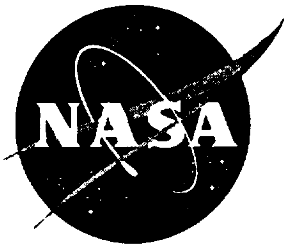
NASA Contractor Report 195037

47184
P. 158



Towards a Formal Semantics for Ada 9X

David Guaspari, John McHugh, Wolfgang Polak, and Mark Saaltink
Odyssey Research Associates, Inc., Ithaca, New York

(NASA-CR-195037) TOWARDS A FORMAL SEMANTICS FOR Ada 9X Final Report (Odyssey Research Associates) 158 p

N95-24634

Unclas

G3/62 0047184

Contract NAS1-18972

March 1995

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virigina 23681-0001

Contents

1 Introduction **1**

2 Method of Description **3**

2.1 External Effects and Nonterminating Programs 4
2.2 Semantic Simplifications 4
2.3 Static Checks and Overload Resolution 5
2.4 Environments, Entities, and Stores 6
2.5 Ordering 6
2.6 Types 7
2.7 Values 7
2.8 Objects 8
2.8.1 Actual Subtypes 8
2.8.2 Initialization 9
2.9 Aliasing 10

3 Semantic Domains **11**

3.1 Basic Notations 11
3.2 Entities and the Environment 12
3.3 Thunks 13
3.4 Values 13
3.4.1 Ranges 14
3.4.2 Index Ranges 14
3.4.3 Tags 14
3.4.4 Bindings 15
3.4.5 Values 15
3.5 Types and Subtypes 15
3.5.1 Subtypes and Constraints 16
3.5.2 Partial Subtypes 17
3.5.3 Derived Types and Classes 18
3.5.4 Scalar Types 19
3.5.5 Array Types 29
3.5.6 Discriminants 29
3.5.7 Record Types 30
3.5.8 Tagged Types and Type Extensions 31
3.5.9 Access Types 31
3.6 States 31

4 Judgements 32
4.1 Domains . . . 32
4.1.1 Types . . . 32
4.1.2 Values . . . 35
4.1.3 Subtypes . . . 36
4.1.4 Environments and Views . . . 38
4.1.5 Memory Model . . . 40
4.1.6 Other Predicates . . . 41
4.2 Judgements . . . 42
4.2.1 Declarations . . . 42
4.2.2 Parameter lists . . . 42
4.2.3 Type Definitions . . . 43
4.2.4 Variant Parts . . . 43
4.2.5 Discriminant Parts . . . 43
4.2.6 Component Lists . . . 43
4.2.7 Subtype Indications . . . 43
4.2.8 Statements . . . 44
4.2.9 Elsif Clauses . . . 44
4.2.10 Case Alternatives . . . 44
4.2.11 Discrete Choice Lists . . . 45
4.2.12 Expressions . . . 45
4.2.13 Names . . . 45
4.2.14 Ranges . . . 45
4.2.15 Record Aggregates . . . 46
4.2.16 Array Aggregates . . . 46
4.2.17 Attributes . . . 46
4.3 Actions . . . 46
4.4 State . . . 49
4.4.1 Classification of States . . . 49
4.4.2 Accessing the Store of a State . . . 49
4.4.3 Reading and Writing the Store . . . 50
4.4.4 The Content of a Location . . . 50
4.5 Order of Execution . . . 51
4.5.1 Sequential Execution . . . 51
4.5.2 Arbitrary Order Execution . . . 51
4.5.3 Executing Individual Actions . . . 52
4.6 Values . . . 53
4.6.1 Ranges . . . 53
4.6.2 Index Ranges . . . 54
4.6.3 Predicates of Values . . . 54
4.7 Types . . . 55
4.7.1 Type Descriptors . . . 55
4.7.2 Ancestry Relation . . . 55
4.7.3 Ranges of Scalar Types . . . 55
4.7.4 Values of a Type . . . 56
4.7.5 Record Fields . . . 57
4.7.6 Classification of Types . . . 57
4.8 Subtypes . . . 59
4.8.1 Constraint Satisfaction . . . 59
4.8.2 Values of a Subtype . . . 60
4.8.3 Actualization . . . 60

4.9 Declarations 61
4.9.1 Declarations 61
4.9.2 Types and Subtypes 61
4.9.3 Objects and Named Numbers 62
4.9.4 Derived Types and Classes 63
4.9.5 Scalar Types 64
4.9.6 Array Types 65
4.9.7 Discriminants 66
4.9.8 Record Types 66
4.9.9 Tagged Types and Type Extensions 67
4.9.10 Access Types 67
4.10 Expressions 68
4.10.1 Names 68
4.10.2 Literals 70
4.10.3 Aggregates 71
4.10.4 Operators and Expression Evaluation 72
4.10.5 Type Conversions 73
4.10.6 Qualified Expressions 74
4.10.7 Allocators 74
4.11 Statements 74
4.11.1 Statement Sequences 74
4.11.2 Assignment Statements 75
4.11.3 If Statements 75
4.11.4 Case Statements 76
4.11.5 Loop Statements 77
4.11.6 Block Statements 78
4.11.7 Exit Statements 78
4.12 Subprograms 79
4.12.1 Subprogram Declarations 79
4.12.2 Formal Parameter Modes 79
4.12.3 Subprogram Bodies 80
4.12.4 Subprogram Calls 81
4.12.5 Return Statements 82
4.13 Attributes 83

5 Exceptions and Optimization **84**
5.1 Introduction 84
5.2 The Ada 9X revision of 11.6 84
5.2.1 [11.6(5)] 84
5.2.2 [11.6(7)] 93
5.3 Living with the "Canonical Semantics" 94
5.3.1 Restricting the execution set size 95
5.3.2 Discovering the execution 96
5.4 Observations on the Reference Manual 96

6 Conclusions **98**
6.1 Implementation Freedoms 98
6.2 Notation and Tools 99
6.3 Bounded Errors 100
6.4 Structure of Models 100

Bibliography **101**

A Official comments submitted **102**

B Intermediate Syntax **104**
B.1 Syntactic Domains . . . 104
B.1.1 Component Associations (*Aca*) . . . 104
B.1.2 Aggregates (*Agg*) . . . 104
B.1.3 Case Alternatives (*Alt*) . . . 105
B.1.4 Choice Lists (*Ccl*) . . . 105
B.1.5 Context Items (*Cit*) . . . 105
B.1.6 Component Declarations (*Cmp*) . . . 105
B.1.7 Compilation Units (*Cmp*) . . . 105
B.1.8 Conditions (*Cnd*) . . . 105
B.1.9 Constraints (*Cns*) . . . 105
B.1.10 Discriminant Associations (*Dca*) . . . 105
B.1.11 Discrete Choices (*Dch*) . . . 105
B.1.12 Declarations (*Dcl*) . . . 106
B.1.13 Discriminant Parts (*Dcp*) . . . 106
B.1.14 Discriminant Specifications (*Dcs*) . . . 106
B.1.15 Exception Choices (*Ech*) . . . 106
B.1.16 Else-If Clauses (*Eif*) . . . 107
B.1.17 Expressions (*Exp*) . . . 107
B.1.18 Modes (*Mde*) . . . 107
B.1.19 Names (*Nam*) . . . 107
B.1.20 Parameter Specifications (*Pms*) . . . 108
B.1.21 Pragmas (*Prg*) . . . 108
B.1.22 Parameter Associations (*Pss*) . . . 108
B.1.23 Record Component Associations (*Rca*) . . . 108
B.1.24 Ranges (*Rng*) . . . 108
B.1.25 Subtype Indications (*Sid*) . . . 108
B.1.26 Subprogram Specifications (*Sps*) . . . 108
B.1.27 Statements (*Stm*) . . . 108
B.1.28 Type Definitions (*Tdf*) . . . 109
B.1.29 Variants (*Vnt*) . . . 110
B.1.30 Variant Parts (*Vrp*) . . . 110
B.1.31 Exception Choices (*Xhd*) . . . 110
B.2 Lexical Elements . . . 110
B.3 Declarations and Types . . . 110
B.3.1 Declarations . . . 110
B.3.2 Types and Subtypes . . . 111
B.3.3 Objects and Named Numbers . . . 112
B.3.4 Derived Types and Classes . . . 113
B.3.5 Scalar Types . . . 113
B.3.6 Array Types . . . 115
B.3.7 Discriminants . . . 116
B.3.8 Record Types . . . 117
B.3.9 Tagged Types and Type Extensions . . . 118
B.3.10 Access Types . . . 119
B.3.11 Declarative Parts . . . 119
B.4 Names and Expressions . . . 120

B.4.1 Names . . . 120
B.4.2 Literals . . . 122
B.4.3 Aggregates . . . 122
B.4.4 Expressions . . . 124
B.4.5 Operators and Expression Evaluation . . . 125
B.4.6 Type Conversions . . . 126
B.4.7 Qualified Expressions . . . 126
B.4.8 Allocators . . . 126
B.4.9 Static Expressions and Static Subtypes . . . 127
B.5 Statements . . . 127
B.5.1 Simple and Compound Statements – Sequences of Statements . . . 127
B.5.2 Assignment Statement . . . 128
B.5.3 If Statements . . . 128
B.5.4 Case Statements . . . 128
B.5.5 Loop Statements . . . 129
B.5.6 Block Statements . . . 129
B.5.7 Exit Statements . . . 130
B.5.8 Goto Statements . . . 130
B.6 Subprograms . . . 130
B.6.1 Subprogram Declarations . . . 130
B.6.2 Formal Parameter Modes . . . 132
B.6.3 Subprogram Bodies . . . 132
B.6.4 Subprogram Calls . . . 132
B.6.5 Return Statements . . . 133
B.6.6 Overloading of Operators . . . 133
B.7 Packages . . . 133
B.7.1 Package Specifications and Declarations . . . 133
B.7.2 Package Bodies . . . 133
B.7.3 Private Type and Private Extensions . . . 133
B.7.4 Deferred Constants . . . 134
B.7.5 Limited Types . . . 134
B.7.6 User-Defined Assignment and Finalization . . . 134
B.8 Visibility Rules . . . 134
B.8.1 Declarative Region . . . 134
B.8.2 Scope of Declarations . . . 134
B.8.3 Visibility . . . 134
B.8.4 Use Clauses . . . 134
B.8.5 Renaming Declarations . . . 134
B.8.6 The Context of Overload Resolution . . . 135
B.9 Tasks and Synchronization . . . 135
B.9.1 Task Units and Task Objects . . . 135
B.9.2 Task Execution – Task Activation . . . 136
B.9.3 Task Dependence – Termination of Tasks . . . 136
B.9.4 Protected Units and Protected Objects . . . 136
B.9.5 Intertask Communication . . . 137
B.9.6 Delay Statements, Duration, and Time . . . 137
B.9.7 Select Statements . . . 138
B.9.8 Selective Accept . . . 138
B.9.9 Timed Entry Calls . . . 138
B.9.10 Conditional Entry Calls . . . 139
B.9.11 Asynchronous Transfer of Control . . . 139

B.9.12 Abort of a Task – Abort of a Sequence of Statements 139
B.9.13 Task and Entry Attributes 140
B.9.14 Shared Variables 140
B.9.15 Example of Tasking and Synchronization 140
B.10 Program Structure and Compilation Issues 140
B.10.1 Separate Compilation 140
B.10.2 Program Execution 142
B.11 Exceptions 142
B.11.1 Exception Declarations 142
B.11.2 Exception Handlers 142
B.11.3 Raise Statements 143
B.11.4 Exception Handling 143
B.11.5 Suppressing Checks 143
B.11.6 Exceptions and Optimization 143
B.12 Generic Units 143
B.12.1 Generic Declarations 143
B.12.2 Generic Bodies 143
B.12.3 Generic Instantiation 143
B.12.4 Formal Objects 144
B.12.5 Formal Types 144
B.12.6 Formal Subprograms 145
B.12.7 Formal Packages 146
B.12.8 Example of a Generic Package 146
B.13 Representation Clauses and Implementation–Dependent Features 146
B.14 Ada 9X Input–Output 147

Index **148**

Chapter 1

Introduction

The Ada 9X Language Precision Team (LPT) was formed in 1990 to study portions of the design of Ada 9X from a mathematical perspective. The first LPT project studied small parts of the language in isolation, formulating fairly simple models to explore the ramifications of the design. The idea was to avoid spending time studying the conventional parts of the language (where, we felt, little would be gained by the analysis) and to focus on the novel proposals such as the object-oriented features, overload resolution, and exception mechanisms. The results of this first project appear in two reports [9, 2].

The second LPT project had two separate goals. The first, similar in approach to the first project, was to study the rules of allowed optimizations. The second goal was rather different. Instead of defining many unrelated small models, and studying new features in depth, the plan was to formulate a broad model to cover a large part of the language. In this way, we hoped to find problems arising from the interactions between different features.

The level of effort of the project was clearly insufficient to define a complete model of the language, and our plan was not to make a complete model. Instead, we planned to define the framework for such a model, and to fill in the details of the framework only in certain areas. Furthermore, the framework was not intended to be complete; we did not expect to be able to describe concurrency or distributed programming. We did, however, hope for the framework to cover most of the sequential features.

Our expectation was not that the resulting incomplete model would necessarily be useful to anyone (although we hope that it could be extended to form the basis for analysis tools). Instead, we expected that the activity of making this model would allow us to influence the design of Ada 9X by identifying problems with specific language features or with the interactions between different features, or by suggesting improvements in the ways that parts of the language are described. As the design of Ada 9X was nearing completion when our project began, it was important to move as quickly as possible. So, the model is quite sketchy in areas. Moreover, the formal definition presented in this report omits several partial models that we built in the course of the project, as we did not have the time to integrate them with the overall framework. None the less, these models have played a useful role in helping us to understand and comment on various language features, and to influence (however slightly) the design of Ada 9X.

This report is organized as follows. The method used for defining the model is described in Chapter 2. A discussion of the model also appears there. The model itself appears in two parts. The "domains" (sets describing the values assumed by different entities in the language) are described in Chapter 3. The "judgements" describing possible executions of a program are described in Chapter 4. Our study of the optimization freedoms described in [11.6] does not fit into the main framework; it is described separately in Chapter 5. Conclusions are drawn in Chapter 6.

Two appendices give some additional information. Appendix A lists the official language com-

ments that were submitted by the LPT; most of these apply to Version 4.0 of the Draft Standard [3], and several have resulted in changes that appear in Version 5.0 [4]. Appendix B describes the intermediate syntax that results from overload resolution (and other static checks).

References to the Ada 83 Reference Manual [10] appear in the form [RM-83 $c.s(p)$], where c is the chapter number, s the section number, and p the paragraph number. References to version 5.0 of the Annotated Draft Standard for Ada 9X [4] appear in the form [$c.s(p)$]; this standard as a whole is often referred to as "the Reference Manual".

Chapter 2

Method of Description

The "broad semantic framework" is defined using *Natural Semantics* [5]. The general idea of natural semantics is to define semantics as one or more relations between syntactic phrases and semantic values. These relations are defined using axioms and rules of inference. As a trivial example, a simple language of arithmetic expressions can be defined by the grammar

```
expression  ::=  numeral | <expression> "plus" <expression>
```

We can define the semantics of these expressions using a relation (between expressions and numbers) of "evaluates to", which we will write as *expression* $\Rightarrow$ *value*, and a relation (between numerals and numbers) of "denotes", which we will write as *numeral* $\longrightarrow$ *value*.

Two rules suffice to define the semantics of expressions (although additional rules not shown here are needed to define the "denotes" relation). The first covers the base case of numerals:

$$\frac{n \longrightarrow v}{n \Rightarrow v} \quad [\, n \in numeral \,]$$

The second gives the semantics of sums:

$$\frac{e1 \Rightarrow v1 \qquad e2 \Rightarrow v2}{e1 \text{ plus } e2 \Rightarrow v1 + v2}$$

The method is easily able to handle nondeterminism, where a phrase can have many possible results. If we extend the expression grammar above to include ranges, with the meaning that any number in the range is allowed, we have the grammar

```
expression  ::=  numeral
              |  <expression> "plus" <expression>
              |  <expression> ".." <expression>
```

Only one additional semantic rule is needed:

$$\frac{e1 \Rightarrow v1 \qquad e2 \Rightarrow v2}{e1 \,..\, e2 \Rightarrow v} \quad [\, v1 \leq v \leq v2 \,]$$

Using these rules, we can deduce

$$(0 \,..\, 2) \text{ plus } (10 \,..\, 20) \Rightarrow x$$

for any x between 10 and 22 inclusive.

In practical applications of Natural Semantics, the judgements used are often more complex than in this simple example. Usually, there is a certain amount of contextual information (such as the definitions of functions or procedures); this is represented by an *environment*. Often a *store* is used to record the values of variables. Furthermore, the evaluation of a phrase may have an effect on the environment or store. So the judgements often have many components, and there are a number of auxiliary domains of semantic values. We write most judgements in a standardized form

$$S, E \vdash_{pt} p \Rightarrow v, \ldots,$$

where S is a store, E is an environment, p is a phrase, the subscript pt gives the kind of phrase (*e.g.,* whether p is a declaration, statement, expression, or name), v is a possible result of executing (or evaluating or elaborating) p, and the "..." are any other results that the execution may have. Usually, there is a final state that reflects any side-effects that the execution may have had.

We describe the domains of semantic values using the Z notation [8], which furnishes a standard toolkit of notations for sets, functions, relations, and "freely constructed" sets.

The Natural Semantics definitions have been written in a machine-readable form in Prolog. Judgements are represented directly as Prolog predicates, and semantic rules as Prolog rules. There is a slight difficulty in transcribing uses of functions, but a simple translation to relations is possible. The Prolog representation has two main advantages. Firstly, we were able to apply a type-checking package developed by Reddy and Lakshman [7] to the definition; this found a number of simple errors in the rules. Secondly, we are able to run the Prolog and determine what outcomes are predicted by the semantic definition. This allows the semantics definition to be tested on small examples.

Several aspects of Ada 9X are tricky to define properly. In the remainder of this chapter, some of the awkward parts of the language and some of the more intricate aspects of the Natural Semantics definition will be explained. Not everything described below has been implemented in the Natural Semantics; some of the discussion describes our plan for dealing with a feature even though we did not have time to include that feature in the semantics definitions.

2.1 External Effects and Nonterminating Programs

Ultimately, the meaning of an Ada program is defined in terms of its sequence of "external effects", as described in [1.1.3(8)]. We can readily define several types of external effects, such as operations on files using the standard I/O packages, propagation of an exception, or return from the main program. Other effects are not covered.

An outside observer can see these external effects during the execution of a program, and does not need to wait until (and if) the program terminates. Therefore, we use a small trick to allow the semantics to assert that a certain sequence of external effects can be viewed whether or not the program terminates. We use a special *incomplete* condition that is treated like an exception that cannot be handled. For any operation having an external effect, one possible result is to "raise" this condition. The semantic definition then propagates this condition out of the main call. Thus, we are able to infer judgements of the form

$$\mathit{Library} \vdash \mathit{program_name} \Rightarrow e,$$

where e is a sequence of external effects, for every sequence e that might be observed during a run of the program.

2.2 Semantic Simplifications

Some language features appear to be very difficult to incorporate into this model. For these features, we have introduced a notion of an *unpredicted* result. When our definition allows the deduction that

unpredicted is a possible result of an execution, it means that the particular program includes a language feature, or encounters a situation, that we decided not to account for in our model. This is similar to erroneous executions, where the language standard does not predict the results of a program.

For example, Section [11.6] of the Ada 9X Reference Manuals allows implementations to produce results at variance with the language rules described elsewhere in the manual (in situations where a language-defined check would fail if those rules were followed). The freedoms allowed by Section [11.6] appear to be very difficult to incorporate into the model defined here. Therefore, we have kept the model simple by treating the failure of a language-defined check as an *unpredicted* execution.

There are some rules new to Ada 9X that constrain the result of an execution that Ada 83 classes as erroneous. These *bounded error* situations can be difficult to model. For example, reading the value of an uninitialized scalar variable is erroneous in Ada 83. In Ada 9X, it is a bounded error, which can result in an exception or an implementation-defined result. Version 4.0 of the proposed Standard introduced the concept of *invalid values* to describe these results. Unfortunately, the introduction of invalid values complicates the semantics of the language considerably, as it is necessary to provide rules for computing with these values. The draft standard does not always provide the complete details of these rules. For example, what is the result of a comparison involving invalid values? Are the ordering operators transitive, even when applied to invalid values? We decided to keep our model simple, and to avoid these questions, by refusing to predict the outcome of a program that reads an uninitialized scalar variable. Version 5.0 of the Standard has changed the description of this situation, but once again the exact rules are vague. Therefore, we are continuing to use the simple model that refuses to predict the outcome of a program in these situations.

2.3 Static Checks and Overload Resolution

It is conventional to process Ada in two (or more) steps; the first step checking syntax, applying all of the "legality" checks, and resolving any overloading. We planned to define the semantic model in a similar way, with two distinct definitions. The first *static semantics* takes Ada source text, and produces a program in *intermediate syntax*. The intermediate syntax differs from Ada source text in several significant ways:

- Intermediate syntax is in the form of a tree, rather than a linear string of characters. Therefore intermediate syntax does not need to be parsed.

- Overloading has been resolved. Identifiers, characters, and operator names have been replaced by *intermediate identifiers* (in the set *Id*) in such a way that no two distinct declarations declare the same intermediate identifier. Any use of an identifier, character literal, or operator name has been replaced by a use of the appropriate element of *Id*. (Some names using selection, e.g., package components, are also replaced by intermediate identifiers.)

- Many of the notational conveniences of Ada have been eliminated. For example, infix operators are replaced by function calls.

- Generics are eliminated. Generic instantiations are expanded to a sequence of declarations.

- Some additional information is included. For example, a *completing declaration* is explicitly marked as such.

The intermediate syntax is described in Appendix B.

We have not formally defined the static analysis, although we believe that a Natural Semantics formulation of the static rules is possible.

2.4 Environments, Entities, and Stores

The Natural Semantics definition is faithful to the Reference Manual in its treatment of entities. We use several different sorts of "entity locations", which serve as unique names or references for entities. When a declaration is elaborated, new names are generated for any of the entities that need to be "created", and the environment is updated to reflect the association of the declared *Id* with a view of one of these entities. The store is a collection of mappings indexed by these different entity locations, which associates a value or meaning with each entity.

This indirect representation, using references to entity locations rather than the meanings of entities, makes it fairly easy to handle situations where an entity has a declaration that is separate from its definition. Between the declaration and definition, any references to the entity cannot make use of the definition (because it is not yet known). The location associated with the entity *is* known, and can be used.

2.5 Ordering

The Ada Standard gives implementations considerable freedom to select the order in which actions are performed. For example, in evaluating a sum, either the left or the right operand might be evaluated first. It is easy to write a program that gives different results depending on which order is chosen.

In order to write a concise description of the possible effects of the evaluation of constructs allowing a choice of orders, we define a set of *actions*, and several ways of combining actions. One combination, written by enclosing the actions in braces, allows the actions to be carried out in an arbitrary order; another, written by enclosing the actions in square brackets, requires the actions to be carried out in strict sequential order.

Actions are similar to judgements, except that the states do not appear explicitly. When actions are executed in some order, suitable states are added and the corresponding judgement is used. We write the actions in a notation that makes obvious the judgement for the corresponding execution. For example, corresponding to the judgement $S_1, E \vdash_{stm} Stm \Rightarrow S_2$ is an action written as $'E \vdash_{stm} Stm \Rightarrow'$; corresponding to the judgement $S_1, E \vdash_{exp} Exp \Rightarrow V, S_2$ is an action written as $'E \vdash_{exp} Exp \Rightarrow V'$.

We also use *states*, which are combinations of *stores* and control flow information (for example, whether an exception has been raised, whether a return command has been executed, whether execution is normal.) The rules defining the execution of a combined action check the control flow information to skip some actions if that is appropriate. For example, in executing "a followed by b", if the execution of "a" propagates an exception, the action "b" is not executed.

One advantage to this approach is that the actions themselves look simpler than the corresponding judgements, because the flow of control through them is described by the way they are combined. For example, in defining the possible results of a sum using explicit ordering, we would need several judgements, including

$$\frac{\begin{array}{l} s, E \vdash_{exp} e1 \Rightarrow v1, s1 \\ s1, E \vdash_{exp} e2 \Rightarrow v2, s2 \end{array}}{s, E \vdash_{exp} e1 + e2 \Rightarrow v1 + v2, s2}$$

and

$$\frac{\begin{array}{l} s, E \vdash_{exp} e2 \Rightarrow v2, s1 \\ s1, E \vdash_{exp} e1 \Rightarrow v1, s2 \end{array}}{s, E \vdash_{exp} e1 + e2 \Rightarrow v1 + v2, s2}$$

and others to account for exceptions. Instead, using actions, we can write

$$\frac{S_1 \vdash \left\{ \begin{array}{l} 'E \vdash_{exp} e1 \Rightarrow v1' \\ 'E \vdash_{exp} e2 \Rightarrow v2' \end{array} \right\} S_2}{S_1, E \vdash_{exp} e1 + e2 \Rightarrow v1 + v2, S_2}$$

which accounts for the different possible orders of evaluation and for the propagation of an exception from one of those evaluations. (We still need something extra to account for an overflow in the addition.)

2.6 Types

It is awkward to define a domain of "type values" that describe types, because the exact characteristics of a type can change through its scope. For example, a type may be limited in some parts of its scope, and nonlimited in other parts (such as the body of a package defining the type of a component); a type can be private in some places and not in others. Furthermore, an incompletely defined type can be used in various ways (such as a record component or designated type of an access type); the characteristics of the using type can change after the incompletely defined type's full definition.

In order to simplify the treatment of these situations, types are described by *descriptors* that refer to other types by their locations (see Section 2.4), rather than by their descriptors. This has the disadvantage that descriptors are not meaningful in isolation, but only with respect to the store that associates descriptors with type locations. However, it has several advantages:

- it gives a simple characterization of when types are the same; each type location represents a distinct type;
- when the characteristics of a type of a component change, that change can be reflected in just one descriptor; and
- circularities in type descriptors are easily handled (without needing any tricky domains allowing for infinite data structures).

An example illustrating these advantages is

```
type A;
type B is access A;
type A is access B;
```

2.7 Values

We expected it to be easy to describe the domain of values that objects might assume. It was surprising that this was not so. As mentioned above, the addition of *invalid* values to scalar types adds several complications, as the nature of such values is not completely specified. The latest version (5.0) of the proposed Standard no longer uses the term "invalid value"; instead, a variable may have an "invalid representation" [13.9.1].

We also argued whether "abnormal" values would be needed in order to model the concept of *abnormal* objects. We were able to avoid this, since the circumstances that can lead to abnormal objects are being treated as unpredictable executions.

There are a few situations where it is difficult to determine the set of values associated with a type. For example, given the declaration

```
subtype Void is Integer range 1 .. 0; -- an empty range
type R is record
  v: Void
  end R;
```

There are no values of subtype `Void`. However, there *are* values of type `R`. For example, a variable of type `R` can be declared, and it is not an error to "read" the value of such a variable, or to assign this value to a second variable of type `R`.

The problems with this type are related to those for uninitialized scalar variables, and we adopt a simple approach to solve them. We use a special indicator to denote an uninitialized scalar value. A scalar subcomponent of an object can have this value. If this "uninitialized" value is read, execution is unpredicted.

The set of values of an enumeration type is not obvious. Given the declarations

```
type E is (red, green);
for E use (red => 1; green => 100);
```

version 4.0 of the Draft Standard suggested that there were two "valid values" of type `E`, and (at least) 98 "invalid values" between them. The number of elements in an array indexed by `E`, then, is open to question. Are there elements corresponding to the invalid members of the base range?

Types declared with *per-object constraints* do not have obvious sets of values (since the constraint applied to a subcomponent might depend on the specific object of the type). Our model simply does not cover the kinds of per-object constraints that lead to this difficulty.

Our model for values uses integers to represent discrete values (even if the value is of an enumeration type). This means that the values of different types are not necessarily different. It would certainly be possible to mark values in such a way that no value belongs to more than one type, but there seems to be no benefit to doing so. An Ada program cannot directly compare values of different types, so there is no way for this detail to influence the outcome of a program.

2.8 Objects

It is normal in semantic definitions to use *location semantics* for variables, but different approaches can be used in accounting for structured (composite) variables and their components.

The approach that seems most convenient for us is to associate locations with *entire* variables (that is, variables that are not subcomponents of other variables). Every object is characterized by its location and a *selector* indicating which component of the entire variable it is.

2.8.1 Actual Subtypes

The *actual subtype* of an object is sometimes different from the *nominal subtype* in its declaration. This is an issue for assignments [5.2(11)] and formal parameters of mode **in out** or **out** that are passed by copy [6.4.1(17)]. So it is important only for variables.

The actual subtype of a variable differs from its nominal subtype in the following circumstances:

- the object is a declared object, and is constant, aliased, or has an indefinite nominal subtype [3.3.1(9)].

- the object is a formal parameter. [6.4.1(16)] states

  > A formal parameter of mode **in out** or **out** with discriminants is constrained if either its nominal subtype or the actual parameter is constrained.

  [6.4.1(12-15) gives additional rules for **out** parameters.

- the object is a generic formal object of mode **in out**. [12.4(8)] states that the nominal subtype is taken from the declaration of the formal, while the actual subtype is taken from the actual.
- the object is declared by an allocator, and the designated type of the result subtype of the allocator is indefinite or has discriminants ([3.3(23) and [3.10(9)]).
- the object is a view of another object, and the subtype of the view is indefinite [3.3(23)].

This leads to the question of how the actual subtype of an object is determined, and where, if needed, the actual subtype information is stored. There are two reasonable choices: the actual subtype might be associated with the object, or with each view of the object. However, two views of an object need not have the same type. Obviously, in such a case the actual subtypes must be different. Moreover, in a procedure call, the actual subtype of the view denoted by the formal can be different from the actual subtype of the actual parameter, even when the views have the same type. For example, the actual might be of an unconstrained discriminated type and the formal constrained. Therefore, we have decided to store the actual subtype as part of every view of an object.

2.8.2 Initialization

The calculation of the implicit initial value for an object is difficult to describe, as there is considerable freedom in the order of evaluation of default expressions used to initialize subcomponents. It is particularly awkward for subcomponents with discriminants; discriminants must be evaluated before any subcomponent that depends on them, but other subcomponents may have their initial values evaluated before them. So, given the declarations

```
type T(a: D := e0, b: D := e1) is record
  u: Integer := e2;
  v: U(a) := e3;
  w: U(b) := e4;
  x: S(a,b) := e5;
  end record;

y: T;
```

we must evaluate `e0` before `e3` and `e5`, and `e1` before `e4` and `e5`. The order of evaluation is otherwise unrestricted (unless references to `a` or `b` occur in `e2`, `e3`, or `e4`.)

In order to accommodate this flexibility, we use a variation on the "in some order" rules described in Section 2.5. We add an additional datum to the left and right of the turnstile; this datum records which discriminants have had their initializing expressions evaluated (and what the resulting value is). The individual actions record their prerequisites (that is, which discriminants must be evaluated before the action can be executed).

In the record of evaluated discriminants, we cannot simply use the name of the discriminant, as two subcomponents might have the same type, and thus have discriminants of the same name. Instead, for each discriminant subcomponent to be evaluated we generate a unique identifier. A "discriminant environment", associating discriminant identifiers with discriminant names, is therefore also used in the judgements for initialization.

Another difficulty in describing the initialization of objects concerns *per-object constraints*. Ada 9X allows the name of a type to be used in its own definition, in which case it stands for the "current instance" of the type. Thus, a constraint on a component can refer to the containing object. Describing this formally can be difficult: the object might not exist until its subtype can be determined (which involves elaborating per-object constraints), yet the elaboration of a per-object constraint may refer to the object. We decided not to consider per-object constraints that refer to the "current instance" (but we allow them to refer to discriminants).

2.9 Aliasing

Some rules concerning aliasing look difficult to model. [6.2(12)] states

> If one name denotes a part of a formal parameter, and a second name denotes a part of a distinct formal parameter or an object that is not a formal parameter, then the two names are considered ***distinct access paths***. If an object is of a type for which the parameter passing mechanism is not specified, then it is a bounded error to assign to the object via one access path, and then read the value of the object via a distinct access path while the first access path still exists. The possible consequences are that Program_Error is raised, or the newly assigned value is read, or some old value of the object is read.

If we are to allow for accurate predictions of the effects of procedure calls (or to refuse to predict the outcome of calls that might involve aliasing), we need to be able to recognize, at a minimum, when the above rule might apply. It is not enough just to say that parameters of certain types may be passed by copy or by reference at the whim of an implementation, because the above paragraph allows for results that might not be produced under either of the two passing mechanisms. We might refuse to predict the result of any call with aliasing, but that can be hard to recognize if access values are used. Unfortunately, the notion of "access paths" is not well defined by the Reference Manual, and the precise meaning of the aliasing rule is unclear.

We have submitted several official comments on the aliasing rules, and had some discussions with the Mapping Team on possible interpretations of these rules. One model that may work can be sketched as follows: we would define a function *access_path* on names, which gives an element of *optional Id*. This "access path" gives the *Id* associated with the declaration of some variable denoting the object denoted by the name. This might be the declaration that created the object, or might be the name of a formal parameter. If the object was dynamically created, the access path is null.

In most cases, the definition of *access_path* is simple, e.g.,

> *access_path*(*Id*) = *some*(*Id*) if *Id* is not declared by a renaming declaration
> *access_path*(*Nam*.*Id*) = *access_path*(*Nam*)
> *access_path*(*Nam*(*exp*, . . .)) = *access_path*(*Nam*)
> *access_path*(*Nam*.`all` = *none*

For *Id*s declared by renaming declarations, we would want to use the access path of the renamed object.

In order to state the rule of 6.2(12), we would associate a "last update path" with every object (including subobjects). Whenever an object is updated by an assignment, the access path of the name used in the assignment statement is recorded in the object (and every subobject and containing object). In addition, formal parameter objects are updated in a call with the formal parameter *Id* as the last update path, and after a call, any **in out** or **out** parameter objects are updated by the access paths of the corresponding actual parameter names. It would be a bounded error to evaluate a name denoting all or part of a formal parameter for which the parameter passing mode is unspecified, if the access path differs from the last update path of the object it denotes. Similarly, it would be a bounded error to evaluate a name if the last update path of the object it denotes is a formal parameter for which the parameter passing mode is not specified.

These rules account for most of the situations described by 6.2(12), but probably need refinement to deal properly with access values created by `Access` attributes.

Chapter 3

Semantic Domains

In this chapter, we define the domains of values used in the semantics. These sets describe the values assumed by the various entities of Ada 9X.

These definitions have been used as the basis for the Prolog representation of the Natural Semantics definitions presented in Chapter 4. However, some of the definitions defined here have not yet been incorporated into the Natural Semantics definition, and some small inconsistencies between the two definitions have not yet been eliminated.

3.1 Basic Notations

In this section, we define some basic notions that will be used in the model.

An *association* provides a finite function with an enumeration of its domain. It is convenient to represent such a function by enumerating (domain, range) pairs; the finite function is then the range of the sequence.

$$X \xrightarrow{a} Y == \{\ s : \operatorname{seq} X \times Y \mid \operatorname{ran} s \in X \twoheadrightarrow Y \land \# \operatorname{ran} s = \# s\ \}$$

Functions *adom* and *aran* give the domain and range of an association. Function $_ \cdot _$ is used to apply an association to an argument (as though it were a finite function).

$$[X, Y]$$

$$\mathit{adom}\ _ : (X \xrightarrow{a} Y) \rightarrow \mathbf{P}\, X$$
$$\mathit{aran}\ _ : (X \xrightarrow{a} Y) \rightarrow \mathbf{P}\, Y$$
$$_ \cdot _ : (X \xrightarrow{a} Y) \times X \twoheadrightarrow Y$$

$$\forall\, a : X \xrightarrow{a} Y \bullet \mathit{adom}\ a = \operatorname{dom}(\operatorname{ran} a)$$
$$\forall\, a : X \xrightarrow{a} Y \bullet \mathit{aran}\ a = \operatorname{ran}(\operatorname{ran} a)$$
$$(A, x) \in \operatorname{dom} _ \cdot _ \Leftrightarrow x \in \mathit{adom}\ A$$
$$x \in \mathit{adom}\ A \Rightarrow A \cdot x = (\operatorname{ran} a)\ x$$

We sometimes use optional values:

$$\mathit{optional}\ X ::= \mathit{none} \mid \mathit{some}\langle\!\langle X \rangle\!\rangle$$

Function *maximal* returns the set of maximal values of an arbitrary relation, where a maximal value of the relation $R : X \leftrightarrow X$ is defined as an element z of X such that no $y \neq z$ satisfies zRy.

$$
\begin{array}{l}
[X] \\
\hline
maximal : (X \leftrightarrow X) \rightarrow \mathbb{P}\, X \\
\hline
maximal(R) = X \setminus \mathrm{dom}(R \setminus \mathrm{id}\, X)
\end{array}
$$

Function *restrict* restricts both the domain and the range of a relation to some given set.

$$
\begin{array}{l}
[X] \\
\hline
restrict : (\mathbb{P}\, X \times (X \leftrightarrow X)) \rightarrow (X \leftrightarrow X) \\
\hline
restrict(S, R) = (S \times S) \cap R
\end{array}
$$

Equivalently,

$$restrict(S, R) = (S \lhd R) \rhd S$$

3.2 Entities and the Environment

After overload resolution, every occurrence of an identifier in an Ada program can be replaced by an *Id*, so that each *Id* has at most one declaration in the program.

The elaboration of a declaration creates an entity, and the *Id* of the declaration then denotes a *view* of this entity. A declaration might be elaborated many times (e.g., if it appears in a subprogram body), denoting a different entity each time.

$$Environment == Id \twoheadrightarrow View$$

A *view* identifies an entity and provides some characteristics that affect the use of the entity. For example, there can be several views of the same subprogram, each having different parameter names and default expressions. Views refer to entities by using *locations* of various types.

$$[Type_location, Subtype_location, Object_location, Subprogram_location]$$

There are no views associated with packages or generic declarations. Packages are significant in their provision of information hiding and modularization, but those aspects concern the static semantics, not the dynamic semantics. Generic declarations are expanded by the static semantics, so that only ordinary (non-generic) declarations appear in the intermediate syntax.

Exceptions are unusual entities. No matter how often an exception declaration is elaborated, the same exception is denoted. This exception is represented by an *Exception_Id* that is determined by the static semantics. (The *Exception_Id* could be chosen to be the *Id* of the declaration, for example).

$$
\begin{array}{rl}
View ::= & object_view \langle\langle Object_location \times Subtype \rangle\rangle \\
\mid & subtype_view \langle\langle Subtype_location \rangle\rangle \\
\mid & subprogram_view \langle\langle Subprogram_location \times Profile \rangle\rangle \\
\mid & exception \langle\langle Exception_Id \rangle\rangle \\
\mid & constant \langle\langle Value \rangle\rangle
\end{array}
$$

Most kinds of entity are held in a *store*. Assigning a *location* to refer to the entity, and placing an entity at that location in the store, corresponds to what the Reference Manual calls "creating" the entity. This activity happens when a declaration is elaborated.

Several kinds of entities are used in the semantics definition:

- objects, which have values;

- subtypes, with their associated type, constraint, and attributes;
- types, with descriptors and optional parents;
- subprograms, with formals and bodies; and
- operations (representing the "predefined operations" of the Reference Manual).

Store

$$types : Type_location \twoheadrightarrow Type$$
$$subtypes : Subtype_location \twoheadrightarrow Subtype$$
$$objects : Object_location \twoheadrightarrow Value$$
$$subprograms : Subprogram_location \twoheadrightarrow SubprogramOrOp$$

$$SubprogramOrOp ::= subprogram\langle\langle Environment \times (\mathrm{seq}\, Id) \times Dcl \times Stm\rangle\rangle \mid operation\langle\langle \ldots \rangle\rangle$$

The different sorts of entities are described in the following sections.

Evaluation is defined in terms of a *state*, which (usually) includes a store, as well as certain control information. States are defined below in Section 3.6. Function *the_store*, giving the store associated with a state, is used in some of the definitions below.

3.3 Thunks

In several situations it is necessary to record an expression together with the environment in which it appears, so that the expression can be evaluated in some other context. The environment is retained so that any *Id*s appearing in the expression have their correct denotation. We call this combination of an expression and an environment a *thunk*.

$$Thunk == Exp \times Environment$$

Thunks appear in record type descriptors (where they describe the initializing values of any explicitly initialized components), and in parameter lists (where they describe any default values for parameters).

3.4 Values

There are several kinds of values of interest:

- discrete values (represented by integers)
- real values (represented by rationals)
- access values (represented by views of objects or of subprograms)
- record values (represented by partial functions)
- array values (represented by partial functions)

It is possible to use a model where the values of each type are distinct; however, the benefit of doing so is not completely clear. The rules of the language do not allow for comparisons of values of different types, so there is no way of telling whether these sets are disjoint.

3.4.1 Ranges

Ranges have two bounds, and determine a set of values of a scalar type.

$$Range ::= discrete_range\langle\langle \mathbb{Z} \times \mathbb{Z} \rangle\rangle \mid real_range\langle\langle Rational \times Rational \rangle\rangle$$

$$Discrete_range == \operatorname{ran} discrete_range$$

$$Real_range == \operatorname{ran} real_range$$

(The definition of the bounds functions contains a forward reference to functions *discrete_value* and *real_value*, defined in Section 3.4.5.)

$$\begin{array}{|l}
low_bound, high_bound : Range \rightarrow Value \\
\hline
low_bound(discrete_range(l,h)) = discrete_value(l) \\
high_bound(discrete_range(l,h)) = discrete_value(h) \\
low_bound(real_range(l,h)) = real_value(l) \\
high_bound(real_range(l,h)) = real_value(h)
\end{array}$$

$$\begin{array}{|l}
make_range : Value \times Value \nrightarrow Range \\
\hline
make_range(discrete_value(v), discrete_value(v')) = discrete_range(v, v') \\
make_range(real_value(v), real_value(v')) = real_range(v, v')
\end{array}$$

$$\begin{array}{|l}
_\ belongs_to\ _ : Value \leftrightarrow Range \\
values_of_range : Range \rightarrow \mathbb{P}\ Value \\
_\ is_included_in\ _ : Range \leftrightarrow Range \\
\hline
v\ belongs_to\ R \Leftrightarrow v \in values_of_range(R) \\
values_of_range(discrete_range(l,h)) = discrete_value(\!|\, l \ldots h \,|\!) \\
values_of_range(real_range(l,h)) = real_value(\!|\, l \ldots h \,|\!) \\
R\ is_included_in\ R' \Leftrightarrow values_of_range(R) \subseteq values_of_range(R')
\end{array}$$

3.4.2 Index Ranges

Arrays are indexed by sequences of discrete values. Index ranges are determined by a sequence of discrete ranges.

$$Array_bounds == \operatorname{seq}_1(Discrete_range)$$

Each sequence of bounds determines a set of indices:

$$\begin{array}{|l}
indices : Array_bounds \rightarrow \mathbb{P}(\operatorname{seq}_1 Value) \\
\hline
\forall B : Array_bounds \bullet indices(B) = \{ s \in \operatorname{seq}_1 Value \mid \\
\qquad \#s = \#B \land \forall n \in \operatorname{dom} s \bullet s(n)\ belongs_to\ B(i) \}
\end{array}$$

3.4.3 Tags

We use a set of *tags*. The precise nature of this set is immaterial.

$$[Tag]$$

3.4.4 Bindings

Bindings are simply partial mappings from *Ids* to values. Most often these *Ids* are the names of record fields or discriminants.

$$Binding == Id \leftrightarrow Value$$

3.4.5 Values

Although it seems redundant, we include the bounds as part of an array value. This is because two arrays with no components (thus, with the same mapping function) can have different bounds.

Record values are furnished with optional tags, discriminants, and other components. This allows the descriptions of the various language rules concerning tagged records, discriminated records, and normal records to be combined.

A special value, ***uninitialized_value***, is used for uninitialized scalar objects. This is used to detect when such an object is read (in which case the result of the execution is ***unpredicted***).

$$\begin{aligned} Value ::= \ & \mathit{uninitialized_value} \\ \mid \ & \mathit{discrete_value}\langle\langle \mathbb{Z} \rangle\rangle \\ \mid \ & \mathit{real_value}\langle\langle Float \rangle\rangle \\ \mid \ & \mathit{access_value}\langle\langle \mathit{optional}\ View \times \mathit{optional}\ SubprogramLabel \rangle\rangle \\ \mid \ & \mathit{record_value}\langle\langle (\mathit{optional}\ Tag) \times Binding \times Binding \rangle\rangle \\ \mid \ & \mathit{array_value}\langle\langle \{\ B : Array_bounds, v : (\mathrm{seq}\ Value) \leftrightarrow Value \mid \mathrm{dom}\, v = \mathit{indices}(B)\ \} \rangle\rangle \end{aligned}$$

We can define various sets of values referred to in the language rules:

$$Discrete_value == \mathrm{ran}\ discrete_value$$

$$Real_value == \mathrm{ran}\ real_value$$

$$Access_value == \mathrm{ran}\ access_value$$

$$Scalar_value == Discrete_values \cup Real_values$$

$$Elementary_value == Scalar_values \cup Access_values$$

$$Composite_value == \mathrm{ran}\ record_value \cup \mathrm{ran}\ array_value$$

3.5 Types and Subtypes

Every type has an associated *type descriptor* giving the characteristics of the type (and possibly referring to other types via their type locations).

The descriptors are defined here, but described in the sections that follow.

$$\begin{aligned} Type_Descriptor ::= \ & \mathit{enumeration_dsc}\langle\langle \mathbb{N}_1 \rangle\rangle \\ \mid \ & \mathit{signed_integer_dsc}\langle\langle \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rangle\rangle \\ \mid \ & \mathit{modular_integer_dsc}\langle\langle \mathbb{N}_1 \rangle\rangle \\ \mid \ & \mathit{universal_integer_dsc}\langle\langle \mathbb{Z} \times \mathbb{Z} \rangle\rangle \\ \mid \ & \mathit{float_dsc}\langle\langle \mathbb{N}_1 \times Rational \times Rational \times Float_Implementation \rangle\rangle \\ \mid \ & \ldots \text{(something for fixed-point types)} \\ \mid \ & \mathit{array_dsc}\langle\langle (\mathrm{seq}_1\ Subtype) \times Subtype \rangle\rangle \\ \mid \ & \mathit{record_dsc}\langle\langle (\mathit{optional}\ Tag) \times Discriminant_descriptor \times Component_list_descriptor \rangle\rangle \\ \mid \ & \mathit{class_dsc}\langle\langle Type_location \rangle\rangle \\ \mid \ & \mathit{access_dsc}\langle\langle Subtype_location \times Access_Mode \rangle\rangle \\ \mid \ & \ldots \end{aligned}$$

A *type* is then a combination of an optional parent (in case the type was derived) plus a descriptor.

$$Type ::= (\textbf{optional}\ Type_location) \times Type_Descriptor$$

For each kind of type in Ada 9X, we define a type descriptor. Additionally, for each descriptor we define the set of values of the type:

$$descriptor_values : Type_Descriptor \times State \rightarrowtail \mathbb{P}_1\ Value$$

Function *type_values* gives the set of values associated with a type location given a state:

$$type_values : Type_location \times State \rightarrow \mathbb{P}\ Value$$

$$\forall\, l : Type_location;\ S : State \bullet type_values(l, S) = descriptor_values(snd(the_store(S).types(u)), S)$$

Note that these sets of values might change over time as the information about a type is updated.

3.5.1 Subtypes and Constraints

A subtype is a combination of a type, a constraint, and certain attributes [3.2(8)]. There are in fact two sorts of subtypes; we will call them "partial" subtypes and "true" subtypes. Partial subtypes can contain unevaluated ***per-object constraints***, for example, references to discriminants. These partial subtypes appear as the subtypes of components of a record with discriminants. When an object of a subtype is created, some of these per-object constraints are elaborated and the true subtype of the object and its components is determined.

There are several cases where a partial subtype cannot be elaborated: in a variant record, per-object constraints in initially unused components are not elaborated; in an initialized object, *none* of the per-object constraints are elaborated. The Reference Manual is not completely clear on this point, and for now we will only consider per-object constraints that are references to discriminants.

Partial subtypes appear only as the subtypes of components of types with discriminants; named subtypes, and the subtypes of objects, will always be "true" subtypes.

The Reference Manual identifies three kinds of constraints: range constraints, index constraints, and discriminant constraints. In fact, the last two can also be applied indirectly, to an access type having a designated subtype to which the constraint would apply directly. (Only one level of indirection is allowed; given

```
type A1 is access String;
type A2 is access A1;
```

an index constraint can be applied to A1, but not to A2.) We may find it useful to distinguish these indirect constraints from their direct counterparts.

$$\begin{aligned} Constraint ::= {} & no_constraint \\ | {} & range_constraint\langle\langle Range\rangle\rangle \\ | {} & index_constraint\langle\langle \mathrm{seq}_1\ Discrete_range\rangle\rangle \\ | {} & discriminant_constraint\langle\langle Binding\rangle\rangle \\ | {} & indirect_index_constraint\langle\langle \mathrm{seq}_1\ Discrete_range\rangle\rangle \\ | {} & indirect_discriminant_constraint\langle\langle Binding\rangle\rangle \end{aligned}$$

Some values *satisfy* a constraint:

$$_\ \mathit{satisfies}\ _ : \mathit{Value} \leftrightarrow \mathit{Constraint}$$

$$\forall v : \mathit{Value} \bullet v\ \mathit{satisfies}\ \mathit{no_constraint}$$

$$\forall v : \mathit{Value} \bullet v\ \mathit{satisfies}\ \mathit{range_constraint}(R) \Leftrightarrow v\ \mathit{belongs_to}\ R$$

$$\mathit{array_value}(B, a)\ \mathit{satisfies}\ \mathit{index_constraint}(S) \Leftrightarrow B = S$$

$$\mathit{record_value}(t, d, r)\ \mathit{satisfies}\ \mathit{discriminant_constraint}(d') \Leftrightarrow d = d'$$

Now we can define subtypes:

Subtype

$$\mathit{type} : \mathit{Type_location}$$
$$\mathit{constraint} : \mathit{Constraint}$$
$$\mathit{attributes} : \mathit{Attributes}$$

For every subtype, there is an associated set of values, namely the values of the associated type that satisfy the constraint.

$$\mathit{subtype_values} : \mathit{Subtype} \rightarrow \mathit{State} \rightarrow \mathbb{P}\ \mathit{Value}$$

$$\forall S : \mathit{State};\ s : \mathit{Subtype} \mid s.\mathit{type} \in \mathrm{dom}\ \mathit{the_store}(S).\mathit{types} \bullet$$
$$\quad \mathit{subtype_values}(s, S) = \{\ v : \mathit{Value} \mid v \in \mathit{type_values}(s.\mathit{type}, S) \land v\ \mathit{satisfies}\ s.\mathit{constraint}\ \}$$

We use function *subtype* to create subtype values:

$$\mathit{subtype} : \mathit{Type_location} \times \mathit{Constraint} \times \mathit{Attributes} \rightarrow \mathit{Subtype}$$

$$\mathit{subtype}(t, c, a).\mathit{type} = t$$
$$\mathit{subtype}(t, c, a).\mathit{constraint} = c$$
$$\mathit{subtype}(t, c, a).\mathit{attributes} = a$$

3.5.2 Partial Subtypes

Partial constraints are similar to constraints, except they may contain references to discriminants in place of values.

$$\mathit{Partial_value} ::= \mathit{value}\langle\langle \mathit{Value}\rangle\rangle \mid \mathit{discriminant_ref}\langle\langle \mathit{Id}\rangle\rangle$$

$$\mathit{Partial_discrete_range} == \mathit{Partial_value} \times \mathit{Partial_value}$$

$$\begin{aligned}\mathit{Partial_constraint} ::= &\ \mathit{no_constraint}\\ \mid &\ \mathit{range_constraint}\langle\langle \mathit{Partial_discrete_range}\rangle\rangle\\ \mid &\ \mathit{index_constraint}\langle\langle \mathrm{seq}_1\ \mathit{Partial_discrete_range}\rangle\rangle\\ \mid &\ \mathit{discriminant_constraint}\langle\langle \mathit{Id} \rightarrow\!\!\!\!\!+\ \mathit{Partial_value}\rangle\rangle\\ \mid &\ \mathit{indirect_index_constraint}\langle\langle \mathrm{seq}_1\ \mathit{Partial_discrete_range}\rangle\rangle\\ \mid &\ \mathit{indirect_discriminant_constraint}\langle\langle \mathit{Id} \rightarrow\!\!\!\!\!+\ \mathit{Partial_value}\rangle\rangle\end{aligned}$$

A partial subtype combines a subtype and a partial constraint. (Using the subtype allows us to do a "dependent compatibility check" at the right time, and also gives us the needed attributes when we actualize.)

$$\mathit{Partial_subtype} == \mathit{Subtype} \times \mathit{Partial_constraint}$$

Given a mapping from discriminant names to values, a partial constraint can be turned into a true constraint.

$$
\begin{array}{l}
actualize_value : Binding \rightarrow Partial_value \rightarrowtail Value \\
actualize_partial_range : Binding \rightarrow Partial_discrete_range \rightarrowtail Range \\
actualize_constraint : Binding \rightarrow Partial_constraint \rightarrowtail Constraint \\
actualize_subtype : Binding \rightarrow Partial_subtype \rightarrowtail Subtype \\
\hline
actualize_value\ f\ (value\ v) = v \\
actualize_value\ f\ (discriminant_ref\ n) = f(n) \\[4pt]
actualize_partial_range\ f\ (l, h) = \\
\quad discrete_range(actualize_value\ f\ l, actualize_value\ f\ h) \\[4pt]
actualize_constraint\ f\ no_constraint = no_constraint \\[4pt]
actualize_constraint\ f\ (range_constraint(l, h)) = \\
\quad range_constraint(actualize_partial_range\ f\ (l, h)) \\[4pt]
actualize_constraint\ f\ (index_constraint\ s) = \\
\quad index_constraint((actualize_partial_range\ f) \circ s) \\[4pt]
actualize_constraint\ f\ (discriminant_constraint\ d) = \\
\quad discriminant_constraint((actualize_value\ f) \circ d) \\[4pt]
actualize_constraint\ f\ (indirect_index_constraint\ s) = \\
\quad indirect_index_constraint((actualize_partial_range\ f) \circ s) \\[4pt]
actualize_constraint\ f\ (indirect_discriminant_constraint\ d) = \\
\quad indirect_discriminant_constraint((actualize_value\ f) \circ d) \\[4pt]
actualize_subtype\ f\ s = \\
\quad \mu\ Subtype \mid type = s.type\ \land \\
\qquad\qquad constraint = actualize_constraint\ f\ s.constraint\ \land \\
\qquad\qquad attributes = s.attributes
\end{array}
$$

3.5.3 Derived Types and Classes

A derived type is a new entity, but it generally uses a copy of the parent descriptor. The set of values of the derived type is then the same as the set of values of the parent type. However, when a derived type definition furnishes new discriminants or defines a type extension, a new descriptor is needed.

The store records some information about derivation, by recording the (optional) parent type [3.4(1)] of every known type.

$$
\begin{array}{l}
parent : State \rightarrow (Type_location \rightarrowtail Type_location) \\
\hline
parent(S) = \{\ t, t' : Type_location \mid fst(the_store(S).types(t)) = some(t')\ \}
\end{array}
$$

Type extensions are considered in Section 3.5.8. Abstract types are considered in Section 3.5.8.1.

3.5.3.1 Derivation Classes

The descriptor for a class_wide type has the form $class_dsc(t)$, where t is the *Type_location* of the root of the class. The values of this type are the values of all types derived (directly or indirectly) from t:

$$
\begin{array}{l}
descriptor_values(class_dsc(t), S) = \\
\quad \bigcup\{\ t' : Type_location \mid (u', u) \in parent(S)^{*} \bullet descriptor_values(t', S)\ \}
\end{array}
$$

The notions of [3.4.1(10)] are easily defined:

$$
\begin{array}{|l}
\textit{descendant}, \textit{ancestor} : \textit{State} \rightarrow (\textit{Type_location} \leftrightarrow \textit{Type_location}) \\
\textit{ultimate_ancestor} : \textit{State} \times \textit{Type_location} \nrightarrow \textit{Type_location} \\
\hline
\textit{ancestor}(S) = \textit{parent}(S)^* \\
\textit{descendant}(S) = \textit{ancestor}(S)^{-1} \\
t \in \operatorname{dom} \textit{the_store}(S).\textit{types} \Rightarrow \\
\quad \textit{ultimate_ancestor}(S, t) = ((\operatorname{ran} \textit{parent}(S)) \lhd \textit{ancestor}(S))\ t
\end{array}
$$

3.5.4 Scalar Types

Every scalar type records a base range, in addition to any other needed information.

$$
|\ \textit{base_range} : \textit{Type_Descriptor} \nrightarrow \textit{Range}
$$

Scalar types are either discrete types or real types. A value of discrete type is simply an integer; a value of a real type is a rational number.

Each scalar subtype determines a range, as specified in [3.5(6)]:

$$
\begin{array}{|l}
\textit{range_of_subtype} : \textit{Subtype} \times \textit{Store} \nrightarrow \textit{Range} \\
\hline
\sigma.\textit{constraint} = \textit{range_constraint}(R) \Rightarrow \textit{range_of_subtype}(\sigma, S) = R \\
\sigma.\textit{constraint} = \textit{no_constraint} \Rightarrow \\
\quad \textit{range_of_subtype}(\sigma, S) = \textit{base_range}(\textit{snd}(\textit{the_store}(S).\textit{types}(\sigma.\textit{type})))
\end{array}
$$

3.5.4.1 Enumeration Types

The type descriptor for an enumeration type has the form $\textit{enumeration_dsc}(n)$, where $n : \mathbb{N}_1$ gives the number of enumerands.

The base range of the enumeration is the set of discrete values with position numbers between 0 and $n - 1$ (inclusive). The values of an enumeration type are the values in the base range.

$$
\begin{array}{|l}
\textit{base_range}(\textit{enumeration_dsc}(n)) = \textit{discrete_range}(0, n - 1) \\
\textit{descriptor_values}(\textit{enumeration_dsc}(n), S) = \textit{discrete_value}(\!| 0 \mathinner{..} n - 1 |\!)
\end{array}
$$

3.5.4.2 Character Types

Character types are simply enumeration types.

3.5.4.3 Boolean Types

Boolean types are simply enumeration types.

3.5.4.4 Integer Types

There are three descriptors for integer types:

- $\textit{signed_integer_dsc}(bf, bl, f, l)$, with $bf, bl, f, l : \mathbb{Z}$ satisfying (according to [3.5.4(7)]),
  1. $bf \leq 0 \leq bl$,
  2. $bf + bl \in \{-1, 0\}$, and
  3. $\{f, l\} \subseteq bf \mathinner{..} bl$.

The base range of this type is $bf \mathinner{..} bl$.

- $modular_integer_dsc(m)$, with $m : \mathbb{N}_1$. (The base range is $0 \mathinner{..} (m-1)$ [3.5.4(7)].
- $universal_integer_dsc(bf, bl)$ for $bf, bl : \mathbb{Z}$ (although the base range seems to be irrelevant here).

`root_integer` is just a particular signed integer type.

In every case, the set of valid values of the type consists of all discrete values [3.5.4(6)].

$$descriptor_values(signed_integer_dsc(bf, bl, f, l), S) = discrete_value(\!|\mathbb{Z}|\!) \cup Invalid_value$$
$$descriptor_values(modular_integer_dsc(m), S) = discrete_value(\!|\mathbb{Z}|\!) \cup Invalid_value$$
$$descriptor_values(universal_integer_dsc(bf, bl), S) = discrete_value(\!|\mathbb{Z}|\!) \cup Invalid_value$$
$$base_range(signed_integer_dsc(bf, bl, f, l)) = discrete_range(bf, bl)$$
$$base_range(modular_integer_dsc(m)) = discrete_range(0, m-1)$$
$$base_range(universal_integer_dsc(bf, bl)) = discrete_range(bf, bl)$$

3.5.4.5 Floating Point Types

As acknowledged in the Ada 9X Rationale, the core language leaves the semantics of floating point operations largely unspecified. By contrast, the floating point annex (Annex G) is quite precise—though some flaws in the annex will be noted below. Therefore our model has two parts, one for the core and the other for Annex G.

The semantics of the Reference Manual refers to the underlying machine values and operations, and makes features of them visible, for example, in the values of attributes. We have attempted to model this semantics directly, so that it will be clear how to tell whether an actual implementation satisfies the semantic rules. This provides a model from the point of view of the implementor. It would have been easier (and from some points of view, perhaps, preferable) to make a model from the user's point of view: e.g., take the values of the attributes as given and simply state axiomatically, in terms of the attributes, the resulting constraints on the values returned by the predefined operations. The user's model is, of course, a consequence of the implementor's model.

The descriptor for a floating point type has four components:

$$float_dsc\langle\!\langle \mathbb{N}_1 \times Rational \times Rational \times Implementation \rangle\!\rangle$$

The first three components are provided directly by the type's definition: the requested precision and the bounds of its constraint. The fourth component characterizes the chosen implementation of the type. The descriptor for an integer type contains a component with analogous information, namely, the bounds of the underlying base type. We could represent the fourth component in a finitary way by listing the values of a large number of floating point attributes determined by the implementation. Instead, this component of the descriptor will consist of a model of the implementation itself, from which the attributes can be calculated.

$$descriptor_values(float_dsc(n, v, v', imp), S) = real_value(\!|Float|\!)$$
$$base_range(float_dsc(n, v, v', imp)) = inf(imp.ma.numbers) \mathinner{..} sup(imp.ma.numbers)$$

3.5.4.6 The core model of floating point

The values of a floating point type are rationals, with the possible addition of some extra things like signed zeroes or NaN's. For now these extra possibilities are ignored.

$Float == Rational \cup \ldots$

There is a problem in the Reference Manual: The possibility of signed zeroes or NaN's is incompatible with [RM-83 3.5.7(8)], which says that the set of values for a floating point type is the set of rational numbers.

3.5.4.6.1 Machine arithmetics Elaboration of a floating point type declaration includes the choice of an appropriate *Implementation* (from some predefined non-empty finite set of them) to model the type. One component of an implementation is a *machine arithmetic*, which consists of a radix, a set of machine numbers, and relations modeling the predefined binary and unary floating point operations. Floating point operations will be modeled not as functions but as relations, in order to model their potential non-determinism. Some of the predefined binary floating point operations return floats and some return booleans; all unary operations return floats. It is convenient to add a special "return value," *overflow*, to represent the possibility of overflow:

$FloatResult ::= overflow \mid result\langle\langle Float\rangle\rangle$

$BinOpFloat == Float^2 \leftrightarrow FloatResult$

$BinOpBool == Float^2 \leftrightarrow Boolean$

$UnOp == Float \leftrightarrow FloatResult$

It will also be handy to have an operation that extracts the (non-overflow) *Float* values from a set of *FloatResults*:

$$floats_of : \mathbb{P}\, FloatResult \rightarrow Float$$
$$floats_of(X) = result^{\sim}(\!| X |\!)$$

In schema *MachineArithmetic*:

- *radix* is the radix of the machine representation
- *numbers* is the set of machine numbers—that is, the set of storable values that will "fit" in any variable of the type.
- *plus*, *equals*, ..., are relations modeling the predefined floating point operations; *convert* represents type conversion of an arbitrary real value to a machine number of this arithmetic.

Notice that operations like *plus* are not restricted to returning machine numbers of the type as values. The machine numbers represent the storable values of the type, but operations may return, e.g., extra-precision values that are not immediately rounded.

$$
\begin{array}{l}
\textit{MachineArithmetic} \\
\hline
radix : \mathbb{N}_1 \\
numbers : \mathbb{F}_1\, Float \\
plus : BinOpFloat \\
\ldots \\
equals : BinOpBool \\
\ldots \\
uminus : UnOp \\
\ldots \\
convert : UnOp \\
float_outcomes : \mathbb{P}\, Float \\
\hline
float_outcomes = \\
\quad numbers \cup floats_of(\mathrm{ran}\, plus) \cup \ldots \cup floats_of(\mathrm{ran}\, convert) \\
(float_outcomes)^2 \subseteq \mathrm{dom}\, plus \\
\ldots \\
(float_outcomes)^2 \subseteq \mathrm{dom}\, equals \\
\ldots \\
float_outcomes \subseteq \mathrm{dom}\, uminus \\
\ldots \\
Float \subseteq \mathrm{dom}\, convert \\
\mathrm{ran}\, convert \subseteq ma.numbers \\
sup(numbers) \leq -inf(numbers)
\end{array}
$$

The axioms involving *float_outcomes* are technical conditions guaranteeing that the (non-overflow) results of any operation can legitimately be passed as arguments to any of the others.

The concluding inequality is all we can represent formally of paragraph 3.5.7(8):

> The base range (see 3.5) of a floating point type is symmetric around zero, except that it can include some extra negative values in some implementations.

Note: It would probably be reasonable to suppose that the machine numbers are (roughly) symmetric in a stronger sense: the set of machine numbers between $-sup(numbers)$ and $sup(numbers)$ is closed under additive inverse. The Reference Manual does not require this.

Note: This definition could be shortened if we simply assumed that the relations modeling all the predefined operations were total. One reason for *not* making that assumption is the desire that there be an obvious relation between this definition and actual floating point implementations. In representing an actual implementation as a machine arithmetic two principles apply: First, the *plus* relation modeling an actual implementation of + should contain $((x, y), z)$ if z is the actual result (presumably computed by the hardware in some register) of summing x and y; and should also include $((x, y), z')$ for every possible "perturbation" z' of z obtained by moving z to and from registers of other precisions or to and from storage. (Similar considerations apply to all other operations.) Second, it is sound to model an implementation by using relations that are supersets of this "minimal" model.

3.5.4.6.2 Parameters of floating point implementations We need two kinds of specifications for describing aspects of floating point implementations: A *MachineParam* is a specification requiring that certain floating point numbers actually be machine numbers of an implementation. It does not constrain the semantics of the floating point operations. The "representation-oriented attributes" of a type will be defined to return, essentially, the "strongest" *MachineParam* satisfied by the type's implementation. (Strictly speaking, we will define what it means for a machine arithmetic

to satisfy a *MachineParam*, and a machine arithmetic is only one component of an implementation.) An *AccurParam* does constrain the behavior of the floating point operations. Annex G will define the "model-oriented attributes" of a type to return, essentially, the strongest *AccurParam* satisfied by the type's implementation. The core of the Reference Manual says very little about the relation between the implementation and these model attributes.

3.5.4.6.3 Machine parameters A *MachineParam* is a triple whose elements are interpreted as, respectively, a mantissa length, a minimum exponent, and a maximum exponent.

$$
\begin{array}{|l}
\hline
\textit{MachineParam} \\
\textit{mantissa} : \mathbb{N}_1 \\
\textit{emin} : \{\, i : \mathbb{N} \mid i < 0 \,\} \\
\textit{emax} : \mathbb{N}_1 \\
\hline
\end{array}
$$

A machine arithmetic satisfies a *MachineParam* if all the canonical numbers defined in terms of these parameters (and of the machine arithmetic's radix) are machine numbers:

$$
\begin{array}{|l}
\textit{sat_float_param} : \textit{MachineArithmetic} \leftrightarrow \textit{MachineParam} \\
\hline
(\textit{ma}, \textit{fp}) \in \textit{sat_float_param} \Leftrightarrow \\
\qquad \textit{BoundedCanonical}(\textit{ma.radix}, \textit{fp.mantissa}, \textit{fp.emin}, \textit{fp.emax}) \subseteq \textit{ma.numbers}
\end{array}
$$

The *BoundedCanonical* numbers are defined in Section 3.5.4.6.4. Note that *sat_float_param* does not constrain the operations of the machine arithmetic in any way.

A first *MachineParam* is "improved by" a second if the second is at least as restrictive a specification as the first.

$$
\begin{array}{|l}
\textit{_improved_by_} : \textit{MachineParam} \leftrightarrow \textit{MachineParam} \\
\hline
(p_1, \textit{emin}_1, \textit{emax}_1)\ \textit{improved_by}\ (p_2, \textit{emin}_2, \textit{emax}_2) \Leftrightarrow \\
\qquad p_1 \leq p_2 \land \textit{emin}_1 \geq \textit{emin}_2 \land \textit{emax}_1 \leq \textit{emax}_2
\end{array}
$$

A *MachineParam* *fp* for *ma* is maximal if *ma* satisfies *fp* but satisfies no strict improvement of *fp*. The function *max_mach_params(ma)* returns the set of all maximal *MachineParams* satisfied by *ma*.

$$
\begin{array}{|l}
\textit{max_mach_params} : \textit{MachineArithmetic} \rightarrow \mathbb{F}\ \textit{MachineParam} \\
\hline
\textit{max_mach_params}(\textit{ma}) = \\
\qquad \textit{maximal}(\textit{restrict}(\{\, \textit{mattr} \mid \textit{sat_float_param}(\textit{ma}, \textit{mattr}) \,\}, \\
\qquad\qquad \textit{_improved_by_}))
\end{array}
$$

The generic constant *maximal* returns the set of maximal values of a relation. The generic constant *restrict* returns the result of restricting both the domain and range of a relation to the same set. These constants are defined in Section 3.1.

There is a problem in the Reference Manual: [RM-83 A.5.3] defines the representation-oriented attributes of a floating point type. They are intended, collectively, to denote a "best" *MachineParam* satisfied by the machine arithmetic of the type, but the definitions given there are not quite right. In particular, if *ma* is the machine arithmetic chosen to implement type **T**, the rules of the Reference Manual do not guarantee that

ma *sat_float_param* (**T'machine_mantissa**, **T'machine_emin**, **T'machine_emax**)

although this is surely one intended consequence of the rules.

3.5.4.6.4 Representations of floating point The canonical represention of floating point numbers are defined in the core semantics, Appendix A.

Our definitions will represent fractions with radix r and mantissa length m by length-m sequences of the "digits" $0, \ldots, r-1$. A normal representation is a representation whose first element is non-zero, or which consists solely of zeroes.

$$
\begin{array}{l}
reps : \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{P}\,\mathrm{seq}(\mathbb{N}) \\
normal_reps : \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{P}\,\mathrm{seq}(\mathbb{N}) \\
\hline
reps(r, m) = 1 \mathrel{..} m \rightarrow 0 \mathrel{..} (r-1) \\
s \in normal_reps(r, m) \Leftrightarrow s \in reps(r, m) \land \\
\quad s(1) = 0 \Rightarrow \forall\, i : \mathrm{dom}(s) \bullet s(i) = 0
\end{array}
$$

The operation *fraction_value* returns the fraction represented by the sequence s in radix r—that is, the "decimal" $.s(1)s(2)\ldots s(m)$, understood as a literal in base r.

$$
\begin{array}{l}
fraction_value : \mathbb{N}_1 \times \mathrm{seq}(\mathbb{N}_1) \rightarrow Rational \\
fractions : \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{P}\, Rational \\
normal_fractions : \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{P}\, Rational \\
\hline
fraction_value(r, s) = \Sigma_{i=1}^{\#s} s(i) * r^{-i} \\
fractions(r, m) = fraction_value(\!| reps(r, m) |\!) \\
normal_fractions(r, m) = fraction_value(\!| normal_reps(r, m) |\!)
\end{array}
$$

The model floating point numbers are those suitably definable in scientific notation, i.e., as fractions times powers of the radix.

$$
\begin{array}{l}
make_floats : \mathbb{P}\, Rational \times \mathbb{Z} \times \mathbb{N}_1 \rightarrow \mathbb{P}\, Rational \\
\hline
make_floats(fracs, exps, rad) = \\
\quad \{\, f : Rational, e : \mathbb{N}_1 \mid f \in fracs \land e \in exps \bullet \pm f * rad^{e} \,\}
\end{array}
$$

We are principally interested in two classes of "canonical" floating point numbers:

$$
\begin{array}{l}
Canonical : \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{P}\, Rational \\
BoundedCanonical : \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{P}\, Rational \\
\hline
Canonical(rad, mant, emin) = \\
\quad make_floats(normal_fractions(rad, mant), \{\, i : \mathbb{Z} \mid emin \leq i \,\}, rad) \\
BoundedCanonical(rad, mant, emin, emax) = \\
\quad make_floats(normal_fractions(rad, mant), \\
\qquad \{\, i : \mathbb{Z} \mid emin \leq i \leq emax \,\}, \\
\qquad rad)
\end{array}
$$

3.5.4.6.5 Implementations An *Implementation* consists of a machine arithmetic, a *MachineParam* modeling the representation-oriented attributes of the arithmetic, and a boolean indicating the response to numeric overflow. The properties of the machine arithmetic do not uniquely determine the appropriate *MachineParam*.

$$
\begin{array}{l}
\underline{\quad Implementation \quad} \\
ma : MachineArithmetic \\
machine_attr : MachineParam \\
overflows : Boolean \\
\hline
machine_attr \in max_mach_params(ma)
\end{array}
$$

An *AccurParam* is a 6-tuple whose elements are interpreted as a radix, a mantissa length, a minimum exponent, the bounds for a safe interval, and an indication of whether overflows are to be reported. (The constraints defined by the other parameters are interpreted more strictly if the "overflows" flag is true.)

$$
\begin{array}{|l}
\hline
\textit{AccurParam} \\
\textit{radix}, \textit{mantissa} : \mathbb{N}_1 \\
\textit{emin} : \{ \, i : \mathbb{N} \mid i < 0 \, \} \\
\textit{sfirst}, \textit{slast} : \textit{Float} \\
\textit{overflows} : \textit{Boolean} \\
\hline
\end{array}
$$

It is convenient to have an abbreviation for the set of safe numbers that an *AccurParam* defines.

$$
\begin{array}{|l}
\textit{safe} : \textit{AccurParam} \rightarrow \mathbb{P}\, \textit{Float} \\
\hline
\textit{safe}(\textit{ap}) = \textit{ap.sfirst} \,..\, \textit{ap.slast}
\end{array}
$$

We will formalize an essential notion of Annex G with *_has_accuracy_*, which says what it means for an implementation to satisfy an *AccurParam*. From the core model, we can extract only some minimal properties of this relation, expressed in the weaker notion *_has_weak_acc_*:

$$
\begin{array}{|l}
\textit{_has_weak_acc_} : \textit{Implementation} \leftrightarrow \textit{AccurParam} \\
\textit{_has_accuracy_} : \textit{Implementation} \leftrightarrow \textit{AccurParam} \\
\hline
\textit{imp has_weak_acc ap} \Leftrightarrow \\
\quad \textit{ap.radix} = \textit{imp.ma.radix} \wedge \\
\quad \textit{ap.mantissa} \leq \textit{imp.machine_attr.mantissa} \wedge \\
\quad \textit{ap.emin} \geq \textit{imp.machine_attr.emin} \wedge \\
\quad \{ \, \textit{ap.sfirst}, \textit{ap.slast} \, \} \subseteq \textit{ma.numbers} \wedge \\
\quad \textit{imp.overflows} = \textit{ap.overflows} \\
\textit{imp has_accuracy ap} \Rightarrow \textit{imp has_weak_acc ap}
\end{array}
$$

For any implementation *imp*, in Annex defines *model_attr(imp)*, a unique "best" *AccurParam* satisfied by an implementation. All we can say in the core semantics is that *imp* satisfies the weak accuracy requirements imposed by *model_attr(imp)*.

$$
\begin{array}{|l}
\textit{model_attr} : \textit{Implementation} \rightarrow \textit{AccurParam} \\
\hline
\textit{imp has_weak_acc model_attr}(\textit{imp})
\end{array}
$$

More precisely, Annex G defines *model_attr(imp)* to be a particular maximal *ap* such that *imp has_accuracy ap*.

The maximum number of decimal digits of accuracy is uniquely determined by the model-oriented attributes of the implementation.

$$
\begin{array}{|l}
\textit{digits} : \textit{Implementation} \rightarrow \mathbb{N}_1 \\
\hline
\textit{imp.radix} = 10 \Rightarrow \textit{digits}(\textit{imp}) = \textit{model_attr}(\textit{imp}).\textit{mantissa} \\
\textit{imp.radix} \neq 10 \Rightarrow \\
\quad \textit{digits}(\textit{imp}) = \textit{ceiling}(\\
\quad\quad (\textit{model_attr}(\textit{imp}).\textit{mantissa} * \textit{log}(10)/\textit{log}(\textit{imp.ma.radix})) + 1 \\
\quad)
\end{array}
$$

There is a problem in the Reference Manual: This definition of "digits" is not given anywhere in the Reference Manual. It is surely the intended one, but it does not seem to follow from anything in the Reference Manual.

There is a problem in the Reference Manual: Section A.5.3(67-68) defines the value of `S'Model_Mantissa` in incompatible ways, depending on whether or not the implementation "supports" Annex G. The same is true for `S'Model_Emin`. In consequence, an implementation can be valid for Core+G but not valid for the Core alone. (By contrast, the core semantics makes the semantics of the model-oriented attributes `'Safe_First` and `'Safe_Last` upward compatible by leaving them implementation-defined.)

The definitions of `S'Model_Mantissa` and `S'Model_Emin` given here are weaker than those of the core semantics. We require only that

$$\begin{aligned} \texttt{S'Model_Mantissa} &\leq \texttt{S'Machine_Mantissa} \\ \texttt{S'Model_Emin} &\geq \texttt{S'Machine_Emin} \end{aligned}$$

(See the definition of *_has_weak_acc_*.)

These definitions make the core semantics compatible with Annex G. In addition, they capture the only information that the present version of the Reference Manual allows a user to rely on across all implementations.

3.5.4.6.6 Satisfaction of a type definition

A floating point declaration supplies a requested precision (a value of $\mathbb{N}$) and, optionally, a constraint (two *Rationals*). The accuracy of the type's implementation must be at least as great as the requested precision and the safe range of the implementation must include the interval defined by the constraint. This requirement is captured by the definition of *sat_float_def*.

The Reference Manual says that any such implementation may be chosen. We represent the particular strategy that the implementation uses for choosing the implementation (such as choosing the coarsest acceptable implementation type) by the relation *implements_float_def*. The judgement defining elaboration of a type definition selects an implementation satisfying *implements_float_def*. All the reference manual requires of this relation is that it be consistent with *sat_float_def*:

$$\begin{array}{|l} sat_float_def, implements_float_def : \\ \quad Implementation \leftrightarrow (\mathbb{N} \times Rational \times Rational) \\ \hline (imp, (n, L, R)) \in sat_float_def \Rightarrow \\ \quad L \mathinner{..} R \subseteq model_attr(imp).sfirst \mathinner{..} model_attr(imp).slast \wedge \\ \quad m \leq digits(imp) \\ implements_float_def \subseteq sat_float_def \end{array}$$

3.5.4.6.7 Attributes

If *imp* is the implementation chosen for type `T`, then the basic implementation-oriented attribute values of `T` are given as follows, where we let $imp.machine_attr = (mant, emin, emax)$:

`T'Machine_Radix`	=	$imp.ma.radix$
`T'Machine_Mantissa`	=	$mant$
`T'Machine_Emin`	=	$emin$
`T'Machine_Emax`	=	$emax$
`T'Base'First`	=	$min(imp.ma.numbers)$
`T'Base'Last`	=	$max(imp.ma.numbers)$

The limits of the base range are the least and greatest machine numbers. This follows from 3.5(6)

> The base range of a scalar type is the range of finite values of the type that can be represented in every unconstrained object of the type

and from 3.5.7(8)

> The machine numbers of a floating point type are the values of the type that can be represented exactly in every unconstrained variable of the type.

The basic model-oriented attribute values of an implementation *imp* are given as follows (where we let $model_attr(imp) = (Mant, Emin, sfirst, slast)$):

`T'Model_Mantissa`	=	*Mant*
`T'Model_Emin`	=	*Emin*
`T'Safe_First`	=	*sfirst*
`T'Safe_Last`	=	*slast*
`T'Base'Digits`	=	$digits(imp)$

The definition of `T'Base'Digits` is something of a guess.

Note: The definitions of various attributes, such as `S'Model` (for floating point subtypes) and `S'Machine` (for fixed and floating point subtypes) say that the value returned is obtained "by rounding or truncating" the operand "to either one of the adjacent" model or machine numbers, as appropriate. It is not clear from this language whether these operations are non-deterministic.

3.5.4.7 Annex G

Annex G defines some more precise constraints on the floating point operations.

3.5.4.7.1 Model numbers and accuracy An accuracy parameter determines a set of model intervals (intervals bounded by the associated canonical real numbers) and associates a model interval with each bounded set (namely, the smallest model interval that contains it):

$$\begin{array}{l} ModelIntervals : AccurParam \rightarrow \mathbb{P}\ Float \\ ModelIntOf : \mathbb{P}\ Float \times AccurParam \rightarrow\!\!\!\!\rightarrow \mathbb{P}\ Float \\ \hline u \in ModelIntervals(ap) \Leftrightarrow \\ \quad \exists\, lo, hi : Float \bullet \\ \quad\quad u = lo \mathinner{\ldotp\ldotp} hi \wedge \\ \quad\quad \{\ lo, hi\ \} \subseteq Canonical(ap.radix, ap.mantissa, ap.emin) \\ ModelIntOf(X, ap) = \bigcap \{\ u \in ModelIntervals(ap) \mid X \subseteq u\ \} \end{array}$$

Given a "paradigm" operation f and an accuracy specification ap, we define for each x the set of results that approximate $f(x)$ to within the demands of ap—namely, the model interval of the set that results from applying f to the model interval of x. (The same applies, ***mutatis mutandis***, to the binary operations.) All definitions follow the same pattern, but the type restrictions of Z require us to provide separate definitions for unary operations, binary operations returning floats, and binary operations returning booleans.

$$\begin{array}{l} ResultUnOp : (Float \rightarrow Float) \times AccurParam \rightarrow (Float \rightarrow \mathbb{P}\ Float) \\ ResultBinOpFloat : (Float^2 \rightarrow Float) \times AccurParam \rightarrow (Float^2 \rightarrow \mathbb{P}\ Float) \\ ResultBinOpBool : (Float^2 \rightarrow Bool) \times AccurParam \rightarrow (Float^2 \rightarrow \mathbb{P}\ Bool) \\ \hline ResultUnOp(f, ap) = \\ \quad \lambda\, x : Float \bullet \\ \quad\quad ModelIntOf(f (\!| ModelIntOf(\{x\}, ap) |\!), ap) \\ ResultBinOpFloat(f, ap) = \\ \quad \lambda\, x, y : Float \bullet \\ \quad\quad ModelIntOf(f (\!| ModelIntOf(\{x\}, ap) \times ModelIntOf(\{y\}, ap) |\!), ap) \\ ResultBinOpBool(f, ap) = \\ \quad \lambda\, x, y : Float \bullet \\ \quad\quad ModelIntOf(f (\!| ModelIntOf(\{x\}, ap) \times ModelIntOf(\{y\}, ap) |\!), ap) \end{array}$$

Operands are "safe for" a paradigm operation if they and all their approximate results are safe.

$$
\begin{array}{|l}
SafeForUnOp : (Float \to Float) \times AccurParam \to \mathbb{P}\, Float \\
SafeForBinOpFloat : (Float^2 \to Float) \times AccurParam \to \mathbb{P}\, Float^2 \\
SafeForBinOpBool : (Float^2 \to Bool) \times AccurParam \to \mathbb{P}\, Float^2 \\
\hline
SafeForUnOp(f, ap) = \\
\quad \{\ x \in safe(ap) \mid ResultUnOp(f, ap)(x) \subseteq safe(ap)\ \} \\
SafeForBinOpFloat(f, ap) = \\
\quad \{\ (x, y) \in (safe(ap))^2 \mid ResultBinOpFloat(f, ap)(x, y) \subseteq safe(ap)\ \} \\
SafeForBinOpBool(f, ap) = \\
\quad \{\ (x, y) \in (safe(ap))^2 \mid ResultBinOpFloat(f, ap)(x, y) \subseteq safe(ap)\ \}
\end{array}
$$

A *UnOp* approximates a "paradigm" function to within some accuracy specification if it associates all operands with results that are acceptable approximations to that function. The same goes for *BinOpFloats* and *BinOpBools*. In particular, safe operands may not return an *overflow*; and if the "overflows" flag is true, unsafe operands must return either an approximately correct result or the *overflow* token.

$$
\begin{array}{|l}
Approx\,UnOp : UnOp \leftrightarrow (Float \to Float) \times AccurParam \\
Approx\,BinOpFloat : BinOpFloat \leftrightarrow (Float^2 \to Float) \times AccurParam \\
Approx\,BinOpBool : BinOpBool \leftrightarrow (Float^2 \to Float) \times AccurParam \\
\hline
op\ Approx\,UnOp\ (f, ap) \Leftrightarrow \\
\quad \forall\, x \in \mathrm{dom}\, op \bullet \\
\qquad (x \in SafeForUnOp(f, ap) \Rightarrow \\
\qquad\quad overflow \notin op(\!|\{\ x\ \}|\!) \\
\qquad\quad \wedge \\
\qquad\quad floats_of(op(\!|\{\ x\ \}|\!)) \subseteq ResultUnOp(f, ap)(x)) \\
\qquad \wedge \\
\qquad (ap.overflows = true \Rightarrow \\
\qquad\quad floats_of(op(\!|\{\ x\ \}|\!)) \subseteq ResultUnOp(f, ap)(x)) \\
op\ Approx\,BinOpFloat\ (f, ap) \Leftrightarrow \\
\quad \forall (x, y) \in \mathrm{dom}\, op \bullet \\
\qquad ((x, y) \in SafeForBinOpFloat(f, ap) \Rightarrow \\
\qquad\quad overflow \notin op(\!|\{\ (x, y)\ \}|\!) \\
\qquad\quad \wedge \\
\qquad\quad floats_of(op(\!|\{\ (x, y)\ \}|\!)) \subseteq ResultBinOpFloat(f, ap)(x, y)) \\
\qquad \wedge \\
\qquad (ap.overflows = true \Rightarrow \\
\qquad\quad floats_of(op(\!|\{\ (x, y)\ \}|\!)) \subseteq ResultBinOpFloat(f, ap)(x, y)) \\
op\ Approx\,BinOpBool\ (f, ap) \Leftrightarrow \\
\quad \forall (x, y) \in \mathrm{dom}\, op \bullet \\
\qquad ((x, y) \in SafeForBinOpBool(f, ap) \Rightarrow \\
\qquad\quad overflow \notin op(\!|\{\ (x, y)\ \}|\!) \\
\qquad\quad \wedge \\
\qquad\quad floats_of(op(\!|\{\ (x, y)\ \}|\!)) \subseteq ResultBinOpBool(f, ap)(x, y)) \\
\qquad \wedge \\
\qquad (ap.overflows = true \Rightarrow \\
\qquad\quad floats_of(op(\!|\{\ (x, y)\ \}|\!)) \subseteq ResultBinOpBool(f, ap)(x, y))
\end{array}
$$

We can now define the property *_has_accuracy_* as the assertion that each machine operation approximates the appropriate paradigm function to within the given accuracy parameters:

$$
\begin{array}{l}
imp\ has_accuracy\ ap \Leftrightarrow \\
\quad imp\ has_weak_acc\ ap \land \\
\quad imp.ma.plus\ ApproxBinOpFloat\,(+, ap) \land \\
\quad \ldots \\
\quad imp.ma.equals\ ApproxBinOpBool\,(=, ap) \land \\
\quad \ldots \\
\quad imp.ma.convert\ ApproxUnOp\,(\mathrm{id}\ Float, ap)
\end{array}
$$

3.5.4.7.2 Model-oriented attributes

To define *modeL_attr* we choose a particular maximal *AccurParam* satisfied by an implementation. (Note: The definitions below do not follow the definition in version 5.0, which is incorrect, but Ken Dritz's subsequent reworking of version 5.0.)

$$
\begin{array}{l}
best_mant : \mathbb{P}\,AccurParam \to \mathbb{P}\,AccurParam \\
best_emin : \mathbb{P}\,AccurParam \to \mathbb{P}\,AccurParam \\
best_first : \mathbb{P}\,AccurParam \to \mathbb{P}\,AccurParam \\
best_last : \mathbb{P}\,AccurParam \to \mathbb{P}\,AccurParam \\
best_ap : \mathbb{P}\,AccurParam \to \mathbb{P}\,AccurParam \\
\hline
best_ap(X) = \\
\quad best_last(best_first(best_emin(best_mant(X)))) \\
model_attr(imp) = \\
\quad \mu\ ap : AccurParam \bullet ap \in best_a p(ap' : AccurParam \mid imp\ has_accuracy\ ap')ap \in best_mant(X) \Leftrightarrow \\
\quad ap \in X \land \forall\, ap' : X \bullet ap.mantissa \geq ap'.mantissa \\
ap \in best_emin(X) \Leftrightarrow \\
\quad ap \in X \land \forall\, ap' : X \bullet ap.emin \leq ap'.emin \\
ap \in best_first(X) \Leftrightarrow \\
\quad ap \in X \land \forall\, ap' : X \bullet ap.sfirst \leq ap'.sfirst \\
ap \in best_last(X) \Leftrightarrow \\
\quad ap \in X \land \forall\, ap' : X \bullet ap.slast \geq ap'.slast
\end{array}
$$

There is, of course, exactly one *best_ap*:

$$\#best_ap(\{\ ap : AccurParam \mid imp\ has_accuracy\ ap\}) = 1$$

3.5.5 Array Types

The descriptor for an array type has the form $array_dsc(i, c)$, where $i : \mathrm{seq}\, Subtype$ is a sequence of discrete subtypes (the *index subtypes*) , and $c : Subtype$ is the component subtype.

$$
\begin{array}{l}
descriptor_values(array_dsc(i, c), S) = \\
\quad \{\ B : Index_bounds, v : (\mathrm{seq}\ Value) \nrightarrow subtype_values(c, S) \\
\qquad \mid Array(B, v) \in Value \land B = range_of_subtype \circ i \\
\qquad \bullet Array(B, v)\ \}
\end{array}
$$

3.5.5.1 String Types

String types are just particular array types.

3.5.6 Discriminants

Discriminants are specialized components of some composite types. We incorporate discriminants (which might be null) into the descriptor for every type that can have discriminants, in order to avoid a tedious duplication of definitions in similar cases.

This model does not account for access discriminants.

$$Discriminant_descriptor == Id \xrightarrow{a} Subtype \times optional\ Thunk$$

$$\begin{array}{|l} discriminant_values : Discriminant_descriptor \times State \rightarrow \mathbf{P}\ Binding \\ \hline discriminant_values(d, S) = \\ \quad \{ f : adom\ d \rightarrow Values \mid \forall n \in adom\ d \bullet f(n) \in subtype_values(first(d \cdot n), S) \} \end{array}$$

3.5.7 Record Types

It seems like a waste of effort to describe records with discriminants separately from records without discriminants, as there is a good deal of overlap in the two cases. Thus, we give every record type descriptor discriminants (which may be null).

Similarly, it seems like a big duplication of effort to describe tagged record types separately. Thus, we will give every record descriptor a tag (which may be null in the case of an untagged record).

$$\begin{array}{l} Component_list_descriptor == \\ \quad (Id \xrightarrow{a} Partial_subtype \times optional\ Thunk) \times optional\ Variant_descriptor \end{array}$$

A record type descriptor consists of a description of the discriminants (if any), and a component list description. The values of such a descriptor are records with fields for the discriminants, and fields for the other components. The subtypes of these latter fields (and even the exact fields present) may depend on a value of a discriminant.

$$\begin{array}{|l} descriptor_values(record_dsc(t, DA, CL), S) = \\ \quad \{ d, r : Binding \mid d \in discriminant_values(DA, \rho) \wedge r \in CL_values(CL, d, S) \\ \qquad\qquad\qquad \bullet record(t, d, r) \} \end{array}$$

A binding giving the values of the discriminants is given to function CL_values, so that the actual subtype of each component can be determined.

$$\begin{array}{|l} CL_values : Component_list_descriptor \times Binding \times State \rightarrow \mathbf{P}\ Binding \\ \hline CL_values((A, V), d, S) = \{f, v : Binding \mid \mathrm{dom} f = adom\ A\ \wedge \\ \qquad\qquad (\forall n \in \mathrm{dom} f \bullet f(n) \in Subtype_values(actualize_subtype(first(A \cdot n), d), S)) \\ \qquad\qquad v \in Variant_values(V, d, S) \\ \qquad \bullet f \cup v \} \end{array}$$

3.5.7.1 Variant Parts and Discrete Choices

A variant descriptor has the Id of the discriminant of the variant part, and a mapping from the possible values of this discriminant to component list descriptors.

$$Variant_descriptor == Id \times (Discrete_value \twoheadrightarrow Component_list_descriptor)$$

$$\begin{array}{|l} Variant_values : (optional\ Variant_descriptor) \times Binding \times State \rightarrow \mathbf{P}\ Binding \\ \hline Variant_values(none, d, S) = \{\varnothing\} \\ d(n) \in \mathrm{dom} f \Rightarrow Variant_values(some(n, f), d, S) = CL_values(f(d(n)), d, S) \\ d(n) \notin \mathrm{dom} f \Rightarrow Variant_values(some(n, f), d, S) = \{\varnothing\} \end{array}$$

3.5.8 Tagged Types and Type Extensions

The model for tagged types has not yet been developed.

3.5.8.1 Abstract Types and Subprograms

[AARM 3.9.3(8.a)] asserts that there are no values of an abstract type. But it is possible to have subprograms for such subtypes, and for non-abstract descendents to inherit them. If we want to say something about the meanings of such subprograms, we probably need to talk about the values of the parameters. (In some sense, these are parameters of type T'Class.)

On the whole, it probably seems easiest to use a model of values as though the type were not abstract; we can also treat abstract subprograms as though their bodies raised **Program_Error**.

3.5.9 Access Types

The descriptor for an access type has the form $access_dsc(s, m)$, where $s : Subtype_location$ describes the designated subtype of the access type, and $m : Access_mode$ describes the access mode.

$$Access_Mode ::= constant_access \mid all_access \mid pool_access$$

A value of an access-to-object type is either a null value, or a view of an object of the designated subtype:

$$\begin{aligned}
&descriptor_values(access_dsc(s, m), S) = \\
&\quad \{access_value(none)\} \cup \\
&\quad \{\ access_value(some(object_view(l, s'))) \mid \\
&\qquad the_store(S).objects(l) \in subtype_values(the_store(S).subtypes(s), S)\ \}
\end{aligned}$$

3.5.9.1 Incomplete Type Declarations

The descriptor *incomplete* describes an incomplete type. When the full type definition for the type is elaborated, the type environment is updated to reflect the appropriate descriptor for the type.

Note that there is an issue about the first subtype of an incomplete type; it is constrained if there is no discriminant part. However, the first subtype corresponding to the full definition may be unconstrained. (See comment 94-3901.c.)

There are no values of an incomplete type.

3.6 States

A *state* combines a store, an external state, and control flow information (in the usual cases), or is *erroneous* or *unpredicted*. We use the state *unpredicted* in those cases where our semantic definition, in the interests of simplicity, makes no prediction about the effect of a program (even though the Standard defines the effect, or calls it *implementation-defined*).

$$\begin{aligned}
State ::= &\ normal\langle\langle Store\rangle\rangle \\
\mid &\ exception\langle\langle ExceptionId \times Store\rangle\rangle \\
\mid &\ exit\langle\langle Loop_Id \times Store\rangle\rangle \\
\mid &\ proc_return\langle\langle Store\rangle\rangle \\
\mid &\ func_return\langle\langle Value \times Store\rangle\rangle \\
\mid &\ intermediate\langle\langle Store\rangle\rangle \\
\mid &\ erroneous \\
\mid &\ unpredicted
\end{aligned}$$

Function *the_store* gives the store associated with a state.

Chapter 4

Judgements

This section summarizes the domains and judgements used in the definition of Ada 9X semantics. The details of the domain definitions are given in Chapter 3. The formal definitions of these judgements are given in later sections.

4.1 Domains

The definition is based on the concepts, domains and support functions, introduced in Chapter 3. Specifically, it uses the domains listed in Table 4.1

All of these domains are generated by term algebras (subject to the constraints defined in Chapter 3). Constructors for these domains are given below. In addition, the domains of component associations and environments are defined as follows:

$$\begin{aligned} \mathit{CompAssoc} &= \mathrm{list}((\mathit{Id} \times (\mathit{Subtype} \times \mathrm{optional}(\mathit{Thunk})))) \\ \mathit{Environment} &= (\mathit{Id} \twoheadrightarrow \mathit{View}) \end{aligned}$$

Also the signatures for all functions (other than contructors) and predicates are given. Note that these signatures are those used in the Prolog representation of judgements and may differ from those given in Chapter 3.

4.1.1 Types

The structure of a type is given by a type descriptor of the form:

type *Type*

- $\mathit{enum_type} : \mathit{integer} \rightarrow \mathit{Type}$
- $\mathit{modular_type} : \mathit{integer} \rightarrow \mathit{Type}$
- $\mathit{signed_integer_type} : \mathit{integer} \times \mathit{integer} \times \mathit{integer} \times \mathit{integer} \rightarrow \mathit{Type}$
- $\mathit{universal_integer_type} : \mathit{integer} \times \mathit{integer} \rightarrow \mathit{Type}$
- $\mathit{array_type} : \mathrm{list}(\mathit{Subtype}) \times \mathit{Subtype} \rightarrow \mathit{Type}$
- $\mathit{class_type} : \mathit{Type_location} \rightarrow \mathit{Type}$
- $\mathit{access_type} : \mathit{Subtype_location} \times \mathit{Access_modifier} \rightarrow \mathit{Type}$
- $\mathit{func_profile} : (\mathit{Id} \xrightarrow{a} \mathit{Parameter}) \times \mathit{Subtype} \rightarrow \mathit{Type}$
- $\mathit{proc_profile} : (\mathit{Id} \xrightarrow{a} \mathit{Parameter}) \rightarrow \mathit{Type}$
- $\mathit{incomplete_type} : \mathit{Discriminant} \rightarrow \mathit{Type}$
- $\mathit{record_type} : \mathrm{optional}(\mathit{Type_location}) \times \mathit{Discriminant} \times \mathit{Record_fields} \rightarrow \mathit{Type}$

Access_modifier	access (type) modifier
Action	executable actions
Attributes	subtype attributes
Bool	truth values (non-Ada)
Choice	choices
Constraint	constraints
Discriminant	discriminants
ExId	internal names for exception identifiers
Record_fields	record fields including variants
LValue	L-values (addresses)
LoopId	internal names for loop identifiers
Mode	parameter modes
Object_location	addresses of top-level objects
Object_mode	indication of constancy and aliasing
Partial_constraint	partial constraints
Partial_subtype	partial subtypes
Partial_value	partial values
Parameter	formal parameters
Range	discrete and real ranges
State	state of a computation
Store	model of memory
Subprogram	subprogram values
Subprogram_label	unique tags for subprogram access values
Subprogram_location	addresses for subprogram values
Subtype	subtypes
Subtype_location	addresses for subtypes
Thunk	an expression with its declaration environment
Type	type descriptors
Type_location	address space for type information
Value	runtime values
Variant	variants
View	views

Table 4.1: Domains used in the judgements

Descriptors of this form are stored in the state and are accessed by unique addresses of sort *Type_location*. A derived type is represented by a new name for an existing type descriptor. The use of type names has the advantage that it is easy to deal with cyclic types and type completion. It has the disadvantage that types can be understood only in the context of a state.

The domain *Type_location* contains constants that represent the predefined types of the language:

type *Type_location*
boolean_tn : *Type_location*
character_tn : *Type_location*
universal_integer_tn : *Type_location*
universal_real_tn : *Type_location*
root_integer_tn : *Type_location*

In the case of tagged types, values of *Type_location* are used as unique tags. The optional *Type_location* component of a record type descriptor defines the tag of the parent type. Values of sort *Type_location* are related by the *ancestor* relation which represents both the derivation and class hierarchy:

ancestor(*State*, *Type_location*, *Type_location*)
descendant(*State*, *Type_location*, *Type_location*)
ultimate_ancestor(*State*, *Type_location*, *Type_location*)

Access modifiers are used in the descriptors of access types with the obvious meaning.

type *Access_modifier*
constant_access : *Access_modifier*
all_access : *Access_modifier*
pool_access : *Access_modifier*

Discriminants are represented as follows:

type *Discriminant*
discr : $(Id \xrightarrow{a} (Subtype \times \text{optional}(Thunk))) \rightarrow Discriminant$
$\langle\rangle : Discriminant$

Two discriminants can be combined using:

$$\text{discriminant_union} : Discriminant \times Discriminant \rightarrow Discriminant$$

A thunk represents an expression together with its declaration environment.

type *Thunk*
thunk : $Environment \times \text{Exp} \rightarrow Thunk$

A component list represents *actualized* record fields, i.e., record fields that do not contain partial information that depends on discriminant values.

type $CompAssoc = \text{list}((Id \times (Subtype \times \text{optional}(Thunk))))$

The fields of a, possibly discriminated, record type are represented by the domain

type *Record_fields*
fields : $(Id \xrightarrow{a} (Partial_subtype \times \text{optional}(Thunk))) \times \text{optional}((Id \times Variant)) \rightarrow Record_fields$

which includes the proper fields (maybe partial) as well as any variant.

Component associations can be constructed from discriminant values and partial component lists or variants:

$actualized_complist : (Id \rightarrow\!\!\!\!\rightarrow Value) \times \text{list}((Id \times (Partial_subtype \times \text{optional}(Thunk)))) \rightarrow CompAssoc$
$actualized_components : (Id \rightarrow\!\!\!\!\rightarrow Value) \times Record_fields \rightarrow CompAssoc$
$actualized_variants : (Id \rightarrow\!\!\!\!\rightarrow Value) \times (Id \times Variant) \rightarrow CompAssoc)$
$append_components : CompAssoc \times CompAssoc \rightarrow CompAssoc$

A variant part is represented by a pair $(Id \times Variant)$ where the identifier specifies the name of the discriminant and the second component represents the actual fields:

type $Variant$
$variant : \text{list}((Choice \times Record_fields)) \rightarrow Variant$
$\langle\rangle : Variant$

Given a discriminant value, the actual record fields of a variant are defined by the predicate

$$the_variant(Variant, Value, Record_fields).$$

Variant parts are combined using function

$$variant_union : \text{optional}(Id \times Variant) \times \text{optional}(Id \times Variant) \rightarrow \text{optional}(Id \times Variant).$$

The representation of subprogram access types uses parameter descriptors of the form:

type $Parameter$
$formal : Mode \times Subtype \times \text{optional}(Thunk) \rightarrow Parameter$

where modes are given by:

type $Mode$
$in_mode : Mode$
$out_mode : Mode$
$in_out_mode : Mode$

The following predicates define the set of values of a given type (descriptor):

$cl_value(State, Record_fields, (Id \rightarrow\!\!\!\!\rightarrow Value), (Id \rightarrow\!\!\!\!\rightarrow Value))$
$discriminant_value(State, Discriminant, (Id \rightarrow\!\!\!\!\rightarrow Value))$
$variant_values(State, \text{optional}((Id \times Variant)), (Id \rightarrow\!\!\!\!\rightarrow Value), (Id \rightarrow\!\!\!\!\rightarrow Value))$
$descriptor_value(State, Type, Value)$

4.1.2 Values

The representation of values is straightforward using the definition

type $Value$
$invalid_val : Value$
$discrete_val : integer \rightarrow Value$
$real_val : real \rightarrow Value$
$access_val : View \times \text{optional}(Subprogram_label) \rightarrow Value$
$null : Value$
$record_val : \text{optional}(Type_location) \times (Id \rightarrow\!\!\!\!\rightarrow Value) \times (Id \rightarrow\!\!\!\!\rightarrow Value) \rightarrow Value$
$array_val : \text{list}(Range) \times (\text{list}(Value) \rightarrow\!\!\!\!\rightarrow Value) \rightarrow Value$

The term *access_val*(...) is used for access to object and access to subprogram values. In the latter case, each time a subprogram access is computed a new unique subprogram label

type *Subprogram_label*

is generated. This label is needed to properly model equality of access to subprogram values.

Note that access values are views. This representation is used to determine the actual subtype of an access value.

Ranges are represented as pairs of values in the obvious way.

type *Range*
discrete_rng : *integer* × *integer* → *Range*
real_rng : *real* × *real* → *Range*

The following functions are defined for ranges.

low_bound : *Range* → *Value*
high_bound : *Range* → *Value*
make_range : *Value* × *Value* → *Range*
base_range : *State* × *Type* → *Range*
indices : list(*Range*) → *set*(list(*Value*))

The latter function defines the set of all index vectors that fit a list of (index) ranges.

Sets of (scalar) values are represented using the domain

type *Choice*
choice_range : *Range* → *Choice*
choice_value : *Value* → *Choice*
choice_lst : list(*Choice*) → *Choice*
choice_default : *Choice*

The following are predicates on values

access_value(*Value*)
array_value(*Value*)
belongs_to(*Value*, *Range*)
composite_value(*Value*)
covers(*Value*, *Choice*, *Bool*)
discrete_value(*Value*)
elementary_value(*Value*)
real_value(*Value*)
satisfies(*Value*, *Constraint*)
scalar_value(*Value*)

4.1.3 Subtypes

The definition of subtypes follows the description provided by the Reference Manual:

type *Subtype*
subtype : *Type_location* × *Constraint* × *Attributes* → *Subtype*

Access to the underlying type is indirect through a type location. The present version of the semantics does not use any subtype attributes.

type *Attributes*
not_used : *Attributes*

Functions defined for subtypes are

$$\begin{array}{l} range_of_subtype : State \times Subtype \rightarrow Range \\ ranges_of_subtypes : State \times \text{list}(Subtype) \rightarrow \text{list}(Range) \\ type_struct : State \times Subtype \rightarrow Type \end{array}$$

Constraints are

type *Constraint*
$no_constraint : Constraint$
$range_constraint : Range \rightarrow Constraint$
$index_constraint : \text{list}(Range) \rightarrow Constraint$
$discriminant_constraint : (Id \nrightarrow Value) \rightarrow Constraint$
$indirect_index_constraint : \text{list}(Range) \rightarrow Constraint$
$indirect_discriminant_constraint : (Id \nrightarrow Value) \rightarrow Constraint$

In the case of discriminated record types, component subtypes can be *partial* if they depend on discriminant values. This leads to the following definitions of partial counterparts of values, constraints, and subtypes.

type *Partial_value*
$p_value : Value \rightarrow Partial_value$
$discriminant_ref : Id \rightarrow Partial_value$

type *Partial_constraint*
$p_no_constraint : Partial_constraint$
$p_range_constraint : (Partial_value \times Partial_value) \rightarrow Partial_constraint$
$p_index_constraint : \text{list}((Partial_value \times Partial_value)) \rightarrow Partial_constraint$
$p_discriminant_constraint : (Id \nrightarrow Partial_value) \rightarrow Partial_constraint$
$p_indirect_index_constraint : \text{list}((Partial_value \times Partial_value)) \rightarrow Partial_constraint$
$p_indirect_discriminant_constraint : (Id \nrightarrow Partial_value) \rightarrow Partial_constraint$

type *Partial_subtype*
$p_subtype : Subtype \times Partial_constraint \rightarrow Partial_subtype$

Given a discriminant constraint, partial entities can be *actualized*. This process is defined by the functions:

$$\begin{array}{l} actualized_partial_range : (Id \nrightarrow Value) \times (Partial_value \times Partial_value) \rightarrow Range \\ actualized_range_list : (Id \nrightarrow Value) \times \text{list}((Partial_value \times Partial_value)) \rightarrow \text{list}(Range) \\ actualized_value : (Id \nrightarrow Value) \times Partial_value \rightarrow Value \\ actualized_constraint : (Id \nrightarrow Value) \times Partial_constraint \rightarrow Constraint \\ actualized_binding_list : binding(Id \nrightarrow Value) \times \text{list}((Id \times Partial_value)) \rightarrow \text{list}((Id \times Value)) \end{array}$$

The following predicates on subtypes define the taxonomy of RM 3.2:

is_access_to_object_type(*State*, *Subtype*)
is_access_to_subprogram_type(*State*, *Subtype*)
is_access_type(*State*, *Subtype*)
is_array_type(*State*, *Subtype*)
is_boolean_type(*State*, *Subtype*)
is_by_copy_type(*State*, *Subtype*)
is_by_reference_type(*State*, *Subtype*)
is_character_type(*State*, *Subtype*)
is_composite_type(*State*, *Subtype*)
is_discrete_type(*State*, *Subtype*)
is_elementary_type(*State*, *Subtype*)
is_enumeration_type(*State*, *Subtype*)
is_integer_type(*State*, *Subtype*)
is_modular_integer_type(*State*, *Subtype*)
is_protected_type(*State*, *Subtype*)
is_real_type(*State*, *Subtype*)
is_record_type(*State*, *Subtype*)
is_scalar_type(*State*, *Subtype*)
is_signed_integer_type(*State*, *Subtype*)
is_string_type(*State*, *Subtype*)
is_tagged_type(*State*, *Subtype*)
is_task_type(*State*, *Subtype*)

Other predicates used in the semantics are

component_type((*Id* ⇸ *Value*), *Type*, *Id*, *Subtype*)
convert_return_value(*State*, *View*, *Subtype*, *View*)
range_constraints(*State*, list(*Subtype*), list(*Range*))
select_component_type((*Id* ⇸ *Value*), *Record_fields*, *Id*, *Subtype*)
subtype_value(*State*, *Subtype*, *Value*)
test_in(*State*, *Value*, *Subtype*)
type_constraint(*Environment*, *Subtype*, *Constraint*, *Type*)
null_range(*Range*)
included_in(*Range*, *Range*)
values_in_range(*Range*, *set*(*Value*))
discrete_range(*Range*)
real_range(*Range*)

Finally, the following predicates describe state transformers related to various checks and conversions related to subtypes.

slice_check(*State*, *Range*, *Range*, *State*)
view_convert(*State*, *Environment*, *Subtype*, *View*, *View*, *State*)
subtype_convert(*State*, *Environment*, *Subtype*, *Value*, *Value*, *State*)
index_list(*State*, list(*Range*), list(*Value*), list(*Value*), *State*)
return_check(*State*, *Subtype*, *State*)

4.1.4 Environments and Views

Environments map identifiers to views.

type *Environment* = (*Id* ⇸ *View*)

An environment is updated using the usual syntax for bindings.

$$_[_ \mapsto _] : \mathit{Environment} \times \mathit{Id} \times \mathit{View} \rightarrow \mathit{Environment}$$

Note that identifiers are assumed to be unique and that all overloading and qualified names have been resolved to unique identifiers.

The lookup of an identifier I in an environment E is written as

$$E \vdash_{lookup} I \Rightarrow V$$

A view describes entities denoted by identifiers as follows:

type *View*
- $\mathit{object_view} : \mathit{LValue} \times \mathit{Subtype} \times \mathit{Object_mode} \rightarrow \mathit{View}$
- $\mathit{loop_view} : \mathit{Id} \rightarrow \mathit{View}$
- $\mathit{constant_view} : \mathit{Value} \rightarrow \mathit{View}$
- $\mathit{subtype_view} : \mathit{Subtype_location} \rightarrow \mathit{View}$
- $\mathit{subprogram_view} : \mathit{Subprogram_location} \times (\mathit{Id} \xrightarrow{a} \mathit{Parameter}) \times \mathrm{optional}(\mathit{Subtype}) \rightarrow \mathit{View}$
- $\mathit{undefined_view} : \mathit{View}$

Object views include a mode description of the form:

type *Object_mode*
- $\mathit{variable_object} : \mathit{Object_mode}$
- $\mathit{constant_object} : \mathit{Object_mode}$
- $\mathit{aliased_object} : \mathit{Object_mode}$

Whether or not an object may be assigned to depends on its mode:

$$\mathit{assignable}(\mathit{Object_mode})$$

Loop views are used in defining exit from named and unnamed loops.

type *LoopId*
- $\mathit{unnamed} : \mathit{LoopId}$
- $\mathit{loop_id} : \mathit{Id} \rightarrow \mathit{LoopId}$

The following predicate defines how the bindings of a parameter association are added to an environment creating the environment in which a subprogram body is executed. Note that the names used in the given parameter association may differ from the names of the formal parameters (due to renaming). The names of the formals are provided by a separate argument.

$$\mathit{bind_actuals}(\mathit{Environment}, (\mathit{Id} \xrightarrow{a} \mathit{View}), \mathrm{list}(\mathit{Id}), \mathit{Environment})$$

The following constants of sort *ExId* denote the language-defined exceptions (others may be added).

type *ExId*
- $\mathit{constraint_error} : \mathit{ExId}$
- $\mathit{program_error} : \mathit{ExId}$
- $\mathit{ex_id} : \mathit{Id} \rightarrow \mathit{ExId}$

New unique names for user-defined exceptions are introduced by static semantics.

4.1.5 Memory Model

Values of sort *Store* describe the current binding of addresses to Ada run-time values (of sort *Value*). In our model, there are four different types of addresses (locations). They include type location (see above) as well as locations for objects, subprograms and subtypes.

type *Object_location*
loc : *integer* $\rightarrow$ *Object_location*

type *Subprogram_location*

type *Subtype_location*

Object locations are associated only with objects that are not components of other objects. Components are specified by the address of the containing object together with a selector sequence. An address together with a selector sequence is a L-value (sort *LValue*).

type *LValue*
location : *Object_location* $\rightarrow$ *LValue*
array_component : *LValue* $\times$ list(*Value*) $\rightarrow$ *LValue*
array_slice : *LValue* $\times$ *Range* $\rightarrow$ *LValue*
record_component : *LValue* $\times$ *Id* $\rightarrow$ *LValue*

Thus, the domain of stores is defined as a 4-tuple as follows:

type *Store*
$\langle _, _, _, _ \rangle$: $(\mathit{Object_location} \nrightarrow \mathit{Value}) \times$
$(\mathit{Subprogram_location} \nrightarrow \mathit{Subprogram}) \times$
$(\mathit{Type_location} \nrightarrow (\text{optional}(\mathit{Type_location}) \times \mathit{Type})) \times$
$(\mathit{Subtype_location} \nrightarrow \mathit{Subtype}) \rightarrow \mathit{Store}$

Individual components of a store can be updated using the following notation:

$_[_ \mapsto_1 _] : \mathit{Store} \times \mathit{Object_location} \times \mathit{Value} \rightarrow \mathit{Store}$
$_[_ \mapsto_2 _] : \mathit{Store} \times \mathit{Subprogram_location} \times \mathit{Subprogram} \rightarrow \mathit{Store}$
$_[_ \mapsto_3 _] : \mathit{Store} \times \mathit{Type_location} \times (\text{optional}(\mathit{Type_location}) \times \mathit{Type}) \rightarrow \mathit{Store}$
$_[_ \mapsto_4 _] : \mathit{Store} \times \mathit{Subtype_location} \times \mathit{Subtype} \rightarrow \mathit{Store}$

Values of sort *State* describe a current store, together with the current status of program execution. A state may represent the propagation of an exception, exit from a subprogram, or exit from a loop.

type *State*
exception : *ExId* $\times$ *Store* $\rightarrow$ *State*
exit : *LoopId* $\times$ *Store* $\rightarrow$ *State*
proc_return : *Store* $\rightarrow$ *State*
func_return : *View* $\times$ *Store* $\rightarrow$ *State*
normal : *Store* $\rightarrow$ *State*

The following functions are defined to access and manipulate the store embedded in a state.

$_[_ \mapsto_1 _] : State \times Object_location \times Value \rightarrow State$
$_[_ \mapsto_2 _] : State \times Subprogram_location \times Subprogram \rightarrow State$
$_[_ \mapsto_3 _] : State \times Type_location \times (\text{optional}(Type_location) \times Type) \rightarrow State$
$_[_ \mapsto_4 _] : State \times Subtype_location \times Subtype \rightarrow State$
$_^{obj}[_] : State \times Object_location \rightarrow Value$
$_^{spg}[_] : State \times Subprogram_location \rightarrow Subprogram$
$_^{typ}[_] : State \times Type_location \rightarrow (\text{optional}(Type_location) \times Type)$
$_^{stp}[_] : State \times Subtype_location \rightarrow Subtype$
$make_state : State \times Store \rightarrow State$
$the_store : State \rightarrow Store$

The allocation of new location of the four different kinds is defined by the predicates:

new_type(*State*, (optional(*Type_location*) × *Type*), *Type_location*, *State*)
new_subtype(*State*, *Subtype*, *Subtype_location*, *State*)
new_object(*State*, *Value*, *Object_location*, *State*)
new_subprogram(*State*, *Subprogram*, *Subprogram_location*, *State*)

For a given state and L-value, the following predicate defines the current value. The definition of this predicate includes access of the appropriate subobjects denoted by an L-value.

content(*State*, *LValue*, *Value*)

States are classified as normal or abnormal. Abnormal states will, in general, alter the control flow.

abnormal_state(*State*)
normal_state(*State*)

The return from a subprogram may be indicated by an abnormal state. The following predicates deal with the cases of procedure and function returns, respectively.

proc_exit(*State*, *State*)
return_value(*State*, *View*, *State*)

Values stored in the subprogram component of a store are of the form:

type *Subprogram*
subprogram : *Environment* × list(*Id*) × *Dcl* × *Stm* → *Subprogram*
operator : *Operator* → *Subprogram*
unelaborated : *Subprogram*

A user-declared subprogram is represented by the declaration environment, the names of the formal parameters and the declarations and statements that comprise the body. Predefined operators of the language are enumerated by a domain *Operator*. The definition of the semantics of the operators is not covered. An attempt to execute a subprogram of the form *unelaborated* will raise program error.

4.1.6 Other Predicates

The following predicates deal with the selection of elements from parameter lists and record component associations.

the_parameter(list(*Pss*), *Id*, *Nam*)
given_parameter(list(*Pss*), *Id*, *Exp*)
find_component(*Id*, list(*Rca*), *Exp*)

4.2 Judgements

Judgements for various syntactic domains define the effect of the elaboration or execution of language phrases of this domain. In most cases, the effect of a phrase depends on the current state and the current environment. The effect typically consists of a change in state and the possible return of a result. Depending on the kind of phrase, the result may be a value, a type, a new environment and so on. The meaning of some phrases depends on additional context beyond the state and the environment. For instance, the meaning of a type definition depends on the discriminant (which is part of the enclosing type declaration).

The general form of a judgement is

$$\textit{state}, \textit{environment}, \textit{context} \vdash \textit{language-phrase} \Rightarrow \textit{result}, \textit{new-state}$$

The following is a list of the judgements used in the definition. For each syntactic domain, the signature of the judgement is given together with an informal rationale.

If the evaluation of a construct raises an exception then the final state represents this information. The propagation of exceptions is described as part of the sequencing of actions described below.

The following conventions are used throughout the description of judgements:

S_1 : *State*	–	The initial state.
E : *Environment*	–	The environment.
S_2 : *State*	–	The final state.

4.2.1 Declarations

The effect of elaborating a (sequence of) declaration(s) is to add bindings corresponding to the newly introduced identifiers to the environment. The elaboration of a declaration may also affect the state.

$$S_1, E_1 \vdash_{dcl} \mathsf{Dcl} \Rightarrow E_2, S_2$$

E_1 : *Environment*	–	The initial environment
Dcl : *Dcl*	–	A declaration.
E_2 : *Environment*	–	A possibly modified environment.

4.2.2 Parameter lists

Elaboration of subprogram declarations is defined in terms of the judgements

$$S_1, E, \vdash_{pas} \mathsf{Pas} \Rightarrow A, S_2$$

Pas : Pms^*	–	Formal parameter list.
A : $Id \xrightarrow{a} Parameter$	–	Parameter signature.

and

$$\vdash_{mod} \mathsf{Mod} \Rightarrow M$$

Mod : *Mde*	–	Parameter mode.
M : *Mode*	–	Mode representation.

The latter judgement uses neither states nor the environment.

4.2.3 Type Definitions

A type definition is elaborated in the context of a (possible) discriminant association. The result is the first-named subtype. Elaborating a type definition may affect the state and the environment because subexpressions may have side-effects and new internal type names may be bound in the environment.

$$S_1, E_1, D \vdash_{tdf} \mathsf{Tdf} \Rightarrow T, E_2, S_2$$

D : *Discriminant*	–	The discriminant.
Tdf : *Tdf*	–	A type definition.
T : *Subtype*	–	The resulting subtype.
E_2 : *Environment*	–	The modified environment.

4.2.4 Variant Parts

$$S_1, E, D \vdash_{vrn} \mathsf{Vrn} \Rightarrow V, S_2$$

D : *Discriminant*	–	A discriminant.
Vrn : Vnt^*	–	A list of variant clauses.
V : *Variant*	–	The resulting variant.

The judgement always uses a discriminant that may be empty.

4.2.5 Discriminant Parts

The evaluation of a discriminant part results in an association that maps discriminant identifiers into their subtype and optional initialization. As with record component lists, the initialization is represented by a thunk.

$$S_1, E \vdash_{dsc} \mathsf{Dcp} \Rightarrow D, S_2$$

Dcp : *Dcp*	–	A discriminant part.
D : *Discriminant*	–	The resulting association.

4.2.6 Component Lists

The result of evaluating a record component list is an association that maps each component identifier into a pair consisting of the component's subtype and an optional initialization. The initialization is given by a *thunk* that represents the initialization expression, together with the environment in which this expression is to be evaluated.

$$S_1, E \vdash_{cmp} \mathsf{Cmp} \Rightarrow C, S_2$$

Cmp : *Cmp*	–	A component list.
C : *CompAssoc*	–	The result.

4.2.7 Subtype Indications

There are three flavors of judgements dealing with subtype indications. The normal case is covered by the declaration:

$$S_1, E \vdash_{sif} \mathsf{Sid} \Rightarrow T, S_2$$

Sid : *Sid*	–	A subtype indication.
T : *Subtype*	–	The denoted subtype.

A special case is needed for subtype indications that appear inside discriminated records because the result will be a partial subtype.

$$S_1, E_1, D \vdash_{psi} \mathsf{Sid} \Rightarrow T, E_2, S_2$$

D : *Discriminant*	–	A discriminant.
Sid : *Sid*	–	A subtype indication.
T : *Partial_subtype*	–	The denoted partial subtype.

Finally, a third form of this judgement is needed to deal with access types to allow references to incomplete types. Instead of a subtype, this judgement returns a subtype location.

$$S_1, E \vdash_{sif_i} \mathsf{Sid} \Rightarrow L, S_2$$

Sid : *Sid*	–	A subtype indication.
L : *Subtype*	–	A subtype location.

A completing type declaration will initialize the subtype location.

4.2.8 Statements

The execution of a statement only affects the state and has no result.

$$S_1, E \vdash_{stm} \mathsf{Stm} \Rightarrow S_2$$

Stm : *Stm*	–	A statement.

4.2.9 Elsif Clauses

An elsif-clause consists of a condition and a sequence of statements. The judgement for this construct describes the evaluation of the condition followed by the conditional execution of the statements. There is a boolean result that indicates whether the condition was true. This result is used in the definition of cascaded elsif clauses.

$$S_1, E \vdash_{elf} \mathsf{Eif} \Rightarrow B, S_2$$

Eif : *Eif*	–	An elsif clause.
B : *Bool*	–	The boolean result.

4.2.10 Case Alternatives

The evaluation of a list of case alternatives depends on the value of the case selector:

$$S_1, E, V \vdash_{clt} \mathsf{Alt} \Rightarrow B, S_2$$

V : *Value*	–	The value of the case selector.
Alt : *Alt*	–	A sequence of case alternatives.
B : *Bool*	–	True, if one of the alternatives has matched the case selector.

The boolean result indicates whether one of the alternatives has been executed. The execution of the enclosing case statement will have to terminate in an exception if this result is false.

4.2.11 Discrete Choice Lists

Discrete choice lists are used in variant, array aggregates and case statements. The following judgement describes the evaluation of a list of choices. The result is a representation of the choices.

$$S_1, E \vdash_{chc} \mathsf{Dch} \Rightarrow C, S_2$$

Dch : Dch^*	–	A discrete choice list.
C : *Choice*	–	The representation of the choice list.

4.2.12 Expressions

The evaluation of an expression results in a value and possible side-effects. In certain cases the meaning of an expression depends on the expected type (e.g., the evaluation of aggregates and string literals). Rather than adding this type information to the judgement, the abstract syntax provides such information where necessary.

$$S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2$$

Exp : *Exp*	–	An expression.
V : *Value*	–	The resulting value.

The judgement for conditions differs from that for expressions by returning a truth value.

$$S_1, E \vdash_{cnd} \mathsf{Cnd} \Rightarrow V, S_2$$

Cnd : *Cnd*	–	A condition.
B : *Bool*	–	A truth value.

4.2.13 Names

The evaluation of a name results in a view of the named entity.

$$S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow W, S_2$$

N : *Nam*	–	A name.
W : *View*	–	The view denoted by the name.

The evaluation of certain kinds of names cannot have side-effects (e.g., subtype indications). Rather than defining a separate judgement for this case the definition will require that the initial and final states are identical in these cases.

4.2.14 Ranges

The following judgement defines the evaluation of ranges. In the abstract syntax ranges include discrete subtype definitions. The judgement also deals with the definition of the range attribute.

$$S_1, E \vdash_{rng} \mathsf{Rng} \Rightarrow R, S_2$$

Rng : *Rng*	–	A range or discrete subtype definition.
R : *Range*	–	The resulting range value.

4.2.15 Record Aggregates

The following judgement defines the value of a record or extension aggregate.

$$S_1, E, C \vdash_{aggr} \mathsf{Agg} \Rightarrow V, S_2$$

C : *CompAssoc*	–	The expected components of the aggregate.
Agg : *Agg*	–	The aggregate.
V : *Id* ⇸ *Value*	–	The resulting binding.

The expected type for the aggregate is provided in the abstract syntax by allowing only qualified aggregates. Qualification is added by static analysis where necessary.

4.2.16 Array Aggregates

A separate judgement is used for array aggregates. It has an additional sequence of index ranges as context.

$$S_1, E, R, T \vdash_{aag} \mathsf{Agg} \Rightarrow V, S_2$$

R : list(*Range*)	–	The index ranges of the aggregate.
T : *Subtype*	–	The type of the elements.
Agg : *Agg*	–	An array aggregate.
V : *Value*	–	The array value.

4.2.17 Attributes

The following judgement defines the values of (parameterless) attributes as defined in Appendix A of the Reference Manual.

$$S_1, E, W_1 \vdash_{att} \mathsf{Id} \Rightarrow W_2, S_2$$

W_1 : *View*	–	A view.
Id : *Id*	–	The attribute name.
W_2 : *View*	–	The view of the attribute.

All core language-defined attributes are free of side-effects. This means that the final state will always equal the initial state. Note that some attributes return a subprogram view which, when called, may have a side-effect or raise an exception. But the effect of such calls is not part of evaluating the attribute itself.

In the case of the range attribute, the signature differs as follows:

$$S_1, E, W_1 \vdash_{att_r} \mathsf{range} \Rightarrow R, S_2$$

W : *View*	–	A view.
R : *Range*	–	A range.

4.3 Actions

A sequence of statements is executed by sequentially executing each statement in the sequence. Execution is abandoned if one of the statements raises an exception or causes some other change in the flow of control. In the case of expressions, the language specifies that, in certain cases, several expressions are evaluated in arbitrary order.

In order to systematically define different kinds of order of execution of program parts, the notion of an action is introduced. An action can be viewed as a representation of a judgement without the state. Consider, for instance, the judgement for statements:

$$S_1, E \vdash_{stm} \mathsf{Stm} \Rightarrow S_2$$

The corresponding action is a term

$$statement_fn(E, \mathsf{Stm})$$

that represents the environment and the statement. For convenience, we shall write actions just like judgements with the initial and final state omitted. In this case, the action is written as

$$E \vdash_{stm} \mathsf{Stm} \Rightarrow$$

Given an action A, it is meaningful to talk about the effect of executing the action in a given state S_0. This is expressed by the predicate *run*:

$$run(S_0, A, S_1)$$

holds if and only if the execution of action A in state S_0 results in state S_1.

Using *run* it is possible to define different orders of evaluation of sets and sequences of actions.

The notation

$$S_0 \left[\begin{array}{c} A_1 \\ \ldots \\ A_n \end{array} \right] S_1$$

means that the actions A_1 through A_n are to be executed sequentially starting in state S_0 and leading to S_1. The definition of this notation needs to consider the propagation of exceptions by any of the actions.

Similarly, the notation

$$S_0 \left\{ \begin{array}{c} A_1 \\ \ldots \\ A_n \end{array} \right\} S_1$$

means that the actions A_1 through A_n are to be executed in arbitrary order.

The set of actions (sort *Action*) is given by the following terms. There is one action constructor for each judgement.

type *Action*

$then : \text{list}(Action) \rightarrow Action$
$new_object_fn : Value \times Object_location \rightarrow Action$
$attribute_fn : Environment \times View \times Id \times View \rightarrow Action$
$param_attribute_fn : Environment \times View \times Id \times Value \times View \rightarrow Action$
$case_alternative_fn : Environment \times Value \times Alt \times Bool \rightarrow Action$
$choices_fn : Environment \times Dch_lst \times Choice \rightarrow Action$
$component_list_fn : Environment \times Discriminant \times Cmp_lst \times (Id \xrightarrow{a} (Partial_subtype \times \text{optional}(Thunk))) \rightarrow Action$
$constraint_fn : Environment \times Subtype \times Cns \times Constraint \rightarrow Action$
$compatible_fn : Environment \times Subtype \times Constraint \rightarrow Action$
$declaration_fn : Environment \times Dcl \times Environment \rightarrow Action$
$default_value_fn : Environment \times Subtype \times Value \rightarrow Action$
$discr_assoc_fn : Environment \times Subtype \times Dca \times Constraint \rightarrow Action$
$discriminant_part_fn : Environment \times Dcp \times Discriminant \rightarrow Action$
$elsif_clause_fn : Environment \times Eif \times Bool \rightarrow Action$
$expression_fn : Environment \times Exp \times Value \rightarrow Action$
$name_fn : Environment \times Nam \times View \rightarrow Action$
$new_type_fn : (\text{optional}(Type_location) \times Type) \times Type_location \rightarrow Action$
$range_fn : Environment \times Rng \times Range \rightarrow Action$
$subtype_indication_fn : Environment \times Sid \times Subtype \rightarrow Action$
$p_subtype_indication_fn : Environment \times Discriminant \times Sid \times Partial_subtype \rightarrow Action$
$type_definition_fn : Environment \times Discriminant \times Tdf \times Subtype \times Environment \rightarrow Action$
$variant_list_fn : Environment \times Discriminant \times Vnt_lst \times Variant \rightarrow Action$
$variant_part_fn : Environment \times Discriminant \times Vrp \times \text{optional}((Id \times Variant)) \rightarrow Action$
$subtype_convert_fn : Environment \times Subtype \times Value \times Value \rightarrow Action$
$view_convert_fn : Environment \times Subtype \times View \times View \rightarrow Action$
$content_fn : LValue \times Value \rightarrow Action$
$assign_fn : Environment \times Subtype \times LValue \times Value \rightarrow Action$
$raw_assign_fn : LValue \times Value \rightarrow Action$
$finalize_fn : Environment \times Subtype \times Value \rightarrow Action$
$value_split_fn : Environment \times Subtype \times Value \rightarrow Action$
$discrete_type_fn : Environment \times Subtype \times Bool \rightarrow Action$
$covers_fn : Value \times Choice \times Bool \rightarrow Action$
$statement_fn : Environment \times Stm \rightarrow Action$
$arb_fn : \text{list}(Action) \rightarrow Action$
$subprogram_body_fn : (Id \xrightarrow{a} View) \times View \rightarrow Action$

The following predicates define a sequence of actions for a variety of different syntactic constructs:

$component_actions(Environment, CompAssoc, \text{list}(Rca), \text{list}((Id \times Value)), \text{list}(Action))$
$expression_list(Environment, \text{list}(Exp), \text{list}(Value), \text{list}(Action))$
$index_actions(Environment, \text{list}(Sid), (Subtype), \text{list}(Action))$
$parameter_action(Environment, \text{list}(Pss), (Id \times Parameter), (Id \times View), Action)$
$parameter_list(Environment, \text{list}(Pss), (Id \xrightarrow{a} Parameter), (Id \xrightarrow{a} View), \text{list}(Action))$

4.4 State

Values of sort *State* describe a current store, together with the current status of program execution. A state may represent the propagation of an exception, exit from a subprogram, or exit from a loop.

4.4.1 Classification of States

States are classified as normal or abnormal. The following judgements define the classification.

$$abnormal_state(exception(I_e, N))$$

$$abnormal_state(exit(I_s, N))$$

$$abnormal_state(proc_return(N))$$

$$abnormal_state(func_return(W, N))$$

$$normal_state(normal(N))$$

4.4.2 Accessing the Store of a State

The following judgements describe the store associated with different states:

$$the_store(exception(I_d, N)) = N$$

$$the_store(exit(I_d, N)) = N$$

$$the_store(proc_return(N)) = N$$

$$the_store(func_return(W, N)) = N$$

$$the_store(normal(N)) = N$$

The following judgements describe the construction of states:

$$make_state(exception(I_d, N), N_1) = exception(I_d, N_1)$$

$$make_state(exit(I_d, N), N_1) = exit(I_d, N_1)$$

$$make_state(proc_return(N), N_1) = proc_return(N_1)$$

$$make_state(func_return(W, N), N_1) = func_return(W, N_1)$$

$$make_state(normal(N), N_1) = normal(N_1)$$

4.4.3 Reading and Writing the Store

$$\frac{\mathit{the_store}(S) = \langle B, ?, ?, ? \rangle}{S^{obj}[I] = B[I]}$$

$$\frac{\mathit{the_store}(S) = \langle ?, B, ?, ? \rangle}{S^{spg}[I] = B[I]}$$

$$\frac{\mathit{the_store}(S) = \langle ?, ?, B, ? \rangle}{S^{typ}[I] = B[I]}$$

$$\frac{\mathit{the_store}(S) = \langle ?, ?, ?, B \rangle}{S^{stp}[I] = B[I]}$$

$$S_1[I \mapsto_1 V] = \mathit{make_state}(S_1, \mathit{the_store}(S_1)[I \mapsto_1 V])$$

$$S_1[I \mapsto_2 V] = \mathit{make_state}(S_1, \mathit{the_store}(S_1)[I \mapsto_2 V])$$

$$S_1[I \mapsto_3 V] = \mathit{make_state}(S_1, \mathit{the_store}(S_1)[I \mapsto_3 V])$$

$$S_1[I \mapsto_4 V] = \mathit{make_state}(S_1, \mathit{the_store}(S_1)[I \mapsto_4 V])$$

$$\langle B_0, B_p, B_t, B_s \rangle [I \mapsto_1 V] = \langle B_0[I \mapsto V], B_p, B_t, B_s \rangle$$

$$\langle B_o, B_0, B_t, B_s \rangle [I \mapsto_2 V] = \langle B_o, B_0[I \mapsto V], B_t, B_s \rangle$$

$$\langle B_o, B_p, B_0, B_s \rangle [I \mapsto_3 V] = \langle B_o, B_p, B_0[I \mapsto V], B_s \rangle$$

$$\langle B_o, B_p, B_t, B_0 \rangle [I \mapsto_4 V] = \langle B_o, B_p, B_t, B_0[I \mapsto V] \rangle$$

4.4.4 The Content of a Location

This function differs from *stored_value* because it works on a state rather than a store and because it allows L-values rather than just locations. In the case of a structured L-value, the appropriate component of a compound object will be returned.

$$S \vdash \mathit{content}(\mathit{location}(I)) \Rightarrow S^{obj}[I]$$

$$\frac{S \vdash \mathit{content}(L_v) \Rightarrow \mathrm{array_val}(R_s, B)}{S \vdash \mathit{content}(\mathrm{array_component}(L_v, V_s)) \Rightarrow B[V_s]}$$

$$\frac{S \vdash \mathit{content}(L_v) \Rightarrow \mathrm{array_val}([\,R_1\,], B)}{S \vdash \mathit{content}(\mathit{array_slice}(L_v, R)) \Rightarrow \mathrm{array_val}([\,R\,], B)}$$

$$\frac{S \vdash \mathit{content}(L_v) \Rightarrow \mathrm{record_val}(T_g, D_v, C_l)}{S \vdash \mathit{content}(\mathit{record_component}(L_v, I_d)) \Rightarrow C_l[I_d]}$$

$$\frac{\begin{array}{c}\mathit{the_store}(S_1) = \langle B, ?, ?, ? \rangle \\ \neg I \in \mathrm{dom}(B)\end{array}}{\mathit{new_object}(S_1, V, I, S_1[I \mapsto_1 V])}$$

$$\frac{\begin{array}{c} the_store(S_1) = \langle ?, B, ?, ? \rangle \\ \neg I \in dom(B) \end{array}}{new_subprogram(S_1, V, I, S_1[I \mapsto_2 V])}$$

$$\frac{\begin{array}{c} the_store(S_0[I \mapsto_3 T_e]) = \langle ?, ?, B, ? \rangle \\ \neg I \in dom(B) \end{array}}{new_type(S_0, T_e, U, S_0[I \mapsto_3 T_e])}$$

$$\frac{\begin{array}{c} the_store(S_0[I \mapsto_4 T_e]) = \langle ?, ?, ?, B \rangle \\ \neg I \in dom(B) \end{array}}{new_subtype(S_0, T_e, U, S_0[I \mapsto_4 T_e])} .$$

4.5 Order of Execution

4.5.1 Sequential Execution

$$S \, [] \, S$$

$$\frac{\begin{array}{c} \mathsf{run}(normal(N), A_0, S_2) \\ S_2 \left[\begin{array}{c} A_1 \\ \ldots \\ A_n \end{array} \right] S_3 \end{array}}{normal(N) \left[\begin{array}{c} A_0 \\ A_1 \\ \ldots \\ A_n \end{array} \right] S_3}$$

$$\frac{abnormal_state(S)}{S \left[\begin{array}{c} A_1 \\ \ldots \\ A_n \end{array} \right] S}$$

4.5.2 Arbitrary Order Execution

$$S \, \{\} \, S$$

$$\frac{abnormal_state(S)}{S \left\{ \begin{array}{c} A_1 \\ \ldots \\ A_n \end{array} \right\} S}$$

$$\frac{\begin{array}{c} pick(A, A_j, A_t) \\ \mathsf{run}(normal(N), A_j, S_1) \\ S_1 \left\{ \begin{array}{c} A_{t1} \\ \ldots \\ A_{tn} \end{array} \right\} S_2 \end{array}}{normal(N) \left\{ \begin{array}{c} A_1 \\ \ldots \\ A_n \end{array} \right\} S_2}$$

$$pick(A \cdot A_s, A, A_s)$$

$$\frac{pick(A_s, A_x, A_t)}{pick(A \cdot A_s, A_x, A \cdot A_t)}$$

4.5.3 Executing Individual Actions

Note that there are predicates that do not involve state. They are included as a matter of convenience. It is possible to include the appropriate terms in a sequential or arbitrary order execution where this makes the definition more readable.

$$\frac{S_1 \begin{bmatrix} A_1 \\ \dots \\ A_n \end{bmatrix} S_2}{\mathsf{run}(S_1, then(A), S_2)}$$

$$\frac{new_object(S_1, V, L, S_2)}{\mathsf{run}(S_1, new_object_fn(V, L), S_2)}$$

$$\frac{S_1, E, V \vdash_{clt} \mathsf{Clt} \Rightarrow B, S_2}{\mathsf{run}(S_1, E, V \vdash_{clt} \mathsf{Clt} \Rightarrow B, S_2)}$$

$$\frac{S_1, E \vdash_{chc} \mathsf{Chc} \Rightarrow C, S_2}{\mathsf{run}(S_1, E \vdash_{chc} \mathsf{Chc} \Rightarrow C, S_2)}$$

$$\frac{S_1, E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Cmp}, A}{\mathsf{run}(S_1, E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Cmp}, S_2)}$$

$$\frac{S_1, E_1 \vdash_{dcl} \mathsf{Dcl} \Rightarrow E_2, S_2}{\mathsf{run}(S_1, E_1 \vdash_{dcl} \mathsf{Dcl} \Rightarrow E_2, S_2)}$$

$$\frac{S_1, E \vdash_{elf} \mathsf{Elf} \Rightarrow R, S_2}{\mathsf{run}(S_1, E \vdash_{elf} \mathsf{Elf} \Rightarrow R, S_2)}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2}{\mathsf{run}(S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2)}$$

$$\frac{S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow D, S_2}{\mathsf{run}(S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow D, S_2)}$$

$$\frac{S_1, E \vdash_{rng} \mathsf{Rng} \Rightarrow R, S_2}{\mathsf{run}(S_1, E \vdash_{rng} \mathsf{Rng} \Rightarrow R, S_2)}$$

$$\frac{S_1, E \vdash_{stm} \mathsf{Stm} \Rightarrow S_2}{\mathsf{run}(S_1, E \vdash_{stm} \mathsf{Stm} \Rightarrow, S_2)}$$

$$\frac{S_1, E_1, D \vdash_{tdf} \mathsf{Tdf} \Rightarrow S_t, E_2, S_2}{\mathsf{run}(S_1, E_1, D \vdash_{tdf} \mathsf{Tdf} \Rightarrow S_t, E_2, S_2)}$$

$$\frac{S_1, E, S_t \vdash subtype_convert(V_1) \Rightarrow V_2, S_2}{\mathsf{run}(S_1, E, S_t \vdash subtype_convert(V_1) \Rightarrow V_2, S_2)}$$

$$\frac{S_1, E, S_t \vdash subtype_convert(W_1) \Rightarrow W_2, S_2}{\mathsf{run}(S_1, E, S_t \vdash subtype_convert(W_1) \Rightarrow W_2, S_2)}$$

$$\frac{S \vdash content(L_v) \Rightarrow V}{\mathsf{run}(S, \vdash content(L_v) \Rightarrow V, S)}$$

$$\frac{S_1 \left\{ \begin{array}{c} A_1 \\ \dots \\ A_n \end{array} \right\} S_2}{\mathsf{run}(S_1, \left\{ \begin{array}{c} A_1 \\ \dots \\ A_n \end{array} \right\}, S_2)}$$

$$\frac{subprogram_body(S_1, A, W, S_2)}{\mathsf{run}(S_1, subprogram_body_fn(A, W), S_2)}$$

4.6 Values

4.6.1 Ranges

$$discrete_range(discrete_rng(I_1, I_2))$$

$$real_range(real_rng(R_1, R_2))$$

$$low_bound(discrete_rng(I, ?)) = discrete_val(I)$$

$$low_bound(real_rng(R, ?)) = real_val(R)$$

$$high_bound(discrete_rng(?, I)) = discrete_val(I)$$

$$high_bound(real_rng(?, R)) = real_val(R)$$

$$make_range(discrete_val(I_1), discrete_val(I_2)) = discrete_rng(I_1, I_2)$$

$$make_range(real_val(R_1), real_val(R_2)) = real_rng(R_1, R_2)$$

$$\frac{R_1 \leq R \quad R \leq R_2}{belongs_to(real_val(R), real_rng(R_1, R_2))}$$

$$\frac{I_1 \leq I \quad I \leq I_2}{belongs_to(discrete_val(I), discrete_rng(I_1, I_2))}$$

$$\frac{I_1 \geq I_3 \quad I_4 \geq I_2}{included_in(discrete_rng(I_1, I_2), discrete_rng(I_3, I_4))}$$

$$\frac{I_1 > I_2}{null_range(discrete_rng(I_1, I_2))}$$

$$\frac{R_1 > R_2}{null_range(real_rng(R_1, R_2))}$$

$$values_in_range(discrete_rng(I, I), set_of([\,discrete_val(I)\,]))$$

$$\frac{I_1 > I_2}{values_in_range(discrete_rng(I_1, I_2), set_of([\,]))}$$

$$\frac{values_in_range(discrete_rng(I_1 + 1, I_2), set_of(V_s))}{values_in_range(discrete_rng(I_1, I_2), set_of(discrete_val(I_1) \cdot V_s))}$$

4.6.2 Index Ranges

$$indices([\,]) = set_of([\,])$$

$$\frac{\begin{array}{c} values_in_range(R, V_r) \\ index_pairing(V_r, indices(R_s), V) \end{array}}{indices(R \cdot R_s) = V}$$

$$index_pairing(set_of([\,]), V_1, set_of([[\,]]))$$

$$\frac{\begin{array}{c} index_pairing(set_of(V_r), V_1, V_x) \\ prefix_set_with_element(V, V_1, V_y) \end{array}}{index_pairing(set_of(V \cdot V_r), V_1, V_x \cup V_y)}$$

$$prefix_set_with_element(V, set_of([\,]), set_of([\,]))$$

$$\frac{prefix_set_with_element(V, set_of(E_r), set_of(E_p))}{prefix_set_with_element(V, set_of(E_1 \cdot E_r), set_of(V \cdot E_1 \cdot E_p))}$$

4.6.3 Predicates of Values

$$discrete_value(discrete_val(X))$$

$$real_value(real_val(X))$$

$$access_value(access_val(X, ?))$$

$$\frac{discrete_value(V)}{scalar_value(V)}$$

$$\frac{real_value(V)}{scalar_value(V)}$$

$$\frac{scalar_value(V)}{elementary_value(V)}$$

$$\frac{access_value(V)}{elementary_value(V)}$$

$$composite_value(record_val(X, Y, Z))$$

$$composite_value(array_val(X, Y))$$

$$\frac{dom(V) = indices(B)}{array_value(array_val(B, V))}$$

4.7 Types

4.7.1 Type Descriptors

$$\frac{S^{typ}[U] = (? \times T_y)}{type_struct(S, subtype(U, ?, ?)) = T_y}$$

4.7.2 Ancestry Relation

$$ancestor(S, U, U)$$

$$\frac{\begin{array}{c} S^{typ}[U] = (some(U_2) \times ?) \\ ancestor(S, U_2, U_1) \end{array}}{ancestor(S, U, U_1)}$$

$$\frac{ancestor(S, U_2, U_1)}{descendant(S, U_1, U_2)}$$

$$\frac{S^{typ}[U] = (none \times ?)}{ultimate_ancestor(S, U, U)}$$

$$\frac{\begin{array}{c} S^{typ}[U] = (some(U_2) \times ?) \\ ultimate_ancestor(S, U_1, U_2) \end{array}}{ultimate_ancestor(S, U, U_2)}$$

4.7.3 Ranges of Scalar Types

$$base_range(S, enum_type(N)) = discrete_rng(0, N)$$

$$base_range(S, modular_type(N)) = discrete_rng(0, N, -, 1)$$

$$base_range(S, signed_integer_type(B_f, B_l, ?, ?)) = discrete_rng(B_f, B_l)$$

$$base_range(S, universal_integer_type(B_f, B_l)) = discrete_rng(B_f, B_l)$$

$$\frac{S_t = subtype(U, no_constraint, A)}{range_of_subtype(S, S_t) = base_range(S, type_struct(S, S_t))}$$

$$range_of_subtype(S, subtype(U, range_constraint(R), A)) = R$$

$$ranges_of_subtypes(S, [\,]) = [\,]$$

$$ranges_of_subtypes(S, S_1 \cdot S_r) = range_of_subtype(S, S_1) \cdot ranges_of_subtypes(S, S_r)$$

4.7.4 Values of a Type

$$\frac{\begin{array}{c} ancestor(S, U, U_p) \\ S^{typ}[U_p] = (? \times T_y) \\ descriptor_value(S, T_y, V) \end{array}}{descriptor_value(S, class_type(U), V)}$$

$$\frac{belongs_to(V, base_range(S, enum_type(N)))}{descriptor_value(S, enum_type(N), V)}$$

$$descriptor_value(S, enum_type(N), invalid_val)$$

$$\frac{belongs_to(V, base_range(S, modular_type(N)))}{descriptor_value(S, modular_type(N), V)}$$

$$descriptor_value(S, modular_type(N), invalid_val)$$

$$descriptor_value(S, signed_integer_type(B_f, B_l, F, L), discrete_val(N))$$

$$descriptor_value(S, signed_integer_type(B_f, B_l, F, L), invalid_val)$$

$$descriptor_value(S, universal_integer_type(B_f, B_l), discrete_val(N))$$

$$descriptor_value(S, universal_integer_type(B_f, B_l), invalid_val)$$

$$\frac{array_value(array_val(ranges_of_subtypes(S, I), B))}{descriptor_value(S, array_type(I, C), array_val(ranges_of_subtypes(S, I), B))}$$

$$\frac{\begin{array}{c} N \in set_of(adom(D)) \\ D[N] = (S_t \times T_h) \\ subtype_value(S, S_t, B[N]) \end{array}}{discriminant_value(S, discr(D), B)}$$

$$\frac{\begin{array}{c} discriminant_value(S, D_a, D_v) \\ cl_value(S, C_l, D_v, C_v) \end{array}}{descriptor_value(S, record_type(T_g, D_a, C_l), record_val(T_g, D_v, C_v))}$$

$$\frac{\begin{array}{c} dom(F) = set_of(adom(A)) \\ A[I_d] = (P_s \times T_h) \\ subtype_value(S, actualize(D_v, P_s), F[I_d]) \\ variant_values(S, V_r, D_v, F_v) \end{array}}{cl_value(S, fields(A, V_r), D_v, F \oplus F_v)}$$

$$variant_values(S, none, D_v, [\,])$$

$$\frac{\begin{array}{c} the_variant(F, D_v[I_d], C_l) \\ cl_value(S, C_l, D_c, V_v) \end{array}}{variant_values(S, some((I_d \times F)), D_v, V_v)}$$

4.7.5 Record Fields

The following definitions are useful to deal with the types of record fields. Binding B represented the values of the discriminants.

$$\frac{D_a[I_d] = (S_t \times T_k)}{component_type(B, record_type(T_g, discr(D_a), C_l), I_d, S_t)}$$

$$\frac{select_component_type(B, C_l, I_d, S_t)}{component_type(B, record_type(T_g, D_a, C_l), I_d, S_t)}$$

$$\frac{C_a[I_d] = (P_s \times T_k)}{select_component_type(B, fields(C_a, ?), I_d, actualize(B, P_s))}$$

Since component names have to be unique, we can quantify over values V in the following rule.

$$\frac{\begin{array}{c} the_variant(V_b, V, C_l) \\ select_component_type(B, C_l, I_d, S_t) \end{array}}{select_component_type(B, fields(C_a, some((I_v \times V_b))), I_d, S_t)}$$

$$\frac{covers(X, C, true)}{the_variant(variant((C \times C_l) \cdot L), X, C_l)}$$

$$\frac{\begin{array}{c} covers(X, C, false) \\ the_variant(variant(L), X, C_l) \end{array}}{the_variant(variant((C \times C_l) \cdot L), X, C_l)}$$

4.7.6 Classification of Types

$$\frac{is_scalar_type(S, T_y)}{is_elementary_type(S, T_y)}$$

$$\frac{is_access_type(S, T_y)}{is_elementary_type(S, T_y)}$$

$$\frac{is_array_type(S, T_y)}{is_composite_type(S, T_y)}$$

$$\frac{is_record_type(S, T_y)}{is_composite_type(S, T_y)}$$

$$\frac{is_tagged_type(S, T_y)}{is_composite_type(S, T_y)}$$

$$\frac{is_protected_type(S, T_y)}{is_composite_type(S, T_y)}$$

$$\frac{is_discrete_type(S, T_y)}{is_scalar_type(S, T_y)}$$

$$\frac{is_real_type(S, T_y)}{is_scalar_type(S, T_y)}$$

$$\frac{is_access_to_object_type(S, T_y)}{is_access_type(S, T_y)}$$

$$\frac{is_access_to_subprogram_type(S, T_y)}{is_access_type(S, T_y)}$$

$$\frac{is_enumeration_type(S, T_y)}{is_discrete_type(S, T_y)}$$

$$\frac{is_integer_type(S, T_y)}{is_discrete_type(S, T_y)}$$

$$\frac{type_struct(S, T_y) = enum_type(N)}{is_enumeration_type(S, T_y)}$$

$$is_boolean_type(S, subtype(boolean_tn, ?, ?))$$

$$is_character_type(S, subtype(character_tn, ?, ?))$$

$$\frac{is_signed_integer_type(S, T_y)}{is_integer_type(S, T_y)}$$

$$\frac{is_modular_integer_type(S, T_y)}{is_integer_type(S, T_y)}$$

$$is_signed_integer_type(S, subtype(root_integer_tn, ?, ?))$$

$$\frac{type_struct(S, T_y) = modular_type(?)}{is_modular_integer_type(S, T_y)}$$

$$\frac{not\ yet\ defined}{is_real_type(S, T_y)}$$

$$\frac{type_struct(S, T_y) = access_type(S_t, A_m)}{is_access_to_object_type(S, T_y)}$$

$$\frac{type_struct(S, T_y) = func_profile(P_a, S_t)}{is_access_to_subprogram_type(S, T_y)}$$

$$\frac{type_struct(S, T_y) = proc_profile(P_a)}{is_access_to_subprogram_type(S, T_y)}$$

$$\frac{type_struct(S, T_y) = array_type(C, B)}{is_array_type(S, T_y)}$$

$$\frac{not\ yet\ defined}{is_string_type(S, T_y)}$$

$$\frac{type_struct(S, T_y) = record_type(T_g, D, C)}{is_record_type(S, T_y)}$$

$$\frac{type_struct(S, T_y) = record_type(some(I_t), D, C)}{is_tagged_type(S, T_y)}$$

$$\frac{not\ yet\ defined}{is_task_type(S, T_y)}$$

$$\frac{not\ yet\ defined}{is_protected_type(S, T_y)}$$

$$\frac{is_elementary_type(S, T_y)}{is_by_copy_type(S, T_y)}$$

$$\frac{is_tagged_type(S, T_y)}{is_by_reference_type(S, T_y)}$$

$$\frac{is_task_type(S, T_y)}{is_by_reference_type(S, T_y)}$$

$$\frac{is_protected_type(S, T_y)}{is_by_reference_type(S, T_y)}$$

4.8 Subtypes

4.8.1 Constraint Satisfaction

$$satisfies(V, no_constraint)$$

$$\frac{belongs_to(V, R)}{satisfies(V, range_constraint(R))}$$

$$satisfies(array_val(B, A), index_constraint(B))$$

$$satisfies(record_val(T_g, D, R), discriminant_constraint(D))$$

4.8.2 Values of a Subtype

$$\frac{S^{typ}[U] = (P \times T_y) \quad descriptor_value(S, T_y, V) \quad satisfies(V, C)}{subtype_value(S, subtype(U, C, A), V)}$$

4.8.3 Actualization

4.8.3.1 Values

$$actualized_value(B, p_value(V)) = V$$

$$actualized_value(B, discriminant_ref(I)) = B[I]$$

4.8.3.2 Ranges

$$\frac{actualized_value(B, L) = discrete_val(L_a) \quad actualized_value(B, H) = discrete_val(H_a)}{actualized_partial_range(B, (L \times H)) = discrete_rng(L_a, H_a)}$$

$$actualized_range_list(B, [\,]) = [\,]$$

$$actualized_range_list(B, R \cdot R_s) = actualized_partial_range(B, R) \cdot actualized_range_list(B, R_s)$$

4.8.3.3 Binding Lists

$$actualized_binding_list(B, [\,]) = [\,]$$

$$actualized_binding_list(B, (I \times V) \cdot R) = (I \times actualized_value(B, V)) \cdot actualized_binding_list(B, R)$$

4.8.3.4 Constraints

$$actualized_constraint(B, p_no_constraint) = no_constraint$$

$$actualized_constraint(B, p_range_constraint(R)) = range_constraint(actualized_partial_range(B, R))$$

$$actualized_constraint(B, p_index_constraint(S)) = index_constraint(actualized_range_list(B, S))$$

$$\frac{C = discriminant_constraint(actualized_binding_list(B, S))}{actualized_constraint(B, p_discriminant_constraint(S)) = C}$$

$$\frac{C = indirect_index_constraint(actualized_range_list(B, S))}{actualized_constraint(B, p_indirect_index_constraint(S)) = C}$$

$$\frac{C = indirect_discriminant_constraint(actualized_binding_list(B, S))}{actualized_constraint(B, p_indirect_discriminant_constraint(S)) = C}$$

4.8.3.5 Subtypes

$$actualize(B, p_subtype(subtype(U, C, A), P_c)) = subtype(U, actualized_constraint(B, P_c), A)$$

4.8.3.6 Components

$$A_1 + A_2 = A_1 + A_2$$

$$actualized_components(B, fields(C_a, some(V_p))) = actualized_complist(B, C_a) + actualized_variants(B, V_p)$$

$$actualized_components(B, fields(C_a, none)) = actualized_complist(B, C_a)$$

4.8.3.7 Component Lists

$$actualized_complist(B, [\,]) = [\,]$$

$$\frac{actualized_complist(B, A) = A_a}{actualized_complist(B, (I \times P_s) \cdot A) = (I \times actualize(B, P_s)) \cdot A_a}$$

$$\frac{the_variant(V_a, B[I], C_l)}{actualized_variants(B, (I \times V_a)) = actualized_components(B, C_l)}$$

4.9 Declarations

4.9.1 Declarations

4.9.2 Types and Subtypes

4.9.2.1 Type Declarations

The semantics of a type definition are determined in the context of a discriminant association. For types without discriminant, this association is empty

$$\frac{\begin{array}{c} S_1, E_1, \langle\rangle \vdash_{tdf} \mathsf{Tdf} \Rightarrow S_t, E_2, S_2 \\ new_subtype(S_2, S_t, L_s, S_3) \end{array}}{S_1, E_1 \vdash_{dcl} \textbf{type } \mathsf{I_d} \textbf{ is } \mathsf{Tdf};\ \Rightarrow E_2[I_d \mapsto subtype_view(L_s)], S_3}$$

$$\frac{\begin{array}{c} S_1 \left[\begin{array}{c} E \vdash_{dsc} \mathsf{Dcp} \Rightarrow D \\ E, D \vdash_{tdf} \mathsf{Tdf} \Rightarrow S_t, E_2 \end{array} \right] S_2 \\ new_subtype(S_2, S_t, L_s, S_3) \end{array}}{S_1, E \vdash_{dcl} \textbf{type } \mathsf{I_d}\ \mathsf{Dcp} \textbf{ is } \mathsf{Tdf};\ \Rightarrow E_2[I_d \mapsto subtype_view(L_s)], S_3}$$

For a given type descriptor, the following rule creates a new unique type name and constructs a *first subtype*.

4.9.2.2 Subtype Declarations

A subtype indication may be a named subtype or a subtype with a constraint. In the former case, evaluation of the subtype indication cannot have side-effects.

$$\frac{S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1}{S_1, E \vdash_{sid} \mathsf{Nam} \Rightarrow S_1^{stp}[L_s], S_1}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1 \\ S_1, E, S_1^{stp}[L_s] \vdash_{cns} \mathsf{Csn} \Rightarrow C, S_2 \\ compatible(E, S_1^{stp}[L_s], C) \\ S_1^{stp}[L_s] = subtype(U, C_1, A) \end{array}}{S_1, E \vdash_{sid} \mathsf{Nam\ Cns} \Rightarrow subtype(U, C, A), S_2}$$

$$\frac{S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1}{S_1, E, D \vdash_{psi} \mathsf{Nam} \Rightarrow p_subtype(S_1^{stp}[L_s], p_no_constraint), S_1}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1 \\ p_constraint(S_1, E, D, S_1^{stp}[L_s], \mathsf{Csn}, P, S_2) \\ compatible(E, S_1^{stp}[L_s], C) \\ S_1^{stp}[L_s] = subtype(U, C_1, A) \end{array}}{S_1, E, D \vdash_{psi} \mathsf{Nam\ Cns} \Rightarrow p_subtype(subtype(U, no_constraint, A), P), S_2}$$

The following syntax represents ranges in a discrete subtype definition. It applies only to constrained array type definitions.

$$\frac{not\ yet\ defined}{S_1, E \vdash_{sid} \mathsf{Rng} \Rightarrow subtype(U, C, A), S_2}$$

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t, S_2 \\ new_subtype(S_2, S_t, L_s, S_3) \end{array}}{S_1, E_1 \vdash_{dcl} \textbf{subtype}\ \mathsf{I_d}\ \textbf{is}\ \mathsf{Sid;} \Rightarrow E_1[I_d \mapsto subtype_view(L_s)], S_3}$$

4.9.3 Objects and Named Numbers

4.9.3.1 Object Declarations

$$\frac{S_1 \left[\begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ new_object_fn(invalid_val, L) \\ default_value_fn(E_1, S_t, V) \\ assign_fn(E_1, S_t, location(L), V) \end{array}\right] S_2}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \mathsf{Sid;} \Rightarrow E_1[I_d \mapsto object_view(location(L), S_t, \mathrm{variable})], S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ new_object_fn(invalid_val, L) \\ default_value_fn(E_1, S_t, V) \\ assign_fn(E_1, S_t, location(L), V) \end{array}\right] S_2}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \textbf{constant}\ \mathsf{Sid;} \Rightarrow E_1[I_d \mapsto object_view(location(L), S_t, \mathrm{constant})], S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ new_object_fn(invalid_val, L) \\ E_1 \vdash_{exp} \mathsf{Exp} \Rightarrow V \\ assign_fn(E_1, S_t, location(L), V) \end{array}\right] S_2}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \mathsf{Sid} := \mathsf{Exp;} \Rightarrow E_1[I_d \mapsto object_view(location(L), S_t, \mathrm{variable})], S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ new_object_fn(invalid_val, L) \\ E_1 \vdash_{exp} \mathsf{Exp} \Rightarrow V \\ assign_fn(E_1, S_t, location(L), V) \end{array}\right] S_2}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \textbf{constant}\ \mathsf{Sid} := \mathsf{Exp};\ \Rightarrow E_1[I_d \mapsto object_view(location(L), S_t, constant)], S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ new_object_fn(invalid_val, L) \\ default_value_fn(E_1, S_t, V) \\ assign_fn(E_1, S_t, location(L), V) \end{array}\right] S_2}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \textbf{aliased}\ \mathsf{Sid};\ \Rightarrow E_1[I_d \mapsto object_view(location(L), S_t, aliased)], S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ new_object_fn(invalid_val, L) \\ default_value_fn(E_1, S_t, V) \\ assign_fn(E_1, S_t, location(L), V) \end{array}\right] S_2}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \textbf{aliased constant}\ \mathsf{Sid};\ \Rightarrow E_1[I_d \mapsto object_view(location(L), S_t, constant)], S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ new_object_fn(invalid_val, L) \\ E_1 \vdash_{exp} \mathsf{Exp} \Rightarrow V \\ assign_fn(E_1, S_t, location(L), V) \end{array}\right] S_2}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \textbf{aliased}\ \mathsf{Sid} := \mathsf{Exp};\ \Rightarrow E_1[I_d \mapsto object_view(location(L), S_t, aliased)], S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ new_object_fn(invalid_val, L) \\ E_1 \vdash_{exp} \mathsf{Exp} \Rightarrow V \\ assign_fn(E_1, S_t, location(L), V) \end{array}\right] S_2}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \textbf{aliased constant}\ \mathsf{Sid} := \mathsf{Exp};\ \Rightarrow E_1[I_d \mapsto object_view(location(L), S_t, constant)], S_2}$$

4.9.3.2 Number Declarations

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2 \\ new_object(S_1, V, L, S_2) \\ E_3 = E_1[I_d \mapsto object_view(location(L), subtype(universal_real_tn, no_constraint, not_used), constant)] \end{array}}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \textbf{constant} := \mathsf{Exp};\ \Rightarrow E_3, S_2}$$

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2 \\ new_object(S_1, V, L, S_2) \\ E_3 = E_1[I_d \mapsto object_view(location(L), subtype(universal_integer_tn, no_constraint, not_used), constant)] \end{array}}{S_1, E_1 \vdash_{dcl} \mathsf{I_d} : \textbf{constant} := \mathsf{Exp};\ \Rightarrow E_3, S_2}$$

4.9.4 Derived Types and Classes

$$\frac{\begin{array}{c} S_0, E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow subtype(P, C, A), S_1 \\ S_1^{typ}[P] = (P_p \times T_y) \\ new_type(S_1, (some(P) \times T_y), U, S_2) \end{array}}{S_0, E_1, D_0 \vdash_{tdf} \textbf{new}\ \mathsf{Sid} \Rightarrow subtype(U, C, A), E_2, S_2}$$

4.9.5 Scalar Types

$$\frac{S_1 \left\{ \begin{array}{c} E \vdash_{exp} \mathsf{Exp}_1 \Rightarrow V_1 \\ E \vdash_{exp} \mathsf{Exp}_2 \Rightarrow V_2 \end{array} \right\} S_2}{S_1, E \vdash_{rng} \mathsf{Exp}_1 .. \mathsf{Exp}_2 \Rightarrow \mathit{make_range}(V_1, V_2), S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow W, S_2 \\ \mathit{abnormal_state}(S_2) \end{array}}{S_1, E \vdash_{rng} \mathsf{Nam}'\mathbf{range} \Rightarrow R, S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow W, S_2 \\ \mathit{normal_state}(S_2) \\ S_2, E, W \vdash_{attr} \mathsf{range} \Rightarrow R, S_3 \end{array}}{S_1, E \vdash_{rng} \mathsf{Nam}'\mathbf{range} \Rightarrow R, S_3}$$

$$\frac{\textit{not yet defined}}{S_1, E \vdash_{rng} \mathsf{Nam}'\mathbf{range}(\mathsf{Exp}) \Rightarrow R, S_2}$$

4.9.5.1 Enumeration Types

4.9.5.2 Character Types

4.9.5.3 Boolean Types

4.9.5.4 Integer Types

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{exp} \mathsf{Exp}_1 \Rightarrow \mathit{discrete_val}(V_1), S_1 \\ S_1, E_1 \vdash_{exp} \mathsf{Exp}_2 \Rightarrow \mathit{discrete_val}(V_2), S_1 \\ \mathit{new_type}(S_1, (\mathit{some}(\mathit{root_integer_tn}) \times \mathit{signed_integer_type}(V_1, V_2, V_1, V_2)), U, S_2) \end{array}}{S_1, E_1, \langle\rangle \vdash_{tdf} \mathbf{range}\ \mathsf{Exp}_1 .. \mathsf{Exp}_2 \Rightarrow \mathit{subtype}(U, \mathit{no_constraint}, \mathit{not_used}), E_1, S_2}$$

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(V), S_1 \\ \mathit{new_type}(S_1, (\mathit{some}(\mathit{root_integer_tn}) \times \mathit{modular_type}(V)), U, S_2) \end{array}}{S_1, E_1, \langle\rangle \vdash_{tdf} \mathbf{mod}\ \mathsf{Exp} \Rightarrow \mathit{subtype}(U, \mathit{no_constraint}, \mathit{not_used}), E_1, S_2}$$

4.9.5.5 Operations of Discrete Types

4.9.5.6 Real Types

4.9.5.7 Floating Point Types

$$\frac{\textit{not yet defined}}{S_1, E_1, D \vdash_{tdf} \mathbf{digits}\ \mathsf{Exp}\ \mathsf{Cns} \Rightarrow S_t, E_2, S_2}$$

$$\frac{\textit{not yet defined}}{S_1, E_1, D \vdash_{tdf} \mathbf{digits}\ \mathsf{Exp} \Rightarrow S_t, E_2, S_2}$$

4.9.5.8 Operations of Floating Point Types

4.9.5.9 Fixed Point Types

$$\frac{\textit{not yet defined}}{S_1, E_1, D \vdash_{tdf} \mathbf{delta}\ \mathsf{Exp}\ \mathbf{digits}\ \mathsf{Exp}\ \mathsf{Cns} \Rightarrow S_t, E_2, S_2}$$

$$\frac{\textit{not yet defined}}{S_1, E_1, D \vdash_{tdf} \mathbf{delta}\ \mathsf{Exp}\ \mathbf{digits}\ \mathsf{Exp} \Rightarrow S_t, E_2, S_2}$$

$$\frac{\textit{not yet defined}}{S_1, E_1, D \vdash_{tdf} \textbf{delta}\ \mathsf{Exp}\ \mathsf{Rng} \Rightarrow S_t, E_2, S_2}$$

4.9.5.10 Operations of Fixed Point Types

4.9.6 Array Types

The evaluation of a subtype mark has no side-effect. Therefore, the evaluation of lists of subtype marks is defined sequentially.

$$S, E \vdash_{Stl} () \Rightarrow [\,]$$

$$\frac{\begin{array}{c} S, E \vdash_{nam} \mathsf{Nam_0} \Rightarrow \textit{subtype_view}(L_s), S \\ S, E \vdash_{Stl} (\mathsf{Nam}, \ldots) \Rightarrow S_t \end{array}}{S, E \vdash_{Stl} (\mathsf{Nam_0}, \mathsf{Nam}, \ldots) \Rightarrow S^{stp}[L_s] \cdot S_t}$$

$$\textit{index_actions}(E, (\,), [\,], [\,])$$

$$\frac{\textit{index_actions}(E, (\mathsf{Sid}, \ldots), S_t, A_c)}{\textit{index_actions}(E, (\mathsf{Sid_0}, \mathsf{Sid}, \ldots), St_0 \cdot S_t, E \vdash_{sid} \mathsf{Sid_0} \Rightarrow St_0 \cdot A_c)}$$

$$\textit{range_constraints}(S, [\,], [\,])$$

$$\frac{\textit{type_constraints}(S, S_t, R)}{\textit{range_constraints}(S, \textit{subtype}(?, \textit{range_constraint}(R_0), ?) \cdot S_t, R_0 \cdot R)}$$

$$\frac{\textit{not yet defined}}{S_1, E_1, D \vdash_{tdf} \textbf{array}(\mathsf{Rng_lst})\textbf{of aliased}\ \mathsf{Sid} \Rightarrow S_t, E_2, S_2}$$

$$\frac{\textit{not yet defined}}{S_1, E_1, D \vdash_{tdf} \textbf{array}(\mathsf{Nam_0}, \mathsf{Nam}, \ldots)\textbf{of aliased}\ \mathsf{Sid} \Rightarrow S_t, E_2, S_2}$$

$$\frac{\begin{array}{c} \textit{index_actions}(E, \mathsf{Idx}, S_i, A_c) \\ S_1 \left\{ \begin{array}{c} E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t \\ A_{c1} \\ \ldots \\ A_{cn} \end{array} \right\} S_2 \\ \textit{range_constraints}(S_2, S_i, R) \\ \textit{new_type}(S_2, (\textit{none} \times \textit{array_type}(S_i, S_t)), U, S_3) \end{array}}{S_1, E_1, \langle\rangle \vdash_{tdf} \textbf{array}(\mathsf{Idx})\textbf{of}\ \mathsf{Sid} \Rightarrow \textit{subtype}(U, \textit{index_constraint}(R), \textit{not_used}), E_2, S_3}$$

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{Stl} \mathsf{Idx} \Rightarrow S_i \\ S_1, E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow S_t, S_2 \\ \textit{new_type}(S_2, (\textit{none} \times \textit{array_type}(S_i, S_t)), U, S_3) \end{array}}{S_1, E_1, \langle\rangle \vdash_{tdf} \textbf{array}(\mathsf{Idx})\textbf{of}\ \mathsf{Sid} \Rightarrow \textit{subtype}(U, \textit{no_constraint}, \textit{not_used}), E_1, S_3}$$

4.9.7 Discriminants

4.9.8 Record Types

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Cmp} \\ E, D \vdash_{vrn} \mathsf{Vrp} \Rightarrow V \\ \mathit{new_type_fn}((\mathit{none} \times \mathit{record_type}(\mathit{none}, D, \mathit{fields}(C, V))), U) \end{array} \right] S_2}{S_1, E, D \vdash_{tdf} \mathsf{Cmp} \Rightarrow \mathit{subtype}(U, \mathit{no_constraint}, \mathit{not_used}), E, S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Cmp} \\ E, D \vdash_{vrn} \mathsf{Vrp} \Rightarrow V \\ \mathit{new_type_fn}((\mathit{none} \times \mathit{record_type}(\mathit{some}(U), D, \mathit{fields}(C, V))), U) \end{array} \right] S_2}{S_1, E, D \vdash_{tdf} \textbf{tagged } \mathsf{Cmp} \Rightarrow \mathit{subtype}(U, \mathit{no_constraint}, \mathit{not_used}), E, S_2}$$

$$S, E \vdash_{cmp} \mathsf{D} \Rightarrow \langle\rangle, [\,]$$

$$\frac{\textit{not yet defined}}{S_1, E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Idn} : \textbf{aliased } \mathsf{Sid};\ \mathsf{Cmp}\ \ldots, A}$$

$$\frac{S_1 \left[\begin{array}{c} E, D \vdash_{psi} \mathsf{Sid} \Rightarrow P_s \\ E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Cmp}\ \ldots \end{array} \right] S_2}{S_1, E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Idn} : \mathsf{Sid};\ \mathsf{Cmp}\ \ldots, A_1[\mathsf{Idn} \mapsto (P_s \times \mathit{none})]}$$

$$\frac{\textit{not yet defined}}{S_1, E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Idn} : \textbf{aliased } \mathsf{Sid} := \mathsf{Exp};\ \mathsf{Cmp}\ \ldots, A}$$

$$\frac{S_1 \left[\begin{array}{c} E, D \vdash_{psi} \mathsf{Sid} \Rightarrow P_s \\ E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Cmp}\ \ldots \end{array} \right] S_2}{S_1, E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Idn} : \mathsf{Sid} := \mathsf{Exp};\ \mathsf{Cmp}\ \ldots, A}$$

4.9.8.1 Variant Parts and Discrete Choices

$$S, E, D \vdash_{vrn}\ \Rightarrow \mathit{none}, S$$

$$\frac{S_1, E, D \vdash_{vrn} \mathsf{Vnt} \Rightarrow B, S_2}{S_1, E, D \vdash_{vrn} \textbf{case } \mathsf{Idn} \textbf{ is } \mathsf{Vnt} \textbf{ end case};\ \Rightarrow \mathit{some}((\mathsf{Idn} \times B)), S_2}$$

$$\langle\rangle = \mathit{variant}([\,])$$

$$S, E, D \vdash_{vrn} \langle\rangle \Rightarrow \langle\rangle, S$$

$$\frac{\begin{array}{c} S_1, E \vdash_{chc} \mathsf{Dch} \Rightarrow C_v, S_1 \\ S_1 \left[\begin{array}{c} E \vdash_{cmp} \mathsf{D} \Rightarrow \mathsf{Cmp} \\ E, D \vdash_{vrn} \mathsf{Vrp} \Rightarrow V \\ E, D \vdash_{vrn} \mathsf{Vnt};\ \ldots \Rightarrow \mathit{variant}(B_1) \end{array} \right] S_2 \end{array}}{S_1, E, D \vdash_{vrn} \textbf{when } \mathsf{Dch} \Rightarrow \mathsf{CmpVrp};\ \mathsf{Vnt};\ \ldots \Rightarrow \mathit{variant}((C_v \times \mathit{fields}(C, V)) \cdot B_2), S_2}$$

4.9.9 Tagged Types and Type Extensions

4.9.9.1 Type Extensions

$$\frac{S_1 \left[\begin{array}{c} S_0, E_1 \vdash_{sid} \mathsf{Sid} \Rightarrow subtype(P, C, A), S_1 \\ S_1^{typ}[P] = (P_p \times record_type(some(U_p), D_1, fields(C_1, V_1))) \\ E_1 \vdash_{cmp} \mathsf{D_0} \Rightarrow \mathsf{Cmp} \\ E_1, D_0 \vdash_{vrn} \mathsf{Vrp} \Rightarrow V_2 \\ new_type_fn((some(U_p) \times record_type(some(U), discriminant_union(D_0, D_1), fields(C_1 \oplus C_2, V_1 \oplus V_2))), U) \end{array}\right]}{S_0, E_1, D_0 \vdash_{tdf} \textbf{new } \mathsf{Sid} \textbf{ with } \mathsf{Cmp} \Rightarrow subtype(U, no_constraint, not_used), E_2, S_2}$$

$$none \oplus V = V$$

$$V \oplus none = V$$

$$discriminant_union(discr(A_1), discr(A_2)) = discr(A_1 \oplus A_2)$$

4.9.10 Access Types

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{Sid_i} \mathsf{Sid} \Rightarrow L_s, S_2 \\ new_type(S_2, (none \times access_type(L_s, pool_access)), U, S_3) \end{array}}{S_1, E_1, D \vdash_{tdf} \textbf{access } \mathsf{Sid} \Rightarrow subtype(U, no_constraint, not_used), E_1, S_3}$$

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{Sid_i} \mathsf{Sid} \Rightarrow L_s, S_2 \\ new_type(S_2, (none \times access_type(L_s, all_access)), U, S_3) \end{array}}{S_1, E_1, D \vdash_{tdf} \textbf{access all } \mathsf{Sid} \Rightarrow subtype(U, no_constraint, not_used), E_1, S_3}$$

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{Sid_i} \mathsf{Sid} \Rightarrow L_s, S_2 \\ new_type(S_2, (none \times access_type(L_s, constant_access)), U, S_3) \end{array}}{S_1, E_1, D \vdash_{tdf} \textbf{access constant } \mathsf{Sid} \Rightarrow subtype(U, no_constraint, not_used), E_1, S_3}$$

$$\frac{S_1, E \vdash_{pas} \mathsf{Pms} \Rightarrow A, S_2}{S_1, E, \langle\rangle \vdash_{tdf} \textbf{access procedure } \mathsf{Pms} \Rightarrow subtype(U, no_constraint, not_used), E, S_2[proc_profile(A) \mapsto_3 U]}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{pas} \mathsf{Pms} \Rightarrow A, S_2 \\ S_2, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(S_l), S_2 \end{array}}{S_1, E_1, \langle\rangle \vdash_{tdf} \textbf{access function } \mathsf{Pms} \Rightarrow subtype(U, no_constraint, not_used), E_2, S_2[func_profile(A, S_2^{stp}[S_l]) \mapsto_3 U]}$$

4.9.10.1 Incomplete Type Declarations

In the case of access type definitions, a subtype indication may denote an incomplete type.

$$\frac{S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1}{S_1, E \vdash_{Sid_i} \mathsf{Nam} \Rightarrow L_s, S_1}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1 \\ S_1, E, S_1^{stp}[L_s] \vdash_{cns} \mathsf{Csn} \Rightarrow C, S_2 \\ compatible(E, S_1^{stp}[L_s], C) \\ S_1^{stp}[L_s] = subtype(U, C_1, A) \\ new_subtype(S_2, subtype(U, C, A), L_t, S_3) \end{array}}{S_1, E \vdash_{Sid_i} \mathsf{Nam\ Cns} \Rightarrow L_t, S_3}$$

$$() = discr([\,])$$

For incomplete type declarations, a new incomplete type descriptor carries the discriminant information.

$$\frac{\begin{array}{c} new_type(S_0, (none \times incomplete_type(())), U, S_1) \\ new_subtype(S_1, subtype(U, no_constraint, not_used), L_s, S_2) \end{array}}{S_0, E_1 \vdash_{dcl} \textbf{type } \mathsf{I_d}; \Rightarrow E_1[I_d \mapsto subtype_view(L_s)], S_2}$$

$$\frac{\begin{array}{c} S_1, E_1 \vdash_{dsc} \mathsf{Dcp} \Rightarrow D, S_2 \\ new_type(S_2, (none \times incomplete_type(D)), U, S_3) \\ new_subtype(S_3, subtype(U, no_constraint, not_used), L_s, S_4) \end{array}}{S_1, E_1 \vdash_{dcl} \textbf{type } \mathsf{I_d}\ \mathsf{Dcp}; \Rightarrow E_1[I_d \mapsto subtype_view(L_s)], S_3}$$

Completing type declarations have their own abstract syntax.

$$\frac{\begin{array}{c} E_1 \vdash_{lookup} \mathsf{I_d} \Rightarrow subtype_view(L_s) \\ S_1, E_1, () \vdash_{tdf} \mathsf{Tdf} \Rightarrow S_t, E_2, S_2 \end{array}}{S_1, E_1 \vdash_{dcl} \textbf{completetype } \mathsf{I_d} \textbf{ is } \mathsf{Tdf}; \Rightarrow E_2, S_2[L_s \mapsto_4 S_t]}$$

$$\frac{\begin{array}{c} E_1 \vdash_{lookup} \mathsf{I_d} \Rightarrow subtype_view(L_s) \\ S_1 \left[\begin{array}{c} E \vdash_{dsc} \mathsf{Dcp} \Rightarrow D \\ E, D \vdash_{tdf} \mathsf{Tdf} \Rightarrow S_t, E_2 \end{array} \right] S_2 \end{array}}{S_1, E_1 \vdash_{dcl} \textbf{completetype } \mathsf{I_d}\ \mathsf{Dcp} \textbf{ is } \mathsf{Tdf}; \Rightarrow E_2, S_2[L_s \mapsto_4 S_t]}$$

4.10 Expressions

4.10.1 Names

$$\frac{E \vdash_{lookup} \mathsf{Idn} \Rightarrow W}{S, E \vdash_{nam} \mathsf{Idn} \Rightarrow W, S}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow object_view(L_1, S_t, ?) \\ \vdash content(L_1) \Rightarrow access_val(W, ?) \end{array} \right] S_2}{S_1, E \vdash_{nam} \mathsf{Nam}.\textbf{all} \Rightarrow W, S_2}$$

4.10.1.1 Indexed Components

The function *expression_list* computes a sequence of actions that corresponds to the evaluation of a sequence of expressions.

$$expression_list(E, [\,], [\,], [\,])$$

$$\frac{\begin{array}{c} expression_list(E, \mathsf{Exs}, V_s, A_s) \\ A = E \vdash_{exp} \mathsf{Exp} \Rightarrow V \end{array}}{expression_list(E, \mathsf{Exp} \cdot \mathsf{Exs}, V \cdot V_s, A \cdot A_s)}$$

$$index_list(S, [\,], [\,], [\,], S)$$

$$\frac{\begin{array}{c} belongs_to(V, R) \\ index_list(S_1, R_s, V_s, I_s, S_2) \end{array}}{index_list(S_1, R \cdot R_s, V \cdot V_s, V \cdot I_s, S_2)}$$

$$\frac{\neg belongs_to(V, R)}{index_list(normal(N), R \cdot R_s, V \cdot V_s, I \cdot I_s, exception(constraint_error, N))}$$

$$\frac{\begin{array}{c} expression_list(E, \mathsf{Exs}, V_i, A_s) \\ S_1 \left\{ \begin{array}{c} E \vdash_{nam} \mathsf{Prefix} \Rightarrow object_view(L_1, S_t, ?) \\ A_{s1} \\ \ldots \\ A_{sn} \end{array} \right\} S_2 \\ abnormal_state(S_2) \end{array}}{S_1, E \vdash_{nam} \mathsf{Prefix(Exs, \ldots)} \Rightarrow undefined_view, S_2}$$

$$\frac{\begin{array}{c} expression_list(E, \mathsf{Exs}, V_i, A_s) \\ S_1 \left\{ \begin{array}{c} E \vdash_{nam} \mathsf{Prefix} \Rightarrow object_view(L_1, S_t, C) \\ A_{s1} \\ \ldots \\ A_{sn} \end{array} \right\} S_2 \\ normal_state(S_2) \\ type_struct(S_2, S_t) = array_type(?, B) \\ S_2 \vdash content(L_1) \Rightarrow array_val(R, A_v) \\ index_list(S_2, R, V_i, I_x, S_3) \end{array}}{S_1, E \vdash_{nam} \mathsf{Prefix(Exs, \ldots)} \Rightarrow object_view(array_component(L_1, I_x), B, C), S_3}$$

4.10.1.2 Slices

$$\frac{null_range(R_s)}{slice_check(S, R_s, R_a, S)}$$

$$\frac{included_in(R_s, R_a)}{slice_check(S, R_s, R_a, S)}$$

$$\frac{\begin{array}{c} \neg null_range(R_s) \\ \neg included_in(R_s, R_a) \end{array}}{slice_check(normal(N), R_s, R_a, exception(constraint_error, N))}$$

$$\frac{\begin{array}{c} S_1 \left\{ \begin{array}{c} E \vdash_{nam} \mathsf{Prefix} \Rightarrow W \\ E \vdash_{rng} \mathsf{Rng} \Rightarrow R_s \end{array} \right\} S_2 \\ abnormal_state(S_2) \end{array}}{S_1, E \vdash_{nam} \mathsf{Prefix(Rng)} \Rightarrow undefined_view, S_2}$$

$$\frac{\begin{array}{c} S_1 \left\{ \begin{array}{c} E \vdash_{nam} \mathsf{Prefix} \Rightarrow object_view(L_v, S_t, C) \\ E \vdash_{rng} \mathsf{Rng} \Rightarrow R_s \end{array} \right\} S_2 \\ normal_state(S_2) \\ S_2 \vdash content(L_v) \Rightarrow array_val([R_a], A_v) \\ constrain(S_t, range_constraint(R_s), S_q) \\ slice_check(S_2, R_s, R_a, S_3) \end{array}}{S_1, E \vdash_{nam} \mathsf{Prefix(Rng)} \Rightarrow object_view(array_slice(L_v, R_s), S_q, C), S_3}$$

4.10.1.3 Selected Components

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow W, S_2 \\ abnormal_state(S_2) \end{array}}{S_1, E \vdash_{nam} \mathsf{Nam.Idn} \Rightarrow undefined_view, S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow object_view(L_v, S_t, ?), S_2 \\ component_type(D_v, type_struct(S_2, S_t), \mathsf{Idn}, S_q) \\ S_2 \vdash content(L_v) \Rightarrow record_val(T_g, D_v, C_v) \\ new_object(S_2, D_v[\mathsf{Idn}], L, S_3) \end{array}}{S_1, E \vdash_{nam} \mathsf{Nam.Idn} \Rightarrow object_view(location(L), S_q, constant), S_3}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow object_view(L_v, S_t, C), S_2 \\ S_2 \vdash content(L_v) \Rightarrow record_val(T_g, D_v, C_v) \\ component_type(D_v, type_struct(S_2, S_t), \mathsf{Idn}, S_q) \end{array}}{S_1, E \vdash_{nam} \mathsf{Nam.Idn} \Rightarrow object_view(record_component(L_v, \mathsf{Idn}), S_q, C), S_2}$$

$$\frac{\begin{array}{c} S, E \vdash_{nam} \mathsf{Nam} \Rightarrow object_view(L_v, S_t, ?), normal(N) \\ \neg component_type(B, type_struct(normal(N), S_t), \mathsf{Idn}, S_q) \end{array}}{S, E \vdash_{nam} \mathsf{Nam.Idn} \Rightarrow undefined_view, exception(constraint_error, N)}$$

4.10.1.4 Expanded Names

4.10.1.5 Attributes

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow W_1 \\ E, W_1 \vdash_{att} \mathsf{Idn} \Rightarrow W \end{array} \right] S_2}{S_1, E \vdash_{nam} \mathsf{Nam'Idn} \Rightarrow W, S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_1 \\ S_1 \left[\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow W_1 \\ E, W_1, V \vdash_{att_p} \mathsf{Idn} \Rightarrow W \end{array} \right] S_2 \end{array}}{S, E \vdash_{nam} \mathsf{Nam'Idn(Exp)} \Rightarrow W, S}$$

Note that the abstract syntax distinguishes N'I(E) for static expression E from N'I(E) where N'I is a function-valued attribute.

4.10.2 Literals

$$S, E \vdash_{exp} \mathbf{null} \Rightarrow null, S$$

In the abstract syntax, the representation of character literals is given by the numeric value of the position of the character.

$$S, E \vdash_{exp} \mathsf{'C'} \Rightarrow discrete_val(C), S$$

$$S, E \vdash_{exp} \mathsf{R} \Rightarrow real_val(R), S$$

$$S, E \vdash_{exp} \mathsf{N} \Rightarrow discrete_val(N), S$$

4.10.3 Aggregates

Only explicitly qualified aggregates are defined. Static semantics adds qualification where needed.

We assume a normalized representation using named associations. This is possible because of [4.3.1(14)].

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1 \\ type_struct(S_1, S_1^{stp}[L_s]) = record_type(T_g, discr(L), C_l) \\ S_1^{stp}[L_s] = subtype(?, no_constraint, ?) \\ S_1, E, L \vdash_{agg} \mathsf{Agg} \Rightarrow B_d, S_2 \\ S_2, E, actualized_components(B_d, C_l) \vdash_{agg} \mathsf{Agg} \Rightarrow B_c, S_3 \end{array}}{S_1, E \vdash_{exp} \mathsf{Nam'Agg} \Rightarrow record_val(T_g, B_d, B_c), S_3}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1 \\ type_struct(S_1, S_1^{stp}[L_s]) = record_type(T_g, D_a, C_l) \\ S_1^{stp}[L_s] = subtype(?, discriminant_constraint(B_d), ?) \\ S_1, E, actualized_components(B_d, C_l) \vdash_{agg} \mathsf{Agg} \Rightarrow B_c, S_2 \end{array}}{S_1, E \vdash_{exp} \mathsf{Nam'Agg} \Rightarrow record_val(T_g, B_d, B_c), S_2}$$

$$find_component(I_d, \mathbf{others} \Rightarrow \mathsf{Exp} \cdot \mathsf{Rca}, \mathsf{Exp})$$

$$\frac{member(I_d, \mathsf{Lst})}{find_component(I_d, \mathsf{Lst}, \ldots \Rightarrow \mathsf{Exp} \cdot \mathsf{Rca}, \mathsf{Exp})}$$

$$component_actions(E, [\,], \mathsf{Rca}, [\,], [\,])$$

$$\frac{\begin{array}{c} find_component(I_d, \mathsf{Rca}, \mathsf{Exp}) \\ component_actions(E, B, \mathsf{Rca}, B_c, A_s) \end{array}}{component_actions(E, (I_d \times ?) \cdot B, \mathsf{Rca}, (I_d \times V) \cdot B_c, E \vdash_{exp} \mathsf{Exp} \Rightarrow V \cdot A_s)}$$

$$\frac{\begin{array}{c} \neg find_component(I_d, \mathsf{Rca}, ?) \\ component_actions(E, B, \mathsf{Rca}, B_c, A_s) \end{array}}{component_actions(E, (I_d \times ?) \cdot B, \mathsf{Rca}, (I_d \times V) \cdot B_c, E_1 \vdash_{exp} \mathsf{Exp} \Rightarrow V \cdot A_s)}$$

$$\frac{\begin{array}{c} component_actions(E, A, \mathsf{Rca}, B, A_s) \\ S_1 \left\{ \begin{array}{c} A_{s1} \\ \ldots \\ A_{sn} \end{array} \right\} S_2 \end{array}}{S_1, E, A \vdash_{agg} (\mathsf{Rca}, \ldots) \Rightarrow B, S_2}$$

$$S, E, [\,] \vdash_{agg} (\mathbf{null\ record}) \Rightarrow [\,], S$$

$$\frac{not\ yet\ defined}{S_1, E, R_s, C_s \vdash_{agg} (\mathsf{Exp}_1, \mathsf{Exp}_2, \ldots) \Rightarrow V, S_2}$$

$$\frac{not\ yet\ defined}{S_1, E, R_s, C_s \vdash_{agg} (\mathsf{Exp}_1, \mathsf{Exp}_2, \ldots, \mathbf{others} \Rightarrow \mathsf{Exp}) \Rightarrow V, S_2}$$

$$\frac{not\ yet\ defined}{S_1, E, R_s, C_s \vdash_{agg} (\mathsf{Aca}_1, \mathsf{Aca}_2, \ldots) \Rightarrow V, S_2}$$

$$\frac{not\ yet\ defined}{S_1, E, R_s, C_s \vdash_{agg} (\mathsf{Exp}_1, \mathsf{Exp}_2, \ldots, \mathbf{others} \Rightarrow \mathsf{Exp}) \Rightarrow V, S_2}$$

4.10.4 Operators and Expression Evaluation

It is assumed that static semantics has resolved all operators into function calls as defined in [4.5] with the following exceptions:

- short-circuit operators
- **in** and **not in** operators

4.10.4.1 Logical Operators and Short-circuit Control Forms

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp_1} \Rightarrow \mathit{discrete_val}(1) \\ E \vdash_{exp} \mathsf{Exp_2} \Rightarrow V \end{array} \right] S_2}{S_1, E \vdash_{exp} \mathsf{Exp_1}\ \textbf{and then}\ \mathsf{Exp_2} \Rightarrow V, S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp_1} \Rightarrow \mathit{discrete_val}(0), S_2}{S_1, E \vdash_{exp} \mathsf{Exp_1}\ \textbf{and then}\ \mathsf{Exp_2} \Rightarrow \mathit{discrete_val}(0), S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp_1} \Rightarrow \mathit{discrete_val}(0) \\ E \vdash_{exp} \mathsf{Exp_2} \Rightarrow V \end{array} \right] S_2}{S_1, E \vdash_{exp} \mathsf{Exp_1}\ \textbf{or else}\ \mathsf{Exp_2} \Rightarrow V, S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp_1} \Rightarrow \mathit{discrete_val}(1), S_2}{S_1, E \vdash_{exp} \mathsf{Exp_1}\ \textbf{or else}\ \mathsf{Exp_2} \Rightarrow \mathit{discrete_val}(1), S_2}$$

4.10.4.2 Relational Operators and Membership Tests

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow \mathit{subtype_view}(L_s), S_1 \\ S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2 \\ \mathit{belongs_to}(V, \mathit{range_of_subtype}(S_1, S_1^{stp}[L_s])) \end{array}}{S_1, E \vdash_{exp} \mathsf{Exp}\ \textbf{in}\ \mathsf{Nam} \Rightarrow \mathit{discrete_val}(1), S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow \mathit{subtype_view}(L_s), S_1 \\ S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2 \\ \neg \mathit{belongs_to}(V, \mathit{range_of_subtype}(S_2, S_1^{stp}[L_s])) \end{array}}{S_1, E \vdash_{exp} \mathsf{Exp}\ \textbf{in}\ \mathsf{Nam} \Rightarrow \mathit{discrete_val}(0), S_2}$$

$$\frac{\begin{array}{c} S_1 \left\{ \begin{array}{c} E \vdash_{rng} \mathsf{Rng} \Rightarrow R \\ E \vdash_{exp} \mathsf{Exp} \Rightarrow V \end{array} \right\} S_2 \\ \mathit{belongs_to}(V, R) \end{array}}{S_1, E \vdash_{exp} \mathsf{Exp}\ \textbf{in}\ \mathsf{Rng} \Rightarrow \mathit{discrete_val}(1), S_2}$$

$$\frac{\begin{array}{c} S_1 \left\{ \begin{array}{c} E \vdash_{rng} \mathsf{Rng} \Rightarrow R \\ E \vdash_{exp} \mathsf{Exp} \Rightarrow V \end{array} \right\} S_2 \\ \neg \mathit{belongs_to}(V, R) \end{array}}{S_1, E \vdash_{exp} \mathsf{Exp}\ \textbf{in}\ \mathsf{Rng} \Rightarrow \mathit{discrete_val}(0), S_2}$$

$$\frac{\begin{array}{c} \neg \mathit{is_tagged_type}(S, S_t) \\ S_t = \mathit{subtype}(U, C, ?) \\ \mathit{satisfies}(V, C) \end{array}}{\mathit{test_in}(S, V, S_1)}$$

$$\frac{\begin{array}{c} S_t = subtype(U, C, ?) \\ satisfies(V, C) \\ V = record_val(some(T_g), ?, ?) \\ ancestor(S, U, T_g) \end{array}}{test_in(S, V, S_t)}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1 \\ S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2 \\ test_in(S_2, V, S_1^{stp}[L_s]) \end{array}}{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{in}\ \mathsf{Nam} \Rightarrow discrete_val(1), S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1 \\ S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2 \\ \neg test_in(S_2, V, S_1^{stp}[L_s]) \end{array}}{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{in}\ \mathsf{Nam} \Rightarrow discrete_val(0), S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{in}\ \mathsf{Nam} \Rightarrow discrete_val(1), S_2}{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{not\ in}\ \mathsf{Nam} \Rightarrow discrete_val(0), S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{in}\ \mathsf{Nam} \Rightarrow discrete_val(0), S_2}{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{not\ in}\ \mathsf{Nam} \Rightarrow discrete_val(1), S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{in}\ \mathsf{Rng} \Rightarrow discrete_val(1), S_2}{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{not\ in}\ \mathsf{Rng} \Rightarrow discrete_val(0), S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{in}\ \mathsf{Rng} \Rightarrow discrete_val(0), S_2}{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{not\ in}\ \mathsf{Rng} \Rightarrow discrete_val(1), S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{in}\ \mathsf{Nam} \Rightarrow discrete_val(1), S_2}{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{not\ in}\ \mathsf{Nam} \Rightarrow discrete_val(0), S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{in}\ \mathsf{Nam} \Rightarrow discrete_val(0), S_2}{S_1, E \vdash_{exp} \mathsf{Exp}\ \mathbf{not\ in}\ \mathsf{Nam} \Rightarrow discrete_val(1), S_2}$$

4.10.5 Type Conversions

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(L_s), S_1 \\ S_1 \left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow V_1 \\ E, S_1^{stp}[L_s] \vdash subtype_convert(V_1) \Rightarrow V_2 \end{array} \right] S_2 \end{array}}{S_1, E \vdash_{exp} \mathsf{Nam(Exp)} \Rightarrow V_2, S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam_1} \Rightarrow subtype_view(L_s), S_1 \\ S_1 \left[\begin{array}{c} E \vdash_{nam} \mathsf{Nam_2} \Rightarrow W_1 \\ E, S_1^{stp}[L_s] \vdash subtype_convert(W_1) \Rightarrow W_2 \end{array} \right] S_2 \end{array}}{S_1, E \vdash_{nam} \mathsf{Nam_1(Nam_2)} \Rightarrow W_2, S_2}$$

4.10.6 Qualified Expressions

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow \mathit{subtype_view}(L_s), S_1 \\ S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2 \\ \mathit{abnormal_state}(S_2) \end{array}}{S_1, E \vdash_{exp} \mathsf{Nam'(Exp)} \Rightarrow V, S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow \mathit{subtype_view}(L_s), S_1 \\ S_1^{stp}[L_s] = \mathit{subtype}(?, C, ?) \\ S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2 \\ \mathit{normal_state}(S_2) \\ \mathit{satisfies}(V, C) \end{array}}{S_1, E \vdash_{exp} \mathsf{Nam'(Exp)} \Rightarrow V, S_2}$$

$$\frac{\begin{array}{c} S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow \mathit{subtype_view}(L_s), S_1 \\ S_1^{stp}[L_s] = \mathit{subtype}(?, C, ?) \\ S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, \mathit{normal}(N) \\ \neg \mathit{satisfies}(V, C) \end{array}}{S_1, E \vdash_{exp} \mathsf{Nam'(Exp)} \Rightarrow V, \mathit{exception}(\mathit{constraint_error}, N)}$$

4.10.7 Allocators

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow V \\ \mathit{new_object_fn}(\mathit{invalid_val}, L) \end{array} \right] S_2}{S_1, E \vdash_{exp} \textbf{new}\ \mathsf{Exp} \Rightarrow \mathit{access_val}(\mathit{object_view}(\mathit{location}(L), ?, \mathit{aliased}), \mathit{none}), S_2}$$

$$\frac{\textit{not yet defined}}{S, E \vdash_{exp} \textbf{new}\ \mathsf{Sid} \Rightarrow V, S}$$

4.11 Statements

4.11.1 Statement Sequences

$$\frac{S_0 \left[\begin{array}{c} E \vdash_{stm} \mathsf{Stm_1} \Rightarrow \\ E \vdash_{stm} \mathsf{Stm_2} \ldots \Rightarrow \end{array} \right] S_1}{S_0, E \vdash_{stm} \mathsf{Stm_1}\ \mathsf{Stm_2} \ldots \Rightarrow S_1}$$

$$S, E \vdash_{stm} \langle\rangle \Rightarrow S$$

$$\frac{S, E \vdash_{stm} \mathsf{Stm} \Rightarrow S}{S, E \vdash_{stm} << \mathsf{Nam} >> \mathsf{Stm} \Rightarrow S}$$

$$S, E \vdash_{stm} \textbf{null};\ \Rightarrow S$$

4.11.2 Assignment Statements

$$\mathit{assignable}(\mathit{variable})$$

$$\mathit{assignable}(\mathit{aliased})$$

$$\frac{S_0\left\{\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow \mathit{object_view}(L, S_t, M) \\ E \vdash_{exp} \mathsf{Exp} \Rightarrow V \end{array}\right\} S_1 \quad \mathit{assignable}(M) \quad S_1\left[\begin{array}{c} E, S_t \vdash \mathit{subtype_convert}(V) \Rightarrow V_1 \\ \vdash \mathit{content}(L) \Rightarrow V_0 \\ \mathit{finalize_fn}(E, S_t, V_0) \\ \mathit{assign_fn}(E, S_t, L, V_1) \\ \mathit{value_split_fn}(E, S_t, V_1) \end{array}\right] S_2}{S_0, E \vdash_{stm} \mathsf{Nam} := \mathsf{Exp} \Rightarrow S_2}$$

$$\frac{S_0\left\{\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow \mathit{object_view}(L, S_t, \mathit{variable}) \\ E \vdash_{exp} \mathsf{Exp} \Rightarrow V \end{array}\right\} S_1 \quad \mathit{not\ yet\ defined}}{S_0, E \vdash_{stm} \mathsf{Nam} := \mathsf{Exp} \Rightarrow S_1}$$

4.11.3 If Statements

$$\frac{S_1\left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(1) \\ E \vdash_{stm} \mathsf{Stm}_1 \Rightarrow \end{array}\right] S_2}{S_1, E \vdash_{stm} \textbf{if } \mathsf{Exp} \textbf{ then } \mathsf{Stm}_1 \ \mathsf{Eif} \textbf{ else } \mathsf{Stm}_2 \textbf{ end if};\ \Rightarrow S_2}$$

$$\frac{S_1\left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(0) \\ E \vdash_{elf} \mathsf{Eif} \Rightarrow \mathit{true} \end{array}\right] S_2}{S_1, E \vdash_{stm} \textbf{if } \mathsf{Exp} \textbf{ then } \mathsf{Stm}_1 \ \mathsf{Eif} \textbf{ else } \mathsf{Stm}_2 \textbf{ end if};\ \Rightarrow S_2}$$

$$\frac{S_1\left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(0) \\ E \vdash_{elf} \mathsf{Eif} \Rightarrow \mathit{false} \\ E \vdash_{stm} \mathsf{Stm}_2 \Rightarrow \end{array}\right] S_2}{S_1, E \vdash_{stm} \textbf{if } \mathsf{Exp} \textbf{ then } \mathsf{Stm}_1 \ \mathsf{Eif} \textbf{ else } \mathsf{Stm}_2 \textbf{ end if};\ \Rightarrow S_2}$$

$$\frac{S_1\left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(1) \\ E \vdash_{stm} \mathsf{Stm}_1 \Rightarrow \end{array}\right] S_2}{S_1, E \vdash_{stm} \textbf{if } \mathsf{Exp} \textbf{ then } \mathsf{Stm}_1 \ \mathsf{Eif} \textbf{ end if};\ \Rightarrow S_2}$$

$$\frac{S_1\left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(0) \\ E \vdash_{elf} \mathsf{Eif} \Rightarrow \mathit{true} \end{array}\right] S_2}{S_1, E \vdash_{stm} \textbf{if } \mathsf{Exp} \textbf{ then } \mathsf{Stm}_1 \ \mathsf{Eif} \textbf{ end if};\ \Rightarrow S_2}$$

$$\frac{S_1\left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(0) \\ E \vdash_{elf} \mathsf{Eif} \Rightarrow \mathit{false} \end{array}\right] S_2}{S_1, E \vdash_{stm} \textbf{if } \mathsf{Exp} \textbf{ then } \mathsf{Stm}_1 \ \mathsf{Eif} \textbf{ end if};\ \Rightarrow S_2}$$

$$S, E \vdash_{elf} \langle\rangle \Rightarrow \mathit{false}, S$$

$$\frac{S_1, E \vdash_{elf} \mathsf{Eif}_1 \Rightarrow \mathit{true}, S_2}{S_1, E \vdash_{elf} \mathsf{Eif}_1\ \mathsf{Eif}_2\ \ldots \Rightarrow \mathit{true}, S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{elf} \mathsf{Eif}_1 \Rightarrow \mathit{false} \\ E \vdash_{elf} \mathsf{Eif}_2 \ \ldots \Rightarrow R \end{array}\right] S_2}{S_1, E \vdash_{elf} \mathsf{Eif}_1\ \mathsf{Eif}_2\ \ldots \Rightarrow R, S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(1) \\ E \vdash_{stm} \mathsf{Stm} \Rightarrow \end{array}\right] S_2}{S_1, E \vdash_{elf} \textbf{elsif}\ \mathsf{Exp}\ \textbf{then}\ \mathsf{Stm} \Rightarrow \mathit{true}, S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow \mathit{discrete_val}(0), S_2}{S_1, E \vdash_{elf} \textbf{elsif}\ \mathsf{Exp}\ \textbf{then}\ \mathsf{Stm} \Rightarrow \mathit{false}, S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow V \\ E, V \vdash_{clt} \mathsf{Alt} \Rightarrow \mathit{true} \end{array}\right] S_2}{S_1, E \vdash_{stm} \textbf{case}\ \mathsf{Exp}\ \textbf{is}\ \mathsf{Alt}\ \textbf{end case};\ \Rightarrow S_2}$$

4.11.4 Case Statements

$$\mathit{normal}(N), E, V \vdash_{clt} \langle\rangle \Rightarrow R, \mathit{exception}(\mathit{constraint_error}, N)$$

$$\frac{S_1, E, V \vdash_{clt} \mathsf{Alt}_1 \Rightarrow \mathit{true}, S_2}{S_1, E, V \vdash_{clt} \mathsf{Alt}_1\ \mathsf{Alt}_2\ \ldots \Rightarrow \mathit{true}, S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E, V \vdash_{clt} \mathsf{Alt}_1 \Rightarrow \mathit{false} \\ E, V \vdash_{clt} \mathsf{Alt}_2\ \ldots \Rightarrow R \end{array}\right] S_2}{S_1, E, V \vdash_{clt} \mathsf{Alt}_1\ \mathsf{Alt}_2\ \ldots \Rightarrow R, S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{chc} \mathsf{Dch} \Rightarrow C \\ \mathit{covers_fn}(V, C, \mathit{true}) \\ E \vdash_{stm} \mathsf{Stm} \Rightarrow \end{array}\right] S_2}{S_1, E, V \vdash_{clt} \textbf{when}\ \mathsf{Dch} => \mathsf{Stm} \Rightarrow \mathit{true}, S_2}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{chc} \mathsf{Dch} \Rightarrow C \\ \mathit{covers_fn}(V, C, \mathit{false}) \end{array}\right] S_2}{S_1, E, V \vdash_{clt} \textbf{when}\ \mathsf{Dch} => \mathsf{Stm} \Rightarrow \mathit{false}, S_2}$$

$$\frac{S, E \vdash_{stm} \mathsf{Stm} \Rightarrow \mathit{exit}(\mathit{unnamed}, N)}{S, E \vdash_{stm} \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow \mathit{normal}(N)}$$

4.11.4.1 Choices

$$\frac{S_1, E \vdash_{rng} \mathsf{Rng} \Rightarrow R, S_2}{S, E \vdash_{chc} \mathsf{Rng} \Rightarrow \mathit{choice_range}(R), S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, S_2}{S_1, E \vdash_{chc} \mathsf{Exp} \Rightarrow \mathit{choice_value}(V), S_2}$$

$$S, E \vdash_{chc} \textbf{others} \Rightarrow \mathit{choice_default}, S$$

$$S, E \vdash_{chc} \langle\rangle \Rightarrow \mathit{choice_lst}([\,]), S$$

$$\frac{\begin{array}{c} S_1, E \vdash_{chc} \mathsf{Dch}_1 \Rightarrow C_1, S_2 \\ S_2, E \vdash_{chc} \mathsf{Dch}_2 \mid \ldots \Rightarrow \mathit{choice_lst}(C_2), S_3 \end{array}}{S_1, E \vdash_{chc} \mathsf{Dch}_1 \mid \mathsf{Dch}_2 \mid \ldots \Rightarrow \mathit{choice_lst}(C_1 \cdot C_2), S_3}$$

$$covers(V, choice_value(V), true)$$

$$\frac{N_1 \neq N_2}{covers(discrete_val(N_1), choice_value(discrete_val(N_2)), false)}$$

$$\frac{belongs_to(V, R)}{covers(V, choice_range(R), true)}$$

$$\frac{\neg belongs_to(V, R)}{covers(V, choice_range(R), false)}$$

$$covers(V, choice_default, true)$$

$$covers(V, choice_lst([\,]), false)$$

$$\frac{covers(V, C_1, true)}{covers(V, choice_lst(C_1 \cdot C_2), true)}$$

$$\frac{\begin{array}{c} covers(V, C_1, false) \\ covers(V, choice_lst(C_2), R) \end{array}}{covers(V, choice_lst(C_1 \cdot C_2), R)}$$

4.11.5 Loop Statements

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{stm} \mathsf{Stm} \Rightarrow \\ E \vdash_{stm} \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow \end{array} \right] S_2}{S_1, E \vdash_{stm} \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow S_2}$$

$$\frac{\begin{array}{c} unique(X) \\ S_1, E_1[\mathsf{Idn} \mapsto loop_view(X)] \vdash_{stm} \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow normal(N) \end{array}}{S_1, E_1 \vdash_{stm} \mathsf{Idn} : \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow normal(N)}$$

$$\frac{\begin{array}{c} unique(X) \\ S_1, E_1[\mathsf{Idn} \mapsto loop_view(X)] \vdash_{stm} \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow exit(loop_id(X), N) \end{array}}{S_1, E_1 \vdash_{stm} \mathsf{Idn} : \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow normal(N)}$$

$$\frac{S \left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow discrete_val(1) \\ E \vdash_{stm} \mathsf{Stm} \Rightarrow \end{array} \right] exit(unnamed, N)}{S, E \vdash_{stm} \textbf{while}\ \mathsf{Exp}\ \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow normal(N)}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{exp} \mathsf{Exp} \Rightarrow discrete_val(1) \\ E \vdash_{stm} \mathsf{Stm} \Rightarrow \\ E \vdash_{stm} \textbf{while}\ \mathsf{Exp}\ \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow \end{array} \right] S_2}{S_1, E \vdash_{stm} \textbf{while}\ \mathsf{Exp}\ \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow S_2}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow discrete_val(0), S_2}{S_1, E \vdash_{stm} \textbf{while}\ \mathsf{Exp}\ \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow S_2}$$

$$\frac{\begin{array}{c} unique(X) \\ S_1, E_1[\mathsf{Idn} \mapsto loop_view(X)] \vdash_{stm} \textbf{while}\ \mathsf{Exp}\ \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow normal(N) \end{array}}{S_1, E_1 \vdash_{stm} \mathsf{Idn} : \textbf{while}\ \mathsf{Exp}\ \textbf{loop}\ \mathsf{Stm}\ \textbf{end loop};\ \Rightarrow normal(N)}$$

$$\frac{\begin{array}{c} unique(X) \\ S_1, E_1[\mathsf{Idn} \mapsto loop_view(X)] \vdash_{stm} \textbf{while } \mathsf{Exp} \textbf{ loop } \mathsf{Stm} \textbf{ end loop}; \Rightarrow exit(loop_id(X), N) \end{array}}{S_1, E_1 \vdash_{stm} \mathsf{Idn} : \textbf{while } \mathsf{Exp} \textbf{ loop } \mathsf{Stm} \textbf{ end loop}; \Rightarrow normal(N)}$$

4.11.6 Block Statements

$$\frac{S_1, E \vdash_{stm} \mathsf{Hsm} \Rightarrow S_2}{S_1, E \vdash_{stm} \textbf{begin } \mathsf{Hsm} \textbf{ end}; \Rightarrow S_2}$$

$$\frac{S_1, E \vdash_{stm} \mathsf{Hsm} \Rightarrow S_2}{S_1, E \vdash_{stm} \mathsf{Nam} : \textbf{begin } \mathsf{Hsm} \textbf{ end}; \Rightarrow S_2}$$

$$\frac{\begin{array}{c} S_1 \left[\begin{array}{c} E_1 \vdash_{dcl} \mathsf{Dcl} \Rightarrow E_2 \\ E_2 \vdash_{stm} \mathsf{Hsm} \Rightarrow \end{array} \right] S_2 \\ S_2, E_2 \vdash finalize(B) \Rightarrow S_3 \end{array}}{S_1, E_1 \vdash_{stm} \textbf{declare } \mathsf{Dcl} \textbf{ begin } \mathsf{Hsm} \textbf{ end}; \Rightarrow S_3}$$

$$\frac{\begin{array}{c} S_1 \left[\begin{array}{c} E_1 \vdash_{dcl} \mathsf{Dcl} \Rightarrow E_2 \\ E_2 \vdash_{stm} \mathsf{Hsm} \Rightarrow \end{array} \right] S_2 \\ S_2, E_2 \vdash finalize(B) \Rightarrow S_3 \end{array}}{S_1, E_1 \vdash_{stm} \mathsf{Nam} : \textbf{declare } \mathsf{Dcl} \textbf{ begin } \mathsf{Hsm} \textbf{ end}; \Rightarrow S_3}$$

4.11.7 Exit Statements

$$normal(N), E \vdash_{stm} \textbf{exit}; \Rightarrow exit(unnamed, N)$$

$$\frac{S, E \vdash_{nam} \mathsf{Nam} \Rightarrow loop_view(X), normal(N)}{S, E \vdash_{stm} \textbf{exit } \mathsf{Nam}; \Rightarrow exit(loop_id(X), N)}$$

$$\frac{S, E \vdash_{exp} \mathsf{Exp} \Rightarrow discrete_val(1), normal(N)}{S, E \vdash_{stm} \textbf{exit when } \mathsf{Exp}; \Rightarrow exit(unnamed, N)}$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow B, S_2}{S_1, E \vdash_{stm} \textbf{exit when } \mathsf{Exp}; \Rightarrow S_2}$$

$$\frac{S \left[\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow loop_view(X) \\ E \vdash_{exp} \mathsf{Exp} \Rightarrow discrete_val(1) \end{array} \right] normal(N)}{S, E \vdash_{stm} \textbf{exit } \mathsf{Nam} \textbf{ when } \mathsf{Exp}; \Rightarrow exit(loop_id(X), N)}$$

$$\frac{S_1 \left[\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow loop_view(X) \\ E \vdash_{exp} \mathsf{Exp} \Rightarrow discrete_val(0) \end{array} \right] S_2}{S_1, E \vdash_{stm} \textbf{exit } \mathsf{Nam} \textbf{ when } \mathsf{Exp}; \Rightarrow S_2}$$

4.12 Subprograms

4.12.1 Subprogram Declarations

$$\frac{S_1, E_1 \vdash_{pas} \mathsf{Pms} \Rightarrow A, S_2 \quad new_subprogram(S_2, unelaborated, U, S_3)}{S_1, E_1 \vdash_{dcl} \textbf{procedure}\ \mathsf{Idn}\ \mathsf{Pms};\ \Rightarrow E_1[\mathsf{Idn} \mapsto subprogram_view(U, A, none)], S_3}$$

$$\frac{S_1, E_1 \vdash_{pas} \mathsf{Pms} \Rightarrow A, S_2 \quad S_2, E_1 \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(S_l), S_2 \quad new_subprogram(S_2, unelaborated, U, S_3)}{S_1, E_1 \vdash_{dcl} \textbf{function}\ \mathsf{Idn}\ \mathsf{Pms};\ \Rightarrow E_1[\mathsf{Idn} \mapsto subprogram_view(U, A, some(S_2^{stp}[S_l]))], S_3}$$

$$\frac{S_1, E \vdash_{pas} \mathsf{Pms} \Rightarrow A, S_2 \quad E \vdash_{lookup} \mathsf{Idn} \Rightarrow subprogram_view(U, A, none)}{S_1, E \vdash_{dcl} \textbf{procedure}\ \mathsf{Idn}\ \mathsf{Pms}\ \textbf{is}\ \mathsf{Dcl}\ \textbf{begin}\ \mathsf{Stm}\ \textbf{end};\ \Rightarrow E, S_2[U \mapsto_2 subprogram(E, adom(A), \mathsf{Dcl}, \mathsf{Stm})]}$$

$$\frac{S_1, E \vdash_{pas} \mathsf{Pms} \Rightarrow A, S_2 \quad S_2, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(S_l), S_2 \quad E \vdash_{lookup} \mathsf{Idn} \Rightarrow subprogram_view(U, A, some(S_2^{stp}[S_l]))}{S_1, E \vdash_{dcl} \textbf{function}\ \mathsf{Idn}\ \mathsf{Pms}\ \textbf{is}\ \mathsf{Dcl}\ \textbf{begin}\ \mathsf{Stm}\ \textbf{end};\ \Rightarrow E, S_2}$$

$$S, E \vdash_{pas} \langle\rangle \Rightarrow [\,], S$$

$$\frac{not\ yet\ defined}{S, E \vdash_{pas} \mathsf{Idn} : \textbf{access}\ \mathsf{Nam};\ \mathsf{Pms};\ \ldots \Rightarrow A, S}$$

$$\frac{not\ yet\ defined}{S, E \vdash_{pas} \mathsf{Idn} : \textbf{access}\ \mathsf{Nam} := \mathsf{Exp};\ \mathsf{Pms};\ \ldots \Rightarrow A, S}$$

$$\frac{\vdash_{mod} \mathsf{Mde} \Rightarrow M \quad S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(S_l), S_1 \quad S_1, E \vdash_{pas} \mathsf{Pms};\ \ldots \Rightarrow A_1, S_2}{S_1, E \vdash_{pas} \mathsf{Idn} : \mathsf{Mde}\ \mathsf{Nam} := \mathsf{Exp};\ \mathsf{Pms};\ \ldots \Rightarrow A_1[\mathsf{Idn} \mapsto formal(M, S_1^{stp}[S_l], some(thunk(E, \mathsf{Exp})))], S_2}$$

$$\frac{\vdash_{mod} \mathsf{Mde} \Rightarrow M \quad S_1, E \vdash_{nam} \mathsf{Nam} \Rightarrow subtype_view(S_l), S_1 \quad S_1, E \vdash_{pas} \mathsf{Pms};\ \ldots \Rightarrow A_1, S_2}{S, E \vdash_{pas} \mathsf{Idn} : \mathsf{Mde}\ \mathsf{Nam};\ \mathsf{Pms};\ \ldots \Rightarrow A, S}$$

4.12.2 Formal Parameter Modes

$$\vdash_{mod} \textbf{in} \Rightarrow in_mode$$

$$\vdash_{mod} \textbf{in out} \Rightarrow in_out_mode$$

$$\vdash_{mod}\ \Rightarrow in_mode$$

$$\vdash_{mod} \textbf{out} \Rightarrow out_mode$$

4.12.3 Subprogram Bodies

There are no semantics associated with subprogram bodies. The declarations and the statement part of a subprogram body, together with the declaration environment, are stored as a subprogram value. The rules given below define the effect of executing a subprogram value.

The following definition creates an environment for the execution of a procedure body by binding the formal parameter names to the views of the actual parameters. The actual parameters are given as an association but, due to renaming, the names in the association may differ from those of the formal parameters.

$$bind_actuals(E, [\,], [\,], E)$$

$$\frac{bind_actuals(E_1[I_1 \mapsto W_1], P, \mathsf{Ids}, E_3)}{bind_actuals(E_1, (P_1 \times W_1) \cdot P, I_1 \cdot \mathsf{Ids}, E_3)}$$

$$proc_exit(exception(X, N), exception(X, N))$$

$$\frac{not\ yet\ defined}{proc_exit(exit(?, N), ?)}$$

$$\frac{not\ yet\ defined}{proc_exit(func_return(?, N), ?)}$$

$$proc_exit(proc_return(N), normal(N))$$

$$proc_exit(normal(N), normal(N))$$

$$\frac{\begin{array}{c} S_1^{spg}[L] = subprogram(E_1, \mathsf{Ids}, \mathsf{Dcl}, \mathsf{Stm}) \\ bind_actuals(E_1, A, \mathsf{Ids}, E_2) \\ S_1 \left[\begin{array}{c} E_2 \vdash_{dcl} \mathsf{Dcl} \Rightarrow E_3 \\ E_3 \vdash_{stm} \mathsf{Stm} \Rightarrow \end{array} \right] S_2 \\ proc_exit(S_2, S_3) \end{array}}{subprogram_body(S_1, A, subprogram_view(L, A_f, \mathsf{none}), S_3)}$$

$$return_check(exception(X, N), S_t, exception(X, N))$$

$$\frac{not\ yet\ defined}{return_check(exit(?, N), S_t, S)}$$

$$\frac{not\ yet\ defined}{return_check(proc_return(N), S_t, S)}$$

$$\frac{convert_return_value(normal(N), W_1, S_t, W_2)}{return_check(func_return(W_1, N), S_t, func_return(W_2, N))}$$

$$return_check(normal(N), S_t, exception(program_error, N))$$

$$\frac{\begin{array}{c} S_1^{spg}[L] = subprogram(E_1, \mathsf{Ids}, \mathsf{Dcl}, \mathsf{Stm}) \\ bind_actuals(E_1, A, \mathsf{Ids}, E_2) \\ S_1 \left[\begin{array}{c} E_2 \vdash_{dcl} \mathsf{Dcl} \Rightarrow E_3 \\ E_3 \vdash_{stm} \mathsf{Stm} \Rightarrow \end{array}\right] S_2 \\ return_check(S_2, S_t, S_3) \end{array}}{subprogram_body(S_1, A, subprogram_view(L, A_f, some(S_t)), S_3)}$$

Appropriate rules need to be defined for all predefined operators.

$$\frac{\begin{array}{c} S_1^{spg}[L] = operator(\mathsf{Opn}) \\ not\ yet\ defined \end{array}}{subprogram_body(S_1, A, subprogram_view(L, A_f, ?), S_2)}$$

A call to an unelaborated subprogram raises program error.

$$\frac{normal(N)^{spg}[L] = \mathbf{\mathit{unelaborated}}}{subprogram_body(normal(N), A, subprogram_view(L, A_f, ?), exception(program_error, N))}$$

4.12.4 Subprogram Calls

The rules given here are incomplete and do not describe subtype and view conversions that are part of a call.

$$return_value(func_return(W, N), W, normal(N))$$

$$return_value(exception(X, N), ?, exception(X, N))$$

$$\frac{\begin{array}{c} parameter_list(E, \mathsf{Pss}, A_f, A_a, P) \\ W = subprogram_view(?, A_f, ?) \\ S_1 \left[\begin{array}{c} \left\{\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow W \\ P_1 \\ \dots \\ P_n \end{array}\right\} \\ subprogram_body_fn(A_a, W) \end{array}\right] S_2 \\ return_value(S_2, W_r, S_3) \end{array}}{S_1, E \vdash_{nam} \mathsf{Nam(Pss, \ldots)} \Rightarrow W_r, S_3}$$

$$\frac{\begin{array}{c} parameter_list(E, \mathsf{Pss}, A_f, A_a, P) \\ W = subprogram_view(?, A_f, ?) \\ S_1 \left[\begin{array}{c} \left\{\begin{array}{c} E \vdash_{nam} \mathsf{Nam} \Rightarrow W \\ P_1 \\ \dots \\ P_n \end{array}\right\} \\ subprogram_body_fn(A_a, W) \end{array}\right] S_2 \end{array}}{S_1, E \vdash_{stm} \mathsf{Nam(Pss, \ldots);} \Rightarrow S_2}$$

4.12.4.1 Parameter Associations

$$\mathit{parameter_list}(E, \mathsf{Pss}, [\,], [\,], [\,])$$

$$\frac{\mathit{parameter_list}(E, \mathsf{Pss}, F, R, A) \quad \mathit{parameter_action}(E, \mathsf{Pss}, F_1, R_1, A_1)}{\mathit{parameter_list}(E, \mathsf{Pss}, F_1 \cdot F, R_1 \cdot R, A_1 \cdot A)}$$

$$\frac{\neg \mathit{given_parameter}(\mathsf{Pss}, \mathsf{Idn}, ?)}{\mathit{parameter_action}(E, \mathsf{Pss}, (\mathsf{Idn} \times \mathit{formal}(\mathit{in_mode}, S_t, \mathit{some}(\mathit{thunk}(E_1, \mathsf{Exp})))), (\mathsf{Idn} \times \mathit{constant_view}(V)), E_1 \vdash_{exp} \mathsf{Exp}}$$

$$\frac{\mathit{given_parameter}(\mathsf{Pss}, \mathsf{Idn}, \mathsf{Exp})}{\mathit{parameter_action}(E, \mathsf{Pss}, (\mathsf{Idn} \times \mathit{formal}(\mathit{in_mode}, S_t, ?)), (\mathsf{Idn} \times \mathit{constant_view}(V)), E \vdash_{exp} \mathsf{Exp} \Rightarrow V)}$$

$$\frac{\mathit{the_parameter}(\mathsf{Pss}, \mathsf{Idn}, \mathsf{Nam})}{\mathit{parameter_action}(E, \mathsf{Pss}, (\mathsf{Idn} \times \mathit{formal}(\mathit{out_mode}, S_t, ?)), (\mathsf{Idn} \times W), E \vdash_{nam} \mathsf{Nam} \Rightarrow W)}$$

$$\frac{\mathit{the_parameter}(\mathsf{Pss}, \mathsf{Idn}, \mathsf{Nam})}{\mathit{parameter_action}(E, \mathsf{Pss}, (\mathsf{Idn} \times \mathit{formal}(\mathit{in_out_mode}, S_t, ?)), (\mathsf{Idn} \times W), E \vdash_{nam} \mathsf{Nam} \Rightarrow W)}$$

$$\mathit{given_parameter}(\mathsf{Idn} => \mathsf{Exp} \cdot \mathsf{Pss}, \mathsf{Idn}, \mathsf{Exp})$$

$$\frac{\mathsf{Idn}_1 \neq \mathsf{Idn}_2 \quad \mathit{given_parameter}(\mathsf{Pss}, \mathsf{Idn}_2, \mathsf{Exp})}{\mathit{given_parameter}(\mathsf{Idn}_1 =>? \cdot \mathsf{Pss}, \mathsf{Idn}_2, \mathsf{Exp})}$$

$$\mathit{the_parameter}(\mathsf{Idn} => \mathsf{Nam} \cdot \mathsf{Pss}, \mathsf{Idn}, \mathsf{Nam})$$

$$\frac{\mathsf{Idn}_1 \neq \mathsf{Idn}_2 \quad \mathit{the_parameter}(\mathsf{Pss}, \mathsf{Idn}_2, \mathsf{Nam})}{\mathit{the_parameter}(\mathsf{Idn}_1 =>? \cdot \mathsf{Pss}, \mathsf{Idn}_2, \mathsf{Nam})}$$

4.12.5 Return Statements

$$\mathit{normal}(N), E \vdash_{stm} \mathbf{return};\ \Rightarrow \mathit{proc_return}(N)$$

$$\frac{S_1, E \vdash_{exp} \mathsf{Exp} \Rightarrow V, \mathit{normal}(N)}{S_1, E \vdash_{stm} \mathbf{return}\ \mathsf{Exp};\ \Rightarrow \mathit{func_return}(\mathit{constant_view}(V), N)}$$

The following needs to be defined to describe the rules of [6.5(6)] through [6.5(21)].

$$\frac{\textit{not yet defined}}{\mathit{convert_return_value}(S, W_1, S_t, W_2)}$$

4.13 Attributes

$$\frac{\begin{array}{c} is_scalar_type(S_1, S_1^{stp}[L_s]) \\ low_bound(range_of_subtype(S_1, S_1^{stp}[L_s])) = V \\ new_object(S_1, V, L, S_2) \end{array}}{S_1, E, subtype_view(L_s) \vdash_{att} \textsf{first} \Rightarrow object_view(L, S_1^{stp}[L_s], constant), S_2}$$

$$\frac{\begin{array}{c} is_scalar_type(S_1, S_1^{stp}[L_s]) \\ high_bound(range_of_subtype(S_1, S_1^{stp}[L_s])) = V \\ new_object(S_1, V, L, S_2) \end{array}}{S_1, E, subtype_view(L_s) \vdash_{att} \textsf{last} \Rightarrow object_view(L, S_1^{stp}[L_s], constant), S_2}$$

$$\frac{\begin{array}{c} S_1^{stp}[L_s] = subtype(U, C, A) \\ is_scalar_type(S_1, subtype(U, C, A)) \\ new_subtype(S_1, subtype(U, no_constraint, A), L_a, S_2) \end{array}}{S_1, E, subtype_view(L_s) \vdash_{att} \textsf{base} \Rightarrow subtype_view(L_a), S_2}$$

$$\frac{\begin{array}{c} S_1^{stp}[L_s] = subtype(?, index_constraint(R \cdot R_s), ?) \\ new_object(S_1, low_bound(R), L, S_2) \end{array}}{S_1, E, subtype_view(L_s) \vdash_{att} \textsf{first} \Rightarrow object_view(L, S_t, constant), S_2}$$

$$\frac{\begin{array}{c} S_1^{stp}[L_s] = subtype(?, index_constraint(R \cdot R_s), ?) \\ new_object(S_1, high_bound(R), L, S_2) \end{array}}{S_1, E, subtype_view(L_s) \vdash_{att} \textsf{last} \Rightarrow object_view(L, S_t, constant), S_2}$$

$$\frac{\begin{array}{c} S_1^{stp}[L_s] = subtype(?, index_constraint(R \cdot R_s), ?) \\ low_bound(R) = discrete_val(N_l) \\ S_t = subtype(universal_integer_tn, no_constraint, ?) \\ high_bound(R) = discrete_val(N_h) \\ new_object(S_1, discrete_val(Nh - Nl + 1), L, S_2) \end{array}}{S_1, E, subtype_view(L_s) \vdash_{att} \textsf{length} \Rightarrow object_view(L, S_t, constant), S_2}$$

$$\frac{\begin{array}{c} S_1, E, W \vdash_{att} \textsf{first} \Rightarrow object_view(L_1, ?, ?), S_2 \\ S_2, E, W \vdash_{att} \textsf{last} \Rightarrow object_view(L_2, ?, ?), S_3 \\ S_3 \vdash content(L_1) \Rightarrow V_1 \\ S_3 \vdash content(L_2) \Rightarrow V_2 \end{array}}{S_1, E, W \vdash_{att_r} \textsf{range} \Rightarrow make_range(V_1, V_2), S_3}$$

Chapter 5

Exceptions and Optimization

5.1 Introduction

Version 5.0 of the Annotated Draft Ada 9X reference manual [4] contains language that obviates many of the problems associated with section 11.6 of the Ada 83 reference manual [10]. The purpose of this chapter is twofold. The first is to examine the Ada revision as represented by Version 5.0 in light of the earlier Language Precision Team work in this area as published in the LPT Task 1 report [9]. The second is to discuss the consequences of the remaining problems that the semantics of Ada 9X present in the areas of predictability and to offer suggestions for accommodating them in practice. The report concludes with a brief commentary on the Annotated Draft used to support this research.

5.2 The Ada 9X revision of 11.6

Section 11.6 of the Ada 83 reference manual contained explicit permissions to reorder operations or to omit some checks that might propagate predefined exceptions. In Ada 83 the notion of the "effect" of a program or of an operation was not as clearly defined as it is in Ada 9X and the language of the section gave rise to endless discussions such as those captured in AI-315.

As revised, [11.6] contains two substantive paragraphs, (5) and (7). The first gives permission to avoid raising exceptions under some circumstances. The second permits more extensive reordering of operations than was generally considered permissible in Ada 83 by relaxing the requirements for state predictability when an exception handler is entered.

5.2.1 [11.6(5)]

This paragraph allows the implementation to avoid raising exceptions in the face of failures of predefined language exceptions under some circumstances. In the context of a clearer notion of "effect," it is somewhat of an improvement over the language of Ada 83. Even so, the language used in [11.6] is less clear than it might be. Consider the language of [RM-83 11.6(7)]:

> A predefined operation need not be invoked at all, if its only possible effect is to propagate a predefined exception. Similarly, a predefined operation need not be invoked if the removal of subsequent operations by the above rule renders this invocation ineffective.

In Ada 83 the term *effect* is not defined[1] and the meaning of the term is the subject of considerable discussion in AI-315 and elsewhere. The gist of many of the discussions concerns the case in which the

[1] The index entry for "effect" in [RM-83 Appendix I] is "[see: elaboration has no other effect]."

programmer has apparently written an operation that is sure to raise an exception as a "shorthand" for a **raise** statement. While this should be considered to be poor programming style, suppressing the operation leads to surprising effects.

We note that Chapter 14 of the Ada 83 rationale[2] which deals with exceptions does not discuss the material contained in [RM-83 11.6] and a search through the text of the rationale for the root string "optimiz" does not provide any appropriate insight.

From the discussions contained in AI-315 it appears that the primary need that the language of [RM-83 11.6(7)] is attempting to capture is the desire to remove code that is dead along its normal execution path even if executing it may (or is certain to) raise an exception due to the failure of a language-defined check. According to [11.6(7.f)], the language of [RM-83 11.6(7)] is now reflected in paragraph [11.6(5)] which reads:

> An implementation need not always raise an exception when a language-defined check fails. Instead, the operation that failed the check can simply yield an *undefined result*. The exception need be raised by the implementation only if, in the absence of raising it, the value of this undefined result would have some effect on the external interactions of the program. In determining this, the implementation shall not presume that an undefined result has a value that belongs to its subtype, nor even to the base range of its type, if scalar. [Having removed the raise of the exception, the canonical semantics will in general allow the implementation to omit the code for the check, and some or all of the operation itself.] 5

5.2.1.1 Defining *undefined*

Unfortunately, the index for Version 5.0 contains exactly one entry for *undefined result*, i.e., [11.6(5)]. Although this reference purports to define *undefined result*, we are given no useful semantics to associate with the term. Thus we are left to attempt to define exactly what is meant by the phrase through other means. A search of the source text for Version 5.0 yields several additional uses of the word *undefined*. The ones that appear to be related to its use in [11.6(5)] are:

13.9.1 NOTES

> 19 Objects can become abnormal due to other kinds of actions that directly update the object's representation; such actions are generally considered directly erroneous, however. 14
>
> *Wording Changes From Ada 83*
>
> In order to reduce the amount of erroneousness, we separate the concept of an undefined value into objects with invalid representation (scalars only) and abnormal objects. 14.a
>
> Reading an object with an invalid representation is a bounded error rather than erroneous; reading an abnormal object is still erroneous. In fact, the only safe thing to do to an abnormal object is to assign to the object as a whole. 14.b

3.8.1

> • The discrete_choice **others** covers all values of its expected type that are not covered by previous discrete_choice_lists of the same construct. 12
>
> **Ramification:** For case_statements, this includes values outside the range of the static subtype (if any) to be covered by the choices. It even includes values outside the base range of the case expression's type, since values of numeric types (and undefined values of any scalar type?) can be outside their base range. 12.a

7.4

> *Extensions to Ada 83*

[2] This document may be obtained by anonymous ftp from **ajpo.sei.cmu.edu** in the directory **public/rationale**.

The rules for too-early uses of deferred constants are modified in Ada 9X to allow more cases, and catch all errors at compile time. This change is necessary in order to allow deferred constants of a tagged type without violating the principle that for a dispatching call, there is always an implementation to dispatch to. It has the beneficial side-effect of catching some Ada-83-erroneous programs at compile time. The new rule fits in well with the new freezing-point rules. Furthermore, we are trying to convert undefined-value problems into bounded errors, and we were having trouble for the case of deferred constants. Furthermore, uninitialized deferred constants cause trouble for the shared variable / tasking rules, since they are really variable, even though they purport to be constant. In Ada 9X, they cannot be touched until they become constant. 15.e

The first item seems to be the key. The remaining two items use the word *undefined* in ways that seem to confirm the impressions given by [13.9.1] as a whole. Thus, we see that *undefined* either applies to a scalar object with an invalid representation or to an abnormal object. Abnormal objects can either be produced by disrupted assignments (with a reference from [13.9.1(5)] back to [11.6], presumably to [11.6(6)]) or (for non-scalars) by a return from a call to either a language defined input procedure or to an imported procedure. It is, perhaps, stretching things to call the latter an *operation* in the sense of the discussion of [3.2].

Discussion: An *operation* is a program entity that operates on zero or more operands to produce an effect, or yield a result, or both. 10.a

It seems more likely that the operations referred to are akin to the *primitive operations* partially defined in 3.2.

A *type* is characterized by a set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. 1

This leads us to consider the invalid values that can be associated with scalar objects and the predefined operations on scalar types. These are discussed in general in [4.5] where the relevant language appears in [4.5(9)–4.5(12)].

For each form of type definition, certain of the above operators are *predefined*; that is, they are implicitly declared immediately after the type definition. For each such implicit operator declaration, the parameters are called Left and Right for *binary* operators; the single parameter is called Right for *unary* operators. [An expression of the form X op Y, where op is a binary operator, is equivalent to a function_call of the form "op"(X, Y). An expression of the form op Y, where op is a unary operator, is equivalent to a function_call of the form "op"(Y). The predefined operators and their effects are described in subclauses 4.5.1 through 4.5.6.] 9

Dynamic Semantics

[The predefined operations on integer types either yield the mathematically correct result or raise the exception Constraint_Error. The predefined operations on real types yield results whose accuracy is defined in Annex G, or raise the exception Constraint_Error.] 10

To be honest: Predefined operations on real types can "silently" give wrong results when the Machine_Overflows attribute is false, and the computation overflows. 10.a

Implementation Requirements

> 11 The implementation of a predefined operator that delivers a result of an integer or fixed point type may raise Constraint_Error only if the result is outside the base range of the result type.
>
> 12 The implementation of a predefined operator that delivers a result of a floating point type may raise Constraint_Error only if the result is outside the safe range of the result type.

Unfortunately, there is a reading of this language that would make it impossible for a predefined operation on integer types to produce an invalid value. Paragraph (10) requires the operation to either yield the mathematically correct result or to raise Constraint_error. Paragraph (11) states that the implementation of the predefined operation may raise Constraint_error only if the result is outside the base range of the result type. Now, if we assume that "result" in paragraph (11) is the value produced by the implementation, it is almost certainly the case that this result will be within the range of the base type; it just will not be mathematically correct. It is likely that what is intended is

> The implementation of a predefined operator that delivers a result of an integer or fixed point type may raise Constraint_Error only if the [mathematically correct] result [of the operation] is outside the base range of the result type.

As the language is currently defined, there is a direct contradiction between the language of [4.5(10-11)] and that of [11.6(5)]

If we assume the revised interpretation, then we have a class of operations that can produce results that are not mathematically correct though they will typically be precisely defined by the implementation. If 11.6(5) is to have any reasonable meaning, it must be the case that results of this kind are the *undefined* results referred to. If this is the case, we have extended the notion of invalid to include representations of scalar objects that do represent values of the object's subtype but are not the mathematically correct values that would be produced without the violated constraint.[3] **This is a fairly serious extension and deserves more consideration.** We will return to this shortly.

5.2.1.2 Use of *undefined* results

The implementation note associated with [11.6(5)] seems to raise two distinct points. One, allowing the removal of dead code, is fairly obvious and seems to be the only clear-cut case. The other discusses implementation assumptions and seems to involve the extension noted above.

> 5.d **Implementation Note:** This permission is intended to allow normal "dead code removal" optimizations, even if some of the removed code might have failed some language-defined check. However, one may not eliminate the raise of an exception if subsequent code presumes in some way that the check succeeded. For example:
>
> 5.e
> ```
> if X * Y > Integer'Last then
>     Put_Line("X * Y overflowed");
> end if;
> ```

[3] Addition in a 2's complement n bit machine produces a result that is either a mathematically correct integer result or the mathematically correct integer result minus 2^n. Consider a 2 bit, 2's complement machine. Its value set is given as

i_2	i_{10}
00	0
01	1
10	-2
11	-1

and + defined as

+	00 (0)	01 (1)	10 (-2)	11 (-1)
00 (0)	00 (0)	01 (1)	10 (-2)	11 (-1)
01 (1)	01 (1)	*10 (-2)*	11 (-1)	00 (0)
10 (-2)	10 (-2)	11 (-1)	*00 (0)*	*01 (1)*
11 (-1)	11 (-1)	00 (0)	*01 (1)*	10 (-2)

where the values in *italics* are not mathematically correct integer results but are both well-defined and have a valid integer representation.

```
exception
  when others =>
      Put_Line("X * Y overflowed");
```

If X*Y does overflow, you may not remove the raise of the exception if the code that does the comparison against Integer'Last presumes that it is comparing it with an in-range Integer value, and hence always yields False.

As another example where a raise may not be eliminated:

5.f

5.g

```
  subtype Str10 is String(1..10);
  type P10 is access Str10;
  X : P10 := null;
begin
  if X.all'Last = 10 then
      Put_Line("Oops");
  end if;
```

In the above code, it would be wrong to eliminate the raise of Constraint_Error on the "X.all" (since X is null), if the code to evaluate 'Last always yields 10 by presuming that X.all belongs to the subtype Str10, without even "looking."

The first point is that if the result of an operation is not subsequently used, then we can ignore the possibility that execution of an operation might have raised an exception. Examples that illustrate this situation are somewhat contrived since programmers generally do not try to write code that is not useful. For example, we might illustrate the permission by writing something like:

```
  subtype Str10 is String(1..10);
  X : Str10 := " ";
begin
  X := "2 01";
    Put_line(X);
  X := "8 01234567";
    Put_line(X);
  X := "10 0123456789";
exception
  when others =>
      Put_Line("OOPS");
end;
```

Since the scope of X does not extend beyond the end of the block, the value produced by the last assignment has no effect along the normal path of execution. [11.6(5)] gives permission to ignore the possibility (in this case, a certainty) that the assignment will raise Constraint_Error. This, in turn, allows the elision of the entire assignment statement using conventional, "dead code" elimination techniques.

Note that without the extra permission of [11.6(5)], the code for the last assignment is not dead since there is a a well-defined "effect" along the exceptional[4] path. The extra permission allows the implementation to restrict its analysis to the normal path. This is important since exception handlers are dynamically bound and an analysis that shows that an operation is dead along its exceptional path is generally intractable, while one that shows that the operation is dead along its normal path may require only local analysis.

While the example is contrived, the situation that it presents appears fairly frequently as the result of other transformations during code generation and optimization. For example, unrolling

[4] See the definitions of *exceptional* and *normal* paths below.

a loop may well leave dead code in the final iteration, e.g., the code intended to initialize the next iteration. Value propagation and common subexpression elimination also serve to create dead variables and dead code to manipulate them.

The language of [11.6(5)] does not ensure that the obviously intended meanings of some realistic examples will be preserved. For example, in AI-315, Robert Dewar presents the following example:

```
function Add_overflows(A, B: Integer) return Boolean is
  T: Integer
begin
  T := A + B;
  return False;
exception
  when Constraint_Error => return True;
end Add_Overflows;
```

The writer of code like this might hope to detect a potential overflow situation and, perhaps, use the knowledge to invoke an alternate more robust computation, however, it appears that the permissions of [11.5] would allow the constraint_error to be ignored, rendering the assignment dead and allowing its elimination permitting the function to always return False. This could cascade, if for example, the function were to be inlined, eliminating the code for the alternate computation which would also appear dead.

Note that both Dewar's example and the example of [11.6(5.e)] have the same intent, the detection of overflow. They differ in minor details with respect to the way the overflow is detected. It is probably unreasonable to expect a casual (or even experienced) user of the language to detect the subtleties. Indeed, the casual observer ought to come to the conclusion that the example of [11.6(5.e)] cannot work because the implementation result (as opposed to the mathematically correct result) of `X * Y` cannot possibly be larger than `Integer'Last` so that the first `Put_Line` cannot appear. This would lead the user towards an example similar to Dewar's which apparently will not work. It is not clear that there is any easy fix. The approach offered with respect to code motion in [11.6(6)], a compromise that allows local code motion with local analysis, but does not insist on global analysis to ensure that the permitted code motion does not disrupt the canonical semantics might also apply here. This would require that analysis proceed along both the normal and exceptional paths following from an operation if there were an exception handler for the exception potentially raised by the operation associated with the innermost sequence of statements containing the operation.

The examples given in (5.e) and (5.g) raise more subtle points. In the absence of [11.6(5)], Ada's exception model is similar to that of Gypsy. If we use a Gypsy-like model to specify the Ada operations, we get a possibility of two execution paths from each operation [6]. We will call these paths the *normal* and *exceptional* paths. If none of the language-defined checks fail during the performance of the operation, execution proceeds along the normal path. If performance of the operation causes a language-defined check to fail, execution proceeds along the exceptional path. Associated with each operation is an entry specification which is assumed[5] to be true when the operation is invoked. Associated with each exit path is an exit specification which is guaranteed to hold if the path is followed.

For example, the implementation of the integer multiplication operation on a given machine might be specified as follows:

```
function Machine_Mul(X, Y : Machine_Integer)
          return Machine_Integer
  entry
```

[5] "Assumed" is with respect to the operator definition. The implementation is required to "prove" that the assumption holds every time the operation is invoked. In the case of operations such as `Machine_Mul` and `Machine_CMP` all possible bit patterns represent valid values of `Machine_Integer` and the entry condition is trivially satisfied.

```
X, Y in (Machine_Integer'First .. Machine_Integer'Last);
normal exit
  Machine_Mul(X, Y) = Integer_Mul(X, Y) and
  Machine_Mul(X, Y) in
    (Machine_Integer'First .. Machine_Integer'Last);
exceptional exit
  Machine_Mul(X, Y) != Integer_Mul(X, Y) and
  Machine_Mul(X, Y) in
    (Machine_Integer'First .. Machine_Integer'Last) and
  Integer_Mul(X, Y) not in
    (Machine_Integer'First .. Machine_Integer'Last);
```

This says that for some (possibly empty) set of input values, machine multiplication is equivalent to abstract integer multiplication and that execution will proceed along the normal path when this is the case. When machine and integer multiplication do not produce the same result, we are told that the true result is not representable as a `Machine_Integer` but that some result[6] representable as a machine integer is produced.[7] Now, if we look at a possible specification for the comparison operator, >, we see a potential problem with the language of the note.

```
function Machine_CMP(X, Y : Machine_Integer)
          return Machine_CC
  entry
    X, Y in (Machine_Integer'First .. Machine_Integer'Last);
  normal exit
    Machine_CMP(X, Y) = GT implies Integer_GT(X, Y) and
    ... -- Specifications for other return values
  exceptional exit
    false;
```

In this case, we assume a comparison instruction at the machine level that sets some condition codes to indicate the results of the comparison. GT is a condition code value that indicates the first operand, interpreted as an abstract integer, was greater than the second operand, also interpreted as an abstract integer. Note that the only entry condition assumes that the inputs are machine integers. This condition is satisfied by the exit condition of the multiply operation under either its normal or exceptional execution. Note also that this operation is defined to always exit normally.

We note that, in program verification, an operational semantics that allows exceptions to be raised when a language-defined check fails is, in a sense a dual of an operational semantics that produces an *undefined* result under the same circumstances. In the absence of a way to effect a meaningful recovery from failed checks,[8] we must show that the exceptional path is not taken. The proofs involved are exactly those that are required to show that operations do not produce *undefined* results. Languages such as Euclid and Verdi (and C for that matter) use an *undefined* semantics while Ada (in the absence of [11.6]) and Gypsy use an exception-based semantics.

For formal reasoning, the differences are largely matters of style. From an implementation standpoint, unless it can be shown that a given program will not have effects based on *undefined* results, the choice is between being able to detect a departure from normal execution and not. [11.6(5)] requires that exceptions not be suppressed if suppressing them would lead to a visible

[6] For most machines, we could specify exactly what this result is, i.e., how to compute it as a function of `X` and `Y`. It is not *undefined* in the sense that we know nothing about it.

[7] The first conjunct is redundant since it could be deduced from the other two.

[8] By a meaningful recovery, we mean undoing or overcoming the failed operation in such a way that computation can resume execution along the normal path, satisfying all the implicit and explicit assumptions of that path. In practice, this is extremely difficult unless the specification of the normal computation is extremely weak.

effect due to the subsequent use of an *undefined* result. We will examine the process of substituting operator definitions that produce "undefined" results for those that raise exceptions.

The stated assumptions associated with the example of [11.6(5.e)] are not sufficiently strong. In most machines, the result of an integer operation that fails an Ada implementation-defined check will be a valid value of the base type of the operation's result and, in many cases, it will be a valid value of the appropriate subtype as well. Thus, the values supplied to the comparison operation will always be "in-range integer value"s. The *value* is not the issue. If we allow the operation to omit its exception check, we must consider the result to be more than a value for the purposes of analysis. In the abstract, the result of an operation that yields an undefined result must be seen as a object having two attributes, **'value** and **'defined**. **'value** is of the base type of the result of the operation while **'defined** is boolean.

Under this view, multiply and compare might be defined as follows:

```
function Machine_Mul(X, Y : Machine_Integer)
          return Machine_Integer
  entry
  X'value, Y'value
    in (Machine_Integer'First .. Machine_Integer'Last) and
  X'defined and Y'defined;
  exit
    if Machine_MUL'defined then
      Machine_Mul'value(X, Y) = Integer_Mul(X, Y) and
      Machine_Mul'value(X, Y) in
        (Machine_Integer'First .. Machine_Integer'Last) and
    else
      Machine_Mul'value(X, Y) != Integer_Mul(X, Y) and
      Machine_Mul'value(X, Y) in
        (Machine_Integer'First .. Machine_Integer'Last) and
      Integer_Mul(X, Y) not in
        (Machine_Integer'First .. Machine_Integer'Last);
    end if

function Machine_CMP(X, Y : Machine_Integer)
          return Machine_CC
  entry
    X, Y in (Machine_Integer'First .. Machine_Integer'Last) and
    X'defined and Y'defined;
  exit
     Machine_CMP'defined(X, Y) and
    (Machine_CMP'value(X, Y) = GT implies Integer_GT(X, Y) and
    ... -- Specifications for other return values)
```

Under this view of operational semantics, the obligation to take appropriate action in the case of exceptional operations has shifted from the operation making the check to the operation using the result. The substitution of "undefined" semantics for "exception" semantics might be done as follows:

1. Tentatively replace an operation using "exception" semantics with the equivalent operation using "undefined" semantics. Note that this substitution is dependent on being able to prove the stronger **entry** specification of the latter.[9]

[9] In the implementation of a language using the "undefined" semantics, we note that there is, in general, no way to determine by inspection that a given bit string represents an "undefined" value. A two's complement machine

2. If it is possible to prove that the **'defined** attribute of the operation's result is always **true** then the substitution is permitted (see the ramification [11.6(5.a)]) and no further analysis is required.

3. Locate all uses of the result of the replaced operation. If there are none, the substitution is permitted. If there are any, substitute the corresponding "undefined" semantics operation, if necessary, and check the entry specification for references to the **'defined** attribute of the result in question. If any using operation assumes that the **'defined** attribute of the result is **true**, the substitution cannot be made.

If the substitution can be made, the net effect is to remove from further consideration the execution path arising from the exception exit of the replaced operator definition. This, in turn, should enable additional program transformations, including removal of the replaced operation since it is known to be without an externally visible effect. The removal of the exception path may permit additional removals since dependencies along the path no longer require consideration. Removal of the operation may permit additional operations to be removed since its inputs are now referenced at fewer places.

Note that this is an analytical approach, not an implementation. Typically, there is no practical way to tag values with an indication that they represent an "undefined" result. When this is the case there is no way for subsequent operations in an implementation to check explicitly for the undefined property. In addition, the amount of analysis required to detect all uses of a result may require extensive reasoning about the values of index expressions, etc., when the values are components of arrays or other composite structures.

Implicit in the assumption of the last paragraph of [11.6(5.e)] appears to be an additional assumption that the values being compared are also the mathematically correct results of the operations that produced them, i.e., that they are not undefined.

It is probably the case that the only permission actually granted by [11.6(5)] is the removal of code that is "dead" along its normal exit path regardless of any effects along its exceptional exit path.

5.2.1.3 Bounded errors and erroneous executions

The Ada 9X revision has made a serious attempt to reduce the number and types of circumstances under which a program's execution can become erroneous. Since an erroneous execution can exhibit arbitrary behavior, this change is highly desirable. Recognizing that most implementations do reasonable things in the face of program errors that violate language semantics, the notion of a bounded error has been introduced. The bounded errors associated with invalid representations are discussed in [13.9.1]

Bounded (Run-Time) Errors

> If the representation of a scalar object does not represent a value of the object's subtype (perhaps because the object was not initialized), the object is said to have an *invalid representation*. It is a bounded error to read the value of such an object. If the error is detected, either Constraint_Error or Program_Error is raised. Otherwise, execution continues using the invalid representation. The rules of the language outside this subclause assume that all objects have valid representations. The semantics of operations on invalid representations are as follows: 9
>
> - If the representation of the object represents a value of the object's type, the value of the type is used. 10

addition, for example, operating on two n bit long bit strings interpreted as integers produces an n bit long bit string that can be interpreted as an integer congruent to the mathematically correct integer result modulo 2^n. Because of this, the entry specifications cannot be executed but must be reasoned about.

- If the representation of the object does not represent a value of the object's type, the semantics of operations on such representations is implementation-defined, but does not by itself lead to erroneous or unpredictable execution, or to other objects becoming abnormal. 11

Erroneous Execution

A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, and the result object has an invalid representation. 12

> **Ramification:** In a typical implementation, every bit pattern that fits in an object of an integer subtype will represent a value of the type, if not of the subtype. However, for an enumeration or floating point type, there are typically bit patterns that do not represent any value of the type. In such cases, the implementation ought to define the semantics of operations on the invalid representations in the obvious manner (assuming the bounded error is not detected): a given representation should be equal to itself, a representation that is in between the internal codes of two enumeration literals should behave accordingly when passed to comparison operators and membership tests, etc. We considered *requiring* such sensible behavior, but it resulted in too much arcane verbiage, and since implementations have little incentive to behave irrationally, such verbiage is not important to have. 12.a
>
> If a stand-alone scalar object is initialized to a an **[sic]** in-range value, then the implementation can take advantage of the fact that any out-of-range value has to be abnormal. Such an out-of-range value can be produced only by things like unchecked conversion, input, and disruption of an assignment due to abort or to failure of a language-defined check. 12.b
>
> This depends on out-of-range values being checked before assignment (that is, checks are not optimized away unless they are proven redundant).

The language of the **Ramification** sounds reasonable, but it flies in the face of the conventions used in many of the logics used to reason about program behavior. Typically, *undefined* is a loaded term in these logics. *Undefined* is used to represent a distinguished value about which nothing can be proven. Thus $\not\vdash$ *undefined* = *undefined*. This is too strong for implementation semantics in most cases. In any implementation in which evaluating x is free of side effects that could change its value, $x = x$ is **true** even if x has an invalid representation or is *undefined* so long as the implementation of = simply involves comparing bit patterns. In the absence of a requirement to actually evaluate x, it should be unconditionally ok to substitute **true** for the equality.

On the other hand, this language seems to have the potential for conflicts with semantics of "undefined" results discussed above in connection with [11.6(5)]. The relationship between undefined as used in [13.9.1] and [11.6(5)] should be further clarified.

5.2.2 [11.6(7)]

This paragraph allows fairly arbitrary reordering of actions within the scope of an exception handler by reducing the expectations that the programmer may have concerning the state of the computation at the time that the handler is entered. This is essentially the "Undefined" execution order of [9, section 2.6.5]. We note that the only effective actions that a programmer can take when an exception handler is entered in the face of this kind of reordering is to assign normal values to all variables that might have become abnormal due to operations disrupted by the exception.

The first sentence of this paragraph is complex and convoluted and calls out to be simplified or clarified. The following discussion may aid in finding more suitable language. An exception_handler is optionally associated with a handled_sequence_of_statements which contains a sequence_of_statements

and is, among other things, the operational portion of a task_body. When an exception is raised, it will either be handled or cause the containing task_body to terminate. In either case, all that the user can expect to know is that the exception was raised somewhere in the code of the sequence_of_statements component of the handled_sequence_of_statements that contains the exception_handler just entered or that constitutes the operational part of the task being terminated. The reordering that can be done is limited in two respects.

1. The operation that raises the exception due to a failed language–defined check cannot have been moved into the code of an independent subprogram, and

2. The operation that raises the exception due to a failed language–defined check cannot have been moved into the code of some abort-deferred operation.

Just breaking up the sentence may help. Instead of

- If an exception is raised due to the failure of a language-defined check, then upon reaching the corresponding exception_handler (or the termination of the task, if none), the external interactions that have occurred need reflect only that the exception was raised somewhere within the execution of the sequence_of_statements with the handler (or the task_body), possibly earlier (or later if the interactions are independent of the result of the checked operation) than that defined by the canonical semantics, but not within the execution of some abort-deferred operation or *independent* subprogram that does not dynamically enclose the execution of the construct whose check failed. 6

perhaps language similar to the following would be more understandable.

- If an exception is raised due to the failure of a language-defined check, then upon reaching the corresponding exception_handler (or the termination of the containing task, if no handler is present), the external interactions that have occurred need reflect only that the exception was raised somewhere within the execution of the sequence_of_statements associated with the handler (or the task_body). It may appear that the exception was raised earlier than defined by the canonical semantics (or later if the interactions are independent of the result of the checked operation). It may not appear as if the exception were raised within the execution of some abort-deferred operation or within the execution of an *independent* subprogram that does not dynamically enclose the execution of the construct whose check failed. 6

5.3 Living with the "Canonical Semantics"

The canonical semantics define a potentially very large family of valid executions. This is due to the numerous places in which the language definition allows operations to be performed in an arbitrary order. An implementation is free to select any order under these circumstances. In the absence of order dependencies and tasking considerations, all canonical executions should produce the same externally visible effect. Order-dependent side effects, including exceptions raised due to the failure of language-defined checks, can affect the effect of the program. The problem is twofold:

1. Reducing the potential effect space of the program, and

2. Determining which execution the implementation has selected.

First of all, it is worth noting that this kind of problem is not unique to Ada (both Ada 83 and Ada 9X). Most programming languages, including C and C++, admit similar behaviors, either implicitly or explicitly. Ada is more explicit about them. In general, the failure of languages to define or enforce restrictive canonical executions is attributed to a need for flexibility in order to achieve run-time efficiency. There is tension between this need and the requirements for predictable

program behavior, which are imposed by small segments of the user community, typically those users associated with safety and security critical applications. Predictable behavior is usually defined as having a rigorous semantic definition that allows the formal verification of programs written in the language, preferably using mechanical aids.

In theory, one could reason about Ada programs by enumerating the set of possible executions and reasoning individually about each one. A program could then be said to shown to exhibit a given property if each of its possible executions could be shown to exhibit the property. In practice, the combinatorics of potential execution choices are likely to render this approach infeasible for any non-trivial program. If we assume that a compiler conforming to the language standard produces code that follows one of the set of canonical executions of a given program, it seems a waste of time to prove properties of the set as a whole unless there is a need to guarantee the behavior of the program under all possible conforming implementations. This is seldom the case. Further complications arise when it is possible to show that some, but not all, members of the canonical execution set exhibit the desired property. In this case, it is essential to determine whether the implementation being considered exhibits the property.

There are several possibilities. The first is to attempt to reduce the size of the set of canonical executions to a tractable size and possibly to a single member. The second is to discover the member of the canonical execution set that has been chosen by a particular implementation and to reason about that execution alone.

5.3.1 Restricting the execution set size

The size of the canonical execution set about which one must reason can be reduced by one of two methods; reducing the choices available to the implementation or finding equivalence classes within the set, or by some combination of the two. Ada 9X provides some means for imposing order. For example, the order of the association of operands with a sequence of operators of the same precedence can be controlled by the explicit use of parentheses. The introduction of explicit intermediate variables and assignments should have a similar effect. For example, suppose that side effects exist such that the value resulting from the evaluation of `<exp1>` depends on whether it is evaluated before or after `<exp2>`, but that there are no other order dependencies between the expressions. Further assume that the evaluations produce results of some integer subtype.

```
A := <exp1> + <exp2>;
```

Either of the possible results is a member of the canonical execution set for this fragment. If we want to ensure that `<exp1>` is evaluated first, we might write:

```
A1 := <exp1>;
A2 := <exp2>;
A := A1 + A2;
```

It is not clear that the additional freedoms to reorder operations granted by [11.6(6)] allow an implementation to ignore structuring of this kind, but aggressive optimizations in compilers for other languages are known to do so in some cases. Presumably, the dependency between the two expressions either becomes explicit or the implementation will be forced to recognize that it cannot assume independence because the expressions invoke separately compiled routines and it will be forced to produce the intended result[10].

Note that the explicitly ordered code may still exhibit a family of canonical executions. In the expression `A1 + A2`, the language allows `A1` or `A2` to be "evaluated" first. We claim that given

[10] If the expressions are sufficiently complex and the dependencies between them limited, it may be possible to interleave their evaluations. This would be permitted under the general freedoms noted in [11.6(3)]

appropriate type declarations (and barring some pathological implementation of +) that both orders will be equivalent and trivial.

By a combination of these two techniques, forcing orders where order makes a difference and creating situations where it is easy to show that, at least locally, all canonical executions are equivalent, it should be possible to reduce the number of canonical executions associated with a program to a tractable number. In most cases, the analysis of the remaining canonical executions should show that language-defined checks will not fail, rendering moot the freedoms of [11.6]. The utility of this approach depends on the implementation or implementations of interest ensuring that the canonical semantics are honored.

If the notion of a subset of Ada 9X for High Integrity systems, as recently proposed by Brian Wichmann, is accepted, the subset definition could restrict the ordering freedoms permitted by the primary language definition. This approach would necessitate subset compilers to enforce the restrictions, but would offer a higher degree of assurance than the use of general purpose compilers. If a subset is adopted with the notion of supporting mechanical verification, it is not unreasonable to expect that integrated environments will be developed in which both the verification and implementation tools are based on the same semantic assumptions.

5.3.2 Discovering the execution

Another approach to the problem of a canonical execution set is the determination of the actual execution produced for a given program by a given implementation. This requires that the compiler output its object code in a form that allows the user to determine the actual execution that will occur when the program is executed. Implementations conforming to the Safety and Security Annex, in particular to section [H.3], will provide this kind of information. With an appropriate transformation of the object code back into an appropriate Ada or Ada-like source form, it should be possible to perform source level analysis or verification on the program while maintaining confidence that the results are, in fact, applicable to the compiled program.

It is clear that this approach requires facilities that are not present in many, if not all, existing compilers, but the Annex should encourage development of this facility.

5.4 Observations on the Reference Manual

In the course of using the Reference Manual in the preparation of this chapter, a number of general shortcomings have been observed. These have more to do with presentation than with substance and can be fixed prior to the release of the final document.

First of all, we wish to compliment the Mapping/Revision Team on the content and style of the manual. Not only is the wording a substantial improvement over the Ada 83 Reference Manual, but the inclusions of the annotations provide useful and substantive insight into the workings of the language. It is to be hoped that the annotated version will be maintained along with its "official" subcomponents and that it will see widespread use by serious students of Ada 9X.

This said, there are ways in which the the Reference Manual could be further improved.

1. The index is not sufficiently comprehensive. On a number of occasions, an attempt to trace the consequences of a definition found that the defining occurrence was the only reference in the index. Fortunately, the source files are available and can be searched as necessary; however, any term important enough to be marked as a definition is important enough to have the consequences of that definition tracked. A presentation similar to that used in the index for syntactic constructs should be adapted for defined terms, i.e., a defining reference followed by using references.

2. The index does not appear to cover the annotations. Extending it to this level would greatly aid in the use of the annotated manual.

3. The Syntax Cross-Reference would be much more useful if references for the defining occurrence as well as the using occurrences were given. For example, we find from the cross-reference that a **task_body** is used in the definition of a **proper_body** in [3.11], but we must go to the main index to discover that a **task_body** is defined in [9.1(6)]. Extending the indexing to the numbered paragraph level as is done in the index would also be useful.

4. The marginal paragraph numbering is incomplete and inconsistent. For example, the paragraphs following the example codes of [11.6(5.e)] and [11.6(5.g)] are not numbered while similar paragraphs elsewhere, e.g., [8.3(29.o)] are. There is a similar problem in [13.9.1(12.b)] as well. This is probably the result of the mechanical approach taken to inserting the annotating scribe commands. In preparing the LaTeX source for the Annotated version of the Ada 83 reference manual, I found it necessary to insert this material manually.

5. In some cases, precision seems to have been sacrificed for readability. This occurs when it is difficult to determine the antecedents for pronouns or where the same noun appears in an ambiguous context. An example is [4.5(11)] discussed in Section 5.2.1.1 on page 87 above. More liberal use of the @Redundant (or a similar) construct might alleviate this problem in the annotated version.

Chapter 6

Conclusions

We did not expect to formulate a complete semantic definition (even for the sequential part of Ada) in this project; there were simply not enough resources to do so. What we did expect was to gain some insights into the structure of the language, and to identify some problems either with the description in the Reference Manual or in the design of the language itself, and to contribute to the development of Ada 9X by suggesting improvements to the description or design. Those expectations were met to some degree; for example, we identified some flaws (that have now been fixed) in the design in the area of per-object constraints; we identified some conceptual and some wording problems in the area of floating point and developed a model that was used in the development of new wording for the Reference Manual; and we identified some incompleteness in the description of actual subtypes.

However, we did not make as much progress in the natural semantics definition as we had originally hoped. It was more difficult to understand the supposedly trivial parts of the language than we had imagined. Large-scale languages like Ada do not have neat, independent parts; rather, each feature is affected in some measure by the others. For example, the type system is affected by the concurrency mechanism (e.g., task types), by the packaging mechanism (e.g., private types), and in several ways by the object-oriented features (e.g., access discriminants, per-object expressions, class-wide types). So, indeed, there are no really trivial aspects of the language. In the original LPT project, we had felt that it would be a waste of effort to develop a formal model for things that "everyone understands". Our recent efforts, however, have shown us that there are interesting problems lurking at the fringes of even these areas.

Even though there are serious gaps in the definition, a considerable amount of groundwork has been done. We have identified most of the basic semantic domains that must be used in a full definition, we developed structuring mechanisms for the definition that allow us to describe many of the implementation freedoms, and we have several tools (such as the type checker for the Prolog representation of the definition, and the tool that derives LaTeX source from the Prolog representation) that help in the production and documentation of the definition. So, we feel that we have made a good start in the direction of a complete description of the sequential part of the language.

We are not sure how easily our framework could be adapted to deal with concurrency. The influence of tasking in DDC's formal definition of Ada 83 [1] is pervasive, and we suspect that incorporating concurrency into our definition would similarly affect every part of the model.

6.1 Implementation Freedoms

One impediment to writing a formal definition like ours is the high degree of underspecification in the Reference Manual. This allows implementations considerable freedom to choose orders of actions, accuracy of results, base ranges of types, and so on. These freedoms can be difficult to

model; where an implementation need only produce one acceptable result, our semantics tries to describe *all* acceptable results.

Modeling these freedoms sometimes forces our formal model to differ in significant ways from an implementation (and to use representations that no implementation is likely to use). For a simple example, consider the following rule about access-to-subprogram values:

> Two access-to-subprogram values are equal if they are the result of the same evaluation of an Access **attribute_reference**, or if both are equal to the null value of the access type. Two access-to-subprogram values are unequal if they designate different subprograms. It is unspecified whether two access values that designate the same subprogram but are the result of distinct evaluations of Access **attribute_references** are equal or unequal.

In order to model this, we are forced to use a representation of access-to-subprogram values that consists of both a reference to the designated subprogram and an "instance" value that tells which evaluation of an Access attribute gave the access value. Each evaluation of an Access attribute increments this instance value, so that we can determine whether two access values derive from the same evaluation or not. Our definition of the equality function checks both the subprogram reference and the instance number, and can give a result of equal, unequal, or unknown. Such a representation is unlikely to be used in any implementation of Ada 9X.

An implementor of the language need not be concerned with all these freedoms; just one particular implementation choice needs to be made and the existence of other possible choices is irrelevant. A programmer does not necessarily need to be concerned about all the alternative orders; it is usually possible to write programs in such a way that the specific choice made by an implementation does not matter (for example, by avoiding side effects in functions, restricting the statements in a **package_body** to affect only variables local to the package, and so on). On the other hand, anyone trying to read an Ada program may indeed be concerned about the different possible outcomes of an execution (especially if the writer has not been careful to avoid situations where the different orders matter). So, our model, while unnatural if compared to an implementation, is quite natural as a description of the complexities that careful readers must deal with.

6.2 Notation and Tools

The natural semantics framework seems to have worked fairly well, although there were a few awkward aspects to our formalization of the language semantics. In particular, our need to introduce explicit sequencing and arbitrary-order combinations of actions (in two slightly different variations) seems somewhat artificial. However, this mechanism of actions allows us to present reasonably concise descriptions of many language features.

The use of Prolog to make the definition executable (and type-checkable) was a great help. We have been able to execute parts of the definition to confirm that it expresses what we intended. The type checker was able to find a number of trivial errors in our semantics. There is some price to be paid, hovever; it is sometimes inconvenient to express a rule in a manner acceptable to Prolog. This is particularly evident in the descriptions of the various semantic domains and the primitive functions acting over those domains. Prolog does not support defined functions (instead, relations must be used). We used a program to convert the Prolog code into the LaTeX source used for this report. This program is able to introduce functional notation in places where we have instructed it to, so at least our published form of the rules can use a more expressive notation than Prolog. But this is still rather unsatisfactory. It is possible that other tools might be able to provide better mechanical support.

6.3 Bounded Errors

A number of rules and concepts were added to Ada 83 in order to make programs more predictable. For example, many situations leading to erroneous executions in Ada 83 have been made into *bounded errors*. For these errors, a range of possible outcomes is described. This seems like a beneficial change. However, there is a price to be paid for this benefit: the model for a feature using bounded errors can be substantially more complex than a model using erroneous executions. For example, in order to change the evaluation of an uninitialized scalar variable from erroneous to a bounded error, it was necessary to introduce the notion of "invalid representations" of scalar objects. The addition of this notion has an influence on a number of other areas of the language (e.g., relational operators, membership tests, and type conversions). So, the formal model is more complex, which means that *formal* predictions about programs are harder to derive. On the other hand, the execution of programs is more predictable in the sense that these executions are more constrained (the old rules allowed any behavior, whereas the new rules are more specific).

6.4 Structure of Models

It does not seem possible, using our methods, to formulate a model for Ada 9X that is simultaneously concise, comprehensible, broad, and accurate. Accounting for all the special cases of features adds so much detail to the model that it becomes unusable.

Textbook writers face a similar dilemma; if too much detail is presented, readers will find the text impenetrable. Therefore, authors present simplified descriptions of parts of the language. These simplified descriptions, even when they lead the reader to draw incorrect conclusions about the behavior of some programs, are nevertheless useful to readers who are first learning the language. In a later part of a book, an author may elaborate on some of these missing details, and may need to contradict some of his earlier oversimplified assertions.

We do not know exactly how to make layered formal models using a similar structure. In most formal notations, it is not possible to override an earlier assertion with a more detailed assertion. Even if this were allowed, it is unclear how a user of such a layered formal model would know when the simpler part of model was applicable.

In the model developed in this report, we have tried to approach this ideal of structured models in a very modest way through our use of the "unpredicted" outcomes to simplify the formal model; we can certainly imagine a more complex version of this model that would, in fact, make predictions where this simpler model refuses to. However, we have not had the resources to develop the more complex model.

Bibliography

[1] *The Draft Formal Definition of Ada.* Denmark, 1987.

[2] David Guaspari. Formal methods in the design of Ada 9X. In *Proceedings of the Ninth Annual Conference on Computer Assurance*, 1994.

[3] Intermetrics, Inc. *Annotated Ada 9X Reference Manual, Version 4.0*, September 1993.

[4] Intermetrics, Inc. *Annotated Ada 9X Reference Manual, Version 5.0*, June 1994.

[5] G. Kahn. Natural semantics. In *Fourth Annual Symposium of Theoretical Aspects of Computer Science*, pages 22–39, 1987. LNCS 247.

[6] John McHugh. *Towards Efficient Code from Verified Programs.* PhD thesis, The University of Texas at Austin, 1983.

[7] U.S. Reddy and T.K. Lakshman. Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System. Submitted to the '91 Intl. Logic Programming Symp., San Deigo, California, February 1991.

[8] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992.

[9] Language Precision Team. Formal studies of Ada 9X, final report. Technical report, Odyssey Research Associates, December 1992.

[10] United States Department of Defense, New York. *Reference Manual for the Ada Programming Language*, February 1983. ANSI/MIL-STD-1815A-1983.

Appendix A

Official comments submitted

This appendix lists the official comments submitted by the LPT. For each comment, we give its official "key" number, its title, and a short description of the comment or its effect on the Standard.

Comment 93-3209.a: *Too much extra permission to remove checks.* The conceptual framework of section 11.6 has been changed.

Comment 93-3398.a: *per-object constraints and the "current instance".* A new rule [3.8(13)] has been added to avoid the problem described.

Comment 93-3511.a: *What is the base range of an enumeration type?.*

Comment 93.3547.a: *constants that aren't.* The problem described has been identified as an erroneous execution in [13.9.1(13)]

Comment 93-3547.b: *subcomponents that are constrained by their initial value.*. A new legality rule [3.6(11)] was added.

Comment 93-3547.c: *conversion to a type with aliased components.* A new legality rule [3.6(11)] was added.

Comment 93-3574.a: *interleaving evaluation and conversion.* Several paragraphs have been modified to clarify the rules.

Comment 93-03574.b: *reassociation of sequences of predefined operators.* A note [4.5(13.b)] has been added to the Annotated Reference Manual. It is unclear, however, that the note clarifies the issue raised in this comment.

Comment 93-3575.a: *aliased subcomponents with per-object constraints.* A new legality rule [3.6(11)] was added.

Comment 94-3621.a: *Initialize a discriminant before any subcomponents that depend on it.* This clause has been added to the rules of [3.3.1(20)].

Comment 94-3760.a: *Phraseology.* An inaccurate statement has been reworded in [M(1)].

Comment 94-3761.a: *Phraseology.* Some wording in [3.3] has been improved.

Comment 94-3762.a: *Phraseology.*

Comment 94-3763.a: *Phraseology.* The wording of [3.2.3] has been clarified.

Comment 94-3764.a: *Phraseology.* A clarifying cross-reference was added.

Comment 94-3765.a: *Phraseology.*

Comment 94-3901.a: *Are first subtypes of enumeration types constrained?.* A clause has been added to [3.5.1(10)] to answer this question.

Comment 94-3901.b: *First subtypes of discriminated types are unconstrained.* This clause has been added to [3.7(26)].

Comment 94.3901.c: *First subtypes of incomplete types.* A new note in the [3.10.1(10.a)] Annotated Reference Manual argues that this issue is unimportant.

Comment 94.3901.d: *Predefined operators and invalid scalar components.* The entire discussion of "invalid" scalars has been modified in version 5.0 of the Reference Manual. This comment and

the following two are cited in the changes.

Comment 94-4045.a: *Model for valid and invalid values.* See above.

Comment 94-4054.a: *Model for valid and invalid values.* See above.

Comment 94-4064.a: *What is the actual subtype of a formal object?.* The question has been answered in [6.4.1(15)].

Comment 94-4065.a: *Normalizing composite objects.* An explicit statement about restoring objects to normal state appears in [13.9.1(7)].

Comment 94-4149.a: *Incomplete definition of 'expected profile' and 'corresponding parameter'.* Some rules have been clarified

Comment 94-4171.a: *Actual subtypes and aliased views.* The rules in [3.10(9)] have been reworded.

The remaining comments were sent too late to affect version 5.0 of the Reference Manual. Some of them are addressed in the electronically-distributed version 5.3, as noted below. Furthermore, the floating-point annex is under revision to address some of the comments on the floating-point model.

Comment 94-4448.a: *Signed zeroes not permitted as floating point values.*

Comment 94-4454.a: *Model-oriented floating point attributes.*

Comment 94-4455.a: *Relation between requested precision and model numbers.*

Comment 94-4481.a: *Inappropriate references to Annex G.*

Comment 94-4482.a: *Are S'Model, S'Machine deterministic?.*

Comment 94-4486.a: *Symmetry of floating point types.*

Comment 94-4489.a: *Derivation from a floating point type.*

Comment 94-4535.a: *Derived types with new discriminants are extensions.* A sentence has been added.

Comment 94-4535.b: *Discriminants used in constraints in derived type definitions.*

Comment 94-4535.c: *Incorrect rules for uses of new discriminants in constraint on parent.* A new rule has been added.

Comment 94-4535.d: *Current instance of a derived type.* A note has been added in a "to be honest" section of the AARM.

Comment 94-4572.a: *An object that is not {a part of} a formal parameter.*

Comment 94-4572.b: *For aliasing, the type of the formal, not the part, matters.*

Comment 94-4572.c: *When does an access path exist?.*

Comment 94-4587.a: *Incompatibility between semantics of the core and annex G?.*

Comment 94-4796.a: *reading a composite with an uninitialized scalar component.* The wording of this rule has been changed.

Appendix B

Intermediate Syntax

This appendix describes the decorated abstract syntax representation of Ada 9X programs. This representation includes all static semantic information needed for defining the dynamic semantics.

One approach, not taken here, is to define a representation of semantic information such as types, subtypes, and overload information and defining necessary tree attributes. Instead of using semantic attributes we chose to give a purely syntactic representation of necessary information by introducing new kinds of tree nodes and new synthetic names.

For instance, the results of overload resolution are captured by the introduction of new unique names and suitable renaming of overloaded entities and their use. Crucial type information is represented using existing syntax for qualified expressions. Thus the result of static semantic analysis is a normalized abstract syntax tree. The details of this normalization are described below.

Declarations that introduce multiple names are replaced by static analysis with multiple declarations introducing single names where this is legal.

Constructs for which no abstract syntax is provided are either not treated in this definition (e.g., tasking) or have not significance for the dynamic semantics (e.g., generics).

B.1 Syntactic Domains

The following is a complete listing of the term algebra used to represent abstract syntax trees. The constructors are grouped by sorts and are arranged alphabetically.

B.1.1 Component Associations (*Aca*)

array_comp$assoc :	$Dch^*, Exp \rightarrow Aca$

B.1.2 Aggregates (*Agg*)

ext$agg :	$Exp, Rca^* \rightarrow Agg$
named_array$agg :	$Aca^* \rightarrow Agg$
null_ext$agg :	$Exp \rightarrow Agg$
null_record$agg :	$\rightarrow Agg$
other_array$agg :	$Exp^*, Exp \rightarrow Agg$
pos_array$agg :	$Exp^* \rightarrow Agg$
record$agg :	$Rca^* \rightarrow Agg$

B.1.3 Case Alternatives (*Alt*)

Alt$lst :	list(*Alt*) $\rightarrow$ *Alt*
case$*Alt* :	*Dch**, *Stm* $\rightarrow$ *Alt*

B.1.4 Choice Lists (*Ccl*)

list$choice :	id* $\rightarrow$ *Ccl*
others$choice :	$\rightarrow$ *Ccl*

B.1.5 Context Items (*Cit*)

with$context :	*Nam** $\rightarrow$ *Cit*

B.1.6 Component Declarations (*Cmp*)

aliased_comp$decl :	*Id*, *Sid* $\rightarrow$ *Cmp*
comp$decl :	*Id*, *Sid* $\rightarrow$ *Cmp*
init_aliased_comp$decl :	*Id*, *Sid*, *Exp* $\rightarrow$ *Cmp*
init_comp$decl :	*Id*, *Sid*, *Exp* $\rightarrow$ *Cmp*

B.1.7 Compilation Units (*Cmp*)

lib$unit :	*Cit**, *Dcl* $\rightarrow$ cmu
private$unit :	*Cit**, *Dcl* $\rightarrow$ cmu
sub$unit :	*Cit**, *Nam*, *Dcl* $\rightarrow$ cmu

B.1.8 Conditions (*Cnd*)

Exp$condition :	*Exp* $\rightarrow$ *Cnd*

B.1.9 Constraints (*Cns*)

constr_delta$constr :	*Exp*, *Cns* $\rightarrow$ *Cns*
constr_digits$constr :	*Exp*, *Cns* $\rightarrow$ *Cns*
delta$constr :	*Exp* $\rightarrow$ *Cns*
digits$constr :	*Exp* $\rightarrow$ *Cns*
discr$constr :	*Dca** $\rightarrow$ *Cns*
index$constr :	*Rng** $\rightarrow$ *Cns*
range$constr :	*Rng* $\rightarrow$ *Cns*

B.1.10 Discriminant Associations (*Dca*)

named$assoc :	*Id*, *Exp* $\rightarrow$ *Dca*

B.1.11 Discrete Choices (*Dch*)

discr_other$choice :	$\rightarrow$ *Dch*
exp$choice :	*Exp* $\rightarrow$ *Dch*
range$choice :	*Rng* $\rightarrow$ *Dch*

B.1.12 Declarations (*Dcl*)

dcl$lst :	list(*Dcl*) → *Dcl*
a_c_i_obj$decl :	*Id*, *Sid*, *Exp* → *Dcl*
a_c_obj$decl :	*Id*, *Sid* → *Dcl*
a_i_obj$decl :	*Id*, *Sid*, *Exp* → *Dcl*
a_obj$decl :	*Id*, *Sid* → *Dcl*
c_i_obj$decl :	*Id*, *Sid*, *Exp* → *Dcl*
c_obj$decl :	*Id*, *Sid* → *Dcl*

real_const$decl :	*Id*, *Exp* → *Dcl*
int_const$decl :	*Id*, *Exp* → *Dcl*
d_ext$decl :	*Id*, *Dcp*, *Sid* → *Dcl*
d_i_type$decl :	*Id*, *Dcp* → *Dcl*

d_type$decl :	*Id*, *Dcp*, *Tdf* → *Dcl*
exception$renaming :	*Id*, *Nam* → *Dcl*
excpt$decl :	*Id* → *Dcl*
ext$decl :	*Id*, *Sid* → *Dcl*

i_obj$decl :	*Id*, *Sid*, *Exp* → *Dcl*
i_type$decl :	*Id* → *Dcl*
obj$decl :	*Id*, *Sid* → *Dcl*
object$renaming :	*Id*, *Nam*, *Nam* → *Dcl*

s_subp$spec :	*Sps* → *Dcl*
subp$body :	*Sps*, *Dcl*, *Stm* → *Dcl*
subp$renaming :	*Sps*, *Nam* → *Dcl*
subp$spec :	*Sps* → *Dcl*
subtype$decl :	*Id*, *Sid* → *Dcl*
type$decl :	*Id*, *Tdf* → *Dcl*
c_type$decl :	*Id*, *Tdf* → *Dcl*
c_d_type$decl :	*Id*, *Tdf*, *Dcp* → *Dcl*

B.1.13 Discriminant Parts (*Dcp*)

box$discr :	→ *Dcp*
list$discr :	Dcs^* → *Dcp*

B.1.14 Discriminant Specifications (*Dcs*)

acc$discr :	*Id*, *Nam* → *Dcs*
acc_init$discr :	*Id*, *Nam*, *Exp* → *Dcs*
init$discr :	*Id*, *Nam*, *Exp* → *Dcs*
simple$discr :	*Id*, *Nam* → *Dcs*

B.1.15 Exception Choices (*Ech*)

named$excpt :	*Nam* → *Ech*
others$excpt :	→ *Ech*

B.1.16 Else-If Clauses (*Eif*)

eif$lst :	*Eif** → *Eif*
elsif$clause :	*Cnd*, *Stm* → *Eif*

B.1.17 Expressions (*Exp*)

and_then$*Exp* :	*Exp*, *Exp* → *Exp*
exp$alloc :	*Exp* → *Exp*
in_name$exp :	*Exp*, *Nam* → *Exp*
in_range$exp :	*Exp*, *Rng* → *Exp*
in_type$exp :	*Exp*, *Nam* → *Exp*
name$exp :	*Nam* → *Exp*
not_in_name$exp :	*Exp*, *Nam* → *Exp*
not_in_range$exp :	*Exp*, *Rng* → *Exp*
not_in_type$exp :	*Exp*, *Nam* → *Exp*

null$exp :	→ *Exp*
integer$exp :	integer → *Exp*
real$exp :	real → *Exp*
char$exp :	integer → *Exp*
or_else$exp :	*Exp*, *Exp* → *Exp*
paren$exp :	*Exp* → *Exp*
qual$aggregate :	*Nam*, *Agg* → *Exp*
qual$exp :	*Nam*, *Exp* → *Exp*
type$alloc :	*Sid* → *Exp*
type$conversion :	*Nam*, *Exp* → *Exp*

B.1.18 Modes (*Mde*)

in$mode :	→ *Mde*
in_out$mode :	→ *Mde*
no$mode :	→ *Mde*
out$mode :	→ *Mde*

B.1.19 Names (*Nam*)

access$attr :	*Nam* → *Nam*
delta$attr :	*Nam* → *Nam*
digits$attr :	*Nam* → *Nam*
deref$name :	*Nam* → *Nam*
direct$name :	*Id* → *Nam*
func$call :	*Nam*, *Pss** → *Nam*
Id$attr :	*Nam*, *Id* → *Nam*
indexed$comp :	*Nam*, *Exp** → *Nam*

name_type$conversion :	*Nam*, *Nam* → *Nam*
param$attr :	*Nam*, *Id*, *Exp* → *Nam*
selected$comp :	*Nam*, *Id* → *Nam*
slice$op :	*Nam*, *Rng* → *Nam*

B.1.20 Parameter Specifications (*Pms*)

access$param :	*Id*, *Nam* → *Pms*
access_default$param :	*Id*, *Nam*, *Exp* → *Pms*
default$param :	*Id*, *Mde*, *Nam*, *Exp* → *Pms*
normal$param :	*Id*, *Mde*, *Nam* → *Pms*

B.1.21 Pragmas (*Prg*)

param$pragma :	*Id*, Pss^* → *Prg*
simple$pragma :	*Id* → *Prg*

B.1.22 Parameter Associations (*Pss*)

named_exp$arg :	*Id*, *Exp* → *Pss*
named_name$arg :	*Id*, *Nam* → *Pss*

B.1.23 Record Component Associations (*Rca*)

choice$assoc :	*Ccl*, *Exp* → *Rca*

B.1.24 Ranges (*Rng*)

attr$range :	*Nam* → *Rng*
explicit$range :	*Exp*, *Exp* → *Rng*
parm_attr$range :	*Nam*, *Exp* → *Rng*

B.1.25 Subtype Indications (*Sid*)

constrained$subtype :	*Nam*, *Cns* → *Sid*
named$subtype :	*Nam* → *Sid*
subtype$range :	*Rng* → *Sid*

B.1.26 Subprogram Specifications (*Sps*)

function$spec :	*Id*, Pms^*, *Nam*→ *Sps*
procedure$spec :	*Id*, Pms^*→ *Sps*

B.1.27 Statements (*Stm*)

stm$lst :	list(*Stm*) → *Stm*
agg_code$stm :	*Nam*, *Agg* → *Stm*
assign$stm :	*Nam*, *Exp* → *Stm*
call$stm :	*Nam*, Pss^* → *Stm*
case$stm :	*Exp*, *Alt* → *Stm*
cond$exit :	*Cnd* → *Stm*
declare$block :	*Dcl*, *Stm* → *Stm*
exp_code$stm :	*Nam*, *Exp* → *Stm*
for$loop :	*Id*, *Rng*, *Stm* → *Stm*
func_return$stm :	*Exp* → *Stm*

goto$stm : Nam → Stm
if$stm : Cnd, Stm, Eif → Stm
if_else$stm : Cnd, Stm, Eif, Stm → Stm
labled$stm : Id, Stm → Stm
name$exit : Nam → Stm
name_cond$exit : Nam, Cnd → Stm
named$block : Id, Stm → Stm
named$loop : Id, Stm → Stm
named_declare$block : Id, Dcl, Stm → Stm
named_for$loop : Id, Id, Rng, Stm → Stm

named_reverse$loop : Id, Id, Rng, Stm → Stm
named_while$loop : Id, Cnd, Stm → Stm
null$stm : → Stm
plain$exit : → Stm
plain$loop : Stm → Stm
raise$stm : Nam → Stm
reraise$stm : → Stm
return$stm : → Stm
reverse$loop : Id, Rng, Stm → Stm
simple$block : Stm → Stm
while$loop : Cnd, Stm → Stm
handled$statement : Stm, Xhd* → Stm
unhandled$statement : Stm → Stm

B.1.28 Type Definitions (*Tdf*)

access$type : Sid → Tdf
aliased_array$type : Rng*, Sid → Tdf
aliased_uc_array$type : Nam*, Sid → Tdf
all_access$type : Sid → Tdf
array$type : Rng*, Sid → Tdf
const_access$type : Sid → Tdf

const_dec_fixed$type : Exp, Exp, Cns → Tdf
const_float$type : Exp, Cns → Tdf
dec_fixed$type : Exp, Exp → Tdf
der$type : Sid → Tdf
enum$type : Id* → Tdf
ext$type : Sid, rcd → Tdf
float$type : Exp → Tdf
func$type : Pms*, Nam→ Tdf
int$type : Exp, Exp → Tdf

mod$type : Exp → Tdf
named$type : Sid → Tdf
ord_fixed$type : Exp, Rng → Tdf
proc$type : Pms*→ Tdf
record$type : Cmp*, Vrp→ Tdf
t_record$type : Cmp*, Vrp→ Tdf
uc_array$type : Nam*, Sid → Tdf

B.1.29 Variants (*Vnt*)

variant$clause : $Dch^*, Cmp^*, Vrp \rightarrow Vnt$

B.1.30 Variant Parts (*Vrp*)

no$variant : Vrp
variant$part : $Nam, Vnt^* \rightarrow Vrp$

B.1.31 Exception Choices (*Xhd*)

choice$handler : $Id, Ech^*, Stm \rightarrow Xhd$
expt$handler : $Ech^*, Stm \rightarrow Xhd$

B.2 Lexical Elements

prgama ::=
    pragma identifier [(pragma_argument_association { , pragma_argument_association})] ;

param$pragma :	$Id, Pss^* \rightarrow Prg$
simple$pragma :	pragma, $Id \rightarrow Prg$

pragma_argument_association ::=
    [identifier =>] name
  | [identifier =>] expression

named_exp$arg :	$Id, Exp \rightarrow Pss$
named_name$arg :	$Id, Nam \rightarrow Pss$

B.3 Declarations and Types

B.3.1 Declarations

basic_declaration ::=
    type_declaration
  | subtype_declaration
  | object_declaration
  | number_declaration
  | subprogram_declaration
  | abstract_subprogram_declaration
  | package_declaration
  | renaming_declaration
  | exception_declaration
  | generic_declaration
  | generic_instantiation

It is convenient to treat sequences of declarations as a single declaration.

Dcl$lst :	$Dcl^* \rightarrow Dcl$

defining_identifier ::=
    identifier

B.3.2 Types and Subtypes

B.3.2.1 Type Declarations

type_declaration ::=
    full_type_declaration
  | incomplete_type_declaration
  | private_type_declaration
  | private_extension_declaration

full_type_declaration ::=
    type defining_identifier [known_discriminant_part] **is** type_definition ;
  | task_type_declaration
  | protected_type_declaration

d_type$decl :	$Id,\ Dcp,\ Tdf \rightarrow Dcl$
type$decl :	$Id,\ Tdf \rightarrow Dcl$

type_definition ::=
    enumeration_type_definition
  | integer_type_definition
  | real_type_definition
  | array_type_definition
  | record_type_definition
  | access_type_definition
  | derived_type_definition

B.3.2.2 Subtype Declarations

subtype_declaration ::=
    subtype defining_identifier **is** subtype_indication ;

subtype$decl :	$Id,\ Sid \rightarrow Dcl$

subtype_indication ::=
    subtype_mark [constraint]

subtype_mark ::=
    name

constrained$subtype :	*Nam, Cns* → *Sid*
named$subtype :	*Nam* → *Sid*
subtype$range :	*Rng* → *Sid*

The form subtype$range applies only to discrete subtype definitions.

constraint ::=
    scalar_constraint
  | composite_constraint

scalar_constraint ::=
    range_constraint
  | digits_constraint
  | delta_constraint

composite_constraint ::=
    index_constraint
  | discriminant_constraint

B.3.2.3 Classification of Operations

B.3.3 Objects and Named Numbers

B.3.3.1 Object Declarations

object_declaration ::=
    defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression] ;
  | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression] ;
  | single_task_declaration
  | single_protected_declaration

a_c_i_obj$decl :	*Id, Sid, Exp* → *Dcl*
a_c_obj$decl :	*Id, Sid* → *Dcl*
a_i_obj$decl :	*Id, Sid, Exp* → *Dcl*
a_obj$decl :	*Id, Sid* → *Dcl*
c_i_obj$decl :	*Id, Sid, Exp* → *Dcl*
c_obj$decl :	*Id, Sid* → *Dcl*
i_obj$decl :	*Id, Sid, Exp* → *Dcl*
obj$decl :	*Id, Sid* → *Dcl*

All forms of object declarations are normalized such that each declaration defines exactly one name. This is always possible by [3.3.1].

defining_identifier_list ::=
    defining_identifier { , defining_identifier }

B.3.3.2 Number Declarations

number_declaration ::=
  defining_identifier_list : **constant** := expression

real_const$decl :	*Id*, *Exp* → *Dcl*
int_const$decl :	*Id*, *Exp* → *Dcl*

Number declarations are disambiguated by static analysis into real and integer number declarations.

B.3.4 Derived Types and Classes

derived_type_definition ::=
  [abstract] **new** subtype_indication [record_extension_part]

der$type :	*Sid* → *Tdf*
ext$type :	*Sid*, rcd → *Tdf*

B.3.4.1 Derivation Classes

B.3.5 Scalar Types

range_constraint ::=
  range range

range$constr :	*Rng* → *Cns*

range ::=
  range_attribute_reference
 | simple_expression .. simple_expression

attr$range :	*Nam* → *Rng*
explicit$range :	*Exp*, *Exp* → *Rng*
parm_attr$range :	*Nam*, *Exp* → *Rng*

B.3.5.1 Enumeration Types

enumeration_type_definition ::=
  (enumeration_literal_specification { , enumeration_literal_specification })

enum$type :	Id^* → *Tdf*

enumeration_literal_specification ::=
  defining_identifier
 | defining_character_literal

defining_character_literal ::=
    character_literal

char$enum :	char → *Id*
id$enum :	*Id* → *Id*

B.3.5.2 Character Types

B.3.5.3 Boolean Types

B.3.5.4 Integer Types

integer_type_definition ::=
    signed_integer_type_definition
  | modular_type_definition

signed_integer_type_definition ::=
    range simple_expression .. simple_expression

int$type :	*Exp*, *Exp* → *Tdf*

modular_type_definition ::=
    mod expression

mod$type :	*Exp* → *Tdf*

B.3.5.5 Operations of Discrete Types

B.3.5.6 Real Types

real_type_definition ::=
    floating_point_definition
  | fixed_point_definition

B.3.5.7 Floating Point Types

floating_point_definition ::=
    digits expression [real_range_specification]

float$type :	*Exp* → *Tdf*
const_float$type :	*Exp*, *Cns* → *Tdf*

real_range_specification ::=
    range simple_expression .. simple_expression

See scalar types (3.5).

B.3.5.8 Operations of Floating Point Types

B.3.5.9 Fixed Point Types

fixed_point_definition ::=
  ordinary_fixed_point_definition
  | decimal_fixed_pointq_definition

ordinary_fixed_point_definition ::=
  delta expression real_range_specification

ord_fixed$type :	*Exp*, *Rng* → *Tdf*

decimal_fixed_point_definition ::=
  delta expression **digits** expression [real_range_specification]

const_dec_fixed$type :	*Exp*, *Exp*, *Cns* → *Tdf*
dec_fixed$type :	*Exp*, *Exp* → *Tdf*

decimal_digits_constraint ::=
  digits expression [range_constraint]

constr_digits$constr :	*Exp*, *Cns* → *Cns*
digits$constr :	*Exp* → *Cns*

B.3.5.10 Operations of Fixed Point Types

B.3.6 Array Types

array_type_definition ::=
  unconstrained_array_definition
  | constrained_array_definition

unconstrained_array_definition ::=
  array (index_subtype_definition { , index_subtype_definition }) **of** component_definition

aliased_uc_array$type :	Nam^*, *Sid* → *Tdf*
uc_array$type :	Nam^*, *Sid* → *Tdf*

index_subtype_definition ::=
  subtype_mark **range** <>

constrained_array_definition ::=
  array (discrete_subtype_definition { , discrete_subtype_definition }) **of** component_definition

aliased_array$type :	Rng^*, $Sid \rightarrow Tdf$
array$type :	Rng^*, $Sid \rightarrow Tdf$

discrete_subtype_definition ::=
    subtype_indication
  | range

Discrete subtype definitions are subsumed under subtype indications (*Sid*).

component_definition ::=
    [**aliased**] subtype_indication

B.3.6.1 Index Constraints and Discrete Ranges

index_constraint ::=
    (discrete_range { , discrete_range })

index$constr :	$Rng^* \rightarrow Cns$

discrete_range ::=
    subtype_indication
  | range

B.3.6.2 Operations of Array Types

B.3.6.3 String Types

B.3.7 Discriminants

discriminant_part ::=
    unknown_discriminant_part
  | known_discriminant_part

unknown_discriminant_part ::=
    (<>)

known_discriminant_part ::=
    (discriminant_specification { ; discriminant_specification })

box$discr :	$\rightarrow Dcp$
list$discr :	$Dcs^* \rightarrow Dcp$

discriminant_specification ::=
    defining_identifier_list : subtype_mark [:= default_expression]
  | defining_identifier_list : access_definition [:= default_expression]

default_expression ::=
        expression

acc\$discr :	*Id, Nam* $\rightarrow$ *Dcs*
acc_init\$discr :	*Id, Nam, Exp* $\rightarrow$ *Dcs*
init\$discr :	*Id, Nam, Exp* $\rightarrow$ *Dcs*
simple\$discr :	*Id, Nam* $\rightarrow$ *Dcs*

B.3.7.1 Discriminant Constraints

discriminant_constraint ::=
        (discriminant_association { , discriminant_association })

discr\$constr :	$Dca^* \rightarrow Cns$

discriminant_association ::=
        [selector_name { | selector_name } =>] expression

named\$assoc :	*Id, Exp* $\rightarrow$ *Dca*

B.3.7.2 Operations of Discriminated Types

B.3.8 Record Types

record_type_definition ::=
        [[abstract] **tagged**] [limited] record_definition

record\$type :	Cmp^*, *Vrp* $\rightarrow$ *Tdf*
t_record\$type :	Cmp^*, *Vrp* $\rightarrow$ *Tdf*

record_definition ::=
        record
        component_list
        end record
    | **null record**

component_list ::=
        component_declaration { component_declaration }
    | { component_declaration } variant_part
    | **null** ;

component_declaration ::=
        defining_identifier_list : component_definition [:= default_expression] ;

aliased_comp$decl :	*Id*, *Sid* → *Cmp*
comp$decl :	*Id*, *Sid* → *Cmp*
init_aliased_comp$decl :	*Id*, *Sid*, *Exp* → *Cmp*
init_comp$decl :	*Id*, *Sid*, *Exp* → *Cmp*

Component declarations with multiple identifiers are replaced by multiple component declarations.

B.3.8.1 Variant Parts and Discrete Choices

variant_part ::=
    case direct_name **is**
    variant
    { variant }
    end case ;

no$variant:	*Vrp*
variant$part :	*Nam*, Vnt^* → *Vrp*

variant ::=
    when discrete_choice_list =>
    component_list

variant$clause :	Dch^*, Cmp^*, *Vrp* → *Vnt*

discrete_choice_list ::=
    discrete_choice {
  | discrete_choice }

discrete_choice ::=
    expression
  | discrete_range
  | others

discr_other$choice :	→ *Dch*
Exp$choice :	*Exp* → *Dch*
range$choice :	*Rng* → *Dch*

B.3.9 Tagged Types and Type Extensions

B.3.9.1 Type Extensions

record_extension_part ::=
    with record_definition

B.3.9.2 Dispatching Operations of Tagged Types

B.3.9.3 Abstract Types and Subprograms

B.3.10 Access Types

access_type_definition ::=
    access_to_object_definition
  | access_to_subprogram_definition

access_to_object_definition ::=
    access [general_access_modifier] subtype_indication

general_access_modifier ::=
    all
  | constant

access$type :	*Sid* → *Tdf*
all_access$type :	*Sid* → *Tdf*
const_access$type :	*Sid* → *Tdf*

access_to_subprogram_definition ::=
    access [**protected**] **procedure** parameter_profile
    access [**protected**] **function** parameter_and_result_profile

func$type :	Pms^*, *Nam*→ *Tdf*
proc$type :	Pms^*→ *Tdf*

access_definition ::=
    access subtype_mark

B.3.10.1 Incomplete Type Declarations

incomplete_type_declaration ::=
    type defining_identifier [discriminant_part] ;

d_i_type$decl :	*Id*, *Dcp* → *Dcl*
i_type$decl :	*Id* → *Dcl*

B.3.10.2 Operations of Access Types

B.3.11 Declarative Parts

declarative_part ::=
    { declarative_item}

declarative_item ::=
    basic_declarative_item
  | body

basic_declarative_item ::=
    basic_declaration
  | representation_clause
  | use_clause

body ::=
    proper_body
  | body_stub

proper_body ::=
    subprogram_body
  | package_body
  | task_body
  | protected_body

B.3.11.1 Completions of Declarations

B.4 Names and Expressions

B.4.1 Names

name ::=
    direct_name
  | explicit_dereference
  | indexed_component
  | slice
  | selected_component
  | attribute_reference
  | type_conversion
  | function_call
  | character_literal

String and character literals that denote operators are included as direct names. String literals that denote string values are represented as aggregates.

char$*Exp* : integer → *Exp*

direct_name ::=
    identifier
  | operator_symbol

direct$name : *Id* → *Nam*

Names that are represented as strings, character literals, and identifiers are all treated uniformly as elements of type *Id*

```
prefix ::=
        name
      | implicit_dereference
```

```
explicit_dereference ::=
        name . all
```

deref$name : *Nam* → *Nam*

The abstract syntax for dereferencing includes explicit as well as implicit dereferencing.

```
implicit_dereference ::=
        name
```

B.4.1.1 Indexed Components

```
indexed_component ::=
        prefix ( expression { , expression } )
```

indexed$comp : $Nam, Exp^* \rightarrow Nam$

B.4.1.2 Slices

```
slice ::=
        prefix ( discrete_range )
```

slice$op : *Nam*, *Rng* → *Nam*

B.4.1.3 Selected Components

```
selected_component ::=
        prefix . selector_name
```

selected$comp : *Nam*, *Id* → *Nam*

Static semantics separates expanded names from selected components.

```
selector_name ::=
        identifier
      | character_literal
      | operator_symbol
```

B.4.1.4 Attributes

attribute_reference ::=
    prefix ' attribute_designator

attribute_designator ::=
    identifier [(expression)]
  | access
  | delta
  | digits

range_attribute_reference ::=
    prefix ' range_attribute_designator

range_attribute_designator ::=
    range [(expression)]

Id\$attr :	*Nam*, *Id* → *Nam*
param\$attr :	*Nam*, *Id*, *Exp* → *Nam*

There is special abstract syntax needed for attributes that are reserved words.

B.4.2 Literals

B.4.3 Aggregates

aggregate ::=
    record_aggregate
  | extension_aggregate
  | array_aggregate

B.4.3.1 Record Aggregates

record_aggregate ::=
    (record_component_association_list)

record_component_association_list ::=
    record_component_association { , record_component_association}
  | **null** record

null_record\$*Agg* :	→ *Agg*
record\$*Agg* :	*Rca** → *Agg*

record_component_association ::=
    [component_choice_list =>] expression

choice$assoc :	*Ccl*, *Exp* → *Rca*

Positional parameter associations have been eliminated by static analysis and are represented with an explicit choice list. This normalization is possible since, by [4.3.1], discriminant values that determine variants are required to be static.

```
component_choice_list ::=
        selector_name { | selector_name }
     |  others
```

list$choice :	Id^* → *Ccl*
others$choice :	→ *Ccl*

B.4.3.2 Extension Aggregates

```
extension_aggregate ::=
        ( expression with record_component_association_list )
```

ext$*Agg* :	*Exp*, Rca^* → *Agg*
null_ext$*Agg* :	*Exp* → *Agg*

B.4.3.3 Array Aggregates

```
array_aggregate ::=
        positional_array_aggregate
     |  named_array_aggregate
```

```
positional_array_aggregate ::=
        ( expression , expression { , expression} )
     |  ( expression { , expression} , others => expression )
```

```
named_array_aggregate ::=
        ( array_component_association { , array_component_association} )
```

named_array$*Agg* :	Aca^* → *Agg*
other_array$*Agg* :	Exp^*, *Exp* → *Agg*
pos_array$*Agg* :	Exp^* → *Agg*

```
array_component_association ::=
        discrete_choice_list => expression
```

array_comp$assoc :	Dch^*, *Exp* → *Aca*

B.4.4 Expressions

expression ::=
relation { **and** relation }
| relation { **and then** relation }
| relation { **or** relation }
| relation { **or else** relation }
| relation { **xor** relation }

and_then$*Exp* :	*Exp*, *Exp* → *Exp*
or_else$*Exp* :	*Exp*, *Exp* → *Exp*

Short-circuit operators are non-strict and require explicit representation.

relation ::=
simple_expression [relational_operator simple_expression]
| simple_expression [**not**] **in** range
| simple_expression [**not**] **in** subtype_mark

in_name$*Exp* :	*Exp*, *Nam* → *Exp*
in_range$*Exp* :	*Exp*, *Rng* → *Exp*
in_type$*Exp* :	*Exp*, *Nam* → *Exp*
not_in_name$*Exp* :	*Exp*, *Nam* → *Exp*
not_in_range$*Exp* :	*Exp*, *Rng* → *Exp*
not_in_type$*Exp* :	*Exp*, *Nam* → *Exp*

Static semantics distinguishes between membership tests where the name denotes an object and those where the name denotes a subtype.

simple_expression ::=
[unary_adding_operator] term { binary_adding_operator term }

term ::=
factor { multiplying_operator factor }

factor ::=
primary [** primary]
| **abs** primary
| **not** primary

primary ::=
numeric_literal
| null
| string_literal
| aggregate
| name
| qualified_expression
| allocator
| (expression)

paren\$*Exp* :	*Exp* → *Exp*
qual\$aggregate :	*Nam*, *Agg* → *Exp*
name\$*Exp* :	*Nam* → *Exp*
null\$*Exp* :	→ *Exp*
integer\$*Exp* :	integer → *Exp*
real\$*Exp* :	real → *Exp*

All aggregates are assumed to be qualified by static analysis. String literals are represented as qualified aggregates. It may be necessary for static analysis to introduce new type names for aggregates of anonymous array types.

Numeric literals are separated into integer and real literals.

B.4.5 Operators and Expression Evaluation

All strict operators on ordinary values are represented in the abstract syntax as function calls. As with other function calls, the function designators specify the unique overload that applies. Non-strict operators or operators that take subtypes as arguments (e.g., **in**) have an explicit representation given below.

```
logical_operator ::=
        and
    |   or
    |   xor


relational_operator ::=
        =
    |   /=
    |   <
    |   <=
    |   >
    |   >=


binary_adding_operator ::=
        +
    |   -
    |   &


unary_adding_operator ::=
        -
    |   +


multiplying_operator ::=
        *
    |   /
    |   mod
    |   rem
```

highest_precedence_operator ::=
        **
    | abs
    | not

B.4.5.1 Logical Operations and Short-Circuit Control Forms

B.4.5.2 Relational Operators and Membership Tests

B.4.5.3 Binary Adding Operators

B.4.5.4 Multiplying Operators

B.4.5.5 Highest Precedence Operators

B.4.6 Type Conversions

type_conversion ::=
        subtype_mark (expression)
    | subtype_mark (name)

type$conversion :	*Nam*, *Exp* → *Exp*
name_type$conversion :	*Nam*, *Nam* → *Nam*

B.4.7 Qualified Expressions

qualified_expression ::=
        subtype_mark' (expression)
    | subtype_mark ' aggregate

qual$*Exp* :	*Nam*, *Exp* → *Exp*

The abstract syntax for qualified aggregates is covered under aggregates.

B.4.8 Allocators

allocator ::=
        new subtype_indication
    | **new** qualified_expression

Exp$alloc :	*Exp* → *Exp*
type$alloc :	*Sid* → *Exp*

B.4.9 Static Expressions and Static Subtypes

B.4.9.1 Statically Matching Constraints and Subtypes

B.5 Statements

B.5.1 Simple and Compound Statements – Sequences of Statements

```
sequence_of_statements ::=
        statement { statement }
```

Sequences of statements can be treated as single statements.

Stm$lst : $Stm^* \rightarrow Stm$

```
statement ::=
        { label } simple_statement
      | { label } compound_statement
```

labeld$stm : $Id, Stm \rightarrow Stm$

```
simple_statement ::=
        null_statement
      | assignment_statement
      | exit_statement
      | goto_statement
      | procedure_call_statement
      | return_statement
      | entry_call_statement
      | requeue_statement
      | delay_statement
      | abort_statement
      | raise_statement
      | code_statement
```

```
compound_statement ::=
        if_statement
      | case_statement
      | loop_statement
      | block_statement
      | accept_statement
      | select_statement
```

```
null_statement ::=
        null ;
```

null$stm : $\rightarrow Stm$

```
label ::=
        << statement_identifier >>
```

```
statement_identifier ::=
        direct_name
```

B.5.2 Assignment Statement

```
assignment_statement ::=
        name := expression
```

```
assign$stm :   Nam, Exp → Stm
```

B.5.3 If Statements

```
if_statement ::=
        if condition then
            sequence_of_statements
        { elsif condition then
            sequence_of_statements}
        [ else
            sequence_of_statements ]
        end if ;
```

```
if$stm :          Cnd, Stm, Eif → Stm
if_else$stm :     Cnd, Stm, Eif, Stm → Stm
Eif$lst :         Eif* → Eif
elsif$clause :    Cnd, Stm → Eif
```

```
condition ::=
        expression
```

```
Exp$condition :   Exp → Cnd
```

B.5.4 Case Statements

```
case_statement ::=
        case expression is
            case_statement_alternative
            { case_statement_alternative }
        end case ;
```

```
case$stm :   Exp, Alt → Stm
```

case_statement_alternative ::=
  when discrete_choice_list =>
  sequence_of_statements

Alt$lst :	$Alt^* \rightarrow Alt$
case$*Alt* :	$Dch^*, Stm \rightarrow Alt$

B.5.5 Loop Statements

loop_statement ::=
  [statement_identifier :]
  [iteration_scheme] loop
  sequence_of_statements
  end loop [identifier] ;

iteration_scheme ::=
  while condition
  | **for** loop_parameter_specification

loop_parameter_specification ::=
  defining_identifier **in** [**reverse**] discrete_subtype_definition

named_for$loop :	$Id, Id, Rng, Stm \rightarrow Stm$
named_reverse$loop :	$Id, Id, Rng, Stm \rightarrow Stm$
named_while$loop :	$Id, Cnd, Stm \rightarrow Stm$
named$loop :	$Id, Stm \rightarrow Stm$
for$loop :	$Id, Rng, Stm \rightarrow Stm$
reverse$loop :	$Id, Rng, Stm \rightarrow Stm$
while$loop :	$Cnd, Stm \rightarrow Stm$
plain$loop :	$Stm \rightarrow Stm$

B.5.6 Block Statements

block_statement ::=
  [statement_identifier :]
    [declare
      declarative_part]
    begin
      handled_sequence_of_statements
  end [identifier] ;

simple$block :	$Stm \rightarrow Stm$
declare$block :	$Dcl, Stm \rightarrow Stm$
named$block :	$Id, Stm \rightarrow Stm$
named_declare$block :	$Id, Dcl, Stm \rightarrow Stm$

B.5.7 Exit Statements

exit_statement ::=
    exit [name] [**when** condition] ;

name$exit :	*Nam* → *Stm*
name_cond$exit :	*Nam, Cnd* → *Stm*
plain$exit :	→ *Stm*
cond$exit :	*Cnd* → *Stm*

B.5.8 Goto Statements

goto_statement ::=
    goto name ;

goto$stm :	*Nam* → *Stm*

B.6 Subprograms

B.6.1 Subprogram Declarations

subprogram_declaration ::=
    subprogram_specification ;

subp$spec :	*Sps* → *Dcl*

abstract_subprogram_declaration ::=
    subprogram_specification **is abstract** ;

subp$spec :	*Sps* → *Dcl*

subprogram_specification ::=
    procedure defining_program_unit_name parameter_profile
  | **function** defining_designator parameter_and_result_profile

function$spec :	Id, Pms^*, $Nam \rightarrow Sps$
procedure$spec :	Id, $Pms^* \rightarrow Sps$

designator ::=
    [parent_unit_name .] identifier
  | operator_symbol

defining_designator ::=
    defining_program_unit_name
  | defining_operator_symbol

defining_program_unit_name ::=
  [parent_unit_name .] defining_identifier

operator_symbol ::=
  string_literal

defining_operator_symbol ::=
  operator_symbol

parameter_profile ::=
  [formal_part]

no$params :	$\rightarrow Psig$
param$list :	$Pms^* \rightarrow Psig$

parameter_and_result_profile ::=
  [formal_part] **return** subtype_mark

formal_part ::=
  (parameter_specification { ; parameter_specification })

parameter_specification ::=
  defining_identifier_list : mode subtype_mark [:= default_expression]
  | defining_identifier_list : access_definition [:= default_expression]

access$param :	$Id, Nam \rightarrow Pms$
access_default$param :	$Id, Nam, Exp \rightarrow Pms$
default$param :	$Id, Mde, Nam, Exp \rightarrow Pms$
normal$param :	$Id, Mde, Nam \rightarrow Pms$

mode ::=
  [**in**]
  | **in** out
  | out

in$mode :	$\rightarrow Mde$
in_out$mode :	$\rightarrow Mde$
no$mode :	$\rightarrow Mde$
out$mode :	$\rightarrow Mde$

B.6.2 Formal Parameter Modes

B.6.3 Subprogram Bodies

```
subprogram_body ::=
        subprogram_specification is
            declarative_part
        begin
            handled_sequence_of_statements
        end [ designator ] ;
```

subp$body :	*Sps, Dcl, Stm* → *Dcl*

B.6.3.1 Conformance Rules

B.6.3.2 Inline Expansion of Subprograms

B.6.4 Subprogram Calls

```
procedure_call_statement ::=
        name
     |  prefix actual_parameter_part ;
```

call$stm :	*Nam*, Pss^* → *Stm*

```
function_call ::=
        name
     |  prefix actual_parameter_part
```

func$call :	*Nam*, Pss^* → *Nam*

```
actual_parameter_part ::=
        ( parameter_association { , parameter_association } )
```

```
parameter_association ::=
        [ selector_name => ] explicit_actual_parameter
```

```
explicit_actual_parameter ::=
        expression
     |  name
```

named_exp$arg :	*Id, Exp* → *Pss*
named_name$arg :	*Id, Nam* → *Pss*

B.6.4.1 Parameter Associations

B.6.5 Return Statements

```
return_statement ::=
        return [ expression ] ;
```

func_return$stm :	$Exp \rightarrow Stm$
return$stm :	$\rightarrow Stm$

B.6.6 Overloading of Operators

B.7 Packages

B.7.1 Package Specifications and Declarations

```
package_declaration ::=
        package_specification ;
```

```
package_specification ::=
        package defining_program_unit_name is
            {basic_declarative_item}
        [ private
            {basic_declarative_item} ]
        end [ [ parent_unit_name . ] identifier ]
```

B.7.2 Package Bodies

```
package_body ::=
        package body defining_program_unit_name is
            declarative_part
        [ begin
            handled_sequence_of_statements
        end [ [ parent_unit_name . ] identifier ] ;
```

B.7.3 Private Type and Private Extensions

```
private_type_declaration ::=
        type defining_identifier [ discriminant_part ] is [ [ abstract ] tagged ] [ limited ] private ;
```

```
private_extension_declaration ::=
        type defining_identifier [ discriminant_part ] is
        [ abstract ] new subtype_indication with private ;
```

d_ext$decl :	$Id,\ Dcp,\ Sid \rightarrow Dcl$
ext$decl :	$Id,\ Sid \rightarrow Dcl$

B.7.3.1 Operations of Private Types and Private Extensions

B.7.4 Deferred Constants

B.7.5 Limited Types

B.7.6 User-Defined Assignment and Finalization

B.7.6.1 Completion and Finalization

B.8 Visibility Rules

B.8.1 Declarative Region

B.8.2 Scope of Declarations

B.8.3 Visibility

B.8.4 Use Clauses

```
use_clause ::=
        use_package_clause
    |   use_type_clause
```

```
use_package_clause ::=
        use name { , name } ;
```

use$clause : $Nam^* \rightarrow Dcl$

```
use_type_clause ::=
        use type subtype_mark { , subtype_mark } ;
```

use_type$clause : $Nam^* \rightarrow Dcl$

B.8.5 Renaming Declarations

```
renaming_declaration ::=
        object_renaming_declaration
    |   exception_renaming_declaration
    |   package_renaming_declaration
    |   subprogram_renaming_declaration
    |   generic_renaming_declaration
```

B.8.5.1 Object Renaming Declarations

```
object_renaming_declaration ::=
        defining_identifier : subtype_mark renames name ;
```

object$renaming : $Id, Nam, Nam \rightarrow Dcl$

B.8.5.2 Exception Renaming Declarations

exception_renaming_declaration ::=
    defining_identifier : **exception renames** name ;

exception$renaming : *Id*, *Nam* → *Dcl*

B.8.5.3 Package Renaming Declarations

package_renaming_declaration ::=
    package defining_program_unit_name **renames** name ;

B.8.5.4 Subprogram Renaming Declarations

subprogram_renaming_declaration ::=
    subprogram_specification **renames** name ;

subp$renaming : *Sps*, *Nam* → *Dcl*

B.8.5.5 Generic Renaming Declarations

generic_renaming_declaration ::=
    generic package defining_program_unit_name **renames** name ;
  | **generic procedure** defining_program_unit_name **renames** name ;
  | **generic function** defining_program_unit_name **renames** name ;

B.8.6 The Context of Overload Resolution

B.9 Tasks and Synchronization

B.9.1 Task Units and Task Objects

task_type_declaration ::=
    task type defining_identifier [known_discriminant_part] [**is** task definition] ;

single_task_declaration ::=
    task defining_identifier [**is** task definition] ;

task_definition ::=
    { task_item }
    [private
    {task_item}]
    end [identifier]

```
task_item ::=
        entry_declaration
      | representation_clause
```

```
task_body ::=
        task body defining_identifier is
        declarative_part
        begin
        handled_ sequence_of_statements
        end [ identifier ] ;
```

B.9.2 Task Execution – Task Activation

B.9.3 Task Dependence – Termination of Tasks

B.9.4 Protected Units and Protected Objects

```
protected_type_declaration ::=
        protected type defining_identifier [ known_discriminant_part ] is protected_definition ;
```

```
single_protected_declaration ::=
        protected defining_identifier is protected_definition ;
```

```
protected_definition ::=
        { protected_operation_declaration }
        [ private
        {protected_element_declaration} ]
        end [ identifier ]
```

```
protected_operation_declaration ::=
        subprogram_declaration
      | entry_declaration
```

```
protected_element_declaration ::=
        protected_operation_declaration
      | component_declaration
```

```
protected_body ::=
        protected body defining_identifier is
        {protected_operation_item}
        end [ identifier ] ;
```

```
protected_operation_item ::=
        subprogram_declaration
      | subprogram_body
      | entry_body
```

B.9.5 Intertask Communication

```
entry_declaration ::=
        entry defining_identifier [ ( discrete_subtype_definition ) ] parameter_profile ;


accept_statement ::=
        accept direct_name [ ( entry_index ) ] parameter_profile [ do
            handled_sequence_of_statements
        end [ identifier ] ] ;


entry_index ::=
        expression


entry_body ::=
        entry defining_identifier entry_body_formal_part entry_barrier is
        declarative_part
        begin
        handled_sequence_of_statements
        end [ identifier ] ;


entry_body_formal_part ::=
        [ ( entry_index_specification ) ]parameter_profile


entry_barrier ::=
        when condition


entry_index_specification ::=
        for defining_identifier in discrete_subtype_definition


entry_call_statement ::=
        name [ actual_parameter_part ] ;


requeue_statement ::=
        requeue name [ with abort ] ;
```

B.9.6 Delay Statements, Duration, and Time

```
delay_statement ::=
        delay_until_statement
     |  delay_relative_statement


delay_until_statement ::=
        delay until expression ;
```

```
delay_relative_statement ::=
        delay expression ;
```

B.9.7 Select Statements

```
select_statement ::=
        selective_accept
      | timed_entry_call
      | conditional_entry_call
      | asynchronous_select
```

B.9.8 Selective Accept

```
selective_accept ::=
        select
            [ guard ] select_alternative
        [ or
            [ guard ] select_alternative ]
        [ else
            sequence_of_statements ]
        end select ;
```

```
guard ::=
        when condition =>
```

```
select_alternative ::=
        accept_alternative
      | delay_alternative
      | terminate_alternative
```

```
accept_alternative ::=
        accept_statement [ sequence_of_statements ]
```

```
delay_alternative ::=
        delay_statement [ sequence_of_statements ]
```

```
terminate_alternative ::=
        terminate ;
```

B.9.9 Timed Entry Calls

```
timed_entry_call ::=
        select
            entry_call_alternative
```

```
        or
            delay_alternative
        end select ;
```

```
entry_call_alternative ::=
        entry_call_statement [ sequence_of_statements ]
```

B.9.10 Conditional Entry Calls

```
conditional_entry_call ::=
        select
            entry_call_alternative
        else
            sequence_of_statements
        end select ;
```

B.9.11 Asynchronous Transfer of Control

```
asynchronous_select ::=
        select
        triggering_alternative
        then abort
        abortable_part
        end select ;
```

```
triggering_alternative ::=
        triggering_statement [ sequence_of_statements ]
```

```
triggering_statement ::=
        entry_call_statement
    |   delay_statement
```

```
abortable_part ::=
        sequence_of_statements
```

B.9.12 Abort of a Task – Abort of a Sequence of Statements

```
abort_statement ::=
        abort name { , name} ;
```

B.9.13 Task and Entry Attributes

B.9.14 Shared Variables

B.9.15 Example of Tasking and Synchronization

B.10 Program Structure and Compilation Issues

B.10.1 Separate Compilation

B.10.1.1 Compilation Units – Library Units

```
compilation ::=
        {compilation_unit}


compilation_unit ::=
        context_clause library_item
    |   context_clause subunit


library_item ::=
        [ private ] library_unit_declaration
    |   library_unit_body


library_unit_declaration ::=
        subprogram_declaration
    |   package_declaration
    |   generic_declaration
    |   generic_instantiation
    |   library_unit_renaming_declaration


library_unit_renaming_declaration ::=
        package_renaming_declaration
    |   generic_renaming_declaration
    |   subprogram_renaming_declaration


library_unit_body ::=
        subprogram_body
    |   package_body


parent_unit_name ::=
        name
```

lib$unit :	Cit^*, $Dcl \rightarrow$ cmu
private$unit :	Cit^*, $Dcl \rightarrow$ cmu
sub$unit :	Cit^*, Nam, $Dcl \rightarrow$ cmu

Renaming of library units is dealt with in static semantics.

B.10.1.2 Context Clauses – With Clauses

```
context_clause ::=
        { context_item}
```

```
context_item ::=
        with_clause
      | use_clause
```

```
with_clause ::=
        with name { , name } ;
```

with$context : $Nam^* \rightarrow Cit$

B.10.1.3 Subunits of Compilation Units

```
body_stub ::=
        subprogram_body_stub
      | package_body_stub
      | task_body_stub
      | protected_body_stub
```

```
subprogram_body_stub ::=
        subprogram_specification is separate ;
```

s_subp$spec : $Sps \rightarrow Dcl$

```
package_body_stub ::=
        package body defining_identifier is separate ;
```

```
task_body_stub ::=
        task body defining_identifier is separate ;
```

```
protected_body_stub ::=
        protected body defining_identifier is separate ;
```

```
subunit ::=
        separate ( parent_unit_name ) proper_body
```

B.10.1.4 The Compilation Process

B.10.1.5 Pragmas and Compilations

B.10.1.6 Environment-Level Visibility Rules

B.10.2 Program Execution

B.10.2.1 Elaboration Control

B.11 Exceptions

B.11.1 Exception Declarations

```
exception_declaration ::=
        defining_identifier_list : exception ;
```

excpt$decl :	*Id* → *Dcl*

As with object declarations, only a single name is defined by each exception declaration.

B.11.2 Exception Handlers

```
handled_sequence_of_statements ::=
        sequence_of_statements
        [ exception
        exception_handler
        { exception_handler } ]
```

handled$statement :	*Stm*, *Xhd** → *Stm*
unhandled$statement :	*Stm* → *Stm*

```
exception_handler ::=
        when [ choice_parameter_specification : ] exception_choice { | exception_choice } =>
            sequence_of_statements
```

choice$handler :	*Id*, *Ech**, *Stm* → *Xhd*
expt$handler :	*Ech**, *Stm* → *Xhd*

```
choice_parameter_specification ::=
        defining_identifier
```

```
exception_choice ::=
        name
      | others
```

named$excpt :	*Nam* → *Ech*
others$excpt :	→ *Ech*

B.11.3 Raise Statements

raise_statement ::=
        raise [name] ;

raise$stm :	*Nam* → *Stm*
reraise$stm :	→ *Stm*

B.11.4 Exception Handling

B.11.5 Suppressing Checks

B.11.6 Exceptions and Optimization

B.12 Generic Units

B.12.1 Generic Declarations

generic_declaration ::=
        generic_subprogram_declaration
    | generic_package_declaration

generic_subprogram_declaration ::=
        generic_formal_part subprogram_specification ;

generic_package_declaration ::=
        generic_formal_part package_specification ;

generic_formal_part ::=
        generic { generic_formal_parameter_declaration
    | use_clause }

generic_formal_parameter_declaration ::=
        formal_object_declaration
    | formal_type_declaration
    | formal_subprogram_declaration
    | formal_package_declaration

B.12.2 Generic Bodies

B.12.3 Generic Instantiation

generic_instantiation ::=
        package defining_program_unit_name **is**
            new name [generic_actual_part] ;
    | **procedure** defining_program_unit_name **is**
            new name [generic_actual_part] ;

| **function** defining_designator is
    new name [generic_actual_part] ;

generic_actual_part ::=
    (generic_association { , generic_association })

generic_association ::=
    [selector_name =>] explicit_generic_actual_parameter

explicit_generic_actual_parameter ::=
    expression
  | name
  | subtype_mark

B.12.4 Formal Objects

formal_object_declaration ::=
    defining_identifier_list : mode subtype_mark [:= default_expression] ;

init_formal$obj :	Id^*, Mde, Nam, $Exp \rightarrow Gpd$
formal$obj :	Id^*, Mde, $Nam \rightarrow Gpd$

B.12.5 Formal Types

formal_type_declaration ::=
    type defining_identifier [discriminant_part] is formal_type_definition ;

formal_type_definition ::=
    formal_private_type_definition
  | formal_derived_type_definition
  | formal_discrete_type_definition
  | formal_signed_integer_type_definition
  | formal_modular_type_definition
  | formal_floating_point_definition
  | formal_ordinary_fixed_point_definition
  | formal_decimal_fixed_point_definition
  | formal_array_type_definition
  | formal_access_type_definition

B.12.5.1 Formal Private and Derived Types

formal_private_type_definition ::=
    [[abstract] tagged] [limited] private

formal_derived_type_definition ::=
[abstract] **new** subtype_mark [with private]

B.12.5.2 Formal Scalar Types

formal_discrete_type_definition ::=
(<>)

formal_signed_integer_type_definition ::=
range <>

formal_modular_type_definition ::=
mod <>

formal_floating_point_definition ::=
digits <>

formal_ordinary_fixed_point_definition ::=
delta <>

formal_decimal_fixed_point_definition ::=
digits <> **delta** <>

B.12.5.3 Formal Array Types

formal_array_type_definition ::=
array_type_definition

B.12.5.4 Formal Access Types

formal_access_type_definition ::=
access_type_definition

B.12.6 Formal Subprograms

formal_subprogram_declaration ::=
with subprogram_specification [is subprogram_default] ;

subprogram_default ::=
default_name
| <>

default_name ::=
name

B.12.7 Formal Packages

```
formal_package_declaration ::=
        with package defining_identifier is new name formal_package_actual_part ;
```

```
formal_package_actual_part ::=
        ( <> )
      | [ generic_actual_part ]
```

B.12.8 Example of a Generic Package

B.13 Representation Clauses and Implementation–Dependent Features

```
representation_clause ::=
        attribute_definition_clause
      | enumeration_representation_clause
      | record_representation_clause
      | at_clause
```

```
attribute_definition_clause ::=

        for direct_name ' attribute_designator use expression ;
      | for direct_name ' attribute_designator use name ;
```

```
enumeration_representation_clause ::=
        for direct_name use enumeration_aggregate ;
```

```
enumeration_aggregate ::=
        array_aggregate
```

```
record_representation_clause ::=
        for direct_name use
        record [ mod_clause ]
        {component_clause}
        end record ;
```

```
component_clause ::=
        component_clause_component_name at position range first_bit .. last_bit ;
```

```
component_clause_component_name ::=
        direct_name
      | direct_name ' attribute_designator
```

```
position ::=
        expression


first_bit ::=
        simple_expression


last_bit ::=
        simple_expression


code_statement ::=
        qualified_expression ;
```

agg_code$stm :	*Nam, Agg* → *Stm*
exp_code$stm :	*Nam, Exp* → *Stm*

```
restriction ::=
        identifier
    |   identifier => expression


delta_constraint ::=
        delta expression [ range_constraint ]
```

constr_delta$constr :	*Exp, Cns* → *Cns*
delta$constr :	*Exp* → *Cns*

```
at_clause ::=
        for directed_name use at expression ;


mod_clause ::=
        at mod expression ;
```

B.14 Ada 9X Input–Output

Index

$\vdash_{Stl}$, *65*
$\vdash_{agg}$, *71*
$\vdash_{chc}$, *76*
$\vdash_{clt}$, *76*
$\vdash_{cmp}$, *66*
$\vdash_{dcl}$, *61–63*, *68*, *79*
$\vdash_{elf}$, *75*, *76*
$\vdash_{exp}$, *71–74*
$\vdash_{nam}$, *68*, *70*, *73*, *81*
$\vdash_{psi}$, *62*
$\vdash_{rng}$, *64*
$\vdash_{sid}$, *62*
$\vdash_{stm}$, *75–78*, *81*, *82*
$\vdash_{tdf}$, *63*, *65*, *66*
$\vdash_{vrn}$, *66*
$\vdash_{attr}$, *83*
$\vdash_{sid_i}$, *67*
Nh-Nl, 83
Rng_lst, 65
. . . , 65, 66, 69, 71, 74, 76, 77, 79, 81
abnormal_state, 49, 51, 64, 69, 70, 74
access_type, 58
access_value, 54, 55
access_val, 54, 68, 74
actualized_binding_list, 60, *60*
actualized_complist, *61*
actualized_components, *61*, 61, 71
actualized_constraint, 60, *60*, 60, 61
actualized_partial_range, *60*, 60
actualized_range_list, 60, *60*
actualized_value, *60*, 60
actualized_variants, *61*, 61
actualize, 57, 61, *61*
adom, 56, 57, 79
all_access, 67
ancestor, 55, *55*, 56, 73
append_components, *61*
array_aggregate, *71*
array_component, 50, 69
array_slice, 50, 69
array_type, 56, 59, 65, 69
array_value, *55*, 55, 56
array_val, 50, 55, 56, 59, 69
assign_fn, 62, 63, 75
base_range, 55, *55*, 56
base, 83
belongs_to, *53*, 53, 56, 59, 69, 72, 77
bind_actuals, 80, *80*, 81
choice_lst, 76, 77
choice_range, 76, 77
choice_value, 76, 77
cl_value, 57, *57*, 57
class_type, 56
compatible, 62, 68
component_actions, 71, *71*
component_type, *57*, 57, 70
composite_value, 55
constant_access, 67
constrain, 69
convert_return_value, 80, *82*, 82
covers_fn, 76
covers, *57*, 77, *77*, 77
default_value_fn, 62, 63
descendant, 55, *55*
descriptor_value, 56, *56*, 57, 60
discrete_range, 53
discrete_rng, 53–56, 60
discrete_value, 54
discrete_val, 53, 54, 56, 60, 64, 70, 72, 76–78, 83
discriminant_constraint, 59, 60, 71
discriminant_ref, 60
discriminant_union, 67, *67*, 67
discriminant_value, *56*, 56, 57
discr, 56, 57, 67, 68, 71
elementary_value, *54*, 55, *55*
enum_type, 55, 56, 58
exception, 49, 69, 70, 74, 76, 80, 81
exit, 49, 76–78, 80
expression_list, *68*, 68, 69
fields, 57, 61, 66, 67
finalize_fn, 75
find_component, 71, *71*, 71
first, 83

formal, 79
func_profile, 59, 67
func_return, 49, 80–82
given_parameter, *82*, 82
high_bound, 53, 83
in_mode, 79, 82
in_out_mode, 79, 82
included_in, *53*, 53, 69
incomplete_type, 68
index_actions, 65, *65*
index_constraint, 59, 60, 65, 83
index_list, *69*, 69
index_pairing, *54*, 54
indices, *54*, 54, 55
indirect_discriminant_constraint, 60
indirect_index_constraint, 60
is_access_to_object_type, 58, *58*
is_access_to_subprogram_type, 58, *58*, *59*, 59
is_access_type, 57, 58, *58*
is_array_type, 58, 59, *59*
is_boolean_type, 58
is_by_copy_type, *59*, 59
is_by_reference_type, 59, *59*
is_character_type, 58
is_composite_type, *57*, *58*, 58, *58*
is_discrete_type, 58, *58*, 58
is_elementary_type, 57, *57*, 59
is_enumeration_type, *58*, 58
is_integer_type, 58, *58*
is_modular_integer_type, 58, *58*, 58
is_protected_type, 58, 59, *59*
is_real_type, 58, *58*
is_record_type, 58, 59, *59*
is_scalar_type, 57, *58*, 58, 83
is_signed_integer_type, 58
is_string_type, *59*, 59
is_tagged_type, 58, 59, *59*, 73
is_task_type, 59, *59*
last, 83
length, 83
location, 50, 62, 63, 70, 74
loop_id, 77, 78
loop_view, 77, 78
low_bound, 53, 83
make_range, 53, 64, 83
make_state, 49, 50
modular_type, 55, 56, 58, 64
new_object_fn, 52, 62, 63, 74
new_object, *50*, 51, 52, 63, 70
new_subprogram, *51*, 51, 79
new_subtype, *51*, 51, 61, 62, 68, 83
new_type_fn, 66, 67
new_type, *51*, 51, 63–65, 67, 68
normal_state, 49, 64, 69, 74
normal, 49, 51, 52, 69, 70, 74, 76–78, 80–82
not_used, 63, 65, 66, 68
null_range, *53*, 54, *54*, 54, 69
object_view, 62, 63, 68–70, 74, 75, 83
operator, 81
out_mode, 79, 82
p_constraint, 62
p_discriminant_constraint, 60
p_index_constraint, 60
p_indirect_discriminant_constraint, 60
p_indirect_index_constraint, 60
p_range_constraint, 60
p_subtype, 61
p_value, 60
parameter_action, 82, *82*, 82, *82*
parameter_list, 81, *82*, 82
parameters, *79*
pick, 52, *52*
pool_access, 67
prefix_set_with_element, 54, *54*
proc_exit, 80, *80*, 80, *80*
proc_profile, 59, 67
proc_return, 49, 80, 82
range_constraints, 65, *65*
range_constraint, 56, 59, 60, 65, 69
range_of_subtype, 56, *56*, 72, 83
ranges_of_subtypes, 56, *56*
real_range, 53
real_rng, 53, 54
real_value, 54
real_val, 54, 70
record_component, 50, 70
record_type, 57, 59, 66, 67, 71
record_val, 50, 55, 57, 59, 70, 71, 73
return_check, 80, *80*, 81
return_value, 81
run, *52*, *53*
satisfies, *59*, 59, 60, 74
scalar_value, 54, *54*, 54, 55
select_component_type, *57*, 57, *57*
set_of, 54, 56, 57
signed_integer_type, 55, 56, 64
slice_check, 69, *69*, 69
some, 57, 59, 61, 63, 64, 66, 67, 73, 79, 81, 82
subprogram_body_fn, 53, 81
subprogram_body, 53, 80, *80*, 81, *81*
subprogram_view, 79–81
subprogram, 79–81

subtype_value, 56, 57, *60*, 60
subtype_view, 61, 62, 65, 67, 68, 71–74, 79, 83
subtype, 55, 56, 58, 60–62, 64–68, 71, 73, 74, 83
test_in, *72*, 73, *73*
the_parameter, *82*, 82
the_store, 49–51
the_variant, 57, 61
then, 52
thunk, 79, 82
type_constraints, 65
type_struct, *55*, 55, 56, 58, 59, 69–71
ultimate_ancestor, 55, *55*, 55
unelaborated, 79, 81
universal_integer_tn, 63, 83
universal_integer_type, 56
universal_real_tn, 63
value_split_fn, 75
values_in_range, 54, *54*
variant_values, 57, *57*
variant, 57, 66
$\vdash content(_)$, *50*
$\vdash_{att}$, *83*
$_[_ \mapsto_1 _]$, *50*
$_[_ \mapsto_2 _]$, *50*
$_[_ \mapsto_3 _]$, *50*
$_[_ \mapsto_4 _]$, *50*
$_[_ \mapsto_1 _]$, *50*
$_[_ \mapsto_2 _]$, *50*
$_^{obj}[_]$, *50*
$_^{spg}[_]$, *50*
$_^{stp}[_]$, *50*
$_^{typ}[_]$, *50*
$\langle\rangle$, *68*
$[\ldots]$, *51*
$\{\ldots\}$, *51*

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1995	Contractor Report

4. TITLE AND SUBTITLE

Towards a Formal Semantics for Ada 9X

5. FUNDING NUMBERS

C NAS1-18972

WU 505-64-10-56

6. AUTHOR(S)

David Guaspari, John McHugh, Wolfgang Polak, and Mark Saaltink

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Odyssey Research Associates
301 Dates Drive
Ithaca, NY 14850-1326

8. PERFORMING ORGANIZATION REPORT NUMBER

ORA-TM-95-0044

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CR-195037

11. SUPPLEMENTARY NOTES

Langley Technical Monitor: C. Michael Holloway
Final Report - Task 11

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Ada 9X Language Precision Team was formed during the revisions of Ada 83, with the goal of analyzing the proposed design, identifying problems, and suggesting improvements, through the use of mathematical models.

This report defines a framework for formally describing Ada 9X, based on Kahn's "Natural Semantics", and applies the framework to portions of the language. The proposals for exceptions and optimization freedoms are also analyzed, using a different technique.

14. SUBJECT TERMS

Ada 95, formal semantics, natural semantics, optimization

15. NUMBER OF PAGES

157

16. PRICE CODE

A08

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		