Formal Semantics of SPARK

Static Semantics

Program Validation Ltd.

29th. October 1994

PVL/SPARK_DEFN/STATIC/V1.3

Author:	William Marsh
Authorisation:	
Type:	Deliverable
Status:	Approved
Circulation:	Restricted

Document Purpose

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

Document History

Version	Date	Who	Why
1.0	08.02.93	DWRM	First delivered version
1.1	05.03.93	DWRM	First revision
1.2	03.03.94	DWRM	Version for external review
1.3	29.10.94	ION	Version with Peer Review changes

Formal Semantics of SPARK

Static Semantics

Program Validation Ltd.

29th. October 1994
PVL/SPARK_DEFN/STATIC/V1.3

Contents

Chapter 1

Introduction

This document is a formal specification of the static semantics of the SPARK subset of Ada. We assume some familiarity with both Ada and SPARK. This Chapter introduces some of the significant concepts on which the rest of the specification is based.

Abstract Syntax The SPARK text, whose well-formation is specified, is presented as an Abstract Syntax term. This document does not describe the Concrete Syntax used to represent Abstract Syntax terms. The Abstract Syntax has the following main categories:

Category	Description
SubDef	Subtype definitions
TypDef	Type definitions
Name	Names
Exp	Expressions
BDecl	Basic Declarations
PDecl	Private Declarations
Stmt	Statements
SDecl	Subprogram Declarations
KDecl	Embedded Package Declarations
FDecl	Subprogram Definitions
YDecl	Subprogram and Package Bodies
Ren	Renaming declarations
Unit	Compilation units

The structure of the document follows the Abstract Syntax. In most cases, the rules for each category are described in a separate chapter. However, SPARK includes many different forms of declarations, so these are described in a number of chapters covering basic declarations, subprograms and embedded packages separately. Attribute expressions are also described separately from the other expressions.

Environment The declarations in a SPARK text, together with the pre-defined identifiers (see Appendix A), create the context which determines the well-formation of terms

in the Abstract Syntax. We therefore require an *environment* which records the properties of declared identifiers. The structure of this environment is described in Chapter 2.

Well-formation Predicates The well-formation of a term in the Abstract Syntax is defined by a *predicate*, in which the well-formation of the term may depend upon the well-formation of its subcomponents, and the current state of the environment. The predicates for each category form a mutually recursive set. The well-formation predicate for each Abstract Syntax category is expressed as a *relation* between the environment in which the term appears, the term itself and sometimes additional arguments such as the modified environment or a type value.

The relations are defined using inference rules, in the following form:

$$\frac{\begin{array}{l}\forall\ declarations \mid \\ \qquad side\ conditions \bullet \\ \\ premise\ 1 \\ premise\ 2\end{array}}{conclusion}$$

The premises and conclusions are well-formation predicates. The relation defined by the inference rules is the smallest relation satisfying all the rules given.

Notation The specification is written using the Z notation. Apart from the inference rules, which are not in a standard Z notation, we have departed from standard Z in the following respects:

1. **Schema Update** The following notation is used for the schema binding in which all the components have the same value as S, except that x is now equal to e:

   $$S[x := e]$$

   This notation is described in detail in Appendix C.1.

2. **Subscripts** In some cases the first argument (or arguments) of a function is written as a subscript to the function name. This purely a lexical convenience which helps to draw attention to the significant arguments (i.e. the other ones) of a function call.

3. **Recursion and Declaration Order** The Abstract Syntax sets are mutually recursive. This is not allowed in some versions of Z. In general, we do not always observe the declaration before use rule of Z, choosing instead the order which (we hope) makes it easier to read our definitions.

Status The following are incompletely treated in the present version:

1. Private and limited private types.
2. Definition of the constant and static properties, and the evaluation of static expressions.
3. Characters and strings.
4. Declaration and definitions of separates. (N.B. One suggestion is to treat these as rewritten in the abstract syntax to eliminate them. While this is a rewrite which could not be achieved lexically, nor without some considerable effort, it is an attractive proposal since the presence of such separates is only a device for separate compilation and can have little impact of interest in practice.)
5. Embedded packages.

Chapter 2

Environment Definitions

Whether or not a term in the abstract syntax is well-formed depends on the context in which it occurs. This context is created by the declarations of SPARK. We therefore require an *environment* to hold information about all the identifiers known about at a particular point in the program text. In SPARK identifiers can be associated with constants, types and subtypes, variables, functions, procedures and packages.

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

2.1 Basic Sets

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 Values

A set of values is required which includes all the values which can be given to static expressions in SPARK, i.e. all the values which need to be evaluated in the static semantics. These values are integers, real numbers and enumeration literals.

$$\begin{aligned} Val ::= & \ intval\langle\langle \mathbb{Z} \rangle\rangle \\ | & \ realval\langle\langle \text{Real} \rangle\rangle \\ | & \ enumval\langle\langle IdDot \rangle\rangle \end{aligned}$$

The set *Val* does not include a character value since the predefined Character type is an enumeration type (see Section A.3.4). Elements of *Char* only appear as character literals.

2.3 Identifiers

A separate component of the environment maintains the set of identifiers which cannot be given a new definition in a declaration.

$$IdEnv \mathrel{\widehat=} [usedids : \mathbb{P}\ Id]$$

In most cases, SPARK does not allow an identifier to be redefined within the scope of an existing definition. In some cases, an identifier is not available even when its existing definition cannot be used.

2.4 Constants

The environment holds the set of visible constants, including emuneration literals. All such constants have a type. Scalar constants also have a value, from the set *Val*.

SPARK allows the use of deferred constants, for which the type is declared (in the visible part of a package specification) before the value is specified by a *full declaration* (in the private part of the same specification). The set *defcons* contains deferred constants awaiting a full-declaration; these constants cannot be referenced.

Enumeration literals are represented as constants in the environment: the value of an enumeration literal is an integer position number; its type is the enumeration type.

ConstEnv

$$\begin{aligned} & contypes : Id \pfun IdDot \\ & convals : Id \pfun Val \\ & defcons : \mathbb{P}\ Id \\ \hline & \text{dom}\ convals \subseteq \text{dom}\ contypes \\ & defcons \subseteq (\text{dom}\ contypes \setminus \text{dom}\ convals) \end{aligned}$$

The *contypes* function gives the type mark used in the declaration, which therefore may be a type or subtype.

References: *Val* p. 6.

2.5 Types and Subtypes

Types in SPARK are always explicitly named. All types and subtypes are described by a *type construction* (belonging to *TypCon*). Subtype are mapped to the name of their *ancestor* type (the *full* type from which the subtype is descended).

TypeEnv

$$types : Id \pfun TypCon$$
$$subtypes : Id \pfun IdDot$$

References: *TypCon* p. 15.

2.6 Operators

Operators are not explicitly declared in SPARK. Type declarations, which implicitly declare operators, and renaming declarations together determine the visibility of operators between particular operand types.

Unary operators (belonging to *Uop*) are defined for some basic types (*BasicType*). Binary operators (belonging to *Bop*) are defined for some pairs of basic types. The basic types include the universal types as well as local and non-local declared types.

OpEnv

$$unops : Uop \times BasicType \pfun BasicType$$
$$binops : BasicType \times Bop \times BasicType \pfun BasicType$$

References: *Uop* p. 100; *BasicType* p. 73; *Bop* p. 102.

2.7 Variables

A variable in SPARK is characterised by the typemark used in the variable declaration and the access modes, which allow some combination of read and write access to the variable.

Access Modes All variables have a set of allowed access modes — read and write. Formal parameters, which are a form of variable, may be declared with restricted access modes. The access modes are not fixed by the declaration of a variable, since the **derives** annotations may restrict the access modes in a particular scope.

$$RdWr ::= rd \mid wr$$

Own Variables In SPARK, an **own** variable annotation at the start of a package specification is used to introduce variables which will later be declared in the outer-most scope of a package. These variables are in the set *ownvars*; until their Ada declaration they have a mode but not a type. Own variables in the set *initowns* are initialised in the package initialisation.

VarEnv

$$
\begin{array}{l}
vartypes : Id \pfun IdDot \\
varmodes : Id \rel RdWr \\
ownvars, initowns : \mathbb{P}\, Id \\
\hline
\mathrm{dom}\, vartypes \subseteq \mathrm{dom}\, varmodes \\
initowns \subseteq ownvars \\
ownvars \subseteq \mathrm{dom}\, varmodes
\end{array}
$$

2.8 Subprograms

In SPARK a subprogram may be described in two separate parts:

1. The *declaration*, which appears in the specification of a package, gives the name, formal parameters and annotations of a procedure.

2. The *definition*, which contains the declarations and statements which implement the procedure.

Procedure Declaration Each formal parameter of a procedure has a type, a set of access modes and a position. Within the procedure a formal parameter becomes a variable.

The **global** and **derives** annotations of a procedure describe the way in which the procedure uses global variables. The set *glvars* contains the global variables used by the procedure, while *exvars* is the set of exports (including any exported formal parameters).

Proc

$$
\begin{array}{l}
glvars, exvars : \mathbb{P}\, IdDot \\
ftypes : Id \pfun IdDot \\
fmodes : Id \rel RdWr \\
fpos : \mathrm{iseq}\, Id \\
\hline
exvars \subseteq glvars \\
\mathrm{dom}\, ftypes = \mathrm{dom}\, fmodes = \mathrm{ran}\, fpos
\end{array}
$$

The set of formal parameter identifiers is the domain of the functions *ftype* and *fmodes*, while *fpos* gives the parameter identifier for each position.

Function Declaration The formal parameters of a function have a type and a position (no mode is required). The set *glvars* is the set of global variables read by the procedure, as given in the **global** annotation.

Fun

$$
\begin{array}{l}
glvars : \mathbb{P}\ IdDot \\
ftypes : Id \pfun IdDot \\
fpos : \mathrm{iseq}\ Id \\
rtype : IdDot \\
\hline
\mathrm{dom}\ ftypes = \mathrm{ran}\ fpos
\end{array}
$$

Subprogram Definitions A subprogram declared in a package specification is defined in the outer-most scope of the body of the package. The environment includes a set *sdef* of the subprograms which can be defined in the current environment. A subprogram is added to *sdef* in the outer-most scope of the body of the package in whose specification the procedure is declared and removed when the subprogram has been defined.

Calling Subprograms The environment includes a set *scall* of subprograms which can be called in the current environment. A subprogram can only be called after it has been defined, provided that its global variables are accessible with the required access modes.

Subprogram Environment The subprogram environment combines these components. The functions *pdecls* and *fdecls* give the properties of all procedures and functions which have been declared.

SubEnv

$$
\begin{array}{l}
pdecls : Id \pfun Proc \\
fdecls : Id \pfun Fun \\
scall, sdef : \mathbb{P}\ Id
\end{array}
$$

References: RdWr p. 7

2.9 Context

Not all the package specifications which have been defined are visible in all parts of the program. The Ada **with** clause specifies which packages are visible in a library unit. The SPARK **inherit** annotation specifies which packages are visible in an annotation within any package or the main program.

All scopes in SPARK are named[1]. The name of the scope, which is a sequence of identifiers, is required to identify a separate declaration.

Context

$$\begin{array}{l} with\!s : \mathbb{P}\ Id \\ inherits : \mathbb{P}\ Id \\ scope : \mathrm{seq}\ Id \end{array}$$

2.10 Packages

A package is described in two parts:

1. The package specification contains all the declarations required by an external user of the package.

2. The package body contains the declarations which implement the externally available operations of the package.

Specification Environment This contains constants, types, operators variables and subprogram declarations.

$$SpecEnv \mathrel{\widehat=} IdEnv \land ConstEnv \land TypeEnv \land OpEnv \land VarEnv \land SubEnv$$

Visible and Private Parts All package specifications have a visible part, containing the declarations which are visible outside the package. A package specification may have a private part, which has additional declarations visible only in the body of the package. A package specification also includes a context which must be recorded since it determines the visibility in the package body.

Pak

$$\begin{array}{l} vis, priv : SpecEnv \\ Context \end{array}$$

Informally, we can consider the private part *priv* to be a superset of the visible part *vis*; this is not precise since the private declarations in the visible part are replaced by the corresponding full-declarations.

[1] The for-loop is an exception, but the only scope it can contain is another for-loop.

Package Environment The function *paks* gives the properties of all the package specifications which have been declared. The package body must appear in a scope at the same level as its specification: the set *bodies_req* contains the identifiers of package bodies expected in the current scope.

PakEnv

$$paks : Id \pfun Pak$$
$$bodies\_req : \mathbb{P}\, Id$$

References: IdEnv p. 6; *ConstEnv* p. 6; *TypeEnv* p. 7; *VarEnv* p. 8; *SubEnv* p. 9; *Context* p. 10.

2.11 Subunits

Procedures, functions and package bodies may be declared to be **separate** . The complete declaration is then made separately and added to the library as a *subunit*.

The name of the subunit includes the complete scope name, starting from a library unit; a sequence of identifiers is therefore required. The subunit name is mapped to the environment in which the stub was declared. This environment may contain any of the elements described above.

$$SepEnv \mathrel{\widehat=} SpecEnv \land PakEnv$$

The set of subunits yet to be defined is given by *sep_req*.

SUnitEnv

$$subunits : \mathrm{seq}\, Id \pfun SepEnv$$
$$sep\_req : \mathbb{P}(\mathrm{seq}\, Id)$$

References: SpecEnv p. 10; *OpEnv* p. 7; *PakEnv* p. 11.

2.12 Main Program

The main program is not necessarily the final compilation unit in a SPARK text, since it may be followed by subunits (possibly separate from the main program itself). It is necessary, therefore, for the environment to hold information about the main program: only its name is required.

$$MainEnv \mathrel{\widehat=} [main : \mathbb{P}\, Id]$$

2.13 The Complete Environment

The complete environment combines all the above elements.

$$Env \mathrel{\widehat{=}} SepEnv \wedge SUnitEnv \wedge MainEnv$$

References: *SepEnv* p. 11; *SUnitEnv* p. 11; *MainEnv* p. 11

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:

Syntax Constructor	Description	Page
int	New integer	22
enum	Enumeration	23
float	Floating point	25
floatr	Floating point with range	27
fixr	Fixed point with range	29
arr	Array	31
uarr	Unconstrained array	34
rec	Record	37

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

Description	Page
Type Constructors	15
Declaration of Operators	17

The Well-Formation Predicate

The well-formation of a type definition is described by a relation between the environment, the type definition and the environment modified by the declaration of the type and its operators. (The name of the new type is not included in the *TypDef* term —

belonging instead to the *Decl* term which represents type declarations. We therefore add this identifier to the well-formation relation). The declaration of this relation is:

$$\_, \_ \vdash_{TypDef} \_ \Longrightarrow \_ \subseteq Env \times Id \times TypDef \times Env$$

Thus the predicate:

$$\delta, t \vdash_{TypDef} typ \Longrightarrow \delta'$$

can be read as "the type t, with definition typ is well-formed in the environment δ, yielding the new environment δ'".

3.1 Type Constructions

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

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

The following subsections describe the different type constructors.

3.1.1 Integer

An integer type is defined by a set of integer ranges; all the values in the set belong to the type.

$$IntTypCon == \{ \; v : Val \mid v \in \operatorname{ran} intval \; \}$$

3.1.2 Enumeration

An enumeration type is defined by a set of enumeration values.

$$EnumTypCon == \{ \; v : Val \mid v \in \operatorname{ran} enumval \; \}$$

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 used by a compiler to select a value for `'SMALL`, 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 type defines a composite type in which all the components of an object of that type have the same *component* type. The elements of the array are indexed by one or more index (sub)types *indexes*. Since SPARK requires explicit intermediate types to be used to create an array of arrays, both the component type and the index (sub)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*).

$$RecTypCon \mathrel{\widehat=} [fields : \mathrm{iseq}_1\ Id;\ types : Id \pfun IdDot \mid \mathrm{ran}\ fields = \mathrm{dom}\ types]$$

3.1.7 Private and Limited Private

Private and limited private types are constructed using $privT$ and $limT$ respectively.
References: *Val* p. 6; *intval* p. 6; *enumval* p. 6; *realval* p. 6

3.2 Declaration of Operators

The declaration of a type also implicitly declares operators, depending on the form of the type declaration. In this section we group together the description of these operators, by defining functions (from *UOpFun* and *BOpFun*) returning sets of operator descriptions for a given basic type. (The set *BasicType* is declared on page 73).

$$UOpFun == BasicType \rightarrow ((Uop \times BasicType) \nrightarrow BasicType)$$

$$BOpFun == BasicType \rightarrow ((BasicType \times Bop \times BasicType) \nrightarrow BasicType)$$

3.2.1 Operations on Enumerated Types

The declaration of an enumerated type is implicitly accompanied by the declaration of the relational operators for the type (LRM 4.5.2).

The following function returns the binary operators of an enumerated type.

$$enumbops : BasicType \rightarrow ((BasicType \times Bop \times BasicType) \nrightarrow BasicType)$$

$$\begin{aligned}
&\forall\, enumtype : BasicType \bullet \\
&\quad enumbops\ enumtype = \{ \\
&\qquad (enumtype, eq, enumtype) \mapsto nameT(id\ boolean), \\
&\qquad (enumtype, noteq, enumtype) \mapsto nameT(id\ boolean), \\
&\qquad (enumtype, lt, enumtype) \mapsto nameT(id\ boolean), \\
&\qquad (enumtype, lte, enumtype) \mapsto nameT(id\ boolean), \\
&\qquad (enumtype, gt, enumtype) \mapsto nameT(id\ boolean), \\
&\qquad (enumtype, gte, enumtype) \mapsto nameT(id\ boolean)\ \}
\end{aligned}$$

3.2.2 Operations on Boolean Types

The logical operators are implicitly declared for the predefined boolean type (see Section A.3), and for any one-dimensional array of booleans [LRM 4.5.1].

The following function returns the unary operators defined for a logical type.

$$loguops : BasicType \rightarrow ((Uop \times BasicType) \nrightarrow BasicType)$$

$$\begin{aligned}
&\forall\, logtype : BasicType \bullet \\
&\quad loguops\ logtype = \{\ (not, logtype) \mapsto logtype\ \}
\end{aligned}$$

The following function returns the binary operators defined for a logical type.

$$logbops : BasicType \rightarrow ((BasicType \times Bop \times BasicType) \nrightarrow BasicType)$$

$$\begin{aligned}
&\forall\, logtype : BasicType \bullet \\
&\quad logbops\ logtype = \{ \\
&\qquad (logtype, and, logtype) \mapsto logtype, \\
&\qquad (logtype, or, logtype) \mapsto logtype, \\
&\qquad (logtype, xor, logtype) \mapsto logtype\ \}
\end{aligned}$$

3.2.3 Operations on Integer Types

The declaration of an integer type is implicitly accompanied by the declaration of the following operators:

1. The relational operators (LRM 4.5.2).

2. The binary adding operators plus and minus (LRM 4.5.3), but not "catenation" (SR 4.5.3).

3. The unary adding operators (LRM 4.5.4).

4. The multiplying operators mul, div, mod and rem (LRM 4.5.5).

5. The "highest precedence operators" abs and exponentiation; the right-hand argument of exponentiation is always the predefined type integer (LRM 4.5.6).

The following function returns the unary operators of an integer type.

$$intuops : BasicType \rightarrow ((Uop \times BasicType) \nrightarrow BasicType)$$

$$\begin{aligned}
&\forall\, inttype : BasicType \bullet \\
&\quad intuops\ inttype = \{ \\
&\qquad (uplus, inttype) \mapsto inttype, \\
&\qquad (uminus, inttype) \mapsto inttype, \\
&\qquad (abs, inttype) \mapsto inttype\}
\end{aligned}$$

The following function returns the binary operators of an integer type.

$$
\begin{array}{|l}
intbops : BasicType \rightarrow ((BasicType \times Bop \times BasicType) \nrightarrow BasicType) \\
\hline
\forall\, inttype : BasicType \bullet \\
\quad intbops\ inttype = \{ \\
\qquad (inttype, eq, inttype) \mapsto nameT(id\ boolean), \\
\qquad (inttype, noteq, inttype) \mapsto nameT(id\ boolean), \\
\qquad (inttype, lt, inttype) \mapsto nameT(id\ boolean), \\
\qquad (inttype, lte, inttype) \mapsto nameT(id\ boolean), \\
\qquad (inttype, gt, inttype) \mapsto nameT(id\ boolean), \\
\qquad (inttype, gte, inttype) \mapsto nameT(id\ boolean), \\
\qquad (inttype, plus, inttype) \mapsto inttype, \\
\qquad (inttype, minus, inttype) \mapsto inttype, \\
\qquad (inttype, mul, inttype) \mapsto inttype, \\
\qquad (inttype, div, inttype) \mapsto inttype, \\
\qquad (inttype, rem, inttype) \mapsto inttype, \\
\qquad (inttype, mod, inttype) \mapsto inttype, \\
\qquad (inttype, power, nameT(id\ integer)) \mapsto inttype\ \}
\end{array}
$$

3.2.4 Floating Point Types

The declaration of a floating point type is implicitly accompanied by the declaration of the following operators:

1. The relational operators (LRM 4.5.2), but excluding equality and its complement (SR 4.5.2).
2. The binary adding operators plus and minus (LRM 4.5.3), but not "catenation" (SR 4.5.3).
3. The unary adding operators (LRM 4.5.4).
4. The multiplying operators mul, div (LRM 4.5.5).
5. The "highest precedence operator" abs (LRM 4.5.6) and exponentiation; the right-hand argument of exponentiation is always the predefined type integer (LRM 4.5.6).

The following function returns the unary operators declared for a floating point type.

$$
\begin{array}{|l}
fltuops : BasicType \rightarrow (Uop \times BasicType) \nrightarrow BasicType \\
\hline
\forall\, flttype : BasicType \bullet \\
\quad fltuops\ flttype = \{ \\
\qquad (uplus, flttype) \mapsto flttype, \\
\qquad (uminus, flttype) \mapsto flttype, \\
\qquad (abs, flttype) \mapsto flttype\ \}
\end{array}
$$

The following function returns the binary operators declared for a floating point type.

$$
\begin{array}{|l}
fltbops : BasicType \rightarrow (BasicType \times Bop \times BasicType) \nrightarrow BasicType \\
\hline
\forall\, flttype : BasicType \bullet \\
\quad fltbops\ flttype = \{ \\
\qquad (flttype, lt, flttype) \mapsto nameT(id\ boolean), \\
\qquad (flttype, lte, flttype) \mapsto nameT(id\ boolean), \\
\qquad (flttype, gt, flttype) \mapsto nameT(id\ boolean), \\
\qquad (flttype, gte, flttype) \mapsto nameT(id\ boolean), \\
\qquad (flttype, plus, flttype) \mapsto flttype, \\
\qquad (flttype, minus, flttype) \mapsto flttype, \\
\qquad (flttype, mul, flttype) \mapsto flttype, \\
\qquad (flttype, div, flttype) \mapsto flttype, \\
\qquad (flttype, power, nameT(id\ integer)) \mapsto flttype\ \}
\end{array}
$$

3.2.5 Fixed Point Types

The declaration of a fixed point type is implicitly accompanied by the declaration of the following operators:

1. The relational operators [AARM, §4.5.2], including equality and its complement.
2. The binary adding operators plus and minus [AARM, §4.5.3], but not "catenation" [SR, §4.5.3].
3. The unary adding operators [AARM, §4.5.4].
4. The multiplying operators taking the new type as the one operand and the result type, with standard integer as the other operand [AARM, §4.5.5 ¶7]. (For div, the fixed point type is allowed only as the left operand). There are no additional homogeneous multiplying operators [AARM, §4.5.5 ¶10].
5. The "highest precedence operator" abs [AARM, §4.5.6 ¶2] but not exponentiation [AARM, §4.5.6 ¶4].

The following function returns the unary operators declared for a fixed point type.

$$
\begin{array}{|l}
fixuops : BasicType \rightarrow (Uop \times BasicType) \nrightarrow BasicType \\
\hline
\forall\, fixtype : BasicType \bullet \\
\quad fixuops\ fixtype = \{ \\
\qquad (uplus, fixtype) \mapsto fixtype, \\
\qquad (uminus, fixtype) \mapsto fixtype, \\
\qquad (abs, fixtype) \mapsto fixtype\ \}
\end{array}
$$

The following function returns the binary operators declared for a fixed point type.

$$
\begin{array}{|l}
fixbops : BasicType \rightarrow (BasicType \times Bop \times BasicType) \nrightarrow BasicType \\
\hline
\forall\, fixtype : BasicType \bullet \\
\quad fixbops\ fixtype = \{ \\
\qquad (fixtype, eq, fixtype) \mapsto nameT(id\ boolean), \\
\qquad (fixtype, noteq, fixtype) \mapsto nameT(id\ boolean), \\
\qquad (fixtype, lt, fixtype) \mapsto nameT(id\ boolean), \\
\qquad (fixtype, lte, fixtype) \mapsto nameT(id\ boolean), \\
\qquad (fixtype, gt, fixtype) \mapsto nameT(id\ boolean), \\
\qquad (fixtype, gte, fixtype) \mapsto nameT(id\ boolean), \\
\qquad (fixtype, plus, fixtype) \mapsto fixtype, \\
\qquad (fixtype, minus, fixtype) \mapsto fixtype, \\
\qquad (fixtype, mul, nameT(id\ integer)) \mapsto fixtype, \\
\qquad (nameT(id\ integer), mul, fixtype) \mapsto fixtype, \\
\qquad (fixtype, div, nameT(id\ integer)) \mapsto fixtype\ \}
\end{array}
$$

3.2.6 Composite Types

The declaration of a composite type may be implicitly accompanied by the declaration of the equality operator and its complement. (The operators are available if the type is an equality type; the conditions for this are given in the section for each of the composite type constructors.)

The following function returns the binary operators declared for a composite *equality* type.

$$
\begin{array}{|l}
combops : BasicType \rightarrow (BasicType \times Bop \times BasicType) \nrightarrow BasicType \\
\hline
\forall\, comtype : BasicType \bullet \\
\quad combops\ comptype = \{ \\
\qquad (comtype, eq, comtype) \mapsto nameT(id\ boolean), \\
\qquad (comtype, noteq, comtype) \mapsto nameT(id\ boolean)\ \}
\end{array}
$$

References: *BasicType* p. 73; *Bop* p. 102; *nameT* p. 73; *boolean* p. 265; *Uop* p. 100; *integer* p. 265

3.3 Integer Types

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

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

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 Static Semantics

1. The expressions giving the bounds must be well-formed, static and of an integer type. The expressions need not have the *same* integer type.
2. The values to which the bounds evaluate determine the range of the type, which must be non-empty [SR, §3.5].
3. The declaration of an integer type declares the standard integer operators.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ IntTypDef;\ lval, uval : Val;\\ \quad ltyp, utyp : ExpType \mid\\ \qquad is\_int\_exp_\delta\ ltyp \wedge\\ \qquad is\_int\_exp_\delta\ utyp \wedge\\ \qquad lval\ {}_{V..\delta}\ uval \neq \varnothing \bullet\\ \quad \delta \vdash_{Exp} lower : ltyp\\ \quad \delta \vdash_{Exp} upper : utyp\\ \quad \delta \vdash lower \Longrightarrow_{\mathrm{StaticEval}} lval\\ \quad \delta \vdash upper \Longrightarrow_{\mathrm{StaticEval}} uval\end{array}}{\delta, t \vdash_{TypDef} int(\theta IntTypDef) \Longrightarrow \delta'} \quad \text{(Int)}$$

where

$$\begin{array}{l}\delta' == \delta[\ unops := \delta.unops \cup intuops\ (nameT\ (id\ t)),\\ \qquad binops := \delta.binops \cup intbops\ (nameT\ (id\ t)),\\ \qquad types := \delta.types \cup \{\ t \mapsto intT(lval\ {}_{V..\delta}\ uval)\ \}\ ]\end{array}$$

References: *Exp* p. 71; *Env* p. 12; *Val* p. 6; *ExpType* p. 73; $is\_int\_exp_\delta$ p. 295; ${}_{V..\delta}$ p. 297; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\mathrm{StaticEval}}$ p. 298; *intuops* p. 18; *intbops* p. 19; *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 Static Semantics

1. The enumeration literals must be distinct.
2. There must be at least two enumeration literals in the type [SR, §3.5.1].
3. The enumeration literals must be different from all currently visible identifiers [SR, §3.5.1, §3.11].
4. The declaration of an enumeration type declares the standard operators for an enumeration type.

$$\frac{\forall\, \delta, \delta_e : Env;\ t : Id;\ is : \mathrm{iseq}\, Id \bullet \quad \delta, t, 0 \vdash_{Enum} is \Longrightarrow \delta_e}{\delta, t \vdash_{TypDef}\ enum\ is \Longrightarrow \delta'} \qquad \text{(Enum)}$$

Note that the position of the first enumeration literal in a enumeration type is zero [AARM, §3.5.1, ¶4]. **where**

$$\delta' == \delta_e[\ \ binops := \delta_e.binops \cup enumbops\ (nameT\ (id\ t)),\ \ types := \delta_e.types \cup \{\ t \mapsto enumT\ (id(\lvert \mathrm{ran}\ is \rvert))\ \}\ ]$$

References: *Env* p. 12; $\vdash_{Enum}$ p. 24; *enumbops* p. 17; *nameT* p. 73; *enumT* p. 15; *intval* p. 6

3.4.3 Enumeration Literals

The enumeration literals are added to the environment as constants, with their value equal to the literal's position number. An additional well-formation relation is required:

$$\_, \_, \_ \vdash_{Enum} \_ \Longrightarrow \_ \subseteq Env \times Id \times \mathbb{N} \times \mathrm{seq}\, Id \times Env$$

the predicate $\delta, t, p \vdash_{Enum} is \Longrightarrow \delta'$ states that the (partial) sequence of enumeration literals *is* for type t, starting at position p are well-formed in the initial environment δ, with new environment δ'.

The empty sequence of enumeration literals does not change the environment.

$$\frac{\forall\, \delta : Env;\ t : Id;\ p : \mathbb{N} \bullet}{\delta, t, p \vdash_{Enum} \langle\rangle \Longrightarrow \delta} \qquad \text{(Enum\_Empty)}$$

An enumeration literal must be unused; the position number determines the value of the corresponding constant.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_e : Env;\ t : Id;\ p : \mathbb{N};\ e : Id;\ es : \mathrm{seq}\, Id \mid \\ \qquad e \notin \delta_e.usedids \bullet \\ \delta, t, p + 1 \vdash_{Enum} es \Longrightarrow \delta_e\end{array}}{\delta, t, p \vdash_{Enum} \langle e \rangle \frown es \Longrightarrow \delta'} \qquad \text{(Enum\_Id)}$$

where

$$\begin{array}{l}\delta' == \delta_e\ [\ usedids := \delta_e.usedids \cup \{e\}, \\ \qquad\qquad convals := \delta_e.convals \cup \{e \mapsto intval\ p\}, \\ \qquad\qquad contypes := \delta_e.contypes \cup \{e \mapsto id\ t\}\ ]\end{array}$$

References: *Env* p. 12; *intval* p. 6

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

3.5.2 Static Semantics

1. The expression specifying the number of digits must be well-formed, static and of an integer type.
2. The number of digits must be positive.
3. *Implementation Dependent* The number of digits must not exceed the maximum allowed by the implementation (given by `system.max_digits`).
4. The declaration of a floating point type declares the standard floating point operators.
5. *Implementation Dependent* The range of the type is determined by the implementation's choice of a pre-defined floating point type as the base type for the new type. Neither the choice of base type, nor the range of the new type, are determined by the rules of SPARK.

   Note that the rules about model numbers determine a minimum range for any given number of digits.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ digexp : Exp;\ dignum : \mathbb{N};\\ \quad dtyp : ExpType;\ vals : \mathbb{P}\ Val \mid\\ \qquad is\_int\_exp_{\delta}\ dtyp \wedge\\ \qquad dignum > 0 \bullet\\ \quad \delta \vdash_{Exp} digexp : dtyp\\ \quad \delta \vdash digexp \Longrightarrow_{\mathrm{StaticEval}} intval\ dignum\end{array}}{\delta, t \vdash_{TypDef} float\ digexp \Longrightarrow \delta'} \quad \text{(Float)}$$

where

$$\begin{array}{l}\delta' == \delta\ [\ unops := \delta.unops \cup \mathit{fltuops}\ (nameT\ (id\ t)),\\ \qquad\quad binops := \delta.binops \cup \mathit{fltbops}\ (nameT\ (id\ t)),\\ \qquad\quad types := \delta.types \cup \{\ t \mapsto \mathit{floatT}(\mu\, FloatTypCon \mid\\ \qquad\qquad range = vals \wedge\\ \qquad\qquad digits = dignum)\ \}\ ]\end{array}$$

References: *Exp* p. 71; *Env* p. 12; *ExpType* p. 73; *Val* p. 6; $is\_int\_exp_{\delta}$ p. 295; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\text{StaticEval}}$ p. 298; *intval* p. 6; *fltuops* p. 20; *nameT* p. 73; *fltbops* p. 20; *floatT* p. 15; *FloatTypCon* p. 15

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	$floatr \langle\!\langle\ digits \mapsto lint\ 7,\ lower \mapsto lreal\ 0.0,\ upper \mapsto lreal\ 100.0\ \rangle\!\rangle$

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 Static Semantics

1. The expression specifying the number of digits must be well-formed, static and of an integer type.

2. The number of digits must be positive.

3. The expressions giving the bounds must be well-formed, static and of a real type. The expressions need not have the *same* real type.

4. The values to which the bounds evaluate determine the range of the type, which must be non-empty [SR, §3.5].

5. The declaration of a floating point type declares the standard floating point operators.

6. *Implementation Dependent* A predefined floating type must exist with sufficiently large range and number of digits to implement the declared type.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ FloatRTypDef;\ dignum : \mathbb{N};\\ \quad dtyp, ltyp, utyp : ExpType;\ lval, uval : Val \mid\\ \qquad is\_int\_exp_\delta\ dtyp \land\\ \qquad is\_real\_exp_\delta\ ltyp \land\\ \qquad is\_real\_exp_\delta\ utyp \land\\ \qquad dignum > 0 \land\\ \qquad lval\ {}_{V..\delta}\ uval \neq \varnothing \bullet\\ \\ \delta \vdash_{Exp} digexp : dtyp\\ \delta \vdash_{Exp} lower : ltyp\\ \delta \vdash_{Exp} upper : utyp\\ \delta \vdash digexp \Longrightarrow_{\text{StaticEval}} intval\ dignum\\ \delta \vdash lower \Longrightarrow_{\text{StaticEval}} lval\\ \delta \vdash upper \Longrightarrow_{\text{StaticEval}} uval\end{array}}{\delta, t \vdash_{TypDef} floatr(\theta FloatRTypDef) \Longrightarrow \delta'} \quad \text{(FloatR)}$$

where

$$\begin{array}{l}\delta' == \delta\ [\ unops := \delta.unops \cup fltuops\ (nameT\ (id\ t)),\\ \qquad binops := \delta.binops \cup fltbops\ (nameT\ (id\ t)),\\ \qquad types := \delta.types \cup \{\ t \mapsto floatT(\mu\, FloatTypCon \mid\\ \qquad\quad range = (lval\ {}_{V..\delta}\ uval) \land\\ \qquad\quad digits = dignum)\ \}\ ]\end{array}$$

References: *Exp* p. 71; *Env* p. 12; *ExpType* p. 73; *Val* p. 6; *is_int_exp*$_\delta$ p. 295; ${}_{V..\delta}$ p. 297; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\text{StaticEval}}$ p. 298; *intval* p. 6; *fltuops* p. 20; *nameT* p. 73; *fltbops* p. 20; *floatT* p. 15; *FloatTypCon* p. 15

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	$fixr \langle\!\langle\ delta \mapsto lreal\ 0.001,\ lower \mapsto lreal\ 0.0,\ upper \mapsto lreal\ 50.0\ \rangle\!\rangle$

3.7.1 Abstract Syntax

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

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

$$TypDef ::= \ldots \mid fixr\langle\langle FixRTypDef \rangle\rangle$$

3.7.2 Static Semantics

1. The expression specifying the accuracy must be well-formed, static and of some real type.

2. The accuracy must be positive.

3. The expressions giving the bounds must be well-formed, static and of a real type. The expressions need not have the *same* real type.

4. The values to which the bounds evaluate determine the range of the type, which must be non-empty [SR, §3.5].

5. The declaration of a fixed point type declares the standard fixed point operators.

6. *Implementation Dependent* A predefined fixed point type must exist with sufficiently large range and accuracy to implement the declared type.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ FixRTypDef;\ acc : \mathrm{Real};\\ \quad dtyp, ltyp, utyp : ExpType;\ lval, uval : Val \mid\\ \qquad is\_real\_exp_\delta\ dtyp \wedge\\ \qquad is\_real\_exp_\delta\ ltyp \wedge\\ \qquad is\_real\_exp_\delta\ utyp \wedge\\ \qquad acc > 0 \wedge\\ \qquad lval\ {}_{V..\delta}\ uval \neq \varnothing \bullet\\ \delta \vdash_{Exp} digexp : dtyp\\ \delta \vdash_{Exp} lower : ltyp\\ \delta \vdash_{Exp} upper : utyp\\ \delta \vdash digexp \Longrightarrow_{\mathrm{StaticEval}} realval\ acc\\ \delta \vdash lower \Longrightarrow_{\mathrm{StaticEval}} lval\\ \delta \vdash upper \Longrightarrow_{\mathrm{StaticEval}} uval\end{array}}{\delta, t \vdash_{TypDef} fixr(\theta FixRTypDef) \Longrightarrow \delta'} \quad \text{(FixR)}$$

where

$$\begin{array}{l}\delta' == \delta\ [\ unops := \delta.unops \cup fixuops\ (nameT\ (id\ t)),\\ \qquad binops := \delta.binops \cup fixbops\ (nameT\ (id\ t)),\\ \qquad types := \delta.types \cup \{\ t \mapsto fixedT(\mu\ FixedTypCon \mid\\ \qquad\quad range = (lval\ {}_{V..\delta}\ uval) \wedge\\ \qquad\quad delta = acc)\ \}\ ]\end{array}$$

References: *Exp* p. 71; *Env* p. 12; *ExpType* p. 73; *Val* p. 6; *is_int_exp*$_\delta$ p. 295; ${}_{V..\delta}$ p. 297; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\mathrm{StaticEval}}$ p. 298; *realval* p. 6; *fixuops* p. 21; *nameT* p. 73; *fixbops* p. 21; *fixedT* p. 15; *FixedTypCon* p. 16

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	$arr \langle\!\langle\ indexes \mapsto \langle dot(bran, code), id\ prod\rangle,$ $component \mapsto id\ cust\ \rangle\!\rangle$

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 Static Semantics

1. The elements of the list *indexes* must be visible discrete typemarks.

2. The component type *component* must be visible and not unconstrained.

3. There are three possible cases concerning the operators declared for an array type:

   (a) The array is boolean, i.e. it has a single dimension and its component type is the predefined boolean. In this case the logical operators, in addition to the standard equality operators for composite types are declared.

   (b) No operators are declared because the type is not an *equality* type. This occurs when the component type is either a limited type or a real type, or a composite type containing a component with this property.

   (c) In all other cases the standard equality operators of composite types are declared.

4. The elements of the array type constructor (*ArrTypCon*) are the same as those of the type definition (*ArrTypDef*).

Boolean Array Note that it is not necessary to add further constraints on the component type, since the predefined boolean type has fixed properties: it is always visible, and it is not unconstrained.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ ArrTypDef \mid \\ \quad \#indexes = 1 \land \\ \quad component = id\ boolean \land \\ \quad is\_visible\_tmark_\delta(indexes\ 1) \land \\ \quad is\_discrete\_tmark_\delta(indexes\ 1) \bullet\end{array}}{\delta, t \vdash_{TypDef}\ arr(\theta ArrTypDef) \Longrightarrow \delta'} \quad \text{(Arr1)}$$

where

$$\begin{array}{l}\delta' == \delta[\ \ unops := \delta.unops \cup loguops\ (nameT\ (id\ t)), \\ \qquad binops := \delta.binops \cup logbops\ (nameT\ (id\ t)) \cup \\ \qquad\qquad combops\ (nameT\ (id\ t)), \\ \qquad types := \delta.types \cup \{t \mapsto arrT(\theta ArrTypCon)\ \}\ ]\end{array}$$

Array of Non-Equality Component

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ ArrTypDef \mid \\ \quad (\forall\, i : \mathrm{dom}\ indexes \bullet \\ \qquad is\_visible\_tmark_\delta(indexes\ i) \land \\ \qquad is\_discrete\_tmark_\delta(indexes\ i)) \\ \quad is\_visible\_tmark_\delta\ component \land \\ \quad \neg\ is\_unconstrained\_tmark_\delta\ component \land \\ \quad \neg\ is\_equality\_tmark_\delta\ component\end{array}}{\delta, t \vdash_{TypDef}\ arr(\theta ArrTypDef) \Longrightarrow \delta'} \quad \text{(Arr2)}$$

where

$$\delta' == \delta[\ \ types := \delta.types \cup \{t \mapsto arrT(\theta ArrTypCon)\ \}\ ]$$

Standard Array

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ ArrTypDef \mid \\ \quad (\forall\, i : \mathrm{dom}\ indexes \bullet \\ \qquad is\_visible\_tmark_\delta(indexes\ i) \land \\ \qquad is\_discrete\_tmark_\delta(indexes\ i)) \\ \quad is\_visible\_tmark_\delta\ component \land \\ \quad \neg\ is\_unconstrained\_tmark_\delta\ component \land \\ \quad is\_equality\_tmark_\delta\ component \land \\ \quad (\#indexes > 1 \lor component \neq id\ boolean) \bullet\end{array}}{\delta, t \vdash_{TypDef}\ arr(\theta ArrTypDef) \Longrightarrow \delta'} \quad \text{(Arr3)}$$

where

$$\delta' == \delta[\ \ binops := \delta.binops \cup combops\ (nameT\ (id\ t)),\\ types := \delta.types \cup \{t \mapsto arrT(\theta ArrTypCon)\ \}\ ]$$

References: ArrTypCon p. 16; *Env* p. 12; *is_visible_tmark*$_\delta$ p. 281; *is_discrete_tmark*$_\delta$ p. 285; *is_unconstrained_tmark*$_\delta$ p. 283; *loguops* p. 17; *logbops* p. 18; *combops* p. 21; *arrT* p. 15; *is_equality_tmark*$_\delta$ p. 290

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	$uarr \langle\!\langle\ indexes \mapsto \langle id\ integer \rangle,$ $component \mapsto id\ customer\ \rangle\!\rangle$

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 Static Semantics

The well-formation rules for unconstrained array definitions are essentially the same as those for constrained array definitions.

1. The elements of the list *indexes* must be visible discrete typemarks.

2. The component type *component* must be visible and not unconstrained.

3. There are three possible cases concerning the operators declared for an array type:

   (a) The array is boolean, i.e. it has a single dimension and its component type is the predefined boolean. In this case the logical operators, in addition to the standard equality operators for composite types are declared.

   (b) No operators are declared because the type is not an *equality* type. This occurs when the component type is either a limited type or a real type, or a composite type containing a component with this property.

   (c) In all other cases the standard equality operators of composite types are declared.

   Although these operators are declared for unconstrained array types, they cannot be applied to objects of the unconstrained types, because unconstrained objects are never well-formed in SPARK. However, the operators can be applied to objects of constrained subtypes of the unconstrained array types.

4. The elements of the array type constructor (*ArrTypCon*) are the same as those of the type definition (*UArrTypDef*).

Boolean Array Note that it is not necessary to add further constraints on the component type, since the predefined boolean type has fixed properties: it is always visible, and it is not unconstrained.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ UArrTypDef \mid \\ \quad \#indexes = 1 \land \\ \quad component = id\ boolean \land \\ \quad is\_visible\_tmark_{\delta}(indexes\ 1) \land \\ \quad is\_discrete\_tmark_{\delta}(indexes\ 1) \bullet\end{array}}{\delta, t \vdash_{TypDef} uarr(\theta\, UArrTypDef) \Longrightarrow \delta'} \quad \text{(UArr1)}$$

where

$$\delta' == \delta[\ \ unops := \delta.unops \cup loguops\ (nameT\ (id\ t)),\ binops := \delta.binops \cup logbops\ (nameT\ (id\ t)) \cup combops\ (nameT\ (id\ t)),\ types := \delta.types \cup \{ t \mapsto uarrT(\theta ArrTypCon) \}\ ]$$

Array of Non-Equality Component

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ UArrTypDef \mid \\ \quad (\forall\, i : \mathrm{dom}\ indexes \bullet \\ \qquad is\_visible\_tmark_{\delta}(indexes\ i) \land \\ \qquad is\_discrete\_tmark_{\delta}(indexes\ i)) \\ \quad is\_visible\_tmark_{\delta}\ component \land \\ \quad \neg\ is\_unconstrained\_tmark_{\delta}\ component \land \\ \quad \neg\ is\_equality\_tmark_{\delta}\ component\end{array}}{\delta, t \vdash_{TypDef} uarr(\theta\, UArrTypDef) \Longrightarrow \delta'} \quad \text{(UArr2)}$$

where

$$\delta' == \delta[\ \ types := \delta.types \cup \{ t \mapsto uarrT(\theta ArrTypCon) \}\ ]$$

Standard Array

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : Id;\ UArrTypDef \mid \\ \quad (\forall\, i : \mathrm{dom}\, indexes \bullet \\ \qquad is\_visible\_tmark_\delta(indexes\ i) \land \\ \qquad is\_discrete\_tmark_\delta(indexes\ i)) \\ \quad is\_visible\_tmark_\delta\ component \land \\ \quad \neg\ is\_unconstrained\_tmark_\delta\ component \land \\ \quad is\_equality\_tmark_\delta\ component \land \\ \quad (\#indexes > 1 \lor component \neq id\ boolean) \bullet \end{array}}{\delta, t \vdash_{TypDef}\ uarr(\theta\, UArrTypDef) \Longrightarrow \delta'} \qquad \text{(UArr3)}$$

where

$$\begin{array}{l}\delta' == \delta[\ \ binops := \delta.binops \cup combops\ (nameT\ (id\ t)), \\ \qquad\qquad types := \delta.types \cup \{t \mapsto uarrT(\theta ArrTypCon)\ \}\ ]\end{array}$$

References: *ArrTypCon* p. 16; *Env* p. 12; *is_visible_tmark*$_\delta$ p. 281; *is_discrete_tmark*$_\delta$ p. 285; *is_unconstrained_tmark*$_\delta$ p. 283; *loguops* p. 17; *logbops* p. 18; *combops* p. 21; *uarrT* p. 15; *is_equality_tmark*$_\delta$ p. 290

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
	$rec\ \langle\ \langle\!\langle\ fld \mapsto name,\ comp \mapsto id\ string30\ \rangle\!\rangle,$
	$\langle\!\langle\ fld \mapsto min,\ comp \mapsto id\ integer\ \rangle\!\rangle,$
	$\langle\!\langle\ fld \mapsto max,\ comp \mapsto id\ integer\ \rangle\!\rangle\ \rangle$

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 Static Semantics

1. The definition of each field must be well-formed (see below).

2. The *types* and *fields* of the record type constructor *RecTypCon* are derived from the field definitions.

3. If all the component types are *equality* types then the record is also an equality type. In this case the equality operators are declared for the type; otherwise there are no operators.

Equality Components

$$\frac{\begin{array}{l}\forall \delta, \delta_f : Env;\ t : Id;\ flddefs : \mathrm{seq}_1\ RecFldTypDef;\ RecTypCon \mid \\ \quad (\forall i : \mathrm{dom}\ flddefs \bullet is\_equality\_tmark_\delta(flddefs\ i).comp) \bullet \\ \delta \vdash_{RFld} flddefs \Longrightarrow \delta_f, fields, types\end{array}}{\delta, t \vdash_{TypDef} rec\ flddefs \Longrightarrow \delta'} \quad \text{(Rec1)}$$

where

$$\delta' == \delta_f[\ \ binops := \delta_f.binops \cup combops\ (nameT\ (id\ t)),\\ types := \delta_f.types \cup \{\ t \mapsto \theta RecTypCon\ \}\ ]$$

A Non-Equality Component

$$\frac{\forall\, \delta, \delta_f : Env;\ t : Id;\ flddefs : \mathrm{seq}_1\, RecFldTypDef;\ RecTypCon \mid (\exists\, i : \mathrm{dom}\, flddefs \bullet \neg\ is\_equality\_tmark_\delta(flddefs\ i).comp) \bullet \quad \delta \vdash_{RFld} flddefs \Longrightarrow \delta_f, fields, types}{\delta, t \vdash_{TypDef}\ rec\ flddefs \Longrightarrow \delta'} \quad \text{(Rec2)}$$

where

$$\delta' == \delta_f[\ \ types := \delta_f.types \cup \{\ t \mapsto \theta RecTypCon\ \}\ ]$$

References: *RecTypCon* p. 16; *Env* p. 12; *is_equality_tmark*$_\delta$ p. 290; $\vdash_{RFld}$ p. 38; *combops* p. 21

3.10.3 Field Definitions

This section describes the well-formation of the list of field definitions, using a well-formation relation:

$$\_ \vdash_{RFld} \_ \Longrightarrow \_, \_, \_ \subseteq Env \times \mathrm{seq}\, RecFldTypDef \times Env \times \mathrm{seq}\, Id \times Id \nrightarrow IdDot$$

The Empty Field List is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{RFld} \langle\rangle \Longrightarrow \delta, \langle\rangle, \varnothing} \quad \text{(Fld1)}$$

The Non-Empty Field List

1. The first field name must be an identifier which is not already declared and is not used elsewhere in the list of fields.

2. The component type must be visible and not unconstrained.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_f : Env;\ RecFldTypDef;\ fs : \mathrm{seq}\, RecFldTypDef \mid \\ \qquad fld \notin \delta_f.usedids \land \\ \qquad is\_visible\_tmark_\delta\ comp \land \\ \qquad \neg\ is\_unconstrained\_tmark_\delta\ comp \bullet \\ \delta \vdash_{RFld} fs \Longrightarrow \delta_f, fields, types\end{array}}{\delta \vdash_{RFld} \langle \theta RecFldTypDef \rangle \frown fs \Longrightarrow \delta', fields', types'} \quad \text{(Fld2)}$$

where

$$fields' == \langle fld \rangle \frown fields$$

$$types' == types \cup \{\ fld \mapsto comp\ \}$$

$$\delta' == \delta_f[\ usedids := \delta_f.usedids \cup \{fld\}\ ]$$

References: *Env* p. 12; *RecFldTypDef* p. 37; $is\_visible\_tmark_\delta$ p. 281; $is\_unconstrained\_tmark_\delta$ p. 283

Chapter 4

Subtype Definitions

A subtype definition is the term in the syntax which defines a subtype from an existing type. For example:

subtype S **is** T *a subtype definition*

We refer to T as the *parent* type of S; the parent type is included in the Abstract Syntax of subtype definitions (*SubDef*). The components of *SubDef* are summarised in the following table:

Syntax Constructor	Description	Page
ran	range constraint on a scalar type	42
dig	floating point digits	44
digr	floating point digits, with range	45
dlt	fixed point accuracy	47
dltr	fixed point accuracy, with range	48
idx	array index constraint	50

The Well-Formation Predicate

The well-formation of a subtype definition is defined by a relation between the environment, an identifier which is the name of the new subtype, the subtype definition and a modified environment. The declaration of this relation is:

$$\_,\_ \vdash_{SubDef} \_ \Longrightarrow \_ \subseteq Env \times Id \times SubDef \times Env$$

Thus the well-formation predicate:

$$\delta, s \vdash_{SubDef} sdef \Longrightarrow \delta'$$

can be read as "subtype s, with definition *sdef*, is well-formed in environment δ yielding environment δ'".

4.1 Range Constraints

A subtype of any scalar (i.e. real or discrete) type can be defined by a range constraint.

Syntax Example	A.S. Representation
Colour **range** Orange .. Blue	$ran \langle\!\!\langle\ parent \mapsto id\ Colour,$ $lower \mapsto nam\ (simp\ Orange),$ $upper \mapsto nam\ (simp\ Blue)\ \rangle\!\!\rangle$

4.1.1 Abstract Syntax

Both the upper and lower bounds are expressions.

$$RangeSubDef \mathrel{\widehat=} [parent : IdDot;\ lower, upper : Exp]$$

$$SubDef ::= ran\langle\langle RangeSubDef\rangle\rangle \mid \ldots$$

4.1.2 Static Semantics

1. The parent type must be a visible scalar type.
2. The expressions ($lower$, $upper$) used to specify the range must be well-formed and static, with the types compatible with the parent.
3. The range of the subtype ($l\,..\,u$) must be non-empty and must not exceed the range of the parent type.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ s : Id;\ RangeSubDef;\\ \quad ltyp, utyp : ExpType;\ lval, uval : Val \mid\\ \qquad is\_visible\_tmark_\delta\ parent \wedge\\ \qquad is\_scalar\_tmark_\delta\ parent \wedge\\ \qquad ltyp\ compat_\delta\ parent \wedge\\ \qquad utyp\ compat_\delta\ parent \wedge\\ \qquad \varnothing \subset lval\ {}_{V..\delta}\ uval \subseteq scalar\_tmark\_range_\delta\ parent \bullet\\ \quad \delta \vdash_{Exp} lower : ltyp\\ \quad \delta \vdash_{Exp} upper : utyp\\ \quad \delta \vdash lower \Longrightarrow_{\text{StaticEval}} lval\\ \quad \delta \vdash upper \Longrightarrow_{\text{StaticEval}} uval\end{array}}{\delta, s \vdash_{SubDef} ran(\theta RangeSubDef) \Longrightarrow \delta'} \quad \text{(Ran)}$$

where

$$\delta' == \delta[\ types := \delta.types \cup \{s \mapsto ran\_typcon_\delta\ parent\ (lval\ {}_{V..\delta}\ uval)\},$$
$$subtypes := \delta.subtypes \cup \{s \mapsto ancestorof_\delta\ parent\}\ ]$$

The function ran_typcon_δ returns the type construction for the new subtype:

$$
\begin{array}{|l}
ran\_typcon_{Env} : IdDot \pfun \mathbb{P}\ Val \fun TypCon \\
\hline
\forall\, \delta : Env;\ p : IdDot;\ r : \mathbb{P}\ Val \mid is\_scalar\_tmark_\delta\ p \bullet \\
\quad (is\_int\_tmark_\delta\ p \Rightarrow \\
\qquad ran\_typcon_\delta\ p\ r = intT\ r) \land \\
\quad (is\_enum\_tmark_\delta\ p \Rightarrow \\
\qquad ran\_typcon_\delta\ p\ r = enumT\ r) \land \\
\quad (is\_float\_tmark_\delta\ p \Rightarrow \\
\qquad ran\_typcon_\delta\ p\ r = floatT\ (\mu\, FloatTypCon \mid \\
\qquad\quad range = r \land \\
\qquad\quad digits = float\_tmark\_digits_\delta\ p)) \land \\
\quad (is\_fixed\_tmark_\delta\ p \Rightarrow \\
\qquad ran\_typcon_\delta\ p\ r = fixedT\ (\mu\, FixedTypCon \mid \\
\qquad\quad range = r \land \\
\qquad\quad delta = fixed\_tmark\_delta_\delta\ p))
\end{array}
$$

References: *Env* p. 12; *ExpType* p. 73; *Val* p. 6; $is\_visible\_tmark_\delta$ p. 281; $is\_scalar\_tmark_\delta$ p. 284; $compat_\delta$ p. 76; $\mathcal{V}..._\delta$ p. 297; $scalar\_tmark\_range_\delta$ p. 296; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\text{StaticEval}}$ p. 298; $ancestorof_\delta$ p. 281; *TypCon* p. 15; $is\_int\_tmark_\delta$ p. 282; *intT* p. 15; $is\_enum\_tmark_\delta$ p. 282; *enumT* p. 15; $is\_float\_tmark_\delta$ p. 282; *floatT* p. 15; *FloatTypCon* p. 15; $float\_tmark\_digits_\delta$ p. 287; $is\_fixed\_tmark_\delta$ p. 283; *fixedT* p. 15; *FixedTypCon* p. 16; $fixed\_tmark\_delta_\delta$ p. 287

4.2 Floating Point Number of Digits

A floating point constraint can specify a number of digits.

Syntax Example	A.S. Representation
T **digits** 10	$dig \langle\!\langle\ parent \mapsto id\ T,\ digits \mapsto lint\ 10\ \rangle\!\rangle$

4.2.1 Abstract Syntax

The number of digits is specified by an expression.

$$DigSubDef \mathrel{\widehat=} [parent : IdDot;\ digits : Exp]$$

$$SubDef ::= \ldots \mid dig\langle\!\langle DigSubDef \rangle\!\rangle$$

4.2.2 Static Semantics

1. The parent type must be a visible floating point type.
2. The expression used to specify the number of digits must be of an integer type and must be static.
3. The number of digits specified for the subtype must be greater than zero and must not exceed the number of digits of the parent type.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ s : Id;\ DigSubDef;\ typ : ExpType;\ d : \mathbb{N} \mid \\ \quad is\_visible\_tmark_\delta\ parent \land \\ \quad is\_float\_tmark_\delta\ parent \land \\ \quad is\_int\_exp_\delta\ typ \land \\ \quad 0 < d \leq float\_tmark\_digits_\delta\ parent \bullet \\ \delta \vdash_{Exp} digits : typ \\ \delta \vdash digits \Longrightarrow_{\text{StaticEval}} intval\ d\end{array}}{\delta, s \vdash_{SubDef} dig(\theta DigSubDef) \Longrightarrow \delta'} \quad \text{(Dig)}$$

where

$$\begin{array}{l}\delta' == \delta[\ types := \delta.types \cup \{s \mapsto floatT\ (\mu\, FloatTypCon \mid \\ \qquad digits = d \land \\ \qquad range = scalar\_tmark\_range_\delta\ parent)\}, \\ \quad subtypes := \delta.subtypes \cup \{s \mapsto ancestorof_\delta\ parent\}\ ]\end{array}$$

References: *Env* p. 12; *ExpType* p. 73; $is\_visible\_tmark_\delta$ p. 281; $is\_float\_tmark_\delta$ p. 282; $is\_int\_exp_\delta$ p. 295; $float\_tmark\_digits_\delta$ p. 287; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\text{StaticEval}}$ p. 298; *intval* p. 6; *floatT* p. 15; *FloatTypCon* p. 15; $scalar\_tmark\_range_\delta$ p. 296; $ancestorof_\delta$ p. 281

4.3 Floating Point, Digits and Range

A floating point constraint can specify an allowed range for elements of the subtype, as well as a number of digits.

Syntax Example	A.S. Representation
K.T **digits** 8 **range** 0.0 .. 100.0	$digr \langle\!\| \begin{array}{l} parent \mapsto dot(K, T), \\ digits \mapsto lint\ 8, \\ lower \mapsto lreal\ 0.0, \\ upper \mapsto lreal\ 100.0 \end{array} \|\!\rangle$

4.3.1 Abstract Syntax

The number of digits, and the bounds of the range, are specified by expressions.

$DigRSubDef \mathrel{\widehat=} [parent : IdDot;\ digits, lower, upper : Exp]$

$SubDef ::= \ldots \mid digr\langle\!\langle DigRSubDef \rangle\!\rangle$

4.3.2 Static Semantics

1. The parent type must be a visible floating point type.

2. The expressions (*lower*, *upper*) used to specify the range must be well-formed and static, with the types compatible with the parent.

3. The range of the subtype must be non-empty and must not exceed the range of the parent type.

4. The expression used to specify the number of digits must be of an integer type and must be static.

5. The number of digits specified for the subtype must be greater than zero and must not exceed the number of digits of the parent type.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ s : Id;\ DigRSubDef;\ ltyp, utyp, dtyp : ExpType;\\ \quad lval, uval : Val;\ d : \mathbb{N} \mid\\ \qquad is\_visible\_tmark_\delta\ parent \land\\ \qquad is\_float\_tmark_\delta\ parent \land\\ \qquad ltyp\ compat_\delta\ parent \land\\ \qquad utyp\ compat_\delta\ parent \land\\ \qquad \varnothing \subset lval\ {}_{V..\delta}\ uval \subseteq scalar\_tmark\_range_\delta\ parent \land\\ \qquad is\_int\_exp_\delta\ dtyp \land\\ \qquad 0 < d \leq float\_tmark\_digits_\delta\ parent \bullet \qquad \text{(DigR)}\\ \\ \quad \delta \vdash_{Exp} lower : ltyp\\ \quad \delta \vdash_{Exp} upper : utyp\\ \quad \delta \vdash_{Exp} digits : dtyp\\ \quad \delta \vdash lower \Longrightarrow_{\text{StaticEval}} lval\\ \quad \delta \vdash upper \Longrightarrow_{\text{StaticEval}} uval\\ \quad \delta \vdash digits \Longrightarrow_{\text{StaticEval}} intval\ d\end{array}}{\delta, s \vdash_{SubDef} digr(\theta DigRSubDef) \Longrightarrow \delta'}$$

where

$$\begin{array}{l}\delta' == \delta[\ types := \delta.types \cup \{s \mapsto floatT\ (\mu\, FloatTypCon \mid\\ \qquad\qquad digits = d \land\\ \qquad\qquad range = lval\ {}_{V..\delta}\ uval)\},\\ \qquad\quad subtypes := \delta.subtypes \cup \{s \mapsto ancestorof_\delta\ parent\}\ ]\end{array}$$

Env p. 12; *ExpType* p. 73; *Val* p. 6; $is\_visible\_tmark_\delta$ p. 281; $is\_float\_tmark_\delta$ p. 282; $compat_\delta$ p. 76; ${}_{V..\delta}$ p. 297; $is\_int\_exp_\delta$ p. 295; $float\_tmark\_digits_\delta$ p. 287; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\text{StaticEval}}$ p. 298; *intval* p. 6; *floatT* p. 15; *FloatTypCon* p. 15; $scalar\_tmark\_range_\delta$ p. 296; $ancestorof_\delta$ p. 281

4.4 Fixed Point Accuracy

A fixed point constraint can specify a *delta*, which is the maximum permissible difference between adjacent values in the type.

Syntax Example	A.S. Representation
S **delta** 0.002	$dlt \langle\!\vert \; parent \mapsto id\ S,$ $delta \mapsto lreal\ 0.002$

4.4.1 Abstract Syntax

The delta value is specified by an expression.

$$DltSubDef \mathrel{\widehat=} [parent : IdDot;\ delta : Exp]$$

$$SubDef ::= \ldots \mid dlt\langle\!\langle DltSubDef \rangle\!\rangle$$

4.4.2 Static Semantics

1. The parent type must be a visible fixed point type.
2. The expression used to specify the accuracy must be of a real type and must be static.
3. The subtype must not have a greater accuracy than the parent type (i.e. the delta may not be decreased).

$$\frac{\begin{array}{l}\forall \delta : Env;\ s : Id;\ DltSubDef;\ typ : ExpType;\ acc : \mathrm{Real} \bullet \\ \quad is\_visible\_tmark_\delta\ parent \land \\ \quad is\_fixed\_tmark_\delta\ parent \land \\ \quad is\_real\_exp_\delta\ typ \land \\ \quad fixed\_tmark\_delta_\delta\ parent \leq acc \\ \delta \vdash_{Exp} delta : typ \\ \delta \vdash delta \Longrightarrow_{\mathrm{StaticEval}} realval\ acc\end{array}}{\delta, s \vdash_{SubDef} dlt(\theta DltSubDef) \Longrightarrow \delta'} \quad \text{(Dlt)}$$

where

$$\begin{array}{l}\delta' == \delta[\ types := \delta.types \cup \{s \mapsto fixedT\ (\mu\, FixedTypCon \mid \\ \qquad delta = acc \land \\ \qquad range = scalar\_tmark\_range_\delta\ parent)\}, \\ \quad subtypes := \delta.subtypes \cup \{s \mapsto ancestorof_\delta\ parent\}\ ]\end{array}$$

References: *Env* p. 12; *ExpType* p. 73; *is_visible_tmark*$_\delta$ p. 281; *is_fixed_tmark*$_\delta$ p. 283; *is_real_exp*$_\delta$ p. 295; *fixed_tmark_delta*$_\delta$ p. 287; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\mathrm{StaticEval}}$ p. 298; *realval* p. 6; *fixedT* p. 15; *FixedTypCon* p. 16; *scalar_tmark_range*$_\delta$ p. 296; *ancestorof*$_\delta$ p. 281

4.5 Fixed Point Accuracy and Range

A fixed point constraint may specify a range, as well as a delta value.

Syntax Example	A.S. Representation
S **delta** 0.002 **range** 10.0 .. 20.0	$dltr \langle\!\langle\ parent \mapsto id\ S,$ $delta \mapsto lreal\ 0.002,$ $lower \mapsto lreal\ 10.0,$ $upper \mapsto lreal\ 20.0\ \rangle\!\rangle$

4.5.1 Abstract Syntax

The delta value and both bounds of the range are specified by expressions.

$$DltRSubDef \mathrel{\hat{=}} [parent : IdDot;\ delta, lower, upper : Exp]$$

$$SubDef ::= \ldots \mid dltr\langle\langle DltRSubDef \rangle\rangle$$

4.5.2 Static Semantics

1. The parent type must be a visible fixed point type.

2. The expression used to specify the accuracy must be of a real type and must be static.

3. The subtype must not have a greater accuracy than the parent type (i.e. the delta may not be decreased).

4. The expressions ($lower$, $upper$) used to specify the range must be well-formed and static, with the types compatible with the parent.

5. The range of the subtype ($l\,..\,u$) must be non-empty and must not exceed the range of the parent type.

$$\begin{array}{l}
\forall\, \delta : Env;\ s : Id;\ DltRSubDef;\ atyp, ltyp, utyp : ExpType; \\
\quad acc : \mathrm{Real};\ lval, uval : Val \bullet \\
\qquad is_visible_tmark_{\delta}\ parent \wedge \\
\qquad is_fixed_tmark_{\delta}\ parent \wedge \\
\qquad is_real_exp_{\delta}\ typ \wedge \\
\qquad fixed_tmark_delta_{\delta}\ parent \leq acc \wedge \\
\qquad ltyp\ compat_{\delta}\ parent \wedge \\
\qquad utyp\ compat_{\delta}\ parent \wedge \\
\qquad \varnothing \subset lval\ {}_{V..\delta}\ uval \subseteq scalar_tmark_range_{\delta}\ parent \bullet \qquad \text{(DltR)} \\
\quad \delta \vdash_{Exp} delta : atyp \\
\quad \delta \vdash delta \Longrightarrow_{\mathrm{StaticEval}} realval\ acc \\
\quad \delta \vdash_{Exp} lower : ltyp \\
\quad \delta \vdash_{Exp} upper : utyp \\
\quad \delta \vdash lower \Longrightarrow_{\mathrm{StaticEval}} lval \\
\quad \delta \vdash upper \Longrightarrow_{\mathrm{StaticEval}} uval \\
\hline
\quad \delta, s \vdash_{SubDef} dltr(\theta DltRSubDef) \Longrightarrow \delta'
\end{array}$$

where

$$\begin{array}{l}
\delta' == \delta[\ types := \delta.types \cup \{s \mapsto fixedT\ (\mu\, FixedTypCon \mid \\
\qquad\qquad delta = acc \wedge \\
\qquad\qquad range = (lval\ {}_{V..\delta}\ uval))\}, \\
\qquad subtypes := \delta.subtypes \cup \{s \mapsto ancestorof_{\delta}\ parent\}\ ]
\end{array}$$

References: *Env* p. 12; *ExpType* p. 73; *Val* p. 6; *is_visible_tmark*$_{\delta}$ p. 281; *is_fixed_tmark*$_{\delta}$ p. 283; *is_real_exp*$_{\delta}$ p. 295; *fixed_tmark_delta*$_{\delta}$ p. 287; *compat*$_{\delta}$ p. 76; $_{V..\delta}$ p. 297; *scalar_tmark_range*$_{\delta}$ p. 296; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\mathrm{StaticEval}}$ p. 298; *realval* p. 6; *fixedT* p. 15; *FixedTypCon* p. 16; *ancestorof*$_{\delta}$ p. 281

4.6 Array Index Constraint

An unconstrained array is "constrained" using an array index constraint. An index type is specified for each dimension of the array.

Syntax Example	A.S. Representation
T(OnetoTen)	$idx \langle\!\langle\ parent \mapsto id\ T,\ idxcons \mapsto \langle id\ OnetoTen \rangle\ \rangle\!\rangle$

4.6.1 Abstract Syntax

Each index constraint is a type name, from *IdDot*.

$$IdxSubDef \mathrel{\widehat=} [parent : IdDot;\ idxcons : seq_1 IdDot]$$

$$SubDef ::= \ldots \mid idx\langle\!\langle IdxSubDef \rangle\!\rangle$$

4.6.2 Static Semantics

1. The *parent* typemark must be a visible unconstrained array type.

2. There must be a visible typemark in the list *idxcons* for each index type of the parent.

3. Each index type must match the corresponding index type of the parent:

   (a) The ancestor type must be the same.

   (b) The range of the new index type must not exceed the range of the index type of the unconstrained array.

$$\frac{\begin{array}{l}\forall \delta : Env;\ s : Id;\ IdxSubDef \mid \\ \quad is\_visible\_tmark_\delta\ parent \land \\ \quad is\_unconstrained\_tmark_\delta\ parent \land \\ \quad (\forall t : \mathrm{ran}\ idxcons \bullet is\_visible\_tmark_\delta\ t) \land \\ \quad ancestorof_\delta \circ idxcons = \\ \qquad ancestorof_\delta \circ (array\_index\_tmark_\delta\ parent) \land \\ \quad (\forall i : \mathrm{dom}\ idxcons \bullet \\ \qquad scalar\_tmark\_range_\delta(idxcons\ i) \subseteq \\ \qquad\quad scalar\_tmark\_range_\delta(array\_index\_tmark_\delta\ parent\ i)) \bullet \end{array}}{\delta, s \vdash_{SubDef} idx(\theta IdxSubDef) \Longrightarrow \delta'} \quad \text{(Idx)}$$

where

$$\begin{aligned}\delta' == \delta[\ & types := \delta.types \cup \{s \mapsto arrT\ (\mu\, ArrTypCon \mid \\ & \quad indexes = idxcons \wedge \\ & \quad component = array\_comp\_tmark_\delta\ parent)\}, \\ & subtypes := \delta.subtypes \cup \{s \mapsto ancestorof_\delta\ parent\}\ ]\end{aligned}$$

References: Env p. 12; $is\_visible\_tmark_\delta$ p. 281; $is\_unconstrained\_tmark_\delta$ p. 283; $ancestorof_\delta$ p. 281; $array\_index\_tmark_\delta$ p. 289; $scalar\_tmark\_range_\delta$ p. 296; $arrT$ p. 15; *ArrTypCon* p. 16; $array\_comp\_tmark_\delta$ p. 289

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	55
slct	selected name	58
pasc	positional association - array or function call	62
nasc	named association - function call	67

In the rest of the chapter there is one section for each component of the syntax, preceded by Section 5.1 which defines the set of type names *NameType*.

Well-Formation Predicate

The well-formation of a name is defined by a relation between the type of the name and the environment. The declaration of this relation is:

$$\_ \vdash_{Name} \_ : \_ \subseteq Env \times Name \times NameType$$

Thus the following well-formation predicate

$$\delta \vdash_{Name} name : type$$

can be read as "Name *name* is well-formed in environment δ with type *type*".

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.

5.1 Name Types

A name is given a *type* just as an expression is. However, the type of a name is different from the type of an expression, because there are some terms which are valid as names but not as expressions. For example, the name of a function is not a valid expression without a list of actual parameters. The following terms in SPARK are described by names:

Term	Type	Parameter of Type
Constant	*objnam*	Constant type or subtype, read-only access
Type	*typnam*	The type or subtype name
Variable	*objnam*	Variable type or subtype, and access modes
Function	*funnam*	The function name
Function call	*objnam*	Function return type or subtype, read-only access
Package	*paknam*	The package identifier

where a variable can be a simple variable or an element of an object of a composite type.

$$\begin{aligned} NameType ::= & \; objnam\langle\langle ObjDict\rangle\rangle \mid typnam\langle\langle IdDot\rangle\rangle \\ \mid & \; funnam\langle\langle IdDot\rangle\rangle \mid paknam\langle\langle Id\rangle\rangle \end{aligned}$$

Uses of Names It is important to note that not every use of these terms in SPARK is a name. Thus the term which specifies the type of a formal parameter does not appear in the Abstract Syntax as a *Name*, but as an *IdDot*. *Name* is only used in contexts where more than one of the syntactic forms of a name could be valid, in particular on the left hand side of an assignment statement.

Objects A name may stand for an object, which has a value. Objects are distinguished by their type (*IdDot*), the access modes (read and write) in which they can be used and the *name* of the variable updated by an assignment to the object. The type, mode and name are represented by a value of *ObjDict*:

$$ObjDict \mathrel{\widehat=} [type : IdDot;\ modes : \mathbb{P}\ RdWr;\ name : IdDot]$$

There are many read-only objects — constants, enumeration literals, function calls — but write-only objects also exist. The **global** and **derives** annotations restrict the access modes in which a variable can be used to be a subset of those with which it was originally declared.

Not all writeable *objnam* can be *assigned* since there are some restrictions on the types of objects which can be used in assignment. For example, an object of limited private type cannot be assigned, even though it can be updated by a subprogram call.

Subtypes Subtype ranges are checked when a value is assigned to an object. It is necessary, therefore, to include the distinction between different subtypes in the type of the name. To achieve this, the *IdDot* parameter of *typnam* and the *type* component of the *ObjDict* parameter of *objnam* identify either a type or a subtype.

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 Static Semantics

Constant The identifier (*con*) can be a constant or an enumeration literal.

1. An identifier declared as a constant (possibly implicitly as part of an enumeration type definition) is a well-formed simple name, forming an object. A deferred constant cannot be referenced [AARM, §7.4.3, ¶2].
2. The type of the object is the type used in the constant declaration.
3. The object is read-only.

$$\frac{\begin{array}{l}\forall \delta : Env;\ con : Id;\ ObjDict \mid \\ \qquad type = \delta.contypes\ con \land \\ \qquad con \notin \delta.defcons \land \\ \qquad modes = \{rd\} \land \\ \qquad name = id\ con \bullet\end{array}}{\delta \vdash_{Name} simp\ con : objnam\ (\theta\, ObjDict)} \quad \text{(Simp1)}$$

Type The identifier can be a type.

1. An identifier declared as a type or a subtype is a well-formed simple name.

$$\frac{\begin{array}{l}\forall \delta : Env;\ typ : Id \mid \\ \qquad typ \in \mathrm{dom}\, \delta.types \bullet\end{array}}{\delta \vdash_{Name} simp\ typ : typnam\ (id\ typ)} \quad \text{(Simp2)}$$

Variable The identifier can be the name of a variable.

1. An identifier declared as a variable is a well-formed simple name.

2. The type of the object is the (sub)type of the variable.

3. The access modes of the object are the access modes of the variable in the current environment.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ var : Id;\ ObjDict \mid \\ \qquad var \in \operatorname{dom} \delta.vartypes \wedge \\ \qquad type = \delta.vartypes\ var \wedge \\ \qquad modes = \{\ m : RdWr \mid (var, m) \in \delta.varmodes\ \} \wedge \\ \qquad name = id\ var \bullet\end{array}}{\delta \vdash_{Name} simp\ var : objnam\ (\theta\, ObjDict)} \quad \text{(Simp3)}$$

Function The identifier can be the name of a function.

1. A callable function with no parameters is a well-formed simple name representing an object.

2. The object is read-only; its type is the return type of the function.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ fun : Id;\ ObjDict \mid \\ \qquad (\delta.fdecls\ fun).ftypes = \varnothing \wedge \\ \qquad fun \in \delta.scall \wedge \\ \qquad type = (\delta.fdecls\ fun).rtype \wedge \\ \qquad modes = \{rd\} \wedge \\ \qquad name = id\ fun \bullet\end{array}}{\delta \vdash_{Name} simp\ fun : objnam\ (\theta\, ObjDict)} \quad \text{(Simp4)}$$

3. A callable function which requires actual parameters is a well-formed simple name.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ fun : Id \mid \\ \qquad fun \in \delta.scall \wedge \\ \qquad (\delta.fdecls\ fun).ftypes \neq \varnothing \bullet\end{array}}{\delta \vdash_{Name} simp\ fun : funnam\ (id\ fun)} \quad \text{(Simp5)}$$

Package The identifier can be a package.

1. An identifier in the with list in the current environment is a well-formed simple name.

$$\frac{\forall\, \delta : Env;\ pak : Id \mid \\ \qquad pak \in \delta.withs \bullet}{\delta \vdash_{Name} simp\ pak : paknam\ pak} \qquad \text{(Simp6)}$$

References: Env p. 12; *ObjDict* p. 54

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	$slct \langle\!\lVert\ prefix \mapsto simp\ K,\ selector \mapsto T\ \rVert\!\rangle$
K.R.F	$slct \langle\!\lVert\ prefix \mapsto slct \langle\!\lVert\ prefix \mapsto simp\ K,\ selector \mapsto R\ \rVert\!\rangle,\ selector \mapsto F\ \rVert\!\rangle$

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 Static Semantics

The prefix can be a package name or an object name. An object, a type or a function can be selected from a package; only an field can be selected from an object, returning what we shall regard as a component "object". (Here, the term object is being used in a looser sense than that generally employed in Ada: we regard an object as a constant or variable, or a component field selected from one of these.)

Selecting a Constant from a Package

1. The prefix must be a well-formed package name.

2. The selector must be a visible constant or enumeration literal in this package.

3. The selected name is an object whose subtype is the type used in the constant declaration.

4. The object is read-only.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SlctName;\ pak : Id;\ ObjDict \mid \\ \qquad name = dot\ (pak, selector) \wedge \\ \qquad is\_visible\_const_{\delta}\ name \wedge \\ \qquad type = const\_tmark_{\delta}\ name \wedge \\ \qquad modes = \{rd\} \bullet \\ \delta \vdash_{Name} prefix : paknam\ pak\end{array}}{\delta \vdash_{Name} slct(\theta SlctName) : objnam\ (\theta\, ObjDict)} \quad \text{(Slct1)}$$

Selecting a Type from a Package

1. The prefix must be a well-formed package name.
2. The selector must be a visible type or subtype in this package.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SlctName;\ pak : Id \mid \\ \qquad is\_visible\_tmark_{\delta}\ dot(pak, selector) \bullet \\ \delta \vdash_{Name} prefix : paknam\ pak\end{array}}{\delta \vdash_{Name} slct(\theta SlctName) : typnam\ (dot\ (pak, selector))} \quad \text{(Slct2)}$$

Selecting a Variable from a Package

1. The prefix must be a well-formed package name.
2. The selector must be a variable in this package.
3. The selected name is an object, whose subtype is the (sub)type of the variable.
4. The access modes of the object are the access modes of the variable in the current environment.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SlctName;\ pak : Id;\ ObjDict \mid \\ \qquad name = dot\ (pak, selector) \wedge \\ \qquad is\_visible\_var_{\delta}\ name \wedge \\ \qquad type = var\_tmark_{\delta}\ name \wedge \\ \qquad modes = var\_modes_{\delta}\ name \bullet \\ \delta \vdash_{Name} prefix : paknam\ pak\end{array}}{\delta \vdash_{Name} slct(\theta SlctName) : objnam\ (\theta\, ObjDict)} \quad \text{(Slct3)}$$

Selecting a Function from a Package There are two cases depending on whether or not the function has parameters.

1. The prefix must be a well-formed package name.

2. The selector may be a visible, callable function with no parameters: the selected name is a read-only object; its type is the return type of the function.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SlctName;\ pak : Id;\ ObjDict \mid \\ \qquad name = dot\ (pak, selector) \wedge \\ \qquad is\_callable\_fun_{\delta}\ name \wedge \\ \qquad fun\_param\_ids_{\delta}\ name = \varnothing \wedge \\ \qquad type = fun\_rtn\_tmark_{\delta} \wedge \\ \qquad modes = \{rd\} \bullet \\ \delta \vdash_{Name} prefix : paknam\ pak\end{array}}{\delta \vdash_{Name} slct(\theta SlctName) : objnam\ (\theta ObjDict)} \quad \text{(Slct4)}$$

3. The selector may be a visible, callable function requiring actual parameters: the selected name is a function name.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SlctName;\ pak : Id \mid \\ \qquad is\_callable\_fun_{\delta}\ dot(pak, selector) \wedge \\ \qquad fun\_param\_ids_{\delta}\ dot(pak, selector) \neq \varnothing \bullet \\ \delta \vdash_{Name} prefix : paknam\ pak\end{array}}{\delta \vdash_{Name} slct(\theta SlctName) : funnam\ dot(pak, selector)} \quad \text{(Slct5)}$$

Selection from an Object

1. The prefix must be a well-formed named, identifying the object obj_{rec}, which must have a record type.

2. The selector must be a field of the record.

3. The selected name identifies an object obj_{fld}, representing a field of the record. The type of the obj_{fld} object is the type of the field.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SlctName;\ obj_{rec}, obj_{fld} : ObjDict \mid \\ \qquad is\_rec\_tmark_{\delta}\ obj_{rec}.type \wedge \\ \qquad obj_{fld}.type = rec\_field\_tmark_{\delta}\ obj_{rec}.type\ selector \wedge \\ \qquad obj_{fld}.modes = obj_{rec}.modes \bullet \\ \\ \delta \vdash_{Name} prefix : objnam\ obj_{rec}\end{array}}{\delta \vdash_{Name} slct(\theta SlctName) : objnam\ obj_{fld}} \quad \text{(Slct6)}$$

References: *Env* p. 12; $is\_visible\_tmark_{\delta}$ p. 279; $const\_tmark_{\delta}$ p. 279; $is\_visible\_var_{\delta}$ p. 280; var_tmark_{δ} p. 280; var_modes_{δ} p. 280; $is\_callable\_fun_{\delta}$ p. 291; $fun\_param\_ids_{\delta}$ p. 291; *ObjDict* p. 54 $fun\_rtn\_tmark_{\delta}$ p. 292; $is\_rec\_tmark_{\delta}$ p. 284; $rec\_field\_tmark_{\delta}$ p. 288

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 Static Semantics

Array Element A *pasc* term can be well-formed as an "indexed expression", if:

1. The prefix is a well-formed name, identifying an object (obj_{arr}) of array type; this includes any object declared using a constrained or unconstrained array type, or an array subtype.

2. The expression-list argument *args* is a well-formed positional association (see below).

3. The type of the indexed object obj_{elem} is the component type of the array.

4. The indexed object has the same access modes as the array object.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PAscName;\ obj_{arr}, obj_{elem} : ObjDict \mid \\ \qquad is\_arr\_tmark_{\delta}\ obj_{arr}.type \wedge \\ \qquad obj_{elem}.type = array\_comp\_tmark_{\delta}\ obj_{arr}.type \wedge \\ \qquad obj_{elem}.modes = obj_{arr}.modes \wedge \\ \qquad obj_{elem}.name = obj_{arr}.name \bullet \\ \delta \vdash_{Name} prefix : objnam\ obj_{arr} \\ \delta, array\_index\_tmark_{\delta}\ obj_{arr}.type \vdash_{PAssoc} args\end{array}}{\delta \vdash_{Name} pasc\ (\theta PAscName) : objnam\ obj_{elem}} \quad \text{(PAsc1)}$$

The function $array\_index\_tmark_{\delta}$ returns the sequence of index types of an array type, and $array\_comp\_tmark_{\delta}$ returns the component type.

Function Call A *pasc* term is well-formed as a "function call" if:

1. The prefix is a well-formed function name.

2. The selector is a list of expressions which, converted to the named-association form using the formal parameter identifiers of the function, is a well-formed.

3. The name is a read-only object *obj*, with the return type of the function.

The parameters of a SPARK function cannot be aliased since it has no *side-effects* — neither the global variables nor the actual parameters can be updated.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PAscName;\ ObjDict \mid \\ \qquad is\_callable\_fun_{\delta}\ name \wedge \\ \qquad type = fun\_rtn\_tmark_{\delta}\ name \wedge \\ \qquad modes = \{rd\} \bullet \\ \delta \vdash_{Name} prefix : funnam\ name \\ \delta, fun\_param\_tmarks_{\delta} \vdash_{NAssoc} args'\end{array}}{\delta \vdash_{Name} pasc\ (\theta PAscName) : objnam\ (\theta ObjDict)} \quad \text{(PAsc2)}$$

where

$$args' == (\lambda\, i : \mathrm{dom}\ args \bullet (\mu\, NamedActual \mid \\ \qquad formal = fun\_param\_ids_{\delta}\ i \wedge \\ \qquad actual = args\ i))$$

Type Conversion A *pasc* term with a single argument may be a type conversion. There are two possible type conversions in SPARK.

An expression of any numeric type can be converted to any other numeric type.

1. The prefix must be a well-formed type name (the target type *type*) — identifying a numeric type.

2. The association must contain a single well-formed expression, with a numeric type.

3. The association is a read-only object of the target type.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PAscName;\ arg : Exp;\ ObjDict;\ etyp : ExpType \mid \\ \quad args = \langle arg \rangle \wedge \\ \quad is\_numeric\_exp_{\delta}\ etyp \wedge \\ \quad is\_numeric\_tmark_{\delta}\ type \wedge \\ \quad mode = \{rd\} \wedge \\ \quad name = type \bullet \\ \delta \vdash_{Name} prefix : typnam\ type \\ \delta \vdash_{Exp} arg : etyp\end{array}}{\delta \vdash_{Name} pasc\ (\theta PAscName) : objnam\ (\theta ObjDict)} \quad \text{(PAsc3)}$$

An expression denoting an array may be convertible to another array type.

1. The prefix must be a well-formed type name (the target type *ttyp*) — identifying an array type, which is not unconstrained.

2. The association must contain a single well-formed expression, of an array type (the converted type *ctyp*).

3. The target and converted types must have the same number of dimensions. The corresponding index types must be the same or convertible; the corresponding index ranges must be equal.

4. The components types of the target and converted types must be the same.

5. The association is a read-only object of the target type.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PAscName;\ ObjDict;\ ctyp, ttyp : IdDot \mid \\ \quad is\_arr\_tmark_\delta\ ttyp \land \\ \quad \neg\ is\_unconstrained\_tmark_\delta\ ttyp \land \\ \quad is\_arr\_tmark_\delta\ ctyp \land \\ \quad array\_arity_\delta\ ttyp = array\_arity_\delta\ ctyp \land \\ \quad (\forall\, i : 1..\, array\_arity_\delta\ ttyp;\ ti, ci : IdDot \mid \\ \qquad ti = array\_index\_tmark_\delta\ ttypi \land \\ \qquad ci = array\_index\_tmark_\delta\ ctypi \bullet \\ \qquad (ti = ci \lor (is\_int\_tmark_\delta\ ti \land is\_int\_tmark_\delta\ ci)) \land \\ \qquad scalar\_tmark\_range_\delta\ ti = scalar\_tmark\_range_\delta\ ci) \land \\ \quad ancestorof_\delta\ array\_comp\_tmark_\delta\ ttyp = \\ \qquad ancestorof_\delta\ array\_comp\_tmark_\delta\ ctyp \land \\ \quad type = ttyp \land \\ \quad mode = \{rd\} \land \\ \quad name = ttyp \bullet \\ \\ \delta \vdash_{Name} prefix : typnam\ ttyp \\ \delta \vdash_{Exp} arg : valueT\ \{nameT\ ctyp\}\end{array}}{\delta \vdash_{Name} pasc\ (\theta PAscName) : (\theta ObjDict)} \qquad \text{(PAsc4)}$$

References: *Env* p. 12; *ObjDict* p. 54; *A is_arr_tmark*$_\delta$ p. 283; *array_comp_tmark*$_\delta$ p. 289; *array_index_tmark*$_\delta$ p. 289; *is_callable_fun*$_\delta$ p. 291; *fun_rtn_tmark*$_\delta$ p. 292; *fun_param_tmarks*$_\delta$ p. 291; $\vdash_{NAssoc}$ p. 68; *fun_param_ids*$_\delta$ p. 291; *NamedActual* p. 178; *is_numeric_exp*$_\delta$ p. 295; *is_numeric_tmark*$_\delta$ p. 285; *ExpType* p. 73; *is_unconstrained_tmark*$_\delta$ p. 283; *array_arity*$_\delta$ p. 288; *is_int_tmark*$_\delta$ p. 282; *scalar_tmark_range*$_\delta$ p. 296; *ancestorof*$_\delta$ p. 281

5.4.3 Positional Association Lists

A positional association contains a list of expressions. When the association is an array element, the well-formation of this list of expressions depends on the current environment and a list of "index" typemarks. The well-formation relation is declared as follows:

$$\_, \_ \vdash_{PAssoc} \_ \subseteq Env \times \mathrm{seq}\, IdDot \times \mathrm{seq}\, Exp$$

The empty list of expressions is a well-formed association if the list of types is empty.

$$\frac{\forall\, \delta : Env \bullet}{\delta, \langle\rangle \vdash_{PAssoc} \langle\rangle} \qquad \text{(PAssoc1)}$$

A non-empty list of expressions is a well-formed association if:

1. the first expression is a well-formed expression, whose type is compatible with the first type in the list,
2. the rest of the list is a well-formed association.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : IdDot;\ ts : \mathrm{seq}\, IdDot;\ arg : Exp;\ args : \mathrm{seq}\, Exp; \\ \quad etype : ExpType \mid \\ \qquad etype\ compat_\delta\ t \bullet \\ \quad \delta, ts \vdash_{PAssoc} args \\ \quad \delta \vdash_{Exp} arg : etype\end{array}}{\delta, \langle t \rangle \frown ts \vdash_{PAssoc} \langle arg \rangle \frown args} \quad \text{(PAssoc2)}$$

References: *ExpType* p. 73; $compat_\delta$ p. 76

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* ⟪	*prefix* ↦ *simp f* *args* ↦ ⟨ ⟪ *formal* ↦ *arg*1, *actual* ↦ ... ⟫, ⟪ *formal* ↦ *arg*2, *actual* ↦ ... ⟫ ⟩

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*).

NAscName

prefix : *Name*
args : seq_1 *NamedActual*

Name ::= ... | *nasc*⟪*NAscName*⟫

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 Static Semantics

1. The prefix must be a well-formed function name.

2. The actual parameter list must be well-formed with the types declared for the parameters of the function.

3. The named association is a read-only object with the return type of the function.

The parameters of a SPARK function cannot be aliased since it has no *side-effects* — neither the global variables nor the actual parameters can be updated.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NAscName;\ ObjDict \mid \\ \qquad is\_callable\_fun_\delta\ name \land \\ \qquad type = fun\_rtn\_tmark_\delta\ name \land \\ \qquad modes = \{rd\} \bullet \\ \delta \vdash_{Name} prefix : funnam\ name \\ \delta, fun\_param\_tmarks_\delta\ name \vdash_{NAssoc} args\end{array}}{\delta \vdash_{Name} nasc\ (\theta NAscName) : objnam\ (\theta ObjDict)} \quad \text{(NAsc)}$$

References: *Name* p. 53; *NamedActual* p. 178; *Env* p. 12; *ObjDict* p. 54; *is_callable_fun*$_\delta$ p. 291; *fun_rtn_tmark*$_\delta$ p. 292; *fun_param_tmarks*$_\delta$ p. 291; $\vdash_{NAssoc}$ p. 178

5.5.3 Named Association List

The named association contains a list of actual parameters, each associating a formal parameter with an expression. The well-formation of this list depends on the current environment and a mapping from each formal parameter identifier to its typemark. The well-formation relation is declared as follows:

$$\_, \_ \vdash_{NAssoc} \_ \subseteq Env \times (Id \pfun IdDot) \times \mathrm{seq}\ NamedActual$$

It is necessary to distinguish the use of names and expressions as actual parameters.

Empty Parameter List The empty list of parameters is a well-formed association if map of parameter types is empty.

$$\frac{\forall\, \delta : Env \bullet}{\delta, \varnothing \vdash_{NAssoc} \langle\rangle} \quad \text{(NAssoc1)}$$

Name Parameter A parameter may be a name.

1. The formal parameter exists.
2. The name must be well-formed and refer to an object.
3. The name can be read.
4. The type of the name is compatible (*nam_compat*$_\delta$) with the type mark of the the formal parameter.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NamedActual;\ as : \operatorname{seq} NamedActual;\ ObjDict;\ n : Name;\\ \quad ft : Id \pfun IdDot \mid\\ \qquad formal \in \operatorname{dom} ft \land\\ \qquad actual = nam\ n \land\\ \qquad rd \in mode \land\\ \qquad type\ nam\_compat_{\delta}\ (ft\ formal) \bullet\\ \quad \delta \vdash_{Name} n : objnam\ (\theta ObjDict)\\ \quad \delta, \{formal\} \ndres ft \vdash_{NAssoc} as\end{array}}{\delta, ft \vdash_{NAssoc} \langle \theta NamedActual \rangle \frown as} \quad \text{(NAs-}$$

soc2)

Note that *type* and *mode* are the element of *ObjDict*.

Expression Parameter An formal parameter may be an expression other than a name.

1. The formal parameter exists.
2. The actual parameter is a well-formed expression which is not a name.
3. The type of the actual parameter *etyp* must be compatible with the type mark of the corresponding formal parameter.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NamedActual;\ as : \operatorname{seq} NamedActual;\ etyp : ExpType;\\ \quad ft : Id \pfun IdDot \mid\\ \qquad formal \in \operatorname{dom} ft \land\\ \qquad actual \notin \operatorname{ran} nam \land\\ \qquad etyp\ compat_{\delta}\ (ft\ formal) \bullet\\ \quad \delta \vdash_{Exp} actual : etyp\\ \quad \delta, \{formal\} \ndres ft \vdash_{NAssoc} as\end{array}}{\delta, ft \vdash_{NAssoc} \langle \theta NamedActual \rangle \frown as} \quad \text{(NAssoc3)}$$

The difference between this rule and NAssoc2 derives from the two different forms of type compatibility. Name type-compatibility nam_compat_{δ} allows the use of objects of unconstrained array types; these objects are not well-formed expressions.
References: *Env* p. 12; *NamedActual* p. 178; *ObjDict* p. 54; *ExpType* p. 73; nam_compat_{δ} p. 182; $compat_{\delta}$ p. 76

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.

Well-Formation Predicate

The well-formation of an expression is specified by a relation between the environment, the expression and its type. The declaration of this relation is:

$$\_ \vdash_{Exp} \_ : \_ \subseteq Env \times Exp \times ExpType$$

The set of expression types (*ExpType*) is defined in Section 6.1. Thus a well-formation predicate

$$\delta \vdash_{Exp} exp : type$$

can be read as "expression *exp* is well-formed in environment δ with type *type*".

Syntax Constructor	Description	Page
lint	integer literal	78
lreal	real literal	79
lchar	character literal	80
lstrg	string literal	81
nam	name	82
dots	range expression	84
in	membership test	85
notin	complement membership test	86
qual	type qualification	87
pagg	positional association aggregate	88
paggoth	...aggregate with others	91
nagg	named association aggregate	93
naggoth	...aggregate with others	98
un	unary operator expressions	100
bin	binary operator expressions	102
andthen	short circuit conjunction	105
orelse	short circuit disjunction	106
cat	string concatenation	107
—	type conversion	108

Table 6.1: Abstract Syntax of Expressions *Exp*

6.1 Expression Types

All expressions in SPARK have a type. This section specifies the set *ExpType* of possible expression types.

An expression in SPARK denotes either a range or a value, whose type is taken from a set of basic types (*BasicType*). There are two special cases — universal fixed point and subtypes of unconstrained arrays.

$$\begin{aligned} ExpType ::= & \; rangeT\langle\langle \mathbb{P}\, BasicType \rangle\rangle \\ | & \; valueT\langle\langle \mathbb{P}\, BasicType \rangle\rangle \\ | & \; ufixT \\ | & \; uarrvalT\langle\langle UArrValT \rangle\rangle \end{aligned}$$

6.1.1 Basic Types

The basic types are the universal types and named types.

$$BasicType ::= uintT \mid urealT \mid nameT\langle\langle IdDot \rangle\rangle$$

Named Types The basis of type compatibility in SPARK is *named equivalence*. Therefore the type of an expression is described by the name (*IdDot*) of a type or a subtype. Thus if **v** is a variable of type **t**, a reference to the variable is an expression with type *nameT* (*id t*). SPARK requires all types used by the programmer to be named explicitly; there are no anonymous types.

Universal Types Numeric literals cannot be assigned a particular numeric type. In Ada, this aspect of the type system is described using *universal* integer and real types, held to be compatible with any integer and real type respectively. A different approach is used to describe types in SPARK (see the next section) but universal types are still required.

6.1.2 Non-unique Types

A unique type cannot be assigned to a numeric literal, nor to an expression composed entirely of literals; rather their type is determined from the *context* in which the expression is used. Some examples of this, for the expression 1 + 2 are:

Context	Type of 1 + 2
T'(1 + 2)	T
T'RANGE(1 + 2)	universal integer
V ** (1 + 2)	predefined INTEGER

One approach would be to determine the context in which an expression is used before checking that it can take the type required. However, we prefer to associate a *set of basic types* with any expression. The set contains all the types which could be assigned to the expression; the expression is well-typed in a particular context if the required type is in the set. This approach allows a type to be calculated for each term, without considering the context in which it is being used.

Universal integer and real types are still required:

1. In some contexts an expression must be of universal type. (Essentially, this requirement is used as a *device* in the definition of Ada to restrict an expression to being composed of literals and named numbers only. Although the universal integer type is "compatible" with all named integer types, there still exist contexts where only one of this set of "compatible" types is acceptable.)

2. There are predefined operators for the universal types.

6.1.3 Ranges and Values

There is no range (or set) type constructor in SPARK which would allow the declaration of a variable holding, for example, a range of integers as its value. However, expressions whose value is a range of values can occur, in three ways:

1. A range, written A .. B, occurring in a membership test or an aggregate choice. See Section 6.7. (The same syntax is used in type and subtype definitions, but in these cases the Abstract Syntax contains a pair of expressions.)

2. A RANGE attribute. See Section 7.1.

3. When the name of a type is used as an expression, it stands for the range of values in the type (or subtype). See Section 6.6.

6.1.4 Unconstrained Array

An unconstrained array type is an array with an index range, and therefore a size, which is not fixed (see Section 3.9). The only objects which can be declared using an unconstrained array type are formal parameters of subprograms. These objects can never be used as expressions in SPARK (though they can be used as names, thus permitting indexing, for example).

To *constrain* the size of such an array a *subtype* is declared, giving the actual index type (see Section 4.6). We refer to these subtypes as *unconstrained-array subtypes*. Objects of unconstrained-array subtypes may be used as expressions in SPARK but they are subject to special type rules.

Implicit Subtype Conversion In Ada, two objects of constrained array subtype are assignment compatible if they are both subtypes of the same unconstrained array type. Assignment results in a runtime error only if the two objects have different lengths. Consider such an assignment, assuming that a run time error does not occur:

```
u : S;
v : T;
...
v := u;
```

Since the value of v must always belong to the subtype T, it is not necessarily the case that the final value of v is equal to the initial value of u. A change of value will occur if the two subtypes have different index ranges. For example, u may be an array from 1 up to 10, while v is an array from 2 up to 11. This is referred to as *implicit subtype conversion.*

Unconstrained-Array Subtypes The type rules of SPARK are designed to ensure that implicit subtype conversion **does not occur**. This requires that the two subtypes:

1. are defined on the same (full) type,
2. have equal index ranges.

Therefore the expression type for an unconstrained-array subtype can be characterised by the type mark of the parent unconstrained array (*tid*) and the ranges of the index types (*constraints*).

$$UArrValT \mathrel{\widehat=} [tid : IdDot;\ constraints : \mathrm{seq}(\mathbb{P}\ Val)]$$

6.1.5 Universal Fixed

Two objects of any fixed point type can be combined using multiplication or division operators with a result of the predefined *universal_fixed* type. The only operation which can be applied to the result is a conversion to a named fixed point type. There are no literals associated with this universal type.

The type *universal_fixed* can be considered to be a conceptual device used in the presentation of the rules associated with these operators. The operators are themselves special cases: they are not 'declared' and cannot be renamed. See Section 6.16.

6.1.6 Subtypes

A variable is declared as an element of a particular type or subtype. Some expressions may also belong to a particular subtype: thus names used as expressions, and qualified

expressions (including aggregates) belong to an identified type or subtype. However, a subtype cannot necessarily be associated with all expressions. In particular, operators are declared between types, rather than subtypes.

A subtype name is used as the *ExpType* of an expression if one exists. We use the function *bases* to convert subtypes to types:

$$bases_{Env} : \mathbb{P}\, BasicType \rightarrow \mathbb{P}\, BasicType$$

$$\forall\, \delta : Env;\ btyps : \mathbb{P}\, BasicType \bullet$$
$$bases_\delta = basic\_ancestor_\delta(\!|\, btyps \,|\!)$$

where the function *basic_ancestor* is defined by:

$$basic\_ancestor_\delta : BasicType \rightarrow BasicType$$

$$\forall\, \delta : Env;\ t : IdDot \bullet$$
$$basic\_ancestor_\delta\ (nameT\ t) = nameT\ (ancestorof_\delta\ t) \land$$
$$basic\_ancestor_\delta\ uintT = uintT \land$$
$$basic\_ancestor_\delta\ urealT = urealT$$

6.1.7 Type Compatibility

The relation $compat_{Env}$ holds between an expression type (*ExpType*) and a type mark ($t : IdDot$) if the expression can be assigned to a name of type t. The are two possible cases:

1. The expression denotes a value, whose set of basic types includes one with the same ancestor as the type mark.
2. The expression denotes a value of an unconstrained array subtype, and the subtype of the name:
   (a) is defined from the same unconstrained array type as the expression type, and
   (b) has the same index ranges as the expression type.

$$\_\ compat_{Env}\ \_ : ExpType \leftrightarrow IdDot$$

$$\forall\, \delta : Env;\ btyps : \mathbb{P}\, BasicType;\ tmark : IdDot \bullet$$
$$(valueT\ btyps)\ compat_\delta\ tmark \Leftrightarrow$$
$$(\exists\, t : IdDot \mid$$
$$ancestorof_\delta\ tmark = ancestorof_\delta\ t \land$$
$$nameT\ t \in btyps)$$

$$\forall\, \delta : Env;\ uarr : UArrValT;\ tmark : IdDot \bullet$$
$$(uarrvalT\ uarr)\ compat_\delta\ tmark \Leftrightarrow$$
$$uarr.tid = ancestorof_\delta\ tmark \land$$
$$uarr.constraints = scalar\_tmark\_range_\delta \circ (array\_index\_tmark_\delta\ tmark)$$

6.1.8 Understanding SPARK Types

It is convenient to specify the type rules of SPARK using the very general description of expression type given above. However, many possible values of *ExpType* could not occur as an expression type in SPARK. This section attempts to show what expression types actually occur in SPARK[1].

SPARK has six type constructors (already described in Section 3.1): integer, enumeration, floating point, fixed point, array and record.

Literals and Universal Types SPARK has two sorts of numeric literal: integer and real. These literals are *polymorphic* with respect to the standard named equivalence type rules of SPARK. This polymorphism results in the types being described by a set of basic types: each set contains all the possible integer (or real, but never a mixture) types and the universal type.

$$\begin{aligned} \mathit{Actual\_Exp\_Type} ::= {} & \mathit{intvalT}\langle\langle \mathbb{P}\ \mathit{BasicType} \rangle\rangle \\ \mid {} & \mathit{realvalT}\langle\langle \mathbb{P}\ \mathit{BasicType} \rangle\rangle \end{aligned}$$

Enumerations Enumerated types have literals (the enumeration literals), but these are not polymorphic in SPARK. Thus a unique type can always be assigned to a value of enumerated type.

$$\begin{aligned} \mathit{Actual\_Exp\_Type} ::= {} & \ldots \\ \mid {} & \mathit{enumvalT}\langle\langle \mathit{IdDot} \rangle\rangle \end{aligned}$$

Ranges of Scalar Values Ranges occur, but only for the scalar (i.e. simple) types. The type is unique exactly when the type of the corresponding value is unique.

$$\begin{aligned} \mathit{Actual\_Exp\_Type} ::= {} & \ldots \\ \mid {} & \mathit{intranT}\langle\langle \mathbb{P}\ \mathit{BasicType} \rangle\rangle \\ \mid {} & \mathit{realranT}\langle\langle \mathbb{P}\ \mathit{BasicType} \rangle\rangle \\ \mid {} & \mathit{enumranT}\langle\langle \mathit{IdDot} \rangle\rangle \end{aligned}$$

Composite Types An object of composite type can always be associated with a unique type.

$$\begin{aligned} \mathit{Actual\_Exp\_Type} ::= {} & \ldots \\ \mid {} & \mathit{arrvalT}\langle\langle \mathit{IdDot} \rangle\rangle \\ \mid {} & \mathit{recvalT}\langle\langle \mathit{IdDot} \rangle\rangle \end{aligned}$$

The two special cases — universal fixed and unconstrained arrays — would need to be added to complete the picture.

References: $\mathit{ancestorof}_\delta$ p. 281; $\mathit{scalar\_tmark\_range}_\delta$ p. 296; $\mathit{array\_index\_tmark}_\delta$ p. 289

[1] The material in this section does not add to the static semantics of SPARK. It is intended for clarification only.

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* 7168
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 Static Semantics

1. The literal is always well-formed.
2. The literal is a value of any integer type, or of universal integer type.

$$\frac{\forall\, \delta : Env;\ n : \mathbb{Z};\ ntyps : \mathbb{P}\, BasicType \mid ntyps = nameT(\!|(is\_int\_tmark_\delta)|) \cup \{\ uintT\ \} \bullet}{\delta \vdash_{Exp} lint\ n : valueT\ ntyps} \quad \text{(LInt)}$$

References: *BasicType* p. 73; *nameT* p. 73; *is_int_tmark*$_\delta$ p. 282; *uintT* p. 73; *valueT* p. 73

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.

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

6.3.2 Static Semantics

1. The literal is always well-formed.
2. The literal is a value of any real type, or of universal real type.

$$\frac{\forall\, \delta : Env;\ r : \text{Real};\ rtyps : \mathbb{P}\, BasicType \mid rtyps = nameT(\!|(is\_real\_tmark_\delta)|\!) \cup \{\ urealT\ \} \bullet}{\delta \vdash_{Exp} lreal\ r : valueT\ rtyps} \qquad \text{(LReal)}$$

It is fairly clear that the set of types includes both floating point and fixed point, and not only floating point. However, there are lots of difficulties, especially with the multiplying operators (see LRM 4.6, ¶15(c))
References: BasicType p. 73; *nameT* p. 73; *is_real_tmark*$_\delta$ p. 285; *urealT* p. 73; *valueT* p. 73

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 literal is represented by an element of the set *Char*.

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

6.4.2 Static Semantics

1. The literal is always well-formed.
2. The literal is a value of the predefined character type.

$$\frac{\forall\, \delta : Env;\ c : Char \bullet}{\delta \vdash_{Exp} lcharc : valueT\ \{(nameT\ (id\ character))\}} \quad \text{(LChar)}$$

References: *Env* p. 12; *Char* p. 5; *valueT* p. 73; *nameT* p. 73; *character* p. 265

6.5 String Literals

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

Syntax Example	A.S. Representation
"a"	*lstrg* ⟨a⟩
"bc"	*lstrg* ⟨b, c⟩

6.5.1 Abstract Syntax

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

$$Exp ::= \ldots \mid lstrg\langle\langle \text{seq } Char \rangle\rangle$$

Pairs of double-quotes may be used in strings to represent a *single* double-quote character.

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 Static Semantics

The name can be the name of an object or of a type. In the case of an object two rules are required: the first applies to types which are neither unconstrained arrays nor their subtypes, the second applies to the special case of object of unconstrained array subtypes.

Object Name The name can be an object, representing a constant, a variable (possibly indexed or selected), an enumeration literal or the value returned by a function call.

1. The name must be a well-formed object *obj* of a type which is neither unconstrained nor a subtype of an unconstrained array, with read access mode.

2. The resulting expression is a value with the unique type of the name.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ n : Name;\ obj : ObjDict \mid \\ \quad \neg\ is\_unconstrained\_tmark_{\delta}\ obj.type \wedge \\ \quad \neg\ is\_unconstrained\_subtmark_{\delta}\ obj.type \wedge \\ \quad rd \in obj.modes \bullet \\ \delta \vdash_{Name} n : objnam\ obj\end{array}}{\delta \vdash_{Exp} nam\ n : valueT\ \{nameT\ obj.type\}} \quad \text{(Nam1)}$$

Unconstrained Subtypes Object Names The second rule is a special case allowing the use of names of constrained array subtypes as expressions (unconstrained types may never be used).

1. The name must be a well-formed object *obj* whose (sub)type is an unconstrained array subtype, with read access mode.

2. The type of the unconstrained array subtype value (*tid*) is the ancestor of the subtype of the name.

3. The constraints of the unconstrained array subtype value (*constraints*) are the ranges of the array's indexes.

4. The resulting expression denotes a value with the special expression type used for subtypes of unconstrained arrays.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ n : Name;\ obj : ObjDict;\ UArrValT \mid \\ \qquad is\_unconstrained\_subtmark_{\delta}\ obj.type \wedge \\ \qquad rd \in obj.modes \wedge \\ \qquad tid = ancestorof_{\delta}\ obj.type \wedge \\ \qquad constraints = scalar\_tmark\_range_{\delta} \circ \\ \qquad\qquad (array\_index\_tmark_{\delta}\ obj.type) \bullet \\ \delta \vdash_{Name} n : objnam\ obj\end{array}}{\delta \vdash_{Exp} nam\ n : uarrvalT\ (\theta\, UArrValT)} \quad \text{(Nam2)}$$

Type The name can be a type.

1. The name must be well-formed and refer to a discrete type.

2. The resulting expression is a range, with the unique type of the name.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ n : Name;\ t : IdDot \mid \\ \qquad is\_discrete\_tmark_{\delta}\ t \bullet \\ \delta \vdash_{Name} n : typnam\ t\end{array}}{\delta \vdash_{Exp} nam\ n : rangeT\ \{nameT\ t\}} \quad \text{(Nam3)}$$

References: *Name* p. 53; *ObjDict* p. 54; *is_unconstrained_tmark*$_{\delta}$ p. 283 *objnam* p. 54; *valueT* p. 73; *is_unconstrained_subtmark*$_{\delta}$ p. 284 *nameT* p. 73; *UArrValT* p. 75; *ancestorof*$_{\delta}$ p. 281; *scalar_tmark_range*$_{\delta}$ p. 296; *uarrvalT* p. 73; *typnam* p. 54; *array_index_tmark*$_{\delta}$ p. 289; *is_discrete_tmark*$_{\delta}$ p. 285; *rangeT* p. 73

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 Static Semantics

1. The expressions giving the bounds of the range must be well-formed and denote values.
2. The expression denotes a range, with any type common to both bounds; there must be at least one such type.
3. **SLI WJ004** *may require the type to be scalar.*

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ DotsExp;\ ltyps, utyps, rtyps : \mathbb{P}\, BasicType \mid \\ \qquad rtyps = bases_\delta\ ltyps \cap bases_\delta\ utyps \wedge \\ \qquad rtyps \neq \varnothing \bullet \\ \delta \vdash_{Exp} upper : valueT\ utyps \\ \delta \vdash_{Exp} lower : valueT\ ltyps\end{array}}{\delta \vdash_{Exp} dots(\theta DotsExp) : rangeT\ (ltyps \cap utyps)} \quad \text{(Dots)}$$

References: *BasicType* p. 73; $bases_\delta$ p. 76; *valueT* p. 73; *rangeT* p. 73

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	$in \langle\!\langle\ value \mapsto nam\ simp\ v,\ range \mapsto dots \langle\!\langle\ lower \mapsto lint\ 1,\ upper \mapsto lint\ 10\ \rangle\!\rangle\ \rangle\!\rangle$

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 Static Semantics

1. The *value* expression must be well-formed, with a value type.
2. The *range* expression must be well-formed, with a range type.
3. The value expression and the range expression must have a common type.
4. The result type is boolean (see page 295).

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ InExp;\ vtyps, rtyps : \mathbb{P}\ BasicType \mid \\ \quad bases_\delta\ vtyps \cap bases_\delta\ rtyps \neq \varnothing \bullet \\ \delta \vdash_{Exp} value : valueT\ vtyps \\ \delta \vdash_{Exp} range : rangeT\ rtyps\end{array}}{\delta \vdash_{Exp} in(\theta InExp) : booltype} \quad \text{(In)}$$

References: BasicType p. 73; $bases_\delta$ p. 76; *valueT* p. 73; *rangeT* p. 73; *booltype* p. 295

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	$notin \langle\!\langle\ value \mapsto nam\ (simp\ today),\ range \mapsto nam\ (simp\ weekday)\ \rangle\!\rangle$

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 Static Semantics

1. The *value* expression must be well-formed, with a value type.
2. The *range* expression must be well-formed, with a range type.
3. The value expression and the range expression must have a common type.
4. The result type is boolean (see page 295).

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ InExp;\ vtyps, rtyps : \mathbb{P}\ BasicType \mid \\ \qquad bases_\delta\ vtyps \cap bases_\delta\ rtyps \neq \varnothing \bullet \\ \delta \vdash_{Exp} value : valueT\ vtyps \\ \delta \vdash_{Exp} range : rangeT\ rtyps\end{array}}{\delta \vdash_{Exp} notin(\theta InExp) : booltype} \quad \text{(NotIn)}$$

References: *BasicType* p. 73; $bases_\delta$ p. 76; *valueT* p. 73; *rangeT* p. 73; *booltype* p. 295

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)	$qual \langle\!\| \; typemark \mapsto dot\ (K, T),$ $value \mapsto nam\ (simp\ V) \; \|\!\rangle$

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 Static Semantics

1. The expression must be well-formed with a type which is assignment compatible with the type mark.

2. The type of the qualified expression is uniquely defined by the type mark.

$$\frac{\begin{array}{l}\forall \delta : Env;\ QualExp;\ etyp : ExpType \mid \\ \qquad etyp\ compat_\delta\ typemark \bullet \\ \delta \vdash_{Exp} value : etyp\end{array}}{\delta \vdash_{Exp} qual(\theta QualExp) : valueT\ \{nameT\ typemark\}} \quad \text{(Qual)}$$

Note *The rules for assignment compatibility prevent an unconstrained array type from being used as a qualifying type.*
References: ExpType p. 73; *compat*$_\delta$ p. 76; *valueT* p. 73; *nameT* p. 73

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)	$pagg\ \langle\!\langle\ typemark \mapsto id\ T,$ $components \mapsto \langle lint\ 1, lint\ 2\rangle\ \rangle\!\rangle$

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 Static Semantics

Separate rules are required for record and array aggregates.

Array Aggregates

1. The typemark must be a visible array which is not unconstrained or limited.

2. The array must be one-dimensional.

3. The number of component expressions must be equal to the length of the array, i.e. the size of the range of the (first and only) array index type.

4. Each expression in the list of components must be well-formed and compatible with the component type of the array.

5. The aggregate denotes a value with a unique type given by the qualifying typemark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PAggExp \mid \\ \qquad is\_visible\_tmark_\delta\ typemark \wedge \\ \qquad is\_arr\_tmark_\delta\ typemark \wedge \\ \qquad \neg\ is\_unconstrained\_tmark_\delta\ typemark \wedge \\ \qquad \neg\ is\_limited\_tmark_\delta\ typemark \wedge \\ \qquad array\_arity_\delta\ typemark = 1 \wedge \\ \qquad \#components = \#\, scalar\_tmark\_range_\delta \\ \qquad\qquad (array\_index\_tmark_\delta\ typemark\ 1) \bullet \\ \delta, array\_comp\_tmark_\delta\ typemark \vdash_{ExpS} components\end{array}}{\delta \vdash_{Exp} pagg\ (\theta PAggExp) : valueT\ \{nameT\ typemark\}} \quad \text{(PAgg1)}$$

The well-formation of a list of expression ($\vdash_{ExpS}$) is defined below.

Record Aggregates

1. The type mark is a visible record type which is not limited.
2. The component is a well-formed positional association containing expressions of the required types.
3. The record aggregate denotes a value with the unique type given by the type mark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PAggExp \mid \\ \qquad is\_visible\_tmark_\delta\ typemark \wedge \\ \qquad \neg\ is\_limited\_tmark_\delta\ typemark \wedge \\ \qquad is\_rec\_tmark_\delta\ typemark \bullet \\ \delta, rec\_seq\_tmark_\delta\ typemark \vdash_{PAssoc} components\end{array}}{\delta \vdash_{Exp} pagg\ (\theta PAggExp) : valueT\ \{nameT\ typemark\}} \quad \text{(PAgg2)}$$

The function $rec\_seq\_tmark_\delta$ returns the list of record field type marks in the order of declaration.
References: $is\_visible\_tmark_\delta$ p. 281; $is\_arr\_tmark_\delta$ p. 283; $is\_unconstrained\_tmark_\delta$ p. 283; *Exp* p. 71; *Env* p. 12; $array\_arity_\delta$ p. 288; $is\_limited\_tmark_\delta$ p. 286; $array\_comp\_tmark_\delta$ p. 289; *valueT* p. 73; *nameT* p. 73; $scalar\_tmark\_range_\delta$ p. 296; $array\_index\_tmark_\delta$ p. 289; $\vdash_{ExpS}$ p. 90; $is\_rec\_tmark_\delta$ p. 284; $rec\_seq\_tmark_\delta$ p. 288; $\vdash_{PAssoc}$ p. 65;

6.11.3 Expression List

The well-formation of a list of expressions all compatible with the same type is described by the relation:

$$\_,\_ \vdash_{ExpS} \_ \subseteq Env \times IdDot \times \text{seq}\ Exp$$

Empty List This is always well-formed.

$$\frac{\forall\, \delta : Env;\ t : IdDot \bullet}{\delta, t \vdash_{ExpS} \langle\rangle} \qquad \text{(ExpS1)}$$

Non-empty List This is well-formed if the first expression, and the rest of the list, are both well-formed.

1. The expression must be well-formed with a type compatible with the given type t.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ t : IdDot;\ exp : Exp;\ exps : \mathrm{seq}\, Exp;\ etyp : ExpType \mid \\ \qquad etyp\ compat_\delta\ t \bullet \\ \delta \vdash_{Exp} exp : etyp \\ \delta, t \vdash_{ExpS} exps\end{array}}{\delta, t \vdash_{ExpS} \langle exp \rangle \frown exps} \qquad \text{(ExpS2)}$$

References: *Env* p. 12; *Exp* p. 71; *ExpType* p. 73; $compat_\delta$ p. 76

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.

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

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

6.12.2 Static Semantics

1. The typemark must be a visible array which is not unconstrained or limited.

2. The array must be one-dimensional.

3. The number of component expressions must be less than the length of the array, i.e. the size of the range of the (first and only) array index type.

4. Each expression in the list of components must be well-formed and compatible with the component type of the array.

5. The *other* expression must be well-formed, with a type compatible with the component type of the array.

6. The aggregate is a value with a unique type given by the qualifying typemark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PAggOthExp;\ etyp : ExpType \mid \\ \quad is\_visible\_tmark_\delta\ typemark \wedge \\ \quad is\_arr\_tmark_\delta\ typemark \wedge \\ \quad \neg\ is\_unconstrained\_tmark_\delta\ typemark \wedge \\ \quad \neg\ is\_limited\_tmark_\delta\ typemark \wedge \\ \quad array\_arity_\delta\ typemark = 1 \wedge \\ \quad \#components < \#\, scalar\_tmark\_range_\delta \\ \qquad\qquad (array\_index\_tmark_\delta\ typemark\ 1) \\ \quad etyp\ compat_\delta\ (array\_comp\_tmark_\delta\ typemark) \bullet \\ \\ \delta \vdash_{Exp} other : etyp \\ \delta, array\_comp\_tmark_\delta\ typemark \vdash_{ExpS} components\end{array}}{\delta \vdash_{Exp} paggoth\ (\theta PAggOthExp) : valueT\ \{nameT\ typemark\}} \quad \text{(PAggOth)}$$

References: *Exp* p. 71; *Env* p. 12; *ExpType* p. 73; $is\_visible\_tmark_\delta$ p. 281; $is\_arr\_tmark_\delta$ p. 283; $is\_unconstrained\_tmark_\delta$ p. 283; $is\_limited\_tmark_\delta$ p. 286; $array\_arity_\delta$ p. 288; $compat_\delta$ p. 76; $scalar\_tmark\_range_\delta$ p. 296; $array\_index\_tmark_\delta$ p. 289; $array\_comp\_tmark_\delta$ p. 289; $\vdash_{ExpS}$ p. 90; *valueT* p. 73; *nameT* p. 73

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

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 Static Semantics

Array Aggregate If the aggregate is an array the *choice* expression indicates a value, or range of values, in the array index type.

1. The typemark must be a visible array which is not unconstrained or limited.

2. The array must be one-dimensional.
3. The association list must be well-formed (using $\vdash_{NAscArr}$, defined below).
4. The choices must exhaust the range of the index type *itype*.
5. The aggregate is a value with a unique type given by the qualifying typemark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NAggExp;\ itype, ctype : IdDot \mid \\ \quad is\_visible\_tmark_\delta\ typemark \wedge \\ \quad is\_arr\_tmark_\delta\ typemark \wedge \\ \quad \neg\ is\_unconstrained\_tmark_\delta\ typemark \wedge \\ \quad \neg\ is\_limited\_tmark_\delta\ typemark \wedge \\ \quad array\_arity_\delta\ typemark = 1 \wedge \\ \quad ctype = array\_comp\_tmark_\delta\ typemark \wedge \\ \quad itype = array\_index\_tmark_\delta\ typemark\ 1 \bullet \\ \delta, ctype, itype \vdash_{NAscArr} assocs \Longrightarrow scalar\_tmark\_range_\delta\ itype\end{array}}{\delta \vdash_{Exp} nagg(\theta NAggExp) : valueT\ \{nameT\ \ typemark\}} \quad \text{(NAgg1)}$$

Record If the aggregate is a record, the choice "expression" can only be a field name.

1. The typemark must be a visible record which is not limited.
2. The association list must be well-formed (using $\vdash_{NAscRec}$, defined below).
3. The aggregate is a value with a unique type given by the qualifying typemark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NAggExp \mid \\ \quad is\_visible\_tmark_\delta\ typemark \wedge \\ \quad \neg\ is\_limited\_tmark_\delta\ typemark \wedge \\ \quad is\_rec\_tmark_\delta\ typemark \bullet \\ \delta, rec\_field\_tmark_\delta\ typemark \vdash_{NAscRec} assocs\end{array}}{\delta \vdash_{Exp} nagg(\theta NAggExp) : valueT\ \{nameT\ \ typemark\}} \quad \text{(NAgg2)}$$

References: *is_visible_tmark*$_\delta$ p. 281; *is_arr_tmark*$_\delta$ p. 283; *is_unconstrained_tmark*$_\delta$ p. 283; *array_arity*$_\delta$ p. 288; *is_limited_tmark*$_\delta$ p. 286; *array_comp_tmark*$_\delta$ p. 289; *Exp* p. 71; *Env* p. 12; *array_index_tmark*$_\delta$ p. 289; *scalar_tmark_range*$_\delta$ p. 296; $\vdash_{NAscArr}$ p. 95; *valueT* p. 73; *is_rec_tmark*$_\delta$ p. 284; *nameT* p. 73; *rec_field_tmark*$_\delta$ p. 288; $\vdash_{NAscRec}$ p. 96

6.13.3 Array Named Association List

The well-formation of a named association list used in an array aggregate is described by the relation:

$$\_,\_,\_ \vdash_{NAscArr} \_ \Longrightarrow \_ \subseteq Env \times IdDot \times IdDot \times \mathrm{seq}\, NAggAssoc \times \mathbb{P}\, Val$$

thus in the well-formation assertion:

$$\delta, ctype, itype \vdash_{NAscArr} assocs \Longrightarrow range$$

ctype is the component type of the array, *itype* is the index type, *range* is the range of values (of the type *itype*) covered by the choices in the association list *assocs*.

Empty Association This is always well-formed.

$$\frac{\forall\, \delta : Env;\ ctype, itype : IdDot \bullet}{\delta, ctype, itype \vdash_{NAscArr} \langle\rangle \Longrightarrow \varnothing} \qquad \text{(NArrAsc1)}$$

Range Choice The choice expression may be a range.

1. The choice expression may be a well-formed expression, with a range type. Its type must be compatible with the array index type.
2. The choice expression must be static; the range of values to which the expression evaluates must not overlap with the values covered by other choices.
3. The component expression must be well-formed; its type must be compatible with the component type of the array.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NAggAssoc;\ ctype, itype : IdDot;\ etyp : ExpType; \\ \quad assocs : \mathrm{seq}\, NAggAssoc;\ cvals, avals : \mathbb{P}\, Val; \\ \quad btyps : \mathbb{P}\, BasicType \mid \\ \qquad (\exists\, t : IdDot \mid nameT\ t \in btyps \wedge \\ \qquad\quad ancestorof_\delta\ itype = ancestorof_\delta\ t) \wedge \\ \qquad avals \cap cvals = \varnothing \wedge \\ \qquad etyp\ compat_\delta\ ctype \bullet \\ \quad \delta \vdash_{Exp} choice : rangeT\ btyps \\ \quad \delta \vdash choice \Longrightarrow_{\mathrm{StaticEval}} rngval\ cvals \\ \quad \delta \vdash_{Exp} component : etyp \\ \quad \delta, ctype, itype \vdash_{NAscArr} assocs \Longrightarrow avals\end{array}}{\begin{array}{l}\delta, ctype, itype \vdash_{NAscArr} \langle \theta NAggAssoc \rangle \frown assocs \Longrightarrow \\ \qquad\qquad avals \cup cvals\end{array}} \qquad \text{(NArrAsc2)}$$

Range Value The choice expression may be a value.

1. The choice expression may be a well-formed expression, with a value type. Its type must be compatible with the array index type.
2. The choice expression must be static; the value to which the expression evaluates must not overlap with the values covered by other choices.
3. The component expression must be well-formed; its type must be compatible with the component type of the array.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NAggAssoc;\ ctype, itype : IdDot;\ etyp : ExpType; \\ \quad assocs : \mathrm{seq}\, NAggAssoc; \\ \quad cval : Val;\ avals : \mathbb{P}\, Val;\ btyps : \mathbb{P}\, BasicType \mid \\ \qquad valueT\ btyps\ compat_\delta\ itype \land \\ \qquad cval \notin cvals \land \\ \qquad etyp\ compat_\delta\ ctype \bullet \\ \quad \delta \vdash_{Exp} choice : valueT\ btyps \\ \quad \delta \vdash choice \Longrightarrow_{\mathrm{StaticEval}} cval \\ \quad \delta \vdash_{Exp} component : etyp \\ \quad \delta, ctype, itype \vdash_{NAscArr} assocs \Longrightarrow avals\end{array}}{\begin{array}{l}\delta, ctype, itype \vdash_{NAscArr} \langle \theta NAggAssoc \rangle \frown assocs \Longrightarrow \\ \qquad\qquad avals \cup \{cval\}\end{array}} \quad \text{(NArrAsc3)}$$

References: *Env* p. 12; *NAggAssoc* p. 93; *Val* p. 6; *ExpType* p. 73; *BasicType* p. 73; *nameT* p. 73; $ancestorof_\delta$ p. 281; $compat_\delta$ p. 76; *rangeT* p. 73; *valueT* p. 73

6.13.4 Record Named Association List

The well-formation of a named association list used in a record aggregate is described by the following relation:

$$\_, \_ \vdash_{NAscRec} \_ \subseteq Env \times (Id \nrightarrow IdDot)\, \mathrm{seq}\, NAggAssoc$$

Empty Association List This is well-formed if there are no fields in the record not already given a value.

$$\frac{\forall\, \delta : Env \bullet}{\delta, \varnothing \vdash_{NAscRec} \langle\rangle} \quad \text{(NRecAsc1)}$$

Non-Empty Association List This is well-formed if the first association in the list is well-formed, and the rest of list is well-formed without repeating the same field identifier.

1. The *choice* expression must a field name of the record for which a value has not already been given.

2. The *component* expression must be well-formed with a type compatible with the type of this record field.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ ftypes : Id \pfun IdDot;\ NAggAssoc; \\ \quad assocs : \mathrm{seq}\ NAggAssoc; \\ \quad etyp : ExpType;\ field : Id \mid \\ \qquad choice = nam\ (simp\ field) \land \\ \qquad etyp\ compat_\delta\ (ftypes\ field) \bullet \\ \quad \delta \vdash_{Exp} component : etyp \\ \quad \delta, \{field\} \ndres ftypes \vdash_{NAscRec} assocs\end{array}}{\delta, ftypes \vdash_{NAscRec} \langle \theta NAggAssoc \rangle \frown assocs} \quad \text{(NRecAsc)}$$

References: Env p. 12; *NAggAssoc* p. 93; $compat_\delta$ p. 76

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)	$pagg\ \langle\!\langle\ typemark \mapsto dot(K, T),\ assocs \mapsto \langle\langle\ choice \mapsto dots\ \langle\!\langle\ lower \mapsto lint\ 1,\ upper \mapsto lint\ 3\ \rangle\!\rangle,\ component \mapsto lint\ 10\ \rangle\rangle,\ other \mapsto lint\ 11\ \rangle\!\rangle$

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.

NAggOthExp

$typemark : IdDot$
$components : \mathrm{seq}\ NAggAssoc$
$other : Exp$

The schema *NAggAssoc* is defined on page 93.

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

6.14.2 Static Semantics

1. The typemark must be a visible array which is not unconstrained or limited.
2. The array must be one-dimensional.
3. The association list must be well-formed (using $\vdash_{NAscArr}$, defined in Section 6.13).
4. The choices must not exhaust the range of the index type *itype*.
5. The *others* expression must be well-formed and compatible with the component type of the array.
6. The aggregate is a value with a unique type given by the qualifying typemark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NAggOthExp;\ itype, ctype : IdDot;\ etyp : ExpType;\\ \quad avals : \mathbb{P}\ Val \mid\\ \qquad is\_visible\_tmark_\delta\ typemark \land\\ \qquad is\_arr\_tmark_\delta\ typemark \land\\ \qquad \neg\ is\_unconstrained\_tmark_\delta\ typemark \land\\ \qquad \neg\ is\_limited\_tmark_\delta\ typemark \land\\ \qquad array\_arity_\delta\ typemark = 1 \land\\ \qquad ctype = array\_comp\_tmark_\delta\ typemark \land\\ \qquad itype = array\_index\_tmark_\delta\ typemark\ 1 \land\\ \qquad avals \subset scalar\_tmark\_range_\delta\ itype \land\\ \qquad etyp\ compat_\delta\ ctype \bullet\\ \quad \delta \vdash_{Exp} others : etyp\\ \quad \delta, ctype, itype \vdash_{NAscArr} assocs \Longrightarrow avals\end{array}}{\delta \vdash_{Exp} naggoth(\theta NAggOthExp) : valueT\ \{nameT\ typemark\}}\ (\text{NAggOth})$$

References: *Exp* p. 71; *Env* p. 12; *ExpType* p. 73; $is\_visible\_tmark_\delta$ p. 281; $is\_arr\_tmark_\delta$ p. 283; $is\_unconstrained\_tmark_\delta$ p. 283; $is\_limited\_tmark_\delta$ p. 286; $array\_arity_\delta$ p. 288; *valueT* p. 73; $array\_comp\_tmark_\delta$ p. 289; $array\_index\_tmark_\delta$ p. 289; $scalar\_tmark\_range_\delta$ p. 296; $compat_\delta$ p. 76; $\vdash_{NAscArr}$ p. 95; *nameT* p. 73

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	$un \langle\!\langle\ op \mapsto not,\ arg \mapsto nam\ (simp\ OPEN)\ \rangle\!\rangle$
- 100	$un \langle\!\langle\ op \mapsto uminus,\ arg \mapsto lint\ 100\ \rangle\!\rangle$

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 Static Semantics

Standard Case

1. The argument expression (arg) must be well-formed value.
2. The result types are the return types of all the visible operator (op) and argument type combinations. At least one such type must exist.

$$\frac{\forall\, \delta : Env;\ UnExp;\ atyps, utyps : \mathbb{P}\, BasicType \mid utyps = \delta.unops(\!|\{op\} \times (bases_\delta\ atyps)|\!) \wedge utyps \neq \varnothing \bullet \quad \delta \vdash_{Exp} arg : valueT\ atyps}{\vdash_{Exp} un(\theta\, UnExp) : valueT\ utyps} \qquad \text{(Un1)}$$

Unconstrained Array Subtypes A special case allows an operator to be applied to an expression of an unconstrained array subtype.

1. The argument expression (arg) must be a well-formed expression, which is a value of an unconstrained array subtype.

2. The operator must be visible for operands of the unconstrained type, from which the subtype is defined.

3. The return type is the unconstrained array subtype.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ UnExp;\ UArrValT \mid \\ \qquad (op, nameT\ tid) \in \delta.unops \bullet \\ \qquad \delta \vdash_{Exp} arg : uarrvalT\ (\theta\, UArrValT)\end{array}}{\delta \vdash_{Exp} un(\theta\, UnExp) : uarrvalT\ (\theta\, UArrValT)} \quad \text{(Un2)}$$

References: *BasicType* p. 73; $bases_{\delta}$ p. 76; *valueT* p. 73; *UArrValT* p. 75; *nameT* p. 73; *uarrvalT* p. 73

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

$$Bop ::= and \mid or \mid xor \mid eq \mid noteq \mid lt \mid lte \mid gt \mid gte$$
$$\mid\ plus \mid minus \mid mul \mid div \mid mod \mid rem \mid power$$

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 Static Semantics

Three cases are required: the standard case and special cases for unconstrained array subtypes and multiplication of fixed point numbers.

Standard Case

1. The arguments must be well-formed value expressions.
2. The expression can have any type which is the result type of a visible operator and operand type combination, where the operand types are possible types of the arguments; at least one such operator and operand combination must be visible.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ BinExp;\ ltyps, rtyps, btyps : \mathbb{P}\ BasicType \mid \\ \qquad btyps = \delta.binops(\!|(bases_\delta\ ltyps) \times \{op\} \times (bases_\delta\ rtyps)|\!) \land \\ \qquad btyps \neq \varnothing \\ \delta \vdash_{Exp} larg : valueT\ ltyps \\ \delta \vdash_{Exp} rarg : valueT\ rtyps\end{array}}{\delta \vdash_{Exp} bin(\theta BinExp) : valueT\ byps} \quad \text{(Bin1)}$$

Constrained Array Subtypes A special case allows expressions of a constrained array subtype to appear in binary expressions.

1. The arguments are well-formed expressions of unconstrained array subtypes.
2. The types of the two operands are equal. The effect of this is to require that the subtype of the two arguments
   (a) are defined on the same (full) type,
   (b) have equal index ranges.
3. The operator is visible for operands of the unconstrained (full) type.
4. The result type is the same unconstrained array subtype.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ BinExp;\ UArrValT \mid \\ \qquad (nameT\ tid, op, nameT\ tid) \in \delta.binops \bullet \\ \delta \vdash_{Exp} larg : uarrvalT\ (\theta\, UArrValT) \\ \delta \vdash_{Exp} rarg : uarrvalT\ (\theta\, UArrValT)\end{array}}{\delta \vdash_{Exp} bin(\theta BinExp) : uarrvalT\ (\theta\, UArrValT)} \quad \text{(Bin2)}$$

Fixed Point Multiply and divide operators also exist between different fixed point types. These operators cannot be renamed, so we do consider them to be declared. A special case is required to handle the expressions which use these operators.

1. The arguments are well-formed expressions each with a uniquely defined fixed point type (not necessarily the same).
2. The operator can be multiply or divide.
3. The result type is universal fixed.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ BinExp;\ lt, rt : IdDot \mid \\ \qquad op \in \{mul, div\} \wedge \\ \qquad is\_fixed\_tmark_{\delta}\ lt \wedge \\ \qquad is\_fixed\_tmark_{\delta}\ rt \\ \\ \delta \vdash_{Exp} larg : valueT\ (nameT\ \{lt\}) \\ \delta \vdash_{Exp} rarg : valueT\ (nameT\ \{rt\})\end{array}}{\delta \vdash_{Exp} bin(\theta BinExp) : ufixT} \qquad \text{(Bin3)}$$

References: *BasicType* p. 73; $bases_{\delta}$ p. 76; *valueT* p. 73; *UArrValT* p. 75; *nameT* p. 73; *uarrvalT* p. 73; $is\_fixed\_tmark_{\delta}$ p. 283; *ufixT* p. 73

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    andthen ⟨| larg ↦ bin ⟨| larg ↦ nam (simp A),
  B / A = 0                                 op ↦ noteq,
                                            rarg ↦ lint 0 |⟩,
                             rarg ↦ bin ⟨| larg ↦ bin ⟨| larg ↦ nam (simp B),
                                                        op ↦ div,
                                                        rarg ↦ nam (simp A) |⟩,
                                          op ↦ eq,
                                          rarg ↦ lint 0 |⟩ |⟩
```

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 Static Semantics

1. Both the subexpressions must be well-formed with boolean type.
2. The *andthen* expression has boolean type.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ AndThenExp \bullet \\ \quad \delta \vdash_{Exp} larg : booltype \\ \quad \delta \vdash_{Exp} rarg : booltype\end{array}}{\delta \vdash_{Exp} andthen(\theta AndThenExp) : booltype} \qquad \text{(AndThen)}$$

References: booltype p. 295

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

$$
\begin{array}{ll}
\text{i} = 0 \textbf{ or else} & \mathit{orelse} \langle\!| \ \mathit{larg} \mapsto \mathit{bin} \langle\!| \ \mathit{larg} \mapsto \mathit{nam}\ (\mathit{simp}\ i), \\
\quad \text{A(i)} = 0 & \qquad\qquad\qquad\qquad\quad \mathit{op} \mapsto \mathit{eq}, \\
& \qquad\qquad\qquad\qquad\quad \mathit{rarg} \mapsto \mathit{lint}\ 0\ |\!\rangle, \\
& \qquad\qquad \mathit{rarg} \mapsto \mathit{bin} \langle\!| \ \mathit{larg} \mapsto \mathit{pasc} \langle\!| \ \mathit{prefix} \mapsto \mathit{nam}\ (\mathit{simp}\ A), \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{args} \mapsto \langle \mathit{nam}\ (\mathit{simp}\ i) \rangle\ |\!\rangle, \\
& \qquad\qquad\qquad\qquad\quad \mathit{op} \mapsto \mathit{eq}, \\
& \qquad\qquad\qquad\qquad\quad \mathit{rarg} \mapsto \mathit{lint}\ 0\ |\!\rangle\ |\!\rangle
\end{array}
$$

6.18.1 Abstract Syntax

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

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

$$\mathit{Exp} ::= \ldots \mid \mathit{orelse} \langle\langle \mathit{OrElseExp} \rangle\rangle$$

6.18.2 Static Semantics

1. Both the subexpressions must be well-formed with boolean type.
2. The *orelse* expression has boolean type.

$$\frac{\begin{array}{l}\forall\, \delta : \mathit{Env};\ \mathit{OrElseExp} \bullet \\ \quad \delta \vdash_{\mathit{Exp}} \mathit{larg} : \mathit{booltype} \\ \quad \delta \vdash_{\mathit{Exp}} \mathit{rarg} : \mathit{booltype}\end{array}}{\delta \vdash_{\mathit{Exp}} \mathit{orelse}(\theta\, \mathit{OrElseExp}) : \mathit{booltype}} \qquad \text{(OrElse)}$$

References: *booltype* p. 295

6.19 Catenation

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

Syntax Example	A.S. Representation
`"A" & "a"`	$concat\ \langle\!\|\ larg \mapsto lstrg\ \langle \mathrm{A} \rangle,\ rarg \mapsto lstrg\ \langle \mathrm{a} \rangle\ \|\!\rangle$

6.19.1 Abstract Syntax

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

$$CatExp \mathrel{\widehat{=}} [larg, rarg : Exp]$$
$$Exp ::= \ldots \mid cat\langle\!\langle CatExp \rangle\!\rangle$$

6.20 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 62 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	110
fst	First element of a scalar type	112
bfst	First element of the base type of a scalar type	112
lst	Last element of a scalar type	113
blst	Last element of the base type of a scalar type	113
fsta	First element of the n'th array index type	114
lsta	Last element of the n'th array index type	116
succ	Successor of an element of a discrete type	118
pred	Predecessor of an element of a discrete type	119
pos	Position of an element of discrete type	120
val	Value of an element of discrete type	121
size	Size of an object	122

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)	$rng \langle\!\langle\ arrtyp \mapsto id\ t,\ indexno \mapsto lint\ 3\ \rangle\!\rangle$

7.1.1 Abstract Syntax

RngExp

$arrtyp : IdDot$
$indexno : Exp$

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

7.1.2 Static Semantics

1. The attribute must be applied to the name of a constrained array type or subtype (rule Rng1), or to an unconstrained array object (rule Rng2).
2. The *indexno* argument must be a static expression.
3. *indexno* must be of type *universal_integer*.
4. *indexno* must be strictly positive and no greater than the dimensionality of the array type or subtype.

The first rule deals with the case when *arrtyp* is a(n array) type mark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ val : Val;\ btyp : BasicType;\ RngExp \mid \\ \quad is\_arr\_tmark_\delta\ arrtyp \\ \quad val \in \mathrm{ran}\ intval \\ \quad 1 \le intval^{\sim} val \le \#\ array\_arity_\delta\ arrtyp \\ \quad btyp = nameT\ (ancestorof_\delta \\ \qquad ((array\_index\_tmark_\delta\ arrtyp)(intval^{\sim} val))) \bullet \\ \delta \vdash_{Exp} indexno : uintT \\ \delta \vdash indexno \Longrightarrow_{\mathrm{ConstEval}} val\end{array}}{\delta \vdash_{Exp} rng(\theta RngExp) : rangeT\ \{\ btyp\ \}} \quad \text{(Rng1)}$$

The next rule deals with the case when *arrtyp* is an identifier denoting an array object.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ val : Val;\ atyp, btyp : BasicType;\ RngExp \mid \\ \quad \neg\ is\_arr\_tmark_{\delta}\ arrtyp \\ \quad arrtyp \in \mathrm{ran}\ id \\ \quad atyp \in \mathrm{ran}\ nameT \\ \quad is\_arr\_tmark_{\delta}(nameT^{\sim} atyp) \\ \quad val \in \mathrm{ran}\ intval \\ \quad 1 \leq intval^{\sim} val \leq \#\, array\_arity_{\delta}(nameT^{\sim} atyp) \\ \quad btyp = nameT\ (ancestorof_{\delta} \\ \qquad ((array\_index\_tmark_{\delta}(nameT^{\sim} atyp))(intval^{\sim} val))) \bullet \\ \delta \vdash_{Exp} indexno : uintT \\ \delta \vdash_{Exp} nam(simp(id^{\sim} arrtyp)) : atyp \\ \delta \vdash indexno \Longrightarrow_{\mathrm{ConstEval}} val\end{array}}{\delta \vdash_{Exp} rng(\theta RngExp) : rangeT\ \{\ btyp\ \}} \qquad \text{(Rng2)}$$

References: BasicType p. 73; $is\_arr\_tmark_{\delta}$ p. 283; $array\_arity_{\delta}$ p. 288; *nameT* p. 73; $ancestorof_{\delta}$ p. 281; $array\_index\_tmark_{\delta}$ p. 289; *uintT* p. 73; *rangeT* p. 73

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 Static Semantics

The well-formation rules are the same for *fst* and *bfst*.

1. The attribute must be applied to the name (from *IdDot*) of a scalar type or subtype.
2. The expression is a value with a unique type, which is the "ancestor" of the type to which the attribute is applied.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ typ : IdDot;\ btyp : BasicType \mid \\ \qquad is\_scalar\_tmark_\delta\ typ \wedge \\ \qquad btyp = nameT\ (ancestorof_\delta\ typ)\ \bullet\end{array}}{\delta \vdash_{Exp} fst\ typ : valueT\ \{\ btyp\ \}} \quad \text{(Fst)}$$

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ typ : IdDot;\ btyp : BasicType \mid \\ \qquad is\_scalar\_tmark_\delta\ typ \wedge \\ \qquad btyp = nameT\ (ancestorof_\delta\ typ)\ \bullet\end{array}}{\delta \vdash_{Exp} bfst\ typ : valueT\ \{\ btyp\ \}} \quad \text{(BFst)}$$

References: *BasicType* p. 73; $is\_scalar\_tmark_\delta$ p. 284; $ancestorof_\delta$ p. 281; *nameT* p. 73; *valueT* p. 73

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 Static Semantics

The well-formation rules are the same for *lst* and *blst*.

1. The attribute must be applied to the name (from *IdDot*) of a scalar type or subtype.
2. The expression is a value with a unique type, which is the "ancestor" of the type to which the attribute is applied.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ typ : IdDot;\ btyp : BasicType \mid \\ \qquad is\_scalar\_tmark_\delta\ typ \wedge \\ \qquad btyp = nameT\ (ancestorof_\delta\ typ) \bullet\end{array}}{\delta \vdash_{Exp} lst\ typ : valueT\ \{\ btyp\ \}} \quad \text{(Lst)}$$

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ typ : IdDot;\ btyp : BasicType \mid \\ \qquad is\_scalar\_tmark_\delta\ typ \wedge \\ \qquad btyp = nameT\ (ancestorof_\delta\ typ) \bullet\end{array}}{\delta \vdash_{Exp} blst\ typ : valueT\ \{\ btyp\ \}} \quad \text{(BLst)}$$

References: *BasicType* p. 73; $is\_scalar\_tmark_\delta$ p. 284; *nameT* p. 73; $ancestorof_\delta$ p. 281; *valueT* p. 73

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)	$fsta \langle\!\langle\ arrtyp \mapsto sensors,\ indexno \mapsto lint\ 1\ \rangle\!\rangle$

7.4.1 Abstract Syntax

FstAExp

$arrtyp : IdDot$
$indexno : Exp$

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

7.4.2 Static Semantics

1. The attribute must be applied to the name of a constrained array type or subtype (rule FstA1), or to an unconstrained array object (rule FstA2).
2. The *indexno* argument must be a static expression.
3. *indexno* must be of type *universal_integer*.
4. *indexno* must be strictly positive and no greater than the dimensionality of the array type or subtype.

The first rule deals with the case when *arrtyp* is a(n array) type mark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ val : Val;\ btyp : BasicType;\ FstAExp \mid \\ \quad is\_arr\_tmark_{\delta}\ arrtyp \\ \quad val \in \mathrm{ran}\ intval \\ \quad 1 \le intval^{\sim} val \le \#\ array\_arity_{\delta}\ arrtyp \\ \quad btyp = nameT\ (ancestorof_{\delta} \\ \qquad ((array\_index\_tmark_{\delta}\ arrtyp)(intval^{\sim} val))) \bullet \\ \delta \vdash_{Exp} indexno : uintT \\ \delta \vdash indexno \Longrightarrow_{\mathrm{ConstEval}} val\end{array}}{\delta \vdash_{Exp} fsta(\theta FstAExp) : valueT\ \{\ btyp\ \}} \quad \text{(FstA1)}$$

The next rule deals with the case when *arrtyp* is an identifier denoting an array object.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ val : Val;\ atyp, btyp : BasicType;\ FstAExp \mid \\ \quad \neg\ is\_arr\_tmark_{\delta}\ arrtyp \\ \quad arrtyp \in \operatorname{ran} id \\ \quad atyp \in \operatorname{ran} nameT \\ \quad is\_arr\_tmark_{\delta}(nameT^{\sim} atyp) \\ \quad val \in \operatorname{ran} intval \\ \quad 1 \leq intval^{\sim} val \leq \#\, array\_arity_{\delta}(nameT^{\sim} atyp) \\ \quad btyp = nameT\ (ancestorof_{\delta} \\ \qquad ((array\_index\_tmark_{\delta}(nameT^{\sim} atyp))(intval^{\sim} val))) \bullet \\ \delta \vdash_{Exp} indexno : uintT \\ \delta \vdash_{Exp} nam(simp(id^{\sim} arrtyp)) : atyp \\ \delta \vdash indexno \Longrightarrow_{\mathrm{ConstEval}} val\end{array}}{\delta \vdash_{Exp} fsta(\theta FstAExp) : valueT\ \{\ btyp\ \}} \qquad \text{(FstA2)}$$

References: BasicType p. 73; $is\_arr\_tmark_{\delta}$ p. 283; $array\_arity_{\delta}$ p. 288; *nameT* p. 73; $ancestorof_{\delta}$ p. 281; $array\_index\_tmark_{\delta}$ p. 289; *uintT* p. 73; *valueT* p. 73

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)	$lsta \langle\!\langle\ arrtyp \mapsto sensors,\ indexno \mapsto lint\ 2\ \rangle\!\rangle$

7.5.1 Abstract Syntax

LstAExp

$arrtyp : IdDot$
$indexno : Exp$

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

7.5.2 Static Semantics

1. The attribute must be applied to the name of a constrained array type or subtype (rule LstA1), or to an unconstrained array object (rule LstA2).
2. The *indexno* argument must be a static expression.
3. *indexno* must be of type *universal_integer*.
4. *indexno* must be strictly positive and no greater than the dimensionality of the array type or subtype.

The first rule deals with the case when *arrtyp* is a(n array) type mark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ val : Val;\ btyp : BasicType;\ LstAExp \mid \\ \quad is\_arr\_tmark_{\delta}\ arrtyp \\ \quad val \in \mathrm{ran}\ intval \\ \quad 1 \leq intval^{\sim} val \leq \#\ array\_arity_{\delta}\ arrtyp \\ \quad btyp = nameT\ (ancestorof_{\delta} \\ \qquad ((array\_index\_tmark_{\delta}\ arrtyp)(intval^{\sim} val))) \bullet \\ \delta \vdash_{Exp} indexno : uintT \\ \delta \vdash indexno \Longrightarrow_{\mathrm{ConstEval}} val\end{array}}{\delta \vdash_{Exp} lsta(\theta LstAExp) : valueT\ \{\ btyp\ \}} \quad \text{(LstA1)}$$

The next rule deals with the case when *arrtyp* is an identifier denoting an array object.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ val : Val;\ atyp, btyp : BasicType;\ LstAExp \mid \\ \quad \neg\ is\_arr\_tmark_{\delta}\ arrtyp \\ \quad arrtyp \in \mathrm{ran}\ id \\ \quad atyp \in \mathrm{ran}\ nameT \\ \quad is\_arr\_tmark_{\delta}(nameT^{\sim} atyp) \\ \quad val \in \mathrm{ran}\ intval \\ \quad 1 \leq intval^{\sim} val \leq \#\, array\_arity_{\delta}(nameT^{\sim} atyp) \\ \quad btyp = nameT\ (ancestorof_{\delta} \\ \qquad ((array\_index\_tmark_{\delta}(nameT^{\sim} atyp))(intval^{\sim} val))) \bullet \\ \delta \vdash_{Exp} indexno : uintT \\ \delta \vdash_{Exp} nam(simp(id^{\sim} arrtyp)) : atyp \\ \delta \vdash indexno \Longrightarrow_{\mathrm{ConstEval}} val\end{array}}{\delta \vdash_{Exp} lsta(\theta LstAExp) : valueT\ \{\ btyp\ \}} \quad \text{(LstA2)}$$

References: BasicType p. 73; $is\_arr\_tmark_{\delta}$ p. 283; $array\_arity_{\delta}$ p. 288; *nameT* p. 73; $ancestorof_{\delta}$ p. 281; $array\_index\_tmark_{\delta}$ p. 289; *uintT* p. 73; *valueT* p. 73

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 Static Semantics

1. The name *distyp* must identify a discrete type or subtype (*perhaps not a subtype, see* **SLI WJ005**).

2. The value *val* must be a well-formed expression with the type *vtyp* which is compatible with *distyp*.

3. The successor expression denotes a value with a unique type, which is the ancestor of *distyp*.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SuccExp;\ vtyp : ExpType;\ styp : BasicType \mid \\ \qquad is\_discrete\_tmark_{\alpha}\ distyp \wedge \\ \qquad vtyp\ compat_{\delta}\ distyp \wedge \\ \qquad styp = nameT\ (ancestorof_{\delta}\ distyp) \bullet \\ \delta \vdash_{Exp} val : vtyp\end{array}}{\delta \vdash_{Exp} succ(\theta SuccExp) : valueT\ \{\ styp\ \}} \quad \text{(Succ)}$$

References: ExpType p. 73; *BasicType* p. 73; *is_discrete_tmark*$_{\delta}$ p. 285; *compat*$_{\delta}$ p. 76; *nameT* p. 73; *ancestorof*$_{\delta}$ p. 281; *valueT* p. 73

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)	$pred \langle\!\langle\ distyp \mapsto dot(traffic, colour),\ val \mapsto nam\ (simp\ orange)\ \rangle\!\rangle$

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 Static Semantics

1. The name *distyp* must identify a discrete type or subtype (*perhaps not a subtype, see* **SLI WJ005**).

2. The value *val* must be a well-formed expression with the type *vtyp* which is compatible with *distyp*.

3. The successor expression denotes a value with a unique type, which is the ancestor of *distyp*.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PredExp;\ vtyp : ExpType;\ ptyp : BasicType \mid \\ \qquad is\_discrete\_tmark_{\alpha}\ distyp \wedge \\ \qquad vtyp\ compat_{\delta}\ distyp \wedge \\ \qquad styp = nameT\ (ancestorof_{\delta}\ distyp) \bullet \\ \delta \vdash_{Exp} val : vtyp\end{array}}{\delta \vdash_{Exp} pred(\theta PredExp) : valueT\ \{\ ptyp\ \}} \quad \text{(Pred)}$$

References: ExpType p. 73; *BasicType* p. 73; $is\_discrete\_tmark_{\delta}$ p. 285; $compat_{\delta}$ p. 76; *nameT* p. 73; $ancestorof_{\delta}$ p. 281; *valueT* p. 73

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)	$pos\ \langle\!\langle\ distyp \mapsto dot(traffic, colour),\ val \mapsto nam\ (simp\ red)\ \rangle\!\rangle$

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 Static Semantics

1. The name *distyp* must identify a discrete type or subtype (*perhaps not a subtype, see* **SLI WJ005**).

2. The value *val* must be a well-formed expression with a type compatible with *distyp*.

3. The position expression is a value expression with the unique type universal integer.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PosExp;\ vtyp : ExpType \mid \\ \qquad is\_discrete\_tmark_\delta\ distyp \wedge \\ \qquad vtyp\ compat_\delta\ distyp \bullet \\ \quad \delta \vdash_{Exp} val : vtyp\end{array}}{\delta \vdash_{Exp} pos(\theta PosExp) : valueT\ \{\ uint\ \}} \qquad \text{(Pos)}$$

References: ExpType p. 73; *is_discrete_tmark*$_\delta$ p. 285; *compat*$_\delta$ p. 76; *valueT* p. 73

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)	$valu \langle\!\langle\ distyp \mapsto dot(traffic, colour),\ val \mapsto lint\ 1\ \rangle\!\rangle$

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 Static Semantics

1. The name *distyp* must identify a discrete type or subtype (*perhaps not a subtype, see* **SLI WJ005**).
2. The value *val* must be a well-formed expression of any integer type.
3. The expression is a value with the unique named type which is the "ancestor" of *distyp*.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ ValuExp;\ vtyp : ExpType;\ utyp : BasicType \mid \\ \qquad is\_discrete\_tmark_\delta\ distyp \wedge \\ \qquad is\_int\_exp_\delta\ vtyp \wedge \\ \qquad utyp = nameT\ (ancestorof_\delta\ distyp) \bullet \\ \delta \vdash_{Exp} val : vtyp\end{array}}{\delta \vdash_{Exp} valu(\theta ValuExp) : valueT\ \{\ utyp\ \}} \quad \text{(Valu)}$$

References: *ExpType* p. 73; *BasicType* p. 73; $is\_discrete\_tmark_\delta$ p. 285; $is\_int\_exp_\delta$ p. 295; *nameT* p. 73; $ancestorof_\delta$ p. 281; *valueT* p. 73

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 ::= ... | *size*⟪*IdDot*⟫

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 proceed 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	139
PDecl	Private Declarations	10	149
SDecl	Subprogram Declarations	12	185
KDecl	Package Declarations	13	205
FDecl	Subprogram Definitions	14	207
YDecl	Subprogram and Package Bodies	15	215

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 well-formation 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.

Well-Formation Predicate

The well-formation of a visible basic declaration scope is specified by a relation between the environment, the scope and a modified environment. The declaration of this relation is:

$$\_ \vdash_{VBasic} \_ \Longrightarrow \_ \subseteq Env \times VBasic \times Env$$

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*.

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

8.3.2 Static Semantics

Each of the allowed forms of declaration must be well-formed according to the corresponding rule.

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\ b : BDecl \bullet \\ \quad \delta \vdash_{BDecl} b \Longrightarrow \delta'\end{array}}{\delta \vdash_{VBasic} vbasic\ b \Longrightarrow \delta'} \quad \text{(VBasic)}$$

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\ p : PDecl \bullet \\ \quad \delta \vdash_{PDecl} p \Longrightarrow \delta'\end{array}}{\delta \vdash_{VBasic} vpriv\ p \Longrightarrow \delta'} \quad \text{(VPriv)}$$

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\ s : SDecl \bullet \\ \quad \delta \vdash_{SDecl}\ s \Longrightarrow \delta'\end{array}}{\delta \vdash_{VBasic}\ vbasic\ s \Longrightarrow \delta'} \qquad \text{(VSub)}$$

A sequence of declarations is well-formed if both declarations are well-formed.

$$\frac{\begin{array}{l}\forall\, \delta, \delta', \delta'' : Env;\ u, v : VBasic \bullet \\ \quad \delta \vdash_{VBasic}\ u \Longrightarrow \delta' \\ \quad \delta' \vdash_{VBasic}\ v \Longrightarrow \delta''\end{array}}{\delta \vdash_{VBasic}\ vseq\ (u, v) \Longrightarrow \delta''} \qquad \text{(VSeq)}$$

The null declaration is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{VBasic}\ vnull \Longrightarrow \delta} \qquad \text{(VNull)}$$

References: *BDecl* p. 139; *PDecl* p. 149; *SDecl* p. 185; $\vdash_{BDecl}$ p. 139; $\vdash_{PDecl}$ p. 149; $\vdash_{SDecl}$ p. 186

8.4 Private Basic Declarations

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

Well-Formation Predicate

The well-formation of a private basic declaration scope is specified by a relation between the environment, the scope and a modified environment. The declaration of this relation is:

$$\_ \vdash_{PBasic} \_ \Longrightarrow \_ \subseteq Env \times PBasic \times Env$$

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 Static Semantics

There are two rules for the well-formation of a basic declaration in a private part:

1. The declaration can be well-formed as the full-declaration of a constant or type, corresponding to a deferred or private declaration in the visible part.

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\ b : BDecl \bullet \\ \quad \delta \vdash_{BDeclFull} b \Longrightarrow \delta'\end{array}}{\delta \vdash_{VBasic} pbasic\ b \Longrightarrow \delta'} \qquad \text{(PBasic1)}$$

2. The declaration can be well-formed as a standard declaration, without a corresponding declaration in the visible part.

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\ b : BDecl \bullet \\ \quad \delta \vdash_{BDecl} b \Longrightarrow \delta'\end{array}}{\delta \vdash_{VBasic} pbasic\ b \Longrightarrow \delta'} \qquad \text{(PBasic2)}$$

A sequence of declarations is well-formed if both declarations are well-formed.

$$\frac{\begin{array}{l}\forall\, \delta, \delta', \delta'' : Env;\ u, v : PBasic \bullet \\ \quad \delta \vdash_{PBasic} u \Longrightarrow \delta' \\ \quad \delta' \vdash_{PBasic} v \Longrightarrow \delta''\end{array}}{\delta \vdash_{PBasic} pseq\ (u, v) \Longrightarrow \delta''} \quad \text{(PSeq)}$$

The null declaration is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{PBasic} pnull \Longrightarrow \delta} \quad \text{(PNull)}$$

References: *BDecl* p. 139; $\vdash_{BDeclFull}$ p. 139; $\vdash_{BDecl}$ p. 139

8.5 Subprogram Body Basic Declarations

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

Well-Formation Predicate

The well-formation of the basic declaration scope of a subprogram body is specified by a relation between the environment, the scope and a modified environment. The declaration of this relation is:

$$\_ \vdash_{KBasic} \_ \Longrightarrow \_ \subseteq Env \times SBasic \times Env$$

8.5.1 Abstract Syntax

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

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

8.5.2 Static Semantics

The well-formation of the different forms of declaration is determined by the appropriate rule.

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\; b : BDecl \bullet \\ \quad \delta \vdash_{BDecl} b \Longrightarrow \delta'\end{array}}{\delta \vdash_{SBasic} sbasic\ b \Longrightarrow \delta'} \quad \text{(SBasic)}$$

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\; k : KDecl \bullet \\ \quad \delta \vdash_{KDecl} k \Longrightarrow \delta'\end{array}}{\delta \vdash_{SBasic} spak\ k \Longrightarrow \delta'} \quad \text{(SPak)}$$

A sequence of declarations is well-formed if both declarations are well-formed.

$$\frac{\begin{array}{l}\forall\, \delta, \delta', \delta'' : Env;\ u, v : SBasic \bullet \\ \quad \delta \vdash_{SBasic} u \Longrightarrow \delta' \\ \quad \delta' \vdash_{SBasic} v \Longrightarrow \delta''\end{array}}{\delta \vdash_{SBasic} sseq\ (u, v) \Longrightarrow \delta''} \quad \text{(SSeq)}$$

The null declaration is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{SBasic} snull \Longrightarrow \delta} \quad \text{(SNull)}$$

8.6 Package Body Basic Declarations

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

Well-Formation Predicate

The well-formation of the basic declaration scope of a package body is specified by a relation between the environment, the scope and a modified environment. The declaration of this relation is:

$$\_ \vdash_{KBasic} \_ \Longrightarrow \_ \subseteq Env \times KBasic \times Env$$

8.6.1 Abstract Syntax

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

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

8.6.2 Static Semantics

There are two possible rules for the well-formation of a basic declaration in a package body:

1. The declaration of a variable which is an own variable of the package, is well-formed.

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\; b : BDecl \bullet \\ \quad \delta \vdash_{BDeclOwn} b \Longrightarrow \delta'\end{array}}{\delta \vdash_{KBasic} kbasic\ b \Longrightarrow \delta'} \qquad \text{(KBasic1)}$$

2. A standard declaration is well-formed.

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\; b : BDecl \bullet \\ \quad \delta \vdash_{BDecl} b \Longrightarrow \delta'\end{array}}{\delta \vdash_{KBasic} kbasic\ b \Longrightarrow \delta'} \qquad \text{(KBasic2)}$$

The well-formation of an embedded package declaration is determined by the appropriate rule.

$$\frac{\forall\, \delta, \delta' : Env;\ k : KDecl \bullet \quad \delta \vdash_{KDecl}\ k \Longrightarrow \delta'}{\delta \vdash_{KBasic}\ kpak\ k \Longrightarrow \delta'} \qquad \text{(KPak)}$$

A sequence of declarations is well-formed if both declarations are well-formed.

$$\frac{\forall\, \delta, \delta', \delta'' : Env;\ u, v : KBasic \bullet \quad \delta \vdash_{KBasic}\ u \Longrightarrow \delta' \quad \delta' \vdash_{KBasic}\ v \Longrightarrow \delta''}{\delta \vdash_{KBasic}\ kseq\ (u, v) \Longrightarrow \delta''} \qquad \text{(KSeq)}$$

The null declaration is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{KBasic}\ knull \Longrightarrow \delta} \qquad \text{(KNull)}$$

References: *BDecl* p. 139; *KDecl* p. 205; $\vdash_{BDeclOwn}$ p. 139; $\vdash_{BDecl}$ p. 139

8.7 Subprogram Later Declarations

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

Well-Formation Predicate

The well-formation of the later declaration scope of a subprogram body is specified by a relation between the environment, the scope and a modified environment. The declaration of this relation is:

$$\_ \vdash_{SLater} \_ \Longrightarrow \_ \subseteq Env \times SLater \times Env$$

8.7.1 Abstract Syntax

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

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

8.7.2 Static Semantics

Each of the allowed forms of declaration must be well-formed according to the corresponding rule.

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\; y : YDecl \bullet \\ \quad \delta \vdash_{YDecl} y \Longrightarrow \delta'\end{array}}{\delta \vdash_{SLater} sbody\ y \Longrightarrow \delta'} \qquad \text{(SLBody)}$$

A sequence of declarations is well-formed if both declarations are well-formed.

$$\frac{\begin{array}{l}\forall\, \delta, \delta', \delta'' : Env;\; u, v : SLater \bullet \\ \quad \delta \vdash_{SLater} u \Longrightarrow \delta' \\ \quad \delta' \vdash_{SLater} v \Longrightarrow \delta''\end{array}}{\delta \vdash_{SLater} sseq\ (u, v) \Longrightarrow \delta''} \qquad \text{(SLSeq)}$$

The null declaration is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{SLater}\ snull \Longrightarrow \delta} \qquad \text{(SLNull)}$$

References: *YDecl* p. 215; $\vdash_{YDecl}$ p. 215

8.8 Package Later Declarations

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

Well-Formation Predicate

The well-formation of the later declaration scope of a package body is specified by a relation between the environment, the scope and a modified environment. The declaration of this relation is:

$$\_ \vdash_{KLater} \_ \Longrightarrow \_ \subseteq Env \times KLater \times Env$$

8.8.1 Abstract Syntax

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

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

8.8.2 Static Semantics

Each of the allowed forms of declaration must be well-formed according to the corresponding rule.

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\ f : FDecl \bullet \\ \quad \delta \vdash_{FDecl} f \Longrightarrow \delta'\end{array}}{\delta \vdash_{KLater} kdefn\ f \Longrightarrow \delta'} \quad \text{(KLDefn)}$$

$$\frac{\begin{array}{l}\forall\, \delta, \delta' : Env;\ y : YDecl \bullet \\ \quad \delta \vdash_{YDecl} y \Longrightarrow \delta'\end{array}}{\delta \vdash_{KLater} kbody\ y \Longrightarrow \delta'} \quad \text{(KLBody)}$$

A sequence of declarations is well-formed if both declarations are well-formed.

$$\frac{\begin{array}{l}\forall\, \delta, \delta', \delta'' : Env;\ u, v : KLater \bullet \\ \quad \delta \vdash_{KLater} u \Longrightarrow \delta' \\ \quad \delta' \vdash_{KLater} v \Longrightarrow \delta''\end{array}}{\delta \vdash_{KLater} kseq\ (u, v) \Longrightarrow \delta''} \quad \text{(KLSeq)}$$

The null declaration is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{KLater} knull \Longrightarrow \delta} \quad \text{(KLNull)}$$

References: *FDecl* p. 207; *YDecl* p. 215; $\vdash_{FDecl}$ p. 207; $\vdash_{YDecl}$ p. 215

Chapter 9

Basic Declarations

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

Syntax Constructor	Description	Page
const	constant	140
var	variable	143
ftype	full type	145
stype	subtype	147

The Well-Formation of Basic Declarations

The well-formation of a declaration is defined by a relation between the environment, the declaration and a modified environment. The declaration of this relation is:

$$\_ \vdash_{BDecl} \_ \Longrightarrow \_ \subseteq Env \times BDecl \times Env$$

Thus the well-formation predicate:

$$\delta \vdash_{BDecl} b \Longrightarrow \delta'$$

can be read as "declaration b is well-formed in the initial environment δ, with the modified environment δ'".

The additional well-formation relations $\vdash_{BDeclOwn}$ and $\vdash_{BDeclFull}$, which apply to the declaration of own variables and the full declaration of private types or deferred constants, have the same declarations.

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;	$const \langle\!\langle\ cid \mapsto C,\ type \mapsto id\ T,\ exp \mapsto lint\ 0\ \rangle\!\rangle$

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 Static Semantics

Standard Constant Declaration

1. The identifier (*cid*) must be unused.

2. The type mark (*type*) must identify a visible type or subtype, which is not unconstrained nor incompletely declared.

3. The expression used to specify the value of the constant must be well-formed with type *etype* and constant evaluating to *val*.

4. The type of the expression (*etype*) must be compatible with the type mark.

5. The value of the expression (*val*) must belong to the values of the type mark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ ConstBDecl;\ etype : ExpType;\ val : Val \mid \\ \qquad cid \notin \delta.usedids \land \\ \qquad is\_visible\_tmark_\delta\ type \land \\ \qquad \neg\ is\_unconstrained\_tmark_\delta\ type \land \\ \qquad \neg\ is\_incomplete\_tmark_\delta\ type \land \\ \qquad etype\ compat_\delta\ type \land \\ \qquad val \in scalar\_tmark\_range_\delta\ type \bullet \\ \\ \delta \vdash_{Exp} exp : etype \\ \delta \vdash exp \Longrightarrow_{\text{ConstEval}} val\end{array}}{\delta \vdash_{BDecl} const(\theta ConstBDecl) \Longrightarrow \delta'} \quad \text{(Const1)}$$

where

$$\begin{array}{l}\delta' == \delta[\ \ usedids := \delta.usedids \cup \{cid\}, \\ \qquad\qquad contypes := \delta.contypes \cup \{cid \mapsto type\}, \\ \qquad\qquad convals := \delta.convals \cup \{cid \mapsto val\}\ ]\end{array}$$

Full-Declaration of Deferred Constant

1. The identifier (*cid*) must be that of a deferred constant.
2. The type mark (*type*) must be the same as that used in the deferred declaration and the type must be fully-declared.
3. The expression used to specify the value of the constant must be well-formed with type *etype* and constant evaluating to *val*.
4. The type of the expression (*etype*) must be compatible with the type mark.
5. The value of the expression (*val*) must belong to the values of the type mark.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ ConstBDecl;\ etype : ExpType;\ val : Val \mid \\ \qquad cid \in \delta.defcons \land \\ \qquad \delta.contypes\ cid = type \land \\ \qquad \neg\ is\_unconstrained\_tmark_\delta\ type \land \\ \qquad \neg\ is\_incomplete\_tmark_\delta\ type \land \\ \qquad etype\ compat_\delta\ type \land \\ \qquad val \in scalar\_tmark\_range_\delta\ type \bullet \\ \\ \delta \vdash_{Exp} exp : etype \\ \delta \vdash exp \Longrightarrow_{\text{ConstEval}} val\end{array}}{\delta \vdash_{BDecl} const(\theta ConstBDecl) \Longrightarrow \delta'} \quad \text{(Const2)}$$

where

$$\delta' == \delta[\ \ defcons := \delta.defcons \setminus \{cid\},\ \ convals := \delta.convals \cup \{cid \mapsto val\}\ ]$$

References: *Exp* p. 71; *ExpType* p. 73; *Val* p. 6; $is\_visible\_tmark_\delta$ p. 281; $is\_unconstrained\_tmark_\delta$ p. 283; $is\_incomplete\_tmark_\delta$ p. 286; $compat_\delta$ p. 76; $scalar\_tmark\_range_\delta$ p. 296; $\vdash_{Exp}$ p. 71; $\Longrightarrow_{\text{ConstEval}}$ p. 298

Note: The rules for static evaluation of constant values are not given explicitly here; they are analogous to a subset of the dynamic semantic rules for expression evaluation (though no reference to store is needed, by the rules of SPARK), though real literals are evaluated in a precise, textual manner.

9.2 Variables

All variables are declared using a named type.

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

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 Static Semantics

There are two cases depending on whether or not the variable has already appeared in an **own** variable annotation.

Standard Variable Declaration

1. The variable identifier (*vid*) must not already be declared.
2. The type mark (*type*) must identify a visible type or subtype, which is not unconstrained nor incompletely declared.
3. In the modified environment δ':
   (a) the identifier *vid* is no longer available,
   (b) the variable *vid* is declared with type *type* and both read and write access modes.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ VarBDecl \mid \\ \quad vid \notin \delta.usedids \land \\ \quad is\_visible\_tmark_\delta\ type \land \\ \quad \neg\ is\_unconstrained\_tmark_\delta\ type \land \\ \quad \neg\ is\_incomplete\_tmark_\delta\ type \bullet\end{array}}{\delta \vdash_{BDecl} var(\theta\, VarBDecl) \Longrightarrow \delta'} \qquad \text{(Var1)}$$

where

$$\delta' == \delta \ [\ usedids := \delta.usedids \cup \{vid\},\ vartypes := \delta.vartypes \cup \{vid \mapsto type\},\ varmodes := \delta.varmodes \cup \{vid \mapsto rd, vid \mapsto wr\}\ ]$$

Own Variable Declaration

1. The variable identifier (*vid*) must be the name of an own variable, whose declaration is allowed in the current scope.
2. The type mark (*type*) must identify a visible type or subtype, which is not unconstrained nor incompletely declared.
3. In the modified environment δ', the variable *vid* is declared with type *type*.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ VarBDecl \mid \\ \quad vid \in \delta.ownvars \land \\ \quad vid \notin \mathrm{dom}\, \delta.vartypes \land \\ \quad is\_visible\_tmark_\delta\ type \land \\ \quad \neg\ is\_unconstrained\_tmark_\delta\ type \land \\ \quad \neg\ is\_incomplete\_tmark_\delta\ type \bullet\end{array}}{\delta \vdash_{BDeclOwn} var(\theta\, VarBDecl) \Longrightarrow \delta'} \qquad \text{(Var2)}$$

where

$$\delta' == \delta \ [\ vartypes := \delta.vartypes \cup \{vid \mapsto type\}\ ]$$

References: *is_visible_tmark*$_\delta$ p. 281; *is_unconstrained_tmark*$_\delta$ p. 283; *is_incomplete_tmark*$_\delta$ p. 286; *Env* p. 12

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;	$ftype \langle\!\langle\ tid \mapsto T,\ def \mapsto int \langle\!\langle\ lower \mapsto lint\ 1,\ upper \mapsto lint\ 9\ \rangle\!\rangle\ \rangle\!\rangle$

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 Static Semantics

There are two cases: the standard case and full-declaration of a type declared as **private** in the visible part of a package specification.

Standard Type Declaration

1. The identifier for the type must be unused.

2. The type definition must be well-formed in the initial environment with the name of the type *tid* used.

$$\frac{\begin{array}{l}\forall \delta, \delta_t : Env;\ FTypeBDecl \mid \\ \qquad tid \notin \delta.usedids \bullet \\ \delta', tid \vdash_{TypDef} def \Longrightarrow \delta_t\end{array}}{\delta \vdash_{BDecl} ftype(\theta FTypeBDecl) \Longrightarrow \delta_t} \quad \text{(FType1)}$$

where

$$\delta' == \delta\ [usedids := \delta.usedids \cup \{tid\}]$$

Full Declaration of Private Types Two case are required, depending on whether the private type is limited or not.

1. The identifier for the type must identify a directly visible private type.
2. The type definition must be well-formed.
3. The type must not be limited.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_t : Env;\ FTypeBDecl \mid \\ \quad (\exists\, t : Id \mid \\ \qquad tid = id\ t \wedge \\ \qquad \delta.type\ t = privT) \wedge \\ \quad \neg\ is\_limited\_tmark_{\delta_t}\ type \bullet \\ \delta', tid \vdash_{TypDef}\ def \Longrightarrow \delta_t\end{array}}{\delta \vdash_{BDecl}\ ftype(\theta FTypeBDecl) \Longrightarrow \delta_t} \quad \text{(FType2)}$$

where

$$\delta' == \delta\ [types := \{tid\} \lhd \delta.types]$$

1. The identifier for the type must identify a directly visible limited private type.
2. The type definition must be well-formed.
3. The type may be limited, unless it has been used in the declaration of a formal of mode **out** [AARM, §7.4.4 ¶4].

$$\frac{\begin{array}{l}\forall\, \delta, \delta_t : Env;\ FTypeBDecl \mid \\ \quad (\exists\, t : Id \mid \\ \qquad tid = id\ t \wedge \\ \qquad \delta.type\ t = limT) \wedge \\ \quad ((\exists\, p : \mathrm{dom}\, \delta.pdecls;\ i : \mathrm{dom}\, \delta.pdecls\ p \bullet \\ \qquad (\delta.pdecls\ p).ftypes\ i = tid \wedge \\ \qquad (\delta.pdecls\ p).fmodes(\!|\{i\}|\!) = \{wr\}) \Rightarrow \\ \quad \neg\ is\_limited\_tmark_{\delta_t}\ type) \bullet \\ \delta', tid \vdash_{TypDef}\ def \Longrightarrow \delta_t\end{array}}{\delta \vdash_{BDecl}\ ftype(\theta FTypeBDecl) \Longrightarrow \delta_t} \quad \text{(FType3)}$$

where

$$\delta' == \delta\ [types := \{tid\} \lhd \delta.types]$$

References: *TypDef* p. 13; *Env* p. 12; $\vdash_{TypDef}$ p. 14; *is_limited_tmark*$_\delta$ p. 286

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 ;	$stype \langle\!\langle\ sid \mapsto S,\ def \mapsto ran \langle\!\langle\ parent \mapsto id\ T,\ lower \mapsto lint\ 1,\ upper \mapsto lint\ 5\ \rangle\!\rangle\ \rangle\!\rangle$

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 Static Semantics

1. The identifier of the subtype must not be already declared.
2. The subtype constraint must be well-formed.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_s : Env;\ STypeBDecl \mid \\ \qquad sid \notin \delta.usedids \bullet \\ \delta', sid \vdash_{SubDef} def \Longrightarrow \delta_s\end{array}}{\delta \vdash_{BDecl} stype(\theta STypeBDecl) \Longrightarrow \delta_s} \quad \text{(SType)}$$

where

$$\delta' == \delta\ [\ usedids := \delta.usedids \cup \{sid\}\ ]$$

References: *SubDef* p. 41; *Env* p. 12; $\vdash_{SubDef}$ p. 41

Chapter 10

Private Declarations

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

Syntax Constructor	Description	Page
dconst	deferred constant	150
priv	private type	151
plim	private limited type	152

Note that these declarations are called *private* declarations because they concern the dcelaration of private types and constants of private types. Such declarations appear in the *visible* part of a package specification.

The Well-Formation of Declarations

The well-formation of a private declaration is defined by a relation between the environment, the declaration and a modified environment. The declaration of this relation is:

$$\_ \vdash_{PDecl} \_ \Longrightarrow \_ \subseteq Env \times PDecl \times Env$$

Thus the well-formation predicate:

$$\delta \vdash_{PDecl} decl \Longrightarrow \delta'$$

can be read as "declaration *decl* is well-formed in the initial environment δ, with the modified environment δ'".

10.1 Deferred Constant

A constant of a private type is declared without a value; this is known as a deferred constant declaration. The value is supplied by another declaration, in the private part.

Syntax Example	A.S. Representation
X : **constant** T;	$dconst \langle\!\| \; cid \mapsto X, \; type \mapsto id \; T \; \|\!\rangle$

10.1.1 Abstract Syntax

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

$$DConstPDecl \mathrel{\widehat=} [cid : Id; \; type : IdDot]$$

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.

$$PDecl ::= \ldots \mid dconst\langle\langle DConstPDecl \rangle\rangle$$

10.1.2 Static Semantics

1. The identifier (cid) must be unused.
2. The type mark ($type$) must identify a private or limited private type, which is directly visible.

$$\frac{\begin{array}{l}\forall \delta : Env; \; DConstPDecl \mid \\ \quad cid \notin \delta.usedids \land \\ \quad (\exists t : Id \bullet type = id \; t \land \\ \qquad \delta.types \; t \in \{privT, limT\}) \bullet\end{array}}{\delta \vdash_{PDecl} dconst(\theta DConstPDecl) \Longrightarrow \delta'} \quad \text{(DConst)}$$

where

$$\begin{array}{l}\delta' = \delta[\; usedids := \delta.usedids \cup \{cid\}, \\ \qquad contypes := \delta.contypes \cup \{cid \mapsto type\}, \\ \qquad defcons := \delta.defcons \cup \{cid\} \;]\end{array}$$

References: *privT* p. 15; *limT* p. 15

10.2 Private Type

A private type is declared without a definition; the definition is given by another declaration in the private part of the package specification.

Syntax Example	A.S. Representation
type Key **is private** ;	*priv Key*

10.2.1 Abstract Syntax

$PDecl ::= \ldots \mid priv\langle\langle Id \rangle\rangle$

10.2.2 Static Semantics

1. The identifier must be unused.

$$\frac{\forall\, \delta : Env;\ pid : Id \mid \\ \qquad pid \notin \delta.usedids}{\delta \vdash_{PDecl}\ priv\ pid \Longrightarrow \delta'} \qquad \text{(Priv)}$$

where

$$\delta' = \delta[\ \ usedids := \delta.usedids \cup \{pid\}, \\ \qquad\quad types := \delta.types \cup \{pid \mapsto privT\}\ ]$$

References: *privT* p. 15

10.3 Private Limited Type

A private limited type is declared without a definition; the definition is given by another declaration in the private part of the specification.

Syntax Example	A.S. Representation
type Key **is private limited** ;	*plim Key*

10.3.1 Abstract Syntax

$$PDecl ::= \ldots \mid plim\langle\langle Id \rangle\rangle$$

10.3.2 Static Semantics

1. The identifier must be unused.

$$\frac{\forall\, \delta : Env;\ lid : Id \mid \\ \qquad lid \notin \delta.usedids}{\delta \vdash_{PDecl}\ plim\ lid \Longrightarrow \delta'} \qquad \text{(Priv)}$$

where

$$\delta' = \delta[\ \ usedids := \delta.usedids \cup \{lid\}, \\ \qquad\quad types := \delta.types \cup \{lid \mapsto limT\}\ ]$$

References: *limT* p. 15

Chapter 11

Statements

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

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

Well-Formation Predicate

The well-formation of a statement is specified by a relation between the environment and the statement. The declaration of this relation is:

$$\_ \vdash_{Stmt} \_ \subseteq Env \times Stmt$$

Thus the well-formation predicate:

$$\delta \vdash_{Stmt} stmt$$

can be read as "statement *stmt* is well-defined in environment δ".

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 Static Semantics

1. Both statements must be well-formed in the initial environment.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ s, t : Stmt \bullet \\ \quad \delta \vdash_{Stmt} s \\ \quad \delta \vdash_{Stmt} t\end{array}}{\delta \vdash_{Stmt} scomp\ (s, t)} \quad \text{(SComp)}$$

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 Static Semantics

1. The null statement is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{Stmt} null} \qquad \text{(Null)}$$

11.3 Assignment

An assignment statement assigns a value to a variable object.

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

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 Static Semantics

1. The name must be a well-formed object, with write access mode.
2. The expression must be well-formed.
3. The type of the expression must be compatible with the type of the name. (The definition of type compatibility ensures that there are no expressions which are compatible with names identifying unconstrained objects).
4. The type of the name must not be limited.

$$\frac{\begin{array}{l}\forall \delta : Env;\ AsgnStmt;\ ObjDict;\ etype : ExpType \mid \\ \quad wr \in modes \land \\ \quad etype\ compat_\delta\ type \land \\ \quad \neg\ is\_limited\_tmark_\delta\ type \bullet \\ \delta \vdash_{Name} var : objnam\ (\theta\, ObjDict) \\ \delta \vdash_{Exp} val : etype\end{array}}{\delta \vdash_{Stmt} asgn(\theta\, AsgnStmt)} \quad \text{(Asgn)}$$

References: *Name* p. 53; *Exp* p. 71; *ObjDict* p. 54; *ExpType* p. 73; $is\_limited\_tmark_\delta$ p. 286; $compat_\delta$ p. 76; $\vdash_{Name}$ p. 53; $\vdash_{Exp}$ p. 71

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 ;
	$if \langle\!\langle\ cond \mapsto binop \langle\!\langle\ larg \mapsto nam\ (simp\ v),$
	$op \mapsto eq,$
	$rarg \mapsto lint\ 0\ \rangle\!\rangle,$
	$ifpart \mapsto asgn \langle\!\langle\ var \mapsto simp\ v,$
	$val \mapsto lint\ 1\ \rangle\!\rangle\ \rangle\!\rangle$

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*).

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

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

11.4.2 Static Semantics

1. The condition must a well-formed expression with boolean type.
2. The *ifpart* statement must be well-formed.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ IfStmt \bullet \\ \quad \delta \vdash_{Exp} cond : booltype \\ \quad \delta \vdash_{Stmt} ifpart\end{array}}{\delta \vdash_{Stmt} if(\theta IfStmt)} \quad \text{(If)}$$

References: *Exp* p. 71; *booltype* p. 295

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

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 ;
	≡	**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 Static Semantics

1. The condition must a well-formed expression with boolean type.
2. The *ifpart* and *elsepart* statements must be well-formed.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ IfElStmt \bullet \\ \quad \delta \vdash_{Exp} cond : booltype \\ \quad \delta \vdash_{Stmt} ifpart \\ \quad \delta \vdash_{Stmt} elsepart\end{array}}{\delta \vdash_{Stmt} if\,(\theta IfElStmt)} \quad \text{(If)}$$

References: Exp p. 71; *booltype* p. 295

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                  case ⟪ casindx ↦ nam (simp colour),
    when red =>                        alterns ↦ ⟨ ⟪ altexp ↦ nam (simp red),
        panic;                                     altstm ↦ ... ⟫,
    when blue | green =>                       ⟪ altexp ↦ nam (simp blue),
        null ;                                   altstm ↦ null  ⟫,
end case ;                                     ⟪ altexp ↦ nam (simp green),
                                                 altstm ↦ null  ⟫⟩ ⟫
```

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.

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

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

11.6.2 Static Semantics

The alternative of the case statements must cover the range of the type used in the case expression. But is this range that of a subtype or of a base type? In the Ada definition [LRM, §5.4 ¶4], names, type conversions and qualified expressions are specially identified as forms of expression for which the case alternative (may) cover the range of a subtype. We do not need these special cases, because the type mark in the *ExpType* of the case expression will be a subtype whenever the form of expression makes this appropriate.

1. The type (or subtype) of the case expression must be discrete.
2. The case "expression" may be a well-formed expression with a *unique* value type.
3. The case alternatives must exhaust the range of the type (possibly a subtype) of the expression.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ CaseStmt;\ type : IdDot \mid \\ \qquad is\_discrete\_tmark_\delta\ type \bullet \\ \delta \vdash_{Exp} casindx : valueT\ \{nameT\ type\} \\ \delta, type \vdash_{Alt} alterns \Longrightarrow (scalar\_tmark\_range_\delta\ type)\end{array}}{\delta \vdash_{Stmt} case\ (\theta CaseStmt)} \quad \text{(Cas)}$$

References: *Exp* p. 71; *is_discrete_tmark*$_\delta$ p. 285; $\vdash_{Exp}$ p. 71; *valueT* p. 73; *nameT* p. 73; $\vdash_{Alt}$ p. 161; *scalar_tmark_range*$_\delta$ p. 296

11.6.3 Case Alternatives

The well-formation of a case alternative depends on the type of the case expression, given by an element of *IdDot*. Each expression covers a certain set of values of the type.

$$\_,\_ \vdash_{Alt} \_ \Longrightarrow \_ \subseteq Env \times IdDot \times \mathrm{seq}\ CaseAltern \times \mathbb{P}\ Val$$

Empty List The empty list is always well-formed (note: it does not belong to the syntax of SPARK).

$$\frac{\forall\, \delta : Env;\ t : IdDot \bullet}{\delta, t \vdash_{Alt} \langle\rangle \Longrightarrow \varnothing} \quad \text{(Alt1)}$$

Range A case expression may be a range.

1. The case expression may be a well-formed range; its type must be compatible with the type *itype* of the case index expression.

2. The case expression must be static; the range of values to which the expression evaluates must not overlap with the values covered by the other alternatives.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ itype : IdDot;\ CaseAltern;\ cs : \mathrm{seq}\ CaseAltern;\\ altvals, vals : \mathbb{P}\ Val;\ btyps : \mathbb{P}\ BasicType \mid\\ \quad (\exists\, t : IdDot \mid nameT\ t \in btyps \wedge\\ \quad\quad ancestorof_\delta\ itype = ancestorof_\delta\ t) \wedge\\ \quad altvals \cap vals = \varnothing \bullet\\ \delta \vdash_{Stmt} altstm\\ \delta \vdash_{Exp} altexp : rangeT\ btyps\\ \delta \vdash altexp \Longrightarrow_{\mathrm{StaticEval}} rngval\ altvals\\ \delta, itype \vdash_{Alt} cs \Longrightarrow vals\end{array}}{\delta, itype \vdash_{Alt} \langle \theta\, CaseAltern \rangle \frown cs \Longrightarrow altvals \cup vals} \quad \text{(Alt2)}$$

Value A case expression may be a value.

1. The case expression may be a well-formed value; its type must be compatible with the type *itype* of the case index expression.

2. The expression must be static; it values must not belong to the set of values covered by the other alternatives.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ itype : IdDot;\ CaseAltern;\ cs : \mathrm{seq}\ CaseAltern;\\ altval : Val;\ vals : \mathbb{P}\ Val;\ btyps : \mathbb{P}\ BasicType \mid\\ \quad (valueT\ btyps)\ compat_\delta\ itype \wedge\\ \quad val \notin altvals \bullet\\ \delta \vdash_{Stmt} altstm\\ \delta \vdash_{Exp} altexp : valueT\ btyps\\ \delta \vdash altexp \Longrightarrow_{\mathrm{StaticEval}} altval\\ \delta, itype \vdash_{Alt} cs \Longrightarrow vals\end{array}}{\delta, itype \vdash_{Alt} \langle \theta\, CaseAltern \rangle \frown cs \Longrightarrow \{altval\} \cup vals} \quad \text{(Alt3)}$$

References: *Exp* p. 71; *Val* p. 6; *BasicType* p. 73; *nameT* p. 73; $ancestorof_\delta$ p. 281; $\vdash_{Exp}$ p. 71; *rangeT* p. 73; $\Longrightarrow_{\mathrm{StaticEval}}$ p. 298; $compat_\delta$ p. 76

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 ⟨‖ *casindx* ↦ *nam* (*simp colour*),
	alterns ↦ ⟨⟨‖ *altexp* ↦ *nam* (*simp red*),
	altstm ↦ ... ‖⟩⟩,
	othcase ↦ **null** ‖⟩

11.7.1 Abstract Syntax

The **others** clause is represented by a statement. *CaseAltern* is defined on page 160.

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

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

11.7.2 Static Semantics

1. The type (or subtype) of the case expression must be discrete.
2. The case "expression" may be a well-formed expression, with a *unique* value type. In this case the alternatives must belong to the range of the type (possibly a subtype) of the expression.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ CaseStmt;\ type : IdDot;\ altvals : \mathbb{P}\ Val \mid \\ \quad is\_discrete\_tmark_\delta\ type \land \\ \quad altvals \subset scalar\_tmark\_range_\delta\ type \bullet \\ \\ \delta \vdash_{Exp} casindx : valueT\ \{nameT\ type\} \\ \delta, type \vdash_{Alt} alterns \Longrightarrow altvals \\ \delta \vdash_{Stmt} othcase\end{array}}{\delta \vdash_{Stmt} case\ (\theta CaseStmt)} \quad \text{(CasO)}$$

References: *Val* p. 6; $is\_discrete\_tmark_\delta$ p. 285; $scalar\_tmark\_range_\delta$ p. 296; *valueT* p. 73; *nameT* p. 73; $\vdash_{Alt}$ p. 161

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;	$exitw\ binop \langle\!\langle\ larg \mapsto nam\ (simp\ v),\ op \mapsto eq,\ rarg \mapsto lint\ 1\ \rangle\!\rangle$
if safe **then** *statement*; **exit** ; **end if**	$ifexit \langle\!\langle\ cond \mapsto nam\ (simp\ safe),\ spart \mapsto \ldots\ \rangle\!\rangle$

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 Static Semantics

The well-formation of exit statements is described by a relation $\vdash_{XStmt}$, declared similarly to $\vdash_{Stmt}$.

1. The exit condition must be a well-formed boolean expression.

$$\frac{\forall\, \delta : Env;\ cond : Exp \bullet \quad \delta \vdash_{Exp} cond : booltype}{\delta \vdash_{XStmt} exitw\ cond} \qquad \text{(Exit1)}$$

2. The statement preceding the exit in a simple if must be well-formed.

$$\frac{\forall\, \delta : Env;\ IfExitStmt \bullet \quad \delta \vdash_{Exp} cond : booltype \quad \delta \vdash_{Stmt} spart}{\delta \vdash_{XStmt} ifexit(\theta IfExitStmt)} \qquad \text{(Exit2)}$$

References: *Exp* p. 71; *booltype* p. 295

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;
	$\langle\!\vert$ *loopstmt* $\mapsto$ *scomp* (*asgn* ..., *asgn* ...),
	loopexit $\mapsto$ *exitw* ... $\vert\!\rangle$

11.9.1 Abstract Syntax

Each segment of a loop has a statement followed by an exit statement (see Section 11.8).

$$LoopSeg \mathrel{\widehat=} [loopstmt : Stmt;\ loopexit : ExitStmt]$$

11.9.2 Static Semantics

The well-formation of a list of loop segments is described by the relation:

$$\_ \vdash_{LSeg} \_ \subseteq Env \times \mathrm{seq}\, LoopSeg$$

Empty List The empty list of loop segments is always well-formed. (The syntax prevents its use in a general loop).

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{LSeg} \langle\rangle} \qquad \text{(LSeg1)}$$

Non-Empty List A non-empty list of loop segments is well-formed if the subcomponents of the first element and the rest of the list are all well-formed.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ LoopSeg;\ ls : \mathrm{seq}\, LoopSeg \bullet \\ \quad \delta \vdash_{Stmt} loopstmt \\ \quad \delta \vdash_{XStmt} loopexit \\ \quad \delta \vdash_{LSeg} ls\end{array}}{\delta \vdash_{LSeg} \langle \theta LoopSeg \rangle \frown ls} \quad \text{(LSeg2)}$$

References: *ExitStmt* p. 164; $\vdash_{XStmt}$ p. 164

11.10 General Loop

A general loop has no iteration scheme.

Syntax Example	A.S. Representation

```
loop                          loop ⟪  segs ↦ ⟨ ⟪  loopstmt ↦ asgn ...,
    x := x + 1;                                   loopexit ↦ exitw ... ⟫,
    exit when x = 10;                           ⟪  loopstmt ↦ null,
    if y = 10 then                                loopexit ↦ ifexit ... ⟫ ⟩,
        y := y + 1;                 spart ↦ asgn ... ⟫
        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*).

$$LoopStmt \mathrel{\widehat=} [segs : \mathrm{seq}\, LoopSeg;\ spart : Stmt]$$

$$LoopStmt1 \mathrel{\widehat=} [LoopStmt \mid \#segs \neq 0]$$

$$Stmt ::= \ldots \mid loop\langle\langle LoopStmt1 \rangle\rangle$$

11.10.2 Static Semantics

The component loop segments and statement must be well-formed.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ LoopStmt1 \bullet \\ \quad \delta \vdash_{Stmt} spart \\ \quad \delta \vdash_{LSeg} segs\end{array}}{\delta \vdash_{Stmt} loop\ (\theta LoopStmt1)} \qquad \text{(Loop)}$$

References: *LoopSeg* p. 165; $\vdash_{LSeg}$ p. 165

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* ⟪ *cond* ↦ ..., *wpart* ↦ ... ⟫

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 Static Semantics

1. The condition must a well-formed expression with boolean type.
2. The *segs* must be a well-formed list of loop segments.
3. The *spart* must be a well-formed statement.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ WLoopStmt \bullet \\ \quad \delta \vdash_{Exp} cond : booltype \\ \quad \delta \vdash_{LSeg} segs \\ \quad \delta \vdash_{Stmt} spart\end{array}}{\delta \vdash_{Stmt} wloop(\theta\, WLoopStmt)} \qquad \text{(WLoop)}$$

Alternative Approach We can regard the while loop as a derived form in SPARK, since it can be expressed as a general loop using the following equivalence.

while b **loop**
  segments ;
  statement ;
end loop ;
	≡	**loop**
		  null ;
		  exit when not b;
		  segments ;
		  statement ;
		end loop ;

This transformation could be formally specified as follows:

$$
\begin{array}{|l}
transform\_while : Stmt \rightarrow Stmt \\
\hline
\forall\, l : LoopStmt;\ w : WLoopStmt;\ LoopSeg \mid \\
\qquad loopstmt = null \land \\
\qquad loopexit = exitw\ (unop\ (\mu\ UnExp \mid op = not \land arg = w.cond)) \land \\
\qquad l.segs = \langle \theta\, LoopSeg \rangle \frown w.segs \land \\
\qquad l.spart = w.spart \bullet \\
\quad transform\_while\ (wloop\ w) = loop\ l \\
\forall\, s : Stmt \mid s \notin \mathrm{ran}\ wloop \bullet \\
\quad transform\_while\ s = s
\end{array}
$$

References: *Exp* p. 71; *LoopStmt* p. 167; $\vdash_{Exp}$ p. 71; $\vdash_{LSeg}$ p. 165; *booltype* p. 295; *LoopSeg* p. 165; *UnExp* p. 100

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**	*floop* ⟪ *indxvar* ↦ *y*,
x := x + 1;	*indxtyp* ↦ *id T*,
end loop ;	*fwdrev* ↦ *forward*,
	fpart ↦ *asgn* ... ⟫

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 Static Semantics

1. The index variable must be unused in the current environment.
2. The index type must be a visible type which is discrete.
3. The component list of segments *segs* and the statement *spart* must be well-formed in the initial environment modified by the addition of the index variable, which has read-only access.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ FLoopStmt \mid \\ \qquad indxvar \notin \delta.usedids \wedge \\ \qquad is\_visible\_tmark_\delta\ indxtyp \wedge is\_discrete\_tmark_\delta\ indxtyp \bullet \\ \delta' \vdash_{Stmt} spart \\ \delta' \vdash_{LSeg} segs\end{array}}{\delta \vdash_{Stmt} floop\ (\theta FLoopStmt)} \quad \text{(For)}$$

where

$$\delta' = \delta[\ \begin{array}{l} usedids := usedids \cup \{indxvar\}, \\ varmodes := varmodes \cup \{indxvar \mapsto rd\}, \\ vartypes := vartypes \cup \{indxvar \mapsto indxtyp\}\ ] \end{array}$$

References: *LoopStmt* p. 167; *is_visible_tmark*$_\delta$ p. 281; *is_discrete_tmark*$_\delta$ p. 285; $\vdash_{LSeg}$ p. 165

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 Static Semantics

1. The index variable must be unused in the current environment.
2. The index type must be a visible type which is discrete.
3. The range must be a well-formed expression, which is a range of a type compatible with the index type.
4. The component list of segments *segs* and the statement *spart* must be well-formed in the initial environment modified by the addition of the index variable, which has read-only access.

$$\frac{\begin{array}{l}\forall \delta : Env;\ FLoopRStmt;\ btyps : \mathbb{P}\ BasicType \mid \\ \quad indxvar \notin \delta.usedids \wedge \\ \quad is\_visible\_tmark_\delta\ indxtyp \wedge is\_discrete\_tmark_\delta\ indxtyp \wedge \\ \quad (rangeT\ btyps)\ compat_\delta\ indxtyp \bullet \\ \delta \vdash_{Exp} rangeT\ btyps \\ \delta' \vdash_{Stmt} spart \\ \delta' \vdash_{LSeg} segs\end{array}}{\delta \vdash_{Stmt} floopr\ (\theta FLoopRStmt)} \quad \text{(ForR)}$$

where

$$\begin{aligned}\delta' = \delta[\ & usedids := usedids \cup \{indxvar\},\\ & varmodes := varmodes \cup \{indxvar \mapsto rd\},\\ & vartypes := vartypes \cup \{indxvar \mapsto indxtyp\}\ ]\end{aligned}$$

References: *FLoopStmt* p. 170; *Exp* p. 71; $is\_visible\_tmark_\delta$ p. 281; $is\_discrete\_tmark_\delta$ p. 285; $compat_\delta$ p. 76; $rangeT$ p. 73; $\vdash_{LSeg}$ p. 165

11.14 Loop Name

A loop may be named using a loop name.

Syntax Example	A.S. Representation
count: **loop** ... **end loop** count;	$slabel \langle\!\|\ slabel \mapsto count,$ $lstmt \mapsto \ldots,$ $elabel \mapsto count\ \|\!\rangle$

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 Static Semantics

1. The start and end label must be the same.
2. The identifier used as a label must be unused in the initial environment.
3. The labelled statement must be well-formed in the initial environment modified by the use of the label identifier.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SLabelStmt \mid \\ \qquad slabel = elabel \bullet \\ \delta' \vdash_{Stmt} lstmt\end{array}}{\delta \vdash_{Stmt} slabel\ (\theta SLabelStmt)} \qquad \text{(Lab)}$$

where

$$\delta' = \delta[\ usedids := usedids \cup \{slabel\}\ ]$$

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);	$callp \langle\!\langle\ pid \mapsto id\ switch,$ $actuals \mapsto \langle nam\ (simp\ x), nam\ (simp\ y)\rangle\ \rangle\!\rangle$

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 Static Semantics

1. The name *pid* must be the name of a callable procedure.
2. The list of actual parameters, converted to the *named association* format using the name specified for each position, must be well-formed, using the declared parameter types and modes. There must be an actual parameter for each formal parameter.
3. To prevent aliasing between the actual parameters and the global variables:
   (a) the global variables of the procedure *glvars* must not be updated in the actual parameter list,
   (b) the exported global variables of the procedure must not be referenced in the actual parameter list.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ CallPStmt;\ updates, refs : \mathbb{P}\ IdDot \mid \\ \quad is\_callable\_proc_\delta\ pid \land \\ \quad proc\_glvars_\delta\ pid \cap updates = \varnothing \land \\ \quad (proc\_exvars_\delta\ pid \setminus \\ \qquad id(\lvert \mathrm{dom}\ proc\_param\_ids_\delta\ pid \rvert)) \cap refs = \varnothing \land \\ \quad \#actuals = \#\ proc\_param\_ids_\delta\ pid \bullet \\ \delta, proc\_param\_tmarks_\delta\ pid, proc\_param\_modes_\delta\ pid \\ \quad \vdash_{Act} actuals' \Longrightarrow refs, updates\end{array}}{\delta \vdash_{Stmt} callp(\theta\, CallPStmt)} \quad \text{(CallP)}$$

where

$$actuals' == (\lambda\, i : \mathrm{dom}\, actuals \bullet (\mu\, NamedActual \mid formal = proc\_param\_ids_\delta\ pid\ i \land actual = actuals\ i))$$

References: *Exp* p. 71; *Env* p. 12; $is\_callable\_proc_\delta$ p. 293; $proc\_glvars_\delta$ p. 294; $proc\_exvars_\delta$ p. 294; $proc\_param\_ids_\delta$ p. 293; $proc\_param\_tmarks_\delta$ p. 293; $proc\_param\_modes_\delta$ p. 294; $\vdash_{Act}$ p. 178; *NamedActual* p. 178

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);	$calln \langle\!\langle\ pid \mapsto id\ switch,$ $actuals \mapsto \langle\ \langle\!\langle\ formal \mapsto fromv,$ $actual \mapsto nam\ (simp\ x)\ \rangle\!\rangle,$ $\langle\!\langle\ formal \mapsto tov,$ $actual \mapsto nam\ (simp\ y)\ \rangle\!\rangle\rangle\ \rangle\!\rangle$

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 Static Semantics

1. The name *pid* must be the name of a callable procedure.
2. The list of actual parameters must be well-formed, using the declared parameter types and modes. Note that an actual parameter list with repeated or missing formal parameters is not well-formed.
3. To prevent aliasing between the actual parameters and the global variables:
   (a) the global variables of the procedure (*glvars*) must not be updated in the actual parameter list,
   (b) the exported global variables of the procedure must not be referenced in the actual parameter list.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ CallNStmt;\ updates, refs : \mathbb{P}\ IdDot \mid \\ \quad is\_callable\_proc_\delta\ pid \land \\ \quad proc\_glvars_\delta\ pid \cap updates = \varnothing \land \\ \quad (proc\_exvars_\delta\ pid \setminus \\ \qquad id(\lvert \mathrm{dom}\ proc\_param\_ids_\delta\ pid \rvert)) \cap refs = \varnothing \bullet \\ \delta, proc\_param\_tmarks_\delta\ pid, proc\_param\_modes_\delta\ pid \\ \quad \vdash_{Act} actuals \Longrightarrow refs, updates\end{array}}{\delta \vdash_{Stmt} calln(\theta\, CallNStmt)} \quad \text{(CallN)}$$

References: *NamedActual* p. 178; *Env* p. 12; $is\_callable\_proc_{\delta}$ p. 293; $proc\_glvars_{\delta}$ p. 294; $proc\_exvars_{\delta}$ p. 294; $proc\_param\_ids_{\delta}$ p. 293; $proc\_param\_tmarks_{\delta}$ p. 293; $proc\_param\_modes_{\delta}$ p. 294; $\vdash_{Act}$ p. 178

11.17 Actual Parameter Lists

This section describes the well-formation of an actual parameter list. 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 174).

Syntax Example	A.S. Representation
switch(fromv => x, tov => y);	$\langle \langle\!\vert\ formal \mapsto fromv,$ $\quad actual \mapsto nam\ (simp\ x)\ \vert\!\rangle,$ $\langle\!\vert\ formal \mapsto tov,$ $\quad actual \mapsto nam\ (simp\ y)\ \vert\!\rangle\rangle$

11.17.1 Abstract Syntax

The formal parameter is an identifier; the actual parameter is an expression.

$$NamedActual \mathrel{\widehat=} [formal : Id;\ actual : Exp]$$

11.17.2 Static Semantics

The well-formation of a list of actual parameters is described by a relation between the environment, the types and modes of the corresponding formal parameters, the parameter list and sets of referenced and updated variables. The sets of variables are required to ensure that the parameter list does not contain aliasing. The declaration of this relation is:

$$\_,\_,\_ \vdash_{Act} \_ \Longrightarrow \_,\_ \subseteq Env \times Id \pfun IdDot \times Id \rightleftharpoons RdWr \times \\ \quad \mathrm{seq}\ NamedActual \times \mathbb{P}\ IdDot \times \mathbb{P}\ IdDot$$

Thus, the predicate

$$\delta, ftypes, fmodes \vdash_{Act} actuals \Longrightarrow refs, updates$$

can be read as "the list of parameters *actuals*, declared to have types *ftypes* and modes *fmodes*, is well-formed in environment δ, updating variables in *updates* and referencing those in *refs*".

Empty Parameter List is well-formed provided no further parameters are expected.

$$\frac{\forall\, \delta : Env \bullet}{\delta, \varnothing, \varnothing \vdash_{Act} \langle\rangle \Longrightarrow \varnothing, \varnothing} \qquad \text{(Act1)}$$

In-Parameter — Name A parameter of **in** mode may be a name. Since this may give rising to aliasing if the compiler uses "pass by reference", it counts as a reference to the object to which the name refers.

1. The formal parameter exists and is **in** mode.
2. The name must be well-formed and refer to an object.
3. The name is not updated elsewhere in the parameter list.
4. The name can be read.
5. The type of the name is compatible (nam_compat_δ) with the type mark of the the formal parameter.
6. The object name is added to the set of referenced variables.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NamedActual;\ as : \mathrm{seq}\, NamedActual;\ ObjDict;\ n : Name;\\ \quad ft : Id \pfun IdDot;\ fm : Id \rel RdWr;\ updates, refs : \mathbb{P}\, IdDot \mid\\ \qquad fm(\!|\{formal\}|\!) = rd \land actual = nam\ n \land\\ \qquad name \notin updates \land\\ \qquad rd \in mode \land\\ \qquad type\ nam\_compat_\delta\ (ft\ formal) \bullet\\ \quad \delta \vdash_{Name} n : objnam\ (\theta ObjDict)\\ \quad \delta, \{formal\} \ndres ft, \{formal\} \ndres fm \vdash_{Act} as \Longrightarrow refs, updates\end{array}}{\delta, ft, fm \vdash_{Act} \langle \theta NamedActual \rangle \frown as \Longrightarrow refs \cup \{name\}, updates} \quad \text{(Act2)}$$

Note that *type*, *mode* and *name* are the element of *ObjDict*.

In-Parameter — Expression An actual parameter corresponding to a formal parameter of mode **in** may be an expression other than a name. In this case it cannot give rise to aliasing.

1. The formal parameter exists and is of **in** mode.
2. The actual parameter is a well-formed expression which is not a name.
3. The type of the actual parameter *etyp* must be compatible with the type mark of the corresponding formal parameter.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NamedActual;\ as : \mathrm{seq}\, NamedActual;\ etyp : ExpType;\\ \quad ft : Id \pfun IdDot;\ fm : Id \rel RdWr;\ updates, refs : \mathbb{P}\, IdDot \mid\\ \qquad fm(\!|\{formal\}|\!) = rd \land actual \notin \mathrm{ran}\, nam \land\\ \qquad etyp\ compat_\delta\ (ft\ formal) \bullet\\ \quad \delta \vdash_{Exp} actual : etyp\\ \quad \delta, \{formal\} \ndres ft, \{formal\} \ndres fm \vdash_{Act} as \Longrightarrow refs, updates\end{array}}{\delta, ft, fm \vdash_{Act} \langle \theta\, NamedActual \rangle \frown as \Longrightarrow refs, updates} \quad \text{(Act3)}$$

Out-Parameter An actual parameter corresponding to a formal parameter of mode **out** or **inout** must be an entire variable (see Section 11.17.3). Aliasing could occur if such a variable is repeated in the parameter list.

1. The formal parameter exists and is of **out** or **inout** mode.
2. The actual parameter is a well-formed entire variable.
3. The entire variable can be accessed in the required modes.
4. The type of the entire variable is compatible (nam_compat_δ) with the type mark of the formal parameter.
5. The entire variable must not be referenced or updated elsewhere in the parameter list.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ NamedActual;\ as : \mathrm{seq}\, NamedActual;\ ObjDict;\ n : Name;\\ \quad ft : Id \pfun IdDot;\ fm : Id \rel RdWr;\ updates, refs : \mathbb{P}\, IdDot \mid\\ \qquad wr \in fm(\!|\{formal\}|\!) \land actual = nam\ n \land\\ \qquad fm(\!|\{formal\}|\!) \subseteq modes \land\\ \qquad type\ nam\_compat_\delta\ (ft\ formal) \land\\ \qquad name \notin refs \cup name \bullet\\ \quad \delta \vdash_{EntV} n : (\theta\, ObjDict)\\ \quad \delta, \{formal\} \ndres ft, \{formal\} \ndres fm \vdash_{Act} as \Longrightarrow refs, updates\end{array}}{\delta, ft, fm \vdash_{Act} \langle \theta\, NamedActual \rangle \frown as \Longrightarrow refs, updates \cup \{name\}} \quad \text{(Act4)}$$

Note that *name*, *type* and *modes* are the elements of *ObjDict*, which describe the entire variable.

References: *Exp* p. 71; *Env* p. 12; *ObjDict* p. 54; *Name* p. 53; *RdWr* p. 7; nam_compat_δ p. 182; $\vdash_{Name}$ p. 53; $compat_\delta$ p. 76; $\vdash_{Exp}$ p. 71; $\vdash_{EntV}$ p. 181

11.17.3 Well-formation of Entire Variables

An entire variable is a restricted form from the Abstract Syntax category of names. Informally, an entire variable is a "whole" object, rather than a subcomponent of an object of composite type.

The well-formation of an entire variable is specified by a relation between the environment, the name and the description of the variable (using *ObjDict*). The declaration of this relation is:

$$\_ \vdash_{EntV} \_ : \_ \subseteq Env \times Name \times ObjDict$$

Simple Variable A simple variable is well-formed as an entire variable.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ ObjDict;\ var : Id \mid \\ \qquad type = \delta.vartypes\ var \wedge \\ \qquad modes = \delta.varmodes(\!|\{var\}|\!) \wedge \\ \qquad name = id\ var \bullet\end{array}}{\delta \vdash_{EntV} simp\ var : \theta ObjDict} \quad \text{(EntVar1)}$$

Selected Variable A variable selected from another package is a well-formed entire variable.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ SlctName;\ ObjDict;\ pak, var : Id \mid \\ \qquad prefix = simp\ pak \wedge \\ \qquad pak \in \delta.withs \wedge \\ \qquad type = (\delta.paks\ pak).vis.vartypes\ var \wedge \\ \qquad modes = (\delta.paks\ pak).vis.varmodes(\!|\{var\}|\!) \wedge \\ \qquad name = dot(pak, var) \bullet\end{array}}{\delta \vdash_{EntV} slct(\theta SlctName) : \theta ObjDict} \quad \text{(EntVar2)}$$

References: *Env* p. 12; *Name* p. 53; *ObjDict* p. 54; *SlctName* p. 58

11.17.4 Type Compatibility of Parameters

An actual parameter is either a name or an expression. A parameter which is an expression (but not a name) must be type compatible ($compat_\delta$) with the type of the corresponding formal parameter.

This section describes the compatibility relation between formal parameter types and actual parameters which are names, allowing for the case of parameters of unconstrained array types. There are three cases:

1. An actual parameter of a type which is neither an unconstrained array type, nor a subtype of such a type, is compatible with a formal parameter which has the same ancestor type.

2. An actual parameter of an unconstrained array type is only compatible with a formal of the same unconstrained type.

3. An actual parameter of an unconstrained array *subtype* is compatible with either:

   (a) a formal of the unconstrained array type which is its ancestor, or

   (b) a formal of another subtype of the same unconstrained array, provided that the two subtypes have equal index ranges.

Note: these rules are required to ensure that implicit subtype conversion cannot occur in parameter substitutions in SPARK.

The relation nam_compat_δ holds between compatible actual and formal parameter type marks.

$$
\begin{array}{|l}
\_\ nam\_compat_{Env}\ \_ : IdDot \leftrightarrow IdDot \\
\hline
\forall\, \delta : Env;\ act, form : IdDot \mid \neg\ is\_unconstrained\_tmark_\delta\ act \land \\
\qquad \neg\ is\_unconstrained\_subtmark_\delta\ act \bullet \\
\quad act\ nam\_compat_\delta\ form \Leftrightarrow \\
\qquad ancestorof_\delta\ act = ancestorof_\delta\ form \\
\forall\, \delta : Env;\ act, form : IdDot \mid \\
\qquad is\_unconstrained\_tmark_\delta\ act \bullet \\
\quad act\ nam\_compat_\delta\ form \Leftrightarrow act = form \\
\forall\, \delta : Env;\ act, form : IdDot \mid \\
\qquad is\_unconstrained\_subtmark_\delta\ act \bullet \\
\quad act\ nam\_compat_\delta\ form \Leftrightarrow \\
\qquad (form = ancestorof_\delta\ act \lor \\
\qquad act\ compat\_uarr\_subtype_\delta\ form)
\end{array}
$$

Two unconstrained array subtypes are compatible only if they have the same ancestor and the same index range.

$$
\begin{array}{|l}
\_\ compat\_uarr\_subtype_{Env}\ \_ : IdDot \leftrightarrow IdDot \\
\hline
\forall\, \delta : Env;\ s, t : IdDot \bullet \\
\quad s\ compat\_uarr\_subtype_\delta\ t \Leftrightarrow \\
\qquad is\_unconstrained\_subtmark_\delta\ s \land \\
\qquad is\_unconstrained\_subtmark_\delta\ t \land \\
\qquad ancestorof_\delta\ s = ancestorof_\delta\ t \land \\
\qquad scalar\_tmark\_range_\delta \circ (array\_index\_tmark_\delta\ s) = \\
\qquad\quad scalar\_tmark\_range_\delta \circ (array\_index\_tmark_\delta\ t)
\end{array}
$$

References: *Env* p. 12; *is_unconstrained_tmark*$_\delta$ p. 283; *is_unconstrained_subtmark*$_\delta$ p. 284; *ancestorof*$_\delta$ p. 281; *scalar_tmark_range*$_\delta$ p. 296; *array_index_tmark*$_\delta$ p. 289

Chapter 12

Subprogram Declarations

This chapter describes the subprogram declarations of SPARK, which form the Abstract Syntax category *SDecl*.

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. In the visible part of a package specification, a subprogram is *declared* by giving its name, formal parameters and annotations: these declarations are the subject of this chapter.

A subprogram declared in this way is *defined*, by giving its implementation, in the package body; these definition are described in Chapter 14. All the components of the subprogram may be given together; this form is termed a *local* subprogram and is described in Chapter 15.

The Abstract Syntax of subprogram declarations is summarised in the following table:

The syntax of these declarations is complex and has some common sub-structures. The well-formation of these sub-structures is specified separately in some initial sections, as follows:

The Well-Formation of Subprogram Declarations

The well-formation of a subprogram declaration is defined by a relation between the environment, the declaration and a modified environment. The declaration of this relation is:

$$\_ \vdash_{SDecl} \_ \Longrightarrow \_ \subseteq Env \times SDecl \times Env$$

Thus the well-formation predicate:

$$\delta \vdash_{SDecl} decl \Longrightarrow \delta'$$

can be read as "subprogram declaration *decl* is well-formed in the initial environment δ, with the modified environment δ'".

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);	$\langle \langle\!\langle\ param \mapsto x,\ mode \mapsto in,\ ptype \mapsto id\ T\ \rangle\!\rangle, \langle\!\langle\ param \mapsto y,\ mode \mapsto out,\ ptype \mapsto id\ S\ \rangle\!\rangle\rangle$

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 Static Semantics

The well-formation of a sequence of formal parameters is described by a relation:

$$\_ \vdash_{Form} \_ \Longrightarrow \_,\_,\_ \subseteq Env \times \mathrm{seq}\, FormalParam \times Id \pfun IdDot \times Id \rel RdWr \times \mathrm{iseq}\, Id$$

Thus, the following predicate

$$\delta \vdash_{Form} fs \Longrightarrow ftypes, fmodes, fpos$$

may be read as "the list of formal parameters *fs* is well-formed in the initial environment δ, where

ftypes	maps each parameter to its type identifier,
fmodes	relates each parameter to the access modes allowed for it,
fpos	gives the name of the parameters in each position."

The values *ftypes*, *fmodes* and *fpos* have the same structure as the elements of the schemas *Proc* and *Fun*, used to record a subprogram interface in the environment.

Empty Parameter List The empty list of parameters is well-formed.

$$\frac{\forall \delta : Env \bullet}{\delta \vdash_{Form} \langle\rangle \Longrightarrow \varnothing, \varnothing, \langle\rangle} \quad \text{(Form1)}$$

Non-empty Parameter List The well-formation rules for a non-empty list of formal parameters are:

1. The first formal parameter identifier is not already declared.
2. The first formal parameter identifier is not repeated in the remainder of the list.
3. The type of the first formal parameter is visible.
4. A limited type may only be used in the declaration of a parameter of mode **out** if the type is directly visible and declared using the **limited private** type constructor [AARM, §7.4.4 ¶4].

The access modes of *param* are determined from its parameter mode, according to the function *mode_to_rdwr* (see below), it is given the type *ptype* and position number one.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ FormalParam;\ fps : \mathrm{seq}\, FormalParam;\ ft : Id \pfun IdDot;\\ \quad fm : Id \rel RdWr;\ fp : \mathrm{iseq}\, Id \mid\\ \qquad param \notin \delta.usedids \land\\ \qquad param \notin \mathrm{dom}\, ft \land\\ \qquad is\_visible\_tmark_{\delta}\ ptype \land\\ \qquad (mode = out \land is\_limited\_tmark_{\delta}\ ptype \Rightarrow\\ \qquad\quad (\exists\, t : Id \bullet ptype = id\ t \land \delta.types\ t = limT)) \bullet\\ \quad \delta \vdash_{Form} fps \Longrightarrow ft, fm, fp\end{array}}{\delta \vdash_{Form} \langle \theta FormalParam \rangle \frown fps \Longrightarrow ft', fm', \langle param \rangle \frown fp} \quad \text{(Form2)}$$

where

$$ft' == ft \cup \{param \mapsto ptype\}$$
$$fm' == fm \cup (\{param\} \times (mode\_to\_rdwr\ mode))$$

References: *Mode* p. 188; *Env* p. 12; *RdWr* p. 7; *Proc* p. 8; *Fun* p. 9; *mode_to_rdwr* p. 189; $is\_visible\_tmark_{\delta}$ p. 281

12.1.3 Parameter Modes

The mode of a formal parameter determines whether the subprogram can use the initial value of the parameter and/or derives a new value.

$$Mode ::= in \mid out \mid inout \mid default$$

The modes of a parameter are related to the allowed access modes of the corresponding variable — *RdWr*:

1. A formal parameter with **in** mode has read access access only.

2. A formal parameter with **out** mode has write access only [AARM, §6.2, ¶5]
3. A formal parameter with **inout** mode can have both read and write access.
4. A formal parameter with the default mode has read access only.

Note that the **derives** annotation determines whether the write access mode of an **inout** parameter is actually used.

$$
\begin{array}{|l}
mode\_to\_rdwr : Mode \rightarrow \mathbb{P}\ RdWr \\
\hline
mode\_to\_rdwr\ in = \{rd\} \\
mode\_to\_rdwr\ out = \{wr\} \\
mode\_to\_rdwr\ inout = \{rd, wr\} \\
mode\_to\_rdwr\ default = \{rd\}
\end{array}
$$

References: RdWr p. 7

12.2 Global Annotations

This annotation specifies the variables in the current scope (i.e. those that are global to the subprogram) which are used within the subprogram.

Syntax Example	A.S. Representation
- -# **global** k.y, z;	$globals \mapsto \langle dot(k, y), id\ z \rangle$

12.2.1 Abstract Syntax

A global annotation is a list of variable names:

$$GlobalAnnot == \text{seq}\ IdDot$$

12.2.2 Static Semantics

The well-formation of a global annotation is specified as:

$$\_ \vdash_{Glob} \_ \Longrightarrow \_ \subseteq Env \times GlobalAnnot \times \mathbb{P}\ IdDot$$

Thus, the well-formation predicate:

$$\delta \vdash_{Glob} gs \Longrightarrow glvars$$

can be read as: "the global annotation gs is well-formed in the environment δ with the set of globals $glvars$".

Empty List of Globals An empty global list is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{Glob} \langle\rangle \Longrightarrow \varnothing} \qquad \text{(Glob1)}$$

Variable from the local Package The global list may contain a directly visible variable.

1. The variable must have been declared, or be an own variable,
2. The variable must not appear elsewhere in the global list.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ gs : \mathrm{seq}\ GlobalAnnot;\ glvars : \mathbb{P}\ IdDot;\ i : Id \mid \\ \qquad i \in \mathrm{dom}\, \delta.varmodes \land \\ \qquad id\ i \notin glvars \bullet \\ \delta \vdash_{Glob} gs \Longrightarrow glvars\end{array}}{\delta \vdash_{Glob} \langle id\ i \rangle \frown gs \Longrightarrow glvars \cup \{id\ i\}} \quad \text{(Glob2)}$$

Variable from Inherited Package The global list may contain a variable inherited from another package.

1. The package must be in the list of inherited packages.
2. The variable must be a visible variable of the package, or an own variable.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ gs : \mathrm{seq}\ GlobalAnnot;\ glvars : \mathbb{P}\ IdDot;\ k, i : Id \mid \\ \qquad k \in \delta.inherits \land \\ \qquad i \in \mathrm{dom}(\delta.paks\ k).vis.varmodes \land \\ \qquad dot\ (k, i) \notin glvars \bullet \\ \delta \vdash_{Glob} gs \Longrightarrow glvars\end{array}}{\delta \vdash_{Glob} \langle dot(k, i) \rangle \frown gs \Longrightarrow glvars \cup \{dot\ (k, i)\}} \quad \text{(Glob3)}$$

References: Env p. 12

12.3 Import Lists

This section describes the well-formation rules for the lists of imported variables which form part of the **derives** annotation (see page 194).

Syntax Example	A.S. Representation
from x, k.y, z ;	$\langle id\ x, dot(k, y), id\ z \rangle$

12.3.1 Abstract Syntax

An export is derived **from** a list of variables.

$$FromList == \text{seq}\ IdDot$$

12.3.2 Static Semantics

The well-formation of a list of imports is specified as:

$$\_ \vdash_{Imp} \_ \Longrightarrow \_ \subseteq Env \times FromList \times \mathbb{P}\ IdDot$$

Thus, the well-formation statement:

$$\delta \vdash_{Imp} ms \Longrightarrow imports$$

can be read as: "the import list ms is well-formed in the environment δ, resulting in a set of imported variables $imports$".

Empty List of Imports An empty list is always well-formed; no imports result.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{Imp} \langle\rangle \Longrightarrow \varnothing} \qquad \text{(Imp1)}$$

Variable from Local Package The import list may contain a directly visible variable.

1. The variable must have been declared, or be an own variable.
2. The variable must not appear elsewhere in the import list.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ ms : FromList;\ i : Id;\ imports : \mathbb{P}\ IdDot \mid \\ \qquad i \in \text{dom}\, \delta.varmodes \wedge \\ \qquad (id\ i) \notin imports \bullet \\ \delta \vdash_{Imp} ms \Longrightarrow imports\end{array}}{\delta \vdash_{Imp} \langle id\ i \rangle \frown ms \Longrightarrow imports \cup \{id\ i\}} \qquad \text{(Imp2)}$$

Variable from Inherited Package The import list may contain a variable inherited from another package.

1. The package must be in the list of inherited packages.
2. The variable must be a visible variable of the package, or an own variable.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ ms : FromList;\ imports : \mathbb{P}\, IdDot;\ k, i : Id \mid \\ \qquad k \in \delta.inherits \wedge \\ \qquad i \in \mathrm{dom}(\delta.paks\ k).vis.varmodes \wedge \\ \qquad dot(k,i) \notin imports \bullet \\ \delta \vdash_{Imp} \Longrightarrow imports\end{array}}{\delta \vdash_{Imp} \langle dot(k,i)\rangle \frown ms \Longrightarrow imports \cup \{dot(k,i)\}} \quad \text{(Imp3)}$$

Read Access of Imports The following program fragment illustrates why there is no requirement for imports to have read access.

```
procedure p(x : out T)
--# global ...;
--# derives ...;
is
    procedure q
    --# global x, y;
    --# derives x from x, y ;
    is begin
        if y = 0 then x := 1 end if ;
    end q;
```

The procedure q *conditionally* derives a new value of x. Thus x is derived from itself even though x cannot be read.

Note that x is a *global* variable of the procedure q. This is significant; since parameters of mode **out** are initially undefined, they cannot be used as imports of the procedure on which they are declared. In the example, this implies that x cannot be an import of p. This rule cannot be stated as part of the well-formation rule for the import list, since it distinguishes global variables from formal parameters; it appears in Sections 12.6, 15.1.
References: Env p. 12

12.4 Derives Annotations

This annotation describes how the global variables and formal parameters of a subprogram are used, as follows:

Exports are given a new value by the subprogram.

Imports have an initial value which is used by the subprogram.

The annotation also specifies the dependency relation between imports and exports. This information is used for program flow analysis; it does not affect the well-formation rules.

Syntax Example	A.S. Representation
- -# **derives** y **from** k.x, z & k.x **from** y;	$\langle \langle\!\langle\ export \mapsto id\ y,\ imports \mapsto \langle dot(k,x), id\ z\ \rangle\ \rangle\!\rangle$ $\langle\!\langle\ export \mapsto dot(k,x),\ imports \mapsto \langle id\ y\ \rangle\ \rangle\!\rangle \rangle$

12.4.1 Abstract Syntax

A derives annotation relates an exported variable to the list of variables whose initial values determine the exported value.

$$DerivesAnnot \mathrel{\widehat=} [derive : IdDot;\ froms : FromList]$$

12.4.2 Static Semantics

The well-formation of a derives annotation is specified by the relation:

$$\_ \vdash_{Deri} \_ \Longrightarrow \_, \_ \subseteq Env \times \mathrm{seq}\ DerivesAnnot \times \mathbb{P}\ IdDot \times \mathbb{P}\ IdDot$$

Thus the predicate:

$$\delta \vdash_{Deri} ds \Longrightarrow imports, exports$$

can be read as "the derives list ds is well-formed in the environment δ, resulting in the sets of export and import variables *exports* and *imports* respectively". The environment δ is the environment in which a subprogram is declared, with variables not appearing in the global annotation removed and the formal parameters added.

Empty Derives List An empty derives list is well-formed, resulting in a subprogram with no imports or exports.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{Deri} \langle\rangle \Longrightarrow \varnothing, \varnothing} \qquad \text{(Deri1)}$$

Directly Visible Export A derives annotation may specify a variable of the current package as an export.

1. The variable must be a declared variable or an own variable with write access.
2. The variable is not specified as an export in the remainder of the derives annotation.
3. The list of imports for this export must be well formed.
4. The exported variable is added to the set of variables specified as exports by the remainder of the derives list.
5. The set of imports for this export are added to the set of variables specified as imports by the remainder of the derives list.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ DerivesAnnot;\ ds : \mathrm{seq}\, DerivesAnnot;\ x : Id; \\ \quad imports, imports', exports : \mathbb{P}\, IdDot \mid \\ \qquad derive = id\ x \land \\ \qquad (x, wr) \in \delta.varmodes \land \\ \qquad derive \notin exports \bullet \\ \delta \vdash_{Deri} ds \Longrightarrow imports, exports \\ \delta \vdash_{Imp} froms \Longrightarrow imports'\end{array}}{\delta \vdash_{Deri} \langle \theta DerivesAnnot \rangle \frown ds \Longrightarrow imports'', exports'} \quad \text{(Deri2)}$$

where

$$exports' == exports \cup \{derive\}$$
$$imports'' == imports \cup imports'$$

Indirectly Visible Export A derives annotation may specify a variable of another package as an export.

1. The package must be inherited.
2. The variable must be a declared variable or an own variable with write access.
3. The variable must not be specified as an export in the remainder of the derives annotation.
4. The list of imports for this export must be well formed.
5. The exported variable is added to the set of variable specified as exports by the remainder of the derives list.

6. The set of imports for this export are added to the set of variable specified as imports by the remainder of the derives list.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ DerivesAnnot;\ ds : \mathrm{seq}\, DerivesAnnot;\ k, x : Id; \\ \quad imports, imports', exports : \mathbb{P}\, IdDot \mid \\ \qquad derive = dot(k, x) \land \\ \qquad k \in \delta.inherits \land \\ \qquad (x, wr) \in (\delta.paks\ k).vis.varmodes \land \\ \qquad derive \notin exports \bullet \\ \quad \delta \vdash_{Deri} ds \Longrightarrow imports, exports \\ \quad \delta \vdash_{Imp} froms \Longrightarrow imports'\end{array}}{\delta \vdash_{Deri} \langle \theta DerivesAnnot \rangle \frown ds \Longrightarrow imports'', exports'} \quad \text{(Deri3)}$$

where

$$exports' == exports \cup \{derive\}$$
$$imports'' == imports \cup imports'$$

References: *Env* p. 12; $\vdash_{Imp}$ p. 192

12.5 Subprogram Scopes

The visibility within a subprogram is determined from the environment in which the subprogram (body) is declared using the **global** and **derives** annotations, as follows:

1. A variable can be used only if it is listed in the **global** annotation.
2. A variable can be updated only if its is specified as an export in the **derives** annotation.
3. A subprogram can be called only if all its global variables are visible in the new environment and those exported can be written to.

The identifier of a variable not listed in the **global** annotation can be re-used within the subprogram (possibily as a formal parameter). However, the identifier of a subprogram from the original environment which is not callable in the new environment cannot be re-used.

We describe the deletion of identifiers not satisfying certain properties as *restricting* the environment. The following sections specify the different restrictions described above.

In the following definitions, the functions *local* and *global k* are used to extract the identifiers of variables in the local package and package k respectively.

$$
\begin{array}{l}
local : \mathbb{P}\ IdDot \to \mathbb{P}\ Id \\
global : Id \to \mathbb{P}\ IdDot \to \mathbb{P}\ Id \\
\hline
\forall\ vars : \mathbb{P}\ IdDot;\ k : Id \bullet \\
\quad local\ vars = \{\ i : Id \mid id\ i \in vars\ \} \land \\
\quad global\ k\ vars = \{\ i : Id \mid dot\ (k, i) \in vars\ \}
\end{array}
$$

12.5.1 Restricting Variables

The environment $\delta \downarrow_G vars$ only includes the variables from δ (directly visible and in all inherited packages) if they are included in the set of variables *vars*.

$$
\begin{array}{l}
\_ \downarrow_G \_ : Env \times (\mathbb{P}\ IdDot) \to Env \\
\hline
\forall\ \delta : Env;\ vars : \mathbb{P}\ IdDot \bullet \delta \downarrow_G vars = \\
\quad \delta[\ usedids := \delta.usedids \setminus (\mathrm{dom}\ \delta.varmodes \setminus (local\ vars)), \\
\quad\ varmodes := (local\ vars) \lhd \delta.varmodes, \\
\quad\ vartypes := (local\ vars) \lhd \delta.vartypes, \\
\quad\ paks := \delta.paks \oplus (\lambda\ k : \delta.inherits \bullet (\delta.paks\ k)[\\
\quad\quad\ vis := (\delta.paks\ k).vis[\\
\quad\quad\quad\ varmodes := (global\ k\ vars) \lhd (\delta.paks\ k).vis.varmodes, \\
\quad\quad\quad\ vartypes := (global\ k\ vars) \lhd (\delta.paks\ k).vis.vartypes\ ]\ ])\ ]
\end{array}
$$

12.5.2 Restricting Write Access

In the environment $\delta \downarrow_X vars$ a variable from δ has write access only if is in the set of exported variables *vars*. Note that the definition given assumes that all the exported variables have write access in the environment in which the subprogram is declared: this is indeed the case in the context of its use.

$$
\begin{array}{|l}
\_ \downarrow_X \_ : Env \times (\mathbb{P}\ IdDot) \to Env \\
\hline
\forall\, \delta : Env;\ vars : \mathbb{P}\ IdDot \bullet \\
\quad \delta \downarrow_X vars = \delta\ [\\
\qquad varmodes := (\delta.varmodes \rhd \{rd\}) \cup ((local\ vars) \times \{wr\}), \\
\qquad paks := (\lambda\, k : \delta.inherits \bullet (\delta.paks\ k)\ [\\
\qquad\quad vis := (\delta.paks\ k).vis\ [\\
\qquad\qquad varmodes := ((\delta.paks\ k).vis.varmodes \rhd \{rd\}) \cup \\
\qquad\qquad\quad ((global\ k\ vars) \times \{wr\})\ ]\ ])\ ]
\end{array}
$$

12.5.3 Restricting Callable Subprograms

The environment $\delta \downarrow_S$ contains only the subprograms of δ satisfying the restriction:

1. All the global variables of the subprogram are visible.
2. All the exported variables of the (procedure) subprogram are writeable.

Removing a global variable from the scope with a subprogram (i.e. not including it in the global annotation) makes subprograms which use it uncallable.

$$
\begin{array}{|l}
\_ \downarrow_S : Env \to Env \\
\hline
\forall\, \delta : Env \bullet \\
\quad \delta \downarrow_S = \delta\ [\\
\qquad scall := \{\ s : \delta.scall \mid \\
\qquad\quad (s \in \mathrm{dom}\, \delta.fdecls \Rightarrow \\
\qquad\qquad (\delta.fdecls\ s).glvars \subseteq vars\_of\ \delta) \wedge \\
\qquad\quad (s \in \mathrm{dom}\, \delta.pdecls \Rightarrow \\
\qquad\qquad (\delta.pdecls\ s).glvars \subseteq vars\_of\ \delta \wedge \\
\qquad\qquad (\delta.pdecls\ s).exvars \subseteq wr\_vars\_of\ \delta)\}, \\
\qquad paks := (\lambda\, k : \delta.inherits \bullet (\delta.paks\ k)[\\
\qquad\quad vis := (\delta.paks\ k).vis[\ scall := \{\ s : (\delta.paks\ k).vis.scall \mid \\
\qquad\qquad (s \in \mathrm{dom}(\delta.paks\ k).vis.fdecl \Rightarrow \\
\qquad\qquad\quad fun\_glvars_\delta\ dot(k, s) \subseteq vars\_of\ \delta) \wedge \\
\qquad\qquad (s \in \mathrm{dom}(\delta.paks\ k).vis.pdecl \Rightarrow \\
\qquad\qquad\quad proc\_glvars_\delta\ dot(k, s) \subseteq vars\_of\ \delta \wedge \\
\qquad\qquad\quad proc\_exvars_\delta\ dot(k, s) \subseteq wr\_vars\_of\ \delta)\ \}\ ]\ ])\ ]
\end{array}
$$

where the function *vars_of* returns the set of all variables visible in annotations,

$$
\begin{array}{|l}
vars\_of : Env \to \mathbb{P}\ IdDot \\
\hline
\forall \delta : Env;\ k, i : Id \bullet \\
\quad id\ i \in vars\_of\ \delta \Leftrightarrow \\
\qquad i \in \mathrm{dom}\, \delta.varmodes \land \\
\quad dot\ (k, i) \in vars\_of\ \delta \Leftrightarrow \\
\qquad k \in \delta.inherits \land \\
\qquad i \in \mathrm{dom}(\delta.paks\ k).vis.varmodes
\end{array}
$$

and the function *wr_vars_of* returns the set of all writeable variables visible in annotations.

$$
\begin{array}{|l}
wr\_vars\_of : Env \to \mathbb{P}\ IdDot \\
\hline
\forall \delta : Env;\ k, i : Id \bullet \\
\quad id\ i \in wr\_vars\_of\ \delta \Leftrightarrow \\
\qquad (i, wr) \in \delta.varmodes \land \\
\quad dot\ (k, i) \in wr\_vars\_of\ \delta \Leftrightarrow \\
\qquad k \in \delta.inherits \land \\
\qquad (i, wr) \in (\delta.paks\ k).vis.varmodes
\end{array}
$$

References: *Env* p. 12; fun_glvars_δ p. 292; $proc\_glvars_\delta$ p. 294; $proc\_exvars_\delta$ p. 294

12.6 Procedure Declaration

A procedure can be declared in the visible part of a package specification.

Syntax Example	A.S. Representation
procedure p(x : **in** T; y : **out** S); --# **global** z; --# **derives** y **from** x,z ;	$pdecl \langle\!\langle\ pid \mapsto p,$ $formals \mapsto \langle \ldots, \ldots \rangle,$ $globals \mapsto \langle id\ z \rangle,$ $derives \mapsto \langle\langle\!\langle\ export \mapsto id\ y,$ $imports \mapsto \langle id\ x, id\ z\ \rangle\ \rangle\!\rangle\rangle\!\rangle$

12.6.1 Abstract Syntax

The declaration gives the name of the procedure, its formal parameters and its global and derives annotations.

PDeclSDecl

pid : *Id*
$formals$: seq *FormalParam*
$globals$: *GlobalAnnot*
$derives$: seq *DerivesAnnot*

$SDecl ::= \ldots \mid pdecl\langle\langle PDeclSDecl \rangle\rangle$

12.6.2 Static Semantics

1. The name of the procedure must not be already declared.
2. The global annotation must be well-formed in the initial environment, with the name of the procedure used.
3. The formal parameters must be well-formed in the environment δ_g, whose variables are restricted to those from the global annotation.
4. The derives annotation must be well-formed in the environment δ_f, to which the formal parameters have been added as variables.
5. The derives annotation must be complete and consistent with the modes of the formal parameters:
   (a) All the global variables and formal parameters must appear as either exported variables ($exvars$) or imported variables ($imvars$) in the derives annotation.

(b) All formal parameters of mode **inout** must be imported. *This rule is required since such parameters are assumed to be defined when the sub-program is called* [AARM, §6.4.1 ¶5-9].

(c) No formal parameter of mode **out** may be used as import.

6. In the modified environment δ_p, the identifier *pid* is used, and the procedure *pid* declared with:

(a) *ftypes*, *fmodes* and *fpos* which describe the formal parameters,

(b) the set of global variables *glvars* given by the **global** annotation,

(c) the set of exported variables *exvars* given by the **derives** annotation.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ PDeclSDecl;\ Proc;\ imvars : \mathbb{P}\ IdDot \mid \\ \quad pid \notin \delta.usedids \land \\ \quad glvars \cup id(|\mathrm{dom}\, ftypes|) = exvars \cup imvars \land \\ \quad (\forall f : \mathrm{dom}\, fmodes \mid \{\ (f, rd)(f, wr)\ \} \subseteq fmodes \bullet \\ \qquad id\ f \in imvars) \land \\ \quad \neg\ (\exists f : \mathrm{dom}\, fmodes \mid (f, rd) \notin fmodes \land id\ f \in imvars) \bullet \\ \delta_u \vdash_{Glob} globals \Longrightarrow glvars \\ \delta_g \vdash_{Form} formals \Longrightarrow ftypes, fmodes, fpos \\ \delta_f \vdash_{Deri} derives \Longrightarrow from\_vars, deri\_vars\end{array}}{\delta \vdash_{SDecl} pdecl(\theta PDeclSDecl) \Longrightarrow \delta_p} \quad \text{(PDecl)}$$

where

$$\begin{array}{l}\delta_u == \delta\ [\ usedids := \delta.usedids \cup \{pid\}] \\ \delta_g == \delta_u \downarrow_G glvars \\ \delta_f == \delta_g\ [\ varmodes := \delta_g.varmodes \cup fmodes, \\ \qquad\qquad vartypes := \delta_g.vartypes \cup ftypes\ ] \\ \delta_p == \delta\ [\ usedids := \delta.usedids \cup \{pid\}, \\ \qquad\qquad pdecls := \delta.pdecls \cup \{pid \mapsto \theta Proc\}, \\ \qquad\qquad sdef := \delta.sdef \cup \{pid\}\ ]\end{array}$$

References: *FormalParam* p. 187; *GlobalAnnot* p. 190; *DerivesAnnot* p. 194; *Env* p. 12; *Proc* p. 8; $\vdash_{Glob}$ p. 190; $\vdash_{Form}$ p. 187; $\vdash_{Deri}$ p. 194; $\downarrow_G$ p. 197

12.7 Function Declaration

A function can be declared in the visible part of a package specification.

Syntax Example	A.S. Representation
function f(x : **in** T) **return** S;	*fdecl* ⟪ *fid* ↦ *f*,
--# **global** z;	*formals* ↦ ⟨⟪ *param* ↦ *x*,
	mode ↦ *in*,
	ptype ↦ *id T* ⟫ ⟩,
	rtype ↦ *id S*,
	globals ↦ ⟨*id z*⟩ ⟫

12.7.1 Abstract Syntax

The declaration gives the name of the function, its formal parameters, its global annotation and the return type.

FDeclSDecl

fid : *Id*
formals : seq *FormalParam*
globals : *GlobalAnnot*
rtype : *IdDot*

SDecl ::= ... | *fdecl*⟪⟪*FDeclSDecl*⟫⟫

12.7.2 Static Semantics

1. The name of the function *fid* must not already be declared.
2. The global annotation must be well-formed in the initial environment, with the name of the procedure used.
3. The formal parameters must be well-formed in the environment δ_g, whose variables are restricted to those from the global annotation.
4. The formal parameters must all be of mode **in** or the default mode.
5. The return type of the function must be visible and not unconstrained.
6. In the modified environment δ_f, the identifier *fid* is used and the function is declared with:
   (a) *ftypes*, *fpos* which describe the formal parameters,

(b) the return type *rtype*,

(c) the set of global variables *glvars* given by the *global* annotation.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ FDeclSDecl;\ Fun;\ fmodes : Id \leftrightarrow RdWr \mid \\ \qquad fid \notin \delta.usedids \land \\ \qquad is\_visible\_tmark_{\delta}\ rtype \land \\ \qquad \neg\ is\_unconstrained\_tmark_{\delta}\ rtype \land \\ \qquad \mathrm{ran}\, fmodes = \{rd\} \bullet \\ \delta_u \vdash_{Glob} globals \Longrightarrow glvars \\ \delta_g \vdash_{Form} formals \Longrightarrow ftypes, fmodes, fpos\end{array}}{\delta \vdash_{SDecl} fdecl\ (\theta FDeclSDecl) \Longrightarrow \delta_f} \quad \text{(FDecl)}$$

where

$$\delta_u == \delta[\ usedids := \delta.usedids \cup \{fid\}\ ]$$

$$\delta_g == \delta_u \downarrow_G glvars$$

$$\begin{array}{l}\delta_f == \delta[\ \ usedids := \delta.usedids \cup \{fid\}, \\ \qquad\qquad fdecls := \delta.fdecls \cup \{fid \mapsto \theta Fun\}, \\ \qquad\qquad sdef := \delta.sdef \cup \{fid\}\ ]\end{array}$$

References: *FormalParam* p. 187; *GlobalAnnot* p. 190; *Env* p. 12; *Fun* p. 9; *RdWr* p. 7; $is\_visible\_tmark_{\delta}$ p. 281; $\vdash_{Glob}$ p. 190; $\vdash_{Form}$ p. 187; $\downarrow_G$ p. 197

Chapter 13

Embedded Package Declarations

This chapter describes the declaration of embedded package 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	206

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;
package k
  --# **own** x,y;
  --# **initializes** x;
is
  declarations ...
end k;
renames ...
	$kspec \langle\!\langle\ inherit \mapsto \langle l, m \rangle,$
	$kid \mapsto k,$
	$own \mapsto \langle x, y \rangle,$
	$init \mapsto \langle x \rangle,$
	$vdecl \mapsto \ldots,$
	$pdecl \mapsto dnull,$
	$rens \mapsto \ldots\ \rangle\!\rangle$

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

$kid : Id$
$inherit : \text{seq}\, Id$
$own : \text{seq}\, Id$
$init : \text{seq}\, Id$
$vdecl : VBasic$
$pdecl : PBasic$
$rens : \text{seq}\, Ren$

$KDecl ::= kspec\langle\!\langle KSpecKDecl \rangle\!\rangle$

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:

Syntax Constructor	Description	Page
pdefn	Procedure definition in a package body	208
pstub	Separate declaration for an exported procedure	211
fdefn	Function definition in a package body	212
fstub	Separate declaration for an exported function	214

The Well-Formation of Subprogram Definitions

The well-formation of a subprogram definition is defined by a relation between the environment, the definition and a modified environment. The declaration of this relation is:

$$\_ \vdash_{FDecl} \_ \Longrightarrow \_ \subseteq Env \times FDecl \times Env$$

Thus the well-formation predicate:

$$\delta \vdash_{FDecl} f \Longrightarrow \delta'$$

can be read as "definition f is well-formed in the initial environment δ, with the modified environment δ'".

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)
is
  declarations ...
begin
  statements ...
end p;
	pdefn ⟪ *pid* ↦ *p*,
	formals ↦ ⟨...⟩,
	dpart ↦ ...,
	spart ↦ ... ⟫

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.

PDefnFDecl

pid : *Id*
formals : seq *FormalParam*
bdecl : *SBasic*
ldecl : *SLater*
stmt : *Stmt*

FDecl ::= *pdefn*⟨⟨*PDefnFDecl*⟩⟩ | ...

14.1.2 Static Semantics

1. The definition of the procedure must be allowed in this scope.

2. The formal parameter declarations must be well-formed in the environment δ_p created by restricting the initial environment using the global annotation of the procedure. The formal parameters must also *conform* to the one given in the specification (see below).

3. The local declarations of the procedure must be well-formed in the environment formed from δ_p by:

   (a) adding the formal parameters as variables, and

(b) preventing the declaration of the body of any packages whose specification is declared in the same scope as this procedure definition,

(c) restricting the writeable variables using the exports of the procedure,

(d) restricting the callable subprograms.

4. The statements of the procedure body must be well-formed in the environment enriched with the local declarations of the procedure.

5. The procedure definition modifies the environment so that the procedure *pid* has been defined and can be called.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_b, \delta_l : Env;\ PDefnFDecl \mid pid \in \delta.sdef \bullet \\ \quad \delta_p \vdash_{Form} formals \Longrightarrow (\delta.pdecls\ pid).ftypes, \\ \qquad\qquad (\delta.pdecls\ pid).fmodes, (\delta.pdecls\ pid).fpos \\ \quad \delta_m \downarrow_X (\delta.pdecls\ pid).exvars \downarrow_S \vdash_{SBasic} bdecl \Longrightarrow \delta_b \\ \quad \delta_b \vdash_{SLater} ldecl \Longrightarrow \delta_l \\ \quad \delta_l \vdash_{Stmt} stmt\end{array}}{\delta \vdash_{FDecl} pdefn(\theta PDefnDecl) \Longrightarrow \delta_f} \quad \text{(PDefn)}$$

where

$$\delta_p == \delta \downarrow_G (\delta.pdecls\ pid).glvars$$

$$\begin{array}{l}\delta_m == \delta_p[\ \ usedids := \delta_p.usedids \cup \mathrm{dom}(\delta.pdecls\ pid).ftypes, \\ \qquad\quad varmodes := \delta_p.varmodes \cup (\delta.pdecls\ pid).fmodes, \\ \qquad\quad vartypes := \delta_p.vartypes \cup (\delta.pdecls\ pid).ftypes, \\ \qquad\quad bodies\_req := \varnothing, \\ \qquad\quad scope := \delta_p.scope \frown \langle pid \rangle\ ]\end{array}$$

$$\begin{array}{l}\delta_f == \delta[\ \ sdef := \delta.sdef \setminus \{pid\}, \\ \qquad\quad scall := \delta.scall \cup \{pid\}\ ]\end{array}$$

References: FormalParam p. 187; *SBasic* p. 131; *SLater* p. 135; *Stmt* p. 153; *Env* p. 12; $\vdash_{Form}$ p. 187; $\vdash_{SBasic}$ p. 131; $\vdash_{SLater}$ p. 135; $\vdash_{Stmt}$ p. 154; $\downarrow_G$ p. 197; $\downarrow_X$ p. 198; $\downarrow_S$ p. 199

14.1.3 Conformance of Repeated Formal Parameters

A subprogram declaration (in a package specification) and its subsequent definition (in the body of the same package) both include the formal parameter specification.

The rules of Ada for the conformance of these repeated parameter specifications (see [AARM, §6.3.1]) imply that the specifications must consist of the same sequence of lexical tokens. (None of the allowed variations apply in SPARK.) This rule cannot be fully expressed in this definition, because there may be many different sequences of lexical tokens

which generate the term in the Abstract Syntax which represents the formal parameter specification.

The specification above requires the repeated formal parameter specification to be well-formed (in the environment with variable restricted to those from the global annotation), giving the same types, modes and positions as the formal parameters used in the package specification.

Using the example of [AARM, §6.3.1, ¶6], the three specifications:

procedure P(X,Y : INTEGER)
procedure P(X : INTEGER; Y : INTEGER)
procedure P(X,Y : **in** INTEGER)

are *incorrectly* considered equivalent.

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** ;	$pstub\ \langle\!\langle\ pid \mapsto p,\ params \mapsto \langle\langle\ param \mapsto x,\ mode \mapsto in,\ ptype \mapsto id\ \ T\ \ \rangle\!\rangle\ \rangle\!\rangle$

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.

PStubFDecl

$pid : Id$

$params : \text{seq}\ FormalParam$

$FDecl ::= \ldots \mid pstub\langle\langle PStubFDecl \rangle\rangle$

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
is
  declarations ...
begin
  statements ...
  return x + 1;
end f;
	fdefn ⟪ *fid* ↦ *f*,
	formals ↦ ⟨...⟩,
	rtype ↦ *id S*,
	dpart ↦ ...,
	spart ↦ ...,
	return ↦ ... ⟫

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

fid : *Id*
formals : seq *FormalParam*
rtype : *IdDot*
bdecl : *SBasic*
ldecl : *SLater*
stmt : *Stmt*
return : *Exp*

$FDecl ::= \ldots \mid fdefn\langle\langle FDefnFDecl\rangle\rangle$

14.3.2 Static Semantics

1. The definition of the function must be allowed in this scope.

2. The formal parameters must be well-formed in the environment δ_n created by restricting the initial environment using the global variables of the function. The formal parameters must also conform (see the discussion on this point in Section 14.1) and the return type be the same as that used in the corresponding declaration.

3. The local declarations of the function must be well-formed in the environment formed from δ_n by:
   (a) adding the formal parameters to this environment as variables,
   (b) preventing the declaration of the bodies of packages whose specifications are declared in the same scope as the procedure definition,
   (c) restricting the writeable variables in the resulting environment using the function's implicit empty list of exports,
   (d) restricting the callable subprograms.
4. The statements of the function body must be well-formed in the environment enriched with the local declarations of the function.
5. The function definition modifies the environment so that the function *fid* has been defined and can be called.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_b, \delta_l : Env;\ FDefnFDecl;\ fmodes : Id \leftrightarrow RdWr;\ etyp : ExpType \mid \\ \quad fid \in \delta.sdef \land \\ \quad rtype = (\delta.fdecls\ fid).rtype \land \\ \quad \mathrm{ran}\, fmodes = \{rd\} \land \\ \quad etyp\ compat_\delta\ (\delta.fdecls\ fid).rtype \bullet \\ \\ \delta_n \vdash_{Form} formals \Longrightarrow (\delta.fdecls\ fid).ftypes, \\ \quad fmodes, (\delta.fdecls\ fid).fpos \\ \delta_m \downarrow_X \varnothing \downarrow_S \vdash_{SBasic} bdecl \Longrightarrow \delta_b \\ \delta_b \vdash_{SLater} ldecl \Longrightarrow \delta_l \\ \delta_l \vdash_{Stmt} stmt \\ \delta_l \vdash_{Exp} return : etyp\end{array}}{\delta \vdash_{FDecl} fdefn\ (\theta FDefnFDecl) \Longrightarrow \delta_f} \quad \text{(FDefn)}$$

where

$$\delta_n == \delta \downarrow_G (\delta.fdecls\ fid).glvars$$

$$\begin{array}{l}\delta_m == \delta_n \ [\ \ usedids := \delta_n.usedids \cup \mathrm{dom}(\delta.pdecls\ fid).ftypes, \\ \qquad\qquad varmodes := \delta_n.varmodes \cup (\mathrm{dom}(\delta.fdecl\ fid).ftypes \times \{rd\}), \\ \qquad\qquad vartypes := \delta_n.vartypes \cup (\delta.fdecl\ fid).ftypes, \\ \qquad\qquad bodies\_req := \varnothing, \\ \qquad\qquad scope := \delta_n.scope \frown \langle fid \rangle\ ]\end{array}$$

$$\begin{array}{l}\delta_f == \delta[\ \ sdef := \delta.sdef \setminus \{fid\}, \\ \qquad\qquad scall := \delta.scall \cup \{fid\}\ ]\end{array}$$

References: *FormalParam* p. 187; *SBasic* p. 131; *SLater* p. 135; *Stmt* p. 153; *Exp* p. 71; *Env* p. 12; $\vdash_{Form}$ p. 187; $\vdash_{SBasic}$ p. 131; $\vdash_{SLater}$ p. 135; $\vdash_{Stmt}$ p. 154; $\vdash_{Exp}$ p. 71; $\downarrow_G$ p. 197; $\downarrow_X$ p. 198; $\downarrow_S$ p. 199

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** ;	*fstub* ⟪ *pid* ↦ *f*, *params* ↦ ⟨ ⟪ *param* ↦ *x*, *mode* ↦ *in*, *ptype* ↦ *id T* ⟫⟩, *rtype* ↦ *id S* ⟫

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.

FStubFDecl

pid : *Id*
params : seq *FormalParam*
rtype : *IdDot*

FDecl ::= ... | *fstub*⟪⟪*FStubFDecl*⟫⟫

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	217
pstua	Separate declaration for a local procedure	220
flocl	Local function declaration in a package body	221
fstua	Separate declaration for a local function	224
kbody	Declaration of package body	225
kstub	Package body stub	226

The Well-Formation of Body Declarations

The well-formation of the declaration of a subprogram or package body is defined by a relation between the environment, the declaration and a modified environment. The declaration of this relation is:

$$\_ \vdash_{YDecl} \_ \Longrightarrow \_ \subseteq Env \times YDecl \times Env$$

Thus the well-formation predicate:

$$\delta \vdash_{YDecl} decl \Longrightarrow \delta'$$

can be read as "declaration *decl* is well-formed in the initial environment δ, with the modified environment δ'".

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* ⟪ *pid* ↦ *p*,
--# **global** y ;	*formals* ↦ ⟨⟩,
--# **derives** y **from** x ;	*globals* ↦ ⟨*id y*⟩,
is	*derives* ↦ ⟨...⟩,
declarations ...	*bdecl* ↦ ...,
begin	*ldecl* ↦ ...,
statements ...	*stmt* ↦ ... ⟫
end p;	

15.1.1 Abstract Syntax

A local procedure declaration combines the procedure declaration (Section 12.6) 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 Static Semantics

1. The name of the procedure must not be already declared.
2. The global annotation must be well-formed in the initial environment.
3. The formal parameters must be well-formed in the environment δ_g, whose variables are restricted to those from the global annotation and with the name of the procedure used.
4. The derives annotation must be well-formed in the environment δ_f, to which the formal parameters have been added as variables.
5. The derives annotation must be complete and consistent with the modes of the formal parameters:
   (a) All the global variables and formal parameters must appear as either exported variables (*exvars*) or imported variables (*imvars*) in the derives annotation.

(b) All formal parameters of mode **inout** must be imported. *This rule is required since such parameters are assumed to be defined when the sub-program is called* [AARM, §6.4.1 ¶5-9].

(c) No formal parameter of mode **out** may be used as import.

6. The local declarations of the procedure must be well-formed in the environment formed by restricting the writeable variables of the environment δ_f using the exported variable of the procedure and restricting the callable subprograms.

7. The statements of the procedure are well-formed in the environment modified by the local declarations.

8. The procedure declaration modifies the environment so that the procedure *pid* has been declared and can be called.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_b, \delta_l : Env;\ PLoclYDecl;\ Proc;\ imvars : \mathbb{P}\ IdDot \mid \\ \quad pid \notin \delta.usedids \land \\ \quad \operatorname{ran} globals \cup id(|\operatorname{dom} ftypes|) = exvars \cup imvars \land \\ \quad (\forall f : \operatorname{dom} fmodes \mid \{\ (f, rd), (f, wr)\ \} \subseteq fmodes \bullet \\ \qquad id\ f \in imvars) \land \\ \quad \neg\ (\exists f : \operatorname{dom} fmodes \mid (f, rd) \notin fmodes \land id\ f \in imvars) \bullet \\ \\ \delta \vdash_{Glob} globals \Longrightarrow glvars \\ \delta_g \vdash_{Form} formals \Longrightarrow ftypes, fmodes, fpos \\ \delta_f \vdash_{Deri} derives \Longrightarrow imvars, exvars \\ \delta_d \vdash_{SBasic} bdecl \Longrightarrow \delta_b \\ \delta_b \vdash_{SLater} ldecl \Longrightarrow \delta_l \\ \delta_l \vdash_{Stmt} stmt\end{array}}{\delta \vdash_{YDecl} plocl(\theta PLoclYDecl) \Longrightarrow \delta_p} \quad \text{(PLocl)}$$

where

$$\delta_g == (\delta[\ usedids := \delta.usedids \cup \{pid\}]) \downarrow_G glvars$$

$$\begin{array}{l}\delta_f == \delta_g[\ \ usedids := \delta_g.usedids \cup \operatorname{dom} ftypes, \\ \qquad\qquad varmodes := \delta_g.varmodes \cup fmodes, \\ \qquad\qquad vartypes := \delta_g.vartypes \cup ftypes, \\ \qquad\qquad bodies\_req := \varnothing, \\ \qquad\qquad scope := \delta_g.scope \frown \langle pid \rangle\ ]\end{array}$$

$$\delta_d == (\delta_f \downarrow_X exvars) \downarrow_S$$

$$\begin{array}{l}\delta_p == \delta[\ \ usedids := \delta.usedids \cup \{pid\}, \\ \qquad\qquad pdecls := \delta.pdecls \cup \{pid \mapsto \theta Proc\}, \\ \qquad\qquad scall := \delta.scall \cup \{pid\}\ ]\end{array}$$

References: PDeclSDecl p. 200; *PDefnFDecl* p. 208; *Env* p. 12; *Proc* p. 8; $\vdash_{Glob}$ p. 190; $\vdash_{Form}$ p. 187; $\vdash_{Deri}$ p. 194; $\vdash_{SBasic}$ p. 131; $\vdash_{SLater}$ p. 135; $\vdash_{Stmt}$ p. 154; $\downarrow_G$ p. 197; $\downarrow_X$ p. 198; $\downarrow_S$ p. 199

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) --# **global** y; --# **derives** y **from** x; **is separate** ;	*pstua* ⟪ *pid* ↦ *p*, *params* ↦ ⟨...⟩, *global* ↦ ⟨*id y*⟩, *derives* ↦ ⟨...⟩ ⟫

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.

PStuaYDecl

$pid : Id$
$params : \text{seq}\ FormalParam$
$global : GlobalAnnot$
$derives : \text{seq}\ DerivesAnnot$

$YDecl ::= \ldots \mid pstua\langle\langle PStuaYDecl \rangle\rangle$

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* ⟪ *fid* ↦ *f*,
- -# **global** y ;	*formals* ↦ ⟨⟩,
is	*rtype* ↦ *id S*,
declarations ...	*global* ↦ ⟨*id y*⟩,
begin	*bdecl* ↦ ...,
statements ...	*ldecl* ↦ ...,
return x + 1;	*stmt* ↦ ...,
end f;	*return* ↦ ... ⟫

15.3.1 Abstract Syntax

A local function declaration combines the function declaration (Section 12.7) 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 Static Semantics

1. The name of the function *fid* must not already be declared.
2. The global annotation must be well-formed in the initial environment.
3. The formal parameters must be well-formed in the environment δ_g, whose variables are restricted to those from the global annotation, with the name of the function used.
4. The formal parameters must all be of mode **in** or the default mode.
5. The return type of the function must be visible.
6. The local declarations of the function must be well-formed in the environment formed by:

(a) restricting the original environment using the global annotation of the function and the implicit empty list of global exports,

(b) adding the formal parameters to this environment as variables,

(c) preventing the declaration of the body of any package declared in the same scope as this function.

7. The statements of the function body must be well-formed in the environment enriched with the local declarations of the function.

8. In the modified environment δ_f, the identifier *fid* is used, the function can be called and the function is declared with:

(a) *ftypes*, *fpos* which describe the formal parameters,

(b) the return type *rtype*,

(c) the set of global variables *glvars* given by the *global* annotation.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_b, \delta_l : Env;\ FLoclYDecl;\ Fun;\ fmodes : Id \leftrightarrow RdWr;\\ \quad etyp : ExpType \mid\\ \qquad fid \notin \delta.usedids \land\\ \qquad is\_visible\_tmark_\delta\ rtype \land\\ \qquad \mathrm{ran}\, fmodes = \{rd\} \land\\ \qquad etyp\ compat_\delta\ rtype \bullet\\ \\ \delta \vdash_{Glob} globals \Longrightarrow glvars\\ \delta_g \vdash_{Form} formals \Longrightarrow ftypes, fmodes, fpos\\ \delta_m \vdash_{SBasic} bdecl \Longrightarrow \delta_b\\ \delta_b \vdash_{SLater} ldecl \Longrightarrow \delta_l\\ \delta_l \vdash_{Stmt} stmt\\ \delta_l \vdash_{Exp} return : etyp\end{array}}{\delta \vdash_{YDecl} flocl\ (\theta FLoclYDecl) \Longrightarrow \delta_f} \quad \text{(FLocl)}$$

where

$$\delta_g == (\delta[\ usedids := \delta.usedids \cup \{fid\}]) \downarrow_G glvars$$

$$\begin{array}{l}\delta_m == (\delta_g[\ \ usedids := \delta_g.usedids \cup \mathrm{dom}\, ftypes,\\ \qquad\qquad varmodes := \delta_g.varmodes \cup (\mathrm{dom}\, ftypes \times \{rd\}),\\ \qquad\qquad vartypes := \delta_g.vartypes \cup ftypes,\\ \qquad\qquad bodies\_req := \varnothing,\\ \qquad\qquad scope := \delta_g.scope \frown \langle fid \rangle\ ]\downarrow_X \varnothing) \downarrow_S\end{array}$$

$$\begin{array}{l}\delta_f == \delta[\ \ usedids := \delta.usedids \cup \{fid\},\\ \qquad\qquad fdecls := \delta.fdecls \cup \{fid \mapsto \theta Fun\},\\ \qquad\qquad scall := \delta.scall \cup \{fid\}\ ]\end{array}$$

References: FDeclSDecl p. 202; *FDefnFDecl* p. 212; *Env* p. 12; *Fun* p. 9; *RdWr* p. 7; *ExpType* p. 73; *is_visible_tmark*$_\delta$ p. 281; *compat*$_\delta$ p. 76; $\vdash_{Glob}$ p. 190; $\vdash_{Form}$ p. 187; $\vdash_{SBasic}$ p. 131; $\vdash_{SLater}$ p. 135; $\vdash_{Stmt}$ p. 154; $\downarrow_G$ p. 197; $\downarrow_X$ p. 198; $\downarrow_S$ p. 199

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* ⟪ *pid* ↦ *f*, *params* ↦ ⟨...⟩, *global* ↦ ⟨*id y*⟩, *rtype* ↦ *id S* ⟫

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.

FStuaYDecl

pid : *Id*
params : seq *FormalParam*
global : *GlobalAnnot*
rtype : *IdDot*

$YDecl ::= \ldots \mid fstua\langle\langle FStuaYDecl\rangle\rangle$

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* ⟪	*kid* ↦ *k*,
declarations ...		*rens* ↦ ⟨⟩,
begin		*bdecl* ↦ *basic declarations ...*,
statements ...		*ldecl* ↦ *later declarations ...*,
end k;		*init* ↦ *statements ...* ⟫

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 : seq *Ren*
bdecl : *KBasic*
ldecl : *KLater*
init : *Stmt*

$$YDecl ::= \ldots \mid kbody\langle\langle KBodyYDecl \rangle\rangle$$

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

$YDecl ::= \ldots \mid kstub\langle\langle Id \rangle\rangle$

Chapter 16

Renames

The use of renaming in SPARK is very restricted:

1. An identifier (or operator) which is visible by selection may be renamed to remove the package name prefix, making it directly visible *by its original name.*

2. Renaming declarations can only occur at the start of a scope, ensuring that an identifier to be renamed has not already been used by its existing name. After it has been renamed, it can only be used by its new name. Application of these rules results in three places where renames can be used in SPARK programs:

   (a) At the start of the body of either a package which is a compilation unit (see Section 17.4) or of an embedded package (see Section 15.5). In this case, the renames apply to operations imported from inherited packages.

   (b) Immediately following the specification of an embedded package (see Section 13.1). In this case, the renames apply to the operations exported by the embedded package.

   (c) At the start of a *main program* (see Sections 17.6 and 17.7). In this case, the renames apply to operations imported from inherited packages.

3. Only procedure and function subprograms (including operators) can be renamed. Objects and packages cannot be renamed.

Since SPARK does not allow the use of an operator which is visible by selection, an operator on a type from an inherited package must aways be renamed.

The following table summarises the Abstract Syntax of renames, which belong to the category *Ren*.

Syntax Constructor	Description	Page
proc	Procedure subprogram rename	229
fun	Function subprogram rename	232
opfun	Operator rename	234

The Well-Formation of Renaming Declarations

The well-formation of a renaming declaration is defined by a relation between the environment, the rename and a modified environment. The declaration of this relation is:

$$\_ \vdash_{Ren} \_ \Longrightarrow \_ \subseteq Env \times Ren \times Env$$

Thus the well-formation predicate:

$$\delta \vdash_{Ren} r \Longrightarrow \delta'$$

can be read as "rename r is well-formed in the initial environment δ, with the modified environment δ'".

The Well-Formation of List of Renames

Renames occur in the Abstract Syntax in list. In this Section, we give a well-formation predicate for a list of renames.

$$\_ \vdash_{Rens} \_ \Longrightarrow \_ \subseteq Env \times \operatorname{seq} Ren \times Env$$

An empty list of renames is always wellformed:

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{Rens} \langle\rangle \Longrightarrow \delta} \qquad \text{(RenS1)}$$

The well-formation of a non-empty list of renames depends on the well-formation of the first rename and the well-formation of the rest of the list in the new environment.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_r, \delta_s : Env;\ r : Ren;\ rs : \operatorname{seq} Ren \bullet \\ \quad \delta \vdash_{Ren} r \Longrightarrow \delta_r \\ \quad \delta_r \vdash_{Rens} rs \Longrightarrow \delta_s\end{array}}{\delta \vdash_{Rens} \langle r \rangle \frown rs \Longrightarrow \delta_s} \qquad \text{(RenS2)}$$

16.1 Procedure Rename

A procedure from an inherited package can be renamed so that it can be called by its original name.

Syntax Example	A.S. Representation
procedure p(x : in k.t) **renames** k.p;	$proc\ \langle\!\langle\ newid \mapsto p,\ formals \mapsto \langle\ \langle\!\langle\ param \mapsto x,\ mode \mapsto in,\ ptype \mapsto dot(k.t)\ \rangle\!\rangle\rangle,\ oldid \mapsto dot(k,p)\ \rangle\!\rangle$

16.1.1 Abstract Syntax

The new name is an identifier (Id), while the old one belongs to $IdDot$.

ProcRen

$newid : Id$
$formals : \text{seq}\ FormalParam$
$oldid : IdDot$

$Ren ::= \ldots \mid proc\langle\!\langle ProcRen \rangle\!\rangle$

16.1.2 Static Semantics

1. The procedure *oldid* must be visible, from a package in the set *withs*. (The procedure in not necessarily callable).
2. The new name *newid* must be unused and the same as the old name devoid of its package prefix.
3. The formal parameters must be the "same".
   (a) The formal parameter identifiers must be equal.
   (b) The formal parameter modes must be equivalent (**in** and the default mode are interchangeable).
   (c) The parameter type mark must refer to the same type (so that a local type in the original declaration is now refered to using the package prefix).

   Note that the well-formation rule for a formal parameter list of a renaming declaration differs from that of a formal parameter list in a subprogram declaration; the formal parameters of a rename do not declare objects and there is no requirement for the for parameter names to be unused.

4. In the new environment the procedure is visible by its new name, but is no longer visible by its old name.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ ProcRen;\ Proc;\ k : Id \mid \\ \quad oldid = dot(k, newid) \land \\ \quad k \in \delta.withs \land \\ \quad newid \in \mathrm{dom}(\delta.paks\ k).vis.pdecls \land \\ \quad newid \notin \delta.usedids \land \\ \quad glvars = proc\_glvars_\delta\ oldid \land \\ \quad exvars = proc\_exvars_\delta\ oldid \land \\ \quad ftypes = proc\_param\_tmarks_\delta\ oldid \land \\ \quad fpos = proc\_param\_ids_\delta\ oldid \land \\ \quad fmodes = proc\_param\_modes_\delta\ oldid \bullet \\ fpos, ftypes \circ fpos, (\lambda\, i : \mathrm{dom}\, fpos \bullet fmodes(\!|fpos\ i|\!)) \vdash_{SameF} formals\end{array}}{\delta \vdash_{Ren} proc(\theta ProcRen) \Longrightarrow \delta_r} \quad \text{(Proc)}$$

where

$$\begin{array}{l}\delta_r == \delta\ [\ usedids := \delta.usedids \cup \{newid\}, \\ \qquad pdecls := \delta.pdecls \cup \{\ newid \mapsto \theta Proc\ \}, \\ \qquad paks := \delta.paks \oplus \{\ k \mapsto (\delta.paks\ k)[\\ \qquad\quad vis := (\delta.paks\ k).vis\ [\\ \qquad\qquad pdecls := \{newid\} \lhd (\delta.paks\ k).vis.pdecls\ ]\ ]\ \}\ ]\end{array}$$

References: *FormalParam* p. 187; *Env* p. 12; *Proc* p. 8; $proc\_glvars_\delta$ p. 294; $proc\_exvars_\delta$ p. 294; $proc\_param\_tmarks_\delta$ p. 293; $proc\_param\_ids_\delta$ p. 293; $proc\_param\_modes_\delta$ p. 294; $\vdash_{SameF}$ p. 230

16.1.3 Equivalent Formal Parameter Specifications

The comparison between the formal specifications is described by the predicate $\vdash_{SameF}$, between the sequence of formal parameters and the (previously declared) sequences of formal parameter identifiers, types and mode sets.

$$\_, \_, \_ \vdash_{SameF} \_ \subseteq \mathrm{seq}\, Id \times \mathrm{seq}\, IdDot \times \mathbb{P}\, RdWr$$

Empty Parameter List This is well-formed if no (more) parameters are expected.

$$\frac{}{\langle\rangle, \langle\rangle, \langle\rangle \vdash_{SameF} \langle\rangle} \quad \text{(SameF1)}$$

Non-Empty Parameter List This is well-formed is the first parameter has matching identifier, type and mode, and the rest of the list is well-formed.

$$\frac{\begin{array}{l}\forall\, fs : \mathrm{seq}\, Id;\ ts : \mathrm{seq}\, IdDot;\ ms : \mathrm{seq}(\mathbb{P}\, RdWr);\ m : \mathbb{P}\, RdWr; \\ \quad FormalParam;\ ps : \mathrm{seq}\, FormalParam \mid \\ \qquad mode\_to\_rdwr\, mode = m \bullet \\ \quad fs, ts, ms \vdash_{SameF} ps\end{array}}{\begin{array}{l}\langle param \rangle \frown fs, \langle ptype \rangle \frown ts, \langle m \rangle \frown ms \vdash_{SameF} \\ \qquad \langle \theta FormalParam \rangle \frown ps\end{array}} \quad \text{(SameF2)}$$

References: *RdWr* p. 7; *FormalParam* p. 187; *mode_to_rdwr* p. 189

16.2 Function Rename

A function from an inherited package can be renamed so that it can be called by its original name.

Syntax Example	A.S. Representation
function f(x : in k.t) **return** k.s **renames** k.f;	$fun \langle\!\langle\ newid \mapsto f,$ $formals \mapsto \langle \langle\!\langle\ param \mapsto x,$ $mode \mapsto in,$ $ptype \mapsto dot(k.t)\ \rangle\!\rangle\rangle,$ $rtype \mapsto dot(k, s),$ $oldid \mapsto dot(k, f)\ \rangle\!\rangle$

16.2.1 Abstract Syntax

The new name is an identifier (Id), while the old one belongs to $IdDot$.

FunRen

$newid : Id$
$formals : \text{seq}\ FormalParam$
$rtype : IdDot$
$oldid : IdDot$

$Ren ::= \ldots \mid fun\langle\langle FunRen \rangle\rangle$

16.2.2 Static Semantics

1. The function *oldid* must be visible, from a package in the set *withs*. (The function in not necessarily callable).

2. The new name *newid* must be unused and the same as the old name devoid of its package prefix.

3. The formal parameters must be the "same".

   (a) The formal parameter identifiers must be equal.

   (b) The formal parameter modes must be equivalent; all the parameter are of mode **in** or the default mode.

   (c) The parameter type mark must refer to the same type (so that a local type in the original declaration is now refered to using the package prefix).

Note that the well-formation rule for a formal parameter list of a renaming declaration differs from that of a formal parameter list in a subprogram declaration; the formal parameters of a rename do not declare objects and there is no requirement for the for parameter names to be unused.

4. The return type mark must refer to the existing return type.

5. In the new environment the function is visible by its new name, but is no longer visible by its old name.

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ FunRen;\ Fun;\ k : Id \mid \\ \quad oldid = dot(k, newid) \land \\ \quad k \in \delta.withs \land \\ \quad newid \in \mathrm{dom}(\delta.paks\ k).vis.fdecls \land \\ \quad newid \notin \delta.usedids \land \\ \quad glvars = fun\_glvars_\delta\ oldid \land \\ \quad ftypes = fun\_param\_tmarks_\delta\ oldid \land \\ \quad fpos = fun\_param\_ids_\delta\ oldid \land \\ \quad rtype = fun\_rtn\_tmark_\delta\ oldid \bullet \\ fpos, ftypes \circ fpos, (\lambda\, i : \mathrm{dom}\, fpos \bullet \{rd\}) \vdash_{SameF} formals\end{array}}{\delta \vdash_{Ren} fun(\theta FunRen) \Longrightarrow \delta_r} \quad \text{(Fun)}$$

where

$$\begin{array}{l}\delta_r == \delta\ [\ usedids := \delta.usedids \cup \{newid\}, \\ \qquad fdecls := \delta.fdecls \cup \{\ newid \mapsto \theta Fun\ \}, \\ \qquad paks := \delta.paks \oplus \{\ k \mapsto (\delta.paks\ k)[\\ \qquad\quad vis := (\delta.paks\ k).vis\ [\\ \qquad\qquad fdecls := \{newid\} \lhd (\delta.paks\ k).vis.fdecls\ ]\ ]\ \}\ ]\end{array}$$

References: FormalParam p. 187; *Env* p. 12; *Fun* p. 9; fun_glvars_δ p. 292; $fun\_param\_tmarks_\delta$ p. 291; $fun\_param\_ids_\delta$ p. 291; $fun\_rtn\_tmark_\delta$ p. 292; $\vdash_{SameF}$ p. 230

16.3 Renaming Operators

If a type declared in the visible part of an inherited package specification has implicitly declared operators, these operators must be renamed to remove the package prefix before they can be used.

Syntax Example	A.S. Representation

function "+" (left, right: **in** k.t) **return** k.t **renames** k."+" ;

$$
\begin{array}{l}
opfun\ \langle\!|\ newop \mapsto splus, \\
\quad formals \mapsto \langle\ \langle\!|\ param \mapsto left, \\
\qquad\qquad\qquad mode \mapsto in, \\
\qquad\qquad\qquad ptype \mapsto dot(k,t)\ |\!\rangle, \\
\qquad\qquad \langle\!|\ param \mapsto right, \\
\qquad\qquad\qquad mode \mapsto in, \\
\qquad\qquad\qquad ptype \mapsto dot(k,t)\ |\!\rangle\rangle, \\
\quad rtype \mapsto dot(k,t), \\
\quad kid \mapsto k, \\
\quad oldop \mapsto splus\ |\!\rangle
\end{array}
$$

16.3.1 Abstract Syntax

A separate declaration of the operator symbols is required, since it combines both unary and binary operators.

$$
\begin{array}{l}
OSym ::= sand \mid sor \mid sxor \mid seq \mid sgt \mid slt \mid sgte \mid slte \\
\qquad \mid\ splus \mid sminus \mid smul \mid sdiv \mid srem \mid smod \mid sabs \mid snot \mid spower
\end{array}
$$

OpFunRen

$newop : OSym$
$formals : \mathrm{seq}\ FormalParam$
$rtype : IdDot$
$kid : Id$
$oldop : OSym$

$$Ren ::= \ldots \mid opfun\langle\!\langle OpFunRen \rangle\!\rangle$$

16.3.2 Static Semantics

1. The new and old operator symbols must be the same.

2. The package *kid* must be visible, belonging to the set *withs* in the initial environment.

3. The operand and return types must be the predefined types integer or boolean, or from the package *kid*. The function *op_simp_tmark* returns the simple name of an operator satisfying this rule.

$$
\begin{array}{|l}
op\_simp\_tmark : Id \to IdDot \nrightarrow Id \\
\hline
\forall\, k : Id \bullet \mathrm{dom}\, op\_simp\_tmark\, k = \\
\quad \{id\ integer, id\ boolean\} \cup \{\ t : Id \bullet dot(k,t)\ \} \\
\forall\, k : Id;\ t : Id \bullet \\
\quad op\_simp\_tmark\, k\ (id\ integer) = integer \land \\
\quad op\_simp\_tmark\, k\ (id\ boolean) = boolean \land \\
\quad op\_simp\_tmark\, k\ dot(k,t) = t
\end{array}
$$

4. The operator must be visible in the visible environment of the package *kid*.

5. Operators can only be renamed once; the new operator must not already be directly visible.

Binary Operator

6. The formal parameter identifiers must be "left" and "right".

$$
\begin{array}{|l}
left, right : Id
\end{array}
$$

7. The "not equals" operator cannot be explicitly renamed; its is implicitly renamed with the "equals" operator.

8. The correspondance between operator symbols and binary operators is given by the following functions.

$$
\begin{array}{|l}
OSym\_to\_Bop : OSym \nrightarrow Bop \\
\hline
OSym\_to\_Bop = \{ \\
\quad sand \mapsto and, sor \mapsto or, sxor \mapsto xor, \\
\quad seq \mapsto eq, sgt \mapsto gt, slt \mapsto lt, \\
\quad sgte \mapsto gte, slte \mapsto lte, splus \mapsto plus, \\
\quad sminus \mapsto minus, smul \mapsto mul, sdiv \mapsto div, \\
\quad srem \mapsto rem, smod \mapsto mod\}
\end{array}
$$

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ OpFunRen;\ lbtyp, rbtyp, nbtyp : BasicType \mid \\ \quad newop = oldop \wedge \\ \quad kid \in \delta.withs \wedge \\ \quad \#formals = 2 \wedge \\ \quad (formals\ 1).param = left \wedge \\ \quad (formals\ 2).param = right \wedge \\ \quad mode\_to\_rdwr(formals\ 1).mode = \{rd\} \wedge \\ \quad mode\_to\_rdwr(formals\ 2).mode = \{rd\} \wedge \\ \quad lbtyp = nameT\ (id\ (op\_simp\_tmark(formals\ 1).ptype)) \wedge \\ \quad rbtyp = nameT\ (id\ (op\_simp\_tmark(formals\ 2).ptype)) \wedge \\ \quad nbtyp = nameT\ (id\ (op\_simp\_tmark\ rtype)) \wedge \\ \quad (\delta.paks\ k).vis.binops\ (lbtyp, bop, rbtyp) = nbtyp \wedge \\ \quad (nameT\ (formals\ 1).ptype, bop, \\ \qquad nameT\ (formals\ 2).ptype) \notin \delta.binops \bullet\end{array}}{\delta \vdash_{Ren} opfun(\theta OpFunRen) \Longrightarrow \delta_r} \quad \text{(OpFun1)}$$

where

$$bop == OSym\_to\_Bop\ newop$$

$$\begin{array}{l}\delta_r == \ \textbf{if}\ newop \neq seq\ \textbf{then} \\ \qquad \delta\ [\ binops := \delta.binops \cup \{ \\ \qquad\quad (nameT\ (formals\ 1).ptype, bop, \\ \qquad\qquad nameT\ (formals\ 2).ptype) \mapsto rtype\ \}\ ] \\ \quad \textbf{else} \\ \qquad \delta\ [\ binops := \delta.binops \cup \{ \\ \qquad\quad (nameT\ (formals\ 1).ptype, eq, \\ \qquad\qquad nameT\ (formals\ 2).ptype) \mapsto rtype, \\ \qquad\quad (nameT\ (formals\ 1).ptype, noteq, \\ \qquad\qquad nameT\ (formals\ 2).ptype) \mapsto rtype\ \}\ ]\end{array}$$

Unary Operator

9. The formal parameter identifier must be "right".

10. The correspondance between operator symbols and unary operators is given by the following functions.

$$\begin{array}{|l} OSym\_to\_Uop : OSym \nrightarrow Uop \\ \hline OSym\_to\_Uop = \{ \quad sabs \mapsto abs, snot \mapsto not, spower \mapsto power\}\end{array}$$

$$\frac{\begin{array}{l}\forall\, \delta : Env;\ OpFunRen;\ rbtyp, nbtyp : BasicType \mid \\ \qquad newop = oldop \wedge \\ \qquad kid \in \delta.withs \wedge \\ \qquad \#formals = 1 \wedge \\ \qquad (formals\ 1).param = right \wedge \\ \qquad mode\_to\_rdwr(formals\ 1).mode = \{rd\} \wedge \\ \qquad rbtyp = nameT\ (id\ (op\_simp\_tmark(formals\ 1).ptype)) \wedge \\ \qquad nbtyp = nameT\ (id\ (op\_simp\_tmark\ rtype)) \wedge \\ \qquad (\delta.paks\ k).vis.unops\ (uop, rbtyp) = nbtyp \wedge \\ \qquad (uop, nameT\ (formals\ 1).ptype)) \notin \delta.unops \bullet\end{array}}{\delta \vdash_{Ren} opfun(\theta OpFunRen) \Longrightarrow \delta_r} \quad \text{(OpFun2)}$$

where

$$uop == OSym\_to\_Uop\ newop$$

$$\delta_r == \delta\ [\ unops := \delta.unops \cup \{\ (uop, nameT\ (formals\ 1).ptype) \mapsto rtype\ \}\ ]$$

References: *FormalParam* p. 187; *integer* p. 265; *boolean* p. 265; *Bop* p. 102; *Env* p. 12; *BasicType* p. 73; *mode_to_rdwr* p. 189; *nameT* p. 73; *Uop* p. 100

Chapter 17

Compilation Units and SPARK Texts

A SPARK program is a collection of one or more compilation units. A compilation unit 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	244
bodyu	package body	248
subu	subunit, i.e. something declared to be separate	251
pmain	procedure main program	252
fmain	function main program	256

A complete SPARK Program (*SPARK*, page 259) is also described in this Chapter.

Well-formation of Compilation Unit

The well-formation of a compilation unit is specified by a relation between the environment, the compilation unit and the environment modified by the declaration of the compilation unit. This relation is declared as:

$$\_ \vdash_{Unit} \_ \Longrightarrow \_ \subseteq Env \times Unit \times Env$$

Thus a well-formation predicate:

$$\delta \vdash_{Unit} comp \Longrightarrow \delta'$$

can be read as "the compilation unit *comp* is well-formed in the environment δ, yielding the modified environment δ'".

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 Own Variables

The well-formation of the **own** annotation is described by a relation:

$$\_ \vdash_{Own} \_ \Longrightarrow \_ \subseteq Env \times \mathrm{seq}\, Id \times Env$$

1. The own variable identifiers must be unused.
2. The list of own variables must not contain duplicates.

The empty list of own variables is well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{Own} \langle\rangle \Longrightarrow \delta} \qquad \text{(Own1)}$$

An own variable is declared with both read and write modes.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_v : Env;\ vs : \mathrm{seq}\, Id;\ v : Id \mid \\ \qquad v \notin \delta_v.usedids \bullet \\ \delta \vdash_{Own} vs \Longrightarrow \delta_v\end{array}}{\delta \vdash_{Own} \langle v \rangle \frown vs \Longrightarrow \delta'} \qquad \text{(Own2)}$$

where

$$\begin{array}{l}\delta' == \delta_v[\ \ usedids := \delta.usedids \cup \{v\}, \\ \qquad\qquad varmodes := \delta.varmodes \cup \{v \mapsto rd, v \mapsto wr\}, \\ \qquad\qquad ownvars := \delta.ownvars \cup \{v\}\ ]\end{array}$$

References: *Env* p. 12

17.1.3 Initialised Own Variables

The well-formation of the **initializes** annotation is described by a relation:

$$\_ \vdash_{Init} \_ \Longrightarrow \_ \subseteq Env \times \mathrm{seq}\, Id \times Env$$

1. Only own variables can be initialised.
2. The list of initialised own variables may not contain duplicates.

The empty initialisation list is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{Init} \langle\rangle \Longrightarrow \delta} \qquad \text{(Init1)}$$

An initialised own variable is added to the environment set *initowns*.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_i : Env;\ is : \mathrm{seq}\, Id;\ i : Id \mid \\ \qquad i \in \mathrm{dom}\, \delta.ownvars \land \\ \qquad i \notin \delta_i.initowns \bullet \\ \delta \vdash_{Init} is \Longrightarrow \delta_i\end{array}}{\delta \vdash_{Init} \langle i \rangle \frown is \Longrightarrow \delta'} \qquad \text{(Init2)}$$

where

$$\delta' == \delta_i \ [\ initowns := \delta_i.initowns \cup \{i\}\ ]$$

References: Env p. 12

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 Inherit Annotation

The well-formation of the **inherit** annotation is described by the relation:

$$\_ \vdash_{Inh} \_ \Longrightarrow \_ \subseteq Env \times \mathrm{seq}\, Id \times Env$$

1. The empty list of inherited packages is well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{Inh} \langle\rangle \Longrightarrow \delta} \qquad \text{(Inher1)}$$

2. An inherited package must have a declared specification.

   Since a package specification must be declared before the package can be named in an **inherit** *annotation, cycles cannot occur in the "inherit" relation.*

3. The list of inherited packages must not contain duplicates.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_i : Env, i : Id, is : \mathrm{seq}\, Id \mid \\ \qquad i \in \mathrm{dom}\, \delta.paks \wedge \\ \qquad i \notin \delta_i.inherits \bullet \\ \quad \delta \vdash_{Inh} is \Longrightarrow \delta_i\end{array}}{\delta \vdash_{Inh} \langle i \rangle \frown is \Longrightarrow \delta'} \qquad \text{(Inher2)}$$

where

$$\delta' == \delta_i[\ \ usedids := \delta_i.usedids \cup \{i\},\ \ inherits := \delta_i.inherits \cup \{i\}\ ]$$

References: Env p. 12

17.2.3 The With Clause

The well-formation of the **with** clause is described by the relation:

$$\_ \vdash_{With} \_ \Longrightarrow \_ \subseteq Env \times \mathrm{seq}\, Id \times Env$$

1. The empty **with** clause is always well-formed.

$$\frac{\forall\, \delta : Env \bullet}{\delta \vdash_{With} \langle\rangle \Longrightarrow \delta} \qquad \text{(With1)}$$

2. Packages listed in the context clause *must also be inherited.* (**SLI WJ018**).

3. The list following the context clause **with** must not contain duplicates.

4. *In Ada, the packages named in the* **with** *clause of a package body are additional to those given in the package specification; similarly the* **with** *clause on a subunit adds to the packages "withed" at the point of declaration of the separate body [AARM, §10.1.1 ¶4]. A rule may be required in SPARK concerning repetition of packages in the combined list* **SLI WJ019**.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_w : Env;\ w : Id;\ ws : \mathrm{seq}\, Id \mid \\ \qquad w \in \delta.inherits \wedge \\ \qquad w \notin \mathrm{ran}\, ws \bullet \\ \delta \vdash_{With} ws \Longrightarrow \delta_w\end{array}}{\delta \vdash_{With} \langle w \rangle \frown ws \Longrightarrow \delta'} \qquad \text{(With2)}$$

where

$$\delta' == \delta_w[\ \ with s := \delta_w.with s \cup \{w\}\ ]$$

References: Env p. 12

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 ⟪  with ↦ ⟨l⟩,
--# inherit l, m;                         inherit ↦ ⟨l, m⟩,
package k                                kid ↦ k,
--# own x,y;                             own ↦ ⟨x, y⟩,
--# initializes x;                       init ↦ ⟨x⟩,
is                                       vdecl ↦ ...,
     declarations ...                    pdecl ↦ dnull  ⟫
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.

SpecUnit

$with$: seq Id
kid : Id
$inherit$: seq Id
own : seq Id
$init$: seq Id
$vdecl$: $VBasic$
$pdecl$: $PBasic$

$Unit ::= specu\langle\langle SpecUnit\rangle\rangle \mid \ldots$

17.3.2 Static Semantics

1. The package name must be distinct from the names of all other library units and unused in the initial environment.

Library units in SPARK comprise package specifications and the main program. We define a function $Library\_Unit\_Ids_\delta$ which returns the set of library unit names below.

$$Library\_Unit\_Ids : Env \rightarrow \mathbb{P}\ Id$$

$$\forall\, \delta : Env \bullet$$
$$Library\_Unit\_Ids_\delta = \mathrm{dom}\, \delta.paks \cup \delta.main$$

2. The **inherit** annotation is well-formed in the initial environment, with the package name used, giving environment δ_i.

3. The **with** clause must be well-formed in the environment δ_i, giving environment δ_w.

4. The **own** variable annotation must be well-formed in the environment δ_w, giving environment δ_o.

5. The **initializes** annotation must be well-formed in the environment δ_o, giving environment δ_l.

6. The visible declarations must be well-formed in the environment δ_l, giving environment δ_v.

7. The private declarations must be well-formed in the environment δ_v, giving environment δ_p. All deferred constants and private types of the visible part must have a full declaration.

8. All the variables declared in the specification must be named in the **own** annotation. However, not all the own-variables need be declared in the package specification.

9. The compilation unit modifies the environment by associating the package name *kid* with the *SpecEnv* generated from the declarations in the specification (see below). All the subprograms of the package are callable in the visible environment. A body is required for this package if any of the following conditions are satisfied:

   (a) the specification declares subprograms,

   (b) the package has **own** variables which are *not* declared in the specification,

   (c) the package has **own** variables which are initialised.

   Note that the package identifier is **not** *added to usedids in the resulting environment; this happens when its is inherited by another library unit.*

$$\frac{\begin{array}{l}\forall\, \delta, \delta_i, \delta_w, \delta_o, \delta_l, \delta_v, \delta_p : Env;\ SpecUnit \mid \\ \quad kid \notin Library\_Unit\_Ids_\delta \cup \delta.usedids \land \\ \quad \delta_p.defcons = \varnothing \land \\ \quad (\forall\, t : \mathrm{dom}\, \delta_p.types \bullet \\ \quad\quad \delta_p.types\ t \neq privT \land \delta_p.types\ t \neq limT) \land \\ \quad \mathrm{dom}\, \delta_p.varmodes \subseteq \mathrm{dom}\, \delta_o.varmodes \bullet \\ \\ \delta' \vdash_{Inh} inherit \Longrightarrow \delta_i \\ \delta_i \vdash_{With} with \Longrightarrow \delta_w \\ \delta_w \vdash_{Own} own \Longrightarrow \delta_o \\ \delta_o \vdash_{Init} init \Longrightarrow \delta_l \\ \delta_l \vdash_{VBasic} vdecl \Longrightarrow \delta_v \\ \delta_v \vdash_{PBasic} pdecl \Longrightarrow \delta_p\end{array}}{\delta \vdash_{Unit} specu(\theta SpecUnit) \Longrightarrow \delta''} \quad \text{(SpecU)}$$

where

$$\delta' == \delta[\ \ usedids := \delta.usedids \cup \{kid\}\ ]$$

$$\begin{array}{l}\delta'' == \delta[\ \ paks := \delta.paks \cup \{kid \mapsto (\mu\, Pak \mid \\ \quad\quad vis = SpecEnv\_less\_Standard(Env\_to\_SpecEnv\ \delta_v) \\ \quad\quad\quad [\ scall := \mathrm{dom}\, \delta_v.pdecls \cup \mathrm{dom}\, \delta_v.fdecls\ ], \\ \quad\quad priv = Env\_to\_SpecEnv\ \delta_p, \\ \quad\quad scope = \langle kid \rangle, \\ \quad\quad inherits = \delta_i.inherits, \\ \quad\quad withs = \delta_w.withs)\ \}, \\ \quad bodies\_req := \delta.bodies\_req \cup \{k : Id \mid \\ \quad\quad k = kid \land \\ \quad\quad (\delta_v.sdef \neq \varnothing \lor \\ \quad\quad (\exists\, own : \delta_p.ownvars \mid own \notin \mathrm{dom}\, \delta_p.vartypes) \lor \\ \quad\quad \delta_l.initowns \neq \varnothing)\ \}\ ]\end{array}$$

Forming the *SpecEnv* The declarations in the visible and private parts of the specification are recored in the environment using the type *SpecEnv*, which has only a subset of the fields of the complete environment *Env*.

$$\begin{array}{|l}Env\_to\_SpecEnv : Env \to SpecEnv \\ \hline \forall\, Env;\ SpecEnv \bullet \\ \quad Env\_to\_SpecEnv(\theta Env) = \theta SpecEnv\end{array}$$

Removing the Standard Declarations The identifiers which are directly visible at the end of the package specification include those declared in the package STANDARD in the predefined environment (see A). Since these predefined identifiers are not included in the visible *SpecEnv* which is associated with the package in the environment, a function is required to delete them.

$$
\begin{array}{|l}
SpecEnv\_less\_Standard : SpecEnv \rightarrow SpecEnv \\
\hline
\forall \delta : SpecEnv \bullet \\
\quad SpecEnv\_less\_Standard\, \delta = \delta\, [\\
\qquad usedids := \delta.usedids \setminus (\theta StandardEnv).usedids, \\
\qquad types := \delta.types \setminus (\theta StandardEnv).types, \\
\qquad subtypes := \delta.subtypes \setminus (\theta StandardEnv).subtypes, \\
\qquad unops := \delta.unops \setminus (\theta StandardEnv).unops, \\
\qquad binops := \delta.binops \setminus (\theta StandardEnv).binops\ ]
\end{array}
$$

References: *VBasic* p. 127; *PBasic* p. 129; *Env* p. 12; *SpecEnv* p. 10; *StandardEnv* p. 271; *privT* p. 15; *limT* p. 15; $\vdash_{Inh}$ p. 242; $\vdash_{With}$ p. 243; $\vdash_{Own}$ p. 240; $\vdash_{Init}$ p. 241; $\vdash_{VBasic}$ p. 127; $\vdash_{PBasic}$ p. 129;

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* ⟪	*with* ↦ ⟨*j*⟩,
package body k **is**		*kid* ↦ *k*,
declarations ...		*rens* ↦ ⟨⟩,
begin		*bdecl* ↦ *basic declarations ...*,
statements ...		*ldecl* ↦ *later declarations ...*,
end k;		*init* ↦ *statements ...* ⟫

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 ::= ... | *bodyu*⟪*BodyUnit*⟫

17.4.2 Static Semantics

1. The package name must identify a specification in the initial environment whose body can be defined in this environment.

2. The list of identifiers following the **with** clause must be well-formed in the initial environment enriched with the declarations from the visible and private parts of the package specification, with

   (a) all the procedures of the specification requiring definition, none being callable, and

(b) no package bodies requiring a declaration.

3. The renaming declarations *ren* must be well-formed in the environment δ_w resulting from the well-formation of the **with** clause.

4. The local declarations *bdecl* and *ldecl* must be well-formed.

5. All the variables declared in the (outer-most scope of) the package body must be named in the own annotation.

6. The package body must be complete:

   (a) A declaration is required for all the package's own variables.

   (b) A definition must be provided for all the subprograms declared in the package specification.

   (c) A body must have been provided for all embedded packages which require bodies.

7. The initialisation statement *init* must be well-formed in the environment δ_l, restricted so that the own variables given in the **initializes** annotation of the package specification are the only variables and no subprograms can be called.

8. In the environment modified by the declaration of the package body, this package no longer requires a body.

9. The identifiers appearing in the **with** clause must appear in the **inherit** annotation of the corresponding package specification.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_w, \delta_b, \delta_l, \delta_r : Env;\ BodyUnit \mid \\ \qquad kid \in bodies\_req \wedge \\ \qquad \mathrm{dom}\, \delta_b.vartypes = (\delta.paks\ kid).ownvars \wedge \\ \qquad \delta_l.sdef = \varnothing \wedge \\ \qquad \delta_l.bodies\_req = \varnothing \\ \\ \delta' \vdash_{With} with \Longrightarrow \delta_w \\ \delta_w \vdash_{Rens} rens \Longrightarrow \delta_r \\ \delta_r \vdash_{KBasic} bdecl \Longrightarrow \delta_b \\ \delta_b \vdash_{KLater} ldecl \Longrightarrow \delta_l \\ \delta'' \vdash_{Stmt} init\end{array}}{\delta \vdash_{Unit} bodyu(\theta BodyUnit) \Longrightarrow \delta'''} \quad \text{(BodyU)}$$

where

$$
\begin{aligned}
\delta' == \delta[\ & usedids := \delta.usedids \cup (\delta.paks\ kid).priv.usedids,\\
& contypes := (\delta.paks\ kid).priv.contypes,\\
& convals := (\delta.paks\ kid).priv.convals,\\
& defcons := \varnothing,\\
& types := (\delta.paks\ kid).priv.types,\\
& subtypes := (\delta.paks\ kid).priv.subtypes,\\
& unops := (\delta.paks\ kid).priv.unops,\\
& binops := (\delta.paks\ kid).priv.binops,\\
& vartypes := (\delta.paks\ kid).priv.vartypes,\\
& varmodes := (\delta.paks\ kid).priv.varmodes,\\
& ownvars := (\delta.paks\ kid).priv.ownvars,\\
& pdecls := (\delta.paks\ kid).priv.pdecls,\\
& fdecls := (\delta.paks\ kid).priv.fdecls,\\
& sdef := (\delta.paks\ kid).priv.sdef,\\
& scall := \varnothing,\\
& withs := (\delta.paks\ kid).withs,\\
& inherits := (\delta.paks\ kid).inherits,\\
& bodies\_req := \varnothing\ ]
\end{aligned}
$$

$$
\begin{aligned}
\delta'' == \delta_l[\ & vartypes := (\delta.paks\ kid).priv.initowns \lhd \delta_l.vartypes,\\
& varmodes := (\delta.paks\ kid).priv.initowns \times \{wr, rd\},\\
& scall := \varnothing,\\
& paks := (\lambda\, k : \delta_l.inherits \bullet (\delta_l.paks\ k)[\\
& \qquad vis := (\delta_l.paks\ k).vis[\ vartypes := \varnothing,\\
& \qquad\qquad\qquad\qquad varmodes := \varnothing,\\
& \qquad\qquad\qquad\qquad scall := \varnothing\ ]\ ])\ ]
\end{aligned}
$$

$$\delta''' == \delta[\ bodies\_req := \delta.bodies\_req \setminus \{kid\}\ ]$$

References: *Ren* p. 227; *KBasic* p. 133; *KLater* p. 137; *Stmt* p. 153; *Env* p. 12; $\vdash_{Rens}$ p. 228; $\vdash_{With}$ p. 243; $\vdash_{KBasic}$ p. 133; $\vdash_{KLater}$ p. 137; $\vdash_{Stmt}$ p. 154

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; **separate** (Parent) *declaration* ...	*subu* ⟪ *withs* ↦ ⟨*K*, *L*⟩, *parent* ↦ ⟨*Parent*⟩, *sdecl* ↦ ... ⟫

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

withs : seq *Id*
parent : seq *Id*
sdecl : *Decl*

CUnit ::= ... | *subu*⟪*SubUCUnit*⟫

17.6 Main Program — Procedure

In SPARK, subprograms cannot generally be library units (though they can be subunits). However, a complete SPARK program must contain one subprogram which is the entry point of the program and is a library unit. It is distinguished by the **main_program** annotation. The main program may be a procedure; SPARK allows the procedure to have parameters.

Syntax Example	A.S. Representation

with k;
--# **inherit** k;
--# **main_program**
procedure SystemW
--# **global** k.u, k.v ;
--# **derives** k.u **from** k.v ;
is
    renames ...
    declarations ...
begin
    statements ...
end SystemW ;

$$
\begin{array}{ll}
pmain \langle\!| & with s \mapsto \langle k \rangle, \\
 & inherit \mapsto \langle k \rangle, \\
 & mid \mapsto SystemW, \\
 & formals \mapsto \langle \rangle, \\
 & globals \mapsto \langle dot(k,u), dots(k,v) \rangle, \\
 & derives \mapsto \langle\!| \ export \mapsto dot(k,u), \\
 & \qquad\qquad imports \mapsto \langle dot(k,v) \rangle \ |\!\rangle, \\
 & rens \mapsto \langle \ldots \rangle, \\
 & bdecl \mapsto \ldots, \\
 & ldecl \mapsto \ldots, \\
 & stmt \mapsto \ldots \ |\!\rangle
\end{array}
$$

17.6.1 Abstract Syntax

The main program has a with list and inherit annotation, and also the components of a local procedure. A list of renames may be used immediately before the local declarations.

PMainUnit

$$
\begin{array}{l}
with : \text{seq } Id \\
inherit : \text{seq } Id \\
mid : Id \\
formals : \text{seq } FormalParam \\
globals : GlobalAnnot \\
derives : \text{seq } DerivesAnnot \\
renames : \text{seq } Ren \\
bdecl : SBasic \\
ldecl : SLater \\
stmt : Stmt
\end{array}
$$

$$Unit ::= \ldots \mid pmain\langle\langle PMainUnit \rangle\rangle$$

17.6.2 Static Semantics

The distinctive rules for main programs are:

1. Only one main program is allowed in any given text.

2. The imported global variables must be initialised own variables of inherited packages. The function $Initialised\_Own\_Vars_\delta$ returns these variables.

$$
\begin{array}{|l}
Initialised\_Own\_Vars_{Env} : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad Initialised\_Own\_Vars_\delta = \\
\qquad \{\ k : \delta.inherits;\ v : IdDot \mid \\
\qquad\quad v \in (\delta.paks\ k).vis.initowns \bullet dot(k, v)\ \}
\end{array}
$$

Other rules are similar to those of a local procedure.

3. The procedure name must be distinct from the names of all other library units and unused in the initial environment.

4. The **inherit** annotation is well-formed in the initial environment giving environment δ_i.

5. The **with** clause must be well-formed in the environment δ_i, giving environment δ_w.

6. The global annotation must be well-formed in the environment δ_w.

7. The formal parameters must be well-formed in the environment δ_f, which is δ_w with the name of the procedure used.

8. The derives annotation must be well-formed in the environment δ_d, to which the formal parameters have been added as variables.

9. The derives annotation must be complete and consistent with the modes of the formal parameters:

   (a) All the global variables and formal parameters must appear as either exported variables (*exvars*) or imported variables (*imvars*) in the derives annotation.

   (b) All formal parameters of mode **inout** must be imported. *This rule is required since such parameters are assumed to be defined when the sub-program is called* [AARM, §6.4.1 ¶5-9].

   (c) No formal parameter of mode **out** may be used as import.

10. The renaming declarations must be well-formed in the environment formed by restricting the writeable variables of the environment δ_f using the exported variables of the procedure, restricting the callable subprograms and adding the formal parameter identifiers as used names.

11. The local declarations must be well-formed in the environment created by the renaming declarations.

12. The statements of the procedure are well-formed in the environment modified by the local declarations.

$$
\begin{array}{l}
\forall\, \delta, \delta_i, \delta_w, \delta_r, \delta_b, \delta_l : Env;\ PMainUnit;\ Proc;\ imvars : \mathbb{P}\ IdDot \mid \\
\qquad mid \notin \delta.usedids \land \\
\qquad \delta.main = \varnothing \land \\
\qquad mid \notin Library\_Unit\_Ids_\delta \land \\
\qquad \mathrm{ran}\ globals \cup id(|\mathrm{dom}\ ftypes|) = exvars \cup imvars \land \\
\qquad (\forall f : \mathrm{dom}\ fmodes \mid \{\ (f, rd), (f, wr)\ \} \subseteq fmodes \bullet \\
\qquad\qquad id\ f \in imvars) \land \\
\qquad \neg\ (\exists f : \mathrm{dom}\ fmodes \mid (f, rd) \notin fmodes \land id\ f \in imvars) \land \\
\qquad glvars \cap imvars \subseteq Initialised\_Own\_Vars_{\delta_i} \bullet \\
\delta \vdash_{Inh} inherit \Longrightarrow \delta_i \\
\delta_i \vdash_{With} with \Longrightarrow \delta_w \\
\delta_w \vdash_{Glob} globals \Longrightarrow glvars \\
\delta_f \vdash_{Form} formals \Longrightarrow ftypes, fmodes, fpos \\
\delta_d \vdash_{Deri} derives \Longrightarrow imvars, exvars \\
\delta_e \vdash_{Rens} renames \Longrightarrow \delta_r \\
\delta_r \vdash_{SBasic} bdecl \Longrightarrow \delta_b \\
\delta_b \vdash_{SLater} ldecl \Longrightarrow \delta_l \\
\delta_l \vdash_{Stmt} stmt \\
\hline
\delta \vdash_{Unit} pmain(\theta PMainUnit) \Longrightarrow \delta_m
\end{array}
\qquad \text{(PMain)}
$$

where

$$\delta_f == \delta_w\ [usedids := \delta.usedids \cup \{mid\}]$$

$$
\begin{array}{l}
\delta_d == \delta_f\ [\ varmodes := \delta_f.varmodes \cup fmodes, \\
\qquad\qquad vartypes := \delta_f.vartypes \cup ftypes\ ]
\end{array}
$$

$$\delta_x == (\delta_f \downarrow_X exvars) \downarrow_S$$

$$
\begin{array}{l}
\delta_e == \delta_x\ [\ usedids := \delta_x.usedids \cup \mathrm{dom}\ ftypes, \\
\qquad\qquad bodies\_req := \varnothing, \\
\qquad\qquad scope := \langle mid \rangle\ ]
\end{array}
$$

$$\delta_m == \delta\ [main := \{mid\}]$$

References: FormalParam p. 187; *GlobalAnnot* p. 190; *DerivesAnnot* p. 194; *Ren* p. 227; *SBasic* p. 131; *SLater* p. 135; *Stmt* p. 153; *Env* p. 12; *Proc* p. 8; $Library\_Unit\_Ids_{\delta}$ p. 245; $\vdash_{Inh}$ p. 242; $\vdash_{With}$ p. 243; $\vdash_{Glob}$ p. 190; $\vdash_{Form}$ p. 187; $\vdash_{Deri}$ p. 194; $\vdash_{Rens}$ p. 228; $\vdash_{SBasic}$ p. 131; $\vdash_{SLater}$ p. 135; $\vdash_{Stmt}$ p. 154; $\downarrow_{X}$ p. 198; $\downarrow_{S}$ p. 199

17.7 Main Program — Function

In SPARK, subprograms cannot generally be library units (though they can be subunits). However, a complete SPARK program must contain one subprogram which is the entry point of the program and is a library unit. It is distinguished by the **main_program** annotation. The main program may be a function.

Syntax Example	A.S. Representation

with k;
--# **inherit** k;
--# **main_program**
function SysF(x: **in** k.t) **return** k.t
--# **global** k.u, k.v ;
is
    renames ...
    declarations ...
begin
    statements ...
    return x + 1;
end SysF ;

$fmain \langle\!|$ $withs \mapsto \langle k \rangle$,
$inherit \mapsto \langle k \rangle$,
$mid \mapsto SystemW$,
$formals \mapsto \langle\rangle$,
$rtype \mapsto dot(k, t)$,
$globals \mapsto \langle dot(k, u), dots(k, v) \rangle$,
$rens \mapsto \langle \ldots \rangle$,
$bdecl \mapsto \ldots$,
$ldecl \mapsto \ldots$,
$stmt \mapsto \ldots$,
$return \mapsto \ldots$ $|\!\rangle$

17.7.1 Abstract Syntax

The main program has a with list and inherit annotation, and also the components of a local function. A list of renames may be used immediately before the local declarations.

FMainUnit

$withs$: seq Id
$inherit$: seq Id
mid : Id
$formals$: seq $FormalParam$
$globals$: $GlobalAnnot$
$rtype$: $IdDot$
$renames$: seq Ren
$bdecl$: $SBasic$
$ldecl$: $SLater$
$stmt$: $Stmt$
$return$: Exp

$Unit ::= \ldots \mid fmain\langle\langle FMainUnit \rangle\rangle$

17.7.2 Static Semantics

The distinctive rules for function main programs are:

1. Only one main program is allowed in any given text.

2. The global variables must be initialised own variables of inherited packages.

Other rules are similar to those of a local function.

3. The function name must be distinct from the names of all other library units and unused in the initial environment.

4. The **inherit** annotation is well-formed in the initial environment δ, giving the new environment δ_i.

5. The **with** clause must be well-formed in the environment δ_i, giving the new environment δ_w.

6. The global annotation must be well-formed in the environment δ_w.

7. The formal parameters must be well-formed in the environment δ_f, which is δ_w with the name of the function used.

8. The formal parameters must all of mode **in** or the default mode.

9. The return type must be visible and not unconstrained.

10. The renaming declarations must be well-formed in the environment formed by restricting writeable variables of the environment δ_f using an empty set of exported variables, restricting the callable subprograms, and adding the formal parameter identifiers as used names.

11. The local declarations must be well-formed in the environment created by the renaming declarations.

12. The statements of the procedure are well-formed in the environment modified by the local declarations.

13. The return expression must be well-formed with a type which is compatible with the return type.

$$\frac{\begin{array}{l}\forall\, \delta, \delta_i, \delta_w, \delta_r, \delta_b, \delta_l : Env;\ FMainUnit;\ Fun;\ fmodes : Id \leftrightarrow RdWr;\\ \quad etyp : ExpType \mid\\ \qquad mid \notin \delta.usedids \land\\ \qquad \delta.main = \varnothing \land\\ \qquad mid \notin Library\_Unit\_Ids_\delta \land\\ \qquad is\_visible\_tmark_\delta\ rtype \land\\ \qquad \neg\ is\_unconstrained\_tmark_\delta\ rtype \land\\ \qquad \operatorname{ran} fmodes = \{rd\} \land\\ \qquad glvars \subseteq Initialised\_Own\_Vars_{\delta_i} \land\\ \qquad etyp\ compat_\delta\ rtype \bullet\\ \quad \delta \vdash_{Inh} inherit \Longrightarrow \delta_i\\ \quad \delta_i \vdash_{With} with \Longrightarrow \delta_w\\ \quad \delta_w \vdash_{Glob} globals \Longrightarrow glvars\\ \quad \delta_f \vdash_{Form} formals \Longrightarrow ftypes, fmodes, fpos\\ \quad \delta_e \vdash_{Rens} renames \Longrightarrow \delta_r\\ \quad \delta_r \vdash_{SBasic} bdecl \Longrightarrow \delta_b\\ \quad \delta_b \vdash_{SLater} ldecl \Longrightarrow \delta_l\\ \quad \delta_l \vdash_{Stmt} stmt\\ \quad \delta_l \vdash_{Exp} return : etyp\end{array}}{\delta \vdash_{Unit} fmain(\theta FMainUnit) \Longrightarrow \delta_m} \quad \text{(FMain)}$$

where

$$\delta_f == \delta_w\ [usedids := \delta.usedids \cup \{mid\}]$$

$$\begin{array}{l}\delta_e == \delta_f\ [\ usedids := \delta_x.usedids \cup \operatorname{dom} ftypes,\\ \qquad\qquad varmodes := \delta_f.varmodes \cup fmodes,\\ \qquad\qquad vartypes := \delta_f.vartypes \cup ftypes,\\ \qquad\qquad bodies\_req := \varnothing,\\ \qquad\qquad scope := \langle mid \rangle\ ]\downarrow_X \varnothing)\downarrow_S\end{array}$$

$$\delta_m == \delta\ [main := \{mid\}]$$

References: *FormalParam* p. 187; *GlobalAnnot* p. 190; *Ren* p. 227; *SBasic* p. 131; *SLater* p. 135; *Stmt* p. 153; *Exp* p. 71; *Env* p. 12; *Fun* p. 9; *ExpType* p. 73; $Library\_Unit\_Ids_\delta$ p. 245; $Initialised\_Own\_Vars_\delta$ p. 253; $\vdash_{Inh}$ p. 242; $\vdash_{With}$ p. 243; $\vdash_{Glob}$ p. 190; $\vdash_{Form}$ p. 187; $\vdash_{Rens}$ p. 228; $\vdash_{SBasic}$ p. 131; $\vdash_{SLater}$ p. 135; $\vdash_{Stmt}$ p. 154; $\vdash_{Exp}$ p. 71; $\downarrow_X$ p. 198; $\downarrow_S$ p. 199

17.8 SPARK Program

A SPARK program consists of one or more compilation units.

17.8.1 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.

$$SPARK == \mathbb{P}\ Unit$$

17.8.2 Static Semantics

1. A SPARK program is well-formed if there exists an order of the compilation units in which each compilation unit is well-formed in the environment created by its predecessors.

2. A SPARK program must be complete:

   (a) All the required package bodies must be declared.

   (b) All the subunit introduced by body stubs must be supplied.

   (c) A main program must be declared.

$$\frac{\begin{array}{l}\forall\, program : SPARK \\ \quad (\exists\, \delta_{0..\#program} : Env;\ units : \mathrm{iseq}\ Unit \mid \\ \qquad \mathrm{ran}\ units = program \land \\ \qquad \delta_0 = \theta StandardEnv \bullet \\ \qquad (\forall\, i : \mathrm{dom}\ units \bullet \\ \qquad\quad \delta_{i-1} \vdash_{Unit} units\ i \Longrightarrow \delta_i) \land \\ \qquad \delta_{\#program}.bodies\_req = \varnothing \land \\ \qquad \delta_{\#program}.sep\_req = \varnothing \land \\ \qquad \delta_{\#program}.main \neq \varnothing)\end{array}}{\vdash_{SPARK}\ program} \qquad \text{(SPARK)}$$

References: *Env* p. 12; *StandardEnv* p. 271

Appendix A

The Predefined Environment

The predefined language environment of SPARK is made up of the packages STANDARD and ASCII. The contents of STANDARD, which does not include the DURATION type, or the predefined exceptions of Ada, are directly visible anywhere in a SPARK program.

The package ASCII is considered to be a predefined library unit in SPARK (rather than a package declared within the specification of the package STANDARD — this is not allowed in SPARK). The declarations of ASCII are visible *by selection* anywhere in a SPARK program.

The following sections describe the components of the predefined language environment in more detail.

A.1 Reserved Identifiers

The are a number of identfiers which cannot be used in a SPARK program, even though they do not have declarations in SPARK. The identifiers are:

$$
\begin{array}{l}
duration : Id \\
constraint\_error, numeric\_error, program\_error, storage\_error, \\
tasking\_error : Id \\
standard : Id
\end{array}
$$

Resids is the set of reserved identifiers.

$$
\begin{array}{l}
Resids : \mathbb{P}\ Id \\
\hline
Resids = \{standard, duration, constraint\_error, numeric\_error, \\
\qquad\qquad program\_error, storage\_error, tasking\_error\}
\end{array}
$$

A.2 Constants

Although there are no predefined constants in SPARK, the values of enumeration the literals of the predefined types (see Section A.3) are represented as constants in the environment.

A.2.1 Boolean Literals

The literals of the predefined boolean type are *true* and *false*.

$$
\begin{array}{l}
predefined\_bool\_convals : Id \pfun Val \\
predefined\_bool\_contypes : Id \pfun IdDot \\
\hline
predefined\_bool\_convals = \{\ false \mapsto intval\ 0, true \mapsto intval\ 1\ \} \\
predefined\_bool\_contypes = \{\ false \mapsto id\ boolean, true \mapsto id\ boolean\ \}
\end{array}
$$

References: *Val* p. 6; *false* p. 266; *intval* p. 6; *true* p. 266; *boolean* p. 265

A.2.2 Character Literals

In Ada the character literals are used as enumeration literals for the predefined *character* type; in SPARK enumeration literals are always identifiers, which are lexically distinct from character literals.

Although the enumeration literals of this type cannot be written in SPARK, their values may be required to evaluate static expressions. The values follow the standard ASCII character codes.

$$
\begin{array}{l}
predefined\_char\_convals : Id \pfun Val \\
\hline
predefined\_char\_convals = \{ \\
\quad \texttt{nul} \mapsto intval\ 0, \\
\qquad \vdots \\
\quad \texttt{'!'} \mapsto intval\ 33, \\
\qquad \vdots \\
\quad \texttt{'a'} \mapsto intval\ 97, \\
\qquad \vdots \\
\quad \texttt{del} \mapsto intval\ 127\ \}
\end{array}
$$

$$
\begin{array}{|l}
predefined\_char\_contypes : Id \nrightarrow IdDot \\
\hline
predefined\_char\_contypes = \{ \\
\quad \texttt{nul} \mapsto id\ character, \\
\qquad \vdots \\
\quad \texttt{'!'} \mapsto id\ character, \\
\qquad \vdots \\
\quad \texttt{'a'} \mapsto id\ character, \\
\qquad \vdots \\
\quad \texttt{del} \mapsto id\ character\ \}
\end{array}
$$

References: *Val* p. 6; *intval* p. 6; *character* p. 265

A.3 Types

The predefined types of SPARK are:

$$boolean, integer, float : Id$$
$$character, string : Id$$

The predefined environment contains only these types.

$$predefined\_types : Id \pfun TypCon$$
$$\begin{aligned} types = \{ & \\ & integer \mapsto intTStandard\_Integer, \\ & float \mapsto floatT\ 0\ Standard\_Float, \\ & boolean \mapsto enumT\ Standard\_Boolean, \\ & character \mapsto enumT\ Standard\_Character, \\ & string \mapsto uarrT\ \theta Standard\_String\ \} \end{aligned}$$

The following sections describe these standard types in more detail.
References: *TypCon* p. 15; *intT* p. 15; *floatT* p. 15; *enumT* p. 15; *uarrT* p. 15

A.3.1 The predefined type Integer

The type *integer* is one of the predefined integer types. The ranges of the predefined integer types are symmetric about zero, excepting an allowed extra negative value [AARM, §3.5.4, ¶7].

$$Standard\_Integer : IntTypCon$$
$$\exists\, minint, maxint : \mathbb{N} \mid minint = maxint \lor minint = maxint + 1 \bullet$$
$$Standard\_Integer = intval(\mid -minint \,..\, maxint \mid)$$

Ada provides two constants SYSTEM.MIN_INT and SYSTEM.MAX_INT. We do not use these in the definition of the range of the predefined integer for two reasons:

1. *SPARK does not define a package SYSTEM. The user is free to provide such a package specification, provided that the declarations can be written in SPARK.*

2. *The two constants define the smallest and largest elements of* any *of the predefined integer types, so they may not belong to the particular type identified as integer.*

References: *IntTypCon* p. 15; *intval* p. 6

A.3.2 The predefined type Float

Although [AARM, §3.5.7 ¶9] describes some properties of the standard floating point type, the number of digits and the range are not determined.

Standard_Float

FloatTypCon

References: *FloatTypCon* p. 15

A.3.3 The predefined type Boolean

The predefined boolean type is an enumeration type. The two enumeration literals are:

$$false, true : Id$$

The type is described by these enumeration literals.

$$Standard\_Boolean : EnumTypCon$$

$$Standard\_Boolean = \{\ enumval\ false, enumval\ true\ \}$$

References: *EnumTypCon* p. 15; *enumval* p. 6

A.3.4 The predefined type Character

The predefined character type is an enumeration type. In Ada the character literals are used as enumeration literals for this type; in SPARK enumeration literals are always identifiers, which are lexically distinct from character literals.

$$Standard\_Character : EnumTypCon$$

$$Standard\_Character == \{$$
$$enumval\ \texttt{nul}, \ldots, enumval\ '!', \ldots, enumval\ 'a', \ldots, enumval\ \texttt{del}\ \}$$

Note that in SPARK user-defined character types cannot be declared [SR, §3.5.2].
References: *EnumTypCon* p. 15; *enumval* p. 6

A.3.5 The predefined type String

Not yet complete

A.4 Operators

The predefined language environment includes operators on the predefined types and also on the universal types.

A.4.1 Operators on Predefined Types

The following table summarises the operators on the predefined types.

Type	Operators
Integer	The standard operators for any integer type are defined for the predefined Integer (these are the relational and arithmetic operators). The functions *intuops* and *intbops* return the required unary and binary operators.
Float	The standard operators of any floating point type are defined for the predefined Float (these are the relational and arithmetic operators, excluding the the equality operators). The functions *fltuops* and *fltbops* return the required unary and binary operators.
Boolean	The logical operators are defined for the boolean type; however, of the relational operators, SPARK only provides equality and its complement on the boolean type [SR 4.5.2]. The functions *loguops* and *logbops* return the required unary and binary operators.
Character	The standard operators of an enumerated type are defined for Character (these are the relational operators). The function *enumbops* returns the required binary operators.
String	*Not yet complete*

$$
\begin{array}{|l}
predefined\_unary : Uop \times BasicType \pfun BasicType \\
\hline
predefined\_unary = \\
\quad loguops\ (nameT\ (id\ boolean)) \cup intuops\ (nameT\ (id\ integer)) \cup \\
\quad fltuops\ (nameT\ (id\ float))
\end{array}
$$

$$
\begin{array}{|l}
predefined\_binary : BasicType \times Bop \times BasicType \pfun BasicType \\
\hline
predefined\_binary = \\
\quad logbops\ (nameT\ (id\ boolean)) \cup intbops\ (nameT\ (id\ integer)) \cup \\
\quad fltbops\ (nameT\ (id\ float)) \cup enumbops\ (nameT\ (id\ character))
\end{array}
$$

References: *BasicType* p. 73; *Uop* p. 100; *Bop* p. 102; *loguops* p. 17; *nameT* p. 73; *boolean* p. 265; *intbops* p. 19; *integer* p. 265; *fltuops* p. 20; *float* p. 265; *logbops* p. 18; *intbops* p. 19; *fltbops* p. 20; *enumbops* p. 17; *character* p. 265

A.4.2 Operators on Universal Types

The universal integer and universal real types have the operators declared for integer and floating point types.

In addition, the following multiplication, division and expontentiation operators exist:

$$SpecialUnivUops : Uop \times BasicType \nrightarrow BasicType$$

$$SpecialUnivUops = intuops\ uintT \cup \mathit{fltuops}\ urealT$$

$$SpecialUnivBops : BasicType \times Bop \times BasicType \nrightarrow BasicType$$

$$\begin{aligned} SpecialUnivBops = {} & intbops\ uintT \cup \mathit{fltbops}\ urealT \cup \\ & \{\ (urealT, mul, uintT) \mapsto urealT, \\ & (uintT, mul, urealT) \mapsto urealT, \\ & (urealT, div, uintT) \mapsto urealT, \\ & (urealT, power, uintT) \mapsto urealT\ \} \end{aligned}$$

References: *BasicType* p. 73; *Uop* p. 100; *Bop* p. 102; *intuops* p. 18; *uintT* p. 73; *fltuops* p. 20; *urealT* p. 73

A.5 Subtypes

The predefined subtypes of SPARK are *natural* and *positive*.

$$natural, positive : Id$$

These subtypes are subtypes of the predefined integer type.

$$predefined\_subtypes : Id \pfun IdDot$$

$$\begin{aligned} &predefined\_subtypes = \{ \\ &\quad natural \mapsto id\ integer, \\ &\quad positive \mapsto id\ integer\ \} \end{aligned}$$

The subtype *natural* contains all the non-negative integer in the standard integer type, *positive* excludes zero.

$$Standard\_Natural : IntTypCon$$
$$Standard\_Positive : IntTypCon$$

$$Standard\_Natural = \{\ n : \mathbb{N} \mid intval\ n \in Standard\_Integer\ \}$$
$$Standard\_Natural = \{\ n : \mathbb{N} \mid n > 0 \land intval\ n \in Standard\_Integer\ \}$$

These type constructions are recorded in the same way as for a type.

$$predefined\_subtypes\_types : Id \pfun TypCon$$

$$\begin{aligned} &predefined\_subtypes\_types = \{ \\ &\quad natural \mapsto intT\ Standard\_Natural, \\ &\quad positive \mapsto intT\ Standard\_Positive\ \} \end{aligned}$$

References: integer p. 265; *IntTypCon* p. 15; *intval* p. 6; *Standard_Integer* p. 265; *intT* p. 15; *TypCon* p. 15

A.6 The Ascii Package

The identifier *ascii* identifies a package specification in the predefined language environment.

$$ascii : Id$$

Package Ascii declares constants of character type with values equal to the control characters, punctuation marks and lowercase letters. The complete set of constants identifiers is given in [AARM, Appendix C, ¶15].

$$nul, \ldots, us : Id$$
$$exclam, \ldots, tilde : Id$$
$$lc\_a, \ldots, lc\_z : Id$$

Since the character type is an enumeration, the value of these constants is an enumeration literal. However, these enumeration literals do not exist in SPARK (a full description of the predefined character type appears in Section A.3).

The complete visible environment of the *ascii* package is:

Ascii

SpecEnv

$$convals = \{$$
$$\quad nul \mapsto enumval\ \texttt{nul}, \ldots, exclam \mapsto enumval\ \texttt{'!'}, \ldots,$$
$$\quad lc\_a \mapsto enumval\ \texttt{'a'}, \ldots, lc\_z \mapsto enumval\ \texttt{'z'}\ \}$$
$$contypes = \{$$
$$\quad nul \mapsto id\ character, \ldots, exclam \mapsto id\ character, \ldots,$$
$$\quad lc\_a \mapsto id\ character, \ldots, lc\_z \mapsto id\ character\ \}$$
$$usedids = \{nul, \ldots, exclam, \ldots, lc\_a, \ldots lc\_z\ \}$$
$$defcons = \varnothing \wedge types = \varnothing$$
$$subtypes = \varnothing \wedge unops = \varnothing$$
$$binops = \varnothing \wedge vartypes = \varnothing$$
$$varmodes = \varnothing \wedge ownvars = \varnothing$$
$$pdecls = \varnothing \wedge fdecls = \varnothing$$
$$sdef = \varnothing \wedge scall = \varnothing$$

References: SpecEnv p. 10; *enumval* p. 6; *character* p. 265

A.7 The Complete Predefined Environment

The different elements of SPARK's package Standard form a environment *StandardEnv*.

StandardEnv

$$
\begin{array}{|l}
\hline
Env \\
\hline
usedids = Resids \cup \{\ true, false, boolean, integer, float, \\
\qquad\qquad character, string, natural, positive, ascii\ \} \\
contypes = predefined\_bool\_contypes \cup predefined\_char\_contypes \\
convals = predefined\_bool\_convals \cup predefined\_char\_convals \\
defcons = \varnothing \\
types = predefined\_types \cup predefined\_subtypes\_types \\
subtypes = predefined\_subtypes \\
unops = predefined\_unary \cup SpecialUnivUops \\
binops = predefined\_binary \cup SpecialUnivBops \\
vartypes = \varnothing \\
varmodes = \varnothing \\
ownvars = \varnothing \\
pdecls = \varnothing \\
fdecls = \varnothing \\
sdef = \varnothing \\
scall = \varnothing \\
withs = \{ascii\} \\
inherits = \{ascii\} \\
scope = \langle\rangle \\
paks = \{\ ascii \mapsto \theta EmptyPak\ [\ vis := \theta Ascii\ ]\ \} \\
bodies\_req := \varnothing \\
subunits = \varnothing \\
sep\_req = \varnothing \\
main = \varnothing \\
\hline
\end{array}
$$

In the above, the schema *EmptyPak* is used; this may be defined by:

EmptyPak

$$
\begin{array}{|l}
\hline
Pak \\
\hline
vis, priv : EmptySpecEnv \\
EmptyContext \\
\hline
\end{array}
$$

EmptyContext

Context

$$
\begin{aligned}
&withs = \varnothing \\
&inherits = \varnothing \\
&scope = \langle\rangle
\end{aligned}
$$

$$
EmptySpecEnv \mathrel{\widehat=} EmptyIdEnv \land EmptyConstEnv \land EmptyTypeEnv \land EmptyOpEnv \land EmptyVarEnv \land EmptySubEnv
$$

EmptyIdEnv

IdEnv

$$usedids = \varnothing$$

EmptyConstEnv

ConstEnv

$$
\begin{aligned}
&contypes = \varnothing \\
&convals = \varnothing \\
&defcons = \varnothing
\end{aligned}
$$

EmptyTypeEnv

TypeEnv

$$
\begin{aligned}
&types = \varnothing \\
&subtypes = \varnothing
\end{aligned}
$$

EmptyOpEnv

OpEnv

$$
\begin{aligned}
&unops = \varnothing \\
&binops = \varnothing
\end{aligned}
$$

EmptyVarEnv

VarEnv

$$
\begin{aligned}
&vartypes = \varnothing \\
&varmodes = \varnothing \\
&ownvars = \varnothing \\
&initowns = \varnothing
\end{aligned}
$$

EmptySubEnv

SubEnv

$pdecls = \varnothing$
$fdecls = \varnothing$
$scall = \varnothing$
$sdef = \varnothing$

References: Resids p. 262; *true* p. 266; *false* p. 266; *boolean* p. 265; *integer* p. 265; *float* p. 265; *character* p. 265; *string* p. 265; *natural* p. 269; *positive* p. 269; *predefined_bool_contypes* p. 263; *predefined_char_contypes* p. 264; *predefined_bool_convals* p. 263; *predefined_char_convals* p. 263; *ascii* p. 270; *predefined_types* p. 265; *predefined_subtypes_types* p. 269; *predefined_subtypes* p. 269; *predefined_unary* p. 267; *SpecialUnivUops* p. 268; *predefined_binary* p. 267; *SpecialUnivBops* p. 268; *Ascii* p. 270

Appendix B

Auxiliary Functions

This chapter contains the definitions of various functions and predicates used elsewhere. The definitions are organised into the following categories:

B.1 Preliminary Definitions

Many of the auxiliary functions are used to look-up properties of names (*IdDot*) in the environment. There are two fundamental steps required in this process.

1. If the name is inherited from another package (i.e. has the form k.i, where k is package identifier), then the required property is determined by looking in the visible environment of the package k.

2. If the look-up was in the visible environment of a package k and the return value of the look-up function is an *IdDot* i which is local to k, the correct return value in the context of the look-up is k.i, i.e. the *IdDot* contained in the visible environment of k must be *prefixed* by the package identifier k.

B.1.1 Prefixing

The function *prefix_id k prefixes* an element of *IdDot* which a package identifier *k*.

$$prefix\_id : Id \to IdDot \to IdDot$$

$$\forall i, j, k : Id \bullet$$
$$prefix\_id\, k(id\ i) = dot\ (k, i) \land$$
$$prefix\_id\, k(dot\ (j, i)) = dot\ (j, i)$$

The function *prefix_val* *prefixes* a value; this is required when the value is an enumeration literal.

$$prefix\_val : Id \to Val \nrightarrow Val$$

$$\forall k : Id;\ v : Val \bullet$$
$$v \in \text{ran}\ enumval \Rightarrow$$
$$prefix\_val\, k\, v = enumval\ \ prefix\_id\, k(enumval^{\sim} v) \land$$
$$v \notin \text{ran}\ enumval \Rightarrow$$
$$prefix\_val\, k\, v = v$$

References: *Val* p. 6

B.1.2 Visible Environment of a Package

The function *vis_env_of* returns the visible environment of a package.

$$
\begin{array}{|l}
vis\_env\_of \ : Env \to Id \nrightarrow Env \\
\hline
\forall\, \delta : Env;\ k : Id \mid k \in \operatorname{dom} \delta.paks \bullet \\
\quad vis\_env\_of\ \delta\ k = (\mu\, Env \mid \\
\qquad usedids = (\delta.paks\ k).vis.usedids \land \\
\qquad contypes = (\delta.paks\ k).vis.contypes \land \\
\qquad convals = (\delta.paks\ k).vis.convals \land \\
\qquad defcons = (\delta.paks\ k).vis.defcons \land \\
\qquad types = (\delta.paks\ k).vis.types \land \\
\qquad subtypes = (\delta.paks\ k).vis.subtypes \land \\
\qquad unops = (\delta.paks\ k).vis.unops \land \\
\qquad binops = (\delta.paks\ k).vis.binops \land \\
\qquad vartypes = (\delta.paks\ k).vis.vartypes \land \\
\qquad varmodes = (\delta.paks\ k).vis.varmodes \land \\
\qquad ownvars = (\delta.paks\ k).vis.ownvars \land \\
\qquad initowns = (\delta.paks\ k).vis.initowns \land \\
\qquad pdecls = (\delta.paks\ k).vis.pdecls \land \\
\qquad fdecls = (\delta.paks\ k).vis.fdecls \land \\
\qquad scall = (\delta.paks\ k).vis.scall \land \\
\qquad sdef = (\delta.paks\ k).vis.sdef \land \\
\qquad withs = \delta.withs \land \\
\qquad inherits = \delta.inherits \land \\
\qquad scope = \delta.scope \land \\
\qquad paks = \delta.paks \land \\
\qquad bodies\_req = \delta.bodies\_req \land \\
\qquad subunits = \delta.subunits \land \\
\qquad sep\_req = \delta.sep\_req \land \\
\qquad main = \delta.main) \\
\end{array}
$$

References: Env p. 12

B.1.3 Promotion

We can define a function *promote* which transforms a look-up function taking simple identifiers (*Id*) into one taking names (*IdDot*).

$$
\begin{array}{|l}
[X] \\
\hline
promote : (Env \to Id \nrightarrow X) \to (Env \to IdDot \nrightarrow X) \\
\hline
\forall\, i : Id, f : Env \to Id \nrightarrow X;\ \delta : Env \bullet \\
\quad (promote\ f)\ \delta\ (id\ i) = f\ \delta\ i \\
\forall\, k, i : Id, f : Env \to Id \nrightarrow X;\ \delta : Env \bullet \\
\quad (promote\ f)\ \delta\ dot(k, i) = \\
\qquad f\ (vis\_env\_of\ \delta\ k)\ i \\
\end{array}
$$

When the return type of the function is an *IdDot*, prefixing is required:

$$prefix\_promote : (Env \to Id \nrightarrow IdDot) \to (Env \to IdDot \nrightarrow IdDot)$$

$$\forall\, i : Id; f : Env \to Id \nrightarrow IdDot;\ \delta : Env \bullet$$
$$(prefix\_promote\ f)\ \delta\ (id\ i) = f\ \delta\ i$$

$$\forall\, k, i : Id; f : Env \to Id \nrightarrow IdDot;\ \delta : Env \bullet$$
$$(prefix\_promote\ f)\ \delta\ dot(k, i) =$$
$$prefix\_id\ k((promote\ f)\ \delta\ dot(k, i))$$

References: *Env* p. 12; *vis_env_of* p. 277; *prefix_id* p. 276

B.2 Constants

The definitions in this section look-up information about names (*IdDot*) identifying constants.

Visible Constants The function *is_visible_const* returns the set of visible constants.

$$
\begin{array}{|l}
is\_visible\_const_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_visible\_const_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad t \in \operatorname{dom} \delta.contypes \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad is\_visible\_const_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad k \in \delta.withs \land \\
\quad\quad is\_visible\_const_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Constant Type The function $const\_tmark_{\delta}$ returns the type of a constant.

$$
\begin{array}{|l}
const\_tmark_{Env} : Id \pfun IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad const\_tmark_{\delta}(id\ t) = \delta.contypes\ t \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad const\_tmark_{\delta}\ dot(k, t) = \\
\quad\quad prefix\_id\ k(const\_tmark_{vis\_env\_of\ \delta\ k}(id\ t))
\end{array}
$$

Constant Value The value of any scalar constant is recorded in the environment.

$$
\begin{array}{|l}
const\_val_{Env} : IdDot \pfun Val \\
\hline
\forall\, \delta : Env;\ c : Id \bullet \\
\quad const\_val_{\delta}(id\ c) = \delta.convals\ c \\
\forall\, \delta : Env;\ k, c : Id \bullet \\
\quad const\_val_{\delta}\ dot(k, c) = prefix\_val\ k\ (const\_val_{vis\_env\_of\ \delta\ k}(id\ c))
\end{array}
$$

References: Env p. 12; *vis_env_of* p. 277; *prefix_id* p. 276

B.3 Variables

The definitions in this section look-up information about names (*IdDot*) identifying variables.

Visible Variables The function *is_visible_var* returns the set of visibles.

$$
\begin{array}{|l}
is\_visible\_var_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_visible\_var_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad t \in \mathrm{dom}\, \delta.contypes \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad is\_visible\_var_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad k \in \delta.withs \land \\
\quad\quad is\_visible\_var_{vis\_env\_of\ \delta\, k}(id\ t)
\end{array}
$$

Variable Type The function var_tmark_{δ} returns the type of a variable.

$$
\begin{array}{|l}
var\_tmark_{Env} : Id \pfun IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad var\_tmark_{\delta}(id\ t) = \delta.vartypes\ t \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad var\_tmark_{\delta}\ dot(k, t) = \\
\quad\quad prefix\_id\ k(var\_tmark_{vis\_env\_of\ \delta\, k}(id\ t))
\end{array}
$$

Variable Access Modes The access modes used in the declaration of a variable are recorded in the environment.

$$
\begin{array}{|l}
var\_modes_{Env} : IdDot \pfun \mathbb{P}\ RdWr \\
\hline
\forall\, \delta : Env;\ c : Id \bullet \\
\quad var\_modes_{\delta}(id\ c) = \delta.varmodes(\!|c|\!) \\
\forall\, \delta : Env;\ k, c : Id \bullet \\
\quad var\_modes_{\delta}\ dot(k, c) = var\_modes_{vis\_env\_of\ \delta\, k}(id\ c)
\end{array}
$$

References: *Env* p. 12; *vis_env_of* p. 277; *prefix_id* p. 276

B.4 Types and Subtypes

The definitions in this section test the properties of names (*IdDot*) which identify types or subtypes.

B.4.1 Visible Types

The function *is_visible_tmark* returns the set of visible type marks. These are the type marks which can be used in a SPARK program.

$$
\begin{array}{|l}
is\_visible\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_visible\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad t \in \operatorname{dom} \delta.types \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad is\_visible\_tmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad k \in \delta.withs \land \\
\quad\quad is\_visible\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

References: Env p. 12; *vis_env_of* p. 277

B.4.2 Ancestor Type

All subtypes are derived, directly or indirectly, from a (full) type; the $ancestorof_{Env}$ function returns the name of this type. The function behaves as the identity when applied to a type mark which already identifies a (full) type.

$$
\begin{array}{|l}
ancestorof_{Env} : IdDot \pfun IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad (t \in (\operatorname{dom} \delta.types \setminus \operatorname{dom} \delta.subtypes) \Rightarrow \\
\quad\quad ancestorof_{\delta}(id\ t) = id\ t) \land \\
\quad (t \in \operatorname{dom} \delta.subtypes \Rightarrow \\
\quad\quad ancestorof_{\delta}(id\ t) = \delta.subtypes\ t) \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad ancestorof_{\delta}\ dot(k, t) = prefix\_id\ k\ ancestorof_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

References: Env p. 12; *prefix_id* p. 276; *vis_env_of* p. 277

B.4.3 Classification of Types

Types can are classified primarily according to the type constructor used in their declaration.

Integer The type mark identifies an integer type.

$$
\begin{array}{|l}
is\_int\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_int\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad \delta.types\ t \in \mathrm{ran}\ intT \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad is\_int\_tmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad is\_int\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Enumerated The type mark identifies an enumerated type.

$$
\begin{array}{|l}
is\_enum\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_enum\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad \delta.types\ t \in \mathrm{ran}\ enumT \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad is\_enum\_tmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad is\_enum\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Floating Point The type mark identifies a floating point type.

$$
\begin{array}{|l}
is\_float\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_float\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad \delta.types\ t \in \mathrm{ran}\ floatT \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad is\_float\_tmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad is\_float\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Fixed Point The type mark identifies a fixed point type.

$$
\begin{array}{|l}
is\_fixed\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_fixed\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad \delta.types\ t \in \operatorname{ran} fixedT \\
\forall\, \delta : Env;\ t, k : Id \bullet \\
\quad is\_fixed\_tmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad is\_fixed\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Array The type mark identifies a constrained or unconstrained array type.

$$
\begin{array}{|l}
is\_arr\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_arr\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad \delta.types\ t \in (\operatorname{ran} arrT \cup \operatorname{ran} uarrT) \\
\forall\, \delta : Env;\ t, k : Id \bullet \\
\quad is\_arr\_tmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad is\_arr\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Unconstrained Array The type mark identifies an unconstrained array type.

$$
\begin{array}{|l}
is\_unconstrained\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_unconstrained\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\quad\quad \delta.types\ t \in \operatorname{ran} uarrT \\
\forall\, \delta : Env;\ t, k : Id \bullet \\
\quad is\_unconstrained\_tmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\quad\quad is\_unconstrained\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Unconstrained Array Subtypes The type mark identifies a subtype of an unconstrained array type (so that objects declared using this subtype are constrained).

$$
\begin{array}{|l}
is\_unconstrained\_subtmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_unconstrained\_subtmark_{\delta}(id\ t) \Leftrightarrow \\
\qquad is\_arr\_tmark_{\delta}(id\ t) \land \\
\qquad is\_unconstrained\_tmark_{\delta}(ancestorof_{\delta}(id\ t)) \\
\forall\, \delta : Env;\ t, k : Id \bullet \\
\quad is\_unconstrained\_subtmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\qquad is\_unconstrained\_subtmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Record The type mark identifies a record type.

$$
\begin{array}{|l}
is\_rec\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_rec\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\qquad \delta.types\ t \in \operatorname{ran} recT \\
\forall\, \delta : Env;\ t, k : Id \bullet \\
\quad is\_rec\_tmark_{\delta}\ dot(k, t) \Leftrightarrow \\
\qquad is\_rec\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

References: *Env* p. 12; *intT* p. 15; *enumT* p. 15; *floatT* p. 15; *fixedT* p. 15; *arrT* p. 15; *uarrT* p. 15; *vis_env_of* p. 277; *ancestorof*$_\delta$ p. 281; *recT* p. 15

B.4.4 Additional Classification of Types

The following terms describe additional classes of types:

Scalar The scalar types are integer, real and enumeration types; values of these types have no components.

$$
\begin{array}{|l}
is\_scalar\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad (is\_scalar\_tmark_{\delta}) = \\
\qquad (is\_int\_tmark_{\delta}) \cup (is\_enum\_tmark_{\delta}) \cup \\
\qquad (is\_float\_tmark_{\delta}) \cup (is\_fixed\_tmark_{\delta})
\end{array}
$$

Composite Value of composite types consist of one or more components. The composite types of SPARK are arrays and records.

$$
\begin{array}{|l}
is\_composite\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad (is\_composite\_tmark_{\delta}) = \\
\qquad (is\_arr\_tmark_{\delta}) \cup (is\_rec\_tmark_{\delta})
\end{array}
$$

Discrete Integer and enumeration types are together known as discrete types. Arrays and case statements must be indexed by a discrete type.

$$
\begin{array}{|l}
is\_discrete\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad (is\_discrete\_tmark_{\delta}) = \\
\qquad (is\_int\_tmark_{\delta}) \cup (is\_enum\_tmark_{\delta})
\end{array}
$$

Real Floating and fixed point types, since they both represent rational numbers, are known as real types.

$$
\begin{array}{|l}
is\_real\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad (is\_real\_tmark_{\delta}) = \\
\qquad (is\_float\_tmark_{\delta}) \cup (is\_fixed\_tmark_{\delta})
\end{array}
$$

Numeric Real and integer types are together known as numeric types.

$$
\begin{array}{|l}
is\_numeric\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad (is\_numeric\_tmark_{\delta}) = \\
\qquad (is\_int\_tmark_{\delta}) \cup (is\_real\_tmark_{\delta})
\end{array}
$$

Limited Types declared as limited private or containing a component type declared in this way are known as limited types.

$$
\begin{array}{|l}
is\_limited\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_limited\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\qquad \delta.types\ t \in \operatorname{ran} limT \lor \\
\qquad (\exists\, a : ArrTypCon \bullet \\
\qquad\quad (\delta.types\ t = arrT\ a \lor \delta.types\ t = uarrT\ a) \land \\
\qquad\quad is\_limited\_tmark_{\delta}\ a.component) \lor \\
\qquad (\exists\, r : RecTypCon \bullet \\
\qquad\quad \delta.types\ t = recT\ r \land \\
\qquad\quad (\exists\, f : \operatorname{dom} r.types \bullet is\_limited\_tmark_{\delta}(r.types\ f))) \\
\forall\, \delta : Env;\ t, k : Id \bullet \\
\quad is\_limited\_tmark_{\delta}\ dot\ (k, t) \Leftrightarrow \\
\qquad is\_limited\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Incompletely Declared A directly-visible type declared as private or limited private, or a type with such a component, is considered incompletely declared until the full type declaration (in the private part of the package specification) has been encountered.

$$
\begin{array}{|l}
is\_incomplete\_tmark_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ t : Id \bullet \\
\quad is\_incomplete\_tmark_{\delta}(id\ t) \Leftrightarrow \\
\qquad \delta.types\ t \in (\operatorname{ran} privT \cup \operatorname{ran} limT) \lor \\
\qquad (\exists\, a : ArrTypCon \bullet \\
\qquad\quad (\delta.types\ t = arrT\ a \lor \delta.types\ t = uarrT\ a) \land \\
\qquad\quad is\_incomplete\_tmark_{\delta}\ a.component) \lor \\
\qquad (\exists\, r : RecTypCon \bullet \\
\qquad\quad \delta.types\ t = recT\ r \land \\
\qquad\quad (\exists\, f : \operatorname{dom} r.types \bullet is\_incomplete\_tmark_{\delta}(r.types\ f))) \\
\forall\, \delta : Env;\ t, k : Id \bullet \\
\quad lnot\ is\_incomplete\_tmark_{\delta}\ dot\ (k, t)
\end{array}
$$

References: *Env* p. 12; *is_int_tmark*$_\delta$ p. 282; *is_enum_tmark*$_\delta$ p. 282; *is_float_tmark*$_\delta$ p. 282; *is_fixed_tmark*$_\delta$ p. 283; *is_arr_tmark*$_\delta$ p. 283; *is_rec_tmark*$_\delta$ p. 284; *ArrTypCon* p. 16; *RecTypCon* p. 16

B.4.5 Floating Point Types

The number of digits (in the mantissa) of a floating point type can be determined from its type construction.

$$
\begin{array}{|l}
float\_tmark\_digits_{Env} : IdDot \pfun \mathbb{N} \\
\hline
\forall\, \delta : Env;\ t : Id \mid is\_float\_tmark_\delta(id\ t) \bullet \\
\quad float\_tmark\_digits_\delta(id\ t) = \\
\quad\quad (floatT^{\sim}(\delta.types\ t)).digits \\
\forall\, \delta : Env;\ k, t : Id \mid is\_float\_tmark_\delta\ dot(k, t) \bullet \\
\quad float\_tmark\_digits_\delta\ dot(k, t) = \\
\quad\quad float\_tmark\_digits_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

References: *Env* p. 12; $is\_float\_tmark_\delta$ p. 282; *floatT* p. 15; *vis_env_of* p. 277

B.4.6 Fixed Point Types

The accuracy of a fixed point type can be determined from its type construction.

$$
\begin{array}{|l}
fixed\_tmark\_delta_{Env} : IdDot \pfun \mathrm{Real} \\
\hline
\forall\, \delta : Env;\ t : Id \mid is\_fixed\_tmark_\delta(id\ t) \bullet \\
\quad fixed\_tmark\_delta_\delta(id\ t) = \\
\quad\quad (fixedT^{\sim}(\delta.types\ t)).delta \\
\forall\, \delta : Env;\ k, t : Id \mid is\_fixed\_tmark_\delta\ dot(k, t) \bullet \\
\quad fixed\_tmark\_delta_\delta\ dot(k, t) = \\
\quad\quad fixed\_tmark\_delta_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

References: *Env* p. 12; $is\_fixed\_tmark_\delta$ p. 283; *fixedT* p. 15; *vis_env_of* p. 277

B.4.7 Records

This section contains functions which return the number of component types of a record type (i.e. a type satisfying $is\_rec\_tmark_{Env}$).

Record Field Types by Name The type of each field in a record is determined from the record type constructor. The following function returns the type of each field identifier.

$$
\begin{array}{|l}
rec\_field\_tmark_{Env} : IdDot \pfun Id \pfun IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad \mathrm{dom}\ rec\_field\_tmark_\delta = (is\_rec\_tmark_\delta) \\
\forall\, \delta : Env;\ RecTypCon;\ t : Id \mid \\
\quad\quad \delta.types\ t = recT(\theta RecTypCon) \bullet \\
\quad rec\_field\_tmark_\delta(id\ t) = types \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad rec\_field\_tmark_\delta(dot(k, t)) = \\
\quad\quad (prefix\_id\ k) \circ rec\_field\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Record Field Types by Position The following function returns the type of each field number, in the order of declaration.

$$
\begin{array}{|l}
rec\_seq\_tmark_{Env} : IdDot \pfun \text{seq } IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad \text{dom } rec\_seq\_tmark_{\delta} = (is\_rec\_tmark_{\delta}) \\
\forall\, \delta : Env;\ RecTypCon;\ t : Id \mid \\
\qquad \delta.types\ t = recT(\theta RecTypCon) \bullet \\
\quad rec\_seq\_tmark_{\delta}(id\ t) = types \circ fields \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad rec\_seq\_tmark_{\delta}(dot(k,t)) = \\
\qquad (prefix\_id\ k) \circ rec\_seq\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

References: *Env* p. 12; *is_rec_tmark*$_{\delta}$ p. 284; *recT* p. 15; *RecTypCon* p. 16; *prefix_id* p. 276; *vis_env_of* p. 277

B.4.8 Arrays

This section contains functions which return the number of index types, the name of each index type and the component type of an array (i.e. a type satisfying $is\_arr\_tmark_{Env}$).

Array Arity The number of index types in an array type can be determined from the array type constructor:

$$
\begin{array}{|l}
array\_arity_{Env} : IdDot \pfun \mathbb{N}_1 \\
\hline
\forall\, \delta : Env \bullet \\
\quad \text{dom } array\_arity_{\delta} = (is\_arr\_tmark_{\delta}) \\
\forall\, \delta : Env;\ ArrTypCon;\ t : Id \mid \\
\qquad \delta.types\ t = arrT(\theta ArrTypCon) \bullet \\
\quad array\_arity_{\delta}(id\ t) = \#indexes \\
\forall\, \delta : Env;\ ArrTypCon;\ t : Id \mid \\
\qquad \delta.types\ t = uarrT(\theta ArrTypCon) \bullet \\
\quad array\_arity_{\delta}(id\ t) = \#indexes \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad array\_arity_{\delta}(dot(k,t)) = \\
\qquad array\_arity_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Array Component Type The component type of a constrained or unconstrained array (full) type is taken from the type constructor.

$$
\begin{array}{|l}
array\_comp\_tmark_{Env} : IdDot \pfun IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad \mathrm{dom}\ array\_comp\_tmark_{\delta} = (is\_arr\_tmark_{\delta}) \\
\forall\, \delta : Env;\ ArrTypCon;\ t : Id \mid \\
\qquad \delta.types\ t = arrT(\theta ArrTypCon) \bullet \\
\quad array\_comp\_tmark_{\delta}(id\ t) = component \\
\forall\, \delta : Env;\ ArrTypCon;\ t : Id \mid \\
\qquad \delta.types\ t = uarrT(\theta ArrTypCon) \bullet \\
\quad array\_comp\_tmark_{\delta}(id\ t) = component \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad array\_comp\_tmark_{\delta}(dot(k,t)) = \\
\qquad prefix\_id\ k\ array\_comp\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

Array Index Types The i'th index type mark of an array is determined from the array's type constructor.

$$
\begin{array}{|l}
array\_index\_tmark_{Env} : IdDot \pfun \mathbb{N}_1 \pfun IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad \mathrm{dom}\ array\_index\_tmark_{\delta} = (is\_arr\_tmark_{\delta}) \\
\forall\, \delta : Env;\ ArrTypCon;\ t : Id \mid \\
\qquad \delta.types\ t = arrT(\theta ArrTypCon) \bullet \\
\quad array\_index\_tmark_{\delta}(id\ t) = indexes \\
\forall\, \delta : Env;\ ArrTypCon;\ t : Id \mid \\
\qquad \delta.types\ t = uarrT(\theta ArrTypCon) \bullet \\
\quad array\_index\_tmark_{\delta}(id\ t) = indexes \\
\forall\, \delta : Env;\ k, t : Id \bullet \\
\quad array\_index\_tmark_{\delta}(dot(k,t)) = \\
\qquad prefix\_id\ k\ array\_index\_tmark_{vis\_env\_of\ \delta\ k}(id\ t)
\end{array}
$$

References: *Env* p. 12; $is\_arr\_tmark_{\delta}$ p. 283; *arrT* p. 15; *ArrTypCon* p. 16; *uarrT* p. 15; *prefix_id* p. 276; *vis_env_of* p. 277

B.4.9 Equality Types

The equality operator is not defined for limited types, or composite types which contain a component of a limited type. (An earlier SPARK restriction, on equality of floating point

types, has been removed from the language subset as a result of the Peer Review process: it was not particularly enforceable, given the presence of the other relational operators for these types.)

We define

$$
\begin{array}{|l}
is\_equality\_tmark_{Env\_} : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env \bullet \\
\quad is\_equality\_tmark_{\delta} = \\
\qquad is\_visible\_tmark_{\delta} \setminus is\_limited\_tmark_{\delta}
\end{array}
$$

References: *Env* p. 12; *is_visible_tmark*$_{\delta}$ p. 281; *is_limited_tmark*$_{\delta}$ p. 286

B.5 Functions

The definitions in this section look-up information about names ($IdDot$) identifying functions.

B.5.1 Callable Functions

The function $is\_callable\_fun_\delta$ returns the set of callable functions.

$$
\begin{array}{|l}
is\_callable\_fun_{Env}\ \_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ f : Id \bullet \\
\quad is\_callable\_fun_\delta(id\ f) \Leftrightarrow \\
\quad\quad f \in \mathrm{dom}\, \delta.fdecls \land \\
\quad\quad f \in \delta.scall \\
\forall\, \delta : Env;\ k, f : Id \bullet \\
\quad is\_callable\_fun_\delta\ dot(k, f) \Leftrightarrow \\
\quad\quad k \in \delta.withs \land \\
\quad\quad is\_callable\_fun_{vis\_env\_of\ \delta\ k}(id\ f)
\end{array}
$$

B.5.2 Formal Parameters

The function $fun\_param\_ids_\delta$ returns the identifier of the formal parameter at a given position in the list of formal parameters.

$$
\begin{array}{|l}
fun\_param\_ids_{Env} : IdDot \pfun \mathrm{seq}\ Id \\
\hline
fun\_param\_ids = promote \\
\quad (\lambda\, \delta : Env \bullet \\
\quad\quad (\lambda\, f : Id \mid f \in \mathrm{dom}\, \delta.fdecls \bullet (\delta.fdecls\ f).fpos))
\end{array}
$$

The function $fun\_param\_tmarks_\delta$ returns the type of a formal parameter of a function.

$$
\begin{array}{|l}
fun\_param\_tmarks_{Env} : IdDot \pfun Id \pfun IdDot \\
\hline
\forall\, \delta : Env;\ f : Id \bullet \\
\quad fun\_param\_tmarks_\delta(id\ f) = (\delta.fdecls\ f).ftypes \\
\forall\, \delta : Env;\ k, f : Id \bullet \\
\quad fun\_param\_tmarks_\delta(dot(k, f)) = \\
\quad\quad (prefix\_id\ k) \circ fun\_param\_tmarks_{vis\_env\_of\ \delta\ k}(id\ f)
\end{array}
$$

B.5.3 Return Type

The function $fun\_rtn\_tmark_{\delta}$ give the return type of a function.

$$fun\_rtn\_tmark_{Env} : IdDot \pfun IdDot$$

$$fun\_rtn\_tmark = prefix\_promote\ (\lambda\, \delta : Env \bullet (\lambda\, f : Id \mid f \in \mathrm{dom}\, \delta.fdecls \bullet (\delta.fdecls\ f).rtype))$$

B.5.4 Global Variables

The function fun_glvars_{δ} returns the set of global variables of a function.

$$fun\_glvars_{Env} : IdDot \pfun \mathbb{P}\, IdDot$$

$$\forall\, \delta : Env;\ f : Id \bullet fun\_glvars_{\delta}(id\ f) = (\delta.fdecls\ f).glvars$$

$$\forall\, \delta : Env;\ k, f : Id \bullet fun\_glvars_{\delta}(dot(k, f)) = (prefix\_id\ k)(\!|fun\_glvars_{vis\_env\_of\ \delta\ k}(id\ f)|\!)$$

References: *Env* p. 12; *prefix_id* p. 276; *promote* p. 278; *prefix_promote* p. 278; *vis_env_of* p. 277

B.6 Procedures

The definitions in this section look-up information about names (*IdDot*) identifying procedures.

B.6.1 Callable Procedures

The function $is\_callable\_proc_\delta$ returns the set of callable functions.

$$
\begin{array}{|l}
is\_callable\_proc_{Env}\_ : \mathbb{P}\ IdDot \\
\hline
\forall\, \delta : Env;\ p : Id \bullet \\
\quad is\_callable\_proc_\delta(id\ p) \Leftrightarrow \\
\qquad p \in \mathrm{dom}\, \delta.pdecls \land \\
\qquad p \in \delta.scall \\
\forall\, \delta : Env;\ k, p : Id \bullet \\
\quad is\_callable\_proc_\delta\, dot(k, p) \Leftrightarrow \\
\qquad k \in \delta.withs \land \\
\qquad is\_callable\_proc_{vis\_env\_of\ \delta\ k}(id\ p)
\end{array}
$$

B.6.2 Formal Parameters

The function $proc\_param\_ids_\delta$ returns the identifier of the formal parameter at a given position in the list of formal parameters.

$$
\begin{array}{|l}
proc\_param\_ids_{Env} : IdDot \pfun \mathrm{seq}\, Id \\
\hline
proc\_param\_ids = promote \\
\quad (\lambda\, \delta : Env \bullet \\
\qquad (\lambda\, p : Id \mid p \in \mathrm{dom}\, \delta.pdecls \bullet (\delta.pdecls\ p).fpos))
\end{array}
$$

The function $proc\_param\_tmarks_\delta$ returns the type of a formal parameter of a procedure.

$$
\begin{array}{|l}
proc\_param\_tmarks_{Env} : IdDot \pfun Id \pfun IdDot \\
\hline
\forall\, \delta : Env;\ p : Id \bullet \\
\quad proc\_param\_tmarks_\delta(id\ p) = (\delta.pdecls\ p).ftypes \\
\forall\, \delta : Env;\ k, p : Id \bullet \\
\quad proc\_param\_tmarks_\delta(dot(k, p)) = \\
\qquad (prefix\_id\ k) \circ proc\_param\_tmarks_{vis\_env\_of\ \delta\ k}(id\ p)
\end{array}
$$

The function $proc\_param\_modes_\delta$ returns the relation between formal parameters and the access modes implied by the parameter modes used in the declaration of a procedure.

$$proc\_param\_modes_{Env} : IdDot \pfun Id \rel RdWr$$

$$proc\_param\_modes = promote\ (\lambda\, \delta : Env \bullet (\lambda\, p : Id \mid p \in \dom \delta.pdecls \bullet (\delta.pdecls\ p).fmodes))$$

B.6.3 Global and Exported Variables

The function $proc\_glvars_\delta$ returns the set of global variables of a procedure.

$$proc\_glvars_{Env} : IdDot \pfun \mathbb{P}\, IdDot$$

$$\forall\, \delta : Env;\ p : Id \bullet proc\_glvars_\delta(id\ p) = (\delta.pdecls\ p).glvars$$

$$\forall\, \delta : Env;\ k, p : Id \bullet proc\_glvars_\delta(dot(k, p)) = (prefix\_id\ k)(\!| proc\_glvars_{vis\_env\_of\ \delta\ k}(id\ p) |\!)$$

The function $proc\_exvars_\delta$ returns the set of exported variables of a procedure.

$$proc\_exvars_{Env} : IdDot \pfun \mathbb{P}\, IdDot$$

$$\forall\, \delta : Env;\ p : Id \bullet proc\_exvars_\delta(id\ p) = (\delta.pdecls\ p).exvars$$

$$\forall\, \delta : Env;\ k, p : Id \bullet proc\_exvars_\delta(dot(k, p)) = (prefix\_id\ k)(\!| proc\_exvars_{vis\_env\_of\ \delta\ k}(id\ p) |\!)$$

References: *Env* p. 12; *prefix_id* p. 276; *promote* p. 278; *prefix_promote* p. 278; *vis_env_of* p. 277

B.7 Expressions

Expressions can also be classified by type.

Boolean Boolean is a named type predefined in package standard (see Section A.3). The type has special significance in the language, since expressions used to control if statements, for example, must be of *the* boolean type. An abbreviation makes this clearer:

$$booltype == valueT\ \{\ nameT\ (id\ boolean)\ \}$$

Integer An expression is a value of integer type if its type is a named type, which is an integer or universal integer type.

$$is\_int\_exp_{Env} \_ : \mathbb{P}\ ExpType$$

$$\forall\, \delta : Env \bullet$$
$$is\_int\_exp_\delta =$$
$$valueT(\!|\mathbb{P}_1(nameT(\!|(is\_int\_tmark_\delta)|\!) \cup \{\ uintT\ \})|\!)$$

Real An expression is a value of real type if its type is a named type, which is a real or universal real type.

$$is\_real\_exp_{Env} \_ : \mathbb{P}\ ExpType$$

$$\forall\, \delta : Env \bullet$$
$$is\_real\_exp_\delta =$$
$$valueT(\!|\mathbb{P}_1(nameT(\!|(is\_real\_tmark_\delta)|\!) \cup \{\ urealT\ \})|\!)$$

Numeric An expression is a value of a numeric type if it is an integer, universal integer, real, universal real or *universal fixed* type.

$$is\_numeric\_exp_{Env} \_ : \mathbb{P}\ ExpType$$

$$\forall\, \delta : Env \bullet$$
$$is\_numeric\_exp_\delta =$$
$$valueT(\!|\mathbb{P}_1(nameT(\!|(is\_numeric\_tmark_\delta)|\!) \cup \{\ uintT, urealT, ufixT\ \})|\!)$$

Notice that the numeric expressions include those of universal fixed type in addition to real and integer expressions.

References: *valueT* p. 73; *nameT* p. 73; *boolean* p. 265; *ExpType* p. 73; *Env* p. 12; $is\_int\_tmark_\delta$ p. 282; *uintT* p. 73; $is\_real\_tmark_\delta$ p. 285; *urealT* p. 73; $is\_numeric\_tmark_\delta$ p. 285; *ufixT* p. 73

B.8 Values

B.8.1 Range of Values in a Scalar Type

The set of values (from *Val*) which belong to a scalar type identified by a typemark is returned by the function *scalar_tmark_range*.

$$scalar\_tmark\_range_{Env} : IdDot \pfun \mathbb{P}\ Val$$

$$\begin{array}{l}
\forall\, \delta : Env;\ t : Id \mid is_scalar_tmark_\delta(id\ t) \bullet \\
\quad (is_int_tmark_\delta(id\ t) \Rightarrow \\
\qquad scalar_tmark_range_\delta\ t = intT^\sim(\delta.types\ t)) \land \\
\quad (is_enum_tmark_\delta(id\ t) \Rightarrow \\
\qquad scalar_tmark_range_\delta\ t = enumT^\sim(\delta.types\ t)) \land \\
\quad (is_float_tmark_\delta(id\ t) \Rightarrow \\
\qquad scalar_tmark_range_\delta\ t = (floatT^\sim(\delta.types\ t)).range) \land \\
\quad (is_fixed_tmark_\delta(id\ t) \Rightarrow \\
\qquad scalar_tmark_range_\delta\ t = (fixedT^\sim(\delta.types\ t)).range) \\
\\
\forall\, \delta : Env;\ t, k : Id \mid is_scalar_tmark_\delta\ dot(k, t) \bullet \\
\quad scalar_tmark_range_\delta\ dot(k, t) = \\
\qquad prefix_val\ k(\lvert (scalar_tmark_range_{vis_env_of\ \delta\ k}(id\ t))\rvert)
\end{array}$$

References: *Val* p. 6; *Env* p. 12; *is_scalar_tmark*$_\delta$ p. 284; *is_int_tmark*$_\delta$ p. 282; *intT* p. 15; *is_enum_tmark*$_\delta$ p. 282; *enumT* p. 15; *is_float_tmark*$_\delta$ p. 282; *floatT* p. 15; *is_fixed_tmark*$_\delta$ p. 283; *fixedT* p. 15; *prefix_id* p. 276; *vis_env_of* p. 277

B.8.2 Ordering Values

The scalar values can be ordered; the relation $_{V}{\leq}_{Env}$ specifies this ordering.

$$\_\ {}_V{\leq}_{Env}\ \_ : Val \leftrightarrow Val$$

$$\begin{array}{l}
\forall\, \delta : Env;\ i, j : \mathbb{Z};\ r, s : \mathrm{Real};\ l, m : IdDot;\ f : Id \pfun Val;\ a : \mathrm{seq}\ Val \bullet \\
\quad intval\ i\ {}_V{\leq}_\delta\ intval\ j \Leftrightarrow i \leq j \land \\
\quad realval\ r\ {}_V{\leq}_\delta\ realval\ s \Leftrightarrow r \leq s \land \\
\quad enumval\ l\ {}_V{\leq}_\delta\ enumval\ m \Leftrightarrow \\
\qquad const_val_\delta\ l \leq const_val_\delta\ m
\end{array}$$

References: *Env* p. 12; *Val* p. 6; *intval* p. 6; *realval* p. 6; *enumval* p. 6; *prefix_id* p. 276; *const_val*$_\delta$ p. 279

B.8.3 Ranges of Values

Ranges of values are required, for example, to describe the elements of a subtype. The function $_{V}..{}_{Env}$ takes all (ordered) pairs of values to the set containing all the elements of

Val in the interval between the first and second values of the pair.

$$\_\ {}_{V}\!\cdot\!\cdot_{Env}\ \_ : Val \times Val \pfun \mathbb{P}\ Val$$

$$\forall\, \delta : Env;\ u, v : Val \bullet \qquad u\ {}_{V}\!\cdot\!\cdot_{\delta}\ v = \{\ w : Val \mid u\ {}_{V}\!\leq_{\delta}\ w\ {}_{V}\!\leq_{\delta}\ v\ \}$$

References: *Env* p. 12; *Val* p. 6; ${}_{V}\!\leq_{\delta}$ p. 296

B.9 Evaluation of Constant and Static Expressions

All constants in SPARK are assigned values which can be determined statically at compile-time, as opposed to run-time (as is the case with full Ada).

In this document, we have used $\Longrightarrow_{\text{StaticEval}}$ to denote evaluation of static expressions and $\Longrightarrow_{\text{ConstEval}}$ for evaluation of expressions which may be assigned to constants. (The latter is a broader class of expressions: aggregate expressions may be assigned to a constant, yet an aggregate is not a static expression in the Ada sense.) These relations have the following signature:

$$\_ \vdash \_ \Longrightarrow_{\text{StaticEval}} \_ \subseteq Env \times Exp \times Val$$
$$\_ \vdash \_ \Longrightarrow_{\text{ConstEval}} \_ \subseteq Env \times Exp \times Val$$

We define them here by:

$$\frac{\forall\, c : \text{seq}\, Id;\ \delta : Env;\ e : Exp;\ v : Val \bullet \\ c, \delta', \varnothing \vdash_e e \Longrightarrow_e v}{\delta \vdash e \Longrightarrow_{\text{StaticEval}} v} \quad \text{(SEvl)}$$

$$\frac{\forall\, c : \text{seq}\, Id;\ \delta : Env;\ e : Exp;\ v : Val \bullet \\ c, \delta', \varnothing \vdash_e e \Longrightarrow_e v}{\delta \vdash e \Longrightarrow_{\text{ConstEval}} v} \quad \text{(CEvl)}$$

In the above rules, δ' is the *dynamic* environment which would correspond to the static environment δ at the same point in the source text, while c, the context used in the dynamic semantics, is the context for the current point in the source text where the constant is being declared. (In principle, Ada static real expressions are evaluated without loss of precision, accuracy being lost only at run-time when the declaration is elaborated and a value associated with the constant, for instance.)

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

T

U

V

W

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.

