, \overbrace{\texttt{g}_1 \parallel \ldots \parallel \texttt{g}_k}^{\texttt{g}})}$$

TUPLE

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{get\_structure}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \text{T\_Tuple}([i = 1..k : \texttt{t}_i]) \qquad i = 1..k : \mathit{base\_value}(\texttt{env}, \texttt{t}_i) \xrightarrow{\text{eval}} (\texttt{v}_i, \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE}}{\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\overbrace{\texttt{NV\_Vector}(\texttt{v}_{1..k})}^{\texttt{v}}, \overbrace{\texttt{g}_1 \parallel \ldots \parallel \texttt{g}_k}^{\texttt{g}})}$$

The predicate `is_contant` checks whether the expression `e` is a variable declared as a constant.

NOTEVAR

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \text{tenv} \overset{\text{is}}{=} (G^{\text{tenv}}, L^{\text{tenv}}) \qquad \mathit{ast\_label}(\text{e}) \neq \text{E\_Var}}{\texttt{is\_contant}(\texttt{env}, \texttt{e}) \rightarrow \texttt{FALSE}}$$

EVAR

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \text{tenv} \overset{\text{is}}{=} (G^{\text{tenv}}, L^{\text{tenv}}) \qquad \mathit{ast\_label}(\text{e}) = \text{E\_Var} \qquad \text{e} \overset{\text{is}}{=} \text{E\_Var}(\text{x}) \qquad \text{b} := G^{\text{tenv}}.\text{constant\_values}(\text{x}) \neq \bot}{\texttt{is\_contant}(\texttt{env}, \texttt{e}) \rightarrow \text{b}}$$

ARRAY_LENGTH_GLOBAL_CONSTANT

$$\frac{\begin{array}{c}\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{get\_structure}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \text{T\_Array}(\texttt{length}, \texttt{v\_ty}) \\ \mathit{base\_value}(\texttt{env}, \texttt{v\_ty}) \xrightarrow{\text{eval}} (\texttt{v\_elem}, \texttt{g}) \;\; /\!/ \;\; \texttt{\#DE} \\ \texttt{is\_contant}(\texttt{env}, \texttt{length}) \rightarrow \texttt{TRUE} \qquad \texttt{length} \overset{\text{is}}{=} \text{E\_Var}(\text{x}) \\ \texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \text{tenv} \overset{\text{is}}{=} (G^{\text{tenv}}, L^{\text{tenv}}) \qquad G^{\text{tenv}}.\text{constant\_values}(\text{x}) \overset{\text{is}}{=} \text{L\_Int}(n)\end{array}}{\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\overbrace{\texttt{NV\_Vector}(i = 1..n : \texttt{v\_elem})}^{\texttt{v}}, \texttt{g})}$$

ARRAY_LENGTH_EXPRESSION

$$\frac{\begin{array}{c}\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \mathit{get\_structure}(\text{tenv}, \texttt{t}) \xrightarrow{\text{type}} \text{T\_Array}(\texttt{length}, \texttt{v\_ty}) \\ \mathit{base\_value}(\texttt{env}, \texttt{v\_ty}) \xrightarrow{\text{eval}} (\texttt{v\_elem}, \texttt{g1}) \;\; /\!/ \;\; \texttt{\#DE} \\ \texttt{is\_contant}(\texttt{length}) \rightarrow \texttt{FALSE} \\ \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{length}) \xrightarrow{\text{eval}} (\texttt{v\_length}, \texttt{g2}) \;\; /\!/ \;\; \texttt{\#DE} \\ \texttt{v\_length} \overset{\text{is}}{=} \texttt{Int}(n)\end{array}}{\mathit{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\overbrace{\texttt{NV\_Vector}(i = 1..n : \texttt{v\_elem})}^{\texttt{v}}, \overbrace{\texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2}}^{\texttt{g}})}$$

17.17 SemanticsRule.IsValOfType

17.17.1 Prose

The relation

$$\mathit{is\_val\_of\_type}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\text{ty}}^{\texttt{t}}) \bigtimes (\overbrace{\mathbb{B}}^{\texttt{b}} \times \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

checks whether the value v can be stored in a variable of type t in the environment env, resulting in a Boolean value b and execution graph g or a dynamic error.

This relation is used in the context of a asserted type conversion, which means the type-checker rule TypingRule.ATC was already applied, thus filtering cases where the type inferred for the converted expression does not type-satisfy t. The semantics takes this into account and only returns FALSE in cases where dynamic information is required.

One of the following applies:

- All of the following apply (BASIC):
  - t has the structure of a Boolean, a real, a string, or an enumeration;
  - b is TRUE;
  - g is the empty graph.
- All of the following apply (INT_UNCONSTRAINED):
  - t has the structure of the unconstrained integer;
  - b is TRUE;
  - g is the empty graph.
- All of the following apply (INT_WELLCONSTRAINED):
  - t has the structure of a well-constrained integer with constraints $c_{1..k}$;
  - v is the native value integer for n;
  - the evaluation of every constraint c_i with n in environment env yields a Boolean value b_i and an execution graph g_i[//#DE];
  - b is the Boolean disjunction of all Boolean values b_i, for $i = 1..k$;
  - g is the parallel composition of all execution graphs g_i, for $i = 1..k$;
- All of the following apply (RECORD):
  - t has the structure of a record or an exception, with a list of field names id_i, for $i = 1..k$, associated with types t_i, for $i = 1..k$;
  - the value of every field id_i in v is u_i, for $i = 1..k$,
  - the evaluation of *is_val_of_type* for every value u_i and corresponding type t_i, for $i = 1..k$, results in a Boolean b_i and execution graph g_i[//#DE];
  - b is the Boolean conjunction of all Boolean values b_i, for $i = 1..k$;
  - g is the parallel composition of all execution graphs g_i, for $i = 1..k$; of the constraints.
- All of the following apply (TUPLE):
  - t has the structure of a tuple with types t_i, for $i = 1..k$;
  - the value at every index $i = 1..k$ of v is u_i, for $i = 1..k$,

  - the evaluation of *is_val_of_type* for every value $\mathtt{u}_i$ and corresponding type $\mathtt{t}_i$, for $i = 1..k$, results in a Boolean $\mathtt{b}_i$ and execution graph $\mathtt{g}_i{}^{/\!/\mathtt{\#DE}}$;
  - $\mathtt{b}$ is the Boolean conjunction of all Boolean values $\mathtt{b}_i$, for $i = 1..k$;
  - $\mathtt{g}$ is the parallel composition of all execution graphs $\mathtt{g}_i$, for $i = 1..k$; of the constraints.

- All of the following apply (ARRAY):
  - $\mathtt{t}$ has the structure of an array with length expression $\mathtt{t}$ and element type $\mathtt{t1}$;
  - evaluating the side-effect-free expression $\mathtt{e}$ in environment $\mathtt{env}$ results in the native value integer for k and execution graph $\mathtt{g}$;
  - obtaining the values at indices $i = 1..k$ from $\mathtt{v}$ result in $\mathtt{v}_i$;
  - evaluating *is_val_of_type* for $\mathtt{v}_i$ and $\mathtt{t1}$, for $i = 1..k$, all result in Boolean values $\mathtt{b}_i$ and execution graphs $\mathtt{g}_i{}^{/\!/\mathtt{\#DE}}$;
  - $\mathtt{b}$ is the Boolean conjunction of all Boolean values $\mathtt{b}_i$, for $i = 1..k$;
  - $\mathtt{g}$ is the parallel composition of all execution graphs $\mathtt{g}_i$, for $i = 1..k$; of the constraints.

17.17.2 Formally

BASIC (BOOL, REAL, STRING, ENUM)

$$\frac{\mathit{ast\_label}(\mathit{get\_structure}(\mathtt{env}, \mathtt{t})) \in \{\mathsf{T\_Bool}, \mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Enum}\} \qquad \mathtt{g} := \emptyset_{\mathtt{g}}}{\mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{v}, \mathtt{t}) \xrightarrow{\mathsf{eval}} (\mathtt{TRUE}, \mathtt{g})}$$

INT_UNCONSTRAINED

$$\frac{\mathit{get\_structure}(\mathtt{env}, \mathtt{t}) \overset{\mathsf{is}}{=} \mathsf{T\_Int}(\mathsf{Unconstrained}) \qquad \mathtt{g} := \emptyset_{\mathtt{g}}}{\mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{v}, \mathtt{t}) \xrightarrow{\mathsf{eval}} (\mathtt{TRUE}, \mathtt{g})}$$

To handle *well-constrained integers* (integers with a non-empty list of constraints), we introduce the helper relation

$$\mathit{int\_constraint\_sat}\Big(\overbrace{\mathbb{E}}^{\mathtt{env}}, \overbrace{\mathsf{int\_constraint}}^{\mathtt{c}}, \overbrace{\mathbb{Z}}^{n}\Big) \bigtimes \Big(\overbrace{\mathbb{B}}^{\mathtt{b}} \times \overbrace{\mathcal{G}}^{\mathtt{g}}\Big)$$

which checks whether the integer value n *meets the constraint* $\mathtt{c}$ (that is, whether n is within the range of values defined by $\mathtt{c}$) in the environment $\mathtt{env}$ and returns a Boolean answer $\mathtt{b}$ and the execution graph $\mathtt{g}$ resulting from evaluating the expressions appearing

in c:

CONSTRAINT_EXACT_SAT

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} (\texttt{Int}(m), \texttt{g}) \;\; /\!/ \;\; \texttt{\#DE} \qquad \texttt{b} := m = n}{\mathit{int\_constraint\_sat}(\texttt{env}, \mathsf{Constraint\_Exact}(\texttt{e}), n) \xrightarrow{\text{eval}} (\texttt{b}, \texttt{g})}$$

CONSTRAINT_RANGE_SAT

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} (\texttt{Int}(a), \texttt{g1}) \;\; /\!/ \;\; \texttt{\#DE} \qquad \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e2}) \xrightarrow{\text{eval}} (\texttt{Int}(b), \texttt{g2}) \;\; /\!/ \;\; \texttt{\#DE} \qquad \texttt{b} := \mathsf{choice}(a \leq n \wedge n \leq b, \texttt{TRUE}, \texttt{FALSE}) \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2}}{\mathit{int\_constraint\_sat}(\texttt{env}, \mathsf{Constraint\_Range}(\texttt{e1}, \texttt{e2}), n) \xrightarrow{\text{eval}} (\texttt{b}, \texttt{g})}$$

Finally, we can check whether an integer value satisfies any of the constraints:

INT_WELLCONSTRAINED

$$\frac{\mathit{get\_structure}(\texttt{env}, \texttt{t}) \overset{\text{is}}{=} \mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{c}_{1..k})) \qquad \texttt{v} \overset{\text{is}}{=} \texttt{Int}(n) \qquad i = 1..k : \mathit{int\_constraint\_sat}(\texttt{env}, \texttt{c}_i, n) \xrightarrow{\text{eval}} (\texttt{b}_i, \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \qquad \texttt{b} := \bigvee_{i=1}^{k} \texttt{b}_i \qquad \texttt{g} := \|_{i=1}^{k} \texttt{g}_i}{\mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{b}, \texttt{g})}$$

RECORD

$$\frac{\mathit{get\_structure}(\texttt{env}, \texttt{t}) \overset{\text{is}}{=} L(i = 1..k : \texttt{id}_i \mapsto \texttt{t}_i) \qquad L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \qquad i = 1..k : \mathit{get\_field}(\texttt{id}_i, \texttt{v}) \xrightarrow{\text{eval}} \texttt{u}_i \qquad i = 1..k : \mathit{is\_val\_of\_type}(\texttt{env}, \texttt{u}_i, \texttt{t}_i) \xrightarrow{\text{eval}} (\texttt{b}_i, \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \qquad \texttt{b} := \bigwedge_{i=1}^{k} \texttt{b}_i \qquad \texttt{g} := \|_{i=1}^{k} \texttt{g}_i}{\mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{b}, \texttt{g})}$$

TUPLE

$$\frac{\mathit{get\_structure}(\texttt{env}, \texttt{t}) \overset{\text{is}}{=} \mathsf{T\_Tuple}(i = 1..k : \texttt{t}_i) \qquad i = 1..k : \mathit{get\_index}(i, \texttt{v}) \xrightarrow{\text{eval}} \texttt{u}_i \qquad i = 1..k : \mathit{is\_val\_of\_type}(\texttt{env}, \texttt{u}_i, \texttt{t}_i) \xrightarrow{\text{eval}} (\texttt{b}_i, \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \qquad \texttt{b} := \bigwedge_{i=1}^{k} \texttt{b}_i \qquad \texttt{g} := \|_{i=1}^{k} \texttt{g}_i}{\mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{b}, \texttt{g})}$$

ARRAY

$$\frac{\begin{array}{c} \mathit{get\_structure}(\texttt{env}, \texttt{t}) \overset{\text{is}}{=} \mathsf{T\_Array}(\texttt{e}, \texttt{t1}) \\ \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} (\texttt{Int}(k), \texttt{g}) \qquad i = 1..k : \mathit{get\_index}(i, \texttt{v}) \xrightarrow{\text{eval}} \texttt{v}_i \\ i = 1..k : \mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}_i, \texttt{t1}) \xrightarrow{\text{eval}} (\texttt{b}_i, \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \\ \texttt{b} := \bigwedge_{i=1}^{k} \texttt{b}_i \qquad \texttt{g} :=\|_{i=1}^{k} \texttt{g}_i \end{array}}{\mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{b}, \texttt{g})}$$

Notice that these rules cover all types, including named types ($\mathsf{T\_Named}$), since *get_structure* replaces named types by their type definitions. Underconstrained integers (integers with an empty set of constraints) cannot appear as a type, since ASL syntax does not allow the following:

- Declaring an underconstrained integer as a variable,
- Declaring an alias to an underconstrained integer type, and
- Declaring an underconstrained integer in a compound type.

17.18 SemanticsRule.UnopValues

The function

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

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

The evaluation is defined in terms of the static evaluation function

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

which is defined in the ASL Typing Reference [6] (see TypingRule.UnopLiterals).

17.18.1 Prose

One of the following applies:

- All of the following apply (NEGATE_INT):
  - `op` is `NEG` and `v` is a literal integer for n;
  - statically evaluating `NEG` on the literal integer for n yields the literal integer for m;
  - `w` is the native integer value for m.
- All of the following apply (NEGATE_REAL):

  - `op` is `NEG` and `v` is a literal real for p;
  - statically evaluating `NEG` on the literal real for n yields the literal real for q;
  - `w` is the native real value for q.

- All of the following apply (NOT_BOOL):
  - `op` is `BNOT` and `v` is a literal Boolean for b;
  - statically evaluating `BNOT` on the literal Boolean for b yields the literal real for c;
  - `w` is the native Boolean value for c.

- All of the following apply (NOT_BITS):
  - `op` is `NOT` and `v` is a literal bitvector for `bits`;
  - statically evaluating `NOT` on the literal bitvector for `bits` yields the literal bitvector for c;
  - `w` is the native bitvector value for c.

17.18.2 Formally

NEGATE_INT

$$\frac{\mathit{unop\_literals}(\texttt{NEG}, \mathsf{L\_Int}(n)) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(m)}{\mathit{unop}(\overbrace{\texttt{NEG}}^{\texttt{op}}, \overbrace{\texttt{Int}(n)}^{\texttt{v}}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Int}(m)}^{\texttt{w}}}$$

NEGATE_REAL

$$\frac{\mathit{unop\_literals}(\texttt{NEG}, \mathsf{L\_Real}(p)) \xrightarrow{\mathsf{type}} \mathsf{L\_Real}(q)}{\mathit{unop}(\overbrace{\texttt{NEG}}^{\texttt{op}}, \overbrace{\texttt{Real}(p)}^{\texttt{v}}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Real}(q)}^{\texttt{w}}}$$

NOT_BOOL

$$\frac{\mathit{unop\_literals}(\texttt{BNOT}, \mathsf{L\_Bool}(b)) \xrightarrow{\mathsf{type}} \mathsf{L\_Real}(c)}{\mathit{unop}(\overbrace{\texttt{BNOT}}^{\texttt{op}}, \overbrace{\texttt{Bool}(b)}^{\texttt{v}}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Bool}(c)}^{\texttt{w}}}$$

NOT_BITS

$$\frac{\mathit{unop\_literals}(\texttt{NOT}, \mathsf{L\_Bitvector}(\texttt{bits})) \xrightarrow{\mathsf{type}} \mathsf{L\_Bitvector}(c)}{\mathit{unop}(\overbrace{\texttt{NOT}}^{\texttt{op}}, \overbrace{\texttt{Bitvector}(\texttt{bits})}^{\texttt{v}}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Bitvector}(c)}^{\texttt{w}}}$$

17.19 SemanticsRule.BinopValues

The function

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

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

The evaluation is defined in terms of the static evaluation function

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

which is defined in the ASL Typing Reference [6] (see TypingRule.BinopLiterals, Chapter "Static Evaluation").

17.19.1 Prose

One of the following applies:

- All of the following apply (INT_ARITH):
  - `v1` is a literal integer for a;
  - `v2` is a literal integer for b;
  - statically evaluating `op` on `v1` and `v2` yields the literal integer for c;
  - `r` is the native integer value for c.
- All of the following apply (INT_REL):
  - `v1` is a literal integer for a;
  - `v2` is a literal integer for b;
  - statically evaluating `op` on `v1` and `v2` yields the literal Boolean for c;
  - `r` is the native Boolean value for c.
- All of the following apply (INT_ERROR):
  - `v1` is a literal integer for a;
  - `v2` is a literal integer for b;
  - statically evaluating `op` on `v1` and `v2` yields a type error with message m;
  - the result is a dynamic error with message m.
- All of the following apply (BOOL_OKAY):
  - `v1` is a literal Boolean for a;
  - `v2` is a literal Boolean for b;
  - statically evaluating `op` on `v1` and `v2` yields the literal Boolean for c;
  - `r` is the native Boolean value for c.
- All of the following apply (BOOL_ERROR):
  - `v1` is a literal Boolean for a;
  - `v2` is a literal Boolean for b;
  - statically evaluating `op` on `v1` and `v2` yields a type error with message m;

    - the result is a dynamic error with message m.

- All of the following apply (REAL_ARITH_OKAY):
    - `v1` is a literal real for a;
    - `v2` is a literal real for b;
    - statically evaluating `op` on `v1` and `v2` yields the literal real for c;
    - `r` is the native real value for c.
- All of the following apply (REAL_REL_OKAY):
    - `v1` is a literal real for a;
    - `v2` is a literal real for b;
    - statically evaluating `op` on `v1` and `v2` yields the literal Boolean for c;
    - `r` is the native Boolean value for c.
- All of the following apply (REAL_REL_ERROR):
    - `v1` is a literal real for a;
    - `v2` is a literal real for b;
    - statically evaluating `op` on `v1` and `v2` yields a type error with message m;
    - the result is a dynamic error with message m.
- All of the following apply (BITVECTOR_REL_OKAY):
    - `v1` is a literal bitvector for a;
    - `v2` is a literal bitvector for b;
    - statically evaluating `op` on `v1` and `v2` yields the literal Boolean for c;
    - `r` is the native Boolean value for c.
- All of the following apply (BITVECTOR_REL_ERROR):
    - `v1` is a literal bitvector for a;
    - `v2` is a literal bitvector for b;
    - statically evaluating `op` on `v1` and `v2` yields a type error with message m;
    - the result is a dynamic error with message m.
- All of the following apply (BITVECTOR_BITS_OKAY):
    - `v1` is a literal bitvector for a;
    - `v2` is a literal bitvector for b;
    - statically evaluating `op` on `v1` and `v2` yields the literal bitvector for c;
    - `r` is the native bitvector value for c.

- All of the following apply (BITVECTOR_BITS_ERROR):
  - `v1` is a literal bitvector for a;
  - `v2` is a literal bitvector for b;
  - statically evaluating `op` on `v1` and `v2` yields a type error with message m;
  - the result is a dynamic error with message m.

17.19.2 Formally

INT_ARITH

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

INT_REL

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Int}(a), \mathsf{L\_Int}(b)) \xrightarrow{\mathsf{type}} \mathsf{L\_Bool}(c)}{\mathit{binop}(\mathsf{op}, \overbrace{\texttt{Int}(a)}^{\texttt{v1}}, \overbrace{\texttt{Int}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Bool}(c)}^{\texttt{r}}}$$

INT_ERROR

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Int}(a), \mathsf{L\_Int}(b)) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(m)}{\mathit{binop}(\mathsf{op}, \overbrace{\texttt{Int}(a)}^{\texttt{v1}}, \overbrace{\texttt{Int}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(m)}$$

BOOL_OKAY

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

BOOL_ERROR

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Bool}(a), \mathsf{L\_Bool}(b)) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(m)}{\mathit{binop}(\mathsf{op}, \overbrace{\texttt{Bool}(a)}^{\texttt{v1}}, \overbrace{\texttt{Bool}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(m)}$$

REAL_ARITH_OKAY

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

REAL_ARITH_ERROR

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Real}(a), \mathsf{L\_Real}(b)) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(m)}{\mathit{binop}(\mathsf{op}, \overbrace{\texttt{Real}(a)}^{\texttt{v1}}, \overbrace{\texttt{Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(m)}$$

REAL_REL_OKAY

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Real}(a), \mathsf{L\_Real}(b)) \xrightarrow{\mathsf{type}} \mathsf{L\_Bool}(c)}{\mathit{binop}(\mathsf{op}, \overbrace{\texttt{Real}(a)}^{\texttt{v1}}, \overbrace{\texttt{Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Bool}(c)}^{\texttt{r}}}$$

REAL_REL_ERROR

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Real}(a), \mathsf{L\_Real}(b)) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(m)}{\mathit{binop}(\mathsf{op}, \overbrace{\texttt{Real}(a)}^{\texttt{v1}}, \overbrace{\texttt{Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(m)}$$

BITVECTOR_REL_OKAY

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \texttt{Bitvector}(a), \texttt{Bitvector}(b)) \xrightarrow{\mathsf{type}} \mathsf{L\_Bool}(c)}{\mathit{binop}(\mathsf{op}, \overbrace{\mathsf{L\_Bitvector}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Bool}(c)}^{\texttt{r}}}$$

BITVECTOR_REL_ERROR

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Bitvector}(a), \mathsf{L\_Bitvector}(b)) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(m)}{\mathit{binop}(\mathsf{op}, \overbrace{\mathsf{L\_Bitvector}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(m)}$$

BITVECTOR_BITS_OKAY

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Bitvector}(a), \mathsf{L\_Bitvector}(b)) \xrightarrow{\mathsf{type}} \mathsf{L\_Bitvector}(c)}{\mathit{binop}(\mathsf{op}, \overbrace{\mathsf{L\_Bitvector}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \overbrace{\texttt{Bitvector}(c)}^{\texttt{r}}}$$

BITVECTOR_BITS_ERROR

$$\frac{\mathit{binop\_literals}(\mathsf{op}, \mathsf{L\_Bitvector}(a), \mathsf{L\_Bitvector}(b)) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(m)}{\mathit{binop}(\mathsf{op}, \overbrace{\mathsf{L\_Bitvector}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(m)}$$

Bibliography

[1] *Arm Architecture Reference Manual for A-profile architecture*, 2023.

[2] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat, 2016.

[3] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Transactions on Programming Languages and Systems*, 43(2):8:1–8:54, 2021.

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2):7:1–7:74, 2014.

[5] Arm Architecture Technology Group. *ASL Abstract Syntax Reference*. 2024.

[6] Arm Architecture Technology Group. *ASL Typing Reference*. 2024.

[7] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., USA, 1992.