Formal Semantics of SPARK

Dynamic Semantics

Program Validation Ltd.

28th. October 1994

PVL/SPARK_DEFN/DYNAMIC/V1.4

Author:	Ian O'Neill
Authorisation:	
Type:	Deliverable
Status:	Approved
Circulation:	Restricted

Document Purpose

The Dynamic Semantics, or evaluation rules, of the SPARK Ada-subset are formally defined using inference rules in the Structured Operational Semantics style.

Document History

Version	Date	Who	Why
0.1	01.02.93	ION	Initial draft version
1.0	08.02.93	ION	First delivered version
1.1	07.04.93	ION	Second delivered version
1.2	02.03.94	ION	First peer review version
1.3	03.03.94	ION	Major typo's corrected
1.4	28.10.94	ION	Revision following Peer Review

Formal Semantics of SPARK

Dynamic Semantics

Program Validation Ltd.

28th. October 1994

PVL/SPARK_DEFN/DYNAMIC/V1.4

Contents

Chapter 1

Introduction

This document is a formal dynamic semantics of the SPARK annotated subset of the Ada programming language. It should be read in conjunction with the Static Semantics of SPARK, which describes the construction of the static environment used in this document and the static well-formedness constraints which apply to a SPARK program text. Both documents employ the same abstract syntax and notational conventions, based (non-strictly) on the Z notation. This Chapter reviews the construction of the document and the links with the Static Semantics document. We recommend you read this chapter first, as an introduction to the rest of the document; there are some suggestions at the end of the chapter for possible reading orders.

1.1 Some Remarks on SPARK

SPARK is an annotated subset of Ada designed to eliminate the ambiguities and insecurities of the full Ada language, in order to facilitate rigorous design and reasoning about Ada program behaviour. The main design considerations used in defining this subset of full Ada were:

- Logical soundness;
- (Non-)Complexity of its formal definition;
- Expressive power;
- Security;
- Verifiability;
- Bounded space and time requirements.

Ada features excluded from the SPARK subset include tasking, exceptions, generic units and access types. Scoping and visibility rules across package boundaries are much tighter, and overloading is avoided as far as possible; the use clause is eliminated as part of this.

Type aliasing and anonymous types are not supported, nor are default values in record declarations or default subprogram parameters. Finally, a "well-formed" control structure is enforced through the elimination of `goto` statements and strict rules on the placement and use of loop exit statements.

In addition to the removal or restriction of Ada features which were felt to compromise one or more of the design goals too strongly, SPARK has a system of annotations, which are ignored by an Ada compiler but used to enforce additional static semantic rules of SPARK. These annotations control, for instance, the visibility of global variables which may be read from and/or written to by a subprogram, and provide some formal documentation of the code's intended behaviour which may be checked for conformance automatically with tool support.

Some important properties of the SPARK annotated subset are:

- Absence of side-effects from expressions;
- No (write) aliasing of variables;
- No redeclaration;
- Semantics of assignment is equality;
- No recursive subprograms.

1.2 Form of the Semantics

The dynamic semantics of SPARK are presented in this document as a collection of inference rules, forming an "evaluation semantics" for the language. The aim has been to steer a course between the excessive detail of a computation semantics and the high level of abstraction often present in a denotational semantics.

There are two main ways in which this dynamic semantics omits some detail applicable to SPARK through its Ada parentage. The first is in the treatment of real numbers and real arithmetic. Ada has a sophisticated model of both floating and fixed point real types, but it was decided early on that this was an aspect of the parent language which had already received much attention elsewhere, and so will not be dwelt upon in this treatment. The second omission is in the area of machine representation, in particular the semantic effects of the use of Ada representation clauses which can modify the internal storage representation of data objects.

The main ingredients of this document are described briefly below.

Abstract Syntax The same Abstract Syntax is used in this document as in the Static Semantics. We have presented the same examples of concrete syntax where appropriate, and used the same order of presentation of the terms of the Abstract Syntax. The key elements of interest for the dynamic semantics are:

Category	Description
Name	Names
Exp	Expressions
Decl	Declarations
Stmt	Statements
CUnit	Compilation units

In addition, components of the environment constructed by the static semantics – including types, variable declarations which affect the store used by the Dynamic Semantics and constants – are of interest and are referenced in this document. For those syntax components whose effect on the Dynamic Semantics is indirect – for example, subtype definitions – we have retained the relevant chapter to enhance the correspondence with the Static Semantics document, but collapsed such sections down to the components of interest to the Dynamic Semantics alone. For ease of reference, we have used the same chapter and section structure in the subsequent chapters of this document as are used in the Static Semantics, except where it has been possible to eliminate a section in this document (to avoid undue repetition, for instance) or to replace it by material on a strongly related topic.

Dynamic Environment We assume the static well-formedness of SPARK texts in presenting this Dynamic Semantics; the reader is referred to the companion document on the Static Semantics of SPARK for further details. In this volume, we construct a dynamic environment, also called *Env* (see next chapter), in which we construct various structures similar to those constructed by the Static Semantics. We use the dynamic environment to reference objects of relevance, such as salient features of types and subtypes. The description of the dynamic environment given in this document replaces the corresponding static environment of the companion volume; an eventual aim may be to unite the two environments into a single one, as a step towards providing a single, unified semantics of SPARK.

Values and Store As well as including a description of the structure of the store after the description of the Static Environment constructs, we have extended the set of values described in the next Chapter to cover all object values which may be encountered in the dynamic evaluation of well-formed SPARK texts.

Evaluation Rules The Dynamic Semantics is presented by giving one or more evaluation rules defining the evaluation predicate for each relevant component of the Abstract Syntax of SPARK. The evaluation of a term of the abstract syntax for a given static environment and store is defined with respect to the evaluation of its subcomponents as far as this is possible (though with some easing of this constraint, notably in the case of the evaluation of loop statements). Each Chapter reviews any predicates introduced for this purpose. Each such evaluation rule takes the form:

$$\begin{array}{l}
\forall \; declarations \\
| \\
\quad side\ \ conditions \\
\bullet \\
\quad premise_1 \\
\quad \ldots \\
\quad premise_n \\
\hline
\quad conclusion
\end{array} \qquad \text{(RuleName)}$$

The *premises* generally involve evaluation of the component objects of the element of the Abstract Syntax under evaluation, while the *conclusion* describes what value this element of the Abstract Syntax may be concluded to have from the evaluation of its constituent parts. Where *side conditions* are present, these frequently involve either the gathering of relevant information from the environment for use in the rule, or "dynamic well-formedness" constraints on the values derived in order to be able to infer that certain Ada run-time exceptions will not be raised. The "RuleName" associated with each inference rule is for ease of reference only and does not affect the meaning of the rules so presented. The evaluation rules for the various terms of the abstract syntax form a mutually-recursive set, with the intention that each of the relations defined by the inference rules is the "smallest" relation satisfying all of the rules given.

Notation As noted earlier, we have adopted a similar style of Z notation in this document to that employed in the corresponding Static Semantics document. Clearly, the inference rules cannot be regarded as a standard part of the Z notation; equally, we have bent certain other conventions, using mutual recursion in our syntax descriptions for instance. Again, as with the Static Semantics, the order of presentation of concepts has been dictated principally by the constraints of ease of understanding, with the result that the order of declaration-before-use required by many tools has frequently been overridden.

1.3 Technical Issues

Below we summarise a number of technical issues of note for the dynamic semantics in particular.

Evaluation Order Note that the abstract syntax of SPARK used in this definition takes no direct account of explicit parenthesization; thus, no distinction is made between `A+B+C` and `(A+B)+C`. (Though the expression `A+(B+C)` is distinguished: left-to-right association is the Ada default: see [LRM, §4.5].) Ada allows many expression forms to be evaluated *"in some order that is not defined by the language"*. While SPARK removes certain of the insecurities that this laxity over evaluation order permits (e.g. by banning functions with side-effects), it is still the case that where minimal parenthesization has been employed, different Ada implementations are allowed to behave in different ways, for the sake of optimization. (Use of parentheses can, however, limit this problem, since the compiler must then ensure that any optimization will give similar exception-raising behaviour as if the strict order implied by the programmer's parentheses were used.)

Real Arithmetic This release of the semantics has been the subject of independent review (under a "Peer Review" contract, for the Defence Research Agency, Malvern). As a part of this review, the subject of Ada real arithmetic (both fixed and floating point) was considered. This is an area which presents variuos difficulties, but Ada 9X does hold out the hope of an improved definition of real arithmetic, with which it should be possible to make greater progress; beyond indications of the difficulties involved in real arithmetic within the text, we have therefore made no serious attempt to formalise Ada 83's real arithmetic here.

Non-Determinism Given the above, and the rounding laxity allowed in principle by full Ada (which is inherited by SPARK), the semantics of code containing loops performing real arithmetic calculations is in principle non-deterministic in a way which, in practice, one would hope to be able to avoid with a proper definition of the real arithmetic used in an actual implementation. Indeed, one suggestion arising from the Peer Review process referred to above was to leave the specification of real arithmetic behaviour to an implementation Annex; we believe this proposal has considerable merit. (A further consequence of the existing non-determinism is that the semantics as written is not in a form to support reasoning about program equivalence, though this was not a goal of the formal definition in practice.)

Scope The following Ada constructs are "tolerated" within the SPARK subset as policed by the SPARK Examiner, but do not form a part of the formal definition:

- Address clauses;
- Representation clauses;
- Code insertions;
- Pragmas.

Status Because of the desire to base the Dynamic Semantics as closely as possible on the same Abstract Syntax for SPARK as used in the Static Semantics and to exploit the synergy by employing as much of the commonality of the static environment constructed by the Static Semantics as possible, items excluded by the Static Semantics document are similarly excluded from this Dynamic Semantics, unless independent progress has been possible.

1.4 Suggested Reading Order

After completion of this chapter, our recommended reading order is that chapter 2 should be studied next: this chapter describes the "dynamic environment", which is somewhat analogous to the environment defined in the Static Semantics document, but skewed instead to inclusion of only those elements of use to execution of a statically well-formed SPARK program. The store, used to keep track of values associated with variables, is also described in chapter 2: the model of store used herein emphasises the way in which static allocation of storage space may in principle be achieved in the SPARK subset.

Following on from chapter 2, the two most interesting chapters from the point of view of the dynamic semantics are chapter 6 (expressions) and chapter 11 (statements). Chapter 6 is also supported by a number of special sorts of expression, such as names (chapter 5) – which are expressions that may appear on the left-hand side of an assignment statement – and attributes, which provide ways of accessing type information (e.g. first element of a range) in a parameterized fashion. After these, chapter 17, and chapters 12-15 provide descriptions of the larger-scale SPARK constructs, including packages and the main program. Other chapters are devoted to the various forms of declaration in SPARK; all chapter and section numbering is designed to be identical, as far as possible, to that employed in the Static Semantics.

1.5 Acknowledgements

This work on the formal semantics of SPARK was supported by the United Kingdom Defence Research Agency (DRA), Malvern, under the guidance of Colin O'Halloran and performed by Program Validation Limited. We also acknowledge the contributions made by the Peer Reviewers of these two volumes, who provided many helpful comments; the reviewers (in name alphabetical order) were: Janet Barnes (John Barnes Informatics); John Barnes (John Barnes Informatics); Ian Cottam (Manchester Informatics); John Fitzgerald (Manchester Informatics); Peter Froome (Adelard); Paul Gardiner (Formal Systems (Europe)); Mike Goldsmith (Formal Systems (Europe)); Trevor Jennings (EDS Defence); Claire Jones (Adelard); Cliff Jones (Adelard and Manchester Informatics); Brian Wichmann (National Physical Laboratory); and Jim Woodcock (Formal Systems (Europe)).

Chapter 2

Environment Definitions

This dynamic semantics constructs and makes use of a "dynamic environment", constructed during traversal of the Abstract Syntax representation of the SPARK program text. We include details of the structure of this dynamic environment in this chapter. This chapter also defines the basic sets and values used in the dynamic semantics in later chapters, and describes the structure of the store, which is used to associate variable references in a SPARK text with their values during "execution".

First, a preliminary section introduces the basic sets which are needed in the environment.

2.1 Basic Values

The basic values of SPARK are the ones which appear in the Abstract Syntax of the language. (The appearance of these values in the concrete syntax of the language is defined by the lexical rules of SPARK. We do not discuss the lexical rules in this document).

Identifiers Identifiers belong to the set *Id*.

Numbers Numeric values are either integers, from the set $\mathbb{Z}$, or rationals from the set Real — following the traditional usage the term *real* is used instead of rational.

Characters Characters, from the set *Char*, may be used as literals in SPARK programs.

Names All visible declarations in a SPARK program are either identified by an *Id*, if they are directly visible, or by a pair of *Id*, if they are visible by selection. The set *IdDot* is used for names such as procedure and types both in the Abstract Syntax and the dictionary.

$$IdDot ::= id\langle\!\langle Id \rangle\!\rangle \mid dot\langle\!\langle Id \times Id \rangle\!\rangle$$

2.2 Expression Values

A set of values is required which includes all the values which can be given to expressions in SPARK, i.e. all the values which need to be evaluated in the dynamic semantics. The scalar values are integers, real numbers and enumeration literals. The composite values are records and arrays. Scalar values may also be "undefined". Finally, we include a range value to allow for value ranges (from type-marks, for instance).

$$\begin{array}{rl} Val ::= & intval\langle\langle \mathbb{Z} \rangle\rangle \\ | & realval\langle\langle \mathrm{Real} \rangle\rangle \\ | & enumval\langle\langle IdDot \rangle\rangle \\ | & recval\langle\langle Id \nrightarrow Val \rangle\rangle \\ | & arrval\langle\langle Array\_Value \rangle\rangle \\ | & rngval\langle\langle \mathbb{P}\ Val \rangle\rangle \\ | & undefined \end{array}$$

Notes

1. The set *Val* does not include a character value since the predefined Character type is an enumeration type (see Appendix A of the Static Semantics document). Elements of *Char* only appear as character literals.

2. The form of an array value used in the above is defined by:

$$\begin{array}{|l} \textit{Array\_Value} \\ \hline lo, hi : \mathbb{Z} \\ arr : \mathbb{Z} \nrightarrow Val \\ \hline lo \leq hi \\ \mathrm{dom}\ arr = lo \mathinner{\ldotp\ldotp} hi \\ \hline \end{array}$$

   Note that this only appears to define a form for one-dimensional arrays, when this is not a constraint of SPARK. We shall regard two- and higher-dimensional arrays as arrays of arrays for dynamic semantics purposes. (The fact that Ada distinguishes between them is not a major problem: there can be no confusion over what is meant, as static semantic checks will preclude accidental assignment of one array form to another.)

3. Enumeration literal values can be either simple identifiers, or pairs (for enumeration literals from other packages, for instance). This can never cause a conflict: if the enumeration type is within the current scope, simple identifiers will suffice (and be necessary) for all objects of the enumeration type, while if the enumeration type is not *directly* visible in the current scope but is instead being accessed from another (with'd) package, all objects of the type will need to be declared via the $K.T$ form.

2.3 The Environment

The environment constructed by the dynamic semantics is considerably simpler than that constructed by the static semantics. This is because we assume in this document that all SPARK texts to which these dynamic semantic evaluation rules are to be applied will first have been "checked" for conformance to the static semantic constraints.

At the top-level, a SPARK program is a collection of compilation units, which must be presented in some appropriate order (in order to pass the static semantic well-formedness requirements) and include a single main program. It is the case, however, that the same identifier may appear multiple times within a well-formed SPARK text, so our dynamic environment must have some structure to represent the nesting of scopes present in a SPARK program. We first define:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Dict}} \\
\hline
\textit{withs} : \operatorname{seq} \textit{Id} \\
\textit{const} : \textit{Id} \pfun \textit{Val} \\
\textit{type} : \textit{Id} \pfun \textit{TypCon} \\
\textit{var} : \textit{Id} \pfun \textit{IdDot} \\
\textit{procs} : \textit{Id} \pfun ((\operatorname{seq} \textit{FormalParam}) \times \textit{Stmt}) \\
\textit{funs} : \textit{Id} \pfun ((\operatorname{seq} \textit{FormalParam}) \times \textit{Stmt} \times \textit{Exp} \times \textit{IdDot}) \\
\hline
\end{array}
$$

This represents one "layer" of scope: it lists the (possibly empty) collection of packages which are with'd to the current unit; provides a mapping from some set of identifiers declared as constants in this scope to their values; provides a mapping from the scope's type identifiers to the corresponding types (represented by *TypCon* – discussed in Chapter 3); records the variables declared in this scope and their type-marks; and records information about the procedures and functions defined in this scope. For a procedure, given the creation of the store (which we shall come to in the next section), there is no need to record the local variables of the procedure: these are indexed by the procedure's "full name" in any case. In fact, all we need to record are the formal parameters of the procedure (using *FormalParam*) and its body, a statement which can be evaluated with the right environment and store to yield the result of execution of the procedure. For a function, in addition to formal parameters and statement-part, we also require the expression (*Exp*) which is returned by the **return** statement – the last statement in the body of the function – and its type-mark.

On working through the Abstract Syntax representation of a SPARK program using the dynamic semantic rules in this document, it is necessary to construct a structured object, in which scopes of the above form are layered over one another. This leads to our dynamic environment, which takes the form:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Env \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
dict : (\mathrm{seq}_1\ Id) \nrightarrow Dict \\
pdecs : \mathbb{P}\ Id;\ pdefs : Id \nrightarrow Stmt \\
\hline
\mathrm{dom}\ pdefs \subseteq pdecs \\
\hline
\end{array}
$$

Note that the *dict* component of this schema is a partial function from *sequences* of identifiers, instead of from single identifiers. As each package specification and body is encountered in the abstract syntax, it is traversed and stored in this environment using a name structure which makes it straightforward, if one knows where one is during execution, to determine what a particular identifier stands for. For instance, given a library unit K, which contains a procedure P, which itself contains another procedure Q, and both P and Q have a local variable X. This gives us three elements for the domain of the dictionary:

$$
\begin{array}{l}
\langle K \rangle \mapsto \ldots, \\
\langle K, P \rangle \mapsto \ldots,\ \ \mathit{and} \\
\langle K, P, Q \rangle \mapsto \ldots.
\end{array}
$$

Both of the latter two will, as part of their respective dictionary entries, have non-empty *var* components, in which the identifier X will appear in the domain of *var* and will be mapped to some type-mark by *var* (which in general will be a different type-mark for each case). In this way, we gain unique "references" for each identifier in a SPARK program, and can determine which X is being referred to, for instance, by a "context" which tells us where in this structure execution is currently pointing. Note that this structure is in marked contrast to the static semantics, which essentially checks everything once and discards as soon as possible from its environment: this is not surprising, since program "execution" involves a process akin to memory allocation, and SPARK's constraints are such that static allocation can in principle be performed. This is what we do with the store, which we now discuss in the next section.

References: Val p. 9; IdDot p. 8; TypCon p. 15; FormalParam p. 160; Stmt p. 121; Exp p. 47; Dict p. 10.

2.4 The Dynamic Store

The dynamic store maps references to visible identifiers (both in the current package and in other packages) to values. We define the type *Store* by:

$$Store == (\mathrm{seq}_1\ Id) \nrightarrow Val$$

Thus, each variable referred to anywhere in a SPARK text can have a location in the store, uniquely defined by its fully expanded name. (The static semantic constraints of SPARK, which severely restrict reuse of identifiers, together with Ada's own visibility constraints, give this uniqueness.)
References: Val p. 9.

Chapter 3

Type Definitions

This chapter describes SPARK type definitions. A type definition is the term in syntax used in a type declaration:

type T **is** *a type definition*

The Abstract Syntax of type definitions (*TypDef*) is summarised in the following table:

In the rest of the chapter there is one section for each component of the syntax, preceded by the following introductory sections:

Finally, we describe the construction of an initial value for each type in the final section on page 27.

Use in Dynamic Semantics

For the dynamic semantics, we are interested in defining the type constructions with their range constraints where applicable, so that we can include checks to attempt to rule out "run-time errors". We use the predicate

$$c, \delta, \sigma \vdash_{typ} typ \Longrightarrow_{typ} typcon$$

to denote that, in context c and with environment δ and store σ, the type definition *typ* yields the type construction *typcon* as the collection of values defined by the type definition. In this, the "context" is a sequence of identifiers which identifies the current program execution context, which is updated whenever a new context is entered (e.g. a new package, a subprogram, etc.). This context is used as the index to fetch the appropriate dictionary from the dynamic environment described in the previous chapter.

3.1 Type Constructions

The declaration of a type in SPARK *constructs* a new type. The set *TypCon* is used to describe type constructions.

$$
\begin{aligned}
TypCon ::= {} & intT \langle\!\langle IntTypCon \rangle\!\rangle \\
\mid {} & enumT \langle\!\langle EnumTypCon \rangle\!\rangle \\
\mid {} & floatT \langle\!\langle FloatTypCon \rangle\!\rangle \\
\mid {} & fixedT \langle\!\langle FixedTypCon \rangle\!\rangle \\
\mid {} & arrT \langle\!\langle ArrTypCon \rangle\!\rangle \\
\mid {} & uarrT \langle\!\langle ArrTypCon \rangle\!\rangle \\
\mid {} & recT \langle\!\langle RecTypCon \rangle\!\rangle
\end{aligned}
$$

The following subsections describe the different type constructors.

3.1.1 Integer

An integer type is defined by a set of integers; all the integer values in the set (with appropriate decoration) belong to the type.

$$IntTypCon == \mathbb{P}\,\mathbb{Z}$$

3.1.2 Enumeration

An enumeration type is defined by a set of enumeration values, which are each associated with a position number (which is unique within the type construction).

$$EnumTypCon == \mathbb{Z} \rightarrowtail Id$$

3.1.3 Floating Point

A floating point type is defined by a range and the number of digits used in the decimal representations of the mantissa and the exponent.

$$FloatTypCon \mathrel{\widehat=} [\ digits : \mathbb{Z};\ range : \mathbb{P}\,Val \mid range \subseteq \operatorname{ran} realval\ ]$$

Note that all the values in the type belong to the *range*, but the converse is not true.

3.1.4 Fixed Point

A fixed point type is so called because it represents a rational value anywhere in its range to a fixed accuracy. The type is defined by the *delta*, which is the difference between successive values of the type, and a range.

$$FixedTypCon \mathrel{\widehat{=}} [\ delta : \mathrm{Real};\ range : \mathbb{P}\ Val \mid range \subseteq \mathrm{ran}\ realval\ ]$$

Note that all the values in the type belong to the *range*, but the converse is not true.

3.1.5 Array

An array is a composite type in which all the components have the same type *component*. The elements of the array are indexed by one or more index types *indexes*. Since SPARK requires explicit intermediate types to be used to create an array of arrays, both the component type and the index types are represented by their names — elements of *IdDot*.

$$ArrTypCon \mathrel{\widehat{=}} [indexes : \mathrm{seq}_1\ IdDot;\ component : IdDot]$$

3.1.6 Record

A record is defined by a non-empty ordered list of distinct field identifiers. The order is significant when a positional aggregate is used to define a value of the type. A type is associated with each field. Since SPARK requires explicit intermediate types to be used to create nested records, the field types are represented by their names (from *IdDot*).

$$RecTypCon \mathrel{\widehat{=}} [fields : \mathrm{iseq}_1\ Id;\ types : Id \mathrel{\ooalign{$\rightarrow$\cr\hfil$\mapstochar$\hfil}} IdDot \mid \mathrm{ran}\, fields = \mathrm{dom}\, types]$$

References: Val p. 9; intval p. 9; enumval p. 9; realval p. 9

3.2 Declaration of Operators

This section is omitted from the dynamic semantics.

3.3 Integer Types

An integer type is defined by a range of permitted values.

Syntax Example	A.S. Representation
range 1 .. 99	$int \langle\!| \; lower \mapsto lint\ 1,$ $upper \mapsto lint\ 99 \; |\!\rangle$

3.3.1 Abstract Syntax

The bounds of the range are expressions.

$$IntTypDef \mathrel{\widehat=} [lower, upper : Exp]$$

$$TypDef ::= int\langle\!\langle IntTypDef\rangle\!\rangle \mid \ldots$$

3.3.2 Dynamic Semantics

All integer values between the lower and upper bound are in the set of values for this type construction.

$$\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ vals : \mathbb{P}\,\mathbb{Z};\ IntTypDef \\
\mid \\
\quad vals = \{v : \mathbb{Z} \mid lv \leq v \leq uv \bullet v\} \\
\bullet \\
\quad \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_e lower \Longrightarrow_e lv \\ c, \delta, \sigma \vdash_e upper \Longrightarrow_e uv \end{array}}{c, \delta, \sigma \vdash_{typ} int(\theta IntTypDef) \Longrightarrow_{typ} intT\ \ vals}
\end{array} \tag{IntD}$$

References: Env p. 11; Store p. 12; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47; *intT p. 15.*

3.4 Enumerations

An enumeration is a (non-empty) list of *enumeration literals.*

Syntax Example	A.S. Representation
(red, green, blue)	$enum\ \langle red, green, blue\rangle$

3.4.1 Abstract Syntax

The enumeration literals are identifiers.

$$TypDef ::= \ldots \mid enum\langle\!\langle \mathrm{seq}_1\, Id\rangle\!\rangle$$

3.4.2 Dynamic Semantics

We associate an *EnumTypCon* with the sequence of identifiers which make up the enumeration literals of the type definition and return this.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ is : \mathrm{iseq}\, Id;\ et : EnumTypCon \mid 0 \vdash_{enum} is \Longrightarrow_{enum} et}{c, \delta, \sigma \vdash_{typ} enum\ is \Longrightarrow_{typ} enumT\ et} \quad \text{(EnumD)}$$

The evaluation predicate which constructs the *EnumTypCon* in the above may be defined by:

$$\frac{\forall\, n : \mathbb{Z} \bullet}{n \vdash_{enum} \langle\rangle \Longrightarrow_{enum} \varnothing} \quad \text{(EnumL1)}$$

$$\frac{\forall\, n : \mathbb{Z};\ e : Id;\ es : \mathrm{iseq}\, Id;\ et : \mathbb{Z} \rightarrowtail Id \bullet n + 1 \vdash_{enum} es \Longrightarrow_{enum} et}{n \vdash_{enum} \langle e\rangle \frown es \Longrightarrow_{enum} et \cup \{n \mapsto e\}} \quad \text{(EnumL2)}$$

References: Env p. 11; Store p. 12.

3.5 Floating Point

A floating point type can be defined by specifying the number of digits to be used for the mantissa.

Syntax Example	A.S. Representation
digits 7	*float* (*lint* 7)

3.5.1 Abstract Syntax

An expression is used to specify the number of digits.

$$TypDef ::= \ldots \mid float\langle\langle Exp\rangle\rangle$$

The number of decimal digits used for the exponent is four times the number of binary digits actually used for the mantissa [AARM, §3.5.7, ¶6,7].

3.5.2 Dynamic Semantics

Implementation dependent: representation of reals.

3.6 Floating Point — with Range

A floating point type can be defined by a number of digits and a range of permitted values.

Syntax Example	A.S. Representation
digits 7 **range** 0.0 .. 100.0	$\mathit{floatr} \langle\!| \; \mathit{digits} \mapsto \mathit{lint}\ 7,$ $\mathit{lower} \mapsto \mathit{lreal}\ 0.0,$ $\mathit{upper} \mapsto \mathit{lreal}\ 100.0 \; |\!\rangle$

3.6.1 Abstract Syntax

Both the number of digits and the bounds of the range are specified using expressions.

$$FloatRTypDef \mathrel{\widehat{=}} [digits, lower, upper : Exp]$$

$$TypDef ::= \ldots \mid floatr\langle\!\langle FloatRTypDef \rangle\!\rangle$$

3.6.2 Dynamic Semantics

Implementation dependent: representation of reals.
References: Exp p. 47.

3.7 Fixed Point

A fixed point type is defined by an accuracy and a range of permitted values.

Syntax Example	A.S. Representation
delta 0.001 **range** 0.0 .. 50.0	$\mathit{fixr}\ \langle\!\vert\ \mathit{delta} \mapsto \mathit{lreal}\ 0.001,\ \mathit{lower} \mapsto \mathit{lreal}\ 0.0,\ \mathit{upper} \mapsto \mathit{lreal}\ 50.0\ \vert\!\rangle$

3.7.1 Abstract Syntax

The accuracy and the bounds of the range are specified by expressions.

$$\mathit{FixRTypDef} \mathrel{\widehat{=}} [\mathit{delta}, \mathit{lower}, \mathit{upper} : \mathit{Exp}]$$

$$\mathit{TypDef} ::= \ldots \mid \mathit{fixr}\langle\!\langle \mathit{FixRTypDef} \rangle\!\rangle$$

3.7.2 Dynamic Semantics

Implementation dependent: representation of reals.
References: Exp p. 47.

3.8 Arrays

An array type definition has a list of index type names and a component type name.

Syntax Example	A.S. Representation
array (bran.code, prod) **of** cust	$arr \langle\!\mid indexes \mapsto \langle dot(bran, code), id\ prod\rangle,$ $component \mapsto id\ cust \mid\!\rangle$

3.8.1 Abstract Syntax

The type names are elements of *IdDot*.

$$ArrTypDef \mathrel{\widehat=} [indexes : \mathrm{seq}_1\, IdDot;\ component : IdDot]$$

$$TypDef ::= \ldots \mid arr\langle\!\langle ArrTypDef\rangle\!\rangle$$

3.8.2 Dynamic Semantics

We regard array values as belonging to the type for which they have been declared.

$$\frac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ ArrTypDef;\ ArrTypCon \\ spot\end{array}}{c, \delta, \sigma \vdash_{typ} arr(\theta ArrTypDef) \Longrightarrow_{typ} arrT(\theta ArrTypCon)} \qquad \text{(ArrD)}$$

References: IdDot p. 8; Env p. 11; Store p. 12.

3.9 Unconstrained Arrays

An unconstrained array type has one or more unconstrained dimensions. Before the type can be used in an object declaration, the actual range of the index type to be used must be specified (by declaring a subtype).

Syntax Example	A.S. Representation
array (integer **range <>**) **of** customer	$uarr \langle\!| \; indexes \mapsto \langle id\ integer \rangle,$ $component \mapsto id\ customer \; |\!\rangle$

3.9.1 Abstract Syntax

$$UArrTypDef \mathrel{\widehat=} [indexes : \mathrm{seq}_1\, IdDot;\ component : IdDot]$$

$$TypDef ::= \ldots \mid uarr\langle\!\langle UArrTypDef \rangle\!\rangle$$

3.9.2 Dynamic Semantics

We regard all array values as belonging to the type for which they have been declared. Unconstrained arrays are not an issue for the dynamic semantics, since the copy-in, copy-out semantics used for parameter passing instantiate array formal parameters with actual, constrained array objects on invocation of a procedure or function.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ UArrTypDef;\ ArrTypCon \\ \bullet}{c, \delta, \sigma \vdash_{typ} uarr(\theta\, UArrTypDef) \Longrightarrow_{typ} uarrT(\theta\, ArrTypCon)} \quad \text{(UArrD)}$$

References: IdDot p. 8; Env p. 11; Store p. 12.

3.10 Records

A record type definition gives a type to a set of field names.

Syntax Example	A.S. Representation
record name : string30; min,max : integer; **end record**	$rec\ \langle\ \langle\!|\ fld \mapsto name,\ comp \mapsto id\ string30\ |\!\rangle, \langle\!|\ fld \mapsto min,\ comp \mapsto id\ integer\ |\!\rangle, \langle\!|\ fld \mapsto max,\ comp \mapsto id\ integer\ |\!\rangle\ \rangle$

3.10.1 Abstract Syntax

The field name is an identifier; the component type name is an element of *IdDot*.

$$RecFldTypDef \mathrel{\widehat=} [fld : Id;\ comp : IdDot]$$

In the Concrete Syntax, field name with the same type may be given in a list; this is represented in the Abstract Syntax by repeating the type name for each field in the list.

$$TypDef ::= \ldots \mid rec\langle\!\langle \mathrm{seq}_1\ RecFldTypDef \rangle\!\rangle$$

3.10.2 Dynamic Semantics

$$\frac{\begin{array}{l}\forall\ c : \mathrm{seq}\ Id;\ \delta : Env;\ \sigma : Store;\ rs : \mathrm{seq}_1\ FldTypDef;\\ \ RecTypCon\\ \mid\\ \qquad mk\_rec\_con(rs) = (fields, types)\\ \bullet\end{array}}{c, \delta, \sigma \vdash_{typ}\ rec(rs) \Longrightarrow_{typ}\ recT(\theta RecTypCon)} \qquad \text{(RecD)}$$

In the above, we use an auxiliary function *mk_rec_con* to traverse the sequence; we define this by:

$$\begin{array}{|l} mk\_rec\_con : \mathrm{seq}\ FldTypDef \rightarrow ((\mathrm{seq}\ Id) \times (Id \rightarrow\!\!\!\!\!\!+\ IdDot))\\ \hline mk\_rec\_con(\langle\rangle) = (\langle\rangle, \varnothing)\\ \forall\ r : FldTypDef;\ s : \mathrm{seq}\ FldTypDef;\ f : \mathrm{seq}\ Id;\ t : Id \rightarrow\!\!\!\!\!\!+\ IdDot \bullet\\ \qquad mk\_rec\_con(s) = (f, t) \Rightarrow\\ \qquad\qquad mk\_rec\_con(\langle r\rangle \frown s) = (\langle r.fld\rangle \frown f, t \oplus \{r.fld \mapsto r.comp\})\end{array}$$

Note that the constraint that the sequence of field identifiers constructed is unique is missing: this should be guaranteed by the static semantics checks, which are assumed to hold in the Dynamic Semantics.
References: IdDot p. 8; Env p. 11; Store p. 12.

3.11 Initial Values for Types

Associated with each type (or subtype) name, we require an "initial value" which may be assigned initially to an object of that type in the store whenever a variable of that type is initialised. We shall define the functions which we provide for this purpose at this point in the document.

We provide three functions for the formation of initial values (for initialisation of the store):

$$
\begin{array}{l}
form\_init\_val_\delta : ((\operatorname{seq} Id) \times IdDot) \rightarrowtail Val \\
form\_init\_array_\delta : ((\operatorname{seq} Id) \times (\operatorname{seq} IdDot) \times IdDot) \rightarrowtail Val \\
form\_init\_rec_\delta : ((\operatorname{seq} Id) \times (\operatorname{iseq} Id) \times (Id \rightarrowtail IdDot)) \rightarrowtail Val
\end{array}
$$

(Each of these functions is defined with respect to the dynamic environment δ, which we regard as an implicit parameter to these functions.) The first is the principal one, and given a type-mark as argument, returns an initial value to associate with an object of that type or subtype. For the scalar types, this initial value is the distinguished element *undefined*; for arrays and records, the appropriate structure is established, with all of its relevant scalar components assuming the *undefined* value. The other two functions are used by (and use) *form_init_val* to perform this construction. We thus define:

$$
\begin{array}{l}
form\_init\_val_\delta : ((\operatorname{seq} Id) \times IdDot) \rightarrowtail Val \\
\hline
\forall c : \operatorname{seq} Id;\ i : IdDot \\
\mid \quad get\_typ\_con_\delta\ (c, i) \in \operatorname{ran} intT \lor \\
\quad get\_typ\_con_\delta\ (c, i) \in \operatorname{ran} enumT \lor \\
\quad get\_typ\_con_\delta\ (c, i) \in \operatorname{ran} floatT \lor \\
\quad get\_typ\_con_\delta\ (c, i) \in \operatorname{ran} fixedT \\
\bullet \\
\quad form\_init\_val_\delta\ (c, i) = undefined \\
\forall c : \operatorname{seq} Id;\ i : IdDot;\ ArrTypCon \\
\mid \quad get\_typ\_con_\delta\ (c, i) = arrT(\theta ArrTypCon) \\
\bullet \\
\quad form\_init\_val_\delta\ (c, i) = form\_init\_array_\delta\ (c, indices, component) \\
\forall c : \operatorname{seq} Id;\ i : IdDot;\ RecTypCon \\
\mid \quad get\_typ\_con_\delta\ (c, i) = recT(\theta RecTypCon) \\
\bullet \\
\quad form\_init\_val_\delta\ (c, i) = form\_init\_rec_\delta\ (c, fields, types)
\end{array}
$$

$$form\_init\_array_\delta : ((\operatorname{seq} Id) \times (\operatorname{seq} IdDot) \times IdDot) \pfun Val$$

$$\forall c : \operatorname{seq} Id;\ i, e : IdDot;\ v : Array\_Value;\ IntTypCon$$
$$\mid \quad get\_typ\_con_\delta\ (i, c) = intT(\theta IntTypCon) \land$$
$$v.lo = min\ range \land v.hi = max\ range \land$$
$$v.arr = (\lambda\, x : x.lo \mathrel{..} x.hi \bullet form\_init\_val_\delta\ (c, e))$$
$$\bullet$$
$$form\_init\_array_\delta\ (c, \langle i \rangle, e) = arrval\ v$$

$$\forall c : \operatorname{seq} Id;\ i, e : IdDot;\ is : \operatorname{seq}_1 IdDot;\ v : Array\_Value;\ IntTypCon$$
$$\mid \quad get\_typ\_con_\delta\ (c, i) = intT(\theta IntTypCon) \land$$
$$v.lo = min\ range \land v.hi = max\ range \land$$
$$v.arr = (\lambda\, x : x.lo \mathrel{..} x.hi \bullet form\_init\_array_\delta\ (c, is, e))$$
$$\bullet$$
$$form\_init\_array_\delta\ (c, \langle i \rangle \frown is, e) = arrval\ v$$

$$form\_init\_rec_\delta : ((\operatorname{seq} Id) \times (\operatorname{iseq} Id) \times (Id \pfun IdDot)) \pfun Val$$

$$\forall c : \operatorname{seq} Id;\ i : Id;\ t : Id \pfun IdDot$$
$$\mid$$
$$i \in \operatorname{dom} t$$
$$\bullet$$
$$form\_init\_rec_\delta\ (c, \langle i \rangle, t) = recval\ \{i \mapsto form\_init\_val_\delta\ (c, t\ i)\}$$

$$\forall c : \operatorname{seq} Id;\ i : Id;\ is : \operatorname{iseq}_1 Id;\ t : Id \pfun IdDot$$
$$\mid$$
$$i \in \operatorname{dom} t \land$$
$$i \notin \operatorname{ran} is$$
$$\bullet$$
$$form\_init\_rec_\delta\ (c, \langle i \rangle \frown is, t) =$$
$$recval\ (\{i \mapsto form\_init\_val_\delta\ (c, t\ i)\} \cup recval^\sim form\_init\_rec_\delta\ (c, is, t))$$

References: IdDot p. 8; Val p. 9; Env p. 11; get_typ_con$_\delta$ p. 216; *intT p. 15; enumT p. 15; floatT p. 15; fixedT p. 15; ArrayValue p. 9.*

Chapter 4

Subtype Definitions

For dynamic semantic purposes, we regard a subtype definition as the same as a type definition in terms of the type construction generated for embedding in the dynamic environment.

For brevity, we do not spell out the mappings necessary to do this here: these are relatively apparent. Thus, range constraints define scalar types (integer, enumerated, fixed or floating point) as per the relevant sections of the preceding chapter; fixed and floating point accuracy constraints define appropriate fixed/floating point types (though note: according to discussions in the Ada Rapporteur Group, there should be no semantic effect of reduced accuracy subtypes); and array index constraints introduce a constrained array type based on an earlier unconstrained type definition. We leave these transformations as an exercise for the reader.

Chapter 5

Names

This Chapter describes the Abstract Syntax category *Name*. Names include all the terms which could appear on the left hand side of an assignment statement, and other terms, such as function calls, which cannot be distinguished syntactically from names. The following table summarises the Abstract Syntax.

Syntax Constructor	Description	Page
simp	simple name	34
slct	selected name	37
pasc	positional association - array or function call	41
nasc	named association - function call	45

In the rest of the Chapter there is one Section for each component of the syntax, preceded by Section 5.1 which is used in the accompanying Static Semantics document to define the set of type names *NameType*, but which is not used here (though it is retained to keep the section numbering consistent between the two documents).

Dynamic Semantics

The dynamic semantics of a name depends upon whether it is used on the left- or the right-hand side of an assignment statement. In this chapter, we give the dynamic semantics of names as right-hand side expressions, rather than as addresses for the updating of store. We defer consideration of the latter usage until the dynamic semantics of the assignment statement.

The evaluation of a name is defined by a relation which may return a value of type *Val*. With $\delta : Env$, $\sigma : Store$, $name : Name$ and $val : Val$, the predicate:

$$\delta, \sigma \vdash_n name \Longrightarrow_n val$$

can be read as "Name *name* evaluates in environment δ and with store σ to the value *val*".

Note: Not all "name" terms denote valid expressions, i.e. objects (whether constants, variables, or components thereof); some names are type marks, for instance, which are not strictly expressions in the Ada sense.
References: Val p. 9; Env p. 11; Store p. 12.

5.1 Name Types

The name types described in the static semantics are not relevant to the dynamic semantics.

5.2 Simple Names

A simple name can be the name of a constant, an enumeration literal, a type, a variable, a function or a package.

Syntax Example	A.S. Representation
AVar	*simp AVar*

A field of a record never appears as a simple name in a selected name, since only the prefix of a selected name (see Section 5.3) is a *name*, while the selector (the field name) is an identifier.

5.2.1 Abstract Syntax

$Name ::= simp\langle\!\langle Id \rangle\!\rangle \mid \ldots$

5.2.2 Dynamic Semantics

First, we look for the identifier in the current scope (as represented by the context c in the first five rules below). If it is present, it will be either a constant, an enumeration literal, a type, a variable or a function name, in which case the relevant rule will apply. If it is none of these, we look in the enclosing scope, and so on until we find an entry for the identifier.

Constant The identifier (con) can be a constant. Its value can be determined from the associated dictionary entry in the dynamic environment.

$$\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ con : Id \mid con \in \operatorname{dom}(\delta.dict\ c).const \bullet \frac{}{c, \delta, \sigma \vdash_n simp\ con \Longrightarrow_n (\delta.dict\ c).const\ con} \qquad \text{(Simp1D)}$$

Enumeration Literal The identifier can be an enumeration literal. Its value is then the identifier as projected into the set Val.

$$
\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ lit : Id \\ | \\ \quad \exists\, t : Id \bullet \\ \qquad t \in \operatorname{dom}(\delta.dict\ c).type \wedge \\ \qquad (\delta.dict\ c).type\ t \in \operatorname{ran} enumT \wedge \\ \qquad lit \in \operatorname{ran}(enumT^{\sim}(\delta.dict\ c).type\ t) \\ \bullet\end{array}}{c, \delta, \sigma \vdash_n simp\ lit \Longrightarrow_n enumval\ (id\ lit)} \qquad \text{(Simp2D)}
$$

Type The identifier can be a type. In this case, the "value" of the name is the set of values in our value domain which are in the subtype specified by this type-mark.

$$
\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ tid : Id \\ | \\ \quad tid \in \operatorname{dom}(\delta.dict\ c).type \\ \bullet\end{array}}{c, \delta, \sigma \vdash_n simp\ tid \Longrightarrow_n rngval\ typrange(c, \delta, id\ tid)} \qquad \text{(Simp3D)}
$$

The function *typrange* used in the above inference rule is defined in the auxiliary functions section of this document.

Variable The identifier can be the name of a variable. Its value can then be determined from the store.

$$
\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ var : Id \\ | \\ \quad c \frown \langle var \rangle \in \operatorname{dom} \sigma \\ \bullet\end{array}}{c, \delta, \sigma \vdash_n simp\ var \Longrightarrow_n \sigma\ (c \frown \langle var \rangle)} \qquad \text{(Simp4D)}
$$

Function The identifier can be the name of a function which does not take any arguments. Its value can then be determined by evaluation in the current context.

To evaluate a function call without any arguments, the following steps must be taken:

1. Variables local to the function are set to their initial values;
2. The function body is executed with this new store;
3. The function return expression is evaluated.

We therefore define:

$$
\begin{array}{l}
\forall\, c : \mathrm{seq}\, \mathit{Id};\ \delta : \mathit{Env};\ \sigma, \sigma', \sigma'' : \mathit{Store};\ \mathit{fun} : \mathit{Id};\\
\quad \mathit{st} : \mathit{Stmt};\ \mathit{exp} : \mathit{Exp};\ \mathit{tid} : \mathit{IdDot}\\
|\\
\qquad \mathit{fun} \in \mathrm{dom}(\delta.\mathit{dict}\ c).\mathit{funs}\\
\qquad (\delta.\mathit{dict}\ c).\mathit{funs}\ \mathit{fun} = (\langle\rangle, \mathit{st}, \mathit{exp}, \mathit{tid})\\
\qquad \sigma' = \mathit{clear\_locals}(\sigma, \delta, c \frown \langle \mathit{fun} \rangle)\\
\bullet\\
\qquad c \frown \langle \mathit{fun} \rangle, \delta, \sigma' \vdash_s \mathit{st} \Longrightarrow_s \sigma''\\
\qquad c \frown \langle \mathit{fun} \rangle, \delta, \sigma'' \vdash_e \mathit{exp} \Longrightarrow_e \mathit{val}\\
\hline
\qquad c, \delta, \sigma \vdash_n \mathit{simp}\ \mathit{fun} \Longrightarrow_n \mathit{val}
\end{array}
\tag{Simp5D}
$$

Package The identifier cannot be a package for a name expression evaluated by the dynamic semantics.

Outer Scope The identifier could be in an enclosing scope, rather than present in the current scope. In this case, the following rule applies.

$$
\begin{array}{l}
\forall\, c : \mathrm{seq}\, \mathit{Id};\ \mathit{ct} : \mathit{Id};\ \delta : \mathit{Env};\ \sigma : \mathit{Store};\ x : \mathit{Id};\ v : \mathit{Val}\\
|\\
\qquad x \notin \mathrm{dom}(\delta.\mathit{dict}\ (c \frown \langle \mathit{ct} \rangle)).\mathit{const}\\
\qquad \neg\ \exists\, t : \mathit{Id} \bullet\\
\qquad\qquad t \in \mathrm{dom}(\delta.\mathit{dict}\ (c \frown \langle \mathit{ct} \rangle)).\mathit{type} \wedge\\
\qquad\qquad (\delta.\mathit{dict}\ (c \frown \langle \mathit{ct} \rangle)).\mathit{type}\ t \in \mathrm{ran}\ \mathit{enumT} \wedge\\
\qquad\qquad x \in \mathrm{ran}(\mathit{enumT}^{\sim}(\delta.\mathit{dict}\ (c \frown \langle \mathit{ct} \rangle)).\mathit{type}\ t)\\
\qquad x \notin \mathrm{dom}(\delta.\mathit{dict}\ (c \frown \langle \mathit{ct} \rangle)).\mathit{type}\\
\qquad c \frown \langle \mathit{ct}, x \rangle \notin \mathrm{dom}\, \sigma\\
\qquad z \notin \mathrm{dom}(\delta.\mathit{dict}\ (c \frown \langle \mathit{ct} \rangle)).\mathit{funs}\\
\bullet\\
\qquad c, \delta, \sigma \vdash_n \mathit{simp}\ x \Longrightarrow_n v\\
\hline
\qquad c \frown \langle \mathit{ct} \rangle, \delta, \sigma \vdash_n \mathit{simp}\ x \Longrightarrow_n v
\end{array}
\tag{Simp6D}
$$

References: Env p. 11; Store p. 12; Val p. 9; typrange p. 216; clear_locals p. 156.

5.3 Selected Names

A selected name can be a name from another package or a field of a record object.

Syntax Example	A.S. Representation
K.T	$slct \langle\!\mid prefix \mapsto simp\ K,\ selector \mapsto T \mid\!\rangle$
K.R.F	$slct \langle\!\mid prefix \mapsto slct \langle\!\mid prefix \mapsto simp\ K,\ selector \mapsto R \mid\!\rangle,\ selector \mapsto F \mid\!\rangle$

5.3.1 Abstract Syntax

The prefix of a selected name is itself a name, for example a call to a function which returns a record, or a package variable of record type. The selector is always an identifier.

$$SlctName \mathrel{\widehat=} [prefix : Name;\ selector : Id]$$

$$Name ::= \ldots \mid slct\langle\!\langle SlctName\rangle\!\rangle$$

5.3.2 Dynamic Semantics

The prefix can be a package name or an object name. In the following rules, we shall use the following definition:

$$idtyp ::= varI \mid constI \mid typeI \mid elitI \mid funI \mid procI \mid pkgI$$

This allows us to categorise identifiers according to their use in the current scope. We fetch the category and the appropriate scope for the object with:

$$get\_id\_ctx : ((\operatorname{seq} Id) \times Env \times Id) \nrightarrow ((\operatorname{seq} Id) \times idtyp)$$

$$\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ i : Id \mid i \in \operatorname{dom}(\delta.dict\ c).procs \bullet \\
\quad get_id_ctx(c, \delta, i) = (c\ cat\langle i\rangle, procI) \\
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ i : Id \mid i \in \operatorname{dom}(\delta.dict\ c).funs \bullet \\
\quad get_id_ctx(c, \delta, i) = (c\ cat\langle i\rangle, funI) \\
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ i : Id \mid i \in \operatorname{dom}(\delta.dict\ c).const \bullet \\
\quad get_id_ctx(c, \delta, i) = (c, constI) \\
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ i : Id \mid i \in \operatorname{dom}(\delta.dict\ c).var \bullet \\
\quad get_id_ctx(c, \delta, i) = (c, varI) \\
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ i : Id \mid i \in \operatorname{dom}(\delta.dict\ c).type \bullet \\
\quad get_id_ctx(c, \delta, i) = (c, typeI) \\
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ i : Id \mid \\
\quad \exists\, t : Id \bullet t \in \operatorname{dom}(\delta.dict\ c).type \land \\
\qquad (\delta.dict\ c).type\ t \in \operatorname{ran} enumT \land i \in \operatorname{ran}(enumT^{\sim}(\delta.dict\ c).type\ t) \bullet \\
\quad get_id_ctx(c, \delta, i) = (c, elitI) \\
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ i : Id \mid \\
\quad c \frown \langle i\rangle \in \operatorname{dom} \delta.dict \land i \notin \operatorname{dom}(\delta.dict\ c).procs \land i \notin \operatorname{dom}(\delta.dict\ c).funs \bullet \\
\quad get_id_ctx(c, \delta, i) = (c \frown \langle i\rangle, pkgI) \\
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ i, ct : Id \mid \\
\quad i \notin \operatorname{dom}(\delta.dict\ (c \frown \langle ct\rangle)).procs \land i \notin \operatorname{dom}(\delta.dict\ (c \frown \langle ct\rangle)).funs \land \\
\quad i \notin \operatorname{dom}(\delta.dict\ (c \frown \langle ct\rangle)).const \land i \notin \operatorname{dom}(\delta.dict\ (c \frown \langle ct\rangle)).var \land \\
\quad i \notin \operatorname{dom}(\delta.dict\ (c \frown \langle ct\rangle)).type \land \\
\quad (\lnot\ \exists\, t : Id \bullet t \in \operatorname{dom}(\delta.dict\ (c \frown \langle ct\rangle)).type \land \\
\qquad (\delta.dict\ (c \frown \langle ct\rangle)).type\ t \in \operatorname{ran} enumT \land \\
\qquad i \in \operatorname{ran}(enumT^{\sim}(\delta.dict\ (c \frown \langle ct\rangle)).type\ t)) \land \\
\quad c \frown \langle ct, i\rangle \notin \operatorname{dom} \delta.dict \bullet \\
\quad get_id_ctx(c \frown \langle ct\rangle, \delta, i) = get_id_ctx(c, \delta, i)
\end{array}$$

Selecting a Constant from a Package The value of the constant can be determined from the dynamic environment.

$$\forall\, c, pc : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ SlctName;\ pak : Id$$
$$|$$
$$\begin{array}{l} prefix = simp\ pak \\ get\_id\_ctx(c, \delta, pak) = (pc, pkgI) \\ get\_id\_ctx(pc, \delta, selector) = (pc, constI) \end{array} \qquad \text{(Slct1D)}$$
$$\bullet$$
$$\overline{c, \delta, \sigma \vdash_n slct(\theta SlctName) \Longrightarrow_n (\delta.dict\ pc).const\ selector}$$

Selecting an Enumeration Literal from a Package The value is the appropriate enumeration literal value.

$$\forall\, c, pc : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ SlctName;\ pak : Id$$
$$|$$
$$\begin{array}{l} prefix = simp\ pak \\ get\_id\_ctx(c, \delta, pak) = (pc, pkgI) \\ get\_id\_ctx(pc, \delta, selector) = (pc, elitI) \end{array} \qquad \text{(Slct2D)}$$
$$\bullet$$
$$\overline{c, \delta, \sigma \vdash_n slct(\theta SlctName) \Longrightarrow_n enumval\ (dot\ (pak, selector))}$$

Selecting a Type from a Package The value of the type is the set of values associated with the type-mark.

$$\forall\, c, pc : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ SlctName;\ pak : Id$$
$$|$$
$$\begin{array}{l} prefix = simp\ pak \\ get\_id\_ctx(c, \delta, pak) = (pc, pkgI) \\ get\_id\_ctx(pc, \delta, selector) = (pc, typeI) \end{array} \qquad \text{(Slct3D)}$$
$$\bullet$$
$$\overline{\begin{array}{l} c, \delta, \sigma \vdash_n slct(\theta SlctName) \Longrightarrow_n \\ \quad rngval\ typrange(\delta, (\delta.dict\ pc).type\ selector) \end{array}}$$

Selecting a Variable from a Package The value of the variable is found in the store for the package.

$$\begin{array}{l}
\forall\, c, pc : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ SlctName;\ pak : Id \\
\mid \\
\qquad prefix = simp\ pak \\
\qquad get_id_ctx(c, \delta, pak) = (pc, pkgI) \\
\qquad get_id_ctx(pc, \delta, selector) = (pc, varI) \\
\bullet \\
\hline
\qquad \delta, \sigma \vdash_n slct(\theta SlctName) \Longrightarrow_n \sigma\ (pc \frown \langle selector \rangle)
\end{array} \tag{Slct4D}$$

Selecting a Function from a Package The value returned by the function (which can take no arguments, since it is in this syntactic category) is that which results from evaluating the function call in the store appropriate to the enclosing package.

$$\begin{array}{l}
\forall\, c, pc, fc : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ SlctName;\ pak : Id; \\
\ st : Stmt;\ exp : Exp;\ tid : IdDot;\ val : Val \\
\mid \\
\qquad prefix = simp\ pak \\
\qquad get_id_ctx(c, \delta, pak) = (pc, pkgI) \\
\qquad get_id_ctx(pc, \delta, selector) = (fc, funI) \\
\qquad (\delta.dict\ pc).funs\ selector = (\langle\rangle, st, exp, tid) \\
\qquad \sigma' = clear_locals(\sigma, \delta, fc) \\
\bullet \\
\qquad fc, \delta, \sigma' \vdash_s st \Longrightarrow_s \sigma'' \\
\qquad fc, \delta, \sigma'' \vdash_e exp \Longrightarrow_e val \\
\hline
\qquad c, \delta, \sigma \vdash_n slct(\theta SlctName) \Longrightarrow_n val
\end{array} \tag{Slct5D}$$

Selection from an Object The object must be a record object, and the selector is the required field.

$$\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ SlctName;\ rv : Val \\
\mid \\
\qquad rv \in \mathrm{ran}\, recval\ \wedge \\
\qquad selector \in \mathrm{dom}(recval^{\sim} rv) \\
\bullet \\
\qquad c, \delta, \sigma \vdash_n prefix \Longrightarrow_n rv \\
\hline
\qquad \delta, \sigma \vdash_n slct(\theta SlctName) \Longrightarrow_n (recval^{\sim} rv)\ selector
\end{array} \tag{Slct7D}$$

References: Env p. 11; Store p. 12; Val p. 9; IdDot p. 8; clear_locals p. 156.

5.4 Positional Associations

A positional association is used to supply arguments either to an array, giving an indexed expression, or to a function name, giving a function call, or to a type name, giving a type conversion.

Syntax Example	A.S. Representation
f(a, b)	$pasc \langle\!\vert\ prefix \mapsto simp\ f,\ args \mapsto \langle \ldots, \ldots \rangle\ \vert\!\rangle$
k.f(e)	$pasc \langle\!\vert\ prefix \mapsto slct \langle\!\vert\ prefix \mapsto simp\ k,\ selector \mapsto f\ \vert\!\rangle,\ args \mapsto \langle \ldots \rangle\ \vert\!\rangle$

The second example could be well-typed in a number of different ways. For example, k could be a package containing a function f; alternatively k is an object of a record type having a field f of an array type.

5.4.1 Abstract Syntax

The prefix of a positional association is a name, being a function, an array object or a type.

$$PAscName \mathrel{\widehat{=}} [prefix : Name;\ args : \mathrm{seq}_1\ Exp]$$

$$Name ::= \ldots \mid pasc\langle\!\langle PAscName \rangle\!\rangle$$

5.4.2 Dynamic Semantics

Array Element The evaluation of an array element expression involves the evaluation of its index expressions, and the fetching of the appropriate element of the sequence of values representing the array. Note that array values are essentially sequences of values; in this way, multi-dimensional arrays are handled rather like sequences of sequences.

The evaluation of an array element expression involves a dynamic well-formation check, that the index values are in the appropriate index range. If any index is out of range, no value is returned.

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ pval : Val; \\
\ vals : \operatorname{seq}_1 Val;\ PAscName \\
\bullet \\
\qquad c, \delta, \sigma \vdash_n prefix \Longrightarrow_n pval \\
\qquad c, \delta, \sigma \vdash_{es} args \Longrightarrow_{es} vals \\
\hline
\qquad c, \delta, \sigma \vdash_n pasc\ (\theta PAscName) \Longrightarrow_n lookup\_element(pval, vals)
\end{array}
\qquad \text{(PAsc1D)}
$$

The function *lookup_element* returns the element of the array selected by the index values sequence; this function is defined in the auxiliary functions section.

Function Call To evaluate a function call, the following steps must be taken:

1. Actual parameters are associated with formal parameters in the store ("copy-in");
2. Variables local to the function are set to their initial values;
3. The function body is executed with this new store;
4. Finally, the function return expression is evaluated in this store.

Note that SPARK functions cannot have side-effects, nor can parameter values be modified (so not "copy-out" is necessary after the call).

There are two cases to consider, according to whether the function *name* is simple or is prefixed by a package-identifier. We therefore define:

$$
\begin{array}{l}
\forall\, c, pc : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ val : Val; \\
\ fun : Id;\ fps : \operatorname{seq} FormalParam;\ st : Stmt;\ exp : Exp; \\
\ tid : IdDot;\ PAscName \\
\mid \\
\qquad prefix = simp\ fun \\
\qquad get\_id\_ctx(c, \delta, fun) = (pc \frown \langle fun \rangle, funI) \\
\qquad (\delta.dict\ pc).funs\ fun = (fps, st, exp, tid) \\
\qquad \sigma'' = clear\_locals(\sigma', \delta', pc \frown \langle fun \rangle) \\
\bullet \\
\qquad c, \delta, \sigma \vdash_{copyin} (pc \frown \langle fun \rangle, fps, args') \Longrightarrow_{copyin} \delta', \sigma' \\
\qquad pc \frown \langle fun \rangle, \delta', \sigma'' \vdash_s st \Longrightarrow_s \sigma''' \\
\qquad pc \frown \langle fun \rangle, \delta', \sigma''' \vdash_e exp \Longrightarrow_e val \\
\hline
\qquad c, \delta, \sigma \vdash_n pasc(\theta PAscName) \Longrightarrow_n val
\end{array}
\qquad \text{(PAsc2aD)}
$$

where

$$\begin{aligned}
args' == (\lambda\, i : \mathrm{dom}\, args \bullet (\mu\, NamedActual \mid\\
&formal = (fps\ i).param \wedge\\
&actual = args\ i))
\end{aligned}$$

$$\frac{\begin{array}{l}\forall\, c, pc, fc : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ val : Val;\\ \quad pak : Id;\ sn : SlctName;\ fps : \mathrm{seq}\, FormalParam;\\ \quad st : Stmt;\ exp : Exp;\ tid : IdDot;\ PAscName\\ \mid\\ \qquad prefix = slct\ sn\\ \qquad sn.prefix = simp\ pak\\ \qquad get\_id\_ctx(c, \delta, pak) = (pc, pkgI)\\ \qquad get\_id\_ctx(pc, \delta, sn.selector) = (fc, funI)\\ \qquad (\delta.dict\ pc).funs\ sn.selector = (fps, st, exp, tid)\\ \qquad \sigma'' = clear\_locals(\sigma', \delta', fc)\\ \bullet\\ \qquad c, \delta, \sigma \vdash_{copyin} (fc, fps, args') \Longrightarrow_{copyin} \delta', \sigma'\\ \qquad fc, \delta', \sigma'' \vdash_s st \Longrightarrow_s \sigma'''\\ \qquad fc, \delta', \sigma''' \vdash_e exp \Longrightarrow_e val\end{array}}{c, \delta, \sigma \vdash_n pasc(\theta PAscName) \Longrightarrow_n val} \qquad \text{(PAsc2bD)}$$

where

$$\begin{aligned}
args' == (\lambda\, i : \mathrm{dom}\, args \bullet (\mu\, NamedActual \mid\\
&formal = (fps\ i).param \wedge\\
&actual = args\ i))
\end{aligned}$$

Type Conversion For a type conversion, the prefix is a type-name. In SPARK, type conversions will involve either no dynamic action (e.g. in the conversion of one integer type to another) or, in the case of reals, the possibility of rounding. Type conversions involve conversion to the base type associated with the type-name, then a check that the value so derived is within the subtype associated with the type-name [LRM, 4.6(3-4)]. Violation of the subtype constraints will result in the Ada exception `CONSTRAINT_ERROR` being raised [LRM, 4.6(12-13)], or a `NUMERIC_ERROR` exception may be raised in a type conversion involving numeric types in which evaluation goes out of range.

For numeric type conversions, we define

$$
\begin{array}{l}
\forall\, \delta : Env;\ \sigma : Store;\ vals : \mathbb{P}\ Val; \\
\quad val, cval : Val;\ PAscName \\
\mid \\
\qquad \#args = 1 \\
\qquad cval \in sufficiently\_close(val, vals) \\
\bullet \\
\qquad \delta, \sigma \vdash_n prefix \Longrightarrow_n rngval\ \ vals \\
\qquad \delta, \sigma \vdash_e args\ 1 \Longrightarrow_e val \\
\hline
\qquad \delta, \sigma \vdash_n pasc\ (\theta PAscName) \Longrightarrow_n cval
\end{array}
\tag{PAsc3D}
$$

The function *sufficiently_close* reflects the fact that the conversion should be to within the accuracy of the specified subtype.

References: Env p. 11; *Store p. 12*; *Val p. 9*; *lookup_element* p. 215; $\vdash_{es}$ p. 47; $\Longrightarrow_{es}$ p. 47; *IdDot p. 8*; *FormalParam p. 160*; *Stmt p. 121*; *Exp p. 47*; *clear_locals* p. 156; $\vdash_s$ p. 122; $\Longrightarrow_s$ p. 122; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47; *NamedActual p. 157*; *get_id_ctx* p. 38; $\vdash_{copyin}$ p. 160; $\Longrightarrow_{copyin}$ p. 160; *sufficiently_close* p. 216.

5.5 Named Association

In a function call, the association between formal parameters and their actual values can be specified using the formal parameter identifiers.

Syntax Example	A.S. Representation
f(arg1 => val1, arg2 => val2)	$nasc \langle\!| \; prefix \mapsto simp\ f$ $args \mapsto \langle \langle\!| \; formal \mapsto arg1,$ $actual \mapsto \ldots \; |\!\rangle,$ $\langle\!| \; formal \mapsto arg2,$ $actual \mapsto \ldots \; |\!\rangle \rangle$

5.5.1 Abstract Syntax

The prefix of a named association must be the name of a function. In the argument list, formal parameter identifiers are mapped to expressions (using the same syntax as procedure call actual parameters — *NamedActual*).

$$
\begin{array}{|l}
\hline
\textit{NAscName} \\
\textit{prefix} : \textit{Name} \\
\textit{args} : \mathrm{seq}_1\ \textit{NamedActual} \\
\hline
\end{array}
$$

$$\textit{Name} ::= \ldots \mid \textit{nasc}\langle\!\langle \textit{NAscName} \rangle\!\rangle$$

This form of function call is syntactically distinct from any term which can be used on the left hand side of an assignment statement; it would therefore be possible for this form of function call to belong to *Exp*, rather than *Name*. However, it seems more satisfactory for the two forms of function call to belong to the same syntactic category.

5.5.2 Dynamic Semantics

To evaluate a function call, the following steps must be taken:

1. Actual parameters are associated with formal parameters in the store ("copy-in");
2. Variables local to the function are set to their initial values;
3. The function body is executed with this new store;
4. Finally, the function return expression is evaluated in this store.

Note that SPARK functions cannot have side-effects, nor can parameter values be modified (so no "copy-out" is necessary after the call).

There are two cases to consider, according to whether the function *name* is simple or is prefixed by a package-identifier. We therefore define:

$$\dfrac{\begin{array}{l}\forall\, c, pc : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ val : Val;\\ \ fun : Id;\ fps : \operatorname{seq} FormalParam;\ st : Stmt;\\ \ exp : Exp;\ tid : IdDot;\ NAscName\\ \mid\\ \qquad prefix = simp\ fun\\ \qquad get\_id\_ctx(c, \delta, fun) = (pc \frown \langle fun \rangle, funI)\\ \qquad (\delta.dict\ pc).funs\ fun = (fps, st, exp, tid)\\ \qquad \sigma'' = clear\_locals(\sigma', \delta', pc \frown \langle fun \rangle)\\ \bullet\\ \qquad c, \delta, \sigma \vdash_{copyin} (pc \frown \langle fun \rangle, fps, args) \Longrightarrow_{copyin} \delta', \sigma'\\ \qquad pc \frown \langle fun \rangle, \delta', \sigma'' \vdash_s st \Longrightarrow_s \sigma'''\\ \qquad pc \frown \langle fun \rangle, \delta', \sigma''' \vdash_e exp \Longrightarrow_e val\end{array}}{c, \delta, \sigma \vdash_n nasc(\theta NAscName) \Longrightarrow_n val} \qquad \text{(NAsc1D)}$$

$$\dfrac{\begin{array}{l}\forall\, c, pc, fc : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ val : Val;\\ \ pak : Id;\ sn : SlctName;\ fps : \operatorname{seq} FormalParam;\\ \ st : Stmt;\ exp : Exp;\ tid : IdDot;\ PAscName\\ \mid\\ \qquad prefix = slct\ sn\\ \qquad sn.prefix = simp\ pak\\ \qquad get\_id\_ctx(c, \delta, pak) = (pc, pkgI)\\ \qquad get\_id\_ctx(pc, \delta, sn.selector) = (fc, funI)\\ \qquad (\delta.dict\ pc).funs\ sn.selector = (fps, st, exp, tid)\\ \qquad \sigma'' = clear\_locals(\sigma', \delta', fc)\\ \bullet\\ \qquad c, \delta, \sigma \vdash_{copyin} (fc, fps, args) \Longrightarrow_{copyin} \delta', \sigma'\\ \qquad fc, \delta', \sigma'' \vdash_s st \Longrightarrow_s \sigma'''\\ \qquad fc, \delta', \sigma''' \vdash_e exp \Longrightarrow_e val\end{array}}{c, \delta, \sigma \vdash_n pasc(\theta PAscName) \Longrightarrow_n val} \qquad \text{(NAsc2D)}$$

References: Env p. 11; *Store p. 12*; *Val p. 9*; *IdDot p. 8*; *FormalParam p. 160*; *Stmt p. 121*; *Exp p. 47*; *clear_locals* p. 156; $\vdash_s$ p. 122; $\Longrightarrow_s$ p. 122; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47; *get_id_ctx* p. 38; $\vdash_{copyin}$ p. 160; $\Longrightarrow_{copyin}$ p. 160; *SlctName p. 37*.

Chapter 6

Expressions

This chapter describes the expressions of SPARK (excluding *attributes*, which are described in Chapter 7). The Abstract Syntax of expressions *Exp* is summarised in table 6.1.

Evaluation Predicate

The evaluation of an expression is defined by a predicate which includes the context, the environment and the store. With $c : \operatorname{seq} Id$, $\delta : Env$, $\sigma : Store$, $exp : Exp$ and $val : Val$ the predicate

$$c, \delta, \sigma \vdash_e exp \Longrightarrow_e val$$

can be read as "expression *exp* evaluates in context c, environment δ and with store σ to the value *val*".

We also use, in this and other chapters, the short-hand notation

$$c, \delta, \sigma \vdash_{es} exps \Longrightarrow_{es} vals$$

to represent the evaluation of a sequence of expressions ($exps : \operatorname{seq} Exp$) to give a sequence of values ($vals : \operatorname{seq} Val$).
References: Env p. 11; Store p. 12; Val p. 9.

Syntax Constructor	Description	Page
lint	integer literal	50
lreal	real literal	51
lchar	character literal	52
lstrg	string literal	53
nam	name	54
dots	range expression	55
in	membership test	56
notin	complement membership test	57
qual	type qualification	58
pagg	positional association aggregate	59
paggoth	...aggregate with others	61
nagg	named association aggregate	62
naggoth	...aggregate with others	67
un	unary operator expressions	68
bin	binary operator expressions	69
andthen	short circuit conjunction	71
orelse	short circuit disjunction	73
cat	string concatenation	75
—	type conversion	76

Table 6.1: Abstract Syntax of Expressions *Exp*

6.1 Expression Types

Repetition of this section of the companion Static Semantics document is avoided here.

6.2 Integer Literals

Integer literals can be written in any base from 2 to 16, with an optional exponent.

Syntax Example	A.S. Representation
100	*lint* 100
2#111#e10	*lint* 28
16#FF	*lint* 255

6.2.1 Abstract Syntax

Integer literals are represented in the Abstract Syntax by numerals – written in the conventional decimal notation.

$$Exp ::= lint\langle\!\langle \mathbb{Z} \rangle\!\rangle$$

6.2.2 Dynamic Semantics

An integer value is represented by the appropriate projection of the value set.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ n : \mathbb{Z} \\ \bullet}{c, \delta, \sigma \vdash_e lint\ \ n \Longrightarrow_e intval\ \ n} \qquad \text{(LintD)}$$

References: Env p. 11; Store p. 12.

6.3 Real Literals

A real literal must contain dot (.) and is always written in decimal. An exponent is optional.

Syntax Example	A.S. Representation
1.21e1	*lreal* 12.1
3_000.1	*lreal* 3000.1

6.3.1 Abstract Syntax

Real literals are represented using the elements of the set Real.

$$Exp ::= \ldots \mid lreal \langle\langle \mathrm{Real} \rangle\rangle$$

6.3.2 Dynamic Semantics

The value of a real literal is the real value as projected into the set *Val*.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ r : \mathrm{Real} \\ \bullet}{c, \delta, \sigma \vdash_e lreal\ r \Longrightarrow_e realval\ r} \qquad \text{(LrealD)}$$

References: Env p. 11; Store p. 12.

6.4 Character Literal

Character literals enclose a keyboard character in single quotes.

Syntax Example	A.S. Representation
'a'	*lchar* a
'b'	*lchar* b

6.4.1 Abstract Syntax

A character is represented by an element of the set *Char*.

$$Exp ::= \ldots \mid lchar \langle\!\langle Char \rangle\!\rangle$$

6.4.2 Dynamic Semantics

The value of a character literal is represented by the enumeration literal of the corresponding standard type.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ ch : Char \\ \bullet}{c, \delta, \sigma \vdash_e lchar\ ch \Longrightarrow_e enumval\ (chartoenum\ ch)} \quad \text{(LCharD)}$$

The function *chartoenum* returns the distinguished predefined identifiers representing the character literals in SPARK; it is not further defined here. Refer to Appendix A for details of the representation of the standard character type in SPARK.
References: Env p. 11; Store p. 12; Char p. 7.

6.5 String Literals

A string literal encloses zero or more keyboard characters in double quotes.

Syntax Example	A.S. Representation
"a"	$\mathit{lstrg}\ \langle \mathrm{a} \rangle$
"bc"	$\mathit{lstrg}\ \langle \mathrm{b}, \mathrm{c} \rangle$

6.5.1 Abstract Syntax

A string literal is represented by a sequence of *Char*.

$$\mathit{Exp} ::= \ldots \mid \mathit{lstrg}\langle\!\langle \mathrm{seq}\ \mathit{Char} \rangle\!\rangle$$

6.5.2 Dynamic Semantics

String literals are evaluated by creating an appropriate array object into which to place the sequence of characters.

$$\frac{\begin{array}{l} \forall\ c : \mathrm{seq}\,\mathit{Id};\ \delta : \mathit{Env};\ \sigma : \mathit{Store};\ \mathit{cs} : \mathrm{seq}\,\mathit{Char}; \\ \quad \mathit{Array\_Value} \\ \mid \\ \qquad \#\mathit{cs} = \#\mathit{arr} \\ \qquad \mathit{lo} = 1 \\ \qquad \mathit{hi} = \#\mathit{arr} \\ \bullet \\ \qquad (\forall\ i \in \mathrm{dom}\ \mathit{cs} \bullet \\ \qquad\qquad c, \delta, \sigma \vdash_e \mathit{lchar}\ (\mathit{cs}\ \ i) \Longrightarrow_e \mathit{arr}\ \ i) \end{array}}{c, \delta, \sigma \vdash_e \mathit{lstrg}\ \ \mathit{cs} \Longrightarrow_e \mathit{arrval}(\theta \mathit{Array\_Value})} \quad \text{(LStrgD)}$$

References: Env p. 11; *Store p. 12*; *Char p. 7*; *Array_Value* p. 9.

6.6 Names Expressions

When a name is used as an expression it stands for the value of an object (simple, indexed or selected), the range of a type or a (fully parameterised) function call.

Syntax Example	A.S. Representation
V(1)	$nam\ pasc \langle\!|\ prefix \mapsto simp\ V,\ args \mapsto \langle lint\ 1 \rangle\ |\!\rangle$

6.6.1 Abstract Syntax

$Exp ::= \ldots \mid nam\langle\!\langle Name \rangle\!\rangle$

6.6.2 Dynamic Semantics

The value of the name expression is the value of the name; it must be an object.

$$\forall\ c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ n : Name;\ val : Val \\ \bullet \\ \frac{c, \delta, \sigma \vdash_n n \Longrightarrow_n val}{c, \delta, \sigma \vdash_e nam\ n \Longrightarrow_e val} \qquad \text{(Nam1D)}$$

References: Env p. 11; Store p. 12; Name p. 31; Val p. 9; $\vdash_n$ p. 31; $\Longrightarrow_n$ p. 31.

6.7 Range Expressions

Two expressions can be combined using dots (..) to give an expression representing a range of values.

Syntax Example	A.S. Representation
L .. U	$dots \langle\!\langle$ $lower \mapsto$ expression L, $upper \mapsto$ expression U $\rangle\!\rangle$

6.7.1 Abstract Syntax

The two expressions define the lower and upper bounds of the range.

$$DotsExp \mathrel{\widehat=} [lower, upper : Exp]$$

$$Exp ::= \ldots \mid dots \langle\!\langle DotsExp \rangle\!\rangle$$

6.7.2 Dynamic Semantics

The value of a range is a set of values within the range specified by the value of the *lower* and *upper* bound expressions.

$$\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ lv, uv : Val;\ DotsExp \bullet \frac{\begin{array}{l} c, \delta, \sigma \vdash_e lower \Longrightarrow_e lv \\ c, \delta, \sigma \vdash_e upper \Longrightarrow_e uv \end{array}}{c, \delta, \sigma \vdash_e dots\ (\theta DotsExp) \Longrightarrow_e rngval\ (lv \ {}_{V}\!\ldotp\!\ldotp_{\delta}\ uv)} \quad \text{(DotsD)}$$

In the above rule, the function ${}_{V}\!\ldotp\!\ldotp_{Env}$ is used to construct the appropriate set of values. This function is defined in the Static Semantics document.
References: Env p. 11; Store p. 12; Val p. 9.

6.8 Membership Tests

A membership test is true if the value belongs to the range of values indicated by the subtype or range.

Syntax Example	A.S. Representation
v **in** 1 .. 10	$in \langle\!\mid value \mapsto nam\ simp\ v,\ range \mapsto dots \langle\!\mid lower \mapsto lint\ 1,\ upper \mapsto lint\ 10 \mid\!\rangle \mid\!\rangle$

6.8.1 Abstract Syntax

The first term of the membership test expression is an expression representing a value; the second an expression representing some range of values.

$$InExp \mathrel{\widehat=} [value, range : Exp]$$

$$Exp ::= \ldots \mid in\langle\!\langle InExp \rangle\!\rangle$$

6.8.2 Dynamic Semantics

The expression evaluates to *true* if the value is in the range specified, and to *false* otherwise.

$$\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ ev : Val;\ rv : \mathbb{P}\, Val;\ InExp \\
\mid \\
\quad ev \in rv \\
\bullet \\
\quad c, \delta, \sigma \vdash_e value \Longrightarrow_e ev \\
\quad c, \delta, \sigma \vdash_e range \Longrightarrow_e rngval\ rv \\
\hline
\quad c, \delta, \sigma \vdash_e in(\theta InExp) \Longrightarrow_e enumval\ (id\ true)
\end{array} \tag{InD1}$$

$$\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ ev : Val;\ rv : \mathbb{P}\, Val;\ InExp \\
\mid \\
\quad ev \notin rv \\
\bullet \\
\quad c, \delta, \sigma \vdash_e value \Longrightarrow_e ev \\
\quad c, \delta, \sigma \vdash_e range \Longrightarrow_e rngval\ rv \\
\hline
\quad c, \delta, \sigma \vdash_e in(\theta InExp) \Longrightarrow_e enumval\ (id\ false)
\end{array} \tag{InD2}$$

References: Env p. 11; Store p. 12; Val p. 9.

6.9 Complement Membership Tests

A complement membership test is true if the value does not belong to the range of values indicated by the subtype or range.

Syntax Example	A.S. Representation
today **not in** weekday	$notin \langle\!| \; value \mapsto nam\ (simp\ today),\; range \mapsto nam\ (simp\ weekday) \; |\!\rangle$

6.9.1 Abstract Syntax

The first term of the membership test expression is an expression representing a value; the second an expression representing some range of values.

$$NotInExp \mathrel{\widehat{=}} [value, range : Exp]$$

$$Exp ::= \ldots \mid notin \langle\!\langle NotInExp \rangle\!\rangle$$

6.9.2 Dynamic Semantics

The expression evaluates to *false* if the value is in the range specified, and to *true* otherwise.

$$\dfrac{\begin{array}{l} \forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ ev : Val;\ rv : \mathbb{P}\ Val;\ NotInExp \\ \mid \\ \qquad ev \in rv \\ \bullet \\ \qquad c, \delta, \sigma \vdash_e value \Longrightarrow_e ev \\ \qquad c, \delta, \sigma \vdash_e range \Longrightarrow_e rngval\ rv \end{array}}{c, \delta, \sigma \vdash_e in(\theta NotInExp) \Longrightarrow_e enumval\ (id\ false)} \qquad \text{(NotInD1)}$$

$$\dfrac{\begin{array}{l} \forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ ev : Val;\ rv : \mathbb{P}\ Val;\ NotInExp \\ \mid \\ \qquad ev \notin rv \\ \bullet \\ \qquad c, \delta, \sigma \vdash_e value \Longrightarrow_e ev \\ \qquad c, \delta, \sigma \vdash_e range \Longrightarrow_e rngval\ rv \end{array}}{c, \delta, \sigma \vdash_e in(\theta NotInExp) \Longrightarrow_e enumval\ (id\ true)} \qquad \text{(NotInD2)}$$

References: Env p. 11; Store p. 12; Val p. 9.

6.10 Type Qualification

A type qualification is an assertion that a value belongs to a type.

Syntax Example	A.S. Representation
K.T'(V)	$qual \ \langle\!\vert \ typemark \mapsto dot\ (K, T),$ $value \mapsto nam\ (simp\ V) \ \vert\!\rangle$

6.10.1 Abstract Syntax

The type is represented by an *IdDot*; the value is an expression.

$$QualExp \mathrel{\widehat{=}} [typemark : IdDot;\ value : Exp]$$

$$Exp ::= \ldots \mid qual\langle\!\langle QualExp\rangle\!\rangle$$

6.10.2 Dynamic Semantics

The qualified expression must evaluate to a value within the range specified by the type-mark, in order to avoid the possibility of an Ada `CONSTRAINT_ERROR`.

$$\dfrac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ QualExp \\ \mid \\ \quad val \in typrange(\delta, typemark) \\ \bullet \\ \quad c, \delta, \sigma \vdash_e value \Longrightarrow_e val\end{array}}{c, \delta, \sigma \vdash_e qual(\theta QualExp) \Longrightarrow_e val} \quad \text{(QualD)}$$

The function *typrange* used in the above rule fetches the set of values associated with the subtype denoted by the type-mark; it is defined in the auxiliary functions section of this document.

References: Env p. 11; Store p. 12; Val p. 9; typrange p. 216.

6.11 Aggregates – Positional, without Others

An aggregate constructs a value of an array or record type. In SPARK, all aggregates are qualified by the type name. In this form of aggregate the *position* of an expression in the list of components determines which element of the array, or field of the record, takes each component value.

Syntax Example	A.S. Representation
T'(1,2)	$pagg \; \langle\!\mid \; typemark \mapsto id \;\; T, \; components \mapsto \langle lint \;\, 1, lint \;\, 2 \rangle \;\; \mid\!\rangle$

An aggregate of this form with only a single component is indistinguishable, in the concrete syntax, from a type qualification. Such an expression is always interpreted as a type qualification (see LRM 4.3, para 4).

6.11.1 Abstract Syntax

The type mark is represented by *IdDot*; the components are expressions.

$$PAggExp \mathrel{\widehat=} [typemark : IdDot; \; components : \mathrm{seq}_1 \, Exp]$$

$$Exp ::= \ldots \mid pagg \langle\!\langle PAggExp \rangle\!\rangle$$

6.11.2 Dynamic Semantics

Array Aggregates The component expressions of the array aggregate are evaluated and associated with the elements of the array-object; it is a static well-formedness requirement that the number of expressions should be equal to the number of elements in the array.

$$\frac{\begin{array}{l} \forall \, c : \mathrm{seq}\, Id; \; \delta : Env; \; \sigma : Store; \; PAggExp; \; vals : \mathrm{seq}_1 \, Val \\ \mid \\ \qquad typbounds_\delta \; (c, typemark) = (lo, hi) \wedge \\ \qquad arr = (\lambda \, i : lo \mathrel{..} hi \bullet vals(i - lo + 1)) \\ \bullet \\ \qquad c, \delta, \sigma \vdash_{es} components \Longrightarrow_{es} vals \end{array}}{c, \delta, \sigma \vdash_e pagg(\theta PAggExp) \Longrightarrow_e arrval(\theta Array\_Value)} \qquad \text{(Pagg1D)}$$

The function $typebounds_{Env}$ used in the above returns the lower and upper bounds of a (single-dimensional) array's index range.

Record Aggregates The component expressions of a record aggregate are evaluated and associated with the fields of the record-object, in the order of declaration of the fields in the field type. The requirement that the number of components should be equal to the number of fields in the record is a static one.

$$
\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ PAggExp;\\
\quad vals : \mathrm{seq}_1\, Val;\ flds : \mathrm{iseq}_1\, Id\\
\mid\\
\qquad rec\_field\_seq_\delta(typemark, c) = flds\\
\bullet\\
\qquad \dfrac{c, \delta, \sigma \vdash_{es} components \Longrightarrow_{es} vals}{\begin{array}{l} c, \delta, \sigma \vdash_e pagg(\theta PAggExp) \Longrightarrow_e \\ \qquad recval(\lambda\, i : \mathrm{ran}\, flds \bullet vals(flds^\sim i))\end{array}}
\end{array}
\qquad \text{(Pagg2D)}
$$

The function $rec\_field\_seq_{Env}$ used in the above returns the sequence of field-name identifiers associated with the record-type and is defined in the auxiliary functions section of this document.

References: Env p. 11; *Store p. 12*; *Val p. 9*; $typbounds_\delta$ p. 217; $\vdash_{es}$ p. 47; $\Longrightarrow_{es}$ p. 47; *Array_Value* p. 9; $rec\_field\_seq_\delta$ p. 216.

6.12 Aggregate – Positional, with Others

A positional aggregate with an **others** clause constructs a value of an array type. In SPARK, all aggregates are qualified by the type name. In this form of aggregate the *position* of an expression in the list of components determines which element of the array takes each component value. The final component of the aggregate is an others clause. SPARK does not allow the use of an **others** clause in a record aggregate.

Syntax Example	A.S. Representation
T'(1, **others** => 2)	$paggoth \langle\!| \; typemark \mapsto id\ T, \; components \mapsto \langle lint\ 1 \rangle, \; other \mapsto lint\ 2 \; |\!\rangle$

6.12.1 Abstract Syntax

The type mark is represented by *IdDot*; the components are expressions, collected into a list. The others clause is represented by an expression.

$$PAggOthExp \mathrel{\widehat{=}} [typemark : IdDot;\ components : \operatorname{seq} Exp;\ other : Exp]$$

$$Exp ::= \ldots \mid paggoth\langle\!\langle PAggOthExp \rangle\!\rangle$$

6.12.2 Dynamic Semantics

The component expressions are evaluated and associated with the elements of the array-object; for the one or more elements which are not associated a value by position, the value of the **others** expression is used. It is a static well-formedness requirement that the number of expressions should be not exceed the number of elements in the array.

$$\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ vals : \operatorname{seq} Val;\\
\quad ov : Val;\ PAggOthExp\\
\mid\\
\qquad typbounds_{\delta}\ (c, typemark) = (lo, hi) \land\\
\qquad arr = (\lambda\, i : lo \mathinner{..} hi \bullet ov) \oplus\\
\qquad\qquad (\lambda\, i : lo \mathinner{..} lo + (\#vals) - 1 \bullet vals(i - lo + 1))\\
\bullet\\
\qquad c, \delta, \sigma \vdash_{es} components \Longrightarrow_{es} vals\\
\qquad c, \delta, \sigma \vdash_{e} other \Longrightarrow_{e} ov\\
\hline
\qquad c, \delta, \sigma \vdash_{e} paggoth(\theta PAggOthExp) \Longrightarrow_{e}\\
\qquad\qquad arrval(\theta Array_Value)
\end{array} \tag{PaggOthD}$$

The function $typebounds_{Env}$ used in the above returns the lower and upper bounds of a (single-dimensional) array's index range.

References: Env p. 11; *Store p. 12*; *Val p. 9*; $typbounds_{\delta}$ p. 217; $\vdash_{es}$ p. 47; $\Longrightarrow_{es}$ p. 47.

6.13 Aggregate – Named, without Others

An aggregate constructs a value of an array or record. In SPARK, all aggregates are qualified by the type name. In this form of aggregate an array index value or a record field name explicitly determines which element of the result takes each component value.

Syntax Example	A.S. Representation
COMPLEX'(RE => 1.0, IM => 2.0)	$nagg \langle\!\lvert\ typemark \mapsto id\ COMPLEX$ $assocs \mapsto \langle\ \langle\!\lvert\ choice \mapsto nam\ (simp\ RE),$ $component \mapsto lreal\ 1.0\ \rvert\!\rangle,$ $\langle\!\lvert\ choice \mapsto nam\ (simp\ IM),$ $component \mapsto lreal\ 2.0\ \rvert\!\rangle \rangle\ \rvert\!\rangle$
T'(1| 2 => 10)	$nagg \langle\!\lvert\ typemark \mapsto id\ T$ $assocs \mapsto \langle\ \langle\!\lvert\ choice \mapsto lint\ 1,$ $component \mapsto lint\ 10\ \rvert\!\rangle,$ $\langle\!\lvert\ choice \mapsto lint\ 2,$ $component \mapsto lint\ 10\ \rvert\!\rangle \rangle\ \rvert\!\rangle$

An aggregate with only a single component can be written using named association.

6.13.1 Abstract Syntax

Each association has a choice expression, specifying the record field or the array index, and a component expression.

$$NAggAssoc \mathrel{\widehat{=}} [choice : Exp;\ component : Exp]$$

The type mark is represented by *IdDot*. There is a list of associations.

$$NAggExp \mathrel{\widehat{=}} [typemark : IdDot;\ assocs : \mathrm{seq}_1\ NAggAssoc]$$

A component association containing more than one choice (separated by |, as in the second example above) is taken as an abbreviation for the longer form in which the expression is repeated for each choice.

$$Exp ::= \ldots \mid nagg\langle\!\langle NAggExp \rangle\!\rangle$$

6.13.2 Dynamic Semantics

Array Aggregates An initial array value is constructed with the $form\_init\_val_\delta$ function defined in chapter 3; this is then updated successively by the values associated with each given index in the aggregate expression. The requirement that each array element is assigned precisely one value by the aggregate expression is a static requirement, dealt with by the Static Semantics of SPARK; it is therefore assumed to be the case here.

We define named association of an array aggregate using some additional evaluation rules. For the evaluation of single associations, we introduce:

$$\begin{array}{|l}
_ \vdash_{naa} _ : (((\operatorname{seq} Id) \times Env \times Store) \times NAggAssoc) \rightarrow \\
\qquad (((\operatorname{seq} Id) \times Env \times Store) \times NAggAssoc) \\
_ \Longrightarrow_{naa} _ : (((\operatorname{seq} Id) \times Env \times Store) \times NAggAssoc) \rightarrow (\mathbb{Z} \times Val) \\
\hline
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ na : NAggAssoc \bullet \\
\qquad (c, \delta, \sigma \vdash_{naa} na) = ((c, \delta, \sigma), na)
\end{array}$$

and define

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ i : \mathbb{Z};\\ \quad cval : Val;\ NAggAssoc \\ \bullet \\ \qquad c, \delta, \sigma \vdash_e choice \Longrightarrow_e intval\ i \qquad c, \delta, \sigma \vdash_e component \Longrightarrow_e cval\end{array}}{c, \delta, \sigma \vdash_{naa} (\theta NAggAssoc) \Longrightarrow_{naa} (i, cval)} \qquad \text{(NArrAggAssoc)}$$

for the evaluation of a single named association. For the evaluation of a sequence of such associations, we then use:

$$\begin{array}{|l}
_ \vdash_{naas} _ : (((\operatorname{seq} Id) \times Env \times Store) \times \operatorname{seq} NAggAssoc) \rightarrow \\
\qquad (((\operatorname{seq} Id) \times Env \times Store) \times \operatorname{seq} NAggAssoc) \\
_ \Longrightarrow_{naas} _ : (((\operatorname{seq} Id) \times Env \times Store) \times \operatorname{seq} NAggAssoc) \rightarrow \operatorname{seq}(\mathbb{Z} \times Val) \\
\hline
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ nas : \operatorname{seq} NAggAssoc \bullet \\
\qquad (c, \delta, \sigma \vdash_{naas} na) = ((c, \delta, \sigma), nas)
\end{array}$$

with

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store \\ \bullet\end{array}}{c, \delta, \sigma \vdash_{naas} \langle\rangle \Longrightarrow_{naas} \varnothing} \qquad \text{(NArrAggAssocSeq1)}$$

and

$$\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ na : NAggAssoc;\\
\quad ns : \mathrm{seq}\, NAggAssoc;\ nv, sv : Id \nrightarrow Val\\
\bullet\\
\qquad \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_{naa} na \Longrightarrow_{naa} v\\ c, \delta, \sigma \vdash_{naas} ns \Longrightarrow_{naas} s\end{array}}{c, \delta, \sigma \vdash_{naas} \langle na \rangle \frown ns \Longrightarrow_{naa} \langle v \rangle \frown s}
\end{array} \qquad \text{(NArrAggAssocSeq2)}$$

We are now in a position to define the rule for evaluation of aggregates using named association for array objects:

$$\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ NAggExp;\ s : \mathrm{seq}(\mathbb{Z} \times Val)\\
\quad av, av' : Array_Value\\
|\\
\qquad is_arr_tmark_\delta\ typemark\\
\qquad form_init_val_\delta\ (c, typemark) \in \mathrm{ran}\ arrval\\
\qquad av = arrval^\sim form_init_val_\delta\ (c, typemark)\\
\bullet\\
\qquad \dfrac{c, \delta, \sigma \vdash_{naas} assocs \Longrightarrow_{naas} s}{c, \delta, \sigma \vdash_e nagg(\theta NAggExp) \Longrightarrow_e arrval\ \ av}
\end{array} \qquad \text{(NAgg1D)}$$

where

$$av' == arr\_agg\_override(av, s)$$

In the above, the function $arr\_agg\_override$ is used; this may be defined by:

$$\begin{array}{|l}
arr_agg_override : (Array_Value \times \mathrm{seq}(\mathbb{Z} \times Val)) \nrightarrow Array_Value\\
\hline
\forall\, av : Array_Value \bullet\\
\qquad arr_agg_override(av, \langle\rangle) = av\\
\forall\, av, av' : Array_Value;\ i : \mathbb{Z};\ v : Val;\ rest : \mathrm{seq}(\mathbb{Z} \times Val) \bullet\\
\qquad (av'.lo = av.lo \wedge av'.hi = av.hi \wedge\\
\qquad\ av'.hi = arr_agg_override(av, rest).arr \oplus \{i \mapsto v\})\\
\qquad\qquad \Rightarrow arr_agg_override(av, \langle (i, v) \rangle \frown rest) = av'
\end{array}$$

(This function is taken to be total, because the constraint that all indices are within range is enforced by the Static Semantics, whose checks are assumed here.)

Record Aggregates The requirement that all record fields should have precisely one association within an aggregate is a static one.

We define named association evaluation of a record aggregate using some additional evaluation rules. For the evaluation of single associations, we introduce:

$$
\begin{array}{|l}
\_ \vdash_{nra} \_ : (((\mathrm{seq}\, Id) \times Env \times Store) \times NAggAssoc) \rightarrow \\
\quad (((\mathrm{seq}\, Id) \times Env \times Store) \times NAggAssoc) \\
\_ \Longrightarrow_{nra} \_ : (((\mathrm{seq}\, Id) \times Env \times Store) \times NAggAssoc) \rightarrow (Id \nrightarrow Val) \\
\hline
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ na : NAggAssoc \bullet \\
\quad (c, \delta, \sigma \vdash_{nra} na) = ((c, \delta, \sigma), na)
\end{array}
$$

and define

$$
\frac{\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ fld : Id; \\
\ val : Val;\ NAggAssoc \\
| \\
\quad choice = nam\ (simp\ fld)) \\
\bullet \\
\quad c, \delta, \sigma \vdash_e component \Longrightarrow_e val
\end{array}}{c, \delta, \sigma \vdash_{nra} (\theta NAggAssoc) \Longrightarrow_{nra} \{fld \mapsto val\}} \qquad \text{(NAggAssoc)}
$$

for the evaluation of a single named association. For the evaluation of a sequence of such associations, we then use:

$$
\begin{array}{|l}
\_ \vdash_{nras} \_ : (((\mathrm{seq}\, Id) \times Env \times Store) \times \mathrm{seq}\, NAggAssoc) \rightarrow \\
\quad (((\mathrm{seq}\, Id) \times Env \times Store) \times \mathrm{seq}\, NAggAssoc) \\
\_ \Longrightarrow_{nras} \_ : (((\mathrm{seq}\, Id) \times Env \times Store) \times \mathrm{seq}\, NAggAssoc) \rightarrow (Id \nrightarrow Val) \\
\hline
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ nas : \mathrm{seq}\, NAggAssoc \bullet \\
\quad (c, \delta, \sigma \vdash_{nras} na) = ((c, \delta, \sigma), nas)
\end{array}
$$

with

$$
\frac{\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store \\
\bullet
\end{array}}{c, \delta, \sigma \vdash_{nras} \langle\rangle \Longrightarrow_{nras} \varnothing} \qquad \text{(NRecAggAssocSeq1)}
$$

and

$$\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\; \delta : Env;\; \sigma : Store;\; na : NAggAssoc; \\ \;\; ns : \mathrm{seq}\, NAggAssoc;\; nv, sv : Id \nrightarrow Val \\ \bullet \\ \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_{nra} na \Longrightarrow_{nra} nv \\ c, \delta, \sigma \vdash_{nras} ns \Longrightarrow_{nras} sv \end{array}}{c, \delta, \sigma \vdash_{nras} \langle na \rangle \frown ns \Longrightarrow_{nra} (nv \oplus sv)} \end{array} \qquad \text{(NRecAggAssocSeq2)}$$

We are now in a position to define the rule for evaluation of aggregates using named association for record objects:

$$\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\; \delta : Env;\; \sigma : Store;\; NAggExp;\; rv : Id \nrightarrow Val \\ \bullet \\ \dfrac{c, \delta, \sigma \vdash_{nras} assocs \Longrightarrow_{nras} rv}{c, \delta, \sigma \vdash_{e} nagg(\theta NAggExp) \Longrightarrow_{e} recval\ rv} \end{array} \qquad \text{(NAgg2D)}$$

References: Env p. 11; *Store p. 12*; *Val p. 9*; *Array_Value* p. 9; $is\_arr\_tmark_{\delta}$ p. 206.

6.14 Aggregate – Named, with Others

In this form of aggregate an array index value explicitly determines which element of the result takes each component value; an **others** clause can be used as the last element of the aggregate, to give a value to components not already determined. In SPARK, an **others** clause cannot be used in a record aggregate.

Syntax Example	A.S. Representation
K.T'(1.. 3 => 10, **others** => 11)	$pagg \langle\!\vert\ typemark \mapsto dot(K, T), assocs \mapsto \langle \langle\!\vert\ choice \mapsto dots \langle\!\vert\ lower \mapsto lint\ 1, upper \mapsto lint\ 3\ \vert\!\rangle, component \mapsto lint\ 10\ \vert\!\rangle \rangle, other \mapsto lint\ 11\ \vert\!\rangle$

6.14.1 Abstract Syntax

The type mark is represented by *IdDot*. There is a list of associations. The **others** clause is represented by an expression.

$$\begin{array}{|l}
\hline
\textit{NAggOthExp} \\
\textit{typemark} : \textit{IdDot} \\
\textit{components} : \mathrm{seq}\, \textit{NAggAssoc} \\
\textit{other} : \textit{Exp} \\
\hline
\end{array}$$

The schema *NAggAssoc* is defined on page 62.

$$Exp ::= \ldots \mid naggoth \langle\!\langle NAggOthExp \rangle\!\rangle$$

6.14.2 Dynamic Semantics

Not complete.

6.15 Unary Operators

The unary operators of SPARK are numeric plus and minus, the absolute value and logical not.

Syntax Example	A.S. Representation
not OPEN	$un\ \langle\!\mid\ op \mapsto not,\ arg \mapsto nam\ (simp\ OPEN)\ \mid\!\rangle$
- 100	$un\ \langle\!\mid\ op \mapsto uminus,\ arg \mapsto lint\ 100\ \mid\!\rangle$

6.15.1 Abstract Syntax

The unary operators belong to the set *Uop*.

$$Uop ::= uplus \mid uminus \mid abs \mid not$$

A unary operator expression has a single argument, which is an expression.

$$UnExp \mathrel{\widehat=} [op : Uop;\ arg : Exp]$$

$$Exp ::= \ldots \mid un\langle\!\langle UnExp\rangle\!\rangle$$

6.15.2 Dynamic Semantics

The argument must evaluate. In Ada, there exists the possibility that the exception `NUMERIC_ERROR` could be raised [LRM, 4.5(7)] by the evaluation of a unary prefix expression; this is implementation-dependent.

$$\frac{\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ UnExp \bullet \quad c, \delta, \sigma \vdash_e arg \Longrightarrow_e val}{c, \delta, \sigma \vdash_e un(\theta\, UnExp) \Longrightarrow_e apply\_uop(op, val)} \quad \text{(UnD)}$$

The function *apply_uop* is defined in the auxiliary functions section of this document.
References: Env p. 11; Store p. 12; Val p. 9; apply_uop p. 207.

6.16 Binary Operators

The binary operators of SPARK include the logical operators, the relational operators and the arithmetic operators. "Catenation", which has only restricted use in SPARK, is not regarded as an operator (see Section 6.19).

Syntax Example	A.S. Representation
open **or** failed	$bin \langle\!| \; larg \mapsto nam\ (simp\ open),\ op \mapsto or,\ rarg \mapsto nam\ (simp\ failed) \; |\!\rangle$
A = 42	$bin \langle\!| \; larg \mapsto nam\ (simp\ A),\ op \mapsto eq,\ rarg \mapsto lint\ 42 \; |\!\rangle$
B > 5	$bin \langle\!| \; larg \mapsto nam\ (simp\ B),\ op \mapsto gt,\ rarg \mapsto lint\ 5 \; |\!\rangle$
C mod 13	$bin \langle\!| \; larg \mapsto nam\ (simp\ C),\ op \mapsto mod,\ rarg \mapsto lint\ 13 \; |\!\rangle$

6.16.1 Abstract Syntax

$$\begin{aligned} Bop ::= {} & and \mid or \mid xor \mid eq \mid noteq \mid lt \mid lte \mid gt \mid gte \\ \mid {} & plus \mid minus \mid mul \mid div \mid mod \mid rem \mid power \end{aligned}$$

A binary expression combines left and right arguments using an operator.

$$BinExp \mathrel{\widehat{=}} [larg, rarg : Exp;\ op : Bop]$$

$$Exp ::= \ldots \mid bin\langle\!\langle BinExp \rangle\!\rangle$$

6.16.2 Dynamic Semantics

An infix represents a value obtained from its two argument expressions by applying the operator to the values of the two operands.

In Ada, evaluation of an infix expression may cause an exception to be raised in a number of ways:

- `NUMERIC_ERROR` may be raised [LRM: 4.5(7), 4.5.5(12)] if the calculation of the result overflows;

- `CONSTRAINT_ERROR` may be raised if integer exponentiation is to a negative power [LRM, 4.5.6(6)]; and
- `CONSTRAINT_ERROR` may be raised if two boolean arrays which are the arguments of a logical infix operator are of different sizes [LRM, 4.5.1(3)].

The first of these is implementation-dependent; the others are guarded against in the definition of the function *apply_bop* which is used below.

$$\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ lval, rval : Val;\ BinExp \\ \bullet \\ \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_e larg \Longrightarrow_e lval \\ c, \delta, \sigma \vdash_e rarg \Longrightarrow_e rval \end{array}}{c, \delta, \sigma \vdash_e bin(\theta BinExp) \Longrightarrow_e apply\_bop(op, lval, rval)} \end{array} \qquad \text{(BinD)}$$

The function *apply_bop* is defined in the auxiliary functions section of this document.

References: Env p. 11; Store p. 12; Val p. 9; apply_bop p. 208.

6.17 Short Circuit Form – and then

SPARK provides the short circuit form **and then**, which is logically equivalent to the boolean **and** operator, but which does not evaluate its right argument if the result can be determined from the left argument alone.

Syntax Example	A.S. Representation
A **/=** 0 **and then** B **/** A = 0	$andthen \langle\!| \; larg \mapsto bin \langle\!| \; larg \mapsto nam\ (simp\ A),\ op \mapsto noteq,\ rarg \mapsto lint\ 0 \; |\!\rangle,\ rarg \mapsto bin \langle\!| \; larg \mapsto bin \langle\!| \; larg \mapsto nam\ (simp\ B),\ op \mapsto div,\ rarg \mapsto nam\ (simp\ A) \; |\!\rangle,\ op \mapsto eq,\ rarg \mapsto lint\ 0 \; |\!\rangle \; |\!\rangle$

6.17.1 Abstract Syntax

Both arguments of the short circuit form **and then** are expressions.

$$AndThenExp \mathrel{\widehat=} [larg, rarg : Exp]$$

$$Exp ::= \ldots \mid andthen\langle\!\langle AndThenExp \rangle\!\rangle$$

6.17.2 Dynamic Semantics

In the case where both arguments are dynamically well-formed (and thus evaluate to either *true* or *false*), the evaluation is equivalent to the binary operator **and** ; where the first expression evaluates to *false*, however, we do not need to consider the value of the second expression and can return *false* regardless of the well-formedness (or value) of the second argument. We therefore require two rules:

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ AndThenExp \\ \bullet \\ \quad c, \delta, \sigma \vdash_e larg \Longrightarrow_e enumval\ (id\ false)\end{array}}{c, \delta, \sigma \vdash_e andthen(\theta AndThenExp) \Longrightarrow_e enumval\ (id\ false)} \qquad \text{(AndThD1)}$$

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ AndThenExp \\ \bullet \\ \quad c, \delta, \sigma \vdash_e larg \Longrightarrow_e enumval\ (id\ true) \\ \quad c, \delta, \sigma \vdash_e rarg \Longrightarrow_e val\end{array}}{c, \delta, \sigma \vdash_e andthen(\theta AndThenExp) \Longrightarrow_e val} \qquad \text{(AndThD2)}$$

References: Env p. 11; Store p. 12; Val p. 9.

6.18 Short Circuit Form – or else

SPARK provides the short circuit form **or else**, which is logically equivalent to the boolean or operator, but which does not evaluates its right argument if the result can be determined from the left argument alone.

Syntax Example	A.S. Representation
i = 0 **or else** A(i) = 0	$orelse \ \langle\!\vert\ larg \mapsto bin \ \langle\!\vert\ larg \mapsto nam\ (simp\ i), op \mapsto eq, rarg \mapsto lint\ 0\ \vert\!\rangle, rarg \mapsto bin\ \langle\!\vert\ larg \mapsto pasc\ \langle\!\vert\ prefix \mapsto nam\ (simp\ A), args \mapsto \langle nam\ (simp\ i)\ \rangle\ \vert\!\rangle, op \mapsto eq, rarg \mapsto lint\ 0\ \vert\!\rangle\ \vert\!\rangle$

6.18.1 Abstract Syntax

Both arguments of the short circuit form **or else** are expressions.

$$OrElseExp \mathrel{\widehat{=}} [larg, rarg : Exp]$$

$$Exp ::= \ldots \mid orelse\langle\!\langle OrElseExp \rangle\!\rangle$$

6.18.2 Dynamic Semantics

In the case where both arguments are dynamically well-formed (and thus evaluate to either *true* or *false*), the evaluation is equivalent to the binary operator **or** ; where the first expression evaluates to *true*, however, we do not need to consider the value of the second expression and can return *true* regardless of the well-formedness (or value) of the second argument. We therefore require two rules:

$$\frac{\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ OrElseExp \\ \bullet \\ c, \delta, \sigma \vdash_e larg \Longrightarrow_e enumval\ (id\ true)}{c, \delta, \sigma \vdash_e orelse(\theta OrElseExp) \Longrightarrow_e enumval\ (id\ true)} \quad \text{(OrElD1)}$$

$$\frac{\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ OrElseExp \\ \bullet \\ c, \delta, \sigma \vdash_e larg \Longrightarrow_e enumval\ (id\ false) \\ c, \delta, \sigma \vdash_e rarg \Longrightarrow_e val}{c, \delta, \sigma \vdash_e orelse(\theta OrElseExp) \Longrightarrow_e val} \quad \text{(OrElD2)}$$

References: Env p. 11; Store p. 12; Val p. 9.

6.19 Catenation

"Catenation" can only be used in SPARK to construct string literals.

Syntax Example	A.S. Representation
"A" & "a"	$concat \langle\!\mid\ larg \mapsto lstrg\ \langle A \rangle,\ rarg \mapsto lstrg\ \langle a \rangle\ \mid\!\rangle$

6.19.1 Abstract Syntax

In the Abstract Syntax, both the operands of "catenation" are expressions.

$$CatExp \mathrel{\widehat{=}} [larg, rarg : Exp]$$

$$Exp ::= \ldots \mid cat\langle\!\langle CatExp \rangle\!\rangle$$

6.19.2 Dynamic Semantics

The value of a "catenation" is the value of the single string literal formed by joining the two sequences of characters.

$$\frac{\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ val : Val; \\ \ cs_1, cs_2 : \mathrm{seq}\, Char;\ CatExp; \\ \mid \\ \quad larg = lstrg\ cs_1 \\ \quad rarg = lstrg\ cs_2 \\ \bullet \\ \quad c, \delta, \sigma \vdash_e lstrg\ (cs_1 \frown cs_2) \Longrightarrow_e val \end{array}}{c, \delta, \sigma \vdash_e cat(\theta\, CatExp) \Longrightarrow_e val} \qquad \text{(CatD)}$$

N.B. A "catenation" is only well-formed if its two arguments are both string literals. This is a static constraint.

References: Env p. 11; Store p. 12; Val p. 9; Char p. 7.

6.20 Type Conversions

Type conversions are syntactically part of the syntactic category of names, in the form of a positional association with a single argument. See 41 for a discussion of positional associations.

(This section is present for ease of reference only.)

Chapter 7

Attribute Expressions

This Chapter describes the attributes allowed in SPARK. Attributes are a form of expression, belonging to the Abstract Syntax category *Exp*. Other expressions are described in Chapter 6. The Abstract Syntax of attributes is summarised in the following table:

Syntax Constructor	Description	Page
rng	Range of the n'th array index type	78
fst	First element of a scalar type	79
bfst	First element of the base type of a scalar type	79
lst	Last element of a scalar type	80
blst	Last element of the base type of a scalar type	80
fsta	First element of the n'th array index type	81
lsta	Last element of the n'th array index type	85
succ	Successor of an element of a discrete type	89
pred	Predecessor of an element of a discrete type	91
pos	Position of an element of discrete type	93
val	Value of an element of discrete type	94
size	Size of an object	96

This section on attributes is preliminary — a number of attributes which are allowed in SPARK have been omitted. In addition, we assume that base-range (see **SLI WJ012***) and optional arguments have been removed from SPARK (see* **SLI WJ010***).*

7.1 Range Attribute

A RANGE attribute can be applied to a type mark of an array type (or a formal parameter of an unconstrained array type). An argument selects one of the index types of the array. The result is the range of the index type.

Syntax Example	A.S. Representation
t'RANGE(3)	$rng \langle\!| \; arrtyp \mapsto id\ t, \; indexno \mapsto lint\ 3 \; |\!\rangle$

7.1.1 Abstract Syntax

$$\begin{array}{|l} \textit{RngExp} \\ \hline arrtyp : IdDot \\ indexno : Exp \\ \hline \end{array}$$

$$Exp ::= \ldots \mid rng\langle\!\langle RngExp \rangle\!\rangle$$

7.1.2 Dynamic Semantics

In defining this attribute, we may make use of the 'FIRST and 'LAST attributes defined elsewhere in this chapter to generate the required bounds, then create the set of values in the range delimited by these values.

$$\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ lo, hi : Val; \\ \quad RngExp;\ FstAExp;\ LstAExp \\ \bullet \\ \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_e fsta(\theta FstAExp) \Longrightarrow_e lo \\ c, \delta, \sigma \vdash_e lsta(\theta LstAExp) \Longrightarrow_e hi \end{array}}{c, \delta, \sigma \vdash_e rng(\theta RngExp) \Longrightarrow_e rngval\ (lo\ {}_{V}\!..{}_{\delta}\ hi)} \end{array} \qquad \text{(RngD)}$$

In the above rule, we have used the auxiliary function ${}_{V}..{}_{\delta}$, defined in the Static Semantics document.

References: Env p. 11; Store p. 12; Val p. 9.

7.2 First and Base First

The FIRST attribute returns the least element of a scalar type, or of the base type of a scalar type.

Syntax Example	A.S. Representation
t'FIRST	*fst* (*id* *t*)
t'BASE'FIRST	*bfst* (*id* *t*)

7.2.1 Abstract Syntax

$Exp ::= \ldots \mid fst\langle\!\langle IdDot\rangle\!\rangle \mid bfst\langle\!\langle IdDot\rangle\!\rangle$

7.2.2 Dynamic Semantics

The value returned is the minimum value of the subtype range associated with the given type-mark or, for base first, the minimum value of the base type associated with the type-mark.

$$\frac{\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ typ : IdDot \\ \bullet}{c, \delta, \sigma \vdash_e fst\ typ \Longrightarrow_e min\ \{x : Val \mid x \in typrange(c, \delta, typ)\}} \quad \text{(FstD)}$$

$$\frac{\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ typ : IdDot \\ \bullet}{c, \delta, \sigma \vdash_e bfst\ typ \Longrightarrow_e \\ min\ \{x : Val \mid x \in typrange(c, \delta, (ancestorof_\delta\, typ))\}} \quad \text{(BfstD)}$$

References: Env p. 11; Store p. 12; IdDot p. 8; typrange p. 216; ancestorof$_\delta$ p. 206.

7.3 Last and Base Last

The LAST attribute returns the greatest element of a scalar type, or of the base type of a scalar type.

Syntax Example	A.S. Representation
t'LAST	*lst* (*id t*)
t'BASE'LAST	*blst* (*id t*)

7.3.1 Abstract Syntax

$Exp ::= \ldots \mid lst \langle\!\langle IdDot \rangle\!\rangle \mid blst \langle\!\langle IdDot \rangle\!\rangle$

7.3.2 Dynamic Semantics

The value returned is the maximum value of the subtype range associated with the given type-mark or, for base last, the maximum value of the base type associated with the type-mark.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ typ : IdDot \\ \bullet}{c, \delta, \sigma \vdash_e lst\ typ \Longrightarrow_e max\ \{x : Val \mid x \in typrange(c, \delta, typ)\}} \quad \text{(LstD)}$$

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ typ : IdDot \\ \bullet}{c, \delta, \sigma \vdash_e blst\ typ \Longrightarrow_e \\ \quad max\ \{x : Val \mid x \in typrange(c, \delta, (ancestorof_\delta\, typ))\}} \quad \text{(BlstD)}$$

References: Env p. 11; *Store p. 12*; *IdDot p. 8*; *typrange p. 216*; *ancestorof*$_\delta$ p. 206.

7.4 First of an Array Index Type

The `'FIRST` attribute can be applied to a type mark of an array type (or a formal parameter of an unconstrained array type) to obtain the least element of one of the index (sub)types of the array. An argument specifies to which index type the attribute is applied.

Syntax Example	A.S. Representation
sensors'FIRST(1)	$fsta \langle\!| \; arrtyp \mapsto sensors,$ $indexno \mapsto lint\ 1 \; |\!\rangle$

7.4.1 Abstract Syntax

$$\begin{array}{|l}
\hline FstAExp \\
arrtyp : IdDot \\
indexno : Exp \\
\hline
\end{array}$$

$Exp ::= \ldots \mid fsta\langle\!\langle FstAExp \rangle\!\rangle$

7.4.2 Dynamic Semantics

If the prefix is the array *type mark*, we look up the appropriate index-type in the dictionary; if it is an array *object*, we look at its index range as stored in the array value in the store. The index number must be an integer in the range $1 \mathrel{..} n$, where n is the dimensionality of the array. The first two rules deal with a type mark (simple name, then a selected name).

$$\begin{array}{l}
\forall\, c, pc : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ n : \mathbb{N}_1; \\
\quad t : IdDot;\ FstAExp \\
\mid \\
\qquad arrtyp \in \operatorname{ran} id \\
\qquad get_id_ctx(c, \delta, id^{\sim} arrtyp) = (pc, typeI) \\
\qquad (\delta.dict\ pc).type\ (id^{\sim} arrtyp) \in \operatorname{ran} arrT \\
\qquad n \leq \#\ (arrT^{\sim}((\delta.dict\ pc).type\ (id^{\sim} arrtyp))).indexes \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e indexno \Longrightarrow_e lintn \\
\hline
\qquad c, \delta, \sigma \vdash_e fsta(\theta FstAExp) \Longrightarrow_e \\
\qquad\qquad get_type_first(c, \delta, t)
\end{array} \qquad \text{(FstAD1)}$$

where

$t == (arrT^{\sim}((\delta.dict\ pc).type\ (id^{\sim} arrtyp))).indexes\ n$

$$
\begin{array}{l}
\forall\, c, pc : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ k, t, t' : Id;\ n : \mathbb{N}_1;\ FstAExp \\
\mid \\
\qquad arrtyp = dot(k, t) \\
\qquad get\_id\_ctx(c, \delta, k) = (pc, pkgI) \\
\qquad t \in \operatorname{dom}(\delta.dict\ pc).type \\
\qquad (\delta.dict\ pc).type\ t \in \operatorname{ran} arrT \\
\qquad n \leq \#\ (arrT^{\sim}((\delta.dict\ pc).type\ t)).indexes \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e indexno \Longrightarrow_e lintn \\
\hline
\qquad c, \delta, \sigma \vdash_e fsta(\theta FstAExp) \Longrightarrow_e \\
\qquad\qquad get\_type\_first(pc, \delta, t')
\end{array}
\tag{FstAD2}
$$

where

$$t' == (arrT^{\sim}((\delta.dict\ pc).type\ t)).indexes\ n$$

In the above two rules, the auxiliary function *get_type_first* is used to fetch the starting value of the index range type (its third argument); this is defined at the end of this section.

The next two rules deal with an array object as prefix to the attribute.

$$
\begin{array}{l}
\forall\, c, pc : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ n : \mathbb{N}_1; \\
\ t : IdDot;\ FstAExp \\
\mid \\
\qquad arrtyp \in \operatorname{ran} id \\
\qquad get\_id\_ctx(c, \delta, id^{\sim} arrtyp) = (pc, varI) \\
\qquad av \in \operatorname{ran} arrval \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e indexno \Longrightarrow_e lintn \\
\qquad c, \delta, \sigma \vdash_e arrtyp \Longrightarrow_e av \\
\hline
\qquad c, \delta, \sigma \vdash_e fsta(\theta FstAExp) \Longrightarrow_e \\
\qquad\qquad get\_index\_first(av, n)
\end{array}
\tag{FstAD3}
$$

$$
\begin{array}{l}
\forall\, c, pc, pc' : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ n : \mathbb{N}_1; \\
\ k, t : IdDot;\ FstAExp \\
| \\
\quad arrtyp = dot(k, t) \\
\quad get\_id\_ctx(c, \delta, k) = (pc, pkgI) \\
\quad get\_id\_ctx(pc, \delta, t) = (pc', varI) \\
\quad av \in \mathrm{ran}\ arrval \\
\bullet \\
\quad c, \delta, \sigma \vdash_e indexno \Longrightarrow_e lintn \\
\quad c, \delta, \sigma \vdash_e arrtyp \Longrightarrow_e av \\
\hline
\quad c, \delta, \sigma \vdash_e fsta(\theta FstAExp) \Longrightarrow_e \\
\qquad get\_index\_first(av, n)
\end{array}
\tag{FstAD4}
$$

In the above two rules, the auxiliary function *get_index_first* is used. This may be defined by:

$$
\begin{array}{|l}
get\_index\_first : (Val \times \mathbb{N}_1) \rightarrow\!\!\!\!\!\!+\ Val \\
\hline
\forall\, v : Val \bullet \\
\quad (v \in \mathrm{ran}\ arrval) \Rightarrow \\
\qquad get\_index\_first(v, 1) = intval\ (arrval^{\sim} v).lo \\
\forall\, v : Val;\ n : \mathbb{N}_1 \bullet \\
\quad (v \in \mathrm{ran}\ arrval) \Rightarrow \\
\qquad get\_index\_first(v, n + 1) = \\
\qquad\quad get\_index\_first((arrval^{\sim} v).arr\ (arrval^{\sim} v).lo, n)
\end{array}
$$

Finally, we define the *get_type_first* function used earlier in this section, by:

$$
\begin{array}{|l}
get\_type\_first : (\mathrm{seq}\, Id) \times Env \times IdDot) \rightarrowtail Val \\
\hline
\forall\, c, pc : \mathrm{seq}\, Id;\ \delta : Env;\ tm : IdDot;\ t : Id \bullet \\
\quad (tm = id\ t \land \\
\quad\ get\_id\_ctx(c, \delta, t) = (pc, typeI) \land \\
\quad\ (\delta.dict\ pc).type\ t \in \mathrm{ran}\ intT) \Rightarrow \\
\qquad get\_type\_first(c, \delta, tm) = \\
\qquad\quad intval\ min\ (intT^{\sim}((\delta.dict\ pc).type\ t)) \\
\forall\, c, pc : \mathrm{seq}\, Id;\ \delta : Env;\ tm : IdDot;\ t : Id \bullet \\
\quad (tm = id\ t \land \\
\quad\ get\_id\_ctx(c, \delta, t) = (pc, typeI) \land \\
\quad\ (\delta.dict\ pc).type\ t \in \mathrm{ran}\ enumT) \Rightarrow \\
\qquad get\_type\_first(c, \delta, tm) = \\
\qquad\quad enumval\ ((enumT^{\sim}((\delta.dict\ pc).type\ t))0) \\
\forall\, c, pc, pc' : \mathrm{seq}\, Id;\ \delta : Env;\ tm : IdDot;\ k, t : Id \bullet \\
\quad (tm = dot(k, t) \land \\
\quad\ get\_id\_ctx(c, \delta, k) = (pc, pkgI) \land \\
\quad\ get\_id\_ctx(pc, \delta, t) = (pc', typeI) \land \\
\quad\ (\delta.dict\ pc').type\ t \in \mathrm{ran}\ intT) \Rightarrow \\
\qquad get\_type\_first(c, \delta, tm) = \\
\qquad\quad intval\ min\ (intT^{\sim}((\delta.dict\ pc').type\ t)) \\
\forall\, c, pc, pc' : \mathrm{seq}\, Id;\ \delta : Env;\ tm : IdDot;\ k, t : Id \bullet \\
\quad (tm = dot(k, t) \land \\
\quad\ get\_id\_ctx(c, \delta, k) = (pc, pkgI) \land \\
\quad\ get\_id\_ctx(pc, \delta, t) = (pc', typeI) \land \\
\quad\ (\delta.dict\ pc').type\ t \in \mathrm{ran}\ enumT) \Rightarrow \\
\qquad get\_type\_first(c, \delta, tm) = \\
\qquad\quad enumval\ ((enumT^{\sim}((\delta.dict\ pc').type\ t))0)
\end{array}
$$

References: Env p. 11; Store p. 12; IdDot p. 8; Val p. 9; typeI p. 37; pkgI p. 37; varI p. 37; get_id_ctx p. 38; *arrT p. 15; intT p. 15; enumT p. 15.*

7.5 Last of an Array Index Type

The `'LAST` attribute can be applied to a type mark of an array type (or a formal parameter of an unconstrained array type) to obtain the greatest element of one of the index (sub)types of the array. An argument specifies to which index type the attribute is applied.

Syntax Example	A.S. Representation
sensors'LAST(2)	$lsta \, \langle\!| \; arrtyp \mapsto sensors, \; indexno \mapsto lint \; 2 \; |\!\rangle$

7.5.1 Abstract Syntax

$$
\begin{array}{|l}
\hline
\textit{LstAExp} \\
arrtyp : IdDot \\
indexno : Exp \\
\hline
\end{array}
$$

$$Exp ::= \ldots \mid lsta\langle\!\langle LstAExp \rangle\!\rangle$$

7.5.2 Dynamic Semantics

If the prefix is the array *type mark*, we look up the appropriate index-type in the dictionary; if it is an array *object*, we look at its index range as stored in the array value in the store. The index number must be an integer in the range $1 \mathrel{..} n$, where n is the dimensionality of the array. The first two rules deal with a type mark (simple name, then a selected name).

$$
\begin{array}{l}
\forall\, c, pc : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ n : \mathbb{N}_1; \\
\quad t : IdDot;\ LstAExp \\
\mid \\
\qquad arrtyp \in \operatorname{ran} id \\
\qquad get\_id\_ctx(c, \delta, id^{\sim}\, arrtyp) = (pc, typeI) \\
\qquad (\delta.dict\ pc).type\ (id^{\sim}\, arrtyp) \in \operatorname{ran} arrT \\
\qquad n \leq \#\ (arrT^{\sim}((\delta.dict\ pc).type\ (id^{\sim}\, arrtyp))).indexes \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e indexno \Longrightarrow_e lint n \\
\hline
\qquad c, \delta, \sigma \vdash_e lsta(\theta LstAExp) \Longrightarrow_e \\
\qquad\qquad get\_type\_last(c, \delta, t)
\end{array}
\qquad \text{(LstAD1)}
$$

where

$$t == (arrT^{\sim}((\delta.dict\ pc).type\ (id^{\sim}\, arrtyp))).indexes\ n$$

$$
\begin{array}{l}
\forall\, c, pc : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ k, t, t' : Id;\ n : \mathbb{N}_1;\ LstAExp \\
\mid \\
\quad arrtyp = dot(k, t) \\
\quad get\_id\_ctx(c, \delta, k) = (pc, pkgI) \\
\quad t \in \mathrm{dom}(\delta.dict\ pc).type \\
\quad (\delta.dict\ pc).type\ t \in \mathrm{ran}\, arrT \\
\quad n \leq \#\ (arrT^{\sim}((\delta.dict\ pc).type\ t)).indexes \\
\bullet \\
\quad c, \delta, \sigma \vdash_e indexno \Longrightarrow_e lintn \\
\hline
\quad c, \delta, \sigma \vdash_e lsta(\theta LstAExp) \Longrightarrow_e \\
\qquad get\_type\_last(pc, \delta, t')
\end{array}
\qquad \text{(LstAD2)}
$$

where

$$t' == (arrT^{\sim}((\delta.dict\ pc).type\ t)).indexes\ n$$

In the above two rules, the auxiliary function *get_type_last* is used to fetch the starting value of the index range type (its third argument); this function is defined at the end of this section.

The next two rules deal with an array object as prefix to the attribute.

$$
\begin{array}{l}
\forall\, c, pc : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ n : \mathbb{N}_1; \\
\ \ t : IdDot;\ LstAExp \\
\mid \\
\quad arrtyp \in \mathrm{ran}\, id \\
\quad get\_id\_ctx(c, \delta, id^{\sim} arrtyp) = (pc, varI) \\
\quad av \in \mathrm{ran}\, arrval \\
\bullet \\
\quad c, \delta, \sigma \vdash_e indexno \Longrightarrow_e lintn \\
\quad c, \delta, \sigma \vdash_e arrtyp \Longrightarrow_e av \\
\hline
\quad c, \delta, \sigma \vdash_e lsta(\theta LstAExp) \Longrightarrow_e \\
\qquad get\_index\_last(av, n)
\end{array}
\qquad \text{(LstAD3)}
$$

$$
\begin{array}{l}
\forall\, c, pc, pc' : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ n : \mathbb{N}_1; \\
\quad k, t : IdDot;\ LstAExp \\
| \\
\qquad arrtyp = dot(k, t) \\
\qquad get\_id\_ctx(c, \delta, k) = (pc, pkgI) \\
\qquad get\_id\_ctx(pc, \delta, t) = (pc', varI) \\
\qquad av \in \operatorname{ran} arrval \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e indexno \Longrightarrow_e lintn \\
\qquad c, \delta, \sigma \vdash_e arrtyp \Longrightarrow_e av \\
\hline
\qquad c, \delta, \sigma \vdash_e lsta(\theta LstAExp) \Longrightarrow_e \\
\qquad\qquad get\_index\_last(av, n)
\end{array}
\tag{LstAD4}
$$

In the above two rules, the auxiliary function *get_index_last* is used. This may be defined by:

$$
\begin{array}{|l}
get\_index\_last : (Val \times \mathbb{N}_1) \rightarrow\!\!\!\!\!+ Val \\
\hline
\forall\, v : Val \bullet \\
\qquad (v \in \operatorname{ran} arrval) \Rightarrow \\
\qquad\qquad get\_index\_last(v, 1) = intval\ (arrval^{\sim} v).hi \\
\forall\, v : Val;\ n : \mathbb{N}_1 \bullet \\
\qquad (v \in \operatorname{ran} arrval) \Rightarrow \\
\qquad\qquad get\_index\_last(v, n + 1) = \\
\qquad\qquad\qquad get\_index\_last((arrval^{\sim} v).arr\ (arrval^{\sim} v).hi, n)
\end{array}
$$

Finally, we define the function *get_type_last* used earlier by:

$$
\begin{array}{l}
get\_type\_last : (\operatorname{seq} Id) \times Env \times IdDot) \nrightarrow Val \\
\hline
\forall\, c, pc : \operatorname{seq} Id;\ \delta : Env;\ tm : IdDot;\ t : Id \bullet \\
\quad (tm = id\ t \wedge \\
\quad get\_id\_ctx(c, \delta, t) = (pc, typeI) \wedge \\
\quad (\delta.dict\ pc).type\ t \in \operatorname{ran} intT) \Rightarrow \\
\qquad get\_type\_last(c, \delta, tm) = \\
\qquad\quad intval\ max\ (intT^{\sim}((\delta.dict\ pc).type\ t)) \\
\forall\, c, pc : \operatorname{seq} Id;\ \delta : Env;\ tm : IdDot;\ t : Id \bullet \\
\quad (tm = id\ t \wedge \\
\quad get\_id\_ctx(c, \delta, t) = (pc, typeI) \wedge \\
\quad (\delta.dict\ pc).type\ t \in \operatorname{ran} enumT) \Rightarrow \\
\qquad get\_type\_last(c, \delta, tm) = \\
\qquad\quad enumval\ ((enumT^{\sim}((\delta.dict\ pc).type\ t)) \\
\qquad\qquad max\ (\operatorname{dom}\ (enumT^{\sim}((\delta.dict\ pc).type\ t)))) \\
\forall\, c, pc, pc' : \operatorname{seq} Id;\ \delta : Env;\ tm : IdDot;\ k, t : Id \bullet \\
\quad (tm = dot(k, t) \wedge \\
\quad get\_id\_ctx(c, \delta, k) = (pc, pkgI) \wedge \\
\quad get\_id\_ctx(pc, \delta, t) = (pc', typeI) \wedge \\
\quad (\delta.dict\ pc').type\ t \in \operatorname{ran} intT) \Rightarrow \\
\qquad get\_type\_last(c, \delta, tm) = \\
\qquad\quad intval\ max\ (intT^{\sim}((\delta.dict\ pc').type\ t)) \\
\forall\, c, pc, pc' : \operatorname{seq} Id;\ \delta : Env;\ tm : IdDot;\ k, t : Id \bullet \\
\quad (tm = dot(k, t) \wedge \\
\quad get\_id\_ctx(c, \delta, k) = (pc, pkgI) \wedge \\
\quad get\_id\_ctx(pc, \delta, t) = (pc', typeI) \wedge \\
\quad (\delta.dict\ pc').type\ t \in \operatorname{ran} enumT) \Rightarrow \\
\qquad get\_type\_last(c, \delta, tm) = \\
\qquad\quad enumval\ ((enumT^{\sim}((\delta.dict\ pc').type\ t)) \\
\qquad\qquad max\ (\operatorname{dom}\ (enumT^{\sim}((\delta.dict\ pc').type\ t))))
\end{array}
$$

References: Env p. 11; Store p. 12; IdDot p. 8; Val p. 9; typeI p. 37; pkgI p. 37; varI p. 37; get_id_ctx p. 38; *arrT p. 15; intT p. 15; enumT p. 15.*

7.6 Successor

The successor to an element of a discrete type is returned by the SUCC attribute. The attribute is applied to a discrete type.

Syntax Example	A.S. Representation
traffic.colour'SUCC(green)	$succ \; \langle\!| \; distyp \mapsto dot(traffic, colour),$ $val \mapsto nam \; (simp \; green) \; |\!\rangle$

7.6.1 Abstract Syntax

In Ada, the form T'SUCC is regarded as a function requiring a single argument. Here, we include the argument in the Abstract Syntax.

$$SuccExp \mathrel{\widehat{=}} [distyp : IdDot; \; val : Exp]$$

$$Exp ::= \ldots \mid succ\langle\!\langle SuccExp \rangle\!\rangle$$

We have assumed that the BASE'SUCC attribute does not exist in SPARK (see **SLI WJ005***).*

7.6.2 Dynamic Semantics

The successor exists provided the value of the expression whose successor is sought is not the last element of the enumeration type. For subtypes, the original type is used; thus, given

```
type day is (mon, tue, wed, thu, fri, sat, sun);
subtype weekday is day range mon .. fri;
```

the expression `weekday'succ(e)` is well-defined, even if `e` evaluates to `fri` or `sat` (though not `sun`) [LRM, 3.5.5(17)].

There are two rules for successor: one for enumeration types, the other for integer types.

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ argval, lastval : Val;\\
\ \ ei : Id \pfun \mathbb{N};\ SuccExp\\
\mid\\
\qquad get\_typ\_con_\delta\ distyp \in \operatorname{ran} enumT\\
\qquad argval \neq lastval\\
\qquad ei = enumT^\sim\ get\_typ\_con_\delta\ distyp\\
\bullet\\
\qquad c, \delta, \sigma \vdash_e blst\ \ distyp \Longrightarrow_e firstval\\
\qquad c, \delta, \sigma \vdash_e val \Longrightarrow_e argval\\
\hline
\qquad c, \delta, \sigma \vdash_e succ(\theta SuccExp) \Longrightarrow_e\\
\qquad\qquad enumval\ (id\ (ei^\sim((ei\ (enumval^\sim argval)) + 1)))
\end{array}
\tag{SuccD1}
$$

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ argval, lastval : \mathbb{Z};\\
\ \ SuccExp\\
\mid\\
\qquad get\_typ\_con_\delta\ distyp \in \operatorname{ran} intT\\
\qquad argval \neq lastval\\
\bullet\\
\qquad c, \delta, \sigma \vdash_e blst\ \ distyp \Longrightarrow_e intval\ \ lastval\\
\qquad c, \delta, \sigma \vdash_e val \Longrightarrow_e intval\ \ argval\\
\hline
\qquad c, \delta, \sigma \vdash_e pred(\theta SuccExp) \Longrightarrow_e intval\ \ (argval + 1)
\end{array}
\tag{SuccD2}
$$

References: Env p. 11; Store p. 12; Val p. 9; enumT p. 15; $get\_typ\_con_\delta$ p. 216; *intT p. 15.*

7.7 Predecessor

The predecessor to an element of a discrete type is returned by the PRED attribute. The attribute is applied to a discrete type.

Syntax Example	A.S. Representation
traffic.colour'PRED(orange)	$pred \langle\!| \; distyp \mapsto dot(traffic, colour), \; val \mapsto nam \; (simp \; orange) \; |\!\rangle$

7.7.1 Abstract Syntax

In Ada, the form T'PRED is regarded as a function requiring a single argument. Here, we include the argument in the Abstract Syntax.

$$PredExp \mathrel{\widehat=} [distyp : IdDot; \; val : Exp]$$

$$Exp ::= \ldots \mid pred \langle\!\langle PredExp \rangle\!\rangle$$

7.7.2 Dynamic Semantics

The predecessor exists provided the value of the expression whose predecessor is sought is not the first element of the enumeration type. For subtypes, the original type is used; thus, given

```
type day is (mon, tue, wed, thu, fri, sat, sun);
subtype weekend is day range sat .. sun;
```

the expression `weekend'pred(e)` is well-defined, even if `e` evaluates to a value in the range `tue` to `fri` (though not `mon`) [LRM, 3.5.5(17)]. The predecessor function can be applied to any discrete type, either enumerated or integer; we provide one rule for each case.

$$\begin{array}{l}
\forall\, c : \operatorname{seq} Id; \; \delta : Env; \; \sigma : Store; \; argval, firstval : Val; \\
\quad ei : Id \rightarrowtail \mathbb{N}; \; PredExp \\
\mid \\
\qquad get_typ_con_\delta \; distyp \in \operatorname{ran} enumT \\
\qquad argval \neq firstval \\
\qquad ei = enumT^\sim \; get_typ_con_\delta \; distyp \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e bfst \;\; distyp \Longrightarrow_e firstval \\
\qquad c, \delta, \sigma \vdash_e val \Longrightarrow_e argval \\
\hline
\qquad c, \delta, \sigma \vdash_e pred(\theta PredExp) \Longrightarrow_e \\
\qquad\qquad enumval \;\; (id \;\; (ei^\sim((ei \;\; (enumval^\sim \, argval)) - 1)))
\end{array} \tag{PredD1}$$

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ argval, firstval : \int;\\
\quad PredExp\\
\mid\\
\qquad get\_typ\_con_{\delta}\ distyp \in \operatorname{ran} intT\\
\qquad argval \neq firstval\\
\bullet\\
\qquad c, \delta, \sigma \vdash_e bfst\ \ distyp \Longrightarrow_e intval\ firstval\\
\qquad c, \delta, \sigma \vdash_e val \Longrightarrow_e intval\ \ argval\\
\hline
\qquad c, \delta, \sigma \vdash_e pred(\theta PredExp) \Longrightarrow_e intval\ \ (argval - 1)
\end{array}
\tag{PredD2}
$$

References: Env p. 11; Store p. 12; Val p. 9; enumT p. 15; $get\_typ\_con_{\delta}$ p. 216; *intT p. 15.*

7.8 Position

The position of an element of a discrete type is returned by the POS attribute. The attribute is applied to a discrete type.

Syntax Example	A.S. Representation
traffic.colour'POS(red)	$pos \langle\!| \; distyp \mapsto dot(traffic, colour), val \mapsto nam\ (simp\ red) \; |\!\rangle$

7.8.1 Abstract Syntax

In Ada, the form T'POS is regarded as a function requiring a single argument. Here, we include the argument in the Abstract Syntax.

$$PosExp \mathrel{\widehat=} [distyp : IdDot;\ val : Exp]$$

$$Exp ::= \ldots \mid pos\langle\!\langle PosExp \rangle\!\rangle$$

We have assumed that the BASE'SUCC attribute does not exist in SPARK (see **SLI WJ005***).*

7.8.2 Dynamic Semantics

The position is an Ada *universal_integer*; for integer types, it is the integer itself, while for enumeration types, it is the position number as given in the corresponding enumeration type construction. We give two rules, to cover these two cases:

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ argval : Val;\ PosExp \\ \mid \\ \quad get\_typ\_con_\delta\ distyp \in \operatorname{ran} intT \\ \bullet \\ \quad c, \delta, \sigma \vdash_e val \Longrightarrow_e argval\end{array}}{c, \delta, \sigma \vdash_e pos(\theta PosExp) \Longrightarrow_e argval} \qquad \text{(PosD1)}$$

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ eid : Id;\ PosExp \\ \mid \\ \quad get\_typ\_con_\delta\ distyp \in \operatorname{ran} enumT \\ \bullet \\ \quad c, \delta, \sigma \vdash_e val \Longrightarrow_e enumval\ (id\ eid)\end{array}}{\begin{array}{l}c, \delta, \sigma \vdash_e pos(\theta PosExp) \Longrightarrow_e \\ \quad intval\ ((enumT^\sim\ get\_typ\_con_\delta\ distyp)\ eid)\end{array}} \qquad \text{(PosD2)}$$

References: Env p. 11; Store p. 12; Val p. 9; enumT p. 15; $get\_typ\_con_\delta$ p. 216; *intT p. 15.*

7.9 Value

The value of an element of a discrete type is returned by the VAL attribute. The attribute is applied to a discrete type.

Syntax Example	A.S. Representation
traffic.colour'VAL(1)	$valu \ \langle\!\mid \ distyp \mapsto dot(traffic, colour), \ val \mapsto lint\ 1 \ \mid\!\rangle$

7.9.1 Abstract Syntax

In Ada, the form T'VAL is regarded as a function requiring a single argument. Here, we include the argument in the Abstract Syntax.

$$ValuExp \mathrel{\widehat=} [distyp : IdDot;\ val : Exp]$$

$$Exp ::= \ldots \mid valu \langle\!\langle ValuExp \rangle\!\rangle$$

7.9.2 Dynamic Semantics

Given a *universal_integer* object as argument, this function returns an object of the base type of the given type, whose position number is the given argument. In Ada, if the argument is outside the range of the base type, a `CONSTRAINT_ERROR` is raised; we include a dynamic well-formedness check to preclude this in our inference rules below.

There are two cases to consider, according to whether the type is an integer type or an enumeration type:

$$\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ argval : \mathbb{Z};\ ValuExp \\
\mid \\
\quad get_typ_con_{\delta}\ distyp \in \mathrm{ran}\ intT \\
\quad argval \in (intT^{\sim}(get_typ_con_{\delta}\ distyp)).range \\
\bullet \\
\quad \dfrac{c, \delta, \sigma \vdash_e val \Longrightarrow_e intval\ \ argval}{c, \delta, \sigma \vdash_e valu(\theta\, ValuExp) \Longrightarrow_e intval\ \ argval}
\end{array} \qquad \text{(ValD1)}$$

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \ \delta : Env;\ \ \sigma : Store;\ \ argval : \mathbb{Z};\ \ ValuExp \\
| \\
\qquad get\_typ\_con_{\delta}\ distyp \in \operatorname{ran} enumT \\
\qquad intval \in \operatorname{ran}(enumT^{\sim}(get\_typ\_con_{\delta}\ distyp)) \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e\ val \Longrightarrow_e\ intval\ \ argval \\
\hline
\qquad c, \delta, \sigma \vdash_e\ valu(\theta\, ValuExp) \Longrightarrow_e \\
\qquad\qquad enumval\ \ (id\ \ ((enumT^{\sim}(get\_typ\_con_{\delta}\ distyp))^{\sim} intval))
\end{array}
\qquad \text{(ValD2)}
$$

References: Env p. 11; *Store p. 12*; *enumT p. 15*; $get\_typ\_con_{\delta}$ p. 216; *intT p. 15.*

7.10 Size of Object

The SIZE attribute can be applied to a type mark or object to obtain the number of bits used to store the object (or an object of the type).

Syntax Example	A.S. Representation
t'SIZE	*size t*

7.10.1 Abstract Syntax

$Exp ::= \ldots \mid size\langle\langle IdDot \rangle\rangle$

7.10.2 Dynamic Semantics

The value returned is implementation dependent.

Chapter 8

Declarations — Overview

The syntax of SPARK includes a number of different declarations, for example variable and subprogram declarations, and a number of different scopes in which declarations can appear, for example package specifications and subprogram bodies. Syntactic rules restrict the forms of declaration which may appear in the different scopes.

This Chapter has two purposes: firstly to give an overview of the Abstract Syntax of declarations and the way it is structured, and secondly to give a number of "glueing" definitions, which join together the definitions given in later chapters.

After some background, this chapter introduces the different categories of declarations which are described in detail in subsequent chapters. The following section describes the different groupings of declarations, which we call *declarative scopes*. Subsequent sections give the semantic rules for each of these scopes.

Background Ada distinguishes between *basic* and *later* declarations; in scopes which contain both, the former precede the latter. Basic declarations are typified by declarations which do not introduce a nested scope, while later declarations include all those which contain nested scopes. However, in Ada, these categories overlap, with subprogram declarations and package declarations (i.e. specifications) appearing in both categories.

The same idea is carried over into SPARK, but with modifications. Subprogram and package declarations are not considered as basic or later declarations, rather the syntax may allow them to be combined with basic declarations, depending on the context.

Summary of Differences between SPARK and Ada The following points summarise the differences between SPARK and Ada concerning the declarations allowed in any context.

1. In SPARK, package specification may not be nested within package specification.
2. In SPARK, subprogram declarations (that is, a declaration of the subprogram name and parameters without the subprogram body) are only allowed in (the visible part of) package specifications.

3. In SPARK, a subprogram definition (that is, a declaration giving a body to a subprogram which has already been declared) can only appear in a package body.

4. The *renaming* declarations of Ada are treated separately from other declarations in SPARK. They are restricted to appear in particular places only — see Chapter 16.

8.1 Syntactic Categories of Declarations

This section introduces the different syntactic categories of SPARK declarations.

Basic Declarations – *BDecl* Constant, variable, type and subtype declarations are the basic declarations.

Private Declarations – *PDecl* Deferred constants, private types and limited private types are termed private declarations.

Subprogram Declarations – *SDecl* Subprogram declarations give the name and parameters of a function or procedure subprogram and its annotations.

Package Declarations – *KDecl* This is a package specification.

Subprogram Definitions – *FDecl* A subprogram definition supplies the body to a subprogram which has already been declared. The annotations are not repeated. A stub may be used if the subprogram body is separate.

Subprogram and Package Bodies – *YDecl* This categories includes bodies for subprogram which have not been declared (in a package specification) and package bodies.

The well-formation rules for the declarations in each category are described in separate chapters, as given in the following table:

Syntax Category	Description	Chapter	Page
BDecl	Basic Declarations	9	113
PDecl	Private Declarations	10	119
SDecl	Subprogram Declarations	12	159
KDecl	Package Declarations	13	165
FDecl	Subprogram Definitions	14	169
YDecl	Subprogram and Package Bodies	15	177

8.2 Declarative Scope

This section introduces the different declarative *scopes* or regions in SPARK, each of which contains declarations from a different selection of the syntactic categories. The dynamic semantics rules for the declarative scopes, which have a very regular structure, are given in the remaining sections of this chapter.

The names of the scopes suggest where in the syntax they occur. We also describe the categories of declarations which each sort of scope may contain.

Visible Basic Declarations – *VBasic* This is the scope formed by the visible part of a package specification (either a compilation unit or an embedded package). It contains the basic declarations of objects and types, the private declarations and the declaration of the subprogram which form the interface to the package. See Section 8.3.

Private Basic Declarations – *PBasic* This is the scope formed by the private part of the package specification. It contains only the basic declarations; however, the declarations of constants and types have an additional use: the completion of the deferred or private declarations given in the visible part. See Section 8.4.

Subprogram Body Basic Declarations – *SBasic* This scope is the first part of the subprogram body. It contains basic declarations and the specifications of (embedded) packages. See Section 8.5.

Package Body Basic Declarations – *KBasic* This scope is the first part of the package body. It contains basic declarations and the specifications of (embedded) packages. See Section 8.6.

The categories *SBasic* and *KBasic* are distinguished because the declaration of own variables is only allowed in the latter.

Subprogram Later Declarations – *SLater* This scope is the second part of a subprogram body. It contains subprogram bodies (for subprograms without declarations) and package bodies. See Section 8.7.

Package Later Declarations – *KLater* This scope is the second part of a package body. It contains subprogram bodies (for subprograms both with and without a declaration in the corresponding package specification) and bodies of embedded packages. See Section 8.8.

8.3 Visible Basic Declarations

The syntax category *VBasic* contains the declarations which appear in the visible part of a package specification.

Dynamic Elaboration of Visible Basic Declarations

The elaboration rules for visible basic declarations are specified by a relation between them and the context, the environment, the store and a modified environment and store. The declaration of this relation is:

$$\_,\_,\_ \vdash_{vbasic} \_ \Longrightarrow_{vbasic} \_,\_ \subseteq (\mathrm{seq}\, Id) \times Env \times Store \times VBasic \times Env \times Store$$

8.3.1 Abstract Syntax

The visible part of a package specification may contain basic declarations *BDecl*, private declarations *PDecl* and declarations of the exported subprograms *SDecl*.

$$\begin{array}{rl} VBasic ::= & vbasic\langle\!\langle BDecl\rangle\!\rangle \\ | & vpriv\langle\!\langle PDecl\rangle\!\rangle \\ | & vsub\langle\!\langle SDecl\rangle\!\rangle \\ | & vseq\langle\!\langle VBasic \times VBasic\rangle\!\rangle \\ | & vnull \end{array}$$

8.3.2 Dynamic Semantics

Each of the allowed forms of declaration must be elaborated according to the corresponding rule below.

$$\forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ b : BDecl$$
$$\bullet$$
$$\frac{c, \delta, \sigma \vdash_{bdecl} b \Longrightarrow_{bdecl} \delta', \sigma'}{c, \delta, \sigma \vdash_{vbasic} vbasic\ b \Longrightarrow_{vbasic} \delta', \sigma'} \qquad \text{(VBasicD)}$$

Note: the private declarations, allowed in the private part of a package specification, have no dynamic semantics effect: they do not affect the store or the dynamic environment.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ p : PDecl \quad \bullet}{c, \delta, \sigma \vdash_{vbasic}\ vpriv\ p \Longrightarrow_{vbasic} \delta, \sigma} \tag{VPrivD}$$

Note: the subprogram declarations of SPARK, allowed in the visible part of a package specification, have no dynamic semantics effect: they do not affect the store or the dynamic environment. (We are only interested in the subprogram *definitions*, given in the package bodies.)

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ s : SDecl \quad \bullet}{c, \delta, \sigma \vdash_{vbasic}\ vbasic\ s \Longrightarrow_{vbasic} \delta, \sigma} \tag{VSubD}$$

A sequence of declarations is elaborated by elaborating each declaration in turn.

$$\frac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta, \delta', \delta'' : Env;\ \sigma, \sigma', \sigma'' : Store;\ u, v : VBasic \\ \bullet \\ \quad c, \delta, \sigma \vdash_{vbasic}\ u \Longrightarrow_{vbasic} \delta', \sigma' \\ \quad c, \delta', \sigma' \vdash_{vbasic}\ v \Longrightarrow_{vbasic} \delta'', \sigma''\end{array}}{c, \delta, \sigma \vdash_{vbasic}\ vseq\ (u, v) \Longrightarrow_{vbasic} \delta'', \sigma''} \tag{VSeqD}$$

Elaboration of the null declaration leaves the store and environment unchanged.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store \quad \bullet}{c, \delta, \sigma \vdash_{vbasic}\ vnull \Longrightarrow_{vbasic} \delta, \sigma} \tag{VNullD}$$

References: Env p. 11; *Store p. 12*; *BDecl p. 113*; $\vdash_{bdecl}$ p. 113; $\Longrightarrow_{bdecl}$ p. 113; *PDecl p. 119*; *SDecl p. 159.*

8.4 Private Basic Declarations

The syntax category *VBasic* contains the declarations which appear in the private part of a package specification.

Dynamic Elaboration of Private Basic Declarations

The elaboration rules for private basic declarations are specified by a relation between them and the context, the environment, the store and a modified environment store. The declaration of this relation is:

$$\_, \_, \_ \vdash_{pbasic} \_ \Longrightarrow_{pbasic} \_, \_ \subseteq (\mathrm{seq}\, Id) \times Env \times Store \times PBasic \times Env \times Store$$

8.4.1 Abstract Syntax

Only basic declarations *BDecl* are allowed in the private part of a package specification.

$$\begin{aligned} PBasic ::= &\ pbasic\langle\langle BDecl\rangle\rangle \\ |\ &\ pseq\langle\langle PBasic \times PBasic\rangle\rangle \\ |\ &\ pnull \end{aligned}$$

8.4.2 Dynamic Semantics

For the dynamic semantics, we are unconcerned whether or not the declaration in the private part has been preceded by a deferred or private declaration in the visible part: the policing of violations of such constraints is a static semantics issue. We therefore have only one rule for basic declarations in the private part:

$$\forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ b : BDecl$$
$$\bullet$$
$$\frac{c, \delta, \sigma \vdash_{bdecl} b \Longrightarrow_{bdecl} \delta', \sigma'}{c, \delta, \sigma \vdash_{vbasic} pbasic\ b \Longrightarrow_{vbasic} \delta', \sigma'} \quad \text{(PBasicD)}$$

A sequence of declarations is elaborated by elaborating each declaration in turn.

$$\forall\, c : \mathrm{seq}\, Id;\ \delta, \delta', \delta'' : Env;\ \sigma, \sigma', \sigma'' : Store;\ u, v : PBasic$$
$$\bullet$$
$$\frac{\begin{array}{l} c, \delta, \sigma \vdash_{pbasic} u \Longrightarrow_{pbasic} \delta', \sigma' \\ c, \delta', \sigma' \vdash_{pbasic} v \Longrightarrow_{pbasic} \delta'', \sigma'' \end{array}}{\delta, \sigma \vdash_{pbasic} pseq\ (u, v) \Longrightarrow_{pbasic} \delta'', \sigma''} \quad \text{(PSeqD)}$$

Elaboration of the null declaration leaves the store and environment unchanged.

$$\frac{\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store \\ \bullet}{c, \delta, \sigma \vdash_{pbasic} pnull \Longrightarrow_{pbasic} \delta, \sigma} \qquad \text{(PNullD)}$$

References: Env p. 11; Store p. 12.

8.5 Subprogram Body Basic Declarations

The syntax category *SBasic* contains the basic declarations which appear in the body of a subprogram.

Dynamic Elaboration of Subprogram Body Basic Declarations

The elaboration rules for basic declarations of a subprogram body are specified by a relation between them and the context, the environment, the store and a modified environment and store. The declaration of this relation is:

$$\_,\_,\_ \vdash_{sbasic} \_ \Longrightarrow_{sbasic} \_,\_ \subseteq (\mathrm{seq}\, Id) \times Env \times Store \times SBasic \times Env \times Store$$

8.5.1 Abstract Syntax

Basic declarations *BDecl* and the declarations of embedded packages *KDecl* can appear in a subprogram body.

$$\begin{array}{rcl} SBasic & ::= & sbasic\langle\!\langle BDecl \rangle\!\rangle \\ & | & spak\langle\!\langle KDecl \rangle\!\rangle \\ & | & sseq\langle\!\langle SBasic \times SBasic \rangle\!\rangle \\ & | & snull \end{array}$$

8.5.2 Dynamic Semantics

The elaboration of the different forms of declaration is determined by the appropriate rule.

$$\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ b : BDecl \\ \bullet \\ \dfrac{c, \delta, \sigma \vdash_{bdecl} b \Longrightarrow_{bdecl} \delta', \sigma'}{c, \delta, \sigma \vdash_{sbasic} sbasic\ b \Longrightarrow_{sbasic} \delta', \sigma'} \end{array} \qquad \text{(SBasicD)}$$

$$\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ k : KDecl \\ \bullet \\ \dfrac{c, \delta, \sigma \vdash_{kdecl} k \Longrightarrow_{kdecl} \delta', \sigma'}{c, \delta, \sigma \vdash_{sbasic} spak\ k \Longrightarrow_{sbasic} \delta', \sigma'} \end{array} \qquad \text{(SPakD)}$$

A sequence of declarations is elaborated by elaborating each declaration in turn.

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta, \delta', \delta'' : Env;\ \sigma, \sigma', \sigma'' : Store;\ u, v : SBasic \\ \bullet \\ \qquad c, \delta, \sigma \vdash_{sbasic} u \Longrightarrow_{sbasic} \delta', \sigma' \\ \qquad c, \delta', \sigma' \vdash_{sbasic} v \Longrightarrow_{sbasic} \delta'', \sigma''\end{array}}{c, \delta, \sigma \vdash_{sbasic} sseq\ (u, v) \Longrightarrow_{sbasic} \delta'', \sigma''} \qquad \text{(SSeqD)}$$

Elaboration of the null declaration leaves the store and environment unchanged.

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store \\ \bullet\end{array}}{c, \delta, \sigma \vdash_{sbasic} snull \Longrightarrow_{sbasic} \delta, \sigma} \qquad \text{(SNullD)}$$

References: Env p. 11; *Store p. 12*; *BDecl p. 113*; $\vdash_{bdecl}$ p. 113; $\Longrightarrow_{bdecl}$ p. 113; *KDecl p. 165*; $\vdash_{kdecl}$ p. 165; $\Longrightarrow_{kdecl}$ p. 165.

8.6 Package Body Basic Declarations

The syntax category *KBasic* contains the basic declarations which appear in a package body.

Dynamic Elaboration of Package Body Basic Declarations

The elaboration rules for basic declarations of a package body are specified by a relation between them and the environment, the store and a modified store. The declaration of this relation is:

$$\_,\_,\_ \vdash_{kbasic} \_ \Longrightarrow_{kbasic} \_,\_ \subseteq (\mathrm{seq}\, Id) \times Env \times Store \times KBasic \times Env \times Store$$

8.6.1 Abstract Syntax

Basic declarations *BDecl* and embedded package declarations *KDecl* may appear in a package body.

$$\begin{array}{rcl} KBasic & ::= & kbasic \langle\!\langle BDecl \rangle\!\rangle \\ & | & kpak \langle\!\langle KDecl \rangle\!\rangle \\ & | & kseq \langle\!\langle KBasic \times KBasic \rangle\!\rangle \\ & | & knull \end{array}$$

8.6.2 Dynamic Semantics

We are not concerned with whether or not a variable declared in a package body has been declared to be an own variable; this is an issue for the static semantics only. Consequently, we have only one rule for elements of *BDecl*.

$$\frac{\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ b : BDecl \\ \bullet \\ \quad c, \delta, \sigma \vdash_{bdecl} b \Longrightarrow_{bdecl} \delta', \sigma' \end{array}}{c, \delta, \sigma \vdash_{kbasic} kbasic\ b \Longrightarrow_{kbasic} \delta', \sigma'} \quad \text{(KBasicD)}$$

The elaboration of an embedded package declaration is determined by the appropriate rule.

$$\frac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ k : KDecl \\ \bullet \\ \quad c, \delta, \sigma \vdash_{kdecl} k \Longrightarrow_{kdecl} \delta', \sigma'\end{array}}{c, \delta, \sigma \vdash_{kbasic} kpak\ k \Longrightarrow_{kdecl} \delta', \sigma'} \quad \text{(KPakD)}$$

A sequence of declarations is elaborated by elaborating each declaration in turn.

$$\frac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta, \delta', \delta'' : Env;\ \sigma, \sigma', \sigma'' : Store;\ u, v : KBasic \\ \bullet \\ \quad c, \delta, \sigma \vdash_{kbasic} u \Longrightarrow_{kbasic} \delta', \sigma' \\ \quad c, \delta', \sigma' \vdash_{kbasic} v \Longrightarrow_{kbasic} \delta'', \sigma''\end{array}}{c, \delta, \sigma \vdash_{kbasic} kseq\ (u, v) \Longrightarrow_{kbasic} \delta'', \sigma''} \quad \text{(KSeqD)}$$

Elaboration of the null declaration leaves the store and environment unchanged.

$$\frac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store \\ \bullet\end{array}}{c, \delta, \sigma \vdash_{kbasic} knull \Longrightarrow_{kbasic} \delta, \sigma} \quad \text{(KNullD)}$$

References: Env p. 11; *Store p. 12*; *BDecl p. 113*; $\vdash_{bdecl}$ p. 113; $\Longrightarrow_{bdecl}$ p. 113; *KDecl p. 165*; $\vdash_{kdecl}$ p. 165; $\Longrightarrow_{kdecl}$ p. 165.

8.7 Subprogram Later Declarations

The syntax category *SLater* contains the later declarations which appear in the body of a subprogram.

Dynamic Elaboration of Subprogram Later Declarations

The elaboration rules for later declarations of a subprogram body are specified by a relation between them and the context, the environment, the store and a modified environment and store. The declaration of this relation is:

$$\_, \_, \_ \vdash_{slater} \_ \Longrightarrow_{slater} \_, \_ \subseteq (\mathrm{seq}\, Id) \times Env \times Store \times SLater \times Env \times Store$$

8.7.1 Abstract Syntax

The later declarations allowed in a subprogram body are the "body" declarations from *YDecl*.

$$\begin{aligned} SLater ::= {} & sbody \langle\!\langle YDecl \rangle\!\rangle \\ \mid {} & sseq \langle\!\langle SLater \times SLater \rangle\!\rangle \\ \mid {} & snull \end{aligned}$$

8.7.2 Dynamic Semantics

Each of the allowed forms of declaration must be elaborated according to the corresponding rule.

$$\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ y : YDecl \\ \bullet \\ \dfrac{c, \delta, \sigma \vdash_{ydecl} y \Longrightarrow_{ydecl} \delta', \sigma'}{c, \delta, \sigma \vdash_{slater} sbody\ y \Longrightarrow_{slater} \delta', \sigma'} \end{array} \qquad \text{(SLBodyD)}$$

A sequence of declarations is elaborated by elaborating each declaration in turn.

$$\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta, \delta', \delta'' : Env;\ \sigma, \sigma', \sigma'' : Env;\ u, v : SLater \\ \bullet \\ \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_{slater} u \Longrightarrow_{slater} \delta', \sigma' \\ c, \delta', \sigma' \vdash_{slater} v \Longrightarrow_{slater} \delta'', \sigma'' \end{array}}{c, \delta, \sigma \vdash_{slater} sseq\ (u, v) \Longrightarrow_{slater} \delta'', \sigma''} \end{array} \qquad \text{(SLSeqD)}$$

Elaboration of the null declaration leaves the store and environment unchanged.

$$\frac{\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store \\ \bullet}{c, \delta, \sigma \vdash_{slater} snull \Longrightarrow_{slater} \delta, \sigma} \qquad \text{(SLNullD)}$$

References: Env p. 11; *Store p. 12*; *YDecl p. 177*; $\vdash_{ydecl}$ p. 177; $\Longrightarrow_{ydecl}$ p. 177.

8.8 Package Later Declarations

The syntax category *KLater* contains the later declarations which appear in the body of a package.

Dynamic Elaboration of Package Later Declarations

The elaboration rules for later declarations of a package body are specified by a relation between them and the context, the environment, the store and a modified environment and store. The declaration of this relation is:

$$\_, \_, \_ \vdash_{klater} \_ \Longrightarrow_{klater} \_, \_ \subseteq (\operatorname{seq} Id) \times Env \times Store \times KLater \times Env \times Store$$

8.8.1 Abstract Syntax

The later declarations allowed in a package body are the subprogram definitions *FDecl* and the body declarations *YDecl*.

$$\begin{aligned} KLater ::= \; & kdefn\langle\!\langle FDecl\rangle\!\rangle \\ \mid \; & kbody\langle\!\langle YDecl\rangle\!\rangle \\ \mid \; & kseq\langle\!\langle KLater \times KLater\rangle\!\rangle \\ \mid \; & knull \end{aligned}$$

8.8.2 Dynamic Semantics

Each of the allowed forms of declaration must be elaborated according to the corresponding rule.

$$\forall\, c : \operatorname{seq} Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ f : FDecl$$
$$\bullet$$
$$\frac{c, \delta, \sigma \vdash_{fdecl} f \Longrightarrow_{fdecl} \delta', \sigma'}{c, \delta, \sigma \vdash_{klater} kdefn\ f \Longrightarrow_{klater} \delta', \sigma'} \qquad \text{(KLDefnD)}$$

$$\forall\, c : \operatorname{seq} Id;\ \delta, \delta' : Env;\ \sigma, \sigma' : Store;\ y : YDecl$$
$$\bullet$$
$$\frac{c, \delta, \sigma \vdash_{ydecl} y \Longrightarrow_{ydecl} \delta', \sigma'}{c, \delta, \sigma \vdash_{klater} kbody\ y \Longrightarrow_{klater} \delta', \sigma'} \qquad \text{(KLBodyD)}$$

A sequence of declarations is elaborated by elaborating each declaration in turn.

$$\forall\, c : \operatorname{seq} Id;\ \delta, \delta', \delta'' : Env;\ \sigma, \sigma', \sigma'' : Store;\ u, v : KLater \\ \bullet \\ \frac{\begin{array}{l} c, \delta, \sigma \vdash_{klater} u \Longrightarrow_{klater} \delta', \sigma' \\ c, \delta', \sigma' \vdash_{klater} v \Longrightarrow_{klater} \delta'', \sigma'' \end{array}}{c, \delta, \sigma \vdash_{klater} kseq\ (u, v) \Longrightarrow_{klater} \delta'', \sigma''} \qquad \text{(KLSeqD)}$$

Elaboration of the null declaration leaves the store unchanged.

$$\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store \\ \bullet \quad c, \delta, \sigma \vdash_{klater} knull \Longrightarrow_{klater} \delta, \sigma \qquad \text{(KLNullD)}$$

References: Env p. 11; *Store p. 12*; *FDecl p. 169*; $\vdash_{fdecl}$ p. 169; $\Longrightarrow_{fdecl}$ p. 169; *YDecl p. 177*; $\vdash_{ydecl}$ p. 177; $\Longrightarrow_{ydecl}$ p. 177.

Chapter 9

Basic Declarations

The Abstract Syntax of declaration (*BDecl*) is summarised in the following table:

Syntax Constructor	Description	Page
const	constant	114
var	variable	115
ftype	full type	116
stype	subtype	117

Dynamic Elaboration of Basic Declarations

The elaboration of a basic declaration is defined by a relation between the context, the environment, the store, the declaration and a modified environment and store. The declaration of this relation is:

$$\_,\_,\_ \vdash_{bdecl} \_ \Longrightarrow_{bdecl} \_,\_ \subseteq (\mathrm{seq}\, Id) \times Env \times Store \times BDecl \times Env \times Store$$

Thus the predicate:

$$c, \delta, \sigma \vdash_{bdecl} b \Longrightarrow_{bdecl} \delta', \sigma'$$

can be read as "declaration b, when elaborated with respect to context c, initial environment δ and initial store σ, yields the modified environment δ' and modified store σ'".

9.1 Constants

A constant has a named type and is given a value in its declaration.

Syntax Example	A.S. Representation
C : **constant** T := 0;	$const \langle\!|\ cid \mapsto C,\ type \mapsto id\ T,\ exp \mapsto lint\ 0\ |\!\rangle$

9.1.1 Abstract Syntax

A type mark, from *IdDot*, is used for the type of the constant; the value is specified by an expression.

$$ConstBDecl \mathrel{\widehat=} [cid : Id;\ type : IdDot;\ exp : Exp]$$

The Concrete Syntax allows a list of constant identifiers to appear on the left of the colon. In the Abstract Syntax, this is taken as an abbreviation for separate declarations with a copy of the term on the right of the colon, so that each constant has a separate declaration.

$$BDecl ::= \ldots \mid const\langle\!\langle ConstBDecl \rangle\!\rangle$$

9.1.2 Dynamic Semantics

The Static Semantics checks that the value assigned to the constant is validly within the type specified, so we do not concern ourselves with a repeated check that this is the case here. The effect of elaboration of the constant declaration is to associate the value of the expression with the collection of constants for the current context in the dynamic environment. (We use dynamic expression evaluation here, though a variant of the Static Semantics' static expression evaluation could – should? – be employed.)

$$\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ val : Val;\ ConstBDecl \bullet \frac{c, \delta, \sigma \vdash_e exp \Longrightarrow_e val}{c, \delta, \sigma \vdash_{bdecl} const(\theta ConstBDecl) \Longrightarrow_{bdecl} \delta', \sigma} \qquad \text{(ConstD)}$$

where

$$\delta' == \delta[\ dict := \delta.dict \oplus \{c \mapsto dict'\}\ ]$$

$$dict' == (\delta.dict\ c)[\ const := (\delta.dict\ c).const \oplus \{cid \mapsto val\}\ ]$$

References: Env p. 11; Store p. 12; IdDot p. 8; Exp p. 47; Val p. 9; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47.

9.2 Variables

All variables are declared using a named type.

Syntax Example	A.S. Representation
V : T	$var \langle\!| \; vid \mapsto V,$ $type \mapsto id\ T \; |\!\rangle$

9.2.1 Abstract Syntax

The variable name is an identifier; its type is given by a type mark, from *IdDot*.

$$VarBDecl \mathrel{\widehat=} [vid : Id;\ type : IdDot]$$

The Concrete Syntax allows a list of variable identifiers to appear on the left of the colon. In the Abstract Syntax, this is taken as an abbreviation for separate declarations with a copy of the term on the right of the colon, so that each variable has a separate declaration.

$$BDecl ::= \ldots \mid var\langle\!\langle VarBDecl \rangle\!\rangle$$

9.2.2 Dynamic Semantics

The dynamic semantics effect of a variable declaration is to modify the local environment and store, creating an entry of the given name with an initial value appropriate for the type of the variable, thus:

$$\frac{\begin{array}{l} \forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ VarBDecl \\ \mid \\ \quad \sigma' = \sigma \oplus \{c \frown \langle vid \rangle \mapsto form\_init\_val_\delta\ (c, type)\} \\ \bullet \end{array}}{c, \delta, \sigma \vdash_{bdecl} var(\theta\, VarBDecl) \Longrightarrow_{bdecl} \delta', \sigma'} \qquad \text{(VarD)}$$

where

$$\delta' == \delta[\ dict := \delta.dict \oplus \{c \mapsto dict'\}\ ]$$

$$dict' == (\delta.dict\ \ c)[\ var := (\delta.dict\ \ c).var \oplus \{vid \mapsto type\}\ ]$$

References: Env p. 11; Store p. 12; IdDot p. 8; form_init_val p. 27.

9.3 Full Types

A full type declaration gives a name to a type definition.

Syntax Example	A.S. Representation
type T **is range** 1 .. 9;	$ftype \langle\!\mid tid \mapsto T, def \mapsto int \langle\!\mid lower \mapsto lint\ 1, upper \mapsto lint\ 9 \mid\!\rangle \mid\!\rangle$

9.3.1 Abstract Syntax

The name of the type is an identifier; its definition is an element of *TypDef*.

$$FTypeBDecl \mathrel{\widehat=} [tid : Id;\ def : TypDef]$$

$$BDecl ::= \ldots \mid ftype\langle\!\langle FTypeBDecl\rangle\!\rangle$$

9.3.2 Dynamic Semantics

A full type declaration has no direct effect on the dynamic semantics; instead, it has its effect indirectly, through its impact on the dynamic environment.

$$\frac{\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma : Store;\ def : TypDef; \\ \ tcon : TypCon;\ d : Dict;\ FTypeBDecl \\ \mid \\ \qquad \delta'.pdecs = \delta.pdecs \\ \qquad \delta'.dict = \delta.dict \oplus \{c \mapsto d\} \bullet \\ \qquad c, \delta, \sigma \vdash_{typ} def \Longrightarrow_{typ} tcon \end{array}}{c, \delta, \sigma \vdash_{bdecl} ftype(\theta FTypeBDecl) \Longrightarrow_{bdecl} \delta', \sigma} \qquad \text{(FTypeD)}$$

where

$$d == (\delta.dictc)[type := (\delta.dictc).type \oplus \{tid \mapsto tcon\}]$$

References: Env p. 11; Store p. 12; TypDef p. 13; TypCon p. 15; Dict p. 10.

9.4 Subtypes

A subtype is declared from the name of an existing type and a subtype constraint.

Syntax Example	A.S. Representation
subtype S **is** T **range** 1 .. 5 ;	$stype \langle\!\mid\ sid \mapsto S,\ def \mapsto ran \langle\!\mid\ parent \mapsto id\ T,\ lower \mapsto lint\ 1,\ upper \mapsto lint\ 5\ \mid\!\rangle\ \mid\!\rangle$

9.4.1 Abstract Syntax

The subtype constraint, including the type from which the subtype is defined, is an element of *SubDef*.

$$STypeBDecl \mathrel{\widehat=} [sid : Id;\ def : SubDef]$$

$$BDecl ::= \ldots \mid stype\langle\!\langle STypeBDecl\rangle\!\rangle$$

9.4.2 Dynamic Semantics

A subtype declaration has no direct effect on the dynamic semantics; instead, it has its effect indirectly, through its impact on the dynamic environment.

$$\frac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta, \delta' : Env;\ \sigma : Store;\ def : SubDef;\\ \ \ tcon : TypCon;\ d : Dict;\ STypeBDecl\\ \mid\\ \qquad \delta'.pdecs = \delta.pdecs\\ \bullet\\ \qquad c, \delta, \sigma \vdash_{typ}\ def \Longrightarrow_{typ}\ tcon\end{array}}{c, \delta, \sigma \vdash_{bdecl}\ stype(\theta STypeBDecl) \Longrightarrow_{bdecl} \delta', \sigma} \qquad \text{(STypeD)}$$

where

$$d == (\delta.dictc)[type := (\delta.dictc).type \oplus \{tid \mapsto tcon\}]$$

References: Env p. 11; Store p. 12; SubDef p. 206; TypCon p. 15; Dict p. 10.

Chapter 10

Private Declarations

The Abstract Syntax of private declarations (*PDecl*) allows private declarations to declare deferred constants, private types and private limited types in the private part of a package specification. None of these declarations have any direct dynamic semantics effect (on the store), their effect instead being indirect through the static environment constructed for a well-formed SPARK text by the Static Semantics.

Chapter 11

Statements

The Abstract Syntax of statements, *Stmt*, is summarised in the following table:

Syntax Constructor	Description	Page
scomp	sequential composition	123
null	null statement	124
asgn	assignment	125
if	if (without else)	130
ifel	if (with else)	131
case	case (without others)	133
caseoth	case (with others)	135
loop	loop (without iteration schema)	140
wloop	while loop	143
floop	for loop	145
floopr	for loop (with range constraint)	149
slabel	statement label (loop name)	151
callp	call (positional association)	152
calln	call (named association)	154

The more complex statements are described using separate syntax categories as follows:

Description	Page
Exit statements	137
Loop segments	139
Actual parameter list	157

Evaluation Predicate

The evaluation predicate for a statement *stmt* includes the environment, and is written thus:

$$c, \delta, \sigma \vdash_s stmt \Longrightarrow_s \sigma'$$

This is read as "statement *stmt* can be evaluated in context c and environment δ with store σ to yield a new store σ'", where $\delta : Env$ and $\sigma, \sigma' : Store$. Context c is a sequence of identifiers, giving a full name prefix for the current scope (e.g. $\langle k, p, q, r \rangle$ for subprogram r embedded within subprogram q embedded within subprogram p of package (or main program) k.) We define the evaluation of statements using inference rules, in which the evaluation of a statement may depend upon the evaluation of its component statements, declarations or expressions.

One other intermediate evaluation predicate we use for the semantics of the SPARK for-loop is

$$c, \delta, \sigma \vdash_{loop} loop \Longrightarrow_{loop} \sigma'$$

which means that the loop representation *loop* (not a statement, but a closely-related object in which the evaluation of the range of values over which the **for** index variable is to range has already been evaluated) is "executed" to give the new store σ'. This is used for the for-loop both with and without a range constraint.

11.1 Sequential Composition

Two or more statements in a list can be used as a statement.

Syntax Example	A.S. Representation
statement1 ; *statement2 ;*	*scomp(statement1, statement2)*

11.1.1 Abstract Syntax

Sequential composition combines two statements.

$$Stmt ::= scomp\langle\!\langle Stmt \times Stmt \rangle\!\rangle \mid \ldots$$

11.1.2 Dynamic Semantics

1. Statements change the store; both statements must be well-formed in the initial environment.

$$\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ s, t : Stmt$$
$$\bullet$$
$$\frac{\begin{array}{l} c, \delta, \sigma \vdash_s s \Longrightarrow_s \sigma' \\ c, \delta, \sigma' \vdash_s t \Longrightarrow_s \sigma'' \end{array}}{c, \delta, \sigma \vdash_s scomp\ (s, t) \Longrightarrow_s \sigma''} \qquad \text{(SCompD)}$$

References: Env p. 11; Store p. 12.

11.2 Null

The null statement does nothing.

Syntax Example	A.S. Representation
null ;	*null*

11.2.1 Abstract Syntax

$Stmt ::= \ldots \mid null$

11.2.2 Dynamic Semantics

The null statement is always executable, and has no effect on the store.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store \\ \bullet}{c, \delta, \sigma \vdash_s null \Longrightarrow_s \sigma} \quad \text{(NullD)}$$

References: Env p. 11; Store p. 12.

11.3 Assignment

An assignment statement assigns a value to a variable object.

Syntax Example	A.S. Representation
k.v := 1	$asgn \langle\!| \; var \mapsto slct \langle\!| \; prefix \mapsto simp\ k, \; selector \mapsto v \; |\!\rangle, \; val \mapsto lint\ 1 \; |\!\rangle$

11.3.1 Abstract Syntax

The variable object is specified by a *Name*; the new value by an expression.

$$AsgnStmt \mathrel{\widehat{=}} [var : Name; \ val : Exp]$$

$$Stmt ::= \ldots \mid asgn \langle\!\langle AsgnStmt \rangle\!\rangle$$

11.3.2 Dynamic Semantics

Assignment can either be to an entire variable (a simple name or a variable declared in another package), or to a field of a record or an element of an array. We cope with this by a syntactic rewrite of assignment statements, and the introduction of an extended expression syntax. For our dynamic semantics, we wish to regard an assignment statement as giving a new value to some *location* in the store σ, this value possibly being some compound object which may be an updated version of the location's previous value.

We first define an extended expression syntax, in which updates to array and record objects can be represented. We define

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{UpdIndComp} \\
arrobj : Name \\
inds : \mathrm{seq}_1\ Exp \\
newval : ExtExp \\
\hline
\end{array}$$

to represent an update to array object *arrobj* in which its *inds*'th element is overwritten by the value *newval*, and

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{UpdSelComp} \\
recobj : Name \\
fldnam : Id \\
newval : ExtExp \\
\hline
\end{array}$$

to represent an update to record object *recobj* in which the field *fldnam* is overwritten by the new value *newval*. Given these schemas, we can define our extended expression syntax by:

$$\begin{aligned} \mathit{ExtExp} ::= {} & \mathit{expr}\langle\!\langle \mathit{Exp} \rangle\!\rangle \\ \mid {} & \mathit{updind}\langle\!\langle \mathit{UpdIndComp} \rangle\!\rangle \\ \mid {} & \mathit{updsel}\langle\!\langle \mathit{UpdSelComp} \rangle\!\rangle \end{aligned}$$

We may now re-express any SPARK assignment statement as a write to a particular variable location in the store, with the value to be assigned to this location calculated by evaluation of a relevant extended expression. Thus, we can visualise each Ada assignment to an array element

$\mathtt{arr(ind)} := \mathtt{e}$

becoming transformed into one of the form

$\mathtt{arr} := \mathtt{updind(arr, ind, e)}$

and similarly for updates to fields of records. The function *rewrite_asgn* which we use to perform this transformation may be defined by:

$$\mathit{rewrite\_asgn} : \mathit{AsgnStmt} \rightarrow\!\!\!\!\!+ (\mathit{Name} \times \mathit{ExtExp})$$
$$\mathit{rewr\_extasgn} : (\mathit{Name} \times \mathit{ExtExp}) \rightarrow\!\!\!\!\!+ (\mathit{Name} \times \mathit{ExtExp})$$

$$\forall\, \mathit{AsgnStmt} \bullet \mathit{rewrite\_asgn}(\theta \mathit{AsgnStmt}) = \mathit{rewr\_extasgn}(\mathit{var}, \mathit{expr}\ \mathit{val})$$

$$\forall\, n : \mathit{Name};\ e : \mathit{ExtExp};\ i : \mathit{Id}$$
$\mid$
$$n \in \operatorname{dom} \mathit{simp}$$
$\bullet$
$$\mathit{rewr\_extasgn}(n, e) = (n, e)$$

$$\forall\, \mathit{pa} : \mathit{PAscName};\ \mathit{ui} : \mathit{UpdIndComp}$$
$\mid$
$$\mathit{ui.arrobj} = \mathit{pa.prefix}$$
$$\mathit{ui.inds} = \mathit{pa.args}$$
$\bullet$
$$\mathit{rewr\_extasgn}(\mathit{pasc}\ \mathit{pa}, \mathit{ui.newval}) = \mathit{rewr\_extasgn}(\mathit{pa.prefix}, \mathit{updind}\ \mathit{ui})$$

$$\forall\, \mathit{sn} : \mathit{SlctName};\ \mathit{us} : \mathit{UpdSelComp}$$
$\mid$
$$\neg\, (\mathit{sn.prefix} \in \operatorname{dom} \mathit{simp})$$
$$\mathit{us.recobj} = \mathit{sn.prefix}$$
$$\mathit{us.fldnam} = \mathit{sn.selector}$$
$\bullet$
$$\mathit{rewr\_extasgn}(\mathit{slct}\ \mathit{sn}, \mathit{us.newval}) = \mathit{rewr\_extasgn}(\mathit{sn.prefix}, \mathit{updsel}\ \mathit{us})$$

We may now define the dynamic semantics of the assignment statement. To do so, we first define a mechanism for the evaluation of extended expressions, using the following operators:

$$\_,\_,\_ \Longrightarrow_{ee} \_ : ((\mathrm{seq}\, Id) \times Env \times Store \times ExtExp) \nrightarrow Val$$

$$\_ \vdash_{ee} \_ : (((\mathrm{seq}\, Id) \times Env \times Store) \times ExtExp) \rightarrow (((\mathrm{seq}\, Id) \times Env \times Store) \times ExtExp)$$

$$\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ e : ExtExp \bullet ((c, \delta, \sigma) \vdash_{ee} e) = ((c, \delta, \sigma), e)$$

For $c : \mathrm{seq}\, Id$, $\delta : Env$, $\sigma : Store$, $e : ExtExp$ and $v : Val$, the predicate

$$c, \delta, \sigma \vdash_{ee} e \Longrightarrow_{ee} v$$

may be read as "in the context c and environment defined by δ and with store σ, the extended-expression e evaluates to value v."

We need just three rules to define the evaluation of extended-expressions in terms of the evaluation of standard expressions:

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ e : Exp;\ v : Val \bullet c, \delta, \sigma \vdash_{e} e \Longrightarrow_{e} v}{c, \delta, \sigma \vdash_{ee} expr\ e \Longrightarrow_{ee} v} \qquad \text{(ExE1)}$$

$$\frac{\begin{array}{l} \forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ ev : Val;\ vals : \mathrm{seq}_1\, Val; \\ \quad v, av : Array\_Value;\ UpdIndComp \\ \mid \\ \qquad av.lo \leq vals\ 1 \leq av.hi \\ \qquad v.lo = av.lo \\ \qquad v.hi = av.hi \\ \qquad v.arr = array\_update(av.arr, vals, ev) \\ \bullet \\ \qquad c, \delta, \sigma \vdash_{e} nam\ arrobj \Longrightarrow_{e} arrval\ av \\ \qquad c, \delta, \sigma \vdash_{es} inds \Longrightarrow_{es} vals \\ \qquad c, \delta, \sigma \vdash_{ee} newval \Longrightarrow_{ee} ev \end{array}}{c, \delta, \sigma \vdash_{ee} updind(\theta\, UpdIndComp) \Longrightarrow_{ee} arrval(v)} \qquad \text{(ExE2)}$$

$$
\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ ev : Val;\\
\quad rv : Id \rightarrow\!\!\!\!\!+\ Val;\ UpdSelComp\\
\mid\\
\qquad \mathit{fldnam} \in \mathrm{dom}\, rv\\
\bullet\\
\qquad c, \delta, \sigma \vdash_e nam\ recobj \Longrightarrow_e recval(rv)\\
\qquad c, \delta, \sigma \vdash_{ee} newval \Longrightarrow_{ee} ev\\
\hline
\qquad c, \delta, \sigma \vdash_{ee} updsel(\theta\, UpdSelComp) \Longrightarrow_{ee}\\
\qquad\qquad rec\ (rv \oplus \{\mathit{fldnam} \mapsto ev\})
\end{array}
\tag{ExE3}
$$

In the above, the function *array_update* is used for the updating of multi-dimensional arrays. We can define this function by:

$$
\begin{array}{|l}
array\_update : (Array\_Value \times \mathrm{seq}_1\ Val \times Val) \rightarrow\!\!\!\!\!+\ Array\_Value\\
\hline
\forall\, av, nv : Array\_Value;\ i, v : Val\\
\mid\\
\qquad av.lo \le i \le av.hi\\
\qquad nv.lo = av.lo\\
\qquad nv.hi = av.hi\\
\qquad nv.arr = av.arr \oplus \{i \mapsto v\}\\
\bullet\\
\qquad update\_array(av, \langle i \rangle, v) = nv\\
\forall\, av, nv : Array\_Value;\ i, v : Val;\ is : \mathrm{seq}_1\ Val\\
\mid\\
\qquad av.lo \le i \le av.hi\\
\qquad nv.lo = av.lo\\
\qquad nv.hi = av.hi\\
\qquad nv.arr = av.arr \oplus \{i \mapsto arrval\ array\_update(arrval^{\sim}(av.arr\ i), is, v)\}\\
\bullet\\
\qquad update\_array(av, \langle i \rangle \frown is, v) = nv
\end{array}
$$

With all of the above, we may now define the dynamic semantics of the assignment statement. There are three cases to consider, according to whether the assignment is to a local variable (via a simple name) or is to a selected component, in which case it may be to a field of a local variable or to a variable from another package.

$$
\begin{array}{l}
\forall\, c, \mathit{fullname} : \mathrm{seq}\, \mathit{Id};\ \delta : \mathit{Env};\ \sigma : \mathit{Store};\ \mathit{ev} : \mathit{Val};\\
\quad \mathit{vid} : \mathit{Id};\ \mathit{vex} : \mathit{ExtExp};\ \mathit{AsgnStmt}\\
\mid\\
\qquad \mathit{rewrite\_asgn}(\theta \mathit{AsgnStmt}) = (\mathit{simp}\ \mathit{vid}, \mathit{vex})\\
\qquad \mathit{fullname} = \mathit{get\_fullname}(c, \delta, \mathit{vid})\\
\bullet\\
\qquad c, \delta, \sigma \vdash_{ee} \mathit{vex} \Longrightarrow_{ee} \mathit{ev}\\
\hline
\qquad c, \delta, \sigma \vdash_{s} \mathit{asgn}(\theta \mathit{AsgnStmt}) \Longrightarrow_{s} \sigma \oplus \{\mathit{fullname} \mapsto \mathit{ev}\}
\end{array}
\qquad \text{(AsgnD1)}
$$

$$
\begin{array}{l}
\forall\, c, \mathit{fullname} : \mathrm{seq}\, \mathit{Id};\ \delta : \mathit{Env};\ \sigma : \mathit{Store};\ \mathit{ev} : \mathit{Val};\\
\quad \mathit{vid} : \mathit{Id};\ \mathit{vex} : \mathit{ExtExp};\ \mathit{pv} : \mathit{SlctName};\ \mathit{AsgnStmt}\\
\mid\\
\qquad \mathit{rewrite\_asgn}(\theta \mathit{AsgnStmt}) = (\mathit{slct}\ \mathit{pv}, \mathit{vex})\\
\qquad \mathit{pv.prefix} \in \mathrm{dom}\, \mathit{simp}\\
\qquad \mathit{fullname} = \mathit{get\_fullname}(c, \delta, \mathit{simp}^{\sim} \mathit{pv.prefix})\\
\qquad \#\mathit{fullname} > 1\\
\bullet\\
\qquad c, \delta, \sigma \vdash_{ee} \mathit{vex} \Longrightarrow_{ee} \mathit{ev}\\
\hline
\qquad c, \delta, \sigma \vdash_{s} \mathit{asgn}(\theta \mathit{AsgnStmt}) \Longrightarrow_{s} \sigma \oplus \{\mathit{fullname} \mapsto \mathit{ev}\}
\end{array}
\qquad \text{(AsgnD2)}
$$

$$
\begin{array}{l}
\forall\, c, \mathit{fullname} : \mathrm{seq}\, \mathit{Id};\ \delta : \mathit{Env};\ \sigma : \mathit{Store};\ \mathit{ev} : \mathit{Val};\\
\quad \mathit{vid} : \mathit{Id};\ \mathit{vex} : \mathit{ExtExp};\ \mathit{pv} : \mathit{SlctName};\ \mathit{AsgnStmt}\\
\mid\\
\qquad \mathit{rewrite\_asgn}(\theta \mathit{AsgnStmt}) = (\mathit{slct}\ \mathit{pv}, \mathit{vex})\\
\qquad \mathit{pv.prefix} \in \mathrm{dom}\, \mathit{simp}\\
\qquad \#\mathit{get\_fullname}(c, \delta, \mathit{simp}^{\sim} \mathit{pv.prefix}) = 1\\
\bullet\\
\qquad c, \delta, \sigma \vdash_{ee} \mathit{vex} \Longrightarrow_{ee} \mathit{ev}\\
\hline
\qquad c, \delta, \sigma \vdash_{s} \mathit{asgn}(\theta \mathit{AsgnStmt}) \Longrightarrow_{s} \sigma \oplus\\
\qquad\qquad \{\langle \mathit{simp}^{\sim} \mathit{pv.prefix}, \mathit{pv.selector}\rangle \mapsto \mathit{ev}\}
\end{array}
\qquad \text{(AsgnD3)}
$$

References: Env p. 11; Store p. 12; Exp p. 47; Val p. 9; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47; *Array_Value* p. 9; *get_fullname* p. 217.

11.4 Simple If

A simple if statement has an if-part, but no else-part.

Syntax Example	A.S. Representation

if v = 0 **then**
v := 1;
end if ;
	$\mathit{if}\ \langle\!|\ \mathit{cond} \mapsto \mathit{binop}\ \langle\!|\ \mathit{larg} \mapsto \mathit{nam}\ (\mathit{simp}\ v),\ \mathit{op} \mapsto \mathit{eq},\ \mathit{rarg} \mapsto \mathit{lint}\ 0\ |\!\rangle,$
	$\mathit{ifpart} \mapsto \mathit{asgn}\ \langle\!|\ \mathit{var} \mapsto \mathit{simp}\ v,\ \mathit{val} \mapsto \mathit{lint}\ 1\ |\!\rangle\ |\!\rangle$

11.4.1 Abstract Syntax

The condition is an expression. The if-part is represented as a statement (a list of statements in the concrete syntax is represented as a composition of statements, using *scomp*).

$$\mathit{IfStmt} \mathrel{\widehat{=}} [\mathit{cond} : \mathit{Exp};\ \mathit{ifpart} : \mathit{Stmt}]$$

$$\mathit{Stmt} ::= \ldots \mid \mathit{if}\langle\!\langle \mathit{IfStmt} \rangle\!\rangle$$

11.4.2 Dynamic Semantics

There are two rules, according to whether the test expression evaluates to *true* or to *false*.

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} \mathit{Id};\ \delta : \mathit{Env};\ \sigma, \sigma' : \mathit{Store};\ \mathit{IfStmt} \\ \bullet \\ \quad c, \delta, \sigma \vdash_e \mathit{cond} \Longrightarrow_e \mathit{enumval}\ (\mathit{id}\ \mathit{true}) \\ \quad c, \delta, \sigma \vdash_s \mathit{ifpart} \Longrightarrow_s \sigma'\end{array}}{c, \delta, \sigma \vdash_s \mathit{if}(\theta \mathit{IfStmt}) \Longrightarrow_s \sigma'} \tag{IfDt}$$

$$\frac{\begin{array}{l}\forall\, c : \operatorname{seq} \mathit{Id};\ \delta : \mathit{Env};\ \sigma : \mathit{Store};\ \mathit{IfStmt} \\ \bullet \\ \quad c, \delta, \sigma \vdash_e \mathit{cond} \Longrightarrow_e \mathit{enumval}\ (\mathit{id}\ \mathit{false})\end{array}}{c, \delta, \sigma \vdash_s \mathit{if}(\theta \mathit{IfStmt}) \Longrightarrow_s \sigma} \tag{IfDf}$$

References: Env p. 11; *Store p. 12*; *Exp p. 47*; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47.

11.5 If-Else Statement

A if-else statement has an if-part and else-part.

Syntax Example	A.S. Representation

if v = 0 **then**
u := 1;
else
u := 2;
end if ;
	$if \; \langle\!\mid \; cond \mapsto binop \; \langle\!\mid \; larg \mapsto nam \; (simp \; v), \; op \mapsto eq, \; rarg \mapsto lint \; 0 \; \mid\!\rangle,$
	$ifpart \mapsto asgn \; \langle\!\mid \; var \mapsto simp \; u, \; val \mapsto lint \; 1 \; \mid\!\rangle,$
	$elsepart \mapsto asgn \; \langle\!\mid \; var \mapsto simp \; u, \; val \mapsto lint \; 2 \; \mid\!\rangle \; \mid\!\rangle$

We regard **elsif** as an abbreviation for the use of nested if statements:

if b1 **then** s1; **elsif** b2 **then** s2; **else** s3; **end if** ;	$\equiv$	**if** b1 **then** s1; **else if** b2 **then** s2; **else** s3; **end if** ; **end if** ;

11.5.1 Abstract Syntax

The condition is an expression. The if-part and else-part are represented as statements.

$$IfElStmt \mathrel{\widehat{=}} [cond : Exp; \; ifpart, elsepart : Stmt]$$

$$Stmt ::= \ldots \mid ifel\langle\!\langle IfElStmt \rangle\!\rangle$$

11.5.2 Dynamic Semantics

There are two rules, according to whether the test expression evaluates to *true* or to *false*.

$$\forall \, c : \mathrm{seq}\, Id; \; \delta : Env; \; \sigma, \sigma' : Store; \; IfElStmt$$
$$\bullet$$
$$\frac{\begin{array}{l} c, \delta, \sigma \vdash_e cond \Longrightarrow_e enumval \; (id \; true) \\ c, \delta, \sigma \vdash_s ifpart \Longrightarrow_s \sigma' \end{array}}{c, \delta, \sigma \vdash_s ifel(\theta IfElStmt) \Longrightarrow_s \sigma'} \qquad \text{(IfElDt)}$$

$$\forall\ c : \mathrm{seq}\ Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ IfElStmt$$
$$\bullet$$
$$\frac{\begin{array}{l} c, \delta, \sigma \vdash_e cond \Longrightarrow_e enumval\ (id\ false) \\ c, \delta, \sigma \vdash_s elsepart \Longrightarrow_s \sigma' \end{array}}{c, \delta, \sigma \vdash_s ifel(\theta IfElStmt) \Longrightarrow_s \sigma'} \qquad \text{(IfElDf)}$$

References: Env p. 11; *Store p. 12*; *Exp p. 47*; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47.

11.6 Case without Others

A case statement selects one of number of alternative statements according to the value of the case index expression.

Syntax Example	A.S. Representation

case colour **is**
  when red =>
    panic;
  when blue | green =>
    null ;
end case ;
	$case \langle\!\mid casindx \mapsto nam\ (simp\ colour),$
	$alterns \mapsto \langle \langle\!\mid altexp \mapsto nam\ (simp\ red),$
	$altstm \mapsto \ldots \mid\!\rangle,$
	$\langle\!\mid altexp \mapsto nam\ (simp\ blue),$
	$altstm \mapsto null \mid\!\rangle,$
	$\langle\!\mid altexp \mapsto nam\ (simp\ green),$
	$altstm \mapsto null \mid\!\rangle \rangle \mid\!\rangle$

11.6.1 Abstract Syntax

Each case alternative has an expression and a statement:

$$CaseAltern \mathrel{\widehat=} [altexp : Exp;\ altstm : Stmt]$$

In the concrete syntax, an alternative can have a list of expressions (separated by |): this is represented in the Abstract Syntax by duplicating the statement of the alternative to form a separate alternative for each expression.

The case statement has an index expression and a list of alternatives.

$$CaseStmt \mathrel{\widehat=} [casindx : Exp;\ alterns : \mathrm{seq}\, CaseAltern]$$

$$Stmt ::= \ldots \mid case \langle\!\langle CaseStmt \rangle\!\rangle$$

11.6.2 Dynamic Semantics

The static semantics checks ensure that the case statement alternatives are complete and disjoint. The evaluation of a case statement therefore boils down to selecting the correct choice from the sequence of alternatives. The static semantics well-formedness constraints ensure that precisely one such alternative exists.

$$\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ n : \mathbb{N};\ val : Val;\ CaseStmt \\
\mid \\
\quad n \in \mathrm{dom}\, alterns \\
\bullet \\
\quad \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_e casindx \Longrightarrow_e val \\ c, \delta, \sigma \vdash_e (alterns\ n).altexp \Longrightarrow_e val \\ c, \delta, \sigma \vdash_s (alterns\ n).altstm \Longrightarrow_s \sigma' \end{array}}{c, \delta, \sigma \vdash_s case(\theta CaseStmt) \Longrightarrow_s \sigma'}
\end{array} \qquad \text{(CaseD)}$$

References: Env p. 11; *Store p. 12*; *Exp p. 47*; *Val p. 9*; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47.

11.7 Case with Others

If the alternatives of a case statement do not cover all the possible values of the index expression, an **others** alternative is given.

Syntax Example	A.S. Representation

case colour **is**
    when red => panic;
    when others => **null** ;
end case ;
	$case \langle\!\mid\ casindx \mapsto nam\ (simp\ colour),$
	$alterns \mapsto \langle\langle\!\mid\ altexp \mapsto nam\ (simp\ red),$
	$altstm \mapsto \ldots \ \mid\!\rangle\rangle,$
	$othcase \mapsto \mathbf{null} \ \mid\!\rangle$

11.7.1 Abstract Syntax

The **others** clause is represented by a statement. *CaseAltern* is defined on page 133.

$$CaseOthStmt \mathrel{\widehat{=}} [casindx : Exp;\ alterns : \mathrm{seq}\, CaseAltern;\ othcase : Stmt]$$

$$Stmt ::= \ldots \mid caseoth\langle\!\langle CaseOthStmt\rangle\!\rangle$$

11.7.2 Dynamic Semantics

The static semantics checks ensure that the case statement alternatives are disjoint, and completeness is given by the presence of the **others** alternative. The evaluation of a case statement therefore boils down to selecting the correct choice from the sequence of alternatives or, if none is applicable, to the use of the **others** statement-part. We use two rules to define this behaviour.

$$\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ n : \mathbb{N};\ val : Val;\\
\quad CaseOthStmt\\
\mid\\
\qquad n \in \mathrm{dom}\, alterns\\
\bullet\\
\qquad c, \delta, \sigma \vdash_e casindx \Longrightarrow_e val\\
\qquad c, \delta, \sigma \vdash_e (alterns\ n).altexp \Longrightarrow_e val\\
\qquad c, \delta, \sigma \vdash_s (alterns\ n).altstm \Longrightarrow_s \sigma'\\
\hline
\qquad c, \delta, \sigma \vdash_s caseoth(\theta CaseOthStmt) \Longrightarrow_s \sigma'
\end{array} \qquad \text{(CasOthD1)}$$

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ val : Val;\ CaseOthStmt \\
\mid \\
\qquad \neg\, (\exists\, n \in \operatorname{dom} alterns \bullet \\
\qquad\qquad c, \delta, \sigma \vdash_e (alterns\ \ n).altexp \Longrightarrow_e val) \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e casindx \Longrightarrow_e val \\
\qquad c, \delta, \sigma \vdash_s othcase \Longrightarrow_s \sigma' \\
\hline
\qquad c, \delta, \sigma \vdash_s caseoth(\theta\, CaseOthStmt) \Longrightarrow_s \sigma'
\end{array}
\qquad \text{(CasOthD2)}
$$

References: Env p. 11; *Store p. 12*; *Exp p. 47*; *Val p. 9*; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47.

11.8 Exit Statements

SPARK includes two forms of conditional exit statement which may be used in loops.

Syntax Example	A.S. Representation
exit when v = 1;	$exitw\ binop\ \langle\!\lvert\ larg \mapsto nam\ (simp\ v),$
	$op \mapsto eq,$
	$rarg \mapsto lint\ 1\ \rvert\!\rangle$
if safe **then**	$ifexit\ \langle\!\lvert\ cond \mapsto nam\ (simp\ safe),$
statement;	$spart \mapsto \ldots\ \rvert\!\rangle$
exit ;	
end if	

The use of exit statements in loops is restricted in SPARK — see Section 11.9.

11.8.1 Abstract Syntax

We introduce the syntax category *ExitStmt* for the two forms of conditional exit statement.

$$IfExitStmt \mathrel{\widehat{=}} [cond : Exp;\ spart : Stmt]$$

$$ExitStmt ::= exitw\langle\!\langle Exp\rangle\!\rangle \mid ifexit\langle\!\langle IfExitStmt\rangle\!\rangle$$

11.8.2 Dynamic Semantics

The dynamic semantics of the exit statement is used in the definition of the dynamic semantics of loop constructs. We regard an exit statement as returning a new store and a continuation flag, of type:

$$ContFlag ::= Cont \mid Exit$$

Given a "continuation" indicator of the above type, we can define the evaluation of an exit statement by its effect on the store and the resulting continuation flag value. Thus, we first introduce

$$\begin{array}{|l}
_ \vdash_{exitstmt} _ : ((\mathrm{seq}\, Id) \times Env \times Store) \times ExitStmt \to \\
\qquad\qquad ((\mathrm{seq}\, Id) \times Env \times Store) \times ExitStmt \\
_ \Longrightarrow_{exitstmt} _ : ((\mathrm{seq}\, Id) \times Env \times Store) \times ExitStmt \nrightarrow (Store \times ContFlag) \\
\hline
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ e : ExitStmt \bullet \\
\qquad (c, \delta, \sigma \vdash_{exitstmt} e) = ((c, \delta, \sigma), e)
\end{array}$$

and define this evaluation of exit statements by the following rules:

$$\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ e : Exp \bullet \frac{c, \delta, \sigma \vdash_e e \Longrightarrow_e enumval\ (id\ true)}{c, \delta, \sigma \vdash_{exitstmt} exitw\ \ e \Longrightarrow_{exitstmt} (\sigma, Exit)} \qquad \text{(Exit1aD)}$$

$$\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ e : Exp \bullet \frac{c, \delta, \sigma \vdash_e e \Longrightarrow_e enumval\ (id\ false)}{c, \delta, \sigma \vdash_{exitstmt} exitw\ \ e \Longrightarrow_{exitstmt} (\sigma, Cont)} \qquad \text{(Exit1bD)}$$

$$\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ IfExitStmt \bullet \frac{\begin{array}{l} c, \delta, \sigma \vdash_e cond \Longrightarrow_e enumval\ (id\ true) \\ c, \delta, \sigma \vdash_s spart \Longrightarrow_s \sigma' \end{array}}{c, \delta, \sigma \vdash_{exitstmt} ifexit(\theta IfExitStmt) \Longrightarrow_{exitstmt} (\sigma', Exit)} \qquad \text{(Exit2aD)}$$

$$\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ IfExitStmt \bullet \frac{c, \delta, \sigma \vdash_e cond \Longrightarrow_e enumval\ (id\ false)}{c, \delta, \sigma \vdash_{exitstmt} ifexit(\theta IfExitStmt) \Longrightarrow_{exitstmt} (\sigma, Cont)} \qquad \text{(Exit2bD)}$$

References: Env p. 11; Store p. 12; Exp p. 47; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47.

11.9 Loop Segments

So that the control flow graph of a SPARK program has an acceptable form, the use of exit statements, which are only allowed within loops, is restricted:

1. An exit always applies to its innermost enclosing loop. Exit statements do not include a loop name.
2. An exit with a **when** part must be immediately enclosed by the loop statement.
3. An unconditional exit must be immediately enclosed within a simple if statement, itself immediately enclosed by the loop statement. The exit must be the last statement in the if statement.

This section introduces a syntax of *loop segments*, which includes only the restricted uses of exit statements. Loop segments are used in the loop bodies — see Sections 11.10, 11.11, 11.12 and 11.13.

Syntax Example	A.S. Representation

x := x + 1;
y := y + 1;
exit when x + y = 10;
	$\langle\!|\ \mathit{loopstmt} \mapsto \mathit{scomp}\ (\mathit{asgn}\ \ldots, \mathit{asgn}\ \ldots),$
	$\mathit{loopexit} \mapsto \mathit{exitw}\ \ldots\ |\!\rangle$

11.9.1 Abstract Syntax

Each segment of a loop has a statement followed by an exit statement (see Section 11.8).

$$\mathit{LoopSeg} \mathrel{\widehat{=}} [\mathit{loopstmt} : \mathit{Stmt};\ \mathit{loopexit} : \mathit{ExitStmt}]$$

11.9.2 Dynamic Semantics

The dynamic semantics of a loop segment is defined implicitly in the definition of the dynamic semantics of loop constructs.
References: Exp p. 47; ExitStmt p. 137.

11.10 General Loop

A general loop has no iteration scheme.

Syntax Example	A.S. Representation
loop	$\mathit{loop}\ \langle\!\vert\ \ \mathit{segs} \mapsto \langle\ \langle\!\vert\ \ \mathit{loopstmt} \mapsto \mathit{asgn}\ \ldots,$
x := x + 1;	$\mathit{loopexit} \mapsto \mathit{exitw}\ \ldots\ \vert\!\rangle,$
exit when x = 10;	$\langle\!\vert\ \ \mathit{loopstmt} \mapsto \mathit{null},$
if y = 10 **then**	$\mathit{loopexit} \mapsto \mathit{ifexit}\ \ldots\ \vert\!\rangle\ \rangle,$
y := y + 1;	$\mathit{spart} \mapsto \mathit{asgn}\ \ldots\ \vert\!\rangle$
exit;	
end if ;	
x := y + 1;	
end loop ;	

11.10.1 Abstract Syntax

The restrictions on the use of exit statements are expressed in the abstract syntax in which a loop contains a list of *segments* (see Section 11.9), with a final statement. Each segment consists of a statement and an exit statement. (Note that the list of segments may not be empty, but the segment statement and the final statements can be *null*).

$$\mathit{LoopStmt} \mathrel{\widehat{=}} [\mathit{segs} : \operatorname{seq} \mathit{LoopSeg};\ \mathit{spart} : \mathit{Stmt}]$$

$$\mathit{LoopStmt1} \mathrel{\widehat{=}} [\mathit{LoopStmt} \mid \#\mathit{segs} \neq 0]$$

$$\mathit{Stmt} ::= \ldots \mid \mathit{loop}\langle\!\langle \mathit{LoopStmt1} \rangle\!\rangle$$

Exit statements (*ExitStmt*) are described in Section 11.8.

11.10.2 Dynamic Semantics

In evaluating a loop body, we must bear in mind that we may either exit from the loop or remain inside it for a further iteration. We thus need to make use of the "continuation" flag returned from the evaluation of the loop body itself.

Before we give the rules for the evaluation of a general loop, we define an evaluation predicate for loop bodies. We use the notation

$$c, \delta, \sigma \vdash_{loopbody} \mathit{lstmt} \Longrightarrow_{loopbody} (\sigma', \mathit{cont})$$

for the evaluation of *lstmt* : *LoopStmt* (the loop statement body) in context c, environment δ : *Env* and with store σ : *Store*, yielding a transformed store σ' : *Store* and a continuation flag *cont* : *ContFlag*.

This evaluation predicate for the loop body may be defined by the following rules:

$$
\frac{
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ LoopStmt \\
| \\
\qquad segs = \langle\rangle \\
\bullet \\
\qquad c, \delta, \sigma \vdash_s spart \Longrightarrow_s \sigma'
\end{array}
}{
c, \delta, \sigma \vdash_{loopbody} (\theta LoopStmt) \Longrightarrow_{loopbody} (\sigma', Cont)
} \qquad \text{(LBody1)}
$$

$$
\frac{
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ LoopStmt_1 \\
\bullet \\
\qquad c, \delta, \sigma \vdash_s (segs\ 1).loopstmt \Longrightarrow_s \sigma' \\
\qquad c, \delta, \sigma' \vdash_{exitstmt} (segs\ 1).loopexit \Longrightarrow_{exitstmt} (\sigma'', Exit)
\end{array}
}{
c, \delta, \sigma \vdash_{loopbody} (\theta LoopStmt) \Longrightarrow_{loopbody} (\sigma'', Exit)
} \qquad \text{(LBody2)}
$$

The next rule (LBody3) derives the new store σ''' via two other intermediate stores; the first, σ', is the value of the store after execution of the first loop segment; this store is then used to evaluate the exit condition, which fails (because the continue flag returned is *Cont*) and returns a new store, σ''. (In fact, from the rules for exit statements, one can see that $\sigma' = \sigma''$ will always hold in such a case.) Finally, the rest of the loop body is executed under store σ'', deriving the new store σ''' and continuation flag *cflg*.

$$
\frac{
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma', \sigma'', \sigma''' : Store;\ loop_1, loop_2 : LoopStmt; \\
\quad cflg : ContFlag \\
| \\
\qquad loop_1.segs \neq \langle\rangle \land \\
\qquad loop_2.spart = loop_1.spart \land \\
\qquad loop_2.segs = (\lambda\, i : 1 \mathrel{..} \#loop_1.segs - 1 \bullet loop_1.segs(i+1)) \\
\bullet \\
\qquad c, \delta, \sigma \vdash_s (loop_1.segs\ 1).loopstmt \Longrightarrow_s \sigma' \\
\qquad c, \delta, \sigma' \vdash_{exitstmt} (loop_1.segs\ 1).loopexit \Longrightarrow_{exitstmt} (\sigma'', Cont) \\
\qquad c, \delta, \sigma'' \vdash_{loopbody} loop_2 \Longrightarrow_{loopbody} (\sigma''', cflg)
\end{array}
}{
c, \delta, \sigma \vdash_{loopbody} loop_1 \Longrightarrow_{loopbody} (\sigma''', cflg)
} \qquad \text{(LBody3)}
$$

We may now deal with the general loop construct, using two rules. The first rule deals with the case when we exit from the loop (having performed the exit action on doing so).

$$\frac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ LoopStmt1 \\ \bullet \\ \quad c, \delta, \sigma \vdash_{loopbody} (\theta LoopStmt1) \Longrightarrow_{loopbody} (\sigma', Exit)\end{array}}{c, \delta, \sigma \vdash_s loop(\theta LoopStmt1) \Longrightarrow_s \sigma'} \quad \text{(LoopD1)}$$

The next rule deals recursively with the meaning of the loop in terms of the repeated "execution" (evaluation) of the loop body.

$$\frac{\begin{array}{l}\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma, \sigma'' : Store;\ LoopStmt1 \\ \bullet \\ \quad c, \delta, \sigma \vdash_{loopbody} (\theta LoopStmt1) \Longrightarrow_{loopbody} (\sigma', Cont) \\ \quad c, \delta, \sigma' \vdash_s loop(\theta LoopStmt1) \Longrightarrow_s \sigma''\end{array}}{c, \delta, \sigma \vdash_s loop(\theta LoopStmt1) \Longrightarrow_s \sigma''} \quad \text{(LoopD2)}$$

N.B. The above rules are not compositional.

References: Env p. 11; *Store p. 12*; *LoopSeg p. 139*; *ContFlag p. 137*; $\vdash_{exitstmt}$ p. 137; $\Longrightarrow_{exitstmt}$ p. 137.

11.11 While Loop

A loop with a while iteration scheme executes the statements for as long as the while condition holds (or until a loop exit is executed).

Syntax Example	A.S. Representation
while x < 10 **loop** x := x + 1; **end loop** ;	$wloop \langle\!| \; cond \mapsto \ldots,$ $wpart \mapsto \ldots \; |\!\rangle$

11.11.1 Abstract Syntax

A while has a condition expression and the components of the general loop (Section 11.10).

$$WLoopStmt \mathrel{\widehat{=}} [cond : Exp;\ LoopStmt]$$

$$Stmt ::= \ldots \mid wloop\langle\!\langle WLoopStmt \rangle\!\rangle$$

11.11.2 Dynamic Semantics

If the condition is false, we do not enter the loop; otherwise, we enter the loop and test the while condition on each successive iteration. This gives rise to the following rules for a while-loop construct.

Firstly, if the while condition is false on entry, the loop body is ignored and the store is left unchanged:

$$\begin{array}{l} \forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, : Store;\ WLoopStmt \\ \bullet \\ \dfrac{\quad c, \delta, \sigma \vdash_e cond \Longrightarrow_e enumval\ (id\ false)}{\quad c, \delta, \sigma \vdash_s wloop(\theta\, WLoopStmt) \Longrightarrow_s \sigma} \end{array} \qquad \text{(WLoopD1)}$$

Next, if the while condition is true, we go around the loop body but encounter an exit statement whose exit test succeeds:

$$\begin{array}{l} \forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma : Store;\ WLoopStmt \\ \bullet \\ \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_e cond \Longrightarrow_e enumval\ (id\ true) \\ c, \delta, \sigma \vdash_{loopbody} (\theta LoopStmt) \Longrightarrow_{loopbody} (\sigma', Exit) \end{array}}{c, \delta, \sigma \vdash_s wloop(\theta\, WLoopStmt) \Longrightarrow_s \sigma'} \end{array} \qquad \text{(WLoopD2)}$$

The final case deals with both a true while condition and no successful exits encountered in the evaluation of the loop body on this iteration:

$$\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma, \sigma'' : Store;\ WLoopStmt \\
\bullet \\
\qquad \dfrac{\begin{array}{l} c, \delta, \sigma \vdash_e cond \Longrightarrow_e enumval\ (id\ true) \\ c, \delta, \sigma \vdash_{loopbody} (\theta LoopStmt) \Longrightarrow_{loopbody} (\sigma', Cont) \\ c, \delta, \sigma' \vdash_s wloop(\theta WLoopStmt) \Longrightarrow_s \sigma'' \end{array}}{c, \delta, \sigma \vdash_s wloop(\theta WLoopStmt) \Longrightarrow_s \sigma''}
\end{array} \qquad \text{(WLoopD3)}$$

This completes all possible cases for the while iteration scheme.

Notes:

1. *As with the general loop construct, the above rules are not compositional.*
2. *An alternative approach to the while-loop, described in the companion Static Semantics document, is to treat the while-loop as a derived form in SPARK, performing a syntactic rewrite to convert each such loop construct into the general loop form.*

References: Env p. 11; *Store p. 12*; *Exp p. 47*; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47; $\vdash_{loopbody}$ p. 140; $\Longrightarrow_{loopbody}$ p. 140.

11.12 For Loop

A loop with a for iteration scheme executes the loop body with each value of an index type assigned to an index variable (unless an exit statement is encountered).

Syntax Example	A.S. Representation
for y **in** T **loop** x := x + 1; **end loop** ;	$floop \langle\!\vert\; indxvar \mapsto y,$ $indxtyp \mapsto id\ \ T,$ $fwdrev \mapsto forward,$ $fpart \mapsto asgn\ \ \ldots\ \ \vert\!\rangle$

11.12.1 Abstract Syntax

The values of the index type are assigned to the index variable either in increasing order (by default) or in decreasing order (if the keyword **reverse** is present).

$$ForRev ::= forward \mid reverse$$

$$FLoopStmt \mathrel{\widehat{=}} [indxvar : Id;\ indxtyp : IdDot;\ fwdrev : ForRev;\ LoopStmt]$$

$$Stmt ::= \ldots \mid floop\langle\!\langle FLoopStmt \rangle\!\rangle$$

11.12.2 Dynamic Semantics

1. SPARK types are non-empty; therefore, the body of this form of for-loop must be executed at least once.

2. The loop body is executed once for each value in the range of values defined by the type-mark, in the order determined by the order (forward or reverse).

We first define a representation of the above loop statement after evaluation of the range part (as given by the index type), and use this to define the evaluation of the loop.

$$
\begin{array}{|l}
\hline
\textit{FLoopEval} \\
\hline
index : \mathrm{seq}\, Id \\
indrng : Val \\
order : ForRev \\
lbody : LoopStmt \\
\hline
indrng \in \mathrm{ran}\ rngval \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ r : \mathbb{P}_1\, Val;\ FloopStmt; \\
\ FLoopEval \\
| \\
\qquad index = c \frown \langle indxvar \rangle \\
\qquad indrng = rngval\ \ r \\
\qquad order = fwdrev \\
\qquad lbody = fpart \\
\qquad \sigma'' = \sigma \oplus (\{index\} \lhd \sigma') \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e indxtyp \Longrightarrow_e rngval\ \ r \\
\qquad c, \delta, \sigma \vdash_{loop} (\theta FLoopEval) \Longrightarrow_{loop} \sigma' \\
\hline
\qquad c, \delta, \sigma \vdash_s floop(\theta FLoopStmt) \Longrightarrow_s \sigma''
\end{array}
\qquad \text{(FLoopD)}
$$

Note that, in the above, the returned value of store, σ'', has all of the effects of execution of the loop body reflected in it *except* for the value of the *index* variable, which (if present in the scope enclosing the loop) is restored to its original, outside-the-loop value.

We define our for-loop evaluation predicate with the following five rules (base case, plus one each for non-empty range in ascending and descending order and with/without encountering a successful exit statement):

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ FLoopEval \\
| \\
\qquad indrng = rngval\ \varnothing \\
\bullet \\
\hline
\qquad c, \delta, \sigma \vdash_{loop} (\theta FLoopEval) \Longrightarrow_{loop} \sigma
\end{array}
\qquad \text{(FLEval1)}
$$

$$
\begin{array}{l}
\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ wholeloop : FLoopEval \\
| \\
\qquad wholeloop.indrng \neq rngval\ \varnothing \\
\qquad wholeloop.order = forward \\
\qquad \sigma' = \sigma \oplus \\
\qquad\qquad \{wholeloop.index \mapsto min\ (rngval^{\sim} wholeloop.indrng)\} \\
\bullet \\
\qquad c, \delta, \sigma' \vdash_{loopbody}\ wholeloop.lbody \Longrightarrow_{loopbody} (\sigma'', Exit) \\
\hline
\qquad c, \delta, \sigma \vdash_{loop}\ wholeloop \Longrightarrow_{loop} \sigma''
\end{array}
\qquad \text{(FLEval2)}
$$

$$\forall\, c : \operatorname{seq} Id;\ \ \delta : Env;\ \ \sigma, \sigma', \sigma'', \sigma''' : Store;\ \ wholeloop : FLoopEval$$
$$|$$
$$\begin{array}{l} wholeloop.indrng \neq rngval\ \varnothing \\ wholeloop.order = forward \\ \sigma' = \sigma \oplus \\ \quad \{wholeloop.index \mapsto min\ (rngval^{\sim} wholeloop.indrng)\} \end{array}$$
$$\bullet$$
$$\frac{\begin{array}{l} c, \delta, \sigma' \vdash_{loopbody}\ wholeloop.lbody \Longrightarrow_{loopbody} (\sigma'', Cont) \\ c, \delta, \sigma'' \vdash_{loop}\ restloop \Longrightarrow_{loop} \sigma''' \end{array}}{c, \delta, \sigma \vdash_{loop}\ wholeloop \Longrightarrow_{loop} \sigma'''} \qquad \text{(FLEval3)}$$

where

$$restloop == wholeloop[\ indrng := rngval\ (rngval^{\sim} wholeloop.indrng \setminus \{min\ (rngval^{\sim} wholeloop.indrng)\})\ ]$$

$$\forall\, c : \operatorname{seq} Id;\ \ \delta : Env;\ \ \sigma, \sigma', \sigma'' : Store;\ \ wholeloop : FLoopEval$$
$$|$$
$$\begin{array}{l} wholeloop.indrng \neq rngval\ \varnothing \\ wholeloop.order = reverse \\ \sigma' = \sigma \oplus \\ \quad \{wholeloop.index \mapsto max\ (rngval^{\sim} wholeloop.indrng)\} \end{array}$$
$$\bullet$$
$$\frac{c, \delta, \sigma' \vdash_{loopbody}\ wholeloop.lbody \Longrightarrow_{loopbody} (\sigma'', Exit)}{c, \delta, \sigma \vdash_{loop}\ wholeloop \Longrightarrow_{loop} \sigma''} \qquad \text{(FLEval4)}$$

$$\forall\, c : \operatorname{seq} Id;\ \ \delta : Env;\ \ \sigma, \sigma', \sigma'', \sigma''' : Store;\ \ wholeloop : FLoopEval$$
$$|$$
$$\begin{array}{l} wholeloop.indrng \neq rngval\ \varnothing \\ wholeloop.order = reverse \\ \sigma' = \sigma \oplus \\ \quad \{wholeloop.index \mapsto max\ (rngval^{\sim} wholeloop.indrng)\} \end{array}$$
$$\bullet$$
$$\frac{\begin{array}{l} c, \delta, \sigma' \vdash_{loopbody}\ wholeloop.lbody \Longrightarrow_{loopbody} (\sigma'', Cont) \\ c, \delta, \sigma'' \vdash_{loop}\ restloop \Longrightarrow_{loop} \sigma''' \end{array}}{c, \delta, \sigma \vdash_{loop}\ wholeloop \Longrightarrow_{loop} \sigma'''} \qquad \text{(FLEval5)}$$

where

$$restloop == wholeloop[\ indrng := rngval\ (rngval^{\sim} wholeloop.indrng \setminus \{max\ (rngval^{\sim} wholeloop.indrng)\})\ ]$$

References: IdDot p. 8; *Env p. 11*; *Store p. 12*; *Val p. 9*; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47; $\vdash_{loop}$ p. 122; $\Longrightarrow_{loop}$ p. 122.

11.13 For Loop with Range

A range constraint can be added to a for iteration scheme so that only the values of the index type within the range are used.

Syntax Example	A.S. Representation
for y **in** T **range** 1 .. 3 **loop** x := x + 1; **end loop** ;	$floopr \langle\!\vert\ indxvar \mapsto y,$ $indxtyp \mapsto id\ T,$ $indxran \mapsto dots \langle\!\vert\ lower \mapsto lint\ 1,$ $upper \mapsto lint\ 3\ \vert\!\rangle,$ $fwdrev \mapsto forward,$ $fpart \mapsto asgn\ \ldots\ \vert\!\rangle$

The concrete syntax of SPARK does not allow the use of a type mark following the keyword **range** in a for iteration schema.

11.13.1 Abstract Syntax

The range constraint is represented by an expression.

$$FLoopRStmt \mathrel{\widehat{=}} [FloopStmt;\ indxran : Exp]$$

$$Stmt ::= \ldots \mid floopr\langle\!\langle FLoopRStmt \rangle\!\rangle$$

11.13.2 Dynamic Semantics

1. If the range is empty, the loop body is not executed.

$$\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ FLoopRStmt$$
$$\bullet$$
$$\frac{c, \delta, \sigma \vdash_e indxran \Longrightarrow_e rngval\ \varnothing}{c, \delta, \sigma \vdash_s floopr(\theta FLoopRStmt) \Longrightarrow_s \sigma} \qquad \text{(FLoopRD1)}$$

2. If the range is non-empty, the loop body is executed once for each value in the range, in the order determined by the order (forward or reverse). In this case, we use a representation of the above loop statement after evaluation of the range part, and use this to define the evaluation of the loop (as with the simple for-loop).

$$
\begin{array}{l}
\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\ r : \mathbb{P}_1\, Val;\\
\ \ FLoopRStmt;\ FLoopEval\\
\mid\\
\qquad index = indxvar\\
\qquad indrng = rngval\ r\\
\qquad order = fwdrev\\
\qquad lbody = fpart\\
\qquad \sigma'' = \sigma \oplus (\{index\} \lhd \sigma')\\
\bullet\\
\qquad c, \delta, \sigma \vdash_e indxran \Longrightarrow_e rngval\ r\\
\qquad c, \delta, \sigma \vdash_{loop} (\theta FLoopEval) \Longrightarrow_{loop} \sigma'\\
\hline
\qquad c, \delta, \sigma \vdash_s floopr(\theta FLoopRStmt) \Longrightarrow_s \sigma''
\end{array}
\qquad \text{(FLoopRD2)}
$$

The for-loop evaluation predicate used above was defined in the preceding section, on for-loops without a range part.

References: Exp p. 47; *Env p. 11*; *Store p. 12*; *Val p. 9*; *FLoopEval p. 145*; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47; $\vdash_{loop}$ p. 122; $\Longrightarrow_{loop}$ p. 122.

11.14 Loop Name

A loop may be named using a loop name.

Syntax Example	A.S. Representation
count: **loop** ... **end loop** count;	$slabel \langle\!\langle\ slabel \mapsto count,\ lstmt \mapsto \ldots,\ elabel \mapsto count\ \rangle\!\rangle$

11.14.1 Abstract Syntax

The representation of loop names in the Abstract Syntax is more general than strictly necessary, since it allows the representation of a label at the start and end of any statement. However, the Concrete Syntax only allows loop statements to be labelled.

$$SlabelStmt \mathrel{\widehat{=}} [slabel, elabel : Id;\ lstmt : Stmt]$$

$$Stmt ::= \ldots \mid slabel\langle\!\langle SLabelStmt\rangle\!\rangle$$

11.14.2 Dynamic Semantics

The presence of a label does not affect the dynamic semantics of a (loop) statement.

$$\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma, \sigma' : Store;\ SlabelStmt \bullet \frac{c, \delta, \sigma \vdash_s lstmt \Longrightarrow_s \sigma'}{c, \delta, \sigma \vdash_s slabel(\theta SlabelStmt) \Longrightarrow_s \sigma'} \qquad \text{(LabD)}$$

References: Env p. 11; Store p. 12.

11.15 Procedure Call — Positional Association

A procedure call requires values to be specified for the formal parameters of a procedure. This can be done using positional association: the order of the actual parameters defines their association to formal parameters.

Syntax Example	A.S. Representation
switch(x,y);	$callp \langle\!\mid\ pid \mapsto id\ switch,$ $actuals \mapsto \langle nam\ (simp\ x), nam\ (simp\ y)\rangle\ \mid\!\rangle$

11.15.1 Abstract Syntax

The procedure name is either an identifier, or may have a package prefix. The actual parameters are specified by a list of expressions.

$$CallPStmt \mathrel{\widehat{=}} [pid : IdDot;\ actuals : \mathrm{seq}\, Exp]$$

$$Stmt ::= \ldots \mid callp\langle\!\langle CallPStmt\rangle\!\rangle$$

11.15.2 Dynamic Semantics

To execute a procedure call (with positional parameter association), the following steps are taken:

1. Actual parameters are associated with formal parameters in the store ("copy-in");
2. Variables local to the procedure are set to their initial values;
3. The procedure body is executed with this new store;
4. Formal parameters are associated with actuals ("copy-out").

This copy-in, copy-out semantics is possible in SPARK — though not in full Ada — because of the static semantic constraints on parameter passign, which precludes the possibility of write-aliasing in a well-formed SPARK text.

We can in fact convert to named association quite simply, as can be seen from the corresponding entry in the Static Semantics document. We therefore do this, allowing us to keep the rules for call by named and positional associations very close together.

$$
\begin{array}{l}
\forall\, c, pc : \operatorname{seq} Id;\ \delta, \delta' : Env;\ \sigma, \sigma_{copiedin}, \sigma_{beforecall},\\
\ \sigma_{aftercall}, \sigma_{copiedout} : Store;\ pn : Id;\ st : Stmt;\\
\ fps : \operatorname{seq} FormalParam;\ CallPStmt\\
\mid\\
\qquad pc \frown \langle pn \rangle = get\_proc\_ctx(c, pid, \delta)\\
\qquad pn \in \operatorname{dom}(\delta.dict\ pc).procs\\
\qquad (\delta.dict\ pc).procs\ pn = (fps, st)\\
\qquad \sigma_{beforecall} = clear\_locals(\sigma_{copiedin}, \delta', pc \frown \langle pn \rangle)\\
\bullet\\
\qquad c, \delta, \sigma \vdash_{copyin} (pc \frown \langle pn \rangle, fps, actuals') \Longrightarrow_{copyin} \delta', \sigma_{copiedin}\\
\qquad pc \frown \langle pn \rangle, \delta', \sigma_{beforecall} \vdash_s st \Longrightarrow_s \sigma_{aftercall}\\
\qquad c, \delta', \sigma_{aftercall} \vdash_{copyout} (pc \frown \langle pn \rangle, fps, actuals') \Longrightarrow_{copyout} \sigma_{copiedout}\\
\hline
\qquad c, \delta, \sigma \vdash_s callp(\theta CallPStmt) \Longrightarrow_s \sigma_{copiedout} \lhd \operatorname{dom} \sigma
\end{array}
\qquad \text{(CallPD)}
$$

where

$$
\begin{array}{l}
actuals' == (\lambda\, i : \operatorname{dom} actuals \bullet (\mu\, NamedActual \mid\\
\qquad\qquad formal = (fps\ i).param \land\\
\qquad\qquad actual = actuals\ i))
\end{array}
$$

References: Env p. 11; *Store p. 12*; *FormalParam p. 160*; *get_proc_ctx* p. 155; *clear_locals* p. 156; $\vdash_{copyin}$ p. 160; $\Longrightarrow_{copyin}$ p. 160; $\vdash_{copyout}$ p. 162; $\Longrightarrow_{copyout}$ p. 162.

11.16 Procedure Call — Named Association

The actual parameters of a procedure call can be given using named association.

Syntax Example	A.S. Representation
switch(fromv => x, tov => y);	$calln \langle\!\mid pid \mapsto id\ switch,$ $actuals \mapsto \langle\ \langle\!\mid formal \mapsto fromv,$ $actual \mapsto nam\ (simp\ x)\ \mid\!\rangle,$ $\langle\!\mid formal \mapsto tov,$ $actual \mapsto nam\ (simp\ y)\ \mid\!\rangle \rangle\ \mid\!\rangle$

11.16.1 Abstract Syntax

A call has a procedure name and a non-empty list of parameters — procedure call with no parameters is considered to be using positional association.

$$CallNStmt \mathrel{\widehat=} [pid : IdDot;\ actuals : \mathrm{seq}_1\ NamedActual]$$

$$Stmt ::= \ldots \mid calln\langle\!\langle CallNStmt\rangle\!\rangle$$

11.16.2 Dynamic Semantics

To execute a procedure call (with named parameter association), the following steps are taken:

1. Actual parameters are associated with formal parameters in the store ("copy-in");
2. Variables local to the procedure are set to their initial values;
3. The procedure body is executed with this new store;
4. Formal parameters are associated with actuals ("copy-out").

The domain of the resulting store can then be restricted to the domain of the store before the call to the procedure.

This copy-in, copy-out semantics is possible in SPARK — though not in full Ada — because of the static semantic constraints on parameter passing, which precludes the possibility of write-aliasing in a well-formed SPARK text.

$$
\begin{array}{l}
\forall\, c, pc : \operatorname{seq} Id;\ \ \delta, \delta' : Env;\ \ \sigma, \sigma_{copiedin}, \sigma_{beforecall},\\
\ \ \sigma_{aftercall}, \sigma_{copiedout} : Store;\ \ pn : Id;\ \ st : Stmt;\\
\ \ fps : \operatorname{seq} FormalParam;\ \ CallNStmt\\
\mid\\
\qquad pc \frown \langle pn \rangle = get\_proc\_ctx(c, pid, \delta)\\
\qquad pn \in \operatorname{dom}(\delta.dict\ \ pc).procs\\
\qquad (\delta.dict\ \ pc).procs\ \ pn = (fps, st)\\
\qquad \sigma_{beforecall} = clear\_locals(\sigma_{copiedin}, \delta', pc \frown \langle pn \rangle)\\
\bullet\\
\qquad c, \delta, \sigma \vdash_{copyin} (pc \frown \langle pn \rangle, fps, actuals) \Longrightarrow_{copyin} \delta', \sigma_{copiedin}\\
\qquad pc \frown \langle pn \rangle, \delta', \sigma_{beforecall} \vdash_{s} st \Longrightarrow_{s} \sigma_{aftercall}\\
\qquad c, \delta', \sigma_{aftercall} \vdash_{copyout} (pc \frown \langle pn \rangle, fps, actuals) \Longrightarrow_{copyout} \sigma_{copiedout}\\
\hline
\qquad c, \delta, \sigma \vdash_{s} calln(\theta CallNStmt) \Longrightarrow_{s} \sigma_{copiedout} \lhd \operatorname{dom} \sigma
\end{array}
\qquad \text{(CallND)}
$$

Two functions *get_proc_ctx* and *clear_locals* are used in the above rule; these can be defined by:

$$
\begin{array}{|l}
get\_proc\_ctx : (\operatorname{seq} Id) \times IdDot \times Env \rightarrow\!\!\!\!\!\!+\ \operatorname{seq} Id\\
\hline
\forall\, c : \operatorname{seq} Id;\ \ \delta : Env;\ \ i : Id \mid i \in \operatorname{dom}(\delta.dict\ \ c).procs \bullet\\
\qquad\qquad get\_proc\_ctx(c, id\ \ i, \delta) = c \frown \langle i \rangle\\
\forall\, c : \operatorname{seq} Id;\ \ \delta : Env;\ \ i, ct : Id \mid i \notin \operatorname{dom}(\delta.dict\ \ c).procs \bullet\\
\qquad\qquad get\_proc\_ctx(c \frown \langle ct \rangle, id\ \ i, \delta) = get\_proc\_ctx(c, id\ \ i, \delta)\\
\forall\, c : \operatorname{seq} Id;\ \ \delta : Env;\ \ k, i : Id \mid i \in \operatorname{dom}(\delta.dict\ \ (c \frown \langle k \rangle)).procs \land\\
\qquad k \notin \operatorname{dom}(\delta.dict\ \ c).procs \land k \notin \operatorname{dom}(\delta.dict\ \ c).funs \bullet\\
\qquad\qquad get\_proc\_ctx(c \frown \langle k, i \rangle, dot(k, i), \delta) = c \frown \langle k, i \rangle\\
\forall\, c : \operatorname{seq} Id;\ \ \delta : Env;\ \ k, i, ct : Id \mid c \frown \langle ct, k \rangle \notin \operatorname{dom} \delta.dict \land\\
\qquad k \notin \operatorname{dom}(\delta.dict\ \ (c \frown \langle ct \rangle)).procs \land k \notin \operatorname{dom}(\delta.dict\ \ (c \frown \langle ct \rangle)).funs \bullet\\
\qquad\qquad get\_proc\_ctx(c \frown \langle ct \rangle, dot(k, i), \delta) = get\_proc\_ctx(c, dot(k, i), \delta)
\end{array}
$$

and

$$
\begin{array}{|l}
clear\_locals : Store \times Env \times (\mathrm{seq}\, Id) \rightarrowtail Store \\
\hline
\forall\, \delta : Env;\ \ c : \mathrm{seq}\, Id;\ \ \sigma_{in}, \sigma_{out} : Store \\
\mid \\
\qquad \mathrm{dom}\, \sigma_{in} = \mathrm{dom}\, \sigma_{out} \\
\qquad \forall\, v : Id \mid v \in \mathrm{dom}(\delta.dict\ \ c).var \bullet \\
\qquad\qquad \sigma_{out}\ (c \frown \langle v \rangle) = form\_init\_val_{\delta}\ (c, (\delta.dict\ \ c).var\ \ v) \\
\qquad (\forall\, s : \mathrm{seq}\, Id \mid \\
\qquad\qquad s \in dom \sigma_{in} \\
\qquad\qquad (\neg\ \exists\, v : Id \bullet s = c \frown \langle v \rangle) \\
\qquad \bullet \\
\qquad\qquad \sigma_{out}\ \ s = \sigma_{in}\ \ s) \\
\bullet \\
\qquad clear\_locals(\sigma_{in}, \delta, c) = \sigma_{out}
\end{array}
$$

References: Env p. 11; *Store p. 12*; *FormalParam p. 160*; $\vdash_{copyin}$ p. 160; $\Longrightarrow_{copyin}$ p. 160; $\vdash_{copyout}$ p. 162; $\Longrightarrow_{copyout}$ p. 162.

11.17 Actual Parameter Lists

This section describes actual parameter lists. Actual parameters can be given using either named or positional association (in SPARK the two forms cannot be mixed). Here, we assume the named association format — a positional association is easily converted to this format (see page 152).

Syntax Example	A.S. Representation
switch(fromv => x, tov => y);	$\langle \langle\!\vert\, \mathit{formal} \mapsto \mathit{fromv}, \mathit{actual} \mapsto \mathit{nam}\ (\mathit{simp}\ x)\ \vert\!\rangle, \langle\!\vert\, \mathit{formal} \mapsto \mathit{tov}, \mathit{actual} \mapsto \mathit{nam}\ (\mathit{simp}\ y)\ \vert\!\rangle\rangle$

11.17.1 Abstract Syntax

The formal parameter is an identifier; the actual parameter is an expression.

$$\mathit{NamedActual} \mathrel{\widehat{=}} [\mathit{formal} : \mathit{Id};\ \mathit{actual} : \mathit{Exp}]$$

11.17.2 Dynamic Semantics

We do not provide any explicit dynamic semantics for actual parameter lists; they are instead handled implicitly where they occur, in procedure and function calls.
References: Exp p. 47.

Chapter 12

Subprogram Declarations

This chapter on subprogram declarations is retained for numbering consistency with the companion Static Semantics document.

The elements of *SDecl* have no effect on the dynamic environment or store constructed by the dynamic semantics. However, the syntax of formal parameters inherited from the Static Semantics is of interest to us and is retained. None of the other sections present in the corresponding chapter of the Static Semantics — global annotations, import lists, derives annotations, subprogram scopes, and procedure and function declarations — are of interest to the dynamic semantics, however. These have therefore been eliminated.

12.1 Formal Parameters

The declaration of function and procedure subprograms includes formal parameters.

Syntax Example	A.S. Representation
(x : **in** T; y : **out** S);	$\langle \langle\!\vert\, param \mapsto x, mode \mapsto in, ptype \mapsto id\ T\ \vert\!\rangle, \langle\!\vert\, param \mapsto y, mode \mapsto out, ptype \mapsto id\ S\ \vert\!\rangle \rangle$,

12.1.1 Abstract Syntax

A formal parameter has a name, a mode and a type. The name is a simple identifier; the type is specified by an element of *IdDot*.

$$FormalParam \mathrel{\widehat{=}} [param : Id;\ mode : Mode;\ ptype : IdDot]$$

12.1.2 Dynamic Semantics

There is no explicit dynamic semantics associated with formal parameters *per se*; rather, the above Abstract Syntax is common to both formal semantics documents, for which reason this section has been retained.

In this section, we shall provide the copy-in, copy-out semantic rules used in the predicates for evaluation of procedure and function calls in this document.

Copy-In Given a context, a full subprogram name, a store, a formal parameter list and an actual parameter list (assumed to be in named association format), we can set up a new store in which the required copying in has been completed. We can do this with the rules below.

First, the empty parameter list: this has no effect on the dynamic environment or the store.

$$\frac{\forall\, c, sid : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store \\ \bullet}{c, \delta, \sigma \vdash_{copyin} (sid, \langle\rangle, \langle\rangle) \Longrightarrow_{copyin} \delta, \sigma} \qquad \text{(CpInD1)}$$

Next, for the case when the first formal parameter needs to be given a value (i.e. when it is not **out** only).

$$\begin{array}{l}
\forall\, c, sid : \mathrm{seq}\, Id;\;\; \delta, \delta'' : Env;\;\; \sigma, \sigma', \sigma'' : Store; \\
\;\; fp : FormalParam;\;\; fps : \mathrm{seq}\, FormalParam; \\
\;\; aps : \mathrm{seq}\, NamedActual;\;\; n : \mathbb{Z};\;\; val : Val \\
| \\
\qquad fp.mode \neq out \\
\qquad n \in \mathrm{dom}\, aps \\
\qquad (aps\;\; n).formal = fp.param \\
\qquad \sigma' = \sigma \oplus \{(sid \frown \langle fp.param \rangle) \mapsto val\} \\
\bullet \\
\qquad c, \delta, \sigma \vdash_e (aps\;\; n).actual \Longrightarrow_e val \\
\qquad c, \delta', \sigma' \vdash_{copyin} (sid, fps, aps) \Longrightarrow_{copyin} \delta'', \sigma'' \\
\hline
\qquad c, \delta, \sigma \vdash_{copyin} (sid, \langle fp \rangle \frown fps, aps) \Longrightarrow_{copyin} \delta'', \sigma''
\end{array} \qquad \text{(CpInD2)}$$

where

$$\delta' == \delta[\; dict := \delta.dict \oplus \{sid \mapsto newdict\}\;]$$

$$newdict == (\delta.dict\;\; sid)[\; var := (\delta.dict\;\; sid).var \oplus \{fp.param \mapsto fp.ptype\}\;]$$

Next, for the case when the first formal parameter is of mode **out**.

$$\begin{array}{l}
\forall\, c, sid : \mathrm{seq}\, Id;\;\; \delta, \delta'' : Env;\;\; \sigma, \sigma', \sigma'' : Store; \\
\;\; fp : FormalParam;\;\; fps : \mathrm{seq}\, FormalParam; \\
\;\; aps : \mathrm{seq}\, NamedActual \\
| \\
\qquad fp.mode = out \\
\qquad \sigma' = \sigma \oplus \{(sid \frown \langle fp.param \rangle) \mapsto \\
\qquad\qquad\qquad form_init_val_\delta\; (sid, fp.ptype)\} \\
\bullet \\
\qquad c, \delta', \sigma' \vdash_{copyin} (sid, fps, aps) \Longrightarrow_{copyin} \delta'', \sigma'' \\
\hline
\qquad c, \delta, \sigma \vdash_{copyin} (sid, \langle fp \rangle \frown fps, aps) \Longrightarrow_{copyin} \delta'', \sigma''
\end{array} \qquad \text{(CpInD3)}$$

where

$$\delta' == \delta[\; dict := \delta.dict \oplus \{sid \mapsto newdict\}\;]$$

$$newdict == (\delta.dict\;\; sid)[\; var := (\delta.dict\;\; sid).var \oplus \{fp.param \mapsto fp.ptype\}\;]$$

This completes the rules for copying in the **in** and **inout** parameters.

N.B. In the above rules, c is the context in which we must evaluate expressions (at the point of call of the subprogram), while *sid* is the context in which the new store elements must be created for use in evaluation of the subprogram body.

Copy-Out Given a context, a full subprogram name, a store, a formal parameter list and an actual parameter list (assumed to be in named association format), we can set up a new store in which the required copying out has been completed. We can do this with the rules below.

First, the empty parameter list: this has no effect on the dynamic environment or the store.

$$\frac{\forall\, c, sid : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store \\ \bullet}{c, \delta, \sigma \vdash_{copyout} (sid, \langle\rangle, \langle\rangle) \Longrightarrow_{copyout} \sigma} \tag{CpOutD1}$$

Next, for the case when the first formal parameter needs to have its value copied back out into the corresponding actual parameter (i.e. when it is not **in** or **default** only).

$$\frac{\begin{array}{l} \forall\, c, sid, actid : \operatorname{seq} Id;\ \delta : Env;\ \sigma, \sigma', \sigma'' : Store; \\ \quad fp : FormalParam;\ fps : \operatorname{seq} FormalParam; \\ \quad aps : \operatorname{seq} NamedActual;\ n : \mathbb{Z};\ val : Val;\ v : IdDot \\ \mid \\ \qquad fp.mode \notin \{in, default\} \\ \qquad n \in \operatorname{dom} aps \\ \qquad (aps\ n).formal = fp.param \\ \qquad (aps\ n).actual \in \operatorname{ran} nam \\ \qquad nam^{\sim}(aps\ n).actual = v \\ \qquad ((v \in \operatorname{ran} simp \wedge actid = get\_fullname(c, \delta, simp^{\sim} v)) \vee \\ \qquad\qquad (v \in \operatorname{ran} slct \wedge actid = slct\_name\_to\_idseq\ v)) \\ \qquad \sigma' = \sigma \oplus \{actid \mapsto val\} \\ \bullet \\ \qquad sid, \delta, \sigma \vdash_e fp.param \Longrightarrow_e val \\ \qquad c, \delta, \sigma' \vdash_{copyout} (sid, fps, aps) \Longrightarrow_{copyout} \sigma'' \end{array}}{c, \delta, \sigma \vdash_{copyout} (sid, \langle fp \rangle \frown fps, aps) \Longrightarrow_{copyout} \sigma''} \tag{CpOutD2}$$

Next, for the case when the first formal parameter is of mode **in** or **default**.

$$\begin{array}{l}
\forall\, c, sid : \operatorname{seq} Id;\; \delta : Env;\; \sigma, \sigma' : Store; \\
\;fp : FormalParam;\; fps : \operatorname{seq} FormalParam; \\
\;aps : \operatorname{seq} NamedActual \\
\mid \\
\qquad fp.mode \in \{in, default\} \\
\bullet \\
\qquad c, \delta, \sigma \vdash_{copyout} (sid, fps, aps) \Longrightarrow_{copyout} \sigma' \\
\hline
\qquad c, \delta, \sigma \vdash_{copyout} (sid, \langle fp \rangle \frown fps, aps) \Longrightarrow_{copyout} \sigma'
\end{array} \qquad \text{(CpOutD3)}$$

This completes the rules for copying out the **inout** and **out** parameters.

N.B. After the copy-out operation, the store should be range-restricted to the same domain as it had prior to the subprogram call; this is to be done in the calling environment.

The function *slct_name_to_idseq* used in the above rules may be defined by:

$$\begin{array}{|l}
slct_name_to_idseq : Name \mathrel{+\!\!\!\!\!\rightarrow} \operatorname{seq} Id \\
\hline
\operatorname{dom} slct_name_to_idseq \subseteq \operatorname{ran} slct \\
\forall\, sn : SlctName \mid \\
\qquad sn.prefix \in \operatorname{ran} simp \bullet \\
\qquad\qquad slct_name_to_idseq\ (slct\ sn) = \\
\qquad\qquad\qquad \langle simp^{\sim} sn.prefix, sn.selector \rangle
\end{array}$$

References: Env p. 11; *Store p. 12*; *NamedActual p. 157*; *Val p. 9*; *IdDot p. 8*; *get_fullname* p. 217; $\vdash_e$ p. 47; $\Longrightarrow_e$ p. 47.

Chapter 13

Embedded Package Declarations

This chapter describes the declaration of embedded packages in SPARK; packages can also be declared as compilation units — see Chapter 17. Embedded package declarations belong to the Abstract Syntax category *KDecl*; the following table summarises the additional forms of declaration:

Syntax Constructor	Description	Page
kspec	declaration of package	166

Embedded package bodies (and body stubs) are described in Chapter 15.

13.1 Package Specification

A package specification declares the types, objects and operations which are to be visible outside the package. Three annotations are required on a package specification:

1. **inherit** lists the other packages whose visible declarations are used in the specification or body of the package being declared.
2. **own** lists the variables which form the state of the package.
3. **initializes** lists the subset of the own-variables which are initialised by the package initialisation.

Syntax Example	A.S. Representation
--# **inherit** l, m;	$kspec \langle\!\lvert\ inherit \mapsto \langle l, m \rangle,$
package k	$kid \mapsto k,$
--# **own** x,y;	$own \mapsto \langle x, y \rangle,$
--# **initializes** x;	$init \mapsto \langle x \rangle,$
is	$vdecl \mapsto \ldots,$
declarations ...	$pdecl \mapsto dnull,$
end k;	$rens \mapsto \ldots \ \rvert\!\rangle$
renames ...	

Additional declarations, which are not visible externally, can be specified in the optional private part. If the package specification is embedded (in the body of another package, or the definition of a subprogram), the package specification may be followed by a list of renames. In the Abstract Syntax, these renames are considered to be part of the package specification. (The use of renames in SPARK is discussed in more detail in Chapter 16.)

13.1.1 Abstract Syntax

Inherited packages and own variables are identifiers.

KSpecKDecl

$kid : Id$
$inherit : \operatorname{seq} Id$
$own : \operatorname{seq} Id$
$init : \operatorname{seq} Id$
$vdecl : VBasic$
$pdecl : PBasic$
$rens : \operatorname{seq} Ren$

$$KDecl ::= kspec \langle\!\langle KSpecKDecl \rangle\!\rangle$$

13.1.2 Dynamic Semantics

The package declaration is evaluated to allow it to be inserted into the current dynamic environment and store.

$$\forall\, c : \mathrm{seq}\, Id;\ \delta, \delta', \delta'' : Env;\ \sigma, \sigma', \sigma'' : Store;\ KSpecKDecl$$
$$\bullet$$
$$\frac{\begin{array}{l} c \frown \langle kid \rangle, \delta, \sigma \vdash_{vbasic} vdecl \Longrightarrow_{vbasic} \delta', \sigma' \\ c \frown \langle kid \rangle, \delta', \sigma' \vdash_{pbasic} pdecl \Longrightarrow_{pbasic} \delta'', \sigma'' \end{array}}{c, \delta, \sigma \vdash_{kdecl} kspec(\theta KSpecKDecl) \Longrightarrow_{kdecl} \delta'', \sigma''} \qquad \text{(KSpecD)}$$

References: Env p. 11; *Store p. 12*; *VBasic p. 101*; *PBasic p. 103*; $\vdash_{vbasic}$ p. 101; $\Longrightarrow_{vbasic}$ p. 101; $\vdash_{pbasic}$ p. 103; $\Longrightarrow_{pbasic}$ p. 103.

Chapter 14

Subprogram Definitions

This chapter describes the subprogram definitions of SPARK, which form the Abstract Syntax category *FDecl*.

In SPARK the declaration of a subprogram can be separated from its definition, giving rise to several forms of declaration for both procedure and function subprograms. A subprogram declared in the visible part of a package specification (see Chapter 12) is *defined*, by giving its implementation, in the package body. When the implementation of a subprogram is to appear in a *separate* file a *stub* is written (note that there are two forms of stub; the other is described in Chapter 15).

The Abstract Syntax of subprogram definitions is summarised in the following table:

14.1 Procedure Definition

A procedure definition appears in a package body to give the implementation of a procedure declared in the visible part of a package specification.

Syntax Example	A.S. Representation
procedure p(x : **in** T; y : **out** S)	$pdefn \langle\!\vert\ pid \mapsto p,$
is	$formals \mapsto \langle \ldots \rangle,$
declarations ...	$dpart \mapsto \ldots,$
begin	$spart \mapsto \ldots \ \vert\!\rangle$
statements ...	
end p;	

14.1.1 Abstract Syntax

The definition repeats the name and formal parameters of the declaration. The annotations are not repeated. Declarations and statements are added to implement the procedure.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PDefnFDecl}} \\
\hline
pid : Id \\
formals : \mathrm{seq}\, FormalParam \\
bdecl : SBasic \\
ldecl : SLater \\
stmt : Stmt \\
\hline
\end{array}
$$

$$FDecl ::= pdefn\langle\!\langle PDefnFDecl \rangle\!\rangle \mid \ldots$$

14.1.2 Dynamic Semantics

On encountering a procedure definition in a given context, we wish to add its salient parts to the dynamic environment for later use when the procedure is called. The components of interest to us are the formal parameters and the procedure body (statement part). We also incorporate the declarations of the procedure in the environment at the same time. We may therefore define:

$$
\begin{array}{l}
\forall\, c, c' : \operatorname{seq} Id;\ \delta, \delta'', \delta''' : Env; \\
\quad \sigma, \sigma', \sigma'' : Store;\ PDefnFDecl \\
| \\
\qquad c' = c \frown \langle pid \rangle \\
\bullet \\
\qquad c', \delta', \sigma \vdash_{sbasic} bdecl \Longrightarrow_{sbasic} \delta'', \sigma' \\
\qquad c', \delta'', \sigma' \vdash_{slater} ldecl \Longrightarrow_{slater} \delta''', \sigma'' \\
\hline
\qquad c, \delta, \sigma \vdash_{fdecl} pdefn(\theta PDefnFDecl) \Longrightarrow_{fdecl} \delta'''', \sigma''
\end{array}
\qquad \text{(PDefnD)}
$$

where

$$\delta' == \delta[\ dict := \delta.dict \oplus \{c' \mapsto EmptyDict\}\ ]$$

$$\delta'''' == \delta'''[\ dict := \delta'''.dict \oplus \{c \mapsto newdict\}\ ]$$

$$newdict == (\delta'''.dict\ \ c)[\ procs :== (\delta.dict\ \ c).procs \oplus \{pid \mapsto (formals, stmt)\}\ ]$$

In the above, the empty dictionary *EmptyDict* is given by:

$$
\begin{array}{|l}
\hline
EmptyDict \\
\hline
Dict \\
\hline
withs = \langle\rangle \\
\operatorname{dom} const = \varnothing \\
\operatorname{dom} type = \varnothing \\
\operatorname{dom} var = \varnothing \\
\operatorname{dom} procs = \varnothing \\
\operatorname{dom} funs = \varnothing \\
\hline
\end{array}
$$

References: Env p. 11; Store p. 12; FormalParam p. 160; SBasic p. 105; SLater p. 109; Stmt p. 121; $\vdash_{sbasic}$ p. 105; $\Longrightarrow_{sbasic}$ p. 105; $\vdash_{slater}$ p. 109; $\Longrightarrow_{slater}$ p. 109.

14.2 Procedure Stub

A package body may also include a stub for a procedure already declared in the package specification.

Syntax Example	A.S. Representation
procedure p(x : **in** T) **is separate** ;	$pstub \; \langle\!| \; pid \mapsto p, \; params \mapsto \langle \langle\!| \; param \mapsto x, \; mode \mapsto in, \; ptype \mapsto id \; T \; |\!\rangle \rangle \; |\!\rangle$

14.2.1 Abstract Syntax

The stub gives the name and formal parameters of the procedure. Since the annotations appear in the declaration (in the package specification) they are not repeated in the stub.

$$
\begin{array}{|l}
\hline
\textit{PStubFDecl} \\
\textit{pid} : \textit{Id} \\
\textit{params} : \operatorname{seq} \textit{FormalParam} \\
\hline
\end{array}
$$

$FDecl ::= \ldots \mid pstub \langle\!\langle PStubFDecl \rangle\!\rangle$

14.2.2 Dynamic Semantics

On encountering a procedure stub in this context, we can ignore it: its formal parameters, procedure body and declarations will be added to the static environment and store when the separate part is encountered.

$$
\frac{\forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ PStubFDecl \\ \bullet}{c, \delta, \sigma \vdash_{fdecl} pstub(\theta PStubFDecl) \Longrightarrow_{fdecl} \delta, \sigma} \qquad \text{(PStubD)}
$$

References: FormalParam p. 160; Env p. 11; Store p. 12.

14.3 Function Definition

A function definition appears in a package body to give the implementation of a function declared in the visible part of a package specification.

Syntax Example	A.S. Representation
function f(x : **in** T) **return** S	$\mathit{fdefn}\ \langle\!|\ \mathit{fid} \mapsto f,$
is	$\mathit{formals} \mapsto \langle \ldots \rangle,$
declarations ...	$\mathit{rtype} \mapsto \mathit{id}\ S,$
begin	$\mathit{dpart} \mapsto \ldots,$
statements ...	$\mathit{spart} \mapsto \ldots,$
return x + 1;	$\mathit{return} \mapsto \ldots\ |\!\rangle$
end f;	

14.3.1 Abstract Syntax

The definition repeats the name, formal parameters and return type of the declaration. The annotations are not repeated. Declarations and statements are added to implement the function.

In SPARK, the last statement of a function body must be a **return** statement. This is the only use of the **return** statment. The expression which specifies the return value is therefore included in the abstract syntax of function definitions.

FDefnFDecl

$\mathit{fid} : \mathit{Id}$
$\mathit{formals} : \mathrm{seq}\,\mathit{FormalParam}$
$\mathit{rtype} : \mathit{IdDot}$
$\mathit{bdecl} : \mathit{SBasic}$
$\mathit{ldecl} : \mathit{SLater}$
$\mathit{stmt} : \mathit{Stmt}$
$\mathit{return} : \mathit{Exp}$

$$\mathit{FDecl} ::= \ldots \mid \mathit{fdefn}\langle\!\langle \mathit{FDefnFDecl} \rangle\!\rangle$$

14.3.2 Dynamic Semantics

On encountering a function definition in a given context, we wish to add its salient parts to the dynamic environment for later use when the function is called. The components of interest to us are the formal parameters, the function body (statement part), the return expression and its type. We also incorporate the declarations of the function in the environment at the same time. We may therefore define:

$$\forall\, c, c' : \operatorname{seq} Id;\ \delta, \delta'', \delta''' : Env;$$
$$\sigma, \sigma', \sigma'' : Store;\ FDefnFDecl$$
$$|$$
$$c' = c \frown \langle fid \rangle$$
$$\bullet$$
$$\frac{\begin{array}{l} c', \delta', \sigma \vdash_{sbasic} bdecl \Longrightarrow_{sbasic} \delta'', \sigma' \\ c', \delta'', \sigma' \vdash_{slater} ldecl \Longrightarrow_{slater} \delta''', \sigma'' \end{array}}{c, \delta, \sigma \vdash_{fdecl} fdefn(\theta FDefnFDecl) \Longrightarrow_{fdecl} \delta'''', \sigma''} \qquad \text{(FDefnD)}$$

where

$$\delta' == \delta[\ dict := \delta.dict \oplus \{c' \mapsto EmptyDict\}\ ]$$
$$\delta'''' == \delta'''[\ dict := \delta'''.dict \oplus \{c \mapsto newdict\}\ ]$$
$$newdict == (\delta'''.dict\ \ c)[\ funs := (\delta.dict\ \ c).funs \oplus \{fid \mapsto (formals, stmt, return, rtype)\}\ ]$$

References: Env p. 11; *Store p. 12*; *FormalParam p. 160*; *IdDot p. 8 SBasic p. 105*; *SLater p. 109*; *Stmt p. 121*; *Exp p. 47*; $\vdash_{sbasic}$ p. 105; $\Longrightarrow_{sbasic}$ p. 105; $\vdash_{slater}$ p. 109; $\Longrightarrow_{slater}$ p. 109; *EmptyDict p. 171.*

14.4 Function Stub

A package body may also include a stub for a function already declared in the package specification.

Syntax Example	A.S. Representation
function f(x : **in** T) **return** S **is separate** ;	$\mathit{fstub}\ \langle\!|\ \mathit{pid} \mapsto f,\ \mathit{params} \mapsto \langle\ \langle\!|\ \mathit{param} \mapsto x,\ \mathit{mode} \mapsto \mathit{in},\ \mathit{ptype} \mapsto \mathit{id}\ T\ |\!\rangle \rangle,\ \mathit{rtype} \mapsto \mathit{id}\ S\ |\!\rangle$

14.4.1 Abstract Syntax

The stub gives the name, formal parameters and return type of the function. The annotations, which appear in the declaration (in the package specification) are not repeated in the stub.

$$
\begin{array}{|l}
\hline
\mathit{FStubFDecl} \\
\hline
\mathit{pid} : \mathit{Id} \\
\mathit{params} : \mathrm{seq}\, \mathit{FormalParam} \\
\mathit{rtype} : \mathit{IdDot} \\
\hline
\end{array}
$$

$$\mathit{FDecl} ::= \ldots \mid \mathit{fstub}\langle\!\langle \mathit{FStubFDecl} \rangle\!\rangle$$

14.4.2 Dynamic Semantics

On encountering a function stub in this context, we can ignore it: its formal parameters, function body and declarations will be added to the static environment and store when the separate part is encountered.

$$\frac{\forall\, c : \mathrm{seq}\, \mathit{Id};\ \delta : \mathit{Env};\ \sigma : \mathit{Store};\ \mathit{FStubFDecl} \\ \bullet}{c, \delta, \sigma \vdash_{\mathit{fdecl}} \mathit{fstub}(\theta \mathit{FStubFDecl}) \Longrightarrow_{\mathit{fdecl}} \delta, \sigma} \qquad \text{(FStubD)}$$

References: FormalParam p. 160; IdDot p. 8; Env p. 11; Store p. 12.

Chapter 15

Subprogram and Package Bodies

This chapter describes the declaration of subprogram and package bodies which form the Abstract Syntax category *YDecl*.

The declarations in this category include only those which can appear within both subprogram and package bodies; thus subprograms whose declarations and definitions are separated (in package specification and body respectively) are excluded. The allowed form, in which all the components of the declarations are given together, is termed a *local* subprogram. The implementation of a subprogram can appear in a *separate* file, giving rise to a form of *stub*. Package bodies can also be declared, possibly using a stub.

The Abstract Syntax of subprogram and package bodies is summarised in the following table:

Syntax Constructor	Description	Page
plocl	Local procedure declaration in a package body	178
pstua	Separate declaration for a local procedure	180
flocl	Local function declaration in a package body	181
fstua	Separate declaration for a local function	183
kbody	Declaration of package body	184
kstub	Package body stub	186

15.1 Local Procedures

A package body may contain a procedure declaration which is not mentioned in the package specification.

Syntax Example	A.S. Representation
procedure p(x : **in** T)	$plocl\ \langle\!|\ pid \mapsto p,$
- -# global y ;	$formals \mapsto \langle\rangle,$
- -# derives y **from** x ;	$globals \mapsto \langle id\ \ y\rangle,$
is	$derives \mapsto \langle\ldots\rangle,$
declarations ...	$bdecl \mapsto \ldots,$
begin	$ldecl \mapsto \ldots,$
statements ...	$stmt \mapsto \ldots\ |\!\rangle$
end p;	

15.1.1 Abstract Syntax

A local procedure declaration combines the procedure declaration (Section 10) and definition (Section 14.1).

$$PLoclYDecl \mathrel{\widehat{=}} PDeclSDecl \wedge PDefnFDecl$$

The complete declaration has a procedure name (*pid*), formal parameters (*formals*), the annotations (*globals* and *derives*), and the declarations (*bdecl* and *ldecl*) and statements (*stmt*) required to implement the procedure.

$$YDecl ::= plocl\langle\!\langle PLoclYDecl\rangle\!\rangle \mid \ldots$$

15.1.2 Dynamic Semantics

On encountering a local procedure definition in a given context, we wish to add its salient parts to the dynamic environment for later use when the procedure is called. The components of interest to us are the formal parameters and the procedure body (statement part). We also incorporate the declarations of the procedure in the environment at the same time. We may therefore define:

$$
\begin{array}{l}
\forall\, c, c' : \mathrm{seq}\, Id;\ \delta, \delta'', \delta''' : Env; \\
\ \ \sigma, \sigma', \sigma'' : Store;\ PLoclYDecl \\
| \\
\qquad c' = c \frown \langle pid \rangle \\
\bullet \\
\qquad \dfrac{\begin{array}{l} c', \delta', \sigma \vdash_{sbasic} bdecl \Longrightarrow_{sbasic} \delta'', \sigma' \\ c', \delta'', \sigma' \vdash_{slater} ldecl \Longrightarrow_{slater} \delta''', \sigma'' \end{array}}{c, \delta, \sigma \vdash_{ydecl} plocl(\theta PLoclFDecl) \Longrightarrow_{ydecl} \delta'''', \sigma''}
\end{array}
\qquad \text{(PLoclD)}
$$

where

$$\delta' == \delta[\ dict := \delta.dict \oplus \{c' \mapsto EmptyDict\}\ ]$$

$$\delta'''' == \delta'''[\ dict := \delta'''.dict \oplus \{c \mapsto newdict\}\ ]$$

$$newdict == (\delta'''.dict\ \ c)[\ procs := (\delta.dict\ \ c).procs \oplus \{pid \mapsto (formals, stmt)\}\ ]$$

References: Env p. 11; *Store p. 12*; $\vdash_{sbasic}$ p. 105; $\Longrightarrow_{sbasic}$ p. 105; $\vdash_{slater}$ p. 109; $\Longrightarrow_{slater}$ p. 109; *EmptyDict p. 171.*

15.2 Local Procedure Stub

A local procedure stub declares a local procedure with a separate implementation.

Syntax Example	A.S. Representation
procedure p(x : **in** T)	$pstua \langle\!\mid\ pid \mapsto p,$
--# **global** y;	$params \mapsto \langle \ldots \rangle,$
--# **derives** y **from** x;	$global \mapsto \langle id\ \ y \rangle,$
is separate ;	$derives \mapsto \langle \ldots \rangle\ \mid\!\rangle$

15.2.1 Abstract Syntax

Since the procedure is local — not declared in the package specification — the stub includes annotations, as well as the name and formal parameters.

$$
\begin{array}{|l}
\hline
\textit{PStuaYDecl} \\
pid : Id \\
params : \mathrm{seq}\, FormalParam \\
global : GlobalAnnot \\
derives : \mathrm{seq}\, DerivesAnnot \\
\hline
\end{array}
$$

$$YDecl ::= \ldots \mid pstua\langle\!\langle PStuaYDecl \rangle\!\rangle$$

15.2.2 Dynamic Semantics

On encountering a procedure stub in this context, we can ignore it: its formal parameters, procedure body and declarations will be added to the static environment and store when the separate part is encountered.

$$
\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ PStuaYDecl \\ \bullet}{c, \delta, \sigma \vdash_{ydecl} pstua(\theta PStuaYDecl) \Longrightarrow_{ydecl} \delta, \sigma} \qquad \text{(PStuaD)}
$$

References: FormalParam p. 160; Env p. 11; Store p. 12.

15.3 Local Functions

A package body may contain a function declaration which is not mentioned in the package specification.

Syntax Example	A.S. Representation
function f(x : **in** T) **return** S	$flocl \langle\!\langle$ $fid \mapsto f$,
- -# global y ;	$formals \mapsto \langle\rangle$,
is	$rtype \mapsto id\ S$,
declarations ...	$global \mapsto \langle id\ y \rangle$,
begin	$bdecl \mapsto \ldots$,
statements ...	$ldecl \mapsto \ldots$,
return x + 1;	$stmt \mapsto \ldots$,
end f;	$return \mapsto \ldots \rangle\!\rangle$

15.3.1 Abstract Syntax

A local function declaration combines the function declaration (Section 14) and definition (Section 14.3).

$$FLoclYDecl \mathrel{\widehat{=}} FDeclSDecl \land FDefnFDecl$$

The complete declaration has a function name (*fid*), formal parameters (*formals*), return type *rtype*, global annotation (*globals*), and the declarations (*bdecl* and *ldecl*), statements (*stmt*) and result expression *return* required to implement the function.

$$YDecl ::= \ldots \mid flocl\langle\!\langle FLoclYDecl \rangle\!\rangle$$

15.3.2 Dynamic Semantics

On encountering a local function definition in a given context, we wish to add its salient parts to the dynamic environment for later use when the function is called. The components of interest to us are the formal parameters, the function body (statement part), the return expression and its type. We also incorporate the declarations of the function in the environment at the same time. We may therefore define:

$$
\begin{array}{l}
\forall\, c, c' : \mathrm{seq}\, Id;\;\; \delta, \delta'', \delta''' : Env; \\
\;\; \sigma, \sigma', \sigma'' : Store;\;\; FLoclYDecl \\
| \\
\qquad c' = c \frown \langle fid \rangle \\
\bullet \\
\qquad c', \delta', \sigma \vdash_{sbasic} bdecl \Longrightarrow_{sbasic} \delta'', \sigma' \\
\qquad c', \delta'', \sigma' \vdash_{slater} ldecl \Longrightarrow_{slater} \delta''', \sigma'' \\
\hline
\qquad c, \delta, \sigma \vdash_{ydecl} flocl(\theta FLoclYDecl) \Longrightarrow_{ydecl} \delta'''', \sigma''
\end{array}
\qquad \text{(FLoclD)}
$$

where

$$\delta' == \delta[\; dict := \delta.dict \oplus \{c' \mapsto EmptyDict\}\;]$$

$$\delta'''' == \delta'''[\; dict := \delta'''.dict \oplus \{c \mapsto newdict\}\;]$$

$$
\begin{array}{l}
newdict == (\delta'''.dict\;\; c)[\; funs := (\delta.dict\;\; c).funs \oplus \\
\qquad\qquad\qquad \{fid \mapsto (formals, stmt, return, rtype)\}\;]
\end{array}
$$

References: Env p. 11; *Store p. 12*; $\vdash_{sbasic}$ p. 105; $\Longrightarrow_{sbasic}$ p. 105; $\vdash_{slater}$ p. 109; $\Longrightarrow_{slater}$ p. 109; *EmptyDict p. 171.*

15.4 Local Function Stub

A local function stub declares a local function with a separate implementation.

Syntax Example	A.S. Representation
function f(x : **in** T) **return** S **- -# global** y; **is separate** ;	$fstua \;\langle\!|\; pid \mapsto f,$ $params \mapsto \langle \ldots \rangle,$ $global \mapsto \langle id\ \ y \rangle,$ $rtype \mapsto id\ \ S \;|\!\rangle$

15.4.1 Abstract Syntax

Since the function is local — not declared in the package specification — the stub includes the global annotation, as well as the name, formal parameters and return type.

$$
\begin{array}{|l}
\hline
\textit{FStuaYDecl} \\
pid : Id \\
params : \operatorname{seq} FormalParam \\
global : GlobalAnnot \\
rtype : IdDot \\
\hline
\end{array}
$$

$$YDecl ::= \ldots \mid fstua\langle\!\langle FStuaYDecl \rangle\!\rangle$$

15.4.2 Dynamic Semantics

On encountering a function stub in this context, we can ignore it: its formal parameters, function body and declarations will be added to the static environment and store when the separate part is encountered.

$$
\frac{\begin{array}{l} \forall\, c : \operatorname{seq} Id;\ \delta : Env;\ \sigma : Store;\ FStuaYDecl \\ \bullet \end{array}}{c, \delta, \sigma \vdash_{ydecl} fstua(\theta FStuaYDecl) \Longrightarrow_{ydecl} \delta, \sigma} \qquad \text{(FStuaD)}
$$

References: FormalParam p. 160; IdDot p. 8; Env p. 11; Store p. 12.

15.5 Package Body

A package body contains declarations implementing the operations declared in the visible part of the package specification. A package body optionally contains statements to initialise some of the own-variables of the package.

Syntax Example	A.S. Representation
package body k **is**	$kbody \langle\!\vert\ kid \mapsto k,$
declarations ...	$rens \mapsto \langle\rangle,$
begin	$bdecl \mapsto basic\ declarations\ \ldots,$
statements ...	$ldecl \mapsto later\ declarations\ \ldots,$
end k;	$init \mapsto statements\ \ldots\ \vert\!\rangle$

The declarations of the package are optionally preceded by a list of renames, which apply to operations from inherited packages. (The use of renames in SPARK is discussed in more detail in Chapter 16.)

15.5.1 Abstract Syntax

The package name is an identifier; a package body contains a declaration.

KBodyYDecl

$kid : Id$
$rens : \mathrm{seq}\, Ren$
$bdecl : KBasic$
$ldecl : KLater$
$init : Stmt$

$$YDecl ::= \ldots \mid kbody\langle\!\langle KBodyYDecl \rangle\!\rangle$$

15.5.2 Dynamic Semantics

The effect of encountering a package body is to update the dynamic environment and store to take into account the declarations in the package body. The package must already have been declared (i.e. its package specification encountered) but not previously defined. We therefore define:

$$
\frac{\begin{array}{l}
\forall\, c' : \mathrm{seq}\, Id;\ \delta, \delta', \delta'' : Env;\ \sigma, \sigma', \sigma'' : Store;\ KBodyYDecl \\
\mid \\
\quad kid \in pdecs \\
\quad kid \notin \mathrm{dom}\, pdefs \\
\quad c' = c \frown \langle kid \rangle \\
\bullet \\
\quad c', \delta, \sigma \vdash_{kbasic} bdecl \Longrightarrow_{kbasic} \delta', \sigma' \\
\quad c', \delta', \sigma' \vdash_{klater} ldecl \Longrightarrow_{klater} \delta'', \sigma''
\end{array}}{c, \delta, \sigma \vdash_{ydecl} kbody(\theta KBodyYDecl) \Longrightarrow_{ydecl} \delta''', \sigma''} \qquad \text{(KBodyD)}
$$

where

$$\delta''' == \delta''[\ pdefs := \delta''.pdefs \cup \{kid \mapsto init\}\ ]$$

References: KBasic p. 107; *KLater p. 111*; *Stmt p. 121*; *Env p. 11*; *Store p. 12*; $\vdash_{kbasic}$ p. 107; $\Longrightarrow_{kbasic}$ p. 107; $\vdash_{klater}$ p. 111; $\Longrightarrow_{klater}$ p. 111.

15.6 Package Body Stub

A package body stub is used to specify that a package, which is not a library unit, is to be compiled separately.

Syntax Example	A.S. Representation
package body k **is separate** ;	*kstub k*

15.6.1 Abstract Syntax

$YDecl ::= \ldots \mid kstub\langle\langle Id\rangle\rangle$

15.6.2 Dynamic Semantics

On encountering a package body stub in this context, we can ignore it: its constituents will be added to the static environment and store when the separate part is encountered.

$$\frac{\forall\, c : \mathrm{seq}\, Id;\ \delta : Env;\ \sigma : Store;\ k : Id \\ \bullet}{c, \delta, \sigma \vdash_{ydecl} kstub\ k \Longrightarrow_{ydecl} \delta, \sigma} \quad \text{(KStubD)}$$

References: Env p. 11; Store p. 12.

Chapter 16

Renames

The renaming of operators supported by the SPARK subset of Ada have no direct impact or bearing on the dynamic semantics of the language and so need not be further discussed in this document.

We do not consider subprogram renaming further here.

Chapter 17

Compilation Units and SPARK Texts

A SPARK program is a collection of one or more compilation units. A compilation is so called because it is a SPARK term which can be compiled separately. In order to allow separate compilation, a list of the other compilation units made use of must be given at the start of a compilation unit. This list is a **with** context clause.

The Abstract Syntax of compilation units (*Unit*) in SPARK is summarized in the following table:

Syntax Constructor	Description	Page
specu	package specification	193
bodyu	package body	195
subu	subunit, i.e. something declared to be separate	198
main	main program	199

A complete SPARK Program (*SPARK*, page 201) is also described in this Chapter.

Dynamic Semantics of Compilation Units

The dynamic semantics effect of encountering a compilation unit is expressed by the effect it has on the dynamic environment and the store. The predicate

$$\delta, \sigma \vdash_c comp \Longrightarrow_c \delta', \sigma'$$

where *comp* is a compilation unit, δ, δ' are dynamic environments and σ, σ' are stores states that the effect of encountering the compilation unit *comp* on environment δ and store σ is to deliver a new environment δ' and store σ'. We assume that, to begin with (i.e. before any compilation units have been encountered), the store's domain is empty and the environment is equal to the empty environment given below:

$$
\begin{array}{|l}
\hline
\textit{EmptyEnv} \\
\hline
\mathit{Env} \\
\hline
\operatorname{dom} \mathit{dict} = \varnothing \\
\mathit{pdecs} = \varnothing \\
\hline
\end{array}
$$

17.1 The Own and Initializes Annotations

The **own** annotation "announces" variables which will later be declared in the outer-most scope of the package specification or its body. The **initializes** annotation gives the subset of these variables which are initialised in the initialisation statement in the package body.

Syntax Example	A.S. Representation
--# **own** x,y;	$\langle x, y \rangle$
--# **initializes** x;	$\langle x \rangle$

17.1.1 Abstract Syntax

The **own** and **initializes** annotations are represented as elements of seq Id.

17.1.2 Dynamic Semantics

The **own** and **initializes** annotations have no dynamic semantic effects on a SPARK program text.

17.2 With Clauses and Inherit Annotations

Compilation units have **with** clauses and **inherit** annotations (except for subunits). Both have a list of package identifiers.

Syntax Example	A.S. Representation
with l;	$\langle l \rangle$
--# **inherit** l, m;	$\langle l, m \rangle$

17.2.1 Abstract Syntax

The **with** clause and **inherit** annotations are represented by elements of seq Id.

17.2.2 Dynamic Semantics

The **with** clause and **inherit** annotations have no effect on the dynamic semantics of SPARK.

17.3 Package Specification

A package specification can be used as a library unit; it declares the types, objects and operations which are to be visible outside the package. Three annotations are required on a package specification:

1. **inherit** lists the other packages whose visible declarations are used in the specification or body of the package being declared.
2. **own** lists the variables which form the state of the package.
3. **initializes** lists the subset of the own-variables which are initialised by the package initialisation.

Syntax Example	A.S. Representation
with l;	$kspec \langle\!| \; with \mapsto \langle l \rangle,$
- -# inherit l, m;	$inherit \mapsto \langle l, m \rangle,$
package k	$kid \mapsto k,$
- -# own x,y;	$own \mapsto \langle x, y \rangle,$
- -# initializes x;	$init \mapsto \langle x \rangle,$
is	$vdecl \mapsto \ldots,$
declarations ...	$pdecl \mapsto dnull \; |\!\rangle$
end k;	

Additional declarations, which are not visible externally, can be specified in the optional private part.

17.3.1 Abstract Syntax

Inherited packages and own variables are identifiers.

$$
\begin{array}{|l}
\hline
\textit{SpecUnit} \\
with : \operatorname{seq} Id \\
kid : Id \\
inherit : \operatorname{seq} Id \\
own : \operatorname{seq} Id \\
init : \operatorname{seq} Id \\
vdecl : VBasic \\
pdecl : PBasic \\
\hline
\end{array}
$$

$$Unit ::= specu\langle\!\langle SpecUnit \rangle\!\rangle \mid \ldots$$

17.3.2 Dynamic Semantics

The dynamic semantic effects of a package is given by its effect on the construction of the dynamic environment and the store.

$$\dfrac{\begin{array}{l} \forall\, \delta, \delta', \delta'', \delta''' : Env;\ \sigma, \sigma', \sigma'' : Store;\ SpecUnit \\ | \\ \quad kid \notin \delta.pdecs \\ \quad (\forall\, p \in \operatorname{ran} with \bullet p \in \delta.pdecs) \\ \bullet \\ \quad \langle kid \rangle, \delta, \sigma \vdash_{vbasic} vdecl \Longrightarrow_{vbasic} \delta', \sigma' \\ \quad \langle kid \rangle, \delta', \sigma' \vdash_{pbasic} pdecl \Longrightarrow_{pbasic} \delta'', \sigma'' \end{array}}{\delta, \sigma \vdash_c specu(\theta SpecUnit) \Longrightarrow_c \delta''', \sigma''} \qquad \text{(SpecUD)}$$

where

$$\delta''' == \delta''[\ pdecs := \delta''.pdecs \cup \{kid\}\ ]$$

References: VBasic p. 101; PBasic p. 103; Env p. 11; Store p. 12; $\vdash_{vbasic}$ p. 101; $\Longrightarrow_{vbasic}$ p. 101; $\vdash_{pbasic}$ p. 103; $\Longrightarrow_{pbasic}$ p. 103.

17.4 Package Body

A package body is a secondary unit, containing declarations which implement the operations declared in the visible part of the package specification. A package body optionally contains statements to initialise some of the own-variables of the package.

Syntax Example		A.S. Representation
with j;	*bodyu* $\langle\!|$	$with \mapsto \langle j \rangle$,
package body k **is**		$kid \mapsto k$,
declarations ...		$rens \mapsto \langle\rangle$,
begin		$bdecl \mapsto$ *basic declarations* ...,
statements ...		$ldecl \mapsto$ *later declarations* ...,
end k;		$init \mapsto$ *statements* ... $|\!\rangle$

The declarations of the package are optionally preceded by a list of renames which apply to operations from inherited packages. (The use of renames in SPARK is discussed in more detail in Chapter 16.)

17.4.1 Abstract Syntax

The package name is an identifier; a package body contains a declaration and an initialising statement.

```
┌─ BodyUnit ──────────────────────────────
│ with : seq Id
│ kid : Id
│ rens : seq Ren
│ bdecl : KBasic
│ ldecl : KLater
│ init : Stmt
└─────────────────────────────────────────
```

$$Unit ::= \ldots \mid bodyu\langle\!\langle BodyUnit \rangle\!\rangle$$

17.4.2 Dynamic Semantics

The dynamic semantic effects of a package are twofold: firstly, on first encounter in the abstract syntax, the dynamic environment and store are updated to include the necessary information from the package body's declarations; subsequently, during execution of the main program, the package initialisation statement of the package will be executed. We therefore provide two rules in this section.

A package body is only valid if it has already been declared (by a preceding package specification), it has not been defined before, and all packages that are with'd by the package body have themselves already been declared. We therefore define:

$$
\frac{\begin{array}{l}
\forall\, \delta, \delta', \delta'', \delta''' : Env;\ \sigma, \sigma', \sigma'' : Store;\ BodyUnit \\
\mid \\
\quad kid \in \delta.pdecs \\
\quad kid \notin \operatorname{dom} \delta.pdefs \\
\quad (\forall\, p \in \operatorname{ran} with \bullet p \in \delta.pdecs) \\
\bullet \\
\quad \langle kid \rangle, \delta, \sigma \vdash_{kbasic} bdecl \Longrightarrow_{kbasic} \delta', \sigma' \\
\quad \langle kid \rangle, \delta', \sigma' \vdash_{klater} ldecl \Longrightarrow_{klater} \delta'', \sigma''
\end{array}}{\delta, \sigma \vdash_c bodyu(\theta BodyUnit) \Longrightarrow_c \delta''', \sigma''} \qquad \text{(BodyUD1)}
$$

where

$$\delta''' == \delta''[\ pdefs := \delta''.pdefs \oplus \{kid \mapsto init\}\ ]$$

The initialization statement of the package body cannot contain any calls to user-defined subprograms and variables declared in other packages cannot be read or updated [SR, 7.3]. Consequently, in SPARK the order in which package initializations are performed is irrelevant, provided they all happen prior to execution of the main program. We therefore define the following rules, for use by the main program (see 199).

1. Empty case.

$$
\frac{\begin{array}{l}
\forall\, \delta : Env;\ \sigma : Store;\ pd : \mathbb{P}\, Id \\
\bullet
\end{array}}{\delta, \sigma \vdash_{pkginit} (\langle\rangle, pd) \Longrightarrow_{pkginit} \sigma, pd} \qquad \text{(BodyUD2a)}
$$

2. Recursion cases.

$$
\frac{\begin{array}{l}
\forall\, \delta : Env;\ \sigma, \sigma', \sigma'', \sigma''' : Store; \\
\quad pd, pd', pd'' : \mathbb{P}\, Id;\ h : Id;\ t : \operatorname{seq} Id \\
\mid \\
\quad h \in pd \\
\bullet \\
\quad \langle h \rangle, \delta, \sigma \vdash_s \delta.pdefs\ h \Longrightarrow_s \sigma' \\
\quad \delta, \sigma' \vdash_{pkginit} ((\delta.dict\ \langle h \rangle).withs, pd \setminus \{h\}) \Longrightarrow_{pkginit} \\
\qquad\qquad \sigma'', pd' \\
\quad \delta, \sigma'' \vdash_{pkginit} (t, pd') \Longrightarrow_{pkginit} \sigma''', pd''
\end{array}}{\delta, \sigma \vdash_{pkginit} (\langle h \rangle \frown t, pd) \Longrightarrow_{pkginit} \sigma''', pd''} \qquad \text{(BodyUD2b)}
$$

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ \sigma, \sigma', \sigma'' : Store;\\ \ pd, pd', pd'' : \mathbb{P}\, Id;\ h : Id;\ t : \mathrm{seq}\, Id\\ \mid\\ \quad h \notin pd\\ \bullet\\ \quad \delta, \sigma \vdash_{pkginit} ((\delta.dict\ \langle h\rangle).withs, pd) \Longrightarrow_{pkginit} \sigma', pd'\\ \quad \delta, \sigma' \vdash_{pkginit} (t, pd') \Longrightarrow_{pkginit} \sigma'', pd''\end{array}}{\delta, \sigma \vdash_{pkginit} (\langle h\rangle \frown t, pd) \Longrightarrow_{pkginit} \sigma'', pd''} \quad \text{(BodyUD2c)}$$

Note that the additional set of identifiers argument allows us to avoid re-execution of a package initialization which is with'd more than once.

References: KBasic p. 107; *KLater p. 111*; *Stmt p. 121*; *Env p. 11*; *Store p. 12*; $\vdash_{kbasic}$ p. 107; $\Longrightarrow_{kbasic}$ p. 107; $\vdash_{klater}$ p. 111; $\Longrightarrow_{klater}$ p. 111.

17.5 Subunit

A subprogram definition or a package body, which is a secondary unit, can be given in a separate file, provided it has been declared to be separate.

Syntax Example	A.S. Representation
with K,L;	$subu \langle\!| \; withs \mapsto \langle K, L \rangle,$
separate (Parent)	$\quad parent \mapsto \langle Parent \rangle,$
declaration ...	$\quad sdecl \mapsto \ldots \; |\!\rangle$

17.5.1 Abstract Syntax

The parent is represented as a list of identifiers, rather than *IdDot*. This is required because the parent can be a subprogram, or an embedded package, so that the full name may have any number of identifiers. This is only place in SPARK where full names are used.

SubuCUnit

$$
\begin{aligned}
&withs : \mathrm{seq}\, Id \\
&parent : \mathrm{seq}\, Id \\
&sdecl : Decl
\end{aligned}
$$

$$CUnit ::= \ldots \mid subu\langle\!\langle SubUCUnit \rangle\!\rangle$$

17.5.2 Dynamic Semantics

The effect of a separate subprogram definition for the dynamic semantics is to update the dynamic environment and store to reflect its declarations and body.

17.6 Main Program

In SPARK, subprograms cannot generally be library units (though they can be subunits). However, a complete SPARK program must contain one procedure subprogram which is the entry point of the program and is a library unit. It is distinguished by the **main_program** annotation.

Syntax Example	A.S. Representation
with k;	$main \langle\!| \ withs \mapsto \langle k \rangle,$
--# **inherit** k;	$inherit \mapsto \langle k \rangle,$
--# **main_program**	$mid \mapsto SystemW,$
procedure SystemW	$globals \mapsto \langle dot(k,u), dots(k,v) \rangle,$
--# **global** k.u, k.v ;	$derives \mapsto \langle \langle\!| \ export \mapsto dot(k,u),$
--# **derives** k.u **from** k.v ;	$imports \mapsto \langle dot(k,v) \rangle \ |\!\rangle \rangle,$
is	$rens \mapsto \langle \ldots \rangle,$
renames ...	$dpart \mapsto \ldots,$
declarations ...	$spart \mapsto \ldots \ |\!\rangle$
begin	
statements ...	
end SystemW ;	

17.6.1 Abstract Syntax

The main program has a with list and inherit annotation, and also the components of a parameterless local procedure. A list of renames may be used immediately before the local declarations.

MainCUnit

$withs : \mathrm{seq}\, Id$
$inherit : \mathrm{seq}\, Id$
$mid : Id$
$globals : GlobalAnnot$
$derives : \mathrm{seq}\, DerivesAnnot$
$renames : \mathrm{seq}\, Ren$
$bdecl : SBasic$
$ldecl : SLater$
$spart : Stmt$

$$CUnit ::= \ldots \mid main \langle\!\langle MainCUnit \rangle\!\rangle$$

17.6.2 Dynamic Semantics

The goal of this dynamic semantics is to define execution of the (unique) main program of a SPARK text. This execution can only occur if:

1. the dynamic environment is valid for this main program and its set of library units; and
2. all of the packages with'd by the main program (and those with'd by these packages, and so on to a finite transitive closure) have been both declared (package specification) and defined (package body).

We assume that the store has been constructed in parallel with the dynamic environment, as in the relevant rules of this dynamic semantics.

The evaluation of the main program proceeds by:

1. elaborating its basic local declarations;
2. elaborating its later local declarations;
3. carrying out the package initialisations for this program, then
4. evaluating its statement-part.

$$\forall\, \delta, \delta', \delta'' : \mathit{Env};\ \sigma, \sigma', \sigma'', \sigma''', \sigma'''' : \mathit{Store};$$
$$\mathit{remainder} : \mathbb{P}\, \mathit{Id};\ \mathit{MainCUnit}$$
$$|$$
$$(\forall\, p \in \operatorname{ran} \mathit{withs} \bullet p \in \operatorname{dom} \delta.\mathit{pdefs})$$
$$\delta.\mathit{pdecs} = \operatorname{dom} \delta.\mathit{pdefs}$$
$$\bullet$$
$$\frac{\begin{array}{l}\langle \mathit{mid} \rangle, \delta, \sigma \vdash_{sbasic} \mathit{bdecl} \Longrightarrow_{sbasic} \delta', \sigma' \\ \langle \mathit{mid} \rangle, \delta', \sigma' \vdash_{slater} \mathit{ldecl} \Longrightarrow_{slater} \delta'', \sigma'' \\ \delta'', \sigma'' \vdash_{pkginit} (\mathit{withs}, \delta''.\mathit{pdecs}) \Longrightarrow_{pkginit} \sigma''', \mathit{remainder} \\ \langle \mathit{mid} \rangle, \delta'', \sigma''' \vdash_{s} \mathit{spart} \Longrightarrow_{s} \sigma''''\end{array}}{\delta, \sigma \vdash_{main} \mathit{main}(\theta \mathit{MainCUnit}) \Longrightarrow_{main} \sigma''''} \qquad \text{(MainD)}$$

Note: The "remainder" set of package identifiers will be empty provided there are no superfluous packages in the program (i.e. which are not with'd by the main program or by any of its with'd packages, and so on). We do not care if such superflous packages are present, as they can have no effect on the dynamic semantics of the main program.

References: SBasic p. 105; SLater p. 109; Stmt p. 121; Env p. 11; Store p. 12; $\vdash_{sbasic}$ p. 105; $\Longrightarrow_{sbasic}$ p. 105; $\vdash_{slater}$ p. 109; $\Longrightarrow_{slater}$ p. 109; $\vdash_{pkginit}$ p. 196; $\Longrightarrow_{pkginit}$ p. 196.

17.7 SPARK Program

A SPARK program consists of one or more compilation units.

17.7.1 Abstract Syntax

Since the order of compilation units is not determined by the syntax, we represent a SPARK program as a set of compilation units.

$$SPARK == \mathbb{P}\ Unit$$

17.7.2 Dynamic Semantics

As noted in the Static Semantics, a main program must be present. The dynamic semantics of a SPARK program is therefore the dynamic semantics of its main program (see page 199).

Appendix A

The Predefined Environment

Refer to the Static Semantics document for a description of the predefined environment of SPARK.

Appendix B

Auxiliary Functions

This chapter contains the definitions of various functions and predicates used elsewhere. The definitions are organised into two categories:

B.1 Inherited Static Semantic Functions

This section lists below those functions, sections and objects defined in the companion volume on the Static Semantics of SPARK and which are referenced in the text of this document. The list below refers the reader to the definition of the relevant item in the Static Semantics.

SS Sect/Chapt	Function Name	SS Page
B.4.2	$ancestorof_\delta$	p.281
B.4.3	$is\_arr\_tmark_\delta$	p.283
Ch.4	SubDef	p.41

B.2 Dynamic Semantics Functions

B.2.1 Apply_Uop

This function is used in the definition of the dynamic semantics of unary operator expressions; it is defined below.

$$
\begin{array}{|l}
apply\_uop : Uop \times Val \rightarrow\!\!\!\!\!+ \; Val \\
\hline
\forall\, x, v : \mathbb{Z} \\
\bullet \\
\qquad apply\_uop(uplus, intval\ x) = intval\ x \\
\qquad v = -x \Rightarrow apply\_uop(uminus, intval\ x) = intval\ v \\
\qquad v \geq 0 \Rightarrow apply\_uop(abs, intval\ v) = intval\ v \\
\qquad (x < 0 \land v = -x) \Rightarrow apply\_uop(abs, intval\ x) = intval\ v \\
apply\_uop(not, enumval\ true) = enumval\ false \\
apply\_uop(not, enumval\ false) = enumval\ true \\
\forall\, a : Array\_Value \\
\bullet \\
\qquad apply\_uop(not, arrval(a)) = arrval(array\_not(a))
\end{array}
$$

This defines the unary arithmetic operators for the integers; again we note that our model of the reals as a given set does not permit us to define the corresponding effect on real arguments in this document.

Ada (and SPARK) allow the unary operator *not* to be applied to a one-dimensional array of booleans [LRM, 3.6.2(12)]. The function *array_not* used in the above definition to reflect this can be defined by:

$$
\begin{array}{|l}
array\_not : Array\_Value \rightarrow\!\!\!\!\!+ \; Array\_Value \\
\hline
\forall\, a, b : Array\_Value \\
\mid \\
\qquad a.lo = b.lo \\
\qquad a.hi = b.hi \\
\qquad b.arr = (\lambda\, i : a.lo \mathrel{..} a.hi \bullet apply\_uop(not, a.arr(i))) \\
\bullet \\
\qquad array\_not(a) = b
\end{array}
$$

Notes:

1. The resulting array has the same index bounds as the array which is being negated [LRM, 4.5.6(3)].
2. This definition also requires that all elements of the array should be defined, in such a way that the *not* operator can be applied to them via *apply_uop*.

3. The use of *apply_uop* to derive the contents of *b.arr* in the above rule would appear to allow logical negation of arrays of arrays of booleans, etc., but this option should be prevented by the static semantic checks performed.

B.2.2 Apply_Bop

To define *apply_bop*, we first define the boolean set of enumeration literals:

$$Bool == \{b : Id \mid b = true \lor b = false \bullet b\}$$

We now define *apply_bop* by cases:

$$apply\_bop : Bop \times Val \times Val \rightarrowtail Val$$

$\forall\, v1, v2 : Val$

$\bullet$

$$\begin{aligned}
&apply_bop(eq, v1, v2) = enumval\ bop_equal(v1, v2)\\
&apply_bop(noteq, v1, v2) = enumval\ bop_not_equal(v1, v2)\\
&apply_bop(lt, v1, v2) = enumval\ bop_less_than(v1, v2)\\
&apply_bop(lte, v1, v2) = enumval\ bop_less_or_equal(v1, v2)\\
&apply_bop(gt, v1, v2) = enumval\ bop_less_than(v2, v1)\\
&apply_bop(gte, v1, v2) = enumval\ bop_less_or_equal(v2, v1)
\end{aligned}$$

$\forall\, b1, b2 : Bool$

$\bullet$

$$\begin{aligned}
&apply_bop(and, enumval\ b1, enumval\ b2) = enumval\ bop_and(b1, b2)\\
&apply_bop(or, enumval\ b1, enumval\ b2) = enumval\ bop_or(b1, b2)\\
&apply_bop(xor, enumval\ b1, enumval\ b2) = enumval\ bop_xor(b1, b2)
\end{aligned}$$

$\forall\, v1, v2, v : Val;\ n : \mathbb{Z}$

$|$

$$v = intval\ n$$

$\bullet$

$$\begin{aligned}
&bop_plus(v1, v2) = v \Rightarrow apply_bop(plus, v1, v2) = v\\
&bop_minus(v1, v2) = v \Rightarrow apply_bop(minus, v1, v2) = v\\
&bop_times(v1, v2) = v \Rightarrow apply_bop(mul, v1, v2) = v\\
&bop_divide(v1, v2) = v \Rightarrow apply_bop(div, v1, v2) = v\\
&bop_mod(v1, v2) = v \Rightarrow apply_bop(mod, v1, v2) = v\\
&bop_rem(v1, v2) = v \Rightarrow apply_bop(rem, v1, v2) = v\\
&bop_power(v1, v2) = v \Rightarrow apply_bop(power, v1, v2) = v
\end{aligned}$$

$\forall\, av1, av2 : Array\_Value$

$\bullet$

$$\begin{aligned}
&apply_bop(and, arrval(av1), arrval(av2)) = arrval(array_and(av1, av2))\\
&apply_bop(or, arrval(av1), arrval(av2)) = arrval(array_or(av1, av2))\\
&apply_bop(xor, arrval(av1), arrval(av2)) = arrval(array_xor(av1, av2))
\end{aligned}$$

We define each of the above functions in the following subsubsections.

Equality

We define the partial function *bop_equal* used for equality comparison by:

$$
\begin{array}{|l}
bop\_equal : Val \times Val \rightarrowtail Bool \\
\hline
\forall\, x, y : Val \\
| \\
\quad x \neq undefined \wedge y \neq undefined \\
\bullet \\
\quad values\_equal(x, y) \Rightarrow bop\_equal(x, y) = true \\
\quad \neg\, values\_equal(x, y) \Rightarrow bop\_equal(x, y) = false
\end{array}
$$

In the above, two (at least partially defined) values are equal if they satisfy the following relation:

$$
\begin{array}{|l}
values\_equal : Val \leftrightarrow Val \\
\hline
\forall\, x, y : Val \\
| \\
\quad x \neq undefined \wedge y \neq undefined \\
\bullet \\
\quad (x, x) \in values\_equal \\
\quad \exists\, xa, ya : Array\_Value \\
\quad | \\
\quad\quad x = arrval(xa) \\
\quad\quad y = arrval(ya) \\
\quad \bullet \\
\quad\quad array\_equal(xa, ya) \Rightarrow (x, y) \in values\_equal
\end{array}
$$

This definition ignores real number equality and the complications caused by the accuracies involved. (In particular, equality (and other comparisons) of real values is complicated: it involves consideration of the model intervals and may in some cases be *"any possible value (that is, either TRUE or FALSE)"* [Ada LRM, 4.5.7].)

In addition, the special case in the above for arrays is to cater for comparison for equality of arrays, as required by [LRM, 4.5.2], particularly paragraphs 5 and 7. This comparison allows "index-shifting", which complicates the definition of equality. We define array equality by

$$
\begin{array}{|l}
array\_equal : Array\_Value \leftrightarrow Array\_Value \\
\hline
\forall\, x, y : Array\_Value \\
\bullet \\
\quad (x.hi - x.lo = y.hi - y.lo \wedge \\
\quad \forall\, i : x.lo \,..\, x.hi \bullet \\
\quad\quad bop\_equal(x.arr(i), y.arr(i - x.lo + y.lo)) = true) \Leftrightarrow \\
\quad (x, y) \in array\_equal
\end{array}
$$

Notice that this uses *bop_equal* in its definition; this is because [LRM, 4.5.2(5)] states the requirement that "...the values of matching components are equal, as given by the predefined equality operator for the component type."

Inequality

For Ada [LRM, 4.5.2(2)], and in consequence for SPARK, the inequality operator gives the complementary result to the equality operator; thus, we need merely define

$$
\begin{array}{|l}
bop\_not\_equal : Val \times Val \nrightarrow Bool \\
\hline
\forall x, y : VALUE \\
| \\
\quad x \neq undefined \wedge y \neq undefined \\
\bullet \\
\quad bop\_equal(x, y) = true \Rightarrow bop\_not\_equal(x, y) = false \\
\quad bop\_equal(x, y) = false \Rightarrow bop\_not\_equal(x, y) = true
\end{array}
$$

Ordering

Ada allows scalar types to be compared for order in the usual way, e.g. $2 < 3$, $3.14 > 2.72$. In addition, the four ordering operators *lt*, *lte*, *gt* and *gte* can be applied in Ada to one dimensional arrays whose components are of a discrete type [LRM 4.5.2(1)] to give a *lexicographic* ordering of such objects. SPARK further restricts the applicability of lexicographic ordering to one-dimensional arrays of base type *STRING* [SRM, 3.6.3], though this restriction is enforced purely via the static semantics (since, in the modelling of values in this document, all one-dimensional arrays of discrete components are modelled as arrays of integers).

We first define the set of array values to which lexicographic orderings is applicable:

$$
\begin{array}{|l}
lexable\_arrays : \mathbb{P}\ Array\_Value \\
\hline
lexable\_arrays = \{Array\_Value \mid \operatorname{ran} arr \subseteq \operatorname{ran} int\}
\end{array}
$$

(Note that in consequence of the above, the arrays which can be ordered are restricted to those whose elements are all fully defined as well: none can be equal to the *undefined* value.) Given such a set, we now define:

$$
\begin{array}{|l}
bop\_less\_than : Val \times Val \nrightarrow Bool \\
\hline
\forall x, y : \mathbb{Z} \\
\bullet \\
\quad x < y \Leftrightarrow bop\_less\_than(intval\ x, intval\ y) = true \\
\quad x \geq y \Leftrightarrow bop\_less\_than(intval\ x, intval\ y) = false \\
\\
\forall x, y : lexable\_arrays \\
\quad bop\_less\_than(arrval(x), arrval(y)) = array\_less\_than(x, y)
\end{array}
$$

A SPARK-specific restriction of relevance here is that all SPARK arrays are non-empty, because of the constraints imposed by [SRM, 3.5]. To compare two arrays for order, we first convert each *Array_Value* into a sequence, using

$$
\begin{array}{|l}
mkseq : Array\_Value \rightarrow \mathrm{seq}\,\mathbb{Z} \\
\hline
\forall\, x, y : \mathbb{Z};\ a : Array\_Value \\
\mid \\
\quad a.lo = x \\
\quad a.hi = x \\
\quad a.arr(x) = intval\ y \\
\bullet \\
\quad mkseq(a) = \langle y \rangle \\
\forall\, x, y : \mathbb{Z};\ a, a' : Array\_Value \\
\mid \\
\quad a.lo = x \\
\quad a.lo < a.hi \\
\quad a.arr(x) = intval\ y \\
\quad a'.lo = a.lo + 1 \\
\quad a'.hi = a.hi \\
\quad a'.arr = (a.lo + 1 \mathinner{\ldotp\ldotp} a.hi) \lhd a.arr \\
\bullet \\
\quad mkseq(a) = \langle y \rangle \frown mkseq(a')
\end{array}
$$

so that we can then define a simple relation

$$
\begin{array}{|l}
\_lex\_less\_ : \mathrm{seq}\,\mathbb{Z} \leftrightarrow \mathrm{seq}\,\mathbb{Z} \\
\hline
\forall\, s, t : \mathrm{seq}\,\mathbb{Z} \bullet \\
\quad s\ lex\_less\ t \Leftrightarrow \\
\quad\quad s \subset t\ \vee \\
\quad\quad (\exists\, r : \mathrm{seq}\,\mathbb{Z};\ x, y : \mathbb{Z} \bullet \\
\quad\quad\quad r \frown \langle x \rangle \subseteq s \wedge r \frown \langle y \rangle \subseteq t \wedge x < y)
\end{array}
$$

We can now define the *array_less_than* function used (for Ada's *lexicographic* ordering [LRM, 4.5.2(9)]) in the above by:

$$
\begin{array}{|l}
array\_less\_than : lexable\_array \times lexable\_array \rightarrow Bool \\
\hline
\forall\, x, y : lexable\_array \\
\bullet \\
\quad mkseq(x)\ lex\_less\ mkseq(y) \Rightarrow array\_less\_than(x, y) = true \\
\quad (\neg\ mkseq(x)\ lex\_less\ mkseq(y)) \Rightarrow array\_less\_than(x, y) = false
\end{array}
$$

Given the above, we have now defined *bop_less_than* and may now complete our definitions of the auxiliary functions needed for ordering with

$$
\begin{array}{|l}
bop\_less\_or\_equal : Val \times Val \nrightarrow Bool \\
\hline
\forall\, x, y : VALUE \\
\bullet \\
\qquad bop\_less\_or\_equal(x, y) = \\
\qquad\qquad bop\_or(bop\_equal(x, y), bop\_less\_than(x, y))
\end{array}
$$

Arithmetic

SPARK's integer arithmetic is relatively straightforward. However, given that real numbers are regarded simply as a given set in this document, the following definitions do not treat arithmetic involving real numbers.

$$
\begin{array}{|l}
bop\_plus : Val \times Val \nrightarrow Val \\
\hline
\forall\, x, y : \mathbb{Z} \\
\bullet \\
\qquad bop\_plus(intval\ x, intval\ y) = intval\ (x + y)
\end{array}
$$

$$
\begin{array}{|l}
bop\_minus : Val \times Val \nrightarrow Val \\
\hline
\forall\, x, y : \mathbb{Z} \\
\bullet \\
\qquad bop\_minus(intval\ x, intval\ y) = intval\ (x - y)
\end{array}
$$

$$
\begin{array}{|l}
bop\_times : Val \times Val \nrightarrow Val \\
\hline
\forall\, x, y : \mathbb{Z} \\
\bullet \\
\qquad bop\_times(intval\ x, intval\ y) = intval\ (x * y)
\end{array}
$$

$$
\begin{array}{|l}
bop\_divide : Val \times Val \nrightarrow Val \\
\hline
\forall\, x, y : \mathbb{Z} \\
\bullet \\
\qquad y \neq 0 \Rightarrow bop\_divide(intval\ x, intval\ y) = intval\ (x\ div\ y)
\end{array}
$$

In the above, and in all other occurrences in this document, $X\ div\ Y$ is defined to be $s * i$, where i is the integer part of $\mid X \mid / \mid Y \mid$ ($\mid X \mid$ being the absolute value of X) and

s being 1 if X and Y have the same sign, or -1 if X and Y have opposite signs. This is different from Z's div operator, which appears to round in a different way.

$$
\begin{array}{l}
bop\_mod : Val \times Val \nrightarrow Val \\
\hline
\forall x, y : \mathbb{Z} \\
\bullet \\
\quad (x > 0 \land y > 0 \lor x < 0 \land y < 0) \Rightarrow \\
\qquad bop\_mod(intval\ x, intval\ y) = intval\ (x - (x\ div\ y) * y) \\
\quad (x > 0 \land y < 0 \lor x < 0 \land y > 0) \Rightarrow \\
\qquad ((x = x\ div\ y * y \Rightarrow \\
\qquad\quad bop\_mod(intval\ x, intval\ y) = intval\ (x - x\ div\ y * y)) \land \\
\qquad (x \neq x\ div\ y * y \Rightarrow \\
\qquad\quad bop\_mod(intval\ x, intval\ y) = intval\ (x - (x\ div\ y + 1) * y)))
\end{array}
$$

$$
\begin{array}{l}
bop\_rem : Val \times Val \nrightarrow Val \\
\hline
\forall x, y : \mathbb{Z} \\
\bullet \\
\quad y \neq 0 \Rightarrow bop\_rem(intval\ x, intval\ y) = intval\ (x - (x\ div\ y) * y)
\end{array}
$$

The above definitions of *bop_div*, *bop_mod* and *bop_rem* are somewhat complicated, but have been compared (via animation in Prolog) with the table of Ada's intended results in [LRM, 4.5.5(16)].

$$
\begin{array}{l}
bop\_power : Val \times Val \nrightarrow Val \\
\hline
\forall x : \mathbb{Z} \\
\bullet \\
\quad bop\_power(intval\ x, intval\ 0) = intval\ 1 \\
\quad \forall y : \mathbb{Z} \bullet y > 0 \Rightarrow bop\_power(intval\ x, intval\ y) = intval\ (x^y)
\end{array}
$$

N.B. The Ada LRM implies X ** 0 is 1 regardless of X; in particular, 0 ** 0 = 1 !!

Logical Operations

The three binary logical operator modelling functions are easily defined by:

$$
\begin{array}{l}
bop\_and : Bool \times Bool \rightarrow Bool \\
\hline
\quad bop\_and(false, false) = false \\
\quad bop\_and(false, true) = false \\
\quad bop\_and(true, false) = false \\
\quad bop\_and(true, true) = true
\end{array}
$$

$$bop\_or : Bool \times Bool \rightarrow Bool$$

$$\begin{aligned}
&bop_or(false, false) = false\\
&bop_or(false, true) = true\\
&bop_or(true, false) = true\\
&bop_or(true, true) = true
\end{aligned}$$

$$bop\_xor : Bool \times Bool \rightarrow Bool$$

$$\begin{aligned}
&bop_xor(false, false) = false\\
&bop_xor(false, true) = true\\
&bop_xor(true, false) = true\\
&bop_xor(true, true) = false
\end{aligned}$$

For logical operations on one-dimensional arrays of booleans involving the binary operators *and*, *or* and *xor* – which are allowed in Ada [LRM, 3.6.2(12)] – we define

$$array\_and : Array\_Value \times Array\_Value \nrightarrow Array\_Value$$

$$\begin{aligned}
&\forall\, a1, a2, a3 : Array_Value\\
&|\\
&\quad a1.lo = a3.lo\\
&\quad a1.hi = a3.hi\\
&\quad a1.hi - a1.lo = a2.hi - a2.lo\\
&\quad a3.arr = (\lambda\, i : a1.lo \mathinner{\ldotp\ldotp} a1.hi \bullet\\
&\qquad apply_bop(and, a1.arr(i), a2.arr(i - a1.lo + a2.lo)))\\
&\bullet\\
&\quad array_and(a1, a2) = a3
\end{aligned}$$

$$array\_or : Array\_Value \times Array\_Value \nrightarrow Array\_Value$$

$$\begin{aligned}
&\forall\, a1, a2, a3 : Array_Value\\
&|\\
&\quad a1.lo = a3.lo\\
&\quad a1.hi = a3.hi\\
&\quad a1.hi - a1.lo = a2.hi - a2.lo\\
&\quad a3.arr = (\lambda\, i : a1.lo \mathinner{\ldotp\ldotp} a1.hi \bullet\\
&\qquad apply_bop(or, a1.arr(i), a2.arr(i - a1.lo + a2.lo)))\\
&\bullet\\
&\quad array_or(a1, a2) = a3
\end{aligned}$$

$$
\begin{array}{|l}
array\_xor : Array\_Value \times Array\_Value \nrightarrow Array\_Value \\
\hline
\forall\, a1, a2, a3 : Array\_Value \\
| \\
\quad a1.lo = a3.lo \\
\quad a1.hi = a3.hi \\
\quad a1.hi - a1.lo = a2.hi - a2.lo \\
\quad a3.arr = (\lambda\, i : a1.lo \mathinner{..} a1.hi \bullet \\
\qquad apply\_bop(xor, a1.arr(i), a2.arr(i - a1.lo + a2.lo))) \\
\bullet \\
\quad array\_xor(a1, a2) = a3
\end{array}
$$

Notes:

1. The two array values being *and*'ed, *or*'ed or *xor*'ed do not need to have identical index ranges, merely an equally large index range; the index range of the resulting array $a3$ adopts the index range of the left-hand operand [LRM, 4.5.1(3)].

2. The use of *apply_bop* to derive the contents of $a3.arr$ in each of the above three rules would appear to allow logical operations on arrays of arrays of booleans, etc., but this option should be prevented by the static semantic checks performed.

3. In each of the above three rules, the requirement that each array has an equally large index range (i.e. effectively, the same *'LENGTH* attribute) is a dynamic well-formedness check for logical operations on arrays; meeting this constraint precludes the possibility of the exception `CONSTRAINT_ERROR` arising during execution [LRM, 4.5.1(3)].

B.2.3 Lookup_Element

This is a partial function, defined only for array indexing. SPARK does not tolerate the raising of exceptions; therefore, all indices used must be within the index range of the array index.

We define:

$$
\begin{array}{|l}
lookup\_element : (Val \times \mathrm{seq}_1\ Val) \nrightarrow Val \\
\hline
\forall\, av, v : Val;\ a : Array\_Value;\ i : \mathbb{Z} \bullet \\
\quad (av = arrval\ a \land v = intval\ i \land a.lo \le i \le l.hi) \\
\qquad \Rightarrow lookup\_element(av, \langle v \rangle) = a.arr\ i \\
\forall\, av, v : Val;\ vs : \mathrm{seq}_1\ Val;\ a : Array\_Value;\ i : \mathbb{Z} \bullet \\
\quad (av = arrval\ a \land v = intval\ i \land a.lo \le i \le l.hi) \\
\qquad \Rightarrow lookup\_element(av, \langle v \rangle \frown vs) = \\
\qquad\quad lookup\_element(a.arr\ i, vs)
\end{array}
$$

B.2.4 Sufficiently_Close

This function is implementation-dependent, for comparison of real numbers.

B.2.5 Get_Typ_Con

This partial function may be defined by:

$$
\begin{array}{|l}
get\_typ\_con_{Env} : (IdDot \times (\mathrm{seq}\, Id)) \nrightarrow TypCon \\
\hline
\forall\, \delta : Env;\ \ c, c' : \mathrm{seq}\, Id;\ \ t : Id \bullet \\
\qquad get\_id\_ctx(c, \delta, t) = (c', typeI) \Rightarrow \\
\qquad\qquad get\_typ\_con_{\delta}(id\ \ t, c) = ((\delta.dict\ \ c').type\ \ t) \\
\forall\, \delta : Env;\ \ c, c', c'' : \mathrm{seq}\, Id;\ \ k, t : Id \bullet \\
\qquad (get\_id\_ctx(c, \delta, k) = (c', pkgI) \wedge \\
\qquad\ \ get\_id\_ctx(c', \delta, t) = (c'', typeI)) \Rightarrow \\
\qquad\qquad get\_typ\_con_{\delta}(dot(k, t), c) = ((\delta.dict\ \ c'').type\ \ t)
\end{array}
$$

It is only validly called when the context c is valid w.r.t. the environment δ and t is an identifier standing for an appropriate type. The corresponding type construction is returned.

B.2.6 Rec_Field_Seq

This partial function may be defined by:

$$
\begin{array}{|l}
rec\_field\_seq_{Env} : (IdDot \times (\mathrm{seq}\, Id)) \nrightarrow \mathrm{iseq}\, Id \\
\hline
\forall\, \delta : Env;\ \ c : \mathrm{seq}\, Id;\ \ t : IdDot;\ \ tc : TypCon \bullet \\
\qquad (get\_typ\_con_{\delta}(t, c) = tc \wedge \\
\qquad\ \ tc \in \mathrm{ran}\ recT) \Rightarrow \\
\qquad\qquad rec\_field\_seq_{\delta}(t, c) = (recT^{\sim} tc).fields
\end{array}
$$

It returns the sequence of record field name identifiers of the typemark t in context c and environment δ.

B.2.7 TypRange

This function may be defined by:

$$
\begin{array}{|l}
typrange : ((\mathrm{seq}\, Id) \times Env \times IdDot) \nrightarrow \mathbb{P}\ Val \\
\hline
\forall\, \delta : Env;\ c : \mathrm{seq}\, Id;\ t : Id;\ lo, hi : Val \bullet \\
\qquad (lo = get\_type\_first(c, \delta, t) \wedge \\
\qquad\ hi = get\_type\_last(c, \delta, t)) \Rightarrow \\
\qquad\qquad typrange(c, \delta, id\ t) = lo\ {}_{V}..{}_{\delta}\ hi \\
\forall\, \delta : Env;\ c, c', c'' : \mathrm{seq}\, Id;\ t : Id;\ lo, hi : Val \bullet \\
\qquad (get\_id\_ctx(c, \delta, k) = (c', pkgI) \wedge \\
\qquad\ get\_id\_ctx(c', \delta, t) = (c'', typeI) \wedge \\
\qquad\ lo = get\_type\_first(c'', \delta, t) \wedge \\
\qquad\ hi = get\_type\_last(c'', \delta, t)) \Rightarrow \\
\qquad\qquad typrange(c, \delta, dot(k, t)) = lo\ {}_{V}..{}_{\delta}\ hi
\end{array}
$$

The ${}_{V}..{}_{\delta}$ operator is defined in the companion Static Semantics document.

B.2.8 TypBounds

This function may be defined by:

$$
\begin{array}{|l}
typbounds_{Env} : ((\mathrm{seq}\, Id) \times IdDot) \nrightarrow (\mathbb{Z} \times \mathbb{Z}) \\
\hline
\forall\, \delta : Env;\ c : \mathrm{seq}\, Id;\ t : Id;\ lo, hi : \mathbb{Z} \bullet \\
\qquad (t \in \mathrm{dom}(\delta.dict\ \ c).type \wedge \\
\qquad\ ((\delta.dict\ \ c).type\ \ t) \in \mathrm{ran}\ arrT \wedge \\
\qquad\ lo = intval^{\sim} get\_type\_first(c, \delta, \\
\qquad\qquad (arrT^{\sim}((\delta.dict\ \ c).type\ \ t)).indexes(1)) \wedge \\
\qquad\ hi = intval^{\sim} get\_type\_last(c, \delta, \\
\qquad\qquad (arrT^{\sim}((\delta.dict\ \ c).type\ \ t)).indexes(1))) \Rightarrow \\
\qquad\qquad\qquad typbounds_{\delta}(c, id\ \ t) = (lo, hi) \\
\forall\, \delta : Env;\ c, c', c'' : \mathrm{seq}\, Id;\ k, t : Id;\ lo, hi : \mathbb{Z} \bullet \\
\qquad (get\_id\_ctx(c, \delta, k) = (c', pkgI) \wedge \\
\qquad\ get\_id\_ctx(c', \delta, t) = (c'', typeI) \wedge \\
\qquad (t \in \mathrm{dom}(\delta.dict\ \ c'').type \wedge \\
\qquad\ ((\delta.dict\ \ c'').type\ \ t) \in \mathrm{ran}\ arrT \wedge \\
\qquad\ lo = intval^{\sim} get\_type\_first(c'', \delta, \\
\qquad\qquad (arrT^{\sim}((\delta.dict\ \ c'').type\ \ t)).indexes(1)) \wedge \\
\qquad\ hi = intval^{\sim} get\_type\_last(c'', \delta, \\
\qquad\qquad (arrT^{\sim}((\delta.dict\ \ c'').type\ \ t)).indexes(1))) \Rightarrow \\
\qquad\qquad\qquad typbounds_{\delta}(c, dot(k, t)) = (lo, hi)
\end{array}
$$

B.2.9 Get_FullName

This function may be defined by:

$$
\begin{array}{|l}
get\_fullname : ((\operatorname{seq} Id) \times Env \times Id) \rightarrow\!\!\!\!\!+ \operatorname{seq} Id \\
\hline
\forall\, \delta : Env;\ \ c : \operatorname{seq} Id;\ \ i : Id \bullet \\
\qquad get\_id\_ctx(c, \delta, i) = (c', varI) \Rightarrow \\
\qquad\qquad get\_fullname(c, \delta, i) = c' \frown \langle i \rangle
\end{array}
$$

The above function is onlu defined for variables.

References: Val p. 9; *arrT p. 15 Env p. 11*; *IdDot p. 8*; *pkgI p. 37*; *typeI p. 37*; *varI p. 37*; *Array_Value* p. 9.

Appendix C

Non-Standard Notation

This chapter defines some notational extensions used in this document, which are not part of "standard" Z.

C.1 Schema Update

Name

[:=] — Schema update

Syntax

$$Expression ::= Expression\ [\ Ident := Expression\ ]$$

Type rules

In the expression $E\ [x := y]$, the sub-expression E must have a schema type of the form $\langle\!|\ x_1 : t_1;\ \ldots;\ x_n : t_n\ |\!\rangle$ and the identifier x must be identical with one of the component names x_i, for some i with $1 \leq i \leq n$. The sub-expression y must be of the corresponding type t_i. The type of the expression is the schema type $\langle\!|\ x_1 : t_1;\ \ldots;\ x_n : t_n\ |\!\rangle$.

Description

This notation is used to create a new binding by giving a new value to one of the components of an existing binding. If b is a binding $\langle x_1 \Rrightarrow v_1;\ \ldots;\ x_i \Rrightarrow v_i;\ \ldots;\ x_n \Rrightarrow v_n \rangle$ and x is identical with x_i then the value of $b\ [x := w]$ is $\langle x_1 \Rrightarrow v_1;\ \ldots;\ x_i \Rrightarrow w;\ \ldots;\ x_n \Rrightarrow v_n \rangle$.

Laws

If b has type $\langle\!|\ x_1 : t_1;\ \ldots;\ x_n : t_n\ |\!\rangle$, then

$$b\ [x_i := v].x_i = v$$

$$b\ [x_i := v].x_j = b.x_j$$

$$b\ [x_i := v]\ [x_i := w] = b\ [x_i := w]$$

where $1 \leq i \leq n$ and $1 \leq j \leq n$ and $i \neq j$.

Extended Form

The following equivalence defines an extended form which is used to express multiple updates to a binding.

$$b\ [x_i := v, x_j := w] \equiv b\ [x_i := v]\ [x_j := w]$$

where b has type $\langle\!|\ x_1 : t_1;\ \ldots;\ x_n : t_n\ |\!\rangle$, and $1 \leq i \leq n$ and $1 \leq j \leq n$ and $i \neq j$.

Index

Bibliography

[AARM] *The Annotated Ada Reference Manual* ANSI-MIL-STD-1815A-1983, Karl A. Nyberg (Editor), 1989.

[SR] *SPARK — The SPADE Ada Kernel* Edition 3.1, B.A. Carré, T.J. Jennings, F.J. Maclennan, P.F. Farrow and J.R. Garnsworthy. Program Validation Ltd. May 1992.