Syntax and Static Semantics of Eiffel
A Case Study in Algebraic Specification Techniques

Eelco Visser
Programming Research Group
Department of Mathematics and Computer Science
University of Amsterdam
email: visser@fwi.uva.nl

December 3, 1992

Abstract
An algebraic specification of the syntax and static semantics as defined in [Mey92] in the specification formalism ASF+SDF is presented. In support of the actual typechecking modules several reusable, general purpose modules, a mechanism based on λ-calculus to reuse these modules, and a language to manipulate symbol tables are designed. Through this language the symbol table can be hidden from the specification of typecheck functions; the symbol table behaves like a global environment for these functions.
1 Introduction

1.1 General

This paper is the result of an undergraduate research project aimed at the specification of the syntax and static semantics of the object oriented programming language Eiffel.

The project fits in the framework of the research on the Generation of Interactive Programming Environments (GIPE) performed at the Centre for Mathematics and Computer Science (CWI) in Amsterdam and at the University of Amsterdam. The core of this research is a specification formalism (ASF+SDF) designed primarily for the description of syntax and semantics of programming languages. In support of the formalism a meta-environment is being developed which enables the programmer to edit and typecheck specifications and which generates, from a specification, a programming environment for the language described.

The current project is one in a series of case studies on the validation of ASF+SDF as a formalism suitable for the specification of large programming languages. Other aims of these studies are to test the meta-environment and feed the results to the implementors of the (prototype) system, and to gain experience in writing complex algebraic specifications.

Another issue in the theory of programming languages is the definition of languages as the composition of atomic language definitions.

One means of transferring experience gained to new projects and to achieve composability is a library of reusable general purpose modules. Therefore a considerable part of the specification is devoted to the definition of modules that describe common datatypes such as sets, sequences and tables, which are heavily used in the definition of symbol tables. Furthermore a (provisory) mechanism, is designed (and implemented) to overcome the absence of a parameter mechanism in ASF+SDF. In section 6 we will evaluate how much composability we have achieved and consider some approaches that could improve on this result.

Eiffel was chosen as the subject of this project because it poses some challenging problems to the language implementor. Among these inheritance, conformance of types, no-definition-before-use, and its unconventional typesystem are the most important ones.

Due to time constraints it was not possible to cover the whole language in the specification. This means that the typechecking of some syntactical constructs is not specified. However it should be straightforward to fill in the blanks.

1.2 Preliminaries

1.2.1 Eiffel

The object oriented programming language Eiffel was designed by Bertrand Meyer in the late 1980’s. A first description of the language appears in [Mey88]. This work is primarily a methodological discussion of the object-oriented paradigm; as a language definition it lacks formality. The book presents the object-oriented method in general (and Eiffel in particular) as a method which enables the software engineer to construct software that is correct, robust, extendible, reusable, and compatible.

The official, fairly formal definition of the syntax, static and dynamic semantics of Eiffel (version 3) is presented in [Mey92]. Except for [Mey92] no publications exist yet, to our knowledge, of specifications or implementations of Eiffel. Indeed only a few compilers for Eiffel exist, although several projects work on implementations.

The syntax and static semantics of Eiffel specified in this paper is, except for some minor details, compatible with the official definition.
Being a mathematical specification of Eiffel, this paper can not and does not want to compete with the informal explanations and examples given in these books; therefore, for a less algebraic clarification of the concepts behind and the constructions of Eiffel we refer the reader to the works mentioned above.

1.2.2 ASF

A specification in ASF (acronym for Algebraic Specification Formalism) consists of a signature and a set of (conditional) equations. The signature specifies a set of sorts, a set of functions over the sorts, and a set of variables over the sorts. The equations identify (open or closed) terms over the signature of the same sort.

Since specifications of non-trivial problems can become quite large, specifications can be divided into modules. The signature of a module is divided into an exports part and a hidden part. To reuse a module it can be imported into a new module.

The model of a specification is an algebra (or family of algebras) that maps each sort in the signature to a set, that maps each function in the signature to a function over these sets, and that satisfies the universal closure of each equation in the specification. Usually the intended model of a specification is its initial algebra. The elements of the sorts of the initial algebra are equivalence classes of terms; two terms being equivalent if they are provably equal with the equations of the specifications as axioms and the derivation rules of conditional equational logic.

Execution of specifications is realized by interpreting the equations as rewrite rules from left to right and applying the resulting term reduction system (TRS) to a term. Termination of the rewrite process can only be guaranteed if the TRS is confluent and weakly-terminating.

ASF is defined in [BHK89b]. The theory of universal algebra, term rewriting, and other theoretical aspects of algebraic specification can be found in (among others) [EM85, Wec92, Wie91]. Implementation of (modular) algebraic specifications and term reduction systems is described in [Wal91, Hen91].

1.2.3 SDF

SDF stands for Syntax Definition Formalism. An SDF specification defines the grammar or syntax of a language. A specification consists of a set of non-terminals or sorts, a set of lexical syntax rules, a set of context-free syntax rules, a set of variable schemas and a set of priority rules.

Lexical syntax rules assign sequences of literals, character classes, sorts, and repetitions of these to sorts. The special sort LAYOUT is used to define the layout of the language.

Context-free syntax rules assign sequences of literals, sorts and list expressions to sorts. List expressions are of the form

\{ SORT "sep"\}+ 

denoting sequences of zero (in case of *) or one (in case of +) or more items of sort SORT separated by sep. (The separator may be left out.)

Variable schemas have the same form as lexical rules and are used to define a set of variables that are distinguished from normal language components in the sentences over the grammar.

The context-free grammar defined by a specification can be interpreted to generate a parser and a syntax directed editor for the language. The result of parsing is an abstract syntax tree; the abstract syntax is derived from the specification.

Since arbitrary context-free grammars can be specified a means to disambiguate the grammar is needed. This is done by a set of priority rules that declares certain constructs to have priority over others. The priority rules are used by considering
them as a partial order on the set of abstract syntax trees defined by the syntax. If more than one tree results after parsing, the ones lower in the order are thrown away. If two or more trees can not be compared the ambiguity is reported.

Specifications can be divided into modules. A module can hide or export syntax rules and can import other modules.

SDF is defined in [HHKR89]. Generation of scanners, parsers and syntax directed editors from specifications and other aspects of the implementation of SDF are described in [Kli91, Rek89a, Rek89b]

1.2.4 ASF+SDF

Together ASF and SDF form a powerful couple for the definition of syntax and semantics of formal (programming) languages.

On the one hand SDF serves in this combination as a front-end for ASF; the abstract syntax derived from the context-free syntax defined in an SDF specification is interpreted as an algebraic signature. On the other hand the SDF part is used to generate a parser and a syntax directed editor for the language specified. The ASF (equations) part gives the semantics of the language.

The ASF+SDF system is a meta-environment for specifying programming environments. One environment enables the programmer to edit, typecheck and debug specifications, and to test the generated programming environments. The meta-environment is described in [Kli90]. Surveys of the syntax directed editor, the debugger, and the user-interface specification language are presented in [Tip91, Koo92a, Koo92b].

1.3 Related Work

Other algebraic specifications of programming languages are described in [Meu88]—a small language with pointers, [Deu91]—static semantics of Pascal, [Ver92]—μCRL: a process algebra specification formalism, and [Lui92]—SQL.

In [Din92] object-oriented programming and particularly inheritance are discussed. Several small languages are specified in ASF+SDF illustrating an alternative to the ‘inheritance implies sub-type’ approach of Eiffel and its problems.

1.4 Outline of the Paper

The next section will survey several aspects of typechecking Eiffel that pose problems to the implementor. Section 3 discusses the tools part of the specification; the library modules that are developed, a mechanism for reusing them, the definition of symbol tables, and a language for specifying modifications of symbol tables. Section 5 outlines the process of typechecking an Eiffel program. Section 7 closes with some conclusions drawn from this project.

The specification is divided over several appendices. The import graphs in appendix A provide a good insight in the structure of the specification. Appendices B through E contain the specifications of the several tools parts of the specification. Appendix F presents type-substitutions and conformance between types. Finally appendix G contains the specification of syntax and static semantics of Eiffel.

2 Checking the Static Semantics of Eiffel

In this section we examine what an Eiffel typechecker should do, what the main problem in specifying it is, and what solutions to this problem there exist.

First we briefly review the structure of Eiffel programs. An Eiffel program (or system) is a set of classes, one of which is designated as the root class. A class
consists of a list of features that may either be fields (called attributes) or routines. A class may inherit another class, which comes down to inheriting its features. Each class denotes a type, that is a set of objects to which the features of the class can be applied. The formal generic parameters of a class are dummy types that can be substituted with any type in the system.

Validity A syntactically correct system is said to be valid if and only if it satisfies all validity rules in [Mey92]. Apart from the rules about type-correctness of expressions and statements (e.g. characters cannot be added to integers) a lot of structural requirements have to be satisfied (e.g. an identifier can only be used once as the name of class). Therefore we will officially call the process of validating a system against these rules validity checking instead of typechecking (although we will sometimes relapse into using the latter). The validity checker we want to design should thus check the system for any violations of the validity rules and, if such a violation occurs, report it by means of an error-message describing the location and type of the violation.

Conformance The central notion in checking for type-correctness of terms is called conformance. Basically a type A conforms to a type B if either they are the same or A is a descendant from B (there is an inheritance path from A to B). The actual definition of conformance is, however, far more complex than this simple rule. The specification of conformance is presented in appendix F.3.

Circularity of Eiffel Programs Although the inheritance graph of an Eiffel program may not contain cycles, mutual dependencies between classes may occur. If a class C refers to another class D by declaring a feature, formal argument or local variable of type D, C is called a client of D. The following sketch of an Eiffel system illustrates the mutual dependency between classes C and D:

```eiffel
class C
  ...  
  feature f: D
  ...  
end class C

class D
  ...  
  feature g: C
  ...  
end class C
```

For validity checking this means we need information from class C to check class D, and the other way around.

At another level we see that features in a class may refer to any other feature of the class; thus two features may refer to each other. Again we need information about the first feature to check the second, and the other way around.

This mutual dependency at two levels entails that we cannot scan the program text from begin to end checking it and storing information in the symbol table for use later on, as can for instance be done with checking Pascal (where forward references must be explicit).

The solution for this problem that is adopted in [Din92] for typechecking the small languages defined there is to use the whole program as symbol table; at any moment during typechecking all information is available. This solution is not viable for an Eiffel validity checker though for the following reasons:

- The program might not be statically correct (otherwise validity checking would make no sense). This means that the information it contains may be inconsistent.
- In [Din92] most checks are of type-correctness; violations against structural constraints are ignored. For instance multiply defined identifiers, are handled
by throwing away all but the first definition of the identifier. Since we want to report all validity violations such a policy is not possible in our case.

- Eiffel has a very complex syntax that would require many lookup and projection functions.

Other compelling reasons will be given in section 3.3.

Since we reject this solution, we will use a symbol table as storage for the information retrieved from the program. But, as we already saw, it is then not possible to do all checks in one traversal of the program. The solution we adopt for this problem is to traverse the program several times, each time checking at a deeper level using the information retrieved in earlier passes.

3 Overview of the Specification—The Tools

3.1 Abstract Datatype Modules

The first part of the specification is devoted to the definition of the symbol table. As remarked in the introduction we want to maximize the reusability of the specification. Therefore we do not build up the symbol table from scratch, but use 'standard' modules which specify well known abstract datatypes. Other reasons for using a limited notation, derived from a few well written modules for the tools, are improving readability (since less syntax with less unusual semantics has to be learned), and limiting difficult and time consuming language design to a minimum.

The standard modules, presented in appendices B and C, will not bring many surprises. We will discuss the modules here.

Layout Module Layout specifies the layout of specification modules. Apart from the oneline comment `%` also multiline (C) comment (between /* and */) is defined.

Booleans The inevitable booleans.

Integers Addition, subtraction, multiplication, comparison (including equality `==`) of integers. The equations are rather complicated and need not be read for a good understanding of the rest of the specification. (This module is a slight modification of the ASF+SDF library module.)

Strings Sequences of characters enclosed in double quotes (`"`). Operations are concatenation (`++`) and equality (`==`) of strings.

Sets of ITEMS. Operations are union (`+`), difference (`-`), intersection (`&`), membership (`in`), subset (`in`), emptiness (`empty?`), cardinality (`\|\|`), and equality (`==`). The operations `first` and `rest` are enumeration operations that divide a set in some element and the set without that element.

Tuples Module Tuples2 defines syntax and operations for a tuple consisting of two elements. In principle for every\( ^{1} n \geq 2 \) a module Tuples\( n \) exists. A term of sort TUPLE\( n \) is a fixed sequence of \( n \) fields. Each field has a name.

The field names (which are not part of a tuple) are used as indices into the tuple. The value of a field can be projected out of a tuple by applying the fieldname to the tuple as in

\[ \text{Coordinate.x-coord} \]

\( ^{1} \)A Tuples\( n \) module can be generated by a Pascal program for every \( n \geq 2 \).
The value of a field can be modified, yielding a new tuple, by applying a modification function to the tuple as in

\[ \text{Coordinate:} x\text{-coord} := 0 \]

(this expression yields a coordinate that is the same as Coordinate except for the \( x\)-coord field which is set to 0. In this way a field of a tuple may be retrieved or modified without having to name all other fields of the tuple, as one would have to do with pattern matching. Furthermore the place of the field in the tuple does not matter if the tuple is only accessed by means of the projection and modification functions.

A ‘modification recipe’ for a tuple can be expressed without mentioning the tuple to which it is applied. The expression

\[ x\text{-coord} := 0 \]

denotes a function (called a TUPLE-UPDATE) that can be postfixed to an arbitrary tuple of the appropriate sort, as in the example above, to modify it.

If Coord-pair is a pair of coordinates as above with fields first-coord and second-coord then we can ‘stack’ the projection functions: the expression

\[ \text{Coord-pair:} \text{first-pair:} y\text{-coord} \]

denotes the \( y\) coordinate of the first pair of Coord-pair. For modifications that does not hold; if we want to set the \( y\) coordinate of the first pair of Coord-pair to 0 we have to write

\[ \text{Coord-pair:} \text{first-pair} := (\text{Coord-pair:} \text{first-pair:} y\text{-coord} := 0) \]

To be able to write this as

\[ \text{Coord-pair:} \text{first-pair:} y\text{-coord} := 0 \]

some extra syntax rules and equations have to be added.

**Sequences** Module Sequences provides operations on sequences of ITEMS. The syntax and the names of the operations were inspired by the functional programming language Miranda ([BW88]), which provides a large set of elegant operations on lists. The set of functions could be improved and extended significantly if ASF+SDF would support polymorphism and higher-order functions.

**Tables** Module Tables introduces sequences of ITEMS which have a ‘key’ attribute; to the items in a table a function .key (postfix) can be applied which yields some value of sort KEY.

Typical of tables is that they are sets with regard to the keys of the items; no two items with the same key should be in a table. Therefore keys should be comparable by a boolean equality function ==. In order to preserve this ‘table’ property insertion functions (\( \rightarrow \) and \( \leftarrow \)) are to be used to add new items to a table. A table may be asked if it contains an item with a certain key (\( ?\text{KEY} \)) and if so what that item is (\( .\text{KEY} \)). Finally a key may be deleted from a table (\( -\)).

An application of this module is for instance an identifier-type-pair table. The key function .id would be a projection function on these pairs which gives the identifier part of the pair.

**Graphs** Module Graphs represents graphs as a table of pairs of nodes and the set of their neighbour nodes. Operations are mainly union (+) of graphs, and transitive closure of a graph (\text{trans}).
3.2 Reusing Modules

In pure ASF (without the SDF part) a parameter mechanism is available that can be used to obtain several instances of a module by filling in parameters (like the \texttt{ITEM}s in the modules discussed above) and by renaming functions and sorts. In ASF+SDF this parameter mechanism is not available.

Therefore some other mechanism or method, that enables the reuse of the ADT modules that were discussed in the previous section, is needed. An obvious but naive (because very laborious) approach is to make copies of a module and ‘manually’ rename sorts and variables. If only a few copies are necessary this method will work, but when the original module is changed all copies have to be changed as well.

A slightly less naive method automatizes the naive method; the textual substitution is done by a program. Although the method has many disadvantages it has one great advantage: ease of implementation. In section 6 some other methods that appear in discussions are mentioned.

3.2.1 The Lambda and Merge Operators

In the lambda calculus (see [Bar85]) the $\lambda$ operator in

$$\lambda x. M$$

transforms the expression $M$ into a function with one argument that, when applied to another expression, $N$ say, yields the expression $M$ with all occurrences of $x$ in $M$ replaced by $N$; in other words

$$(\lambda x. M) N \equiv M[x := N].$$

This is precisely the kind of operation described above. So we define an operator \texttt{lambda} that takes one or more arguments and abstracts those from the module it is applied to. When the resulting function is applied to the same number of arguments as were abstracted, the actual arguments are substituted for the formal arguments in the module, yielding a renamed instance of the module.

As in the lambda calculus (textual) substitution has several drawbacks. Furthermore we can not guarantee that the resulting module is syntactically correct even if the original one was. Therefore we define a lambda application to be statically correct if the resulting module is, and we define the semantics of a lambda application to be the semantics of the resulting module.

As an example consider module \texttt{Sets} discussed in the previous section. If we want a module \texttt{Integer-Set}s, that defines the sort \texttt{INT-SET} (set of integers), we first abstract out items such as \texttt{SET}, and \texttt{ITEM}, and then fill in \texttt{INT-SET} and \texttt{INT}.

In addition to the \texttt{lambda} operator we define the + operator, which merges two or more modules into one. This operator is useful in making modules that are combinations of lambda instantiations of other modules.

In appendix D.1 the syntax and a sketch of the semantics of the \texttt{lambda} and + operators is presented in module \texttt{Abstraction}. The semantics is sketched because it is not (yet) possible to manipulate ASF+SDF modules in ASF+SDF specifications.

3.2.2 Abbreviations

The lambda expressions that are needed to reuse the standard modules do not give much insight into the module they declare. Therefore some syntactic sugar is introduced in module \texttt{Abbreviations} (appendix D.1).

With these constructs we can for example declare the coordinates we saw earlier as
module Coordinates
COORDINATE = tuple of
    x-coord : INT var _Int from Integers
    y-coord : INT var _Int from Integers
variables Coordinate : COORDINATE
end module Coordinates

3.2.3 Implementation

The \texttt{lambda} and + operator have been implemented for practical use on a SUN under the Unix operating system using \texttt{sed} in a series of shell scripts and makefiles.

The script \texttt{lambda} takes a number of strings, the first of which must be the name of the module to be abstracted, it then translates the module to a shell script which has an argument for each of the abstracted strings. When applied to the right number of arguments this script produces an instance of the original module.

The \texttt{merge} script takes the names of the modules to merge and the name of the resulting module. It then merges the modules section by section into a new module. For this purpose modules must be spread over files, one for each section (e.g. exported sorts, exported context-free syntax, equations).

A set of abbreviations can be translated to a makefile (instead to module expressions) with entries for all declared modules. The dependency rules in the makefile call the \texttt{lambda} and/or \texttt{merge} operator with the appropriate arguments.

3.2.4 What Parameter Mechanism to Choose?

There is an apparent need for a parameter mechanism or, in general, a mechanism for ASF+SDF that enables reuse of modules in a more expressive way than just plain import. In recent discussions several possibilities have been proposed. We will discuss some of them here.

The mechanism presented in the previous section can not be considered as a serious option, because the operators work on the text of modules instead of on their structural components. A better way to define them would be as operators on the algebra of modules. At a theoretical level such operations on modules are described in [BHK90]. The module algebra equivalents of the \texttt{lambda} and + operators are the \texttt{renaming (.)} and \texttt{union (+)} operators.

A possibility that can coexist with other solutions is to allow arbitrary functional types for functions, that is higher-order functions instead of just first order functions as in the current formalism. The expressive strength of higher-order functions is maximized by allowing type-variables, that stand for any type, in function types. In the presence of higher-order functions we can define such functions as \texttt{map} that applies an arbitrary function to all elements in a list and yields the list of the results. Contemplations about the implementation (in $\lambda$-Prolog) of such an extension of algebraic specification appear in [Hee91].

Another option is to let the specifier specify his own operations on modules, and thus his own parameter mechanism or equivalent ([Kil92]). The problem of this so-called meta-specification with ASF+SDF is the free syntax of the modules to be manipulated.

3.3 The Symbol Table or Context

Now that we have a set of modules specifying some abstract data types and a method to make instantiations of these modules, we are able to define a symbol table for our validity checker. Since the symbol table represents the context of some program fragment when it is checked, the symbol table will be called \texttt{CONTEXT}. 

9
Before going into the details of the definition we will first give some reasons why it is a good idea to define the symbol table using these abstract datatypes (and not using the constructs of the language).

- The definition of the symbol table can be made independently from the definition of the syntax of the language. Only primitive sorts like identifiers and constants are needed in the symbol table.

- Defining the symbol table by using standard abstract datatype modules ensures that the operations on the symbol table have a uniform notation and functionality. This will improve readability of the specification. Furthermore since we have defined a table lookup function in our Tables module we do not have to write a lookup functions for lists of variable type in a list of variable declarations, for features in a feature list and for classes in a class list.

- Using the program as symbol table leads to the definition of many lookup and projection functions on the syntax of the language; a lot of non-reusable work and non-uniform notation. Changes in the syntax of the language entail changes of the lookup functions.

- The symbol table can be defined such that it contains all information needed by a second (e.g. compilation) pass; another traverse of the program is avoided.

### 3.3.1 Components of the Context

Let us examine the components a context should have. First of all a context should contain a table of classes (indexed by class name) which contains information about all the classes in the system. Then, since our goal is to report all validity rule violations in the system, a list of errors should be part of the context. In section 2 we decided that the validity checker makes several passes over the program text so a context should record the current pass. A class, inheritance clause or feature is examined, and information about it is gathered in each pass. Because we want new information about an item to complement the old information that already existed about it, some mechanism is needed to update an item's entry in the context. For this purpose the context records the ‘current focus’, so all modifications to the context can be done relative to this focus. Finally the context should record types of, e.g. subexpressions, for typechecking expressions. This is achieved by a stack of types in the context.

Table 1, shows the structure of a CONTEXT tuple, which reflects the requirements described above. Tables 2, 3, 4, and 5 show the structures of respectively SYSTEM-CONTEXTS, CLASS-CONTEXTS, PARENT-CONTEXTS, and FEATURE-CONTEXTS. To stack modifications of CONTEXTs, as described in section 3.1, extra syntax and equations have been added to module Contexts.

Here we have examined only the top structure of contexts; the full declaration of the modules that define contexts are given in section E.1.

### 3.3.2 Examples of Usage

To give some idea of the operations possible on contexts, at this point, some examples are discussed below. In these examples let \( c \) be a variable denoting a CONTEXT. The examples in this list all make a modification to context \( c \). The result of the expressions are contexts that are the same as context \( c \) except for the modifications.
Field | Sort | Meaning
---|---|---
cs | SYSTEM-CONTEXT | Current System
cc | CLASS-CONTEXT | Current Class
cp | PARENT-CONTEXT | Current Parent
cf | FEATURE-CONTEXT | Current Feature
c-clients | ID-SET | Current Clients
type-stack | TYPE-SEQ | Current Pass
pass | PASS | Current Pass

Table 1: Fields of CONTEXT.

Field | Sort
---|---
sys-name | ID
root-name | ID
class-table | CLASS-TABLE
errors | ERRORS
params | SYSTEM-PARAMETERS

Table 2: Fields of SYSTEM-CONTEXT.

Field | Sort
---|---
class-name | ID
formal-generics | TYPE-PAIR-TABLE
parents | PARENT-TABLE
class-features | FEATURE-TABLE
creators | ID-SET
is-deferred | BOOL
is-expanded | BOOL
deferred-check | BOOL

Table 3: Fields of CLASS-CONTEXT.

Field | Sort
---|---
parent-name | ID
parent-features | FEATURE-TABLE

Table 4: Fields of PARENT-CONTEXT.
<table>
<thead>
<tr>
<th>Field</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature-name</td>
<td>ID</td>
</tr>
<tr>
<td>signature</td>
<td>SIGNATURE</td>
</tr>
<tr>
<td>clients</td>
<td>ID-SET</td>
</tr>
<tr>
<td>formals</td>
<td>ID-TYPE-TABLE</td>
</tr>
<tr>
<td>locals</td>
<td>ID-TYPE-TABLE</td>
</tr>
<tr>
<td>is-creator</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-deferred</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-internal</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-frozen</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-attribute</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-var-attribute</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-const-attribute</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-unique</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-routine</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-procedure</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-function</td>
<td>BOOL</td>
</tr>
<tr>
<td>is-once</td>
<td>BOOL</td>
</tr>
</tbody>
</table>

Table 5: Fields of FEATURE-CONTEXT.

Note that these expressions can be written without the C: prefix and then represent ‘modification recipes’ that can be applied to an arbitrary context.

- C:type-stack.push(class-type(INTEGER,[[]]))
  Push type class-type(INTEGER,[[]]) on the stack.
- C:cf.signature := ([class-type(INTEGER,[[]]),[]])
  Set the signature of the current feature to ([class-type(INTEGER,[[]]),[]]).
- C:cf.locals := C.cf.locals < (X, formal(Y))
  Add local variable X with type formal(Y) to the table of local variables (also called local entities).
- C:cs.errors := C.cs.errors ++ X:validity(VXXX,999,1)
  Add the error message ‘validity rule VXXX item 1 described at page 999 of [Mey92] is violated near identifier X’ to the list of error-messages.
- C:cc := C.cs.class-table.LIST
  Put the information in the class table associated with the class LIST into the current class focus.

Apart from modifying a context we can also query the context. Here are some examples of queries:

- C.type-stack.top
  The type on top of the stack.
- C.cf.signature.args
  The sequence of types of the formal arguments of the current feature.
- hd C.cs.class-table.LIST.feature-table.HEAD.signature.res
  The result type of feature HEAD in class LIST.
3.4 A Context Modification Language

Now we have at our disposal a symbol table with operations to modify and query it. To express more complex modifications on contexts we define a *Context Modification Language* (CML) which provides some conventional constructs like sequential composition and if-then-else.

### Syntax of CML

The atomic components of the language are the context modifications (CONTEXT-UPDATEs) as discussed in the previous section. To these the following are added:

- **error ERROR**—add ERROR to the list of errors;
- **require BOOL else ERROR**—BOOL must be true, if not add ERROR to the error-list;
- **enter-class(ID)**—the current focus is class ID;
- **leave-class**—save the information in the cc field in the class-table;
- similar pairs for parent and feature.

Combinations of CONTEXT-UPDATEs can be formed by sequential composition (;), and two flavours of conditional choice (if-then-else and case). Finally a CONTEXT-UPDATE can be applied to a context by the with-do construct.

Figure 1 shows an example of an expression in CML (the numbers are line numbers and do not belong to the expression). The @'s in the queries of the example are ‘dumm y contexts’ or ‘pointers to the current context’; @ is just a syntactical reference to the context. Note that as each modification is applied to the context the current context changes; the @ in line (3) has a different reference than the @ in line (13).

### Semantics of CML

The semantics of CML brings no surprises. For instance the sequential composition of CE1;CE2 applied to a context will apply CE1 to the context and will then CE2 to the resulting context.

The only problem in the definition of the semantics is that occurrences of @ in the queries in atomic modifications have to be replaced by the current context. For this purpose an operator in CONTEXT is defined on each sort that appears in contexts. So there are functions

- CONTEXT in CONTEXT -> CONTEXT
- BOOL in CONTEXT -> BOOL
- INT in CONTEXT -> INT
- E-TYPE in CONTEXT -> E-TYPE

and so on. If the operator reaches the @ it is replaced, so if Context is a context
with C do
begin
  cf.signature.res := [0.type-stack.top];
  type-stack.pop;
  case
    @.cf.locals?_id
    error _id:validity(VYYY,999,1)
    @.cf.formals?_id
    error _id:validity(VYYY,999,2)
  else
    cf.locals := cf.locals <- (_id, @.type-stack.top)
  end;
require ~ @.cc.is-deferred
  else validity(VXYZ, 993)
end

Figure 1: An example CML expression.

@ in Context = Context

But we also need an interpretation in context for the boolean expression

\( b \land b' \)

where \( b \) and \( b' \) are terms of sort bool, because \( \@ \) may be hidden in either \( b \) or \( b' \).

So we add an equation

\((b \land b') \text{ in Context} = (b \text{ in Context}) \land (b' \text{ in Context})\)

This results in an equation for each function in each module that is part of the definition of contexts. The equations are just like the last example; the operator \( \text{ in Context} \) distributes over all functions. In general if \( f \) is a function and \( s \) and \( s' \) are variables of the appropriate sorts then

\( f(s,s') \text{ in Context} = f(s \text{ in Context}, s' \text{ in Context})\)

The only exception to this rule is the interpretation of \( \text{CONTEXT-UPDATE} \); their queries are interpreted in the context and the resulting \( \text{CONTEXT-UPDATE} \) is applied to the context. For instance, if \( _\text{SC} \) denotes a \text{SYSTEM-CONTEXT},

\((cs := _\text{SC}) \text{ in Context} = \text{Context}:cs := (_\text{SC in Context})\)

The syntax and semantics of the \( \text{in CONTEXT} \) operator are defined in module \text{In-Context} which is declared in appendix E.2.

3.4.3 The Context as Global Environment

The example in Figure 1 showed a complex context-update applied to a context. If we leave out the \textbf{with C do} line a context-update results that can be applied to any context. With this observation in mind we can now specify typecheck functions as context-updates without referring to the context it will be applied to.

Figure 2 shows an example equation written in this manner. It shows the specification of the function that checks the identifiers in a local variable declaration list. (For instance the identifiers in the declaration

\[ a, b, c : \text{TYPE} \]
that appears in some routine.) After the rest of the list is checked and some requirements have been satisfied the identifier with its type (which is on top of the type stack) is added to the table of local variables of the current feature.

Nowhere in this example we see an occurrence of the context, only references to it are made by the '@'s. We can thus consider the context as a global environment (as in Pascal, the set of all global variables) on which the typecheck 'procedures' operate. After typechecking error messages and information for compilation can be derived from the global environment.

The ultimate consequence of this is that nowhere in the specification an 'initial' context has to be defined. The userinterface just provides a (consistent) context, that lives in some term editor, and which may be empty or may be the result of typechecking a 'default' system that contains classes like INTEGER, ANY, and PLATFORM.

### 3.4.4 Reusability of the Contexts Specification

The basic components of CML are the abstract datatype modules, their in-Context modules, and the specification of CML itself. A symbol-table manipulation language for the typechecking of any language can be specified by only declaring the symbol-table by means of the abbreviations and perhaps some minor changes to the module CML. Our conclusion is that the method is highly reusable in other specifications.

In a more general fashion the approach of a symbol-table or database as global environment is interesting for many applications.

### 4 Overview of the Specification—Operations on Types

On top of CML several operations on types are defined in support of the typechecking modules.

#### 4.1 Type Substitutions

The operator subst TYPE-PAIR-TABLE postfixed to a type or other structures containing types, substitutes actual generic parameters for formal generic parameters. For instance, if class LIST has formal parameter X and if feature HEAD of class LIST has result type X then the result type of HEAD in the actualization LIST[INTEGER] is
4.2 Bitsize of Types

Function \texttt{bit-size} calculates the number of bits an object of some type will occupy. The size of an object is defined as the sum of the sizes of the attributes of its class. The bottomline of this recursion are the 'primitive' classes like \texttt{INTEGER} and \texttt{BOOLEAN}. The sizes of these classes are implementation dependent and are known to the function by parameter fields of the \texttt{cs} field of contexts. For instance,

\begin{verbatim}
0.cs.params.int-size
\end{verbatim}

should contain the number of bits by which an integer is represented.

4.3 Conformance

Conformance of types, type-sequences and (feature-)signatures by the \texttt{<} operator is specified. The specification is rather complex as a result of the many exceptions.

5 Overview of the Specification—The Validity Checker

The validity checker as defined by the specification is the function

\begin{verbatim}
vc-system "[" SYSTEM "]" -> CONTEXT-UPDATE
\end{verbatim}

that generates from a system a CML expression. The application of that CML expression to a (possibly empty) context yields a symbol table representing the system, including a list of errors.

The syntax of Eiffel is divided in several modules, each representing a logical unit. Along with each syntax module a validity module is specified. The specification presents the syntax modules with their associated validity modules in bottom-up order. In addition to these modules equality modules for identifiers and feature-names, defining case conversion (\texttt{lower-name} and \texttt{upper-name}) and equality (\texttt{==}) of identifiers, and normalization of feature-names, are specified along with the syntax modules.

For a good understanding of the specification a combination of top-down and bottom-up reading is advised.

5.1 Outline of the Validity Checking Process

Function \texttt{vc-system} first checks if there are no cycles in the inheritance graph by deriving the inheritance graph and the transitive closure thereof. (If the transitive closure is irreflexive there are no cycles in the graph.) Then the classes in the system are sorted by the partial order on classes that the transitive closure of the inheritance graph induces.

The validity checker then makes several passes over the sorted classes:

\textbf{class-names} The names of all classes in the system are gathered.

\textbf{formal-generics1} For each class the number and names of formal generic parameters are collected. (We can not do this in the previous pass since class names may not be used as formal generic parameters.)

\textbf{formal-generics2} The constraints of the formal generic parameters of each class are checked and stored.
The signatures of inherited features are included in the feature-table of a class, signatures of new and redeclared features are checked and added to the table and the creators clause is checked.

Feature-bodies Feature-bodies, assertions and class-invariant are checked.

5.2 Conformance to the Definition

The syntax in [Mey92] is not translated literally to SDF for the following reasons:

- Some constructs used in the BNF notation in [Mey92] do not exist in SDF (e.g. optional parts in productions).
- In a few cases checking can be shifted from the validity checker to the parser; the requirements of some validity rules can be expressed syntactically.
- Some parts in the BNF, such as all different flavours of identifiers that are really the same, are redundant.
- Some constructs, such as expressions, can be expressed more elegantly using SDF's context-free syntax and priority rules.
- The only real deviation from [Mey92] is that case-insensitiveness can not be expressed easily in SDF. Therefore only the lowercase variants of keywords are recognized as keywords. Identifiers are all translated to their upper-case variant.

Due to time constraints not all details of validity checking are specified. When some validity rule is not checked an error message like

```
error "WARNING: ... not checked"
```

is generated.

The most important omissions are:

System Level Validity

Multibranche Statement

Repeated inheritance

Binary operators (detail: INTEGER + REAL should be translated to REAL.binary(+)(INTEGER) instead of to INTEGER.binary(+)(REAL))

5.3 Future Work

Here is a list of ideas for improvements and extensions of the specification:

- Fill in the gaps mentioned above.
- Currently the specification is annotated at some places with the validity rules that are checked. These texts can be used to provide more verbose error messages; an error message could be expanded by a help function to the text of the validity rule they are based on. The definitions of these expansions can be placed in the module where they are checked at the same time providing a documentation to the specification.
- Store errors locally. An error in some feature should be listed in the error field of the corresponding FEATURE-CONTEXT. This could help in specifying a userinterface that shows the source of an error.
• On top of the division of a program in classes [Mey92] divides families of classes into clusters and these into universes. Furthermore in a real system classes would be on separate files in the filesystem.

• Can the CONTEXT sort be used to represent other object-oriented programming languages? More generally, can the CML approach be used for other languages?

• Make the Context Modification Language more uniform. One can think of complex modifications of all structures and dummies like @ that point to other sorts than CONTEXT.

• Single out constructs and static semantic rules that hold for many programming languages as a basis for a set of language building blocks.

• Make the specification incremental. After a system is found to be valid by the validity checker new classes may be added. Then only these classes should be checked using the information from the previous check.

How much must the specification be altered to be able to deal with local changes in the program (and not having to check the whole program again)?

• Specify short and flat tools that respectively give the 'interface' of a class and the normalized class (inheritance resolved by including the inherited features). Other tools like a class browser could be specified.

• Specification of dynamic semantics (interpreted or translational) based on the context delivered by the validity checker. (The (annotated) bodies of routines, pre- and postconditions, and class invariants should be stored in the context for this purpose.)

6 Discussion

The following is a list of remarks, ideas, and questions generated by the project.

• We could have specified the symbol table more compactly; instead of making explicit algebraic sorts for all components of the the symbol-table we could have made one polymorphic sort ITEM. Items may be sets of items, sequences of items, tables of items and so on. Then we could define tuples as being tables with an identifier key field; records with a variable number of arguments.

The danger of this approach is that the datastructure is not fixed, but is constructed at run-time, possibly leading to inconsistent usage of the datastructure; In the current approach inconsistent usage is detected by the parser.

• Instead of several passes over all classes it would be nice to typecheck all classes in parallel. The separate processes would communicate the type information to each other. A process will wait with typechecking until the information it needs is available. (Although it should still provide all information it already has.) The specification of vc-classes would then look something like

\[
\text{vc-classes [ Class1 \ldots Classn ]} \\
= \text{vc-class [ Class1 ] // ... // vc-class [ Classn ]}
\]

where we write // for the merge operator from Proces Algebra.

Earlier we declined the use of the program as symbol table since we would have to typecheck the symbol table before we could use it. It might be interesting though to consider the following scheme:
where tc yields a symbol table with errors and typechecks a program until the symbol table it gets does not provide enough information. In this way the passes that are described explicitly in this paper are made implicit.

- The approach using CML and global variables leads to an imperative style of specification. A more declarative way of describing the validity constraints would be better.

In [Deu91] a language for declaring the static semantics of a language is considered. Such a language would allow the declaration of context-sensitive constraints along with the declaration of the context-free grammar rules.

A way to make this specification more declarative would be done as follows. All conditions should be expressed by the require construct (no more error recovery). The specification of require should be changed, such that if the condition is not met, require should not reduce and should list the query on the context as well as the validity rule reference.

- How composable is the specification? Or otherwise put, can we reuse the specification of parts of the language to build a new language (including specification of validity)?

- Recurring question: Design new language or use abstraction mechanisms in current language to achieve reuse?

- In the specification a 'meta-equality' operator (==) is defined on many sorts that is used to compare terms in the conditional CONTEXT-UPDATE if and case. Considering the fact that an algebraic specification already specifies equality of terms, it comes to mind that this algebraic equality could be used in user defined functions (yielding a boolean value), and not only in the conditions of conditional equations.

- Currently the system is not capable to handle large specifications like the one in this paper. A better mechanism for reusing modules (instead of making renamed copies) would already make a big difference.

- The syntax definition part of the formalism is superior to lexical analyzer and parser specification languages such as Lex and Yacc.

- ASF+SDF now provides a formalism and toolbox that relieves the language designer of the burden of many low level implementation details of programming environments (like lexical analyzer, parser and user-interface). The next step is to relieve him of the low level details of type and value environments.

- To improve readability of specifications the tools part should introduce no more new notation then necessary. And when it does the notation should be as uniform as possible.

- The in Context operator is similar to renaming or substitution operators; it distributes over all function symbols except for a few cases where it changes or substitutes something.

7 Conclusions

In this paper we developed
• a set of general purpose abstract data type modules which can be reused in many specifications,
• an approach to reuse modules by a copy and rename policy,
• a method for compact specification of symbol-tables,
• a language (CML) to describe complex symbol-table (context) modification procedures that see the context as a global environment, and finally
• a specification of the syntax and static semantics (validity) of Eiffel systems.

References


[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In Algebraic Specification [BHK89a], chapter 1, pages 1–66.


[Rek89a] J. Rekers. An implementation of SDF. In Bergstra et al. [BHK89a], chapter 8, pages 339–358.


A Import Graphs

The graphs on this page and the next pages an overview of the modules in the next appendices. An arrow denotes the import of a module (tail) in another module (head).

---

Figure 3: Import Graph for Constant-Syntax

---

Figure 4: Import Graph for Routine-Syntax
Figure 5: Import Graph for Feature-Syntax

Figure 6: Import Graph for Class-Syntax

Figure 7: Import Graph for System-Syntax
Figure 8: Import Graph for Contexts, CML, Type-Substitutions and Conformance
B Library Modules

The modules defined in this appendix are (semi-)standard modules specifying, respectively, the layout of specification modules, booleans, and integers that need no further explanation.

B.1 Layout

module Layout
exports
lexical syntax
[ \t\n] -> LAYOUT
"%%" "[\n]* -> LAYOUT
"*/" "[*]/" "*/" -> LAYOUT
end module Layout

B.2 Booleans

module Booleans
imports Layout
exports
sorts BOOL
context-free syntax
true -> BOOL
false -> BOOL
"(" BOOL ")" -> BOOL \{brack\}
"~" BOOL -> BOOL
BOOL "+" BOOL -> BOOL \{left\}
BOOL "+" BOOL -> BOOL \{right\}
BOOL "<=" BOOL -> BOOL \{non-assoc\}
BOOL "/=" BOOL -> BOOL \{non-assoc\}
variables
[Bb]ool[0-9]* -> BOOL
"_"[Bb]ool[0-9]* -> BOOL
priorities
{"~" BOOL -> BOOL

CML/Conformance

Imported-1
Validiy

Imported-n
Validiy

Unit
Syntax

Unit
Validiy

Figure 9: Import Graph for Validity (Generic)
> {BOOL "&" BOOL -> BOOL}
> {BOOL "|" BOOL -> BOOL}
> {BOOL "=>" BOOL -> BOOL}
> {BOOL "<=>" BOOL -> BOOL}
equations
[00] " " bool = bool
[00] " true = false
[01] " false = true

[02] true & bool = bool
[03] false & bool = false
[04] bool & true = bool
[05] bool & false = false

[06] true | bool = true
[07] false | bool = bool
[08] bool | true = true
[09] bool | false = bool

[10] true => bool = bool
[12] bool => true = true

[13] true <=> true = true
[14] false <=> false = true
[15] true <=> false = false
[16] false <=> true = false

[16] bool1 == bool2 = bool1 <=> bool2
[17] bool1 /= bool2 = ~(bool1 <=> bool2)
end module Booleans

B.3 Integers

module Integers
imports Layout Booleans
exports
sorts NAT-CON NAT INT
lexical syntax
[0-9]+ -> NAT-CON
context-free syntax
NAT-CON -> NAT
NAT "-" NAT -> NAT {left}
"(" NAT ")" -> NAT {bracket}
NAT -> INT
"+" NAT -> INT
"-" NAT -> INT
INT "*" INT -> INT {left}
INT "-" INT -> INT {left}
INT "/" INT -> INT {left}
INT ">" INT -> BOOL
INT ">=" INT -> BOOL
INT "<" INT -> BOOL
INT "<=" INT -> BOOL
INT "==" INT -> BOOL
"(" INT ")" -> INT {bracket}
variables
"_Nat-con"[0-9]* -> NAT-CON
"_Nat"[0-9]* -> NAT
"_Int"[0-9]* -> INT
Char"+"[0-9]* -> CHAR+
Char[0-9]* -> CHAR

hiddens
context-free syntax
  gt "(" NAT "," NAT ")" -> BOOL
  NAT "/-//" NAT -> NAT
variables
  n [0-9]* -> NAT-CON
  c [0-9]* -> CHAR
  x [0-9]* -> CHAR*
  y [0-9]* -> CHAR+
  Nat-con [0-9]* -> NAT-CON
  Nat [0-9]* -> NAT
  Int [0-9]* -> INT

priorities
  {left: INT "*" INT -> INT, INT "-" INT -> INT}
  < {INT "*" INT -> INT}

equations
  [4] nat-con("0") y = nat-con(y)

% -- addition --
  [2] 0 + Nat = Nat
  [3]  Nat + 0 = Nat
  [16] 2 + 4 = 6 [17] 2 + 5 = 7 [18] 2 + 6 = 8
  [22] 3 + 1 = 4 [23] 3 + 2 = 5 [24] 3 + 3 = 6
  [25] 3 + 4 = 7 [26] 3 + 5 = 8 [27] 3 + 6 = 9
  [31] 4 + 1 = 5 [32] 4 + 2 = 6 [33] 4 + 3 = 7
  [34] 4 + 4 = 8 [35] 4 + 5 = 9 [36] 4 + 6 = 10
  [37] 4 + 7 = 11 [38] 4 + 8 = 12 [39] 4 + 9 = 13
  [40] 5 + 1 = 6 [41] 5 + 2 = 7 [42] 5 + 3 = 8
  [43] 5 + 4 = 9 [44] 5 + 5 = 10 [45] 5 + 6 = 11
  [49] 6 + 1 = 7 [50] 6 + 2 = 8 [51] 6 + 3 = 9
  [52] 6 + 4 = 10 [53] 6 + 5 = 11 [54] 6 + 6 = 12
  [55] 6 + 7 = 13 [56] 6 + 8 = 14 [57] 6 + 9 = 15
  [58] 7 + 1 = 8 [59] 7 + 2 = 9 [60] 7 + 3 = 10
  [64] 7 + 7 = 14 [65] 7 + 8 = 15 [66] 7 + 9 = 16
  [67] 8 + 1 = 9 [68] 8 + 2 = 10 [69] 8 + 3 = 11
  [70] 8 + 4 = 12 [71] 8 + 5 = 13 [72] 8 + 6 = 14
  [73] 8 + 7 = 15 [74] 8 + 8 = 16 [75] 8 + 9 = 17
  [76] 9 + 1 = 10 [77] 9 + 2 = 11 [78] 9 + 3 = 12
  [79] 9 + 4 = 13 [80] 9 + 5 = 14 [81] 9 + 6 = 15
  [82] 9 + 7 = 16 [83] 9 + 8 = 17 [84] 9 + 9 = 18

[85a] nat-con(c1) + nat-con(c2) = nat-con(x c),
nat-con (y1) + nat-con(y2) + nat-con("0" x) = nat-con(y)

====================================================================
nat-con(y1 c1) + nat-con(y2 c2) = nat-con(y c)

[85b] nat-con(c1) + nat-con(c2) = nat-con(x c),
nat-con("0" x1) + nat-con(y2) + nat-con("0" x) = nat-con(y)
====================================================================
nat-con(x1 c1) + nat-con(y2 c2) = nat-con(y c)

[85c] nat-con(c1) + nat-con(c2) = nat-con(x c),
nat-con (y1) + nat-con("0" x2) + nat-con("0" x) = nat-con(y)
====================================================================
nat-con(y1 c1) + nat-con(x2 c2) = nat-con(y c)

%%% -- auxiliary function --// for cut off subtraction

[86] nat-con(c) --// nat-con(c) = 0
[87] nat-con(c) --// 0 = nat-con(c)
[88] 2 --// 1 = 1
[89] 3 --// 1 = 2 [90] 3 --// 2 = 1
[91] 4 --// 1 = 3 [92] 4 --// 2 = 2 [93] 4 --// 3 = 1
[94] 5 --// 1 = 4 [95] 5 --// 2 = 3 [96] 5 --// 3 = 2
[97] 5 --// 4 = 1
[98] 6 --// 1 = 5 [99] 6 --// 2 = 4 [100] 6 --// 3 = 3
[101] 6 --// 4 = 2 [102] 6 --// 5 = 1
[103] 7 --// 1 = 6 [104] 7 --// 2 = 5 [105] 7 --// 3 = 4
[106] 7 --// 4 = 3 [107] 7 --// 5 = 2 [108] 7 --// 6 = 1
[109] 8 --// 1 = 7 [110] 8 --// 2 = 6 [111] 8 --// 3 = 5
[112] 8 --// 4 = 4 [113] 8 --// 5 = 3 [114] 8 --// 6 = 2
[115] 8 --// 7 = 1
[116] 9 --// 1 = 8 [117] 9 --// 2 = 7 [118] 9 --// 3 = 6
[119] 9 --// 4 = 5 [120] 9 --// 5 = 4 [121] 9 --// 6 = 3
[122] 9 --// 7 = 2 [123] 9 --// 8 = 1
[124] 10 --// 1 = 9 [125] 10 --// 2 = 8 [126] 10 --// 3 = 7
[127] 10 --// 4 = 6 [128] 10 --// 5 = 5 [129] 10 --// 6 = 4
[130] 10 --// 7 = 3 [131] 10 --// 8 = 2 [132] 10 --// 9 = 1

%%% -- auxiliary function gt

[133] nat-con(c1) --// nat-con(c2) = nat-con(c), nat-con(c) != 0
====================================================================

gt(nat-con(c1), nat-con(c2)) = true

[134] gt(nat-con(y1 c1), nat-con(c2)) = true
[135] gt(nat-con(y c1), nat-con(y c2)) = gt(nat-con(c1), nat-con(c2))
[136] gt(nat-con(y1), nat-con(y2)) = true
====================================================================

gt(nat-con(y1 c1), nat-con(y2 c2)) = true

%%% -- cut off subtraction --/

[137] 0 -- Nat = 0
[138] Nat -- 0 = Nat

[00] nat-con(c1) --// nat-con(c2) = nat-con(c1) --// nat-con(c2)

[139] nat-con(c1) --// nat-con(c2) = nat-con(c)

====================================================================

28
nat-con(y1 c1) -/ nat-con(c2) = nat-con(y1 c)

[140] nat-con(c2) -/ nat-con(c1) = nat-con(c3),
nat-con(y1) -/ 1 = nat-con(y),
10 -/ nat-con(c3) = nat-con(c)
=================================================================
nat-con(y1 c1) -/ nat-con(c2) = nat-con(y c)

[139] nat-con(c1) -/ nat-con(c2) = nat-con(c),
nat-con(y1) -/ nat-con(y2) = nat-con(y)
=================================================================
nat-con(y1 c1) -/ nat-con(y2 c2) = nat-con(y c)

[140] nat-con(c2) -/ nat-con(c1) = nat-con(c3),
nat-con(y1) + 1 = Nat,
nat-con(y1) -/ Nat = nat-con(y),
10 -/ nat-con(c3) = nat-con(c)
=================================================================
nat-con(y1 c1) -/ nat-con(y2 c2) = nat-con(y c)

[141] gt(Nat1, Nat2) != true
=================================================================
Nat1 -/ Nat2 = 0

%%% -- subtraction of naturals
[142] gt(Nat1, Nat2) = true => Nat1 - Nat2 = Nat1 -/ Nat2
[143a] gt(Nat1, Nat2) != true, Nat1 != Nat2 => Nat1 - Nat2 = -(Nat2 -/ Nat1)
[143b] Nat - Nat = 0

%%% -- multiplication of naturals --
[144] Nat * 0 = 0
[145] Nat * 1 = Nat
[146] gt(nat-con(c), 1) = true
=================================================================
Nat * nat-con(c) = Nat + Nat * (nat-con(c) - 1)

[147] nat-con(y1) * nat-con(y2) = nat-con(y)
=================================================================
nat-con(y1) * nat-con(y2 c) =
nat-con(y "0") + nat-con(y1) * nat-con(c)

%%% -- addition, subtraction, and multiplication of integers
[148] Nat = Nat

[149] Nat1 + -Nat2 = Nat1 - Nat2
[150] -Nat1 + Nat2 = Nat2 - Nat1

[152] Nat1 - -Nat2 = Nat1 + Nat2
[154] -Nat1 - -Nat2 = Nat2 - Nat1


%%% -- relational operators
[158] gt(Nat1, Nat2) = true => Nat1 > Nat2 = true

29
gt(Nat1, Nat2) != true => Nat1 > Nat2 = false
Nat1 > -Nat2 = true
-Nat1 > Nat2 = false
-Nat1 > -Nat2 = Nat2 > Nat1

Nat1 != Int2 ==> Int1 > Int2 = Int1 > Int2
Int >= Int = true

Int < Int2 = ~(Int1 >= Int2)
Int <= Int2 = ~(Int1 > Int2)

Int == Int = true
Int1 = Nat-con1, Int2 = Nat-con2, Nat-con1 != Nat-con2
===============================================
Int1 == Int2 = false
end module Integers
C Abstract Datatypes

In this appendix modules defining the abstract datatypes strings, sets, tuples, sequences, tables and graphs are presented. They are intended to be reusable in the tools part of many specifications.

C.1 Strings

A string is just a sequence of characters enclosed by double-quotes. The only operations provided on strings are concatenation and equality.

module Strings
  exports
  sorts STRING
  lexical syntax
  "\"" "[\x]* "\"" -> STRING
  context-free syntax
  STRING "++" STRING -> STRING
  "(" STRING ")" -> STRING {bracket}
  STRING "==" STRING -> BOOL
  variables
  _.String[0-9]* -> STRING
  equations
  [0] string("""" Char* """"") ++ string("""" Char* """"") = string("""" Char* Char* """")
  [1] _String == String = true
  [2] _String != _String' ==> _String == _String' = false
end module Strings

C.2 Sets

Module Sets defines syntax and operations on a set of items. The items must be defined in a module that is similar to the following module.

C.2.1 Items for Sets

module Items
  exports
  sorts ITEM
  context-free syntax
  ITEM "==" ITEM -> BOOL
  variables
  Item[0-9]* -> ITEM
end module Items

C.2.2 Module Sets

module Sets
  imports Integers Booleans Items
  exports
  sorts SET
  context-free syntax
  "{\{ITEM\",\"\}"*\}" -> SET
  ITEM"*" SET -> SET
  SET "+" ITEM -> SET
  SET "*" SET -> SET {assoc}
SET "-" SET -> SET {left}
SET "&" SET -> SET {assoc}
rest SET -> SET
first SET -> ITEM
"empty?" SET -> BOOL
ITEM in SET -> BOOL
SET in SET -> BOOL
SET "=" SET -> BOOL
"|" SET "|" -> INT
"(" SET ")" -> SET {bracket}

variables
"Set"[0-9]* -> SET
"Item"[0-9]* -> ITEM
"'Item'"[0-9]* -> {ITEM ",",}*
"'Item'"[+][0-9]* -> {ITEM ",",}+

equations
% unicity of members
[00] {Item*1, Item, Item*2, Item, Item*3} = {Item*1, Item, Item*2, Item*3}

% union
[01] Item + {Item*} = {Item, Item*}
[02] {Item*1} + {Item*2} = {Item*1, Item*2}

% difference
[03] {} - Set = {}
[04] Set - {} = Set

[05] Item in Set = false

Set - {Item, Item*} = Set - {Item*}

[06] {Item*1, Item, Item*2} - {Item, Item*}
  = {Item*1, Item*2} - {Item, Item*}

% intersection
[07] Set1 & Set2 = Set1 - (Set1 - Set2)

% enumeration (partial)
[08] rest {Item, Item*} = {Item*}
[09] first {Item, Item*} = Item

% emptyness
[10] empty? {} = true

% membership
[12] Item in {} = false
[13] Item in {Item*1, Item, Item*2} = true
[14] Item1 != Item2

Item1 in {Item*1, Item2, Item*2} = Item1 in {Item*1, Item*2}

% subset
[15] {} in Set2 = true
C.3 Tuples

Module Tuples2 defines syntax and operations for a tuple consisting of two elements. In principle for every $n \in \mathbb{N}$ a module Tuples$n$ exists.

It is required for each module Tuples$n$ that for each $i \in \{1, \ldots, n\}$ a module Fields$i$ is defined which is similar to the following. (Note that fields may have identical sorts and may thus be defined by the same module.)

C.3.1 Module Fields$i$

module Fields$i$
exports
sorts FIELD$i$
context-free syntax
FIELD$i$ ":=" FIELD$i$ -> BOOL
variables
Fieldi[0-9]* -> FIELD$i$
end module Fields$i$

C.3.2 Module Tuples2

module Tuples2
imports Fields1 Fields2
exports
sorts TUPLE2 TUPLE2-UPDATE
context-free syntax
"(" FIELD1 "," FIELD2 ")" -> TUPLE2
"(" TUPLE2 ")" -> TUPLE2 [bracket]
"(" TUPLE2-UPDATE ")" -> TUPLE2-UPDATE [bracket]
TUPLE2 ":=" TUPLE2 -> BOOL

% Projection
TUPLE2 "." field1 -> FIELD1
TUPLE2 "." field2 -> FIELD2

% Modification
TUPLE2 ":=" TUPLE2-UPDATE -> TUPLE2
field1 ":=" FIELD1 -> TUPLE2-UPDATE
field2 ":=" FIELD2 -> TUPLE2-UPDATE
variables
"Tuple2"[0-9]*/ -> TUPLE2
"Tuple2-Update"[0-9]*/ -> TUPLE2-UPDATE
equations
C.4 Sequences

C.4.1 Items for Sequences

The module requires the existence of a module similar to the following.

module Items
exports
  sorts ITEM
  context-free syntax
    ITEM "==" ITEM -> BOOL
  variables
    Item[0-9]* -> ITEM
end module Items

C.4.2 Module Sequences

module Sequences
imports Integers Booleans Items
exports
  sorts SEQ
  context-free syntax
    "[" {ITEM ","}* "]" -> SEQ
    "(" SEQ ")" -> SEQ {bracket}
    ITEM "++" SEQ -> SEQ
    SEQ "++" ITEM -> SEQ
    SEQ "+-" SEQ -> SEQ {assoc}
    SEQ "+-" ITEM -> SEQ
    tl SEQ -> SEQ % partial
    hd SEQ -> ITEM % partial
    init SEQ -> SEQ % partial
    last SEQ -> ITEM % partial
    SEQ "!" INT -> ITEM % partial
    SEQ ":" INT ":=" ITEM -> SEQ % partial
    take INT SEQ -> SEQ
    take-until ITEM SEQ -> SEQ
    drop INT SEQ -> SEQ
    drop-until ITEM SEQ -> SEQ
    reverse SEQ -> SEQ
    multiples SEQ -> SEQ
    unique SEQ -> SEQ
    "#" SEQ -> INT
    "empty?" SEQ -> BOOL
    ITEM in SEQ -> BOOL
    SEQ "is-prefix?" SEQ -> BOOL
    SEQ in SEQ -> BOOL
end module Sequences
SEQ "=" SEQ -> BOOL
variables
"Seq"[0-9]* -> SEQ
"Item"[0-9]* -> {ITEM ","}*
"Item"[+][0-9]* -> {ITEM ","}+
equations
%% pre- and postfixing and concatenation
[cn1] Item ++ [Item] = [Item, Item]
[cn2] [Item] ++ Item = [Item*, Item]
[cn3] [Item*1] ++ [Item*2] = [Item*1, Item*2]
%% remove
[rm1] empty? Seq = true ===> Seq - Item = Seq
[rm2] Item = Item2 = true ===> [Item1, Item*] - Item2 = ([Item1] - Item2)
[rm3] Item1 = Item2 = false ===> [Item1, Item*] - Item2

= Item1 ++ ([Item1] - Item2)
%% difference
[d1] Seq1 - Seq = Seq when empty? Seq = true
[d2] Seq - Seq1 = Seq when empty? Seq = true
[d3] Seq - [Item*, Item*] = (Seq - Item) - [Item*]
%% head and tail (partial)
[tl0] tl [Item*, Item*] = [Item*]
[hd0] hd [Item*, Item*] = Item
%% last and init
[init0] init [Item*, Item*] = [Item*]
[last0] last [Item*, Item*] = Item
%% projection (partial)
[pr1] [Item, Item*] ! 1 = Item
[pr2] [Item, Item*] ! _Int = [Item*] ! _Int - 1 when _Int > 1 = true
%% Update
[upd] [Item, Item*] : 1 := Item' = [Item', Item*]
[upd] _Int > 1 = true

 ===> [Item, Item*] : _Int := Item' = Item ++ ([Item*] : _Int - 1 := Item')
%% take
[tk1] empty? Seq = true

==> take 0 Seq = [] when
[tk2] _Int > 0 = true, empty? Seq = true

==> take _Int Seq = Seq
[tk3] _Int > 0 = true

==> take _Int [Item, Item*] = Item ++ take (_Int - 1) [Item*]
%% take-until
[take-u1] empty? Seq = true

==> take-until Item Seq = []
[take-u2] Item = Item2 = true

==> take-until Item1 [Item2, Item*] = []
[take-u3] Item1 = Item2 = false

==> take-until Item1 [Item2, Item*] = Item2 ++ take-until Item1 [Item*]
%% drop-until
[drop-u1] empty? Seq = true

==> drop-until Item Seq = []
[drop-u2] Item = Item2 = true

==> drop-until Item1 [Item2, Item*] = [Item2, Item*]
[drop-u3] Item1 = Item2 = false

==> drop-until Item1 [Item2, Item*] = drop-until Item1 [Item*]
%% drop
[dr1] drop 0 Seq = Seq
[dr2] drop _Int Seq = Seq

when empty? Seq = true
[dr3] drop _Int [Item, Item*] = drop (_Int - 1) [Item*] when _Int > 0 = true
% reverse
[rv1] reverse Seq = Seq when empty? Seq = true
[rv2] reverse [Item, Item*] = (reverse [Item*]) ++ Item

% multiples (the elements that occur more than once in a sequence)
[mt1] multiples Seq = Seq when empty? Seq = true
[mt2] Item in [Item*] = true
    ===> multiples [Item, Item*] = Item ++ multiples ([Item*])
[mt2] Item in [Item*] = false
    ===> multiples [Item, Item*] = multiples (Seq - hd Seq)

% unique (all elements exactly once)
[un1] unique Seq = Seq when empty? Seq = true
[un2] unique Seq = (hd Seq) ++ unique (Seq - hd Seq) when empty? Seq = false

% length
[l1] #Seq = 0 when empty? Seq = true
[l2] #[Item, Item*] = 1 + #[Item*]

% empty
[e1] empty? Seq = true when Seq = []
[e2] empty? [Item, Item*] = false

% membership
[m1] Item in Seq = false when empty? Seq = true
[m2] Item in [Item2, Item*] = Item1 == Item2 | Item1 in [Item*]

% prefix
[p1] Seq is-prefix? Seq1 = true when empty? Seq = true
[p2] empty? Seq = true, empty? Seq1 = false ===> Seq1 is-prefix? Seq = false
[p3] empty? Seq1 = false, empty? Seq2 = false
                                  ==============================================================
                      Seq1 is-prefix? Seq2 = hd Seq1 == hd Seq2 & (tl Seq1) is-prefix? (tl Seq2)

% subsequence
[in1] empty? Seq = true, empty? Seq1 = false ===> Seq1 in Seq = false
[in2] empty? Seq2 = false
    ===> Seq1 in Seq2 = Seq1 is-prefix? Seq1 | Seq1 in tl Seq2

% equality
[eq1] empty? Seq = true
    ===> Seq == Seq = true
[eq2] empty? Seq1 /= empty? Seq2 = true
    ===> Seq1 == Seq2 = false
[eq2] empty? Seq1 = false, empty? Seq2 = false
    ===> Seq1 == Seq2 = hd Seq1 == hd Seq2 & tl Seq1 == tl Seq2

end module Sequences

C.5 Tables
Modules, defining keys and items, similar to the following should exist.

C.5.1 Keys for Tables

module Keys
exports
  sorts KEY
C.5.2 Items for Tables

module Items
imports Keys
exports
sorts ITEM
context-free syntax
ITEM "=" ITEM -> BOOL
ITEM "," key -> KEY
variables
Item[0-9]* -> ITEM
end module Items

C.5.3 Module Tables

module Tables
imports Item-Sequences
exports
context-free syntax
TABLE "</-" ITEM -> TABLE
TABLE "</-" TABLE -> TABLE
ITEM ":-" TABLE -> TABLE
TABLE ":-" TABLE -> TABLE
TABLE "," KEY -> ITEM
TABLE "," KEY -> BOOL
TABLE "," KEY -> TABLE
variables
"Table"[0-9]* -> TABLE
"Item"[0-9][0-9]* -> {ITEM ",",[0-9]*

equations
%%% update
[ud1] Table <- Item = (Table - Item.key) ++ Item
[ud2] Item -> Table = Item ++ (Table - Item.key)

[ud3] Table <- [] = Table
[ud4] Table <- [Item, Item*] = (Table <- Item) <- [Item*]

[ud5] [] -> Table = Table
[ud6] [Item*, Item] -> Table = [Item*] -> (Item -> Table)

%%% lookup (partial)
[l1] Item.key == Key = true ==> [Item, Item*].Key = Item
[l2] Item.key == Key = false ==> [Item, Item*].Key = [Item*].Key

%%% present
[pri] empty? Table = true ==> Table ? Key = false
[pri] Item.key == Key = true ==> [Item, Item*] ? Key = true
[pri] Item.key == Key = false ==> [Item, Item*] ? Key = [Item*] ? Key
C.6 Graphs

A module defining the NODEs of a graph should exist. On top of this module the following modules should be defined (as instantiations of the Sets, Tuples and Tables modules respectively), modules Node-Sets, Node-Node-Set-Pairs, NNP-Tables. The sort TABLE from this last module should be renamed to GRAPH.

C.6.1 Module Graphs

module Graphs
imports NNSP-Tables
%% via NNSP-Tables also Nodes Node-Sets Node-Node-Set-Pairs %
exports
context-free syntax
NNSP "+" GRAPH -> GRAPH
GRAPH "+" GRAPH -> GRAPH

"nb*"(" NODE "," NODE-SET "," NODE-SET "," GRAPH ")" -> NODE-SET

neighbours "(" NODE "," GRAPH ")" -> NODE-SET
"neighbours*""(" NODE "," GRAPH ")" -> NODE-SET

trans"(" GRAPH "," GRAPH ")" -> GRAPH
trans"(" GRAPH ")" -> GRAPH

variables
"Graph"[0-9]* -> GRAPH
"Node"[0-9]* -> NODE
"Node"[\*][0-9]* -> \{ NODE ",\}* 
"Node-Set"[0-9]* -> NODE-SET
"nns\*"[0-9]* -> NNSP
"nns\*"[\*][0-9]* -> \{ NNSP ",\}* 

equations
%% NNSP + GRAPH
[01] Graph\?Node = true , (Node, Node-Set') = Graph,Node
==============================================
(Node, Node-Set') + Graph = (Node, Node-Set + Node-Set') -> Graph

[02] Graph?Node = false ===>( Node, Node-Set ) + Graph = ( Node, Node-Set ) -> Graph

%% GRAPH + GRAPH
[03] empty? Graph = true ===>( Graph + Graph' = Graph' 
[04] [nns\*, nns\*] + Graph' = [nns\*] + (NNSP + Graph')
%%% neighbours(Node, Graph)
[05] Graph?Node = false ===> neighbours(Node, Graph) = {}


neighbours(Node, Graph) = Node-Set

%%% nb*
[07] empty? Node-Set2 = true ==> nb*(Node, Node-Set1, Node-Set2, Graph) = Node-Set1

[08] Node' in Node-Set1 = true ==>
    nb*(Node, Node-Set1, {Node', Node*}, Graph) = nb*(Node, Node-Set1, {Node*}, Graph)

[09] Node == Node' = true
    ==> nb*(Node, Node-Set1, {Node', Node*}, Graph)
        = nb*(Node, Node' + Node-Set1, {Node*}, Graph)

[10] Node' in Node-Set1 = false, Node' == Node = false,
    nb*(Node, Node-Set1, {Node', Node*}, Graph)
    = nb*(Node, Node' + Node-Set1, {Node*} + neighbours(Node', Graph), Graph)

%%% neighbours*(NODE, GRAPH)
[11] neighbours*(Node, Graph) = nb*(Node, {}, neighbours(Node, Graph), Graph)

%%% trans(GRAPH, GRAPH)
[12] trans([], Graph) = []
[13] trans([(Node, Node-Set), nmsp*], Graph)
    = (Node, neighbours*(Node, Graph)) + trans([nmsp*], Graph)

%%% trans(GRAPH)
[14] trans(Graph) = trans(Graph, Graph)
end module Graphs
D Reusing Modules

D.1 Module Abstraction

Module Abstraction describes the lambda abstraction and instantiation function, as well as the + function which serves to merge several modules into one. The sort MODULE is supposed to represent the set of all ASF+SDF modules.

module Abstraction
  imports %ASF-SDF-Modules %
  exports
    sorts MODULE ABSTR
    context-free syntax
      lambda ID+ ":" MODULE -> MODULE ABSTR
      MODULE ABSTR ID+ -> MODULE
      MODULE +" MODULE -> MODULE
      module ID = MODULE -> MODULE
    equations
      [0] (lambda id1 ... idn . Module) id1' ... idn'
          = Module[id1 := id1' ... idn := idn']
      [1] Module + Module'
          = (Module.syntax-part + Module'.syntax-part,
                  Module.equations-part + Module'.equations-part)
  end module Abstraction

D.2 Module Abbreviation

Module Abbreviation defines some syntactic sugar for lambda instantiations with many parameters. With the declaration provided by this module it is possible to declare a complex datastructure with many standard operations on it by just naming the composing sorts. (Compare this to the declaration of a structured type like array [x .. y] of integer in Pascal.)

module Abbreviation
  import Abstraction
  export
    sort FIELD-DECL TUPLE-DECL SET-DECL SEQ-DECL TABLE-DECL GRAPH-DECL
    context-free syntax
      module ID TUPLE-DECL end module ID -> MODULE
      module ID SET-DECL end module ID -> MODULE
      module ID SEQ-DECL end module ID -> MODULE
      module ID TABLE-DECL end module ID -> MODULE
      module ID GRAPH-DECL end module ID -> MODULE
      ID ":=" SORT var ID from ID -> FIELD-DECL
      SORT = tuple of {FIELD-DECL ":"}+
          variables ID ":=" SORT -> TUPLE-DECL
      SORT ":=" set of SORT from ID
          variables ID ":=" SORT;" ID ":=" SORT -> SET-DECL
      SORT ":=" sequence of SORT from ID
          variables ID ":=" SORT;" ID ":=" SORT -> SEQ-DECL
      SORT ":=" table of SORT from ID
key ID "-->" SORT from ID
variables
ID ";" SORT;" ID ";" SORT;"
ID ";" SORT --> TABLE-DECL

SORT ";=" graph of SORT from ID
variables
ID ";" SORT;" ID ";" SORT;"
ID ";" SORT --> TABLE-DECL

equations

[0] module Mod
Sort = tuple of
   id1 : Sort1 var Vari from Mod1
   ...
   idn : Sortn var Varn from Modn
variables Var : Sort
end module Mod
= (lambda Tuplesn TUPLEn TUPLEN Tuple1 Fields1 field1 FIELD1 Field1
   ...
   Fieldsn fieldn FIELDn Fieldn
   .
   module Tuplesn ... end module Tuplesn)

Mod Sort Var
Modi id1 Sort1 Vari
   ...
Modn idn Sortn Varn

[1] module Mod
Sort = set of Sort' from Mod'
variables Var : Sort; Var' : Sort'
end module Mod
= (lambda Sets SET Set Items ITEM Item . module Sets ... end module Sets)

Mod Sort Var Mod' Sort' Var'

[2] module Mod
Sort = sequence of Sort' from Mod'
variables Var : Sort; Var' : Sort'
end module Mod
= (lambda Sequences SEQ Seq Items ITEMS Item . module Sequences ... end module Sequences)

) Mod Sort Var Mod' Sort' Var'

[3] module Mod
Sort = table of Sort' from Mod'
key id --> Sort'' from Mod''
variables
   Var : Sort; Var' : Sort'; Var'' : Sort''
end module Mod
= (lambda Tables TABLE Table Items ITEM Item Keys KEY Key key . module Tables ... end module Tables)

) Mod Sort Var Mod' Sort' Var' Mod'' Sort'' Var'' id

[3] module Mod
Sort = graph of Sort' from Mod'
variables
   Var : Sort; Var' : Sort'; Var'' : Sort'';
end module Mod
=
(\lambda Graphs GRAPH Graph NNSP-Tables NODE Mode NNSP NNSP nmsp
 . module Tables ... end module Tables
 ) Mod Sort Var Mod' Sort' Var' Sort'' Var'' id

SORT "=" graph of SORT from ID;
   SORT from ID
variables
   ID ":=" SORT"; ID ":=" SORT";"
   ID ":=" SORT        -> TABLE-DECL
end module Abbreviation
E  Eiffel—Symbol Table

E.1 Declaration of Contexts

In the declarations below the syntax of Eiffel identifiers is imported. The definition of modules Identifier-Syntax and Identifier-Equality follow in appendix G.

E.1.1 Type-Sequences

module Type-Sequences
=
module Type-Sequences1
     TYPE-SEQ = sequence of E-TYPE from ""
     variables
     _Type-Seq : TYPE-SEQ;
     _E-Type : E-TYPE
end module Type-Sequences'

module E-Type
imports Identifier-Equality Integers
exports
sorts E-TYPE
context-free syntax
    formal"(" ID ")" -> E-TYPE
    anchor"(" ID ")" -> E-TYPE
    bit-type"(" INT ")" -> E-TYPE
    class-type"(" ID "," TYPE-SEQ ")" -> E-TYPE
    expanded"(" E-TYPE ")" -> E-TYPE
    void -> E-TYPE
    type-mark
        "(" E-TYPE ")" -> E-TYPE

    variables
        "_E-Type"[0-9]* -> E-TYPE

equations
    [01] _E-Type == _E-Type = true
    [02] _E-Type1 == _E-Type2 = false when _E-Type1 != _E-Type2

    [03] class-type(_id, _Type-Seq).actuals = _Type-Seq
    [04] class-type(_id, _Type-Seq).base = _id

    [05] expanded(expanded(_E-Type)) = expanded(_E-Type)

    [06] is-class-type(class-type(_id,_Type-seq)) = true
    [07] is-class-type(expanded(class-type(_id,_Type-seq))) = true
    [08] is-class-type(formal(_id)) = false
    [09] is-class-type(bit-type(_Int)) = false

    %
    % ...

    [10] is-reference-type(class-type(_id, _Type-Seq)) = true
    [11] is-reference-type(anchor(_id)) = true
    [12] is-reference-type(expanded(_E-Type)) = false
    [13] is-reference-type(formal(_id)) = false
    [14] is-reference-type(bit-type(n)) = false % ?????
[15] is-non-anchored-ref-type(class-type(_id,_Type-Seq)) = true
[16] _E-Type != class-type(_id,_Type-Seq) ==> is-non-anchored-ref-type(_E-Type) = false
end module E-Type

E.1.2 Errors

The error messages produced by the typechecker are either references to the validity rules in [Mey92] (rule name, page number and if appropriate item-number in the rule) that are being violated or a string explaining the error. The references could be expanded to explanations of the error, to copies of the validity rule, or left as they are. The typechecker merely administrates the errors, it is up to the user interface to present these properly to the user.

module Errors
imports Type-Sequences Booleans Strings
exports
sorts ERROR
context-free syntax
STRING => ERROR
validity("" ID "p." INT ") => ERROR
validity("" ID "p." INT item INT ") => ERROR
ID ":" ERROR => ERROR
E-TYPE ":" ERROR => ERROR
ERROR ":" ID => ERROR
ERROR ":" E-TYPE => ERROR
"(" ERROR ")" => ERROR {bracket}
ERROR "==" ERROR => BOOL
variables
_"_Error"[0-9]* => ERROR
_"_Error"[*][0-9]* => ERROR*
priorities
{ID ":" ERROR => ERROR, E-TYPE ":" ERROR => ERROR}
> {ERROR ":" ID => ERROR, ERROR ":" E-TYPE => ERROR}
equations
[000] _String = _Error == _Error == _Error = true
end module Errors

E.1.3 Error-Sequences

module Error-Sequences
ERRORS = sequence of ERROR from Errors
variables _Errors : ERRORS; _Error : ERROR
end module Error-Sequences

E.1.4 Signatures

module Signatures
SIGNATURE = tuple of
args : TYPE-SEQ var Type-Seq from Type-Sequences
res : TYPE-SEQ var Type-Seq from Type-Sequences
variables Signature : SIGNATURE
end module Signatures
E.1.5 Id-Sets
module Id-Sets
  ID-SET = set of ID from Feature-Name-Equality
  variables _Id-Set : ID-SET; _id : ID
end module Id-Sets

E.1.6 Id-Id-Set-Pairs
module Id-Id-Set-Pairs
  ID-ID-SET-PAIR = tuple of
    id : ID var _id from Identifier-Equality
    id-set : ID-SET var _Id-Set from Id-Sets
  variables _IISP : ID-ID-SET-PAIR
end module Signatures

E.1.7 Id-Id-Set-Pair-Sequences
module Id-Id-Set-Pair-Sequences
  INHERITANCE-GRAPH = sequence of ID-ID-SET-PAIR from Id-Id-Set-Pairs
  variables _ig : INHERITANCE-GRAPH; _IISP : ID-ID-SET-PAIR
end module Id-Id-Set-Pair-Sequences

E.1.8 Id-Id-Set-Pair-Tables
module Id-Id-Set-Pair-Tables
= module Type-Pair-Tables1
  INHERITANCE-GRAPH = table of ID-ID-SET-PAIR from Id-Id-Set-Pair-Sequences
  variables _ig : INHERITANCE-GRAPH
    _IISP : ID-ID-SET-PAIR
    _id : ID
end module Id-Id-Set-Pair-Tables

E.1.9 Inheritance-Graphs
module Inheritance-Graphs
  INHERITANCE-GRAPH = graph of ID-ID-SET-PAIR from Id-Id-Set-Pair-Sequences
  variables _ig : INHERITANCE-GRAPH
    _IISP : ID-ID-SET-PAIR
    _id : ID
end module Inheritance-Graphs

E.1.10 Type-Sets
module Type-Sets
  TYPE-SET = set of E-TYPE from Type-Sequences
  variables _Type-Set : TYPE-SET; _E-Type : E-TYPE
end module Type-Sets

E.1.11 Type-Pairs
module Type-Pairs
  TYPE-PAIR = tuple of
    source : E-TYPE var _E-Type from Type-Sequences
    target : E-TYPE var _E-Type from Type-Sequences
  variables _Type-pair : TYPE-PAIR
end module Type-Pairs
E.1.12 Type-Pair-Sequences

module Type-Pair-Sequences
  TYPE-PAIR-TABLE = sequence of TYPE-PAIR from Type-Pairs
  variables _Type-Pair-Table : TYPE-PAIR-TABLE; Type-pair : TYPE-PAIR
end module Type-Pair-Sequences

E.1.13 Type-Pair-Tables

module Type-Pair-Tables
  =
    module Type-Pair-Tables1
      TYPE-PAIR-TABLE = table of TYPE-PAIR from Type-Pair-Sequences
      key source -> E-TYPE from Type-Sequences
      variables
        _Type-Pair-Table : TYPE-PAIR-TABLE
        _Type-Pair : TYPE-PAIR
        _E-Type : E-TYPE
    end module Type-Pair-Tables1
  +
    module Type-Pair-Tables2
      exports
        context-free syntax
        sources("" TYPE-PAIR-TABLE ")" -> TYPE-SEQ
      equations
        [0] sources([]) = []
        [1] sources([_Type-Pair, _Type-Pair*])
          = _Type-Pair.source ++ sources([_Type-Pair*])
    end module Type-Pair-Tables2

E.1.14 Id-Type-Pairs

module Id-Type-Pairs
  ID-TYPE = tuple of
    id : ID var _id from Identifier-Equality
    type : E-TYPE var _E-Type from Type-Sequences
  variables
    _Id-Type : ID-TYPE
end module Id-Type-Pairs

E.1.15 Id-Type-Sequences

module Id-Type-Sequences
  ID-TYPE-TABLE = sequence of ID-TYPE from Id-Type-Pairs
  variables
    _Id-Type-Table : ID-TYPE-TABLE
    _Id-Type : ID-TYPE
  variables
end module Id-Type-Sequences

E.1.16 Id-Type-Tables

module Id-Type-Tables
  ID-TYPE-TABLE = table of ID-TYPE from Id-Type-Sequences
  key id -> ID from Identifier-Equality
  variables
    _Id-Type-Table : ID-TYPE-TABLE
    _Id-Type : ID-TYPE
    _id : ID
end module Id-Type-Tables
E.1.17 Feature-Contexts

module Feature-Contexts

FEATURE-CONTEXT = tuple of
  feature-name : ID var _id from Feature-Name-Equality ;
  signature : SIGNATURE var _Signature from Signatures ;
  clients : ID-SET var _Id-Set from Id-Set ;
  formals : ID-TYPE-TABLE var _Id-Type-Table from Id-Type-Table ;
  locals : ID-TYPE-TABLE var _Id-Type-Table from Id-Type-Table ;
  is-creator : BOOL var _Bool from Booleans ;
  is-deferred : BOOL var _Bool from Booleans ;
  is-internal : BOOL var _Bool from Booleans ;
  is-external : BOOL var _Bool from Booleans ;
  is-frozen : BOOL var _Bool from Booleans ;
  is-attribute : BOOL var _Bool from Booleans ;
  is-var-attribute : BOOL var _Bool from Booleans ;
  is-constant-attribute : BOOL var _Bool from Booleans ;
  is-unique : BOOL var _Bool from Booleans ;
  is-routine : BOOL var _Bool from Booleans ;
  is-procedure : BOOL var _Bool from Booleans ;
  is-function : BOOL var _Bool from Booleans ;
  is-one : BOOL var _Bool from Booleans ;
variables
  _FC : FEATURE-CONTEXT
end module Feature-Contexts

E.1.18 Feature-Sequences

module Feature-Sequences

FEATURE-TABLE = sequence of FEATURE-CONTEXT from Feature-Contexts
variables
  _Feature-Table : FEATURE-TABLE
  _FC : FEATURE-CONTEXT
end module Feature-Sequences

E.1.19 Feature-Tables

module Feature-Tables

FEATURE-TABLE = table of FEATURE-CONTEXT from Feature-Sequences
key feature-name -> ID from Identifier-Equality
variables
  _Feature-Table : FEATURE-TABLE
  _FC : FEATURE-CONTEXT
  _id : ID
end module Feature-Tables

E.1.20 Parent-Contexts

module Parent-Contexts

PARENT-CONTEXT = tuple of
  parent-name : E-TYPE var _E-Type from Type-Sequences ;
  parent-features : FEATURE-TABLE var _Feature-Table from Feature-Tables
variables _PC : PARENT-CONTEXT
end module Parent-Contexts
E.1.21 Parent-Sequences
module Parent-Sequences
  PARENT-TABLE = sequence of PARENT-CONTEXT from Parent-Contexts
  variables
    _Parent-Table : PARENT-TABLE
    _PC : PARENT-CONTEXT
end module Parent-Sequences

E.1.22 Parent-Tables
module Parent-Tables
  =
    module Parent-Tables1
      PARENT-TABLE = table of PARENT-CONTEXT from Parent-Sequences
      key parent-name -> E-TYPE from Type-Sequences
      variables
        _Parent-Table : PARENT-TABLE
        _PC : PARENT-CONTEXT
        _E-Type : E-TYPE
    end module Parent-Tables1
    +
    module Parent-Tables2
      imports Type-Sets
      exports
        context-free syntax
        parent-types("(" PARENT-TABLE ")") -> TYPE-SET
      equations
        [0] parent-types([]) = {}
        [1] parent-types([_PC,*]) = _PC.parent-name + parent-types([_PC*])
    end module Parent-Tables2

E.1.23 Class-Contexts
module Class-Contexts
  CLASS-CONTEXT = tuple of
    class-name : ID var _id from Identifier-Equality ;
    formal-generics : TYPE-PAIR-TABLE var _Type-Pair-Table from Type-Pair-Tables ;
    parents : PARENT-TABLE var _Parent-Table from Parent-Tables ;
    class-features : FEATURE-TABLE var _Feature-Table from Feature-Tables ;
    creators : ID-SET var _Id-Set from Id-Sets ;
    is-deferred : BOOL var _Bool from Booleans ;
    is-expanded : BOOL var _Bool from Booleans ;
    deferred-check : BOOL var _Bool from Booleans ;
  variables _CC : CLASS-CONTEXT
end module Class-Contexts

E.1.24 Class-Sequences
module Class-Sequences
  CLASS-TABLE = sequence of CLASS-CONTEXT from Class-Contexts
  variables
    _Class-Table : CLASS-TABLE;
    _CC : CLASS-CONTEXT;
end module Class-Sequences
E.1.25 Class-Tables
module Class-Tables
    CLASS-TABLE = table of CLASS-CONTEXT from Class-Sequences
    key class-name -> ID from Identifier-Equality
    variables
        _Class-Table : CLASS-TABLE;
        _CC : CLASS-CONTEXT;
        _id : ID
end module Class-Tables

E.1.26 System-Parameters
module System-Parameters
    SYSTEM-PARAMETERS = tuple of
        int-size : INTEGER var _Int from Integers;
        real-size : INTEGER var _Int from Integers;
        double-size : INTEGER var _Int from Integers;
        char-size : INTEGER var _Int from Integers;
        bool-size : INTEGER var _Int from Integers;
    variables _Param : SYSTEM-PARAMETERS
end module System-Parameters

E.1.27 System-Contexts
module System-Contexts
    SYSTEM-CONTEXT = tuple of
        sys-name : ID var _id from Identifier-Equality;
        root-name : ID var _id from Identifier-Equality;
        class-table : CLASS-TABLE var _Class-Table from Class-Tables;
        ig : INHERITANCE-GRAPH var _ig from Inheritance-Graphs;
        errors : ERRORS var _Errors from Error-Sequences;
        params : SYSTEM-PARAMETERS var _Param from System-Parameters;
    variables _SC : SYSTEM-CONTEXT
end module System-Contexts

E.1.28 Passes
module Passes
    exports
        sorts PASS
        context-free syntax
            class-names -> PASS
            formal-generics -> PASS
            feature-signatures -> PASS
            feature-bodies -> PASS
            "(" PASS ")" -> PASS {brackets}
            PASS ":=" PASS -> BOOL
        variables
            "_Pass"[0-9]* -> PASS
        equations
            [ph0] _Phase == _Phase = true
            [ph1] _Phase1 == _Phase2 = false when _Phase1 != _Phase2
end module Passes
E.1.29 Contexts

module Contexts =
module Contexts1
CONTEXT = tuple of
  cs : SYSTEM-CONTEXT var _SC from System-Contexts ;
  cc : CLASS-CONTEXT var _CC from Class-Contexts ;
  cp : PARENT-CONTEXT var _PC from Parent-Contexts ;
  cf : FEATURE-CONTEXT var _FC from Feature-Contexts ;
  c-clients : ID-SET var _Id-Set from Id-Sets ;
  cid : ID var _id from Feature-Name-Equality ;
  type-stack : TYPE-SEQ var _Type-Seq from Type-Sequences ;
  pass : PASS var _Pass from Passes
variables C : CONTEXT
end module Contexts1
module Contexts2
export context-free syntax
% Update
cs "." SYSTEM-CONTEXT-UPDATE -> CONTEXT-UPDATE
cc "." CLASS-CONTEXT-UPDATE -> CONTEXT-UPDATE
cp "." PARENT-CONTEXT-UPDATE -> CONTEXT-UPDATE
cf "." FEATURE-CONTEXT-UPDATE -> CONTEXT-UPDATE

% Type-Stack operations
type-stack "." push "(" E-TYPE ")" -> CONTEXT-UPDATE
type-stack "." push "(" INT ")" -> CONTEXT-UPDATE
type-stack "." update "(" INT "," E-TYPE ")" -> CONTEXT-UPDATE
type-stack "." pop -> CONTEXT-UPDATE
type-stack "." pop "(" INT ")" -> CONTEXT-UPDATE
CONTEXT "." type-stack "." top -> E-TYPE

signature "." SIGNATURE-UPDATE -> FEATURE-CONTEXT-UPDATE
CLASS-CONTEXT "." type -> E-TYPE

new-context"(" ID "," ID ")" -> CONTEXT
new-feature-context"(" ID ")" -> FEATURE-CONTEXT
no-feature-context -> FEATURE-CONTEXT
new-class-context"(" ID ")" -> CLASS-CONTEXT
no-class-context -> CLASS-CONTEXT
new-system-context"(" ID "," ID ")" -> SYSTEM-CONTEXT
no-system-context -> SYSTEM-CONTEXT
new-parent-context"(" E-TYPE ")" -> PARENT-CONTEXT
no-parent-context -> PARENT-CONTEXT
new-params -> SYSTEM-PARAMETERS

equations
% Update
[01] C:cs._SC-Update = C:cs := C:cs:_SC-Update
[02] C:cc._CC-Update = C:cc := C:cc:_CC-Update
[03] C:cp._PC-Update = C:cp := C:cp:_PC-Update
[04] C:cf._FC-Update = C:cf := C:cf:_FC-Update

% Type-Stack operations
[05] C:type-stack.push(_E-Type) = C:type-stack := _E-Type ++ C:type-stack
[06] C:type-stack.push(_Int) = C:type-stack.push(C:type-stack!_Int)
[07] C:type-stack.update(_Int,_E-Type) = C:type-stack := C:type-stack:_Int := _E-Type
[10] C:type-stack.top = hd C:type-stack


%%% Type of Current Class
[12] _CC.type = class-type(_CC.class-name, sources(_CC.formal-generics))

%%% new-...
[nc] new-context(_id1, _id2) = (new-system-context(_id1, _id2), no-class-context, no-parent-context, no-feature-context, SOME_ID, [], [], class-names)
[ncsc] new-system-context(_id1, _id2) = (_id1, _id2, [], [], new-params)
[ncse] new-class-context(_id) = (_id, [], [])
[ncfe] new-feature-context(_id) = (_id, ([], []), [], [], false, false, false, false, false, false, false, false, false, false, false, false, false, false, false)
[np] new-params = (0, 0, 0, 0)
end module Contexts

E.2 Interpretation In Context

First the in-Context modules for the ADT modules are defined. Then the abbreviations for these modules are specified. Finally module In-Context is declared.

E.2.1 Module Bool-in-Context

module Bool-in-Context
imports Contexts Booleans
exports
context-free syntax
  BOOL in CONTEXT -> BOOL
equations
[bic0] true in C = true
[bic1] false in C = false
[bic2] (~ Bool) in C = ~(Bool in C)
[bic3] (Bool1 | Bool2) in C = (Bool1 in C) | (Bool2 in C)
[bic4] (Bool1 & Bool2) in C = (Bool1 in C) & (Bool2 in C)
[bic5] (Bool1 => Bool2) in C = (Bool1 in C) => (Bool2 in C)
[bic6] (Bool1 <> Bool2) in C = (Bool1 in C) <> (Bool2 in C)
end module Bool-in-Context

E.2.2 Module Identifier-in-Context

module Identifier-in-Context
imports Identifier-Equality Contexts
exports
context-free syntax
  ID in CONTEXT -> ID
equations
[00] id(Char+) in C = id(Char+)
[00] lower-name(_id) in C = lower-name(_id in C)
[00] upper-name(_id) in C = upper-name(_id in C)
[00] (_id1 == _id2) in C = (_id1 in C) == (_id2 in C)
end module Identifier-in-Context
E.2.3 Module Integers-in-Context

module Integers-in-Context
imports Integers Contexts
exports
  context-free syntax
  INT in CONTEXT -> INT
  NAT-CON in CONTEXT -> NAT-CON
  NAT in CONTEXT -> NAT
priorities
  INT in CONTEXT -> INT
  > NAT in CONTEXT -> NAT
  > NAT-CON in CONTEXT -> NAT-CON
equations
  [00] _Int in C = _Nat-con when _Int = _Nat-con
  [00] (+ _Nat-con) in C = + (_Nat-con in C)
  [00] (- _Nat-con) in C = - (_Nat-con in C)
  [00] (_Int1 == _Int2) in C = (_Int1 in C) == (_Int2 in C)
  [00] (_Int1 + _Int2) in C = (_Int1 in C) + (_Int2 in C)
  [00] (_Int1 - _Int2) in C = (_Int1 in C) - (_Int2 in C)
  [00] (_Int1 *= _Int2) in C = (_Int1 in C) * (_Int2 in C)
  [00] (_Int1 > _Int2) in C = (_Int1 in C) > (_Int2 in C)
  [00] (_Int1 >= _Int2) in C = (_Int1 in C) >= (_Int2 in C)
  [00] (_Int1 < _Int2) in C = (_Int1 in C) < (_Int2 in C)
  [00] (_Int1 <= _Int2) in C = (_Int1 in C) <= (_Int2 in C)
end module Integers-in-Context

E.2.4 Module String-in-Context

module String-in-Context
imports Strings Contexts
exports
  context-free syntax
  STRING in CONTEXT -> STRING
equations
  [0] string(Char+) in Context = string(Char+)
  [1] (_String1 ++ _String2) in Context = (_String1 in Context) ++ (_String2 in Context)
end module String-in-Context

E.2.5 Module Sets-in-Context

module Sets-in-Context
imports Contexts Sets Items
exports
  context-free syntax
  SET in CONTEXT -> SET
equations
  [sic0] {} in Context = {}
  [sic1] {Item, Items} in Context = (Item in Context) + ({Items} in Context)
  [sic2] (Item + Set) in Context = (Item in Context) + (Set in Context)
  [sic3] (Set + Item) in Context = (Set in Context) + (Item in Context)
  [sic4] (Set + Set) in Context = (Set in Context) + (Set in Context)
  [sic5] (Set - Set) in Context = (Set in Context) - (Set in Context)
  [sic6] (Set & Set) in Context = (Set in Context) & (Set in Context)
  [sic7] (rest Set) in Context = rest (Set in Context)
  [sic8] (first Set) in Context = first (Set in Context)

52
E.2.6 Module Tuples2-in-Context

module Tuples2-in-Context
imports Tuples2 Contexts
exports
context-free syntax
TUPLE2 in CONTEXT \(\rightarrow\) TUPLE2
TUPLE2-UPDATE in CONTEXT \(\rightarrow\) TUPLE2-UPDATE
equations
[T2iC0] (Field11, Field22) in Context = (Field11 in Context, Field22 in Context)
[T2iC1] Tuple2.field1 in Context = (Tuple2 in Context).field1
[T2iC2] Tuple2.field2 in Context = (Tuple2 in Context).field2
[T2iC1] (Tuple2:field1 := Field1) in Context
    = (Tuple2 in Context):field1 := (Field1 in Context)
[T2iC2] (Tuple2:field2 := Field2) in Context
    = (Tuple2 in Context):field2 := (Field2 in Context)
end module Tuples2-in-Context

E.2.7 Module Sequences-in-Context

module Sequences-in-Context
imports Sequences Contexts
exports
context-free syntax
SEQ in CONTEXT \(\rightarrow\) SEQ
equations
[sic0] Seq in Context
    = Seq when empty? Seq = true
[sic1] [Item, Item\*] in Context
    = [Item in Context] ++ ([Item\*] in Context)
[sic2] (Item ++ Seq) in Context
    = (Item in Context) ++ (Seq in Context)
[sic3] (Seq ++ Item) in Context
    = (Seq in Context) ++ (Item in Context)
[sic4] (Seq - Item) in Context
    = (Seq in Context) - (Item in Context)
[sic5] (Seq - Item) in Context
    = (Seq in Context) - (Item in Context)
[sic6] (hd Seq) in Context
    = hd (Seq in Context)
[sic7] (tl Seq) in Context
    = tl (Seq in Context)
[sic8] (last Seq) in Context
    = last (Seq in Context)
[sic9] (init Seq) in Context
    = init (Seq in Context)
[sic0] (Seq ! _Int) in Context
    = (Seq in Context) ! (_Int in Context)
[sic0] (Seq : _Int := Item) in Context
    = (Seq in Context) : (_Int in Context) := (Item in Context)
[sic0] (take _Int Seq) in Context
    = take (_Int in Context) (Seq in Context)
[sic0] (drop _Int Seq) in Context
    = drop (_Int in Context) (Seq in Context)
[sic0] (take-until Item Seq) in Context
    = take-until (Item in Context) (Seq in Context)
[sic0] (drop-until Item Seq) in Context
    = drop-until (Item in Context) (Seq in Context)
[sic0] (reverse Seq) in Context
    = reverse (Seq in Context)
[sic0] (multiples Seq) in Context
    = multiples (Seq in Context)
[sic0] (unique Seq) in Context
    = unique (Seq in Context)
[sic0] (# Seq) in Context
    = # (Seq in Context)
[sic0] (empty? Seq) in Context
    = empty? (Seq in Context)
[sic0] (Item in Seq) in Context
    = (Item in Context) in (Seq in Context)
E.2.8 Module Tables-in-Context

module Tables-in-Context
imports Contexts
exports
context-free syntax
  TABLE in CONTEXT -> TABLE
equations
[tic0] (Table <- Item) in Context = (Table in Context) <- (Item in Context)
[tic0] (Table . Key) in Context = (Table in Context) . (Key in Context)
[tic0] (Table ? Key) in Context = (Table in Context) ? (Key in Context)
[tic0] (Table - Key) in Context = (Table in Context) - (Key in Context)
end module Tables-in-Context

E.2.9 Module Graphs-in-Context

module Graphs-in-Context imports Contexts
exports
context-free syntax
  GRAPH in CONTEXT -> GRAPH
equations
[08] (NNSP + Graph) in Context = (NNSP in Context) + (Graph in Context)
[08] (Graph + Graph') in Context = (Graph in Context) + (Graph' in Context)
[08] nb*(Node, Node-Set, Node-Set', Graph) in Context
    = nb*(Node in Context, Node-Set in Context, Node-Set' in Context, Graph in Context)
[08] neighbours(Node, Graph) in Context
    = neighbours(Node in Context, Graph in Context)
[08] neighbours*(Node, Graph) in Context
    = neighbours*(Node in Context, Graph in Context)
[08] trans(Graph, Graph') in Context
    = trans(Graph in Context, Graph' in Context)
[08] trans(Graph) in Context = trans(Graph in Context)
end module Graphs-in-Context

E.2.10 Abbreviations for in-Context Modules

module In-Context-Abbreviations
import Abbreviation
export
context-free syntax
  module integers ID SORT -> MODULE
  module booleans ID SORT -> MODULE
  module strings ID SORT -> MODULE
  module ID ID SORT SET-DECL end module ID -> MODULE
module ID ID SORT TUPLE-DECL end module ID -> MODULE
module ID ID SORT SEQ-DECL end module ID -> MODULE
module ID ID SORT TABLE-DECL end module ID -> MODULE
module ID ID SORT GRAPH-DECL end module ID -> MODULE

hiddens
variables
under -> ID
Renaming -> SORT

equations
[0] module integers under Renaming
  = (lambda in Context
    . module Integers-in-Context ... end module Integers-in-Context
  ) under Renaming

[0] module booleans under Renaming
  = (lambda in Context
    . module Booleans-in-Context ... end module Booleans-in-Context
  ) under Renaming

[0] module strings under Renaming
  = (lambda in Context
    . module Strings-in-Context ... end module Strings-in-Context
  ) under Renaming

[0] module Mod under Renaming
  Sort = tuple of
    id1 : Sort1 var Vari from Mod1
    ...
    idn : Sortn var Varn from Modn
  variables Var : Sort
  end module Mod
  = (lambda in Context
    Tuplen TUPLEn Tuplen
    Fieldsi fieldi FIELDi Fieldi
    ...
    Fieldsn fieldn FIELDn Fieldn
    .
    module Tuplesn-in-Context ... end module Tuplesn-in-Context
  ) under Renaming

  Mod Sort Var
  Modi id1 Sort1 Vari
  ...
  Modn idn Sortn Varn

[1] module Mod under Renaming
  Sort = set of Sort' from Mod'
  variables Var : Sort; Var' : Sort'
  end module Mod
  =
  (lambda in Context Sets SET Set Items ITEM Item
  . module Sets-in-Context ... end module Sets-in-Context
  ) under Renaming

[2] module Mod under Renaming
  Sort = sequence of Sort' from Mod'
  variables Var : Sort; Var' : Sort'
  end module Mod
  =
E.2.11 Declaration of Module In-Context

module In-Context =

module Contexts-in-Context
imports Contexts
exports
context-free syntax
  CONTEXT in CONTEXT -> CONTEXT
  CONTEXT-UPDATE in CONTEXT -> CONTEXT  %%%
equations
  %%% Context in Context
  [00] (_SC1, _CC2, _PC3, _FC4, _Type-Seq5) in C' = (_SC1 in C', _CC2 in C', _PC3 in C', _FC4 in C', _Type-Seq5 in C')

  %%% Context Projection in Context
  [00] C.cs in C' = (C in C').cs
  [00] C.cc in C' = (C in C').cc
  [00] C.cp in C' = (C in C').cp
  [00] C.cf in C' = (C in C').cf
  [00] C.type-stack in C' = (C in C').type-stack

  %%% Interpretation of CONTEXT-UPDATE in CONTEXT
  [00] (cs := _SC) in C = C:cs := (_SC in C)
  [00] (cc := _CC) in C = C:cc := (_CC in C)
  [00] (cp := _PC) in C = C:cp := (_PC in C)
  [00] (cf := _FC) in C = C:cf := (_FC in C)
  [00] (type-stack := _Type-Seq) in C = C:type-stack := (_Type-Seq in C)
  [00] cs._SC-Update in C = C:cs.(_SC-Update in C)
% Type stack operations in Context
[00] type-stack.push(_E-Type) in C = C:type-stack.push(_E-Type in C)
[00] type-stack.push(_Int) in C = C:type-stack.push(_Int in C)
[00] type-stack.update(_Int, _E-Type) in C = C:type-stack.update(_Int in C, _E-Type in C)
[00] type-stack.pop in C = C:type-stack.pop
[00] C'.type-stack.top in C = C':type-stack.top

% Signature Update in Context
[00] (signature..Signature-Update) in C = signature.(..Signature-Update in C)

% type of class-context in Context
[00] _CC.type in C = (_CC in C).type

% No-... in Context
[00] no-feature-context in C = no-feature-context
[00] no-class-context in C = no-class-context
[00] no-parent-context in C = no-parent-context
[00] no-system-context in C = no-system-context
end module Contexts-in-Context

+ module Passes-in-Context
exports context-free syntax
   PASS in CONTEXT -> PASS
equations
end module Passes-in-Context

+ module Errors-in-Context
imports Errors Contexts
exports context-free syntax
   ERROR in CONTEXT -> ERROR
equations
[00] _Error in C = _Error when _Error = _String
[00] validity(_id p..Int2) in C = validity(_id in C p..Int2 in C)
[00] validity(_id p..Int2 item _Int3) in C
   = validity(_id in C p..(Int2 in C item (_Int3 in C))

[00] (_id : _Error) in C = (_id in C) : (_Error in C)
[00] (_E-Type : _Error) in C = (_E-Type in C) : (_Error in C)
[00] (_Error : _id) in C = (_Error in C) : (_id in C)
[00] (_Error : _E-Type) in C = (_Error in C) : (_E-Type in C)
end module Errors-in-Context

+ module Identifier-in-Context
imports Identifier-Equality Contexts
exports
  context-free syntax
  ID in CONTEXT -> ID
equations
  [00] id(Char+) in C = id(Char+)
  [00] lower-name(_id) in C = lower-name(_id in C)
  [00] upper-name(_id) in C = upper-name(_id in C)
  [00] (_id == _id2) in C = (_id in C) == (_id2 in C)
end module Identifier-in-Context

module Operators-in-Context
imports Operator-Equality Contexts
exports
  context-free syntax
  INFIX-OPERATOR in CONTEXT -> INFIX-OPERATOR
  PREFIX-OPERATOR in CONTEXT -> PREFIX-OPERATOR
equations
  [oiC0] Prefix-Op in C = Prefix-Op
  [oiC1] Infix-Op in C = Infix-Op
end module Operators-in-Context

module Feature-Name-in-Context
imports Feature-Name-Equality Contexts
equations
  [fniC0] binary(Infix-Op) in C = binary(Infix-Op)
  [fniC1] unary(Prefix-Op) in C = unary(Prefix-Op)
  [fniC2] normal(Feature-Name) in C = normal(Feature-Name)
end module Feature-Name-in-Context

module booleans in Context

module integers in Context

module strings in Context

module Type-Sequences in Context
  TYPE-SEQ = sequence of E-TYPE from E-Types
variables
  _Type-Seq : TYPE-SEQ;
  _E-Type : E-TYPE
end module Type-Sequences'

module E-Type-in-Context
exports
  context-free syntax
  E-TYPE in CONTEXT -> E-TYPE
equations
  [ticC0] (_E-Type1 == _E-Type2) in C = (_E-Type1 in C) == (_E-Type2 in C)
  [ticC0] class-type(_id, _Type-Seq) in C = class-type(_id in C, _Type-Seq in C)
  [ticC0] anchor(_id) in C = anchor(_id in C)
[tic00] formal(_id) in C = formal(_id in C)
[tic00] bit-type(_Int) in C = bit-type(_Int in C)
[tic00] expanded(_E-Type) in C = expanded(_E-Type in C)
[tic00] type-mark in C = type-mark
[tic00] void in C = void
[tic00] (_E-Type.actuals) in C = (_E-Type in C).actuals
[tic00] (_E-Type.base) in C = (_E-Type in C).base
end module E-Type-in-Context

+ module Signatures in Context
  SIGNATURE = tuple of
  args : TYPE-SEQ var Type-Seq from Type-Sequences
  res : TYPE-SEQ var Type-Seq from Type-Sequences
  variables Signature : SIGNATURE
end module Signatures

+ module Id-Sets in Context
  ID-SET = set of ID from Identifier-Equality
  variables _Id-Set : ID-SET; _id : ID
end module Id-Sets

+ module Id-Id-Set-Pairs in Context
  ID-SET-PAIR = tuple of
  id : ID var _id from Identifier-Equality
  id-set : ID-SET var _Id-Set from Id-Sets
  variables _IISP : ID-ID-SET-PAIR
end module Signatures

+ module Id-Id-Set-Pair-Sequences in Context
  INHERITANCE-GRAPH = sequence of ID-ID-SET-PAIR from Id-Id-Set-Pairs
  variables _ig : INHERITANCE-GRAPH; _IISP : ID-ID-SET-PAIR
end module Id-Id-Set-Pair-Sequences

+ module Id-Id-Set-Pair-Tables in Context
  INHERITANCE-GRAPH = table of ID-ID-SET-PAIR from Id-Id-Set-Pair-Sequences
  variables _ig : INHERITANCE-GRAPH
  _IISP : ID-ID-SET-PAIR
  _id : ID
end module Id-Id-Set-Pair-Tables

+ module Inheritance-Graphs in Context
  INHERITANCE-GRAPH = graph of ID-ID-SET-PAIR from Id-Id-Set-Pair-Sequences
  variables _ig : INHERITANCE-GRAPH
  _IISP : ID-ID-SET-PAIR
  _id : ID
end module Inheritance-Graphs

+ module Type-Sets in Context
  TYPE-SET = set of E-TYPE from Type-Sequences
  variables _Type-Set : TYPE-SET; _E-Type : E-TYPE
end module Type-Sets

+ module Type-Pairs in Context
  TYPE-PAIR = tuple of
  source : E-TYPE var _E-Type from Type-Sequences
target : E-TYPE var _E-Type from Type-Sequences
variables _Type-pair : TYPE-PAIR
end module Type-Pairs
+
module Type-Pair-Sequences in Context
  TYPE-PAIR-TABLE = sequence of TYPE-PAIR from Type-Pairs
variables _Type-Pair-Table : TYPE-PAIR-TABLE; Type-pair : TYPE-PAIR
end module Type-Pair-Sequences
+
module Type-Pair-Tables1 in Context
  TYPE-PAIR-TABLE = table of TYPE-PAIR from Type-Pair-Sequences
key source -> E-TYPE from Type-Sequences
variables
  _Type-Pair-Table : TYPE-PAIR-TABLE
  _Type-Pair : TYPE-PAIR
  _E-Type : E-TYPE
end module Type-Pair-Tables
+
module Type-Pair-Tables2 in Context
  equations
  [0] sources(_Type-Pair-Table) in Context = sources(_Type-Pair-Table in Context)
end module Type-Pair-Tables2
+
module Id-Type-Pairs in Context
  ID-TYPE = tuple of
    id : ID var _id from Identifier-Equality
    type : E-TYPE var _E-Type from Type-Sequences
variables
  _Id-Type : ID-TYPE
end module Id-Type-Pairs
+
module Id-Type-Sequences in Context
  ID-TYPE-TABLE = sequence of ID-TYPE from Id-Type-Pairs
variables
  _Id-Type-Table : ID-TYPE-TABLE
  _Id-Type : ID-TYPE
variables
end module Id-Type-Sequences
+
module Id-Type-Tables in Context
  ID-TYPE-TABLE = table of ID-TYPE from Id-Type-Sequences
key id -> ID from Identifier-Equality
variables
  _Id-Type-Table : ID-TYPE-TABLE
  _Id-Type : ID-TYPE
  _id : ID
end module Id-Type-Tables
+
module Feature-Contexts in Context
  FEATURE-CONTEXT = tuple of
    feature-name : ID var _id from Identifier-Equality;
    signature : SIGNATURE var _Signature from Signatures;
    clients : ID-SET var _Id-Set from Id-Sets;
    formals : ID-TYPE-TABLE var _Id-Type-Table from Id-Type-Tables;
    locals : ID-TYPE-TABLE var _Id-Type-Table from Id-Type-Tables;
    is-creator : BOOL var _Bool from Booleans;

is-deferred : BOOL var _Bool from Booleans ;
is-internal : BOOL var _Bool from Booleans ;
is-external : BOOL var _Bool from Booleans ;
is-frozen : BOOL var _Bool from Booleans ;
is-attribute : BOOL var _Bool from Booleans ;
is-var-attribute : BOOL var _Bool from Booleans ;
is-const-attribute : BOOL var _Bool from Booleans ;
is-unique : BOOL var _Bool from Booleans ;
is-routine : BOOL var _Bool from Booleans ;
is-procedure : BOOL var _Bool from Booleans ;
is-function : BOOL var _Bool from Booleans ;
is-once : BOOL var _Bool from Booleans ;

variables
.FC : FEATURE-CONTEXT
end module Feature-Contexts
+
module Feature-Sequences in Context

FEATURE-TABLE = sequence of FEATURE-CONTEXT from Feature-Contexts

variables
._Feature-Table : FEATURE-TABLE
.FC : FEATURE-CONTEXT
end module Feature-Sequences
+
module Feature-Tables in Context

FEATURE-TABLE = table of FEATURE-CONTEXT from Feature-Sequences
key feature-name -> ID from Identifier-Equality

variables
._Feature-Table : FEATURE-TABLE
.FC : FEATURE-CONTEXT
.id : ID
end module Feature-Tables
+
module Parent-Contexts in Context

PARENT-CONTEXT = tuple of

parent-name : E-TYPE var _E-Type from Type-Sequences ;
parent-features : FEATURE-TABLE var _Feature-Table from Feature-Tables

variables
._PC : PARENT-CONTEXT
end module Parent-Contexts
+
module Parent-Sequences in Context

PARENT-TABLE = sequence of PARENT-CONTEXT from Parent-Contexts

variables
._Parent-Table : PARENT-TABLE
.FC : PARENT-CONTEXT
end module Parent-Sequences
+
module Parent-Tables1 in Context

PARENT-TABLE = table of PARENT-CONTEXT from Parent-Sequences
key parent-name -> E-TYPE from Type-Sequences

variables
._Parent-Table : PARENT-TABLE
.FC : PARENT-CONTEXT
._E-Type : E-TYPE
end module Parent-Tables
+
module Parent-Tables2 in Context

equations
[0] parent-types(_Parent-Table) in Context

61
E.3 A Context Modification Language
sorts COND-CONTEXT-EXPR ELSE-PART
context-free syntax
  with CONTEXT do CONTEXT-UPDATE -> CONTEXT
  "@" -> CONTEXT % Dummy Context

%%% Structuring Constructs
id-context -> CONTEXT-UPDATE
begin {CONTEXT-UPDATE";;"} end -> CONTEXT-UPDATE
if COND-CONTEXT-EXPR ELSE-PART end -> CONTEXT-UPDATE
BOOL then {CONTEXT-UPDATE ";;"} -> COND-CONTEXT-EXPR
  -> ELSE-PART
else if COND-CONTEXT-EXPR ELSE-PART -> ELSE-PART
else {CONTEXT-UPDATE ";;"} -> ELSE-PART
case CASE* OTHERWISE end -> CONTEXT-UPDATE
BOOL {CONTEXT-UPDATE";;"} -> CASE
otherwise {CONTEXT-UPDATE";;"} -> OTHERWISE

%%% Abbreviations
require BOOL else ERROR -> CONTEXT-UPDATE
error ERROR -> CONTEXT-UPDATE
enter-class"("ID")" -> CONTEXT-UPDATE
leave-class -> CONTEXT-UPDATE
enter-parent"("E-TYPE")" -> CONTEXT-UPDATE
leave-parent -> CONTEXT-UPDATE
enter-feature"(" ID ")" -> CONTEXT-UPDATE
leave-feature -> CONTEXT-UPDATE
variables
  "CU"[0-9]* -> CONTEXT-UPDATE
  "CU"[+][0-9]* -> {CONTEXT-UPDATE ","}+
  "CU"[+][0-9]* -> {CONTEXT-UPDATE ","}+
  "Cond-CU"[0-9]* -> COND-CONTEXT-EXPR
hiddens
  context-free syntax
  if(" BOOL ", CONTEXT-UPDATE "," CONTEXT-UPDATE ")" -> CONTEXT-UPDATE
variables
  Cond[0-9]* -> BOOL
  Claim -> BOOL
  "EP" -> ELSE-PART
equations
  %%% Interpretation of CONTEXT-UPDATE in Context
  [with-0] with C do CU = CU in C
  [dummy] @ in C = C
  [id-c] id-context in C = C
  [be-1] begin end in C = C
  [be-2] begin CU; CU* end in C = begin CU* end in (CU in C)

%%% Conditional Context Expressions
  [if-0] if(false, CU1, CU2) = CU2
  [if-1] if(true, CU1, CU2) = CU1
  [case-0] case Case* otherwise CU* end = case Case* true CU* end
  [case-1] case Cond CU1* end in C = if(Cond in C, begin CU1* end, id-context) in C
  [case-2] case Cond CU1 Case+ end in C
    = if(Cond in C, begin CU1 end, case Case+ end) in C
  [if-2] if Cond1 then CU1* end in C

63
= if(Cond1 in C, begin CU*1 end, id-context) in C
[if-3] if Cond1 then CU*1 else CU*2 end in C
    = if(Cond1 in C, begin CU*1 end, begin CU*2 end) in C
[if-4] if Cond1 then CU*1 else if Cond-CU EP end in C
    = if(Cond1 in C, begin CU*1 end, if Cond-CU EP end) in C

%% Abbreviations

%% Errors
[req] require Claim else _Error = if "Claim" then error _Error end
[er] error _Error = cs.errors := @.cs.errors ++ _Error

%% Enter/Leave Class
[ec1] enter-class(_id)
    = begin
        if @.cs.class-table ? _id then
            cc := @.cs.class-table._id
        else
            cc := new-class-context(_id)
        end;
        error "errors in clas" : _id
    end
[lc] leave-class
    = begin
        cs.class-table := @.cs.class-table <- @.cc;
        error "end class" : @.cc.class-name;
        cc := no-class-context
    end

%% Enter/Leave Parent
[ep1] enter-parent(_E-Type)
    = begin
        if @.cc.parent-table ? _E-Type then
            cp := @.cc.parent-table._E-Type
        else
            cp := new-parent-context(_E-Type)
        end;
        error "errors in parent" : _Type
    end
[lp] leave-parent
    = begin
        cc.parent-table := @.cp -> @.cc.parent-table;
        error "end parent" : @.cp._Type
    end

%% Enter/Leave Feature
[ef] enter-feature(_id)
    = begin
        if @.cc.class-features ? _id then
            cf := @.cc.class-features._id
        else
            cf := new-feature-context(_id)
        end;
        error "errors in feature" : _id
    end
[lf] leave-feature
    = begin
        cc.class-features := @.cc.class-features <- @.cf;
error "end feature" : @.cf.feature-name;
cf := no-feature-context
end
end module CML
F Eiffel—Operations on Types

F.1 Type Substitutions

The function subst takes a table of type pairs and uses it to substitute types for the formal generic parameters to which they are associated in the table. The function is extended to several sorts containing types.

```plaintext
module Type-Substitutions
imports CML
exports
context-free syntax
make-substitution(" E-TYPE ") -> TYPE-PAIR-TABLE
make-substitution(" E-TYPE , CONTEXT ") -> TYPE-PAIR-TABLE
E-TYPE subst TYPE-PAIR-TABLE -> E-TYPE
SIGNATURE subst TYPE-PAIR-TABLE -> SIGNATURE
TYPE-PAIR subst TYPE-PAIR-TABLE -> TYPE-PAIR
TYPE-PAIR-TABLE subst TYPE-PAIR-TABLE -> TYPE-PAIR-TABLE
ID-TYPE subst TYPE-PAIR-TABLE -> ID-TYPE
ID-TYPE-TABLE subst TYPE-PAIR-TABLE -> ID-TYPE-TABLE
FEATURE-CONTEXT subst TYPE-PAIR-TABLE -> FEATURE-CONTEXT
FEATURE-TABLE subst TYPE-PAIR-TABLE -> FEATURE-TABLE
variables
sigma[0-9]* -> TYPE-PAIR-TABLE
hiddens
context-free syntax
make-substitution(" TYPE-SEQ , TYPE-SEQ ") -> TYPE-PAIR-TABLE
equations
% MAKE-SUBSTITUTION
[ms-0] make-substitution(_E-Type) in C = make-substitution(_E-Type in C, C)
[ms-1] make-substitution(class-type(_id, _Type-Seq), C)
 = make-substitution(sources(C.cs.class-table._id.formal-generics), _Type-Seq)
[ms-2] make-substitution([], []) = []
[ms-3] make-substitution([_Type1, _Type*1],[_Type2, _Type*2])
 = (_Type1, _Type2) ++ make-substitution([_Type1, _Type*2])
% E-TYPE
[00] sigma?formal(_id) = true ===> formal(_id) subst sigma = sigma.formal(_id).target
[00] sigma?formal(_id) = false ===> formal(_id) subst sigma = formal(_id)
[00] class-type(_id, _Type-Seq) subst sigma = class-type(_id, _Type-Seq subst sigma)
[00] expanded(_E-Type) subst sigma = expanded(_E-Type subst sigma)
[00] anchor(_id) subst sigma = anchor(_id)
[00] bit-type(_Int) subst sigma = bit-type(_Int)
[00] type-mark subst sigma = type-mark
% SIGNATURE
[00] _Signature subst sigma = (_Signature.args subst sigma, _Signature.args subst sigma)
% TYPE-SET
[00] _Type-Set subst sigma = _Type-Set when empty? _Type-Set = true
[00] empty? _Type-Set = false
 ==> _Type-Set subst sigma = (first _Type-set subst sigma)
 + ((rest _Type-Set subst sigma))
% TYPE-PAIR
[00] _Type-Pair subst sigma = (_Type-pair.source subst sigma, _Type-pair.target subst sigma)
% TYPE-PAIR-TABLE
```
empty? _Type-Pair-Table = true ==> _Type-Pair-Table subst sigma = _Type-Pair-Table
empty? _Type-Pair-Table = false
  ==> _Type-Pair-Table subst sigma = (hd _Type-Pair-Table) subst sigma
      ++ ((tl _Type-Pair-Table) subst sigma)

%% ID-TYPE
_Id-Type subst sigma = (_Id-Type.id, _Id-Type.type subst sigma)

%% ID-TYPE-TABLE
empty? _Id-Type-Table = true ==> _Id-Type-Table subst sigma = _Id-Type-Table
empty? _Id-Type-Table = false
  ==> _Id-Type-Table subst sigma = (hd _Id-Type-Table) subst sigma
      ++ ((tl _Id-Type-Table) subst sigma)

%% FEATURE-CONTEXT
_FC subst sigma = _FC:signature := (_FC.signature subst sigma)
        :formals := (_FC.formals subst sigma)
        :locals := (_FC.locals subst sigma)

%% FEATURE-TABLE
empty? _Feature-Table = true ==> _Feature-Table subst sigma = _Feature-Table
empty? _Feature-Table = false
  ==> _Feature-Table subst sigma = (hd _Feature-Table) subst sigma
        ++ ((tl _Feature-Table) subst sigma)

%% SUBSTITUTION in CONTEXT
(E-Type subst sigma) in C = (E-Type in C) subst (sigma in C)
(Signature subst sigma) in C = (Signature in C) subst (sigma in C)
(Type-Set subst sigma) in C = (Type-Set in C) subst (sigma in C)
(Type-pair subst sigma) in C = (Type-pair in C) subst (sigma in C)
(Type-Pair-Table subst sigma) in C = (Type-Pair-Table in C) subst (sigma in C)
(Id-Type subst sigma) in C = (Id-Type in C) subst (sigma in C)
(Id-Type-Table subst sigma) in C = (Id-Type-Table in C) subst (sigma in C)
(FC subst sigma) in C = (FC in C) subst (sigma in C)
(Feature-Table subst sigma) in C = (Feature-Table in C) subst (sigma in C)

end module Type-Substitutions

F.2 Bitsize of Types

Function bit-size(A) computes the size, in bits, of an object of a type A. This value can be computed by a recursive procedure that sums up the bit-sizes of the types of the features of the class. The bit-sizes of the primitive types INTEGER, REAL, DOUBLE, CHARACTER, and BOOLEAN from which all classes are built are implementation dependent.

module Bit-Size
  imports Type-Substitutions
  exports
    context-free syntax
    bit-size(" E-TYPE ")" -> INT
  hidden
    context-free syntax
    bit-size(" E-TYPE "," CONTEXT ")" -> INT
    bit-size(" FEATURE-TABLE "," CONTEXT ")" -> INT
  equations
[0] bit-size(_E-Type) in C = bit-size(_E-Type in C, C)
[1] bit-size(class-type(INTEGER, []), C) = C.cs.params.int-size
[2] bit-size(class-type(REAL, []), C) = C.cs.params.real-size
[3] bit-size(class-type(DOUBLE, []), C) = C.cs.params.double-size
[4] bit-size(class-type(CHARACTER, []), C) = C.cs.params.char-size
[5] bit-size(class-type(BOOLEAN, []), C) = C.cs.params.bool-size

end module Bit-Size

67
[6] bit-size(bit-type(_.Int), C) = _.Int
[7] bit-size(bit-type(_.id), C) = 0 % !!!! lookup constant value

[6] _.id != INTEGER, _.id != REAL, _.id != DOUBLE,
   _.id != CHARACTER, _.id != BOOLEAN,
sigma = make-substitution(class-type(_.id, _.Type-Seq), C),
   _Feature-Table = subst(sigma, C.cs.class-table..id.feature-table)
   ==============================================================
   bit-size(class-type(_.id, _.Type-Seq), C) = bit-size(_Feature-Table, C)

[7] bit-size([], C) = 0

[8] % bit-size of attribute
   _FC.signature = ([],[E-Type]), _FC.is-var-attribute = true
   ==============================================================
   bit-size([_FC, _FC*], C) = bit-size(E-Type, C) + bit-size([_FC*], C)

[9] % bit-size of routine is 0
   _FC.is-attribute := false
   ==============================================================
   bit-size([_FC, _FC*], C) = bit-size([_FC*], C)
end module Bit-Size

F.3 Conformance of Types, Typesequences and Signatures

Module Conformance specifies the important notion of conformance of types. This notion is used in determining the type correctness of expressions, assignments and feature calls. For instance, the assignment

   x := y

is type correct if the type of x conforms to the type of y. The main rule is that type A conforms to type B if class A is a descendant (inherits from or is equal to) class B.

The transcription of the rules from [Mey92] resulted in a rather non-declarative specification. The equation labels are named after the rules on which they are based.

module Conformance
   imports Bit-Size
   exports
      context-free syntax
         E-TYPE "<" E-TYPE -> BOOL
         SIGNATURE "<" SIGNATURE -> BOOL
         TYPE-SEQ "<" TYPE-SEQ -> BOOL
   hidden
      context-free syntax
         SIGNATURE "<" SIGNATURE with CONTEXT -> BOOL
         TYPE-SEQ "<" TYPE-SEQ with CONTEXT -> BOOL
         E-TYPE "<" E-TYPE with CONTEXT -> BOOL
         E-TYPE "<" E-TYPE -> BOOL
         E-TYPE "<" E-TYPE with CONTEXT -> BOOL
         E-TYPE "<" E-TYPE with CONTEXT -> BOOL
         E-TYPE "<" E-TYPE with CONTEXT -> BOOL
         TYPE-SET "<" E-TYPE with CONTEXT -> BOOL

   direct conformance from E-TYPE in CONTEXT -> TYPE-SET
variables

68
equations
[0] (E-Type < E-Type') in C = (E-Type in C) < (E-Type' in C) with C
[1] (Signature < Signature') in C = (Signature in C) < (Signature' in C) with C
[2] (Type-Seq < Type-Seq') in C = (Type-Seq in C) < (Type-Seq' in C) with C

%% SIGNATURE CONFORMANCE (p. 219)
[VNCS-0]
Signature < Signature' with C = (Signature.args < Signature'.args with C)
& (Signature.res < Signature'.res with C)

%% TYPE-SEQUENCE CONFORMANCE
[VNCS-1]
empty? _Type-Seq1 = true, empty? _Type-Seq2 = true
=================================
_Type-Seq1 < _Type-Seq2 with C = true
[VNCS-2]
empty? _Type-Seq1 */= empty? _Type-Seq2 = true
=================================
_Type-Seq1 < _Type-Seq2 with C = false
[VNCS-3]
empty? _Type-Seq1 = false, empty? _Type-Seq2 = false
=================================
_Type-Seq1 < _Type-Seq2 with C = (hd _Type-Seq1 < hd _Type-Seq2 with C)
& (tl _Type-Seq1 < tl _Type-Seq2 with C)

%% BIT-TYPE CONFORMANCE (p. 229)
[VNB-1] bit-type(n) bt-< bit-type(m) = n <= m
[VNB-2] _Type != bit-type(m) ==> _Type bt-< bit-type(n) = bit-size(_Type) <= n

%% GENERAL CONFORMANCE (of types)

%% conforms (p. 229)
[VGCC-a1]
_Type2 = bit-type(n) ==> _Type1 < _Type2 with C = _Type1 bt-< _Type2
[VGCC-a2]
_Type1 = bit-type(m), _Type2 != bit-type(n)
=================================
_Type1 < _Type2 with C = false
[VGCC-a3]
_Type1 != bit-type(m), _Type2 != bit-type(n), _Type2 != anchor(_id)
=================================
_Type1 < _Type2 with C = _Type1 <1 _Type2 with C
[VNB] % no type conforms (directly) to an anchored type (p. 225)
_Type2 = anchor(_id)
=================================
_Type1 < _Type2 with C = false

%% conforms1 (p. 219)
[VGCC-1] _Type <1 _Type with C = true
[VGCC-2]
_Type != class-type(lower-name(NULL),[])  
=================================
class-type(lower-name(NULL),[]) <1 _Type with C = is-reference-type(_Type)
[VGCC-b] % 1 or 2 do not apply
_Type1 != class-type(lower-name(NULL),[]), _Type1 != _Type2
=================================
_Type1 <1 _Type2 with C = _Type1 <2 to _Type2 with C

%% conforms2 (p. 219)
[VNCC-3]
class-type(_id, _Type-Seq1) < 2 class-type(_id, _Type-Seq2) = _Type-Seq1 < _Type-Seq2
% VNCC-3 does not apply
[VNCC-c1]
_id1 == _id2 = false
===============================================================================
class-type(_id1, _Type-Seq1) < 2 class-type(_id2, _Type-Seq2)
= class-type(_id1, _Type-Seq1) < 3 class-type(_id2, _Type-Seq2)
[VNCC-c2]
_Type1 != class-type(_id1, _Type-Seq1), _Type2 = class-type(_id2, _Type-Seq2)
===============================================================================
_Type1 < 2 _Type2 = _Type1 < 3 _Type2
[VNCC-c3]
_Type1 != class-type(_id1, _Type-Seq1), _Type2 != class-type(_id2, _Type-Seq2)
===============================================================================
_Type1 < 2 _Type2 = _Type1 < 3 _Type2
[VNCC-c4]
_Type1 != class-type(_id1, _Type-Seq1), _Type2 != class-type(_id2, _Type-Seq2)
===============================================================================
_Type1 < 2 _Type2 = _Type1 < 3 _Type2
% conforms3 (p.219)
[VNCC-d1] is-reference-type(_Type2) = false ==> _Type1 < 3 _Type2 = false
[VNCC-d2]
is-reference-type(_Type2) = true, _Type-Set = direct conformance from _Type1 with C
===============================================================================
_Type1 < 3 _Type2 = Type-set < 4 _Type2
% type-set conforms to type (p.219)
[VNCC-4-1] empty? _Type-set = true ==> Type-set < _Type2 with C = false
[VNCC-4-2]
empty? _Type-Set = false
===============================================================================
_Type-Set < _Type with C = first _Type-Set < _Type with C
| rest _Type-Set < _Type with C
% DIRECT CONFORMANCE
% by inheritance (p.221 & p.222)
[VNCH-VNCG]
C.cs.class-table._id.is-expanded = false,
sigma = make-substitution(class-type(_id, _Type-Seq), C),
_Type-Set = parent-types(C.cs.class-table._id.parents),
_Type-Set’ = subst(sigma, Type-set)
===============================================================================
direct conformance from class-type(_id, _Type-Seq) with C = _Type-Set’
% from formal generic parameter (p.224)
[VNCF]
_Type = C.cc.formal-generics._id.target
===============================================================================
direct conformance from formal(_id) with C = {_Type}
% from Anchor (p.225)
[VNCH-1]
_Type = class-type(C.cc.class-name, sources(C.cc.formal-generics))
===============================================================================
direct conformance from anchor(Current) with C = {_Type}
[VNCH-2a]
_id != Current, C.cf.formals?_id = true, _Type = C.cf.formals._id.type
direct conformance from anchor(_id) with C = {_Type}
[VPCH-2b]
  _id != Current, C.cc.class-features?._id = true,
  _Type = C.cc.class-features._id.signature.res

end module Conformance
G Eiffel—Syntax and Static Semantics

The syntax of Eiffel is divided in several small modules; along with each syntax module the appropriate typechecking module is presented. The modules are in bottom-up order. This may not be the best order to read the specification; starting at the top (modules System…) may give more overview.

Typechecking takes place by first generating a CML program (a large context modifier) and then applying that program to the initial (empty) context. The initial context can be filled with default information (classes INTEGER, CHARACTER and such) and the initial context can be the final context of typechecking a previous program. (This can only yield something meaningful if the classes in the previous program are all higher up in the inheritance and dependency hierarchies.)

G.1 Layout

module Eiffel-Layout
exports
lexical syntax
[ \t\n] -> LAYOUT
"--" "[\n]* -> LAYOUT
end module Eiffel-Layout

G.2 Identifiers
G.2.1 Identifier-Syntax

module Identifier-Syntax
imports Eiffel-Layout
exports
sorts ID
lexical syntax
[a-zA-Z][0-9_]* -> ID
variables
"_id"[0-9_]* -> ID
end module Identifier-Syntax

G.2.2 Identifier-Equality

module Identifier-Equality
imports Identifier-Syntax Layout Booleans
exports
context-free syntax
"(" ID ")" -> ID {bracket}
ID "=" ID -> BOOL
upper-name(" ID ") -> ID
lower-name(" ID ") -> ID
variables
"_id"[0-9_]* -> ID
Char[0-9_]* -> CHAR
Char[+][0-9_]* -> CHAR*
Char[+][0-9_]* -> CHAR
hiddens
sorts LETTER
lexical syntax
[a-zA-Z0-9_] -> LETTER
context-free syntax
"2upper"(" LETTER ") -> LETTER
"2lower"(" LETTER ") -> LETTER
variables
C[*] -> CHAR*
equations
/* Conversion of identifiers to lowercase. These equations make the lower-name and upper-name functions worthless. The reason they are put in the specification is to simplify the equations over constructs containing identifiers; no explicit conversion is needed. What we really want is one place in the specification (f.i. vc-id) where the conversion is done and the result is stored in the context. With these equations we have no longer control over how identifiers are represented. */
[cA] id(C* "A" C*) = id(C* "a" C*)
[cB] id(C* "B" C*) = id(C* "b" C*)
[cC] id(C* "C" C*) = id(C* "c" C*)
[cD] id(C* "D" C*) = id(C* "d" C*)
[cE] id(C* "E" C*) = id(C* "e" C*)
[cF] id(C* "F" C*) = id(C* "f" C*)
[cG] id(C* "G" C*) = id(C* "g" C*)
[ch] id(C* "H" C*) = id(C* "h" C*)
[cl] id(C* "I" C*) = id(C* "i" C*)
[cJ] id(C* "J" C*) = id(C* "j" C*)
[cK] id(C* "K" C*) = id(C* "k" C*)
[cl] id(C* "L" C*) = id(C* "l" C*)
[cm] id(C* "M" C*) = id(C* "m" C*)
[cm] id(C* "N" C*) = id(C* "n" C*)
[co] id(C* "O" C*) = id(C* "o" C*)
[cp] id(C* "P" C*) = id(C* "p" C*)
[cq] id(C* "Q" C*) = id(C* "q" C*)
[cr] id(C* "R" C*) = id(C* "r" C*)
[cs] id(C* "S" C*) = id(C* "s" C*)
[ct] id(C* "T" C*) = id(C* "t" C*)
[ch] id(C* "U" C*) = id(C* "u" C*)
[cy] id(C* "V" C*) = id(C* "v" C*)
[cw] id(C* "W" C*) = id(C* "w" C*)
[cx] id(C* "X" C*) = id(C* "x" C*)
[cy] id(C* "Y" C*) = id(C* "y" C*)
[cz] id(C* "Z" C*) = id(C* "z" C*)
/* Equality */
[0] id(Char+) == id(Char+) = true
[1] id(Char+) == id(Char+) = false when id(Char+) != id(Char+)
/* The following definition of the equality predicate would be better (no conversion to lower-name needed) but are *very* inefficient. */
[0] lower-name(id(Char+)) = lower-name(id(Char+))
===============================================
id(Char+) == id(Char+) = true
[1] lower-name(id(Char+)) != lower-name(id(Char+))
===============================================
id(Char+) == id(Char+) = false
/*
[2] letter(Char') = 2lower(letter(Char))
===============================================
lower-name(id(Char)) = id(Char')
[3] letter(Char') = 2lower(letter(Char)), id(Char+)) = lower-name(id(Char+))
===============================================
lower-name(id(Char Char+)) = id(Char' Char+)'
[4] letter(Char') = 2upper(letter(Char))
===============================================
upper-name(id(Char)) = id(Char')

73
[5] letter(Chr') = 2upper(letter(Chr)), id(Chr') = upper-name(id(Chr'))
==============================================================================
  upper-name(id(Chr Char')) = id(Chr' Char')

[21] 2lower(_ _) = _
[21a] 2lower(0) = 0 [21b] 2lower(1) = 1 [21c] 2lower(2) = 2
[21d] 2lower(3) = 3 [21e] 2lower(4) = 4 [21f] 2lower(5) = 5
[21g] 2lower(6) = 6 [21h] 2lower(7) = 7 [21i] 2lower(8) = 8
[21j] 2lower(9) = 9
[21k] 2lower(a) = a [21l] 2lower(b) = b [21m] 2lower(c) = c
[21n] 2lower(d) = d [21o] 2lower(e) = e [21p] 2lower(f) = f
[21q] 2lower(g) = g [21r] 2lower(h) = h [21s] 2lower(i) = i
[21t] 2lower(j) = j [21u] 2lower(k) = k [21v] 2lower(l) = l
[21w] 2lower(m) = m [21x] 2lower(n) = n [21y] 2lower(o) = o
[21z] 2lower(p) = p [21A] 2lower(q) = q [21B] 2lower(r) = r
[21C] 2lower(s) = s [21D] 2lower(t) = t [21E] 2lower(u) = u
[21F] 2lower(v) = v [21G] 2lower(w) = w [21H] 2lower(x) = x
[21I] 2lower(y) = y [21J] 2lower(z) = z
[21K] 2lower(A) = a [21L] 2lower(B) = b [21M] 2lower(C) = c
[21N] 2lower(D) = d [21O] 2lower(E) = e [21P] 2lower(F) = f
[21Q] 2lower(G) = g [21R] 2lower(H) = h [21S] 2lower(I) = i
[21T] 2lower(J) = j [21U] 2lower(K) = k [21V] 2lower(L) = l
[21W] 2lower(M) = m [21X] 2lower(N) = n [21Y] 2lower(O) = o
[21Z] 2lower(P) = p [21A] 2lower(Q) = q [21B] 2lower(R) = r
[21C] 2lower(S) = s [21D] 2lower(T) = t [21E] 2lower(U) = u
[21F] 2lower(V) = v [21G] 2lower(W) = w [21H] 2lower(X) = x
[21I] 2lower(Y) = y [21J] 2lower(Z) = z
[21K] 2upper(_ _) = _
[21L] 2upper(0) = 0 [21a1] 2upper(1) = 1 [21a2] 2upper(2) = 2
[21b] 2upper(3) = 3 [21c] 2upper(4) = 4 [21d] 2upper(5) = 5
[21e] 2upper(6) = 6 [21f] 2upper(7) = 7 [21g] 2upper(8) = 8
[21h] 2upper(9) = 9
[21i] 2upper(a) = B [21j] 2upper(b) = B [21k] 2upper(c) = C
[21l] 2upper(d) = D [21m] 2upper(e) = E [21n] 2upper(f) = F
[21o] 2upper(g) = G [21p] 2upper(h) = H [21q] 2upper(i) = I
[21r] 2upper(j) = J [21s] 2upper(k) = K [21t] 2upper(l) = L
[21u] 2upper(m) = M [21v] 2upper(n) = N [21w] 2upper(o) = O
[21x] 2upper(p) = P [21y] 2upper(q) = Q [21z] 2upper(r) = R
[21A] 2upper(s) = S [21B] 2upper(t) = T [21C] 2upper(u) = U
[21G] 2upper(y) = Y [21H] 2upper(z) = Z
[21L] 2upper(D) = D [21M] 2upper(E) = E [21N] 2upper(F) = F
[21O] 2upper(G) = G [21P] 2upper(H) = H [21Q] 2upper(I) = I
[21X] 2upper(P) = P [21Y] 2upper(Q) = Q [21Z] 2upper(R) = R
[21a] 2upper(S) = S [21b] 2upper(T) = T [21c] 2upper(U) = U
[21d] 2upper(V) = V [21e] 2upper(W) = W [21f] 2upper(X) = X
[21g] 2upper(Y) = Y [21h] 2upper(Z) = Z

end module Identifier-Equality
G.2.3 Identifier-Validity

module Identifier-Validity
  imports CML
  exports
  context-free syntax
  vc-id "[" ID "]" -> CONTEXT-UPDATE
  eiffel-reserved-words -> ID-SET
  equations
  [0] vc-id [_id] = require ~lower-name(_id) in eiffel-reserved-words
  else _id : validity(VIRW, 418);
  [1] eiffel-reserved-words =
  { alias, all, and, as, bit,
    /*boolean,*/ /*character,*/ check, class, creation,
    current, debug, deferred, do, /*double,*/
    else, elsif, end, ensure, expanded,
    export, external, false, feature, from,
    frozen, if, implies, indexing, infix,
    inherit, inspect, /*integer,*/ invariant, is,
    like, local, loop, /*none,*/ obsolete,
    old, once, or, prefix, /*real,*/
    redefine, rename, require, rescue, /*result,*/
    retry, select, separate, /*string,*/ strip,
    then, true, undefined, unique, until,
    variant, when, xor }
end module Identifier-Validity

G.3 Operators

G.3.1 Operator-Syntax

module Operator-Syntax
  imports Eiffel-Layout
  exports
  sorts FREE-OPERATOR PREFIX-OPERATOR INFIX-OPERATOR UNARY BINARY
  lexical syntax
  [0-4] "[\t\n]* -> FREE-OPERATOR
  not -> UNARY "~" -> UNARY "~" -> UNARY
  "/" -> BINARY "\" -> BINARY "\" -> BINARY
  "<" -> BINARY "<" -> BINARY "<" -> BINARY
  "=" -> BINARY "=" -> BINARY "=" -> BINARY
  ">" -> BINARY ">" -> BINARY and -> BINARY
  or -> BINARY xor -> BINARY and then -> BINARY
  or else -> BINARY implies -> BINARY
  UNARY -> PREFIX-OPERATOR
  FREE-OPERATOR -> PREFIX-OPERATOR
  BINARY -> INFIX-OPERATOR
  FREE-OPERATOR -> INFIX-OPERATOR
  variables
  Prefix-[0-9]+[0-9]* -> PREFIX-OPERATOR
  Infix-[0-9]+[0-9]* -> INFIX-OPERATOR
  Unary[0-9]* -> UNARY
  Binary[0-9]* -> BINARY
end module Operator-Syntax
### G.3.2 Operator-Equality

module Operator-Equality
imports Operator-Syntax
exports
context-free syntax
   "(" PREFIX-OPERATOR ")" -> PREFIX-OPERATOR {bracket}
   "(" INFIX-OPERATOR ")" -> INFIX-OPERATOR {bracket}
PREFIX-OPERATOR "==" PREFIX-OPERATOR -> BOOL
INFIX-OPERATOR "==" INFIX-OPERATOR -> BOOL
equations
[0] Prefix-op == Prefix-op' = true
[1] Prefix-op == Prefix-op' = false when Prefix-op != Prefix-op'
[2] Infix-op == Infix-op' = true
[3] Infix-op == Infix-op' = false when Infix-op != Infix-op'
end module Operator-Equality

### G.4 Feature-Names

#### G.4.1 Feature-Name-Syntax

module Feature-Name-Syntax
imports Identifier-Syntax Operator-Syntax
exports
sorts NEW-FEATURE-LIST NEW-FEATURE
FEATURE-NAME PREFIX INFIX FROZEN
context-free syntax
   { NEW-FEATURE "," }+ -> NEW-FEATURE-LIST
   FROZEN FEATURE-NAME -> NEW-FEATURE
   -> FROZEN
   ID -> FEATURE-NAME
   PREFIX -> FEATURE-NAME
   INFIX -> FEATURE-NAME
prefix """" PREFIX-OPERATOR """" -> PREFIX
infix """" INFIX-OPERATOR """" -> INFIX
variables
   New-Feature[+][0-9]* -> NEW-FEATURE-LIST
   New-Feature[+][0-9]* -> { NEW-FEATURE "," }*
   New-Feature[0-9]* -> NEW-FEATURE
   Frozen[0-9]* -> FROZEN
   Feature-[Nn]ame[0-9]* -> FEATURE-NAME
   Prefix[0-9]* -> PREFIX

end module Feature-Name-Syntax

#### G.4.2 Feature-Name-Equality

module Feature-Name-Equality
imports Feature-Name-Syntax Identifier-Equality Operator-Equality
exports
context-free syntax
   binary"(" INFIX-OPERATOR ")" -> ID
   unary "(" PREFIX-OPERATOR ")" -> ID
   normal"(" FEATURE-NAME ")" -> ID
equations

[00] normal(id(Char+)) = upper-name(id(Char+))
[01] normal(prefix "Prefix-op") = unary(Prefix-op)
[02] normal(infix "Infix-op") = binary(Infix-op)

[03] upper-name(unary(Prefix-op)) = unary(Prefix-op)
[04] lower-name(unary(Prefix-op)) = unary(Prefix-op)
[05] upper-name(binary(Infix-op)) = binary(Infix-op)
[06] lower-name(binary(Infix-op)) = binary(Infix-op)

[07] unary(Prefix-op) == unary(Prefix-op') = Prefix-op == Prefix-op'
[08] binary(Infix-op) == binary(Infix-op') = Infix-op == Infix-op'
[09] unary(Prefix-op) == binary(Infix-op) = false
[10] binary(Infix-op) == unary(Prefix-op) = false
[11] unary(Prefix-op) == id(Char+) = false
[12] id(Char+) == unary(Prefix-op) = false
[13] binary(Infix-op) == id(Char+) = false
[14] id(Char+) == binary(Infix-op) = false

end module Feature-Name-Equality

G.5 Bit-Sequences

G.5.1 Bit-Sequence-Syntax

module Bit-Sequence-Syntax
imports Eiffel-Layout
exports
sorts BIT-SEQUENCE
lexical syntax
[01]+[bB] -> BIT-SEQUENCE
variables
[Bb]it"-"[Ss]eq[0-9]* -> BIT-SEQUENCE

end module Bit-Sequence-Syntax

G.5.2 Bit-Sequence-Validity

module Bit-Sequence-Validity
imports Bit-Sequence-Syntax CML
exports
context-free syntax
vc-bit-sequence ""[" BIT-SEQUENCE "]" -> CONTEXT-UPDATE
count-bits ""[" BIT-SEQUENCE "]" -> INT
hiddens
variables
n -> INT
Char[']* -> CHAR
equations
[0] n = count-bits [[ Bit-Seq ]]

vc-bit-sequence [[ Bit-Seq ]] = type-stack.push(bit-type(n))

[1] count-bits [[ bit-sequence(Char Char') ]] = 1
[2] count-bits [[ bit-sequence(Char Char+ Char') ]] = 1 + count-bits [[ bit-sequence(Char+ Char') ]] end module Bit-Sequence-Validity
G.6 Booleans

G.6.1 Boolean-Syntax

module Boolean-Syntax
imports Eiffel-Layout
exports
  sorts BOOLEAN-CONSTANT
  lexical syntax
  true -> BOOLEAN-CONSTANT
  false -> BOOLEAN-CONSTANT
  variables
    [Bb]ool-[Cc]onst[0-9]* -> BOOLEAN-CONSTANT
end module Boolean-Syntax

G.6.2 Boolean-Validity

module Boolean-Validity
imports Booleans-Syntax CML
exports
  context-free syntax
  vc-bool-const "[[ " BOOLEAN-CONSTANT "]" -> CONTEXT-EXPR
  equations
    [0] vc-bool-const [[ true ]] = type-stack.push(BOOLEAN)
    [1] vc-bool-const [[ false ]] = type-stack.push(BOOLEAN)
end module Boolean-Validity

G.7 Integers

G.7.1 Integer-Syntax

module Integer-Syntax
exports
  sorts INTEGER DIGIT SHORT-INT
  lexical syntax
    [0-9] -> DIGIT
    "," DIGIT DIGIT DIGIT -> SHORT-INT
    DIGIT DIGIT* -> INTEGER
    DIGIT SHORT-INT* -> INTEGER
    DIGIT DIGIT SHORT-INT+ -> INTEGER
  variables
    E-Int[0-9]* -> INTEGER
    Digit[0-9]* -> DIGIT
    Short-Int[0-9]* -> SHORT-INT
end module Integer-Syntax

G.7.2 Integer-Validity

module Integer-Validity
imports Integer-Syntax CML
exports
  context-free syntax
    vc-integer "[[ " INTEGER "]" -> CONTEXT-UPDATE
    "2int" "[[ " INTEGER "]" -> INT

78
equations

[0] \text{vc-integer } [[\text{ E-Int }]] = \text{type-stack.push(class-type(INTEGER,[]))}


\begin{align*}
\text{Char } &= _-
\end{align*}

\begin{align*}
\text{2int } [[\text{ integer(Char Char Char') }]] &= \text{2int } [[\text{ integer(Char Char Char') }]]
\end{align*}

\begin{align*}
\text{Char } &= _-
\end{align*}

\begin{align*}
\text{2int } [[\text{ integer(Char) }]] &= \text{int(Char)}
\end{align*}

\begin{align*}
\text{Char } &= _-, \text{\ int(Char+')} &= \text{2int } [[\text{ Char+ }]]
\end{align*}

\begin{align*}
\text{2int } [[\text{ integer(Char Char+)}]] &= \text{int(Char Char+')}
\end{align*}

end module Integer-Validity

G.8 Reals

G.8.1 Real-Syntax

module Real-Syntax
import Integer-Syntax
exports
sorts REAL EXPONENT
lexical syntax

\begin{align*}
\text{INTEGER } "." & \rightarrow \text{REAL} \\
"." \text{ INTEGER} & \rightarrow \text{REAL} \\
\text{INTEGER } "." \text{ INTEGER} & \rightarrow \text{REAL} \\
\text{INTEGER } "." \text{ EXPONENT} & \rightarrow \text{REAL} \\
"." \text{ INTEGER EXPONENT} & \rightarrow \text{REAL} \\
\text{INTEGER } "." \text{ INTEGER EXPONENT} & \rightarrow \text{REAL} \\
\text{E-Real}[0-9]* & \rightarrow \text{REAL}
\end{align*}

end module Real-Syntax

G.8.2 Real-Validity

module Real-Validity
import Real-Syntax Integer-Validity
exports
context-free syntax
\text{vc-real } "[\\  \text{ REAL } ]" \rightarrow \text{CONTEXT-UPDATE}
equations

\begin{align*}
\text{0] \text{vc-real } [[\text{ E-Real }]]} &= \text{type-stack.push(class-type(REAL,[]))}
\end{align*}

end module Real-Validity

G.9 Characters and Strings

G.9.1 String-Char-Syntax

module String-Char-Syntax
exports
sorts MANIFEST-STRING CHARACTER-CONSTANT
lexical syntax
FIRST-PART* SIMPLE-STRING-CHAR* "\"" -> MANIFEST-STRING
[']' SIMPLE-STRING-CHAR [']'] -> CHARACTER-CONSTANT

variables

[\[\]manifest\]"[\[\]s\]tring\]0-9\]* -> MANIFEST-STRING
[\[\]char\]"[\[\]c\]onst\]0-9\]* -> CHARACTER-CONSTANT

hiddens

sorts WS FIRST-PART SIMPLE-STRING-CHAR

lexical syntax

[\[\][]] -> WS
SIMPLE-STRING-CHAR* "\"n\" WS* "\" -> FIRST-PART
"\"n] -> SIMPLE-STRING-CHAR
"\" [\[ABCDFHLMNRSTU][VX]()]<>] -> SIMPLE-STRING-CHAR
"\"/ [0-9]+ "/" -> SIMPLE-STRING-CHAR

end module String-Char-Syntax

G.9.2 String-Char-Validity

module String-Char-Validity

imports String-Char-Syntax CML

exports

context-free syntax

vc-manifest-string "[" MANIFEST-STRING "]" -> CONTEXT-UPDATE
vc-char-const "[" CHARACTER-CONSTANT "]" -> CONTEXT-UPDATE

equations

[0] vc-manifest-string [[_Manifest-string]] = type-stack.push(class-type(STRING,[]))
[1] vc-char-const [[_Char-const]] = type-stack.push(class-type(CHARACTER,[]))

end module String-Char-Validity

G.10 Constants

G.10.1 Constant-Syntax

module Constant-Syntax

imports Boolean-Syntax String-Char-Syntax Real-Syntax Bit-Sequence-Syntax

exports

sorts MANIFEST-CONSTANT CONSTANT-ATTRIBUTE

SIGN INTEGER-CONSTANT REAL-CONSTANT BIT-CONSTANT

context-free syntax

BOOLEAN-CONSTANT -> MANIFEST-CONSTANT
CHARACTER-CONSTANT -> MANIFEST-CONSTANT
INTEGER-CONSTANT -> MANIFEST-CONSTANT
REAL-CONSTANT -> MANIFEST-CONSTANT
MANIFEST-STRING -> MANIFEST-CONSTANT
BIT-CONSTANT -> MANIFEST-CONSTANT

SIGN INTEGER -> INTEGER-CONSTANT
SIGN REAL -> REAL-CONSTANT
SIGN "+" -> SIGN
SIGN "-" -> SIGN

BIT-SEQUENCE -> BIT-CONSTANT

variables

_.Const[0-9]* -> MANIFEST-CONSTANT
._Int-const[0-9]* -> INTEGER-CONSTANT
._Real-const[0-9]* -> REAL-CONSTANT
"_Sign"[0-9]* -> SIGN

end module Constant-Syntax

G.10.2 Constant-Validity

module Constant-Validity
imports Constant-Syntax Bit-Sequence-Validity String-Char-Validity Real-Validity
exports
context-free syntax
vc-const "[" MANIFEST-CONSTANT "]" -> CONTEXT-UPDATE
"2int" "[" INTEGER-CONSTANT "]" -> INT
equations
[0] 2int [ + E-Int ] = + 2int [ E-Int ]

[3] vc-const [_[Char-const ]] = vc-char-const [_[Char-const ]]
[5] vc-const [_[Bit-seq ]] = vc-bit-sequence [_[Bit-seq ]]
[7] vc-const [_[Sign E-Int ]]
   = begin
      vc-integer [ E-Int ]
   end
end module Constant-Validity

G.11 Types

G.11.1 Type-Syntax

module Type-Syntax
imports Constant-Syntax Identifier-Syntax
exports
sorts TYPE CLASS-TYPE ACTUAL-GENERICS TYPE-LIST
   CLASS-TYPE-EXPANDED BIT-TYPE ANCHORED ANCHOR
context-free syntax
   CLASS-TYPE   -> TYPE
   CLASS-TYPE-EXPANDED -> TYPE
   ANCHORED     -> TYPE
   BIT-TYPE     -> TYPE
   ID           -> CLASS-TYPE
   ID ACTUAL-GENERICS -> CLASS-TYPE
   "[" TYPE-LIST "]"   -> ACTUAL-GENERICS
   {TYPE ","+    -> TYPE-LIST
expanded CLASS-TYPE -> CLASS-TYPE-EXPANDED
   "BIT" ID     -> BIT-TYPE
   "BIT" INTEGER -> BIT-TYPE
   like ANCHORED -> ANCHORED
   ID           -> ANCHOR
%%% "Current"   -> ANCHOR
variables
"."[Tk]type[0-9]* -> TYPE
"."[Tk]Type[0-9]* -> (TYPE ","+)*
"."[Tk]Class"[Tk]type[0-9]* -> CLASS-TYPE
"."[Tk]Actual-Generics"[0-9]* -> ACTUAL-GENERICS
end module Type-Syntax

G.11.2 Type-Validity

module Type-Validity
imports Type-Syntax Identifier-Validity Constant-Validity
exports
context-free syntax
vc-type "[" TYPE "]" -> CONTEXT-UPDATE
vc-actual-generics "[" ACTUAL-GENERICS "]" -> CONTEXT-UPDATE
vc-type-list "[" TYPE-LIST "]" -> CONTEXT-UPDATE
hiddens
context-free syntax
sat-vcg(" TYPE-SEQ "," TYPE-PAIR-TABLE ")" -> CONTEXT-UPDATE
equations
% TYPE
/* validity VTCT p.199 (Class Type rule)
   An Identifier CC is valid as the Class-name part of a
   Class-type if and only if it is the name of a class in
   the surrounding universe. */
[tv01]
vc-type [[_id]]
= begin
  vc-id [[_id]];
  case
    @.cc.formal-generics?formal(_id)
      type-stack.push(formal(_id))
    @.cs.class-table?_id
      case
        # @.cs.class-table._id.formal-generics == 0
        if @.cs.class-table._id.is-expanded then
          type-stack.push(expanded(class-type(_id,[])))
        else
          type-stack.push(class-type(_id,[]))
        end
    otherwise
      error validity(VTUG, 201, 2);
      type-stack.push(class-type(ANY,[]))
    end
  otherwise
    error _id:validity(VTUG,199);
    type-stack.push(class-type(ANY,[]))
  end
end

/* validity VTUG p.201 (Unconstrained Genericity rule)
Let C be an unconstrained generic class. A Class-type CT having C
as base class is valid if and only if it satisfies the following
two conditions:
1. C is a generic class.
2. The number of Type components in CT's Actual-generics list is the same as the number of Formal-generic parameters in the Formal-generic-list of C's declaration. */

/* validity VTCG p.203 (Constrained Genericity rule)
Let C be a constrained generic class. A Class-type CT having C as base class is valid if and only if CT satisfies the Unconstrained Genericity rule (VTUG p.201) and, in addition:
3. For any Formal-generic parameter in the declaration of C having a constraint of the form \(-/> D\), the corresponding Type in the Actual-generic list of CT conforms to D. */

[two2]

vc-type \[\_id \_Actual-Generics \] = begin
  vc-id \[\_id \];
  vc-actual-generics \[\_Actual-Generics \];
  type-stack := class-type(_id, take-until type-mark @.type-stack)
    ++ tl (drop-until type-mark @.type-stack);
  case
    @.cc.formal-generics?formal(_id) | ~ @.cs.class-table?_id
      error (@.type-stack.top) : validity(VTUG, 199);
      type-stack.pop; type-stack.push(class-type(ANY,[]))
    otherwise
      sat-vgc(@.type-stack.top.actuals,
        @.cs.class-table._id.formal-generics);
      if @.cs.class-table._id.is-expanded then
        type-stack.update(1, expanded(@.type-stack.top))
  end
end

end

/* Satisfaction of VTCG */

[0] sat-vgc(_Type-seq, _Type-pair-seq) in C
  = sat-vgc(_Type-seq in C, _Type-pair-seq in C)

[0] sat-vgc([],[]) = id-context

[0] empty? _Type-seq =/= empty? _Type-pair-seq = true

==========================================================================
sat-vgc(_Type-seq, _Type-pair-seq) = begin
  error (@.type-stack.top) : validity(VTUG, 201, 2);
  type-stack.pop; type-stack.push(class-type(ANY,[]))
end

[0] empty? _Type-seq = false, empty? _Type-pair-seq = false

==========================================================================
sat-vgc(_Type-seq, _Type-pair-seq) = if ~ (hd _Type-seq) < (hd _Type-pair-seq).target then
  error (hd _Type-seq) : validity(VTUG, 203, 3);
  type-stack.pop; type-stack.push(class-type(ANY,[]))
else
  sat-vgc(tl _Type-seq, tl _Type-pair-seq)
end

/* validity VTEC p.209 (Expanded Type rule)
An expanded type of the form expanded CT, where CT is a Class-type of base class C, is valid if and only if it satisfies the following two conditions:
1. C is not a deferred class.
2. C either has no creation procedure, or has only one creation procedure with no argument.*/

```plaintext
[two03]
vctype [[ expanded _Class-Type ]]
= begin
  vc-type [[ _Class-Type ]];
  case
    0.cs.class-table.(0.type-stack.top.base).is-deferred
      error expanded(0.type-stack.top) : validity(VTEC, 209, 1);
      type-stack.pop; type-stack.push(class-type(ANY,[]))
  (~ empty? 0.cc.creators
    & ( | 0.cc.creators | == 1
      => ~ 0.cc.class-features.(first 0.cc.creators).signature
      == ([],[]) ))
    in (enter-class(0.type-stack.top.base) in 0)
      error expanded(0.type-stack.top) : validity(VTEC, 209, 2);
      type-stack.pop; type-stack.push(class-type(ANY,[]))
  otherwise
    type-stack.update(1, expanded(hd 0.type-stack))
  end
end

validity VTAT p.214
An anchored type of the form like anchor appearing in a class C is valid if and only if one of the following holds:
1. anchor is the final name of an attribute or function of C, whose declared type is a non-Anchored reference type.
2. The type appears in the text of a routine r of C, and anchor is a formal argument of r, whose declared type is a non-anchored reference type.
3. anchor is the reserved word Current */

[two04]
vctype [[ like _id ]]
= begin
  vc-id [[ _id ]];
  if ( 0.cc.class-features?._id
      & is-non-anchored-ref-type(0.cc.class-features
      ._id.signature.res)
    )
  | ( 0.phase == feature-bodies
      & 0.cf.formals?._id
      & is-non-anchored-ref-type(0.cf.formals._id)
    )
  | _id == CURRENT
    then
      type-stack.push(anchor(CURRENT))
    else
      error anchor(_id) : validity(VTAT, 214);

84
```
/* validity VTBT p.210 A Bit-type declaration is valid if and only if its Constant is of type Integer, and has a positive value. */

[vw05] vc-type [[ BIT _E-Int ]] = begin
  vc-const [[ _E-Int ]];
  if (hd @.type-stack) == class-type(lower-name(INTEGER),[])
    & 2int(_E-Int) > 0
    then
      type-stack.pop;
      type-stack.push(bit-type(2int(_E-Int)))
    else
      error validity(VTBT, 210);
      type-stack.pop;
      type-stack.push(bit-type(8)) -- A default value
    end
end

[vw06] vc-type [[ BIT _id ]] = begin
  vc-expr [[ _id ]];
  if (hd @.type-stack) == class-type(INTEGER,[])
    then
      type-stack.pop;
      /*
       type-stack.push(bit-type(@.value))
       not supported; value of constant should be obtained instead a default value is provided
       */
      error _id: "Warning: vc-type [[ BIT _id ]]";
      type-stack.push(bit-type(16))
    else
      error validity(VTBT, 210);
      type-stack.pop;
      type-stack.push(bit-type(8)) -- A default value
    end
end

% ACTUAL GENERICs

[vw08] vc-type-list [[ ] ]  = type-stack.push(type-mark)
[vw09] vc-type-list [[ _Type, _Type* ] ] = begin
  vc-type-list [[ _Type* ] ];
  vc-type  [[ _Type ]] end
end module Type-Validity

G.12 Formal Arguments and Local Entities

G.12.1 Formals-Locals-Syntax

module Formals-Locals-Syntax
imports Type-Syntax
exports
sorts FORMAL-ARGUMENTS ENTITY-DECLARATION-LIST
ENTITY-DECLARATION-GROUP ID-LIST TYPE-MARK
LOCAL-DECLARATIONS
context-free syntax
{ENTITY-DECLARATION-GROUP "";}* -> ENTITY-DECLARATION-LIST
ID-LIST TYPE-MARK -> ENTITY-DECLARATION-GROUP
{ID ",";}+ -> ID-LIST
":" TYPE -> TYPE-MARK
"(" ENTITY-DECLARATION-LIST ")" -> FORMAL-ARGUMENTS
-> LOCAL-DECLARATIONS
local ENTITY-DECLARATION-LIST -> LOCAL-DECLARATIONS
variables
Formal-arguments[0-9]* -> FORMAL-ARGUMENTS
Entity-decl-list[0-9]* -> ENTITY-DECLARATION-LIST
Entity-decl-group[0-9]* -> ENTITY-DECLARATION-GROUP
Entity-decl-group[/*] [0-9]* -> {ENTITY-DECLARATION-GROUP ";";}*
"_id" [0-9]* -> {ID ",";}*
"_id" [/*] [0-9]* -> {ID ",";}*
Type-mark[0-9]* -> TYPE-MARK
Locals[0-9]* -> LOCAL-DECLARATIONS
end module Formals-Locals-Syntax

G.12.2 Formals-Locals-Validity

module Formals-Locals-Validity
imports Formals-Locals-Syntax Type-Validity
exports
context-free syntax
vc-type-mark ""] TYPE-MARK "]" -> CONTEXT-UPDATE
vc-formals ""] FORMAL-ARGUMENTS "]" -> CONTEXT-UPDATE
vc-formals-decl ""] ENTITY-DECLARATION-LIST "]" -> CONTEXT-UPDATE
vc-formals-id-list ""] { ID ",";}* "]" -> CONTEXT-UPDATE
vc-locals ""] LOCAL-DECLARATIONS "]" -> CONTEXT-UPDATE
vc-locals-decl ""] ENTITY-DECLARATION-LIST "]" -> CONTEXT-UPDATE
vc-locals-id-list ""] { ID ",";}* "]" -> CONTEXT-UPDATE
equations
% TYPE-MARK
[type-mark] vc-type-mark [[ : _Type ]] = vc-type [[ _Type ]]
%
FORMALS
[formals]
vc-formals [[ ( Entity-decl-list ) ]]
= begin
vc-formals-decl [[ Entity-decl-list ]];
if signature.terms := take-until type-mark @.type-stack;
type-stack := tl drop-until type-mark @.type-stack
end
[formals-decl-1]
vc-formals-decl [[ ]]
= begin
cf.formals := [];
type-stack.push(type-mark)
end
[formals-decl-2]
vc-formals-decl [ _id+ ; Type; Entity-decl-group* ]
  = begin
    vc-formals-decl [ Entity-decl-group* ];
    vc-type [ _Type ];
    vc-formals-id-list [ _id+ ];
    type-stack.pop (1)
  end
[formals-id-list-0]
vc-formals-id-list [ ] = id-context
[formals-id-list-n]
vc-formals-id-list [ _id*, _id ]
  = begin
    vc-id [ _id ];
    require "@.cf.formals ? _id 
      else _id : validity(VREG p.110);
    case
      @.cc.class-features ? _id
        error _id : validity(VRFA p.110)
    otherwise
      cf.formals := @.cf.formals <- (_id, @.type-stack.top)
    end;
    type-stack.push(1);
    vc-formals-id-list [ _id* ]
  end

/* Formal Argument Rule validity VRFA p.110
Let fa be the Formal_arguments part of a routine r in
class C. Let formals be the concatenation of every
Identifier_list of every Entity_declaration_group in fa.
Then fa is valid if and only if no Identifier e appearing
in formals is the final name of a feature of C. */

/* validity VREG p.110
Let el be an Entity_declaration_list. Let identifiers
be the concatenation of every Identifier_list of every
Entity_declaration_group in fa. Then el is valid if and
only if no Identifier appears more than once in the list
identifiers. */

%% LOCALS
[locals-0]
vc-locals [ ] = cf.locals := []
[locals-n]
vc-locals [ local Entity-decl-list ]
  = vc-locals-decl [ Entity-decl-list ]
[locals-decl-0]
vc-locals-decl [ ] = cf.locals := []
[locals-decl-n]
vc-locals-decl [ _id+ ; Type; Entity-decl-group* ]
  = begin
    vc-locals-decl [ Entity-decl-group* ];
    vc-type [ _Type ];
    vc-locals-id-list [ _id+ ];
    type-stack.pop
  end
[locals-id-list-0]

87
vc-locals-id-list |[]| = id-context
[locals-id-list-n]
vc-locals-id-list |[ _id, _id* ]|
    = begin
        vc-locals-id-list |[ _id* ]|
        require ~ @.cf.locals?_id
        else _id:validity(VREG, 110);
        case
            @.cc.class-features?_id
                error _id:validity(VRLE, 115, 1)
            @.cf.formals?_id
                error _id:validity(VRLE, 115, 2)
            otherwise
                cf.locals := (_id, hd @.type-stack) -> @.cflocals
        end
    end

/* Local Entity Rule, validity VRLE p.115
Let ld be the Local_declarations part of a routine r
in a class C. Let locals be the concatenation of every
Identifier_list of every Entity_declaration_group in ld.
Then ld is valid if and only if every Identifier e in ld
satisfies the following two conditions:
1. No feature of C has e as its final name
2. No formal argument of r has e as its Identifier. */
end module Formals-Locals-Validity

G.13 Expressions

G.13.1 Expression-Syntax

module Expression-Syntax
    imports Constant-Syntax Identifier-Syntax
    exports
        sorts EXPR BOOLEAN-EXPR CALL UNQUALIFIED-CALL ACTUALS ACTUAL
        context-free syntax
        \%( EXPR \% -> BOOLEAN-EXPR
        \% -> ACTUALS
        \%( {ACTUAL \%,\%}* \%) \% -> ACTUALS
        \%( EXP % \%) -> ACTUAL
        \%( EXP % \%) \% -> ACTUAL
        ID ACTUALS -> UNQUALIFIED-CALL
        UNQUALIFIED-CALL -> CALL
        \%( EXP \% \%) -> UNQUALIFIED-CALL \% -> CALL
        \%( CALL \% \%) \% -> CALL
        \%( EXP \% \%) \% -> EXP {bracket}
        CALL \% -> EXP
        \%( {EXP \%,\%}* \%) \% -> EXP
        old EXP \% -> EXP
        strip\%( {ID \%,\%}* \%) \% -> EXP
        MANIFEST-CONSTANT \% -> EXP
        \%( EXP \% \%) \% -> EXP
        88
EXPR "/\=" EXPR -> EXPR

not EXPR -> EXPR

"+" EXPR -> EXPR

"-" EXPR -> EXPR

EXPR "+" EXPR -> EXPR {left}
EXPR "-" EXPR -> EXPR {left}
EXPR "/" EXPR -> EXPR {left}
EXPR ">" EXPR -> EXPR {non-assoc}
EXPR "<" EXPR -> EXPR {non-assoc}
EXPR ">=\" EXPR -> EXPR {non-assoc}
EXPR "/\=" EXPR -> EXPR {non-assoc}
EXPR ">/\=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR "/=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-assoc}
EXPR ">=" EXPR -> EXPR {non-asso
vc-bool-expr "\[" BOOLEAN-EXPRESSION "\]" -> CONTEXT-UPDATE
vc-expr "\[" EXPRESSION "\]" -> CONTEXT-UPDATE
vc-call "\[" CALL "\]" -> CONTEXT-UPDATE
vc-unqualified "\[" UNQUALIFIED "\]" E-TYPE -> CONTEXT-UPDATE
vc-actuals "\[" ACTUALS "\]" -> CONTEXT-UPDATE
vc-actual-list "\[" \{ACTUAL ";"\}* "\]" -> CONTEXT-UPDATE
vc-manifest-array "\[" MANIFEST-ARRAY "\]" -> CONTEXT-UPDATE
vc-manifest-array[1] "\[" MANIFEST-ARRAY "\]" -> CONTEXT-UPDATE
vc-old "\[" OLD "\]" -> CONTEXT-UPDATE
vc-strip "\[" STRIP "\]" -> CONTEXT-UPDATE

equations
[bool-expr]
vc-bool-expr [[ _expr ]]
= begin
vc-expr [[ _expr ]];
require @.type-stack.top == class-type(boolean,[])  
else validity(VWBE, 374)
end

call-1]
vccall [[ current ]| _E-Type' = type-stack.push(@.cc.type)
[call-2]

_id != current

==================================================

call-1]
vccall [[ _id ]]
= begin
vc-id [[ _id ]];
case
@.cf.locals?_id
  type-stack.push(@.cf.locals._id.type)
@.cf.formals?_id
  type-stack.push(@.cf.formals._id.type)
otherswise
  vc-unqualified [[ _id ]| @.cc.type
end
end

call-3] vc-call [[ _id_Actuals ]| = vc-unqualified [[ _id_Actuals ]| @.cc.type
[call-4]
vc-call [[ (_expr). _Unqualified ]]
= begin
vc-expr [[ _expr ]];
if @.type-stack.top == void then
  error validity(VXXX, 999)
else
  vc-unqualified [[ _Unqualified ]| @.type-stack.top
end
end

call-5]
vccall [[ _Call._Unqualified ]]
= begin
vc-call [[ _Call ]];
if @.type-stack.top == void then
  error validity(VXXX, 999)
else
  vc-unqualified [[ _Unqualified ]| @.type-stack.top
end
end

[unqualified]
vc-unqualified [[ _id _Actuals ]] _E-Type’ in C

begin
  vc-id [[ _id ]];
  vc-actuals [[ _Actuals ]];
  case
    _CC.class-features?._id
      require (# _Signature.args) == (# take-until type-mark @.type-stack)
        else validity(VUAR, 369, 1);
      require (take-until type-mark @.type-stack) < (_Signature.args)
        else validity(VUAR, 369, 2);
      require "empty"? {any, _E-Type.base} & _CC.class-features._id.clients
        else validity(VUAR, 369, 2);
      type-stack := tl (drop-until type-mark @.type-stack);
      if (# _Signature.res) == 0 then
        type-stack.push(void)
      else
        type-stack.push(_Signature.res!1)
      end
    otherwise
      error validity(VUAR, 369, 2);
      type-stack := tl (drop-until type-mark @.type-stack);
      type-stack.push(class-type(any,[]))
    end
  end

[actuals-0] vc-actuals [[ ]] = type-stack.push(type-mark)
[actuals-n] vc-actuals [[ _Actual* ]] = vc-actual-list[[ _Actual* ]]

[act-list-0] vc-actual-list [[ ]] = type-stack.push(type-mark)
[act-list-n1]
  vc-actual-list [[ _expr, _Actual* ]]
  = begin
    vc-actual-list [[ _Actual* ]];
    vc-expr [[ _expr ]]
  end
[act-list-n2]
  vc-actual-list [[ $ _id, _Actual* ]]
  = begin
    vc-actual-list [[ _Actual* ]];
    require @.cc.class-features?._id & "@.cc.class-feature.is-const-attribute"
      else validity(VUAR, 369, 4);
    type-stack.push(class-type(any,[])); -- type of $ _id ?;
    error _id : "WARNING: ID not typechecked; vc-actual-list"
  end

[manifest-array]
  vc-manifest-array [[ _Manifest-Array ]]
  = begin
    error "WARNING: manifest arrays not fully checked";
    vc-manifest-array1 [[ _Manifest-Array ]];
    type-stack.update(1, class-type(array, [0.type-stack.top]))
  end
[manifest-array-0]
vc-manifest-array1 [[ << >> ]]
= begin
  type-stack.push(class-type(none, []));
  -- look at validity rule VWMA at page 393;
  -- one has to find a suitable type T
end

[manifest-array-n]
vc-manifest-array1 [[ << _expr, _expr* >> ]]
= begin
  vc-expr [[ _expr ]];
  type-stack.push(type-mark);
  vc-manifest-array1 [[ << _expr* >> ]];
  type-stack := tl (drop-until type-mark @.type-stack)
end

[old]
vc-old [[ old _expr ]]
= begin
  error "WARNING: validity of old not checked";
  vc-expr [[ _expr ]]
end

/* validity VAOL p.124
An Old expression of the form old e, where e is an
expression of type TE, is valid if and only if it
satisfies the following two conditions:
1. It appears in a Postcondition clause of a Routine r.
2. Transforming r into a function with result type TE
(by adding a result type if r is procedure, or
changing its result type if it is already a function)
and replacing its entire Routine part by
  do Result:= e end
would yield a valid routine. */

[strip] vc-strip1 [[ strip(_id*) ]]
= begin
  error "WARNING: validity of old not checked";
  type-stack.push(class-type(any, []))
end

%% EXPRESSIONS
[expr-01] vc-expr1 [[ _Call ]]
        = vc-call [[ _Call ]]
[expr-02] vc-expr1 [[ _Const ]]
        = vc-const [[ _Const ]]
[expr-03] vc-expr1 [[ _Manifest-Array ]]
        = vc-manifest-array [[ _Manifest-Array ]]
[expr-04] vc-expr1 [[ _Old ]]
        = vc-old [[ _Old ]]
[expr-05] vc-expr1 [[ _Strip ]]
        = vc-strip1 [[ _Strip ]]
[expr-06]
  vc-expr1 [[ _expr1 = _expr2 ]]
  = begin
    vc-expr1 [[ _expr1 ]];
    vc-expr1 [[ _expr2 ]];
    require @.type-stack!1 < @.type-stack!2 | @.type-stack!2 < @.type-stack!1
    else validity(VWEQ, 375);
    type-stack.pop; type-stack.pop;
    type-stack.push(class-type(boolean, []))
  end
[expr-07] vc-expr1 [[ _expr1 /= _expr2 ]]
        = vc-expr1 [[ _expr1 = _expr2 ]]

92
/* Operators validity VWUE p.379

An Operator Expression is valid if and only if its equivalent dot form is a valid call.

We translate operator expressions to their equivalent dot form.
The equivalent name is given by mappings 'unary: UNARY -> ID' and 'binary: BINARY -> ID'. */

%% Unary Operators
[expr-08] vc-expri [not _expr1] = vc-expri [._expr1.unary(not) ]
[expr-09] vc-expri [_+ _expr_1] = vc-expri [._expr1.unary(+) ]
[expr-10] vc-expri [_-_expr1] = vc-expri [._expr1.unary(-) ]

%% Binary Operators
[expr-12] vc-expri [._expr1 Free-op _expr2] = vc-expri [._expr1.binary(Free-op) (_expr2)]
[expr-13] vc-expri [._expr1 + _expr2] = vc-expri [._expr1.binary(+) (_expr2)]
[expr-14] vc-expri [._expr1. - _expr2] = vc-expri [._expr1.binary(-) (_expr2)]
[expr-16] vc-expri [._expr1 / _expr2] = vc-expri [._expr1.binary(/) (_expr2)]
[expr-17] vc-expri [._expr1 < _expr2] = vc-expri [._expr1.binary(<) (_expr2)]
[expr-18] vc-expri [._expr1 > _expr2] = vc-expri [._expr1.binary(>) (_expr2)]
[expr-19] vc-expri [._expr1 <= _expr2] = vc-expri [._expr1.binary(<=) (_expr2)]
[expr-20] vc-expri [._expr1 >= _expr2] = vc-expri [._expr1.binary(>=) (_expr2)]
[expr-21] vc-expri [._expr1 || _expr2] = vc-expri [._expr1.binary(||) (_expr2)]
[expr-22] vc-expri [._expr1 \_expr2] = vc-expri [._expr1.binary(\_expr2)]
[expr-23] vc-expri [._expr1 ^ _expr2] = vc-expri [._expr1.binary(^) (_expr2)]
[expr-24] vc-expri [._expr1 and _expr2] = vc-expri [._expr1.binary(and) (_expr2)]
[expr-25] vc-expri [._expr1 or _expr2] = vc-expri [._expr1.binary(or) (_expr2)]
[expr-26] vc-expri [._expr1 xor _expr2] = vc-expri [._expr1.binary(xor) (_expr2)]
[expr-27] vc-expri [._expr1 and then _expr2] = vc-expri [._expr1.binary(and then) (_expr2)]
[expr-28] vc-expri [._expr1 or else _expr2] = vc-expri [._expr1.binary(or else) (_expr2)]
[expr-29] vc-expri [._expr1 implies _expr2] = vc-expri [._expr1.binary(implies) (_expr2)]

end module Expression-Validity

G.14 Assertions

G.14.1 Assertion-Syntax

module Assertion-Syntax
%% [Mey92]: pp.120, 130
imports Expression-Syntax
exports
sorts
PRECONDITION POSTCONDITION INVARIANT ASSERTION ASSERTION-CLAUSE
UNLABELED-ASSERTION-CLAUSE TAG-MARK TAG VARIANT
context-free syntax
require ASSERTION -> PRECONDITION
require else ASSERTION -> PRECONDITION
ensure ASSERTION -> POSTCONDITION
ensure then ASSERTION -> POSTCONDITION
ensure then ASSERTION -> INVARIANT

93
G.14.2 Assertion-Validity

module Assertion-Validity
imports Expression-Validity
exports
context-free syntax
vc-precondition "[\[ PRECONDITION \]]" -> CONTEXT-UPDATE
vc-postcondition "[\[ POSTCONDITION \]]" -> CONTEXT-UPDATE
vc-invariant "[\[ INARIANT \]]" -> CONTEXT-UPDATE
vc-assertion "[\[ ASSERTION \]]" -> CONTEXT-UPDATE
vc-variant "[\[ VARIANT \]]" -> CONTEXT-UPDATE

equations
validity WAPE p.122
A Precondition of a routine r of a class C is valid
if and only if every feature whose final name appears
in any Assertion_clause is available to every class
to which r is available.

/*

/* Some extra work has to be done (extra flags in Context)
to check <old EXPR> */

[precondition-1] vc-precondition [[ ]] = id-context
[precondition-2]
vcc-precondition [[ require Assertion ]]
  = begin
    --in(precondition);
    vc-assertion(Assertion);
    --out(precondition);

end module Assertion-Syntax
end

[precondition-3]
vc-precondition |[ require else Assertion ]|
= begin
  --in(precondition);
  vc-assertion(Assertion);
  --out(precondition);
end

[postcondition-1] vc-postcondition |[ ]| = id-context
[postcondition-2]
vc-postcondition |[ ensure Assertion ]|
= begin
  --in(postcondition);
  vc-assertion(Assertion);
  --out(postcondition);
end
[postcondition-3]
vc-postcondition |[ ensure then Assertion ]|
= begin
  --in(postcondition);
  vc-assertion(Assertion);
  --out(postcondition);
end

[invariant-1] vc-invariant |[ ]| = begin end
[invariant-2]
vc-invariant |[ invariant Assertion ]|
= begin
  --in(invariant);
  vc-assertion(Assertion);
  --out(invariant);
end

[assertion-1] vc-assertion |[ ]| = id-context
[assertion-2]
vc-assertion |[ Tag-mark Unlabeled-Assertion; Assertion-clause* ]|
= begin
  vc-unlabeled-assertion |[ Unlabeled-Assertion ]|;
  vc-assertion |[ Assertion-clause* ]|
end

[unl-assertion-1] vc-unlabeled-assertion |[ % % Comment % % ]| = id-context

[variant-1] vc-variant |[ ]| = id-context
[variant-2]
vc-variant |[ variant Tag-mark _expr ]|
= begin
  vc-expr |[ _expr ]|;
  require 0.typestack!1 == class-type(integer, [])
else "Syntactical constraint p.130"
  end
end module Assertion-Validity
G.15 Assignment

G.15.1 Assignment-Syntax

module Assignment-Syntax
imports Expression-Syntax Type-Syntax
exports
sorts ASSIGNMENT REVERSE-ASSIGNMENT-ATTEMPT CREATION CALL OPT-TYPE
context-free syntax
ID "=" EXPR -> ASSIGNMENT
ID "?=" EXPR -> REVERSE-ASSIGNMENT-ATTEMPT
"!" OPT-TYPE "!" ID CREATION-CALL -> CREATION
"." UNQUALIFIED-CALL -> CREATION-CALL
TYPE -> OPT-TYPE
variables
Assignment[0-9]* -> ASSIGNMENT
Rev-Assignment[0-9]* -> REVERSE-ASSIGNMENT-ATTEMPT
Creation[0-9]* -> CREATION
end module Assignment-Syntax

G.15.2 Assignment-Validity

module Assignment-Validity
imports Assignment-Syntax Expression-Validity Type-Validity
exports
context-free syntax
vc-assignment "[" ASSIGNMENT "]" -> CONTEXT-UPDATE
vc-reverse-assignment "[" REVERSE-ASSIGNMENT-ATTEMPT "]" -> CONTEXT-UPDATE
vc-creation "[" CREATION "]" -> CONTEXT-UPDATE
vc-writable "[" ID "]" -> CONTEXT-UPDATE
vc-creation-call "[" CREATION-CALL "]" -> CONTEXT-UPDATE
equations
[assignment]
vc-assignment "[ _id := _expr ]" = begin
vc-lhs "[ _id ]";
vc-expr "[ _expr ]";
require $@.type-stack!1 < $@.type-stack!2
else validity(VIAR, 311);
type-stack.pop(2)
end
[rev-assigment]
vc-reverse-assignment "[ _id := _expr ]" = begin
vc-writable "[ _id ]";
vc-expr "[ _expr ]";
require $@.type-stack!2 < $@.type-stack!1
else validity(VJRV, 332);
type-stack.pop(2)
end
[lhs-1]
vc-writable | [ result ] |
= case
  # @.cf.signature.res == 0
  type-stack.push(void);
  error result: "Used in procedure"
  otherwise
  type-stack.push(@.cf.signature.res!1)
end

[lhs-2]
_id != result
===============================================================================
vc-writable | [ _id ] |
= begin
  vc-id | [ _id ] |
  case
    @.cc.class-features?._id & @.cc.class-features._id.is-attribute
      type-stack.push(@.cc.class-features._id.signature.res!1)
    @.cf.locals?._id
      type-stack.push(@.cf.locals._id.type)
    otherwise
      error "Syntactical constraint p.276"
      type-stack.push(class-type(any, []))
  end
end

[creation-1]
vc-creation | [ !! _id ] |
= begin
  error "WARNING: Check System-Level validity of creation instructions"
end

[creation-2]
vc-creation | [ !! _id_Unclassified ] |
= begin
  error "WARNING: Check System-Level validity of creation instructions"
end

[creation-3]
vc-creation | [ ._Type! _id_Creation-Call] |
= begin
  vc-writable | [ _id ] |
  vc-type | [ _Type ] |
  if is-formal(@.type-stack.top) then
    error validity(VGCC, 286, 1)
  else
    require @.cs.class-table.(@.type-stack.top.base).is-effective
    else validity(VGCC, 286, 2);
    require @.type-stack!1 < @.type-stack!2
    else validity(VGCC, 286, 3);
  end;
  type-stack.pop(2)
end

[creation-call-1]
vc-creation-call | [ ] |
= require empty? @.cs.class-table.(@.type-stack.top.base).creators
  else validity(VGCC, 286, 4)
[creation-call-2]
vc-creation-call | [ _id_Actuals ] |
= begin
if empty? @.cs.class-table.(@.type-stack.top.base).creators error validity(VGCC, 286, 5)
else
  vc-unqualified |[ _Unqualified ]| @.cs.class-table.(@.type-stack.top.base).type;
  require @.cs.class-table.(@.type-stack.top.base).class-features._id.is-procedure
  & " @.cs.class-table.(@.type-stack.top.base).class-features._id.is-once
  & " empty? (@.cc.class-name) &
  @.cs.class-table.(@.type-stack.top.base).class-features._id.clients)
  else validity(VGCC, 286, 6)
end
end module Assignment-Validity

G.16 Instructions

G.16.1 Instruction-Syntax

module Instruction-Syntax
imports Assertion-Syntax Assignment-Syntax
exports
  sorts COMPOUND INSTRUCTION CONDITIONAL THEN-PART-LIST
  THEN-PART ELSE-PART MULTI-BRANCH WHEN-PART-LIST
  WHEN-PART CHOICES CHOICE INTERVAL INTEGER-INTERVAL
  CHARACTER-INTERVAL LOOP INITIALIZATION LOOP-BODY
  EXIT DEBUG DEBUG-KEYS DEBUG-KEY-LIST DEBUG-KEY
  RESCUE RETRY CHECK
context-free syntax
  {INSTRUCTION ";";}* -> COMPOUND
  % In the definition the ";" is optional. Not in this specification
  % p.234

  % NULL % -> INSTRUCTION
  CREATION -> INSTRUCTION
  CALL -> INSTRUCTION
  ASSIGNMENT -> INSTRUCTION
  REVERSE-ASSIGNMENT-ATTEMPT -> INSTRUCTION
  CONDITIONAL -> INSTRUCTION
  MULTI-BRANCH -> INSTRUCTION
  LOOP -> INSTRUCTION
  DEBUG -> INSTRUCTION
  CHECK -> INSTRUCTION
  RETRY -> INSTRUCTION
  if THEN-PART-LIST ELSE-PART end -> CONDITIONAL
  {THEN-PART "elsif";}* -> THEN-PART-LIST
  BOOLEAN-EXPR then COMPOUND -> THEN-PART
  -> ELSE-PART
  else COMPOUND -> ELSE-PART
  inspect EXPR WHEN-PART-LIST ELSE-PART end -> MULTI-BRANCH
  WHEN-PART* -> WHEN-PART-LIST
  when CHOICES then COMPOUND -> WHEN-PART
  {CHOICE ",";}* -> CHOICES
  CONSTANT -> CHOICE
  INTERVAL -> INTERVAL
  INTEGER-INTERVAL -> INTERVAL
  CHARACTER-INTERVAL -> INTERVAL
  INTEGER-CONSTANT ".." INTEGER-CONSTANT -> INTEGER-INTERVAL
CHARACTER-CONSTANT "." CHARACTER-CONSTANT -> CHARACTER-INTERVAL

INITIALIZATION INVARIANT VARIANT LOOP-BODY end -> LOOP
from COMPOUND -> INITIALIZATION
EXIT Loop COMPOUND -> LOOP-BODY
until BOOLEAN-EXPR -> EXIT

debug DEBUG-KEYS COMPOUND end -> DEBUG
"(" DEBUG-KEY-LIST ")" -> DEBUG-KEYS
{DEBUG-KEY ","}* -> DEBUG-KEY-LIST
MANIFEST-STRING -> DEBUG-KEY

check ASSERTION end -> CHECK
retry
rescue COMPOUND -> RESCUE

variables
Compound[0-9]* -> COMPOUND Instruction[0-9]* -> INSTRUCTION
Instruction[+][0-9]* -> COMPOUND Conditional[0-9]* -> CONDITIONAL
Then-part[+][0-9]* -> THEN-PART-LIST Then-part+[0-9]* -> THEN-PART
Else-part[0-9]* -> ELSE-PART When-part[+][0-9]* -> WHEN-PART* Multi-branch[0-9]* -> MULTI-BRANCH Choice[0-9]* -> CHOICE
Interval[0-9]* -> INTERVAL When-part[+][0-9]* -> WHEN-PART-LIST
Loop[0-9]* -> LOOP Char-interval[0-9]* -> CHARACTER-INTERVAL
Loop-body[0-9]* -> LOOP-BODY Exit[0-9]* -> EXIT
Debug[0-9]* -> DEBUG Loop-init[0-9]* -> INITIALIZATION
Debug-key[*][0-9]* -> DEBUG-KEY-LIST Debug-key[0-9]* -> DEBUG-KEY
Check[0-9]* -> CHECK
end module Instruction-Syntax

G.16.2 Instruction-Validity

module Instruction-Validity
imports Instruction-Syntax Assignment-Validity
exports
context-free syntax
vc-comp "|" COMPOUND "|" -> CONTEXT-UPDATE
vc-instr "|" INSTRUCTION "|" -> CONTEXT-UPDATE
vc-then-part-list "|" {THEN-PART "elsif"}* "|" -> CONTEXT-UPDATE
vc-then-part "|" THEN-PART "|" -> CONTEXT-UPDATE
vc-else-part "|" ELSE-PART "|" -> CONTEXT-UPDATE
vc-when-part-list "|" WHEN-PART-LIST "|" -> CONTEXT-UPDATE
vc-choices "|" CHOICES "|" -> CONTEXT-UPDATE
vc-rescue "|" RESCUE "|" -> CONTEXT-UPDATE

equations
[comp-0] vc-comp |[]| = id-context
[comp-n] vc-comp |[ Instruction; Compound ]| = begin vc-instr |[ Instruction ]|; vc-comp |[ Compound ]| end
[instr-1] vc-instr |[ % % NULL % % ]| = id-context
[instr-2] vc-instr |[ Creation ]| = vc-creation |[ Creation ]|
[instr-3] vc-instr |[ _Assignment ]| = vc-assignment |[ _Assignment ]|
[instr-5]
vc-instr [[_Call]]
= begin
  vc-call [[_Call]];
  require @.type-stack.top == void
  else "Functional call used as instruction (which validity rule?)";
  type-stack.pop
end

[cond]
vc-instr [[ if Then-part-list Else-part end ]]
= begin
  vc-then-part-list [[ Then-part-list ]];
  vc-else-part [[ Else-part ]]
end

[then-part-list-1]
vc-then-part-list [[ ]] = id-context
[then-part-list-2]
vc-then-part-list [[ Then-part elsif Then-part* ]] = begin
  vc-then-part [[ Then-part ]];
  vc-then-part-list [[ Then-part* ]]
end

[then-part]
vc-then-part [[ Boolean-expression then Compound ]]
= begin
  vc-bool-expr [[ Boolean-expression ]];
  vc-comp [[ Compound ]]
end

[else-part-1] vc-else-part [[ ]] = id-context
[else-part-2] vc-else-part [[ else Compound ]] = vc-comp [[ Compound ]]

[multi-branch]
vc-instr [[ inspect _expr When-part-list Else-part end ]]
= begin
  error "Warning: Multibranch not yet specified (see MULTI-BRANCH RULE (validity VOMB p.239))"
  vc-expr [[ _expr ]];
  vc-when-part-list [[ When-part-list ]];
  vc-else-part [[ Else-part ]]
end

[when-part-list-0]
vc-when-part-list [[ ]] = id-context
end

[when-part-list-n]
vc-when-part-list [[ when then Choice* then Compound When-Part* ]] = begin
  vc-choices [[ Choice* ]];
  vc-comp [[ Compound ]];
  vc-when-part-list [[ When-Part* ]]
end

[choices] vc-choices [[ Choice* ]] = id-context

[loop]
vc-instr [[ from Compound1 Invariant Variant until _expr loop Compound2 end ]]
= begin
  vc-comp [[ Compound1 ]];
  vc-invariant [[ Invariant ]];
  vc-variant [[ Variant ]];
  vc-bool-expr [[ _expr ]];
vc-comp |[ Compound2 ]|
end

[debug]
vc-instr |[ debug Debug-keys Compound end ]| = vc-comp |[ Compound ]|

[check]
vc-instr |[ check Assertion end ]|
= begin
  -- in(check);
  vc-assertion |[ Assertion ]|
end

[retry]
vc-instr |[ retry ]|
= begin
  -- require @.in == rescue
  -- else validity(VXRT, 256)
end

[rescue]
vc-rescue |[ rescue Compound ]|
= begin
  -- in(rescue)
  vc-comp |[ Compound ]|
end
end module Instruction-Validity

G.17 External
G.17.1 External-Syntax

module External-Syntax
imports String-Char-Syntax
exports
  sorts EXTERNAL LANGUAGE-NAME EXTERNAL-NAME
  context-free syntax
    external LANGUAGE-NAME EXTERNAL-NAME -> EXTERNAL
    MANIFEST-STRING -> LANGUAGE-NAME
    -> EXTERNAL-NAME
    alias MANIFEST-STRING -> EXTERNAL-NAME
variables
  External -> EXTERNAL

end module External-Syntax

G.17.2 External-Validity

module External-Validity
imports External-Syntax String-Validity
exports
  context-free syntax
    vc-external "" |[" EXTERNAL ""]| -> CONTEXT-UPDATE
equations
  [0] vc-external |[ External ]| = error "WARNING: external not checked"
end module External-Validity
G.18 Obsolete Messages

module Obsolete-Syntax
  imports String-Char-Syntax
  exports
    sorts OBSOLETE MESSAGE
    context-free syntax
      obsolete MESSAGE -> OBSOLETE
      MANIFEST-STRING -> MESSAGE
    variables
      Obsolete[0-9]* -> OBSOLETE
  end module Obsolete-Syntax

G.19 Routines

G.19.1 Routine-Syntax

module Routine-Syntax
  imports Instruction-Syntax External-Syntax Formals-Locals-Syntax Obsolete-Syntax
  exports
    sorts ROUTINE ROUTINE-BODY EFFECTIVE INTERNAL ROUTINE-MARK DEFERRED
    context-free syntax
      OBSOLETE PRECONDITION deferred POSTCONDITION end -> ROUTINE
      OBSOLETE PRECONDITION EXTERNAL POSTCONDITION end -> ROUTINE
      OBSOLETE PRECONDITION LOCAL-DECLARATIONS INTERNAL POSTCONDITION RESCUE end -> ROUTINE
      ROUTINE-MARK COMPOUND -> INTERNAL
      do -> ROUTINE-MARK
      once -> ROUTINE-MARK
    variables
      Routine[0-9]* -> ROUTINE
      Routine-body[0-9]* -> ROUTINE-BODY
      Effective[0-9]* -> EFFECTIVE
      Internal[0-9]* -> INTERNAL
      Routine-mark[0-9]* -> ROUTINE-MARK
      Deferred[0-9]* -> DEFERRED
  end module Routine-Syntax

G.19.2 Routine-Validity

module Routine-Validity
  imports Routine-Syntax Instruction-Validity
  exports
    context-free syntax
      vc-routine "\[" ROUTINE \"]\" -> CONTEXT-UPDATE
      vc-internal "\[" INTERNAL \"]\" -> CONTEXT-UPDATE
  equations
    /* Routine Rule validity VRRR p.113
    A Routine part of a routine declaration is valid if and only if one of the following conditions holds:
    1. Its Routine_body is an Internal body (beginning with do or once).
    2. In the other cases (where Routine_body is External or deferred), there is neither a Local_declarations

102
Satisfied by syntax; a program that violates the Routine Rule is not syntactically correct (see Routine-Syntax). This implies that the syntax is less permissive than the official Eiffel syntax.

```
/*//
[routine-deferred]
vc-routine [[ Obsolete Precondition deferred Postcondition end ]] = case
  feature-signatures == @.pass
  cc.check-deferred := true;
  cf.is-deferred := true;
  feature-bodies == @.pass
  vc-precondition [[ Precondition ]];
  vc-postcondition [[ Postcondition ]];
end
[routine-external]
vc-routine [[ Obsolete Precondition External Postcondition end ]] = case
  feature-signatures == @.pass
  cf.is-external := true;
  feature-bodies == @.pass
  vc-precondition [[ Precondition ]];
  vc-postcondition [[ Postcondition ]];
end
[routine-internal]
vc-routine [[ Obsolete Precondition Locals Internal Postcondition Rescue end ]] = case
  feature-signatures == @.pass
  cf.is-internal := true;
  vc-locals [[ Locals ]];
  feature-bodies == @.pass
  vc-internal [[ Internal ]];
  vc-precondition [[ Precondition ]];
  vc-postcondition [[ Postcondition ]];
  vc-rescue [[ Rescue ]]
end

[internal]
vcc-internal [[ do Compound ]] -- effective, internal
  = begin -- in(do);
  vc-compound [[ Compound ]]
end
[internal] vc-internal [[ once Compound ]] -- effective, internal
  = begin -- in(once);
  vc-compound [[ Compound ]]
end
end module Routine-Validity
```

G.20 Features

G.20.1 Feature-Syntax

module Feature-Syntax
  imports Feature-Name-Syntax Routine-Syntax
  exports
sorts FEATURE-DECLARATION DECLARATION-BODY
context-free syntax
   NEW-FEATURE-LIST DECLARATION-BODY -> FEATURE-DECLARATION
   TYPE-MARK is ROUTINE -> DECLARATION-BODY
   TYPE-MARK is ROUTINE -> DECLARATION-BODY
   TYPE-MARK is ROUTINE -> DECLARATION-BODY
   TYPE-MARK is MANIFEST-CONSTANT -> DECLARATION-BODY
   TYPE-MARK is unique -> DECLARATION-BODY
   TYPE-MARK is ROUTINE -> DECLARATION-BODY
   TYPE-MARK is ROUTINE -> DECLARATION-BODY

variables
   Feature[0-9]* -> FEATURE-DECLARATION
   Decl-Body[0-9]* -> DECLARATION-BODY

end module Feature-Syntax

G.20.2 Feature-Validity

module Feature-Validity
   imports Feature-Name-Validity Routine-Validity
   exports
   context-free syntax
      vc-feature "[" FEATURE-DECLARATION "]" -> CONTEXT-UPDATE
      vc-feature1 "[" FEATURE-DECLARATION "]" -> CONTEXT-UPDATE
      vc-decl-body "[" DECLARATION-BODY "]" -> CONTEXT-UPDATE
      vc-const-routine "[" CONSTANT-OR-ROUTINE "]" -> CONTEXT-UPDATE

   equations
      [feature-1]
      vc-feature1 [[ New-feature, New-Feature+ Decl-Body ]] = begin
         vc-feature1 [[ New-feature Decl-Body ]];
         vc-feature1 [[ New-Feature+ Decl-Body ]];
      end
      [feature-2]
      vc-feature [[ Feature-Name Decl-Body ]] = vc-feature1 [[ normal(Feature-Name) Decl-Body ]] 
      [feature-3]
      vc-feature1 [[ frozen Feature-Name Decl-Body ]] = vc-feature1 [[ frozen normal(Feature-Name) Decl-Body ]] 
      [feature-4]
      vc-feature1 [[ frozen _id Decl-Body ]] = begin
         enter-feature(_id);
         if @.pass == feature-signatures then cf.is-frozen := true end;
         vc-feature1 [[ _id Decl-Body ]];
      end
      [feature-5]
      vc-feature1 [[ _id Decl-Body ]] = begin
         enter-feature(_id);
         vc-feature1 [[ _id Decl-Body ]];
      end
      [feature1]
      vc-feature1 [[ _id Decl-Body ]] = case

104
feature-signatures == @.pass

case
  "@.cc.feature-table ? Feature.name
   -- add origin and seed
   vc-decl-body [[] Decl-body ]]
   leave-feature;
@.cf.origin == @.cc.class-name
/* Same name used twice for a direct feature in this class */
error _id/: validity(VFFD, 69, 2)
otherwise
  /* require origin of last redeclaration /== @.cc.name
   else -- p.163 ?
   Check the redeclaration as immediate feature and compare
   results with old entry by the Redeclaration Rule p.163
   This means saving the current entry and cleaning up the
   old one; remembering to later check the RR.
   */
end

feature-bodies == @.pass

vc-decl-body [[] Decl-body ]];
require @.cf.is-var-attribute | @.cf.is-const-attribute
 | @.cf.is-procedure | @.cf.is-function
else validity(VFFD, 69, 1);
require @.cf.is-frozen --> "@.cf.is-deferred
else validity(VFFD, 69, 4);
require @.cf.feature-name.is-prefix
   --> (@.cf.signature.args) == 0
       & (@.cf.is-attribute | @.cf.is-function)
else validity(VFFD, 69, 5);
require @.cf.feature-name.is-infix
   --> (@.cf.signature.args) == 1 & @.cf.is-function
else validity(VFFD, 69, 6);
require @.cf.is-once --> "@.cc.generics?(@.cf.signature.res!1)
else validity(VFFD, 69, 7);

-- Redeclaration Rule ?
leave-feature;
end
end -- vc-feature

[decl-body-manifest-const]
vc-decl-body [[] Type-Mark is _Manifest-Const ]]
  = case
    feature-signatures == @.pass
    cf.is-const-attribute := true;
    vc-type-mark [[] Type-mark ]];
    cf.signature.res := @.type-stack.top;
    vc-const [[] _Manifest-Const ]];
    require @.type-stack!1 < @.type-stack!2
    else validity(VXXX, 999);
    type-stack.pop (2);
    feature-bodies == @.pass
end

[decl-body-unique]
vc-decl-body [[] Type-Mark is unique ]]
  = case
feature-signatures == @.pass
  cf.is-unique := true;
  -- assign unique value
  vc-type-mark [[ Type-mark ]];
  require @.type-stack.top == class-type(integer,[])
    else validity(VXXX, 999)
  cf.signature.res := [class-type(integer, [])];
  type-stack.pop
feature-bodies == @.pass
end
[decl-body-attribute]
vc-decl-body [[ Type-Mark ]]
= case
  feature-signatures == @.pass
    cf.is-attribute := true;
    cf.is-var-attribute := true;
    vc-type-mark [[ Type-mark ]];
    cf.signature.res := [@.type-stack.top];
    type-stack.pop
  feature-bodies == @.pass
end
[decl-body-procedure]
vc-decl-body [[ is Routine ]]
= case
  feature-signatures == @.pass
    cf.is-routine := true;
    cf.is-procedure := true;
    vc-routine [[ Routine ]];
  feature-bodies == @.pass
    vc-routine [[ Routine ]];
end
[decl-body-function]
vc-decl-body [[ Type-Mark is Routine ]]
= case
  feature-signatures == @.pass
    cf.is-routine := true;
    cf.is-function := true;
    vc-type-mark [[ Type-mark ]];
    cf.signature.res := [@.type-stack.top];
    type-stack.pop
    vc-routine [[ Routine ]];
  feature-bodies == @.pass
    vc-routine [[ Routine ]];
end
[decl-body-procedure-arguments]
vc-decl-body [[ Formal-Arguments is Routine ]]
= case
  feature-signatures == @.pass
    cf.is-routine := true;
    cf.is-procedure := true;
    vc-formals [[ Formal-arguments ]];
    vc-routine [[ Routine ]];
  feature-bodies == @.pass
    vc-routine [[ Routine ]];
end
[decl-body-function-arguments]
vc-decl-body [[ Formal-arguments Type-mark is Routine ]]
= case

106
feature-signatures == @.pass
  cf.is-routine := true;
  cf.is-function := true;
  vc-formals [[ Formal-arguments ]];
  vc-type-mark [[ Type-mark ]];
  cf.signature.res := [@.type-stack.top];
  type-stack.pop
  vc-routine [[ Routine ]];
feature-bodies == @.pass
  vc-routine [[ Routine ]];
end
end module Feature-Validity

G.21 Clients

G.21.1 Client-Syntax

module Client-Syntax
  imports Identifier-Syntax
  exports
    sorts CLIENTS CLASS-LIST
    context-free syntax
      -> CLIENTS
      "{" CLASS-LIST "}" -> CLIENTS
      { ID "," }* -> CLASS-LIST
    variables
      Clients[0-9]* -> CLIENTS
      Client[*][0-9]* -> CLASS-LIST
      Client[0-9]* -> ID
end module Client-Syntax

G.21.2 Client-Validity

module Client-Validity
  imports Client-Syntax Identifier-Validity
  exports
    context-free syntax
      vc-clients "{" CLIENTS "}" -> CONTEXT-EXPR
    equations
      [clients-1]
        vc-clients [[ ] ] = vc-clients [[ {any} ]]
      [clients-2]
        vc-clients [[ {} ] ] = c-clients := {}
      [clients-3]
        vc-clients [[ { _id, _id* } ]]
        = begin
          vc-clients [[ { _id* } ]];
          if ~ @.cs.class-table?_id then
            error validity(VLCP, 101)
          else
            c-clients := _id + @.c-clients
          end
        end
end module Client-Validity
G.22 Feature-Clauses

G.22.1 Feature-Clause-Syntax

module Feature-Clause-Syntax
imports Feature-Syntax
exports
  sorts FEATURES FEATURE-CLAUSE FEATURE-DECLARATION-LIST
  context-free syntax
    FEATURE-CLAUSE* -> FEATURES
    feature CLIENTS FEATURE-DECLARATION-LIST -> FEATURE-CLAUSE
    { FEATURE-DECLARATION ";;" }* -> FEATURE-DECLARATION-LIST
variables
  Features[0-9]* -> FEATURES
  Feature-clause[0-9]* -> FEATURE-CLAUSE
  Feature-clause[*][0-9]* -> FEATURE-CLAUSE*
  Feature-clause[+][0-9]* -> FEATURE-CLAUSE+
end module Feature-Clause-Syntax

G.22.2 Feature-Clause-Validity

module Feature-Clause-Validity
imports Feature-Clause-Syntax Clients-Validity Feature-Validity
exports
  context-free syntax
    vc-feature-clauses "|" FEATURE-CLAUSE* "|" -> CONTEXT-UPDATE
    vc-feature-clause "|" FEATURE-CLAUSE "|" -> CONTEXT-UPDATE
    vc-features "|" FEATURE-DECLARATION-LIST "|" -> CONTEXT-UPDATE
  equations
    [feature-clauses-0] vc-feature-clauses [[ ]] = id-context
    [feature-clauses-n]
      vc-feature-clauses [[ Feature-clause; Feature-clause* ]] = begin
        vc-feature-clause [[ Feature-clause ]];
        vc-feature-clauses[[ Feature-clause* ]]
      end

    [feature-clause]
      vc-feature-clause [[ feature Clients Feature* ]] = begin
        vc-clients [[ Clients ]];
        vc-features [[ Feature* ]]
      end

    [features-0] vc-features [[ ]] = id-context
    [features-n]
      vc-features [[ Feature; Feature* ]] = begin
        vc-feature [[ Feature ]];
        vc-features [[ Feature* ]]
      end
end module Feature-Clause-Validity
G.23 Inheritance

G.23.1 Parent-Syntax

module Parent-Syntax
  imports Client-Syntax Feature-Name-Syntax Type-Syntax
  exports
    sorts
      INHERITANCE PARENT-LIST PARENT FEATURE-ADAPTATION
      RENAME RENAME-LIST RENAME-PAIR
      NEW-EXPORTS NEW-EXPORT-LIST NEW-EXPORT-ITEM FEATURE-SET
      FEATURE-LIST UNDEFINE REDEFINE SELECT
  context-free syntax
    -> INHERITANCE
    inherit PARENT-LIST -> INHERITANCE
    { PARENT ";" }* -> PARENT-LIST
    CLASS-TYPE FEATURE-ADAPTATION -> PARENT
    RENAME
    NEW-EXPORTS
    UNDEFINE
    REDEFINE
    SELECT end -> FEATURE-ADAPTATION
    rename RENAME-LIST -> RENAME
    { RENAME-PAIR "," }* -> RENAME-LIST
    FEATURE-NAME as FEATURE-NAME -> RENAME-PAIR
    export NEW-EXPORT-LIST -> NEW-EXPORTS
    { NEW-EXPORT-ITEM ";" }* -> NEW-EXPORT-LIST
    CLIENTS FEATURE-SET -> NEW-EXPORT-ITEM
    FEATURE-LIST -> FEATURE-SET
    all -> FEATURE-SET
    { FEATURE-NAME "," }* -> FEATURE-LIST
    undefine FEATURE-LIST -> UNDEFINE
    redefine FEATURE-LIST -> REDEFINE
    select FEATURE-LIST -> SELECT

variables
  Inheritance[0-9]* -> INHERITANCE
  Parent*[0-9]* -> { PARENT ";" }+ Parent[0-9]* -> PARENT-LIST
  Feature-adaptation[0-9]* -> FEATURE-ADAPTATION
  Rename[0-9]* -> RENAME-LIST
  Renaming[0-9]* -> RENAME-LIST
  New-exports[0-9]* -> NEW-EXPORTS
  New-export-list[0-9]* -> NEW-EXPORT-LIST
  New-export-item[0-9]* -> NEW-EXPORT-ITEM
  Feature-set[0-9]* -> FEATURE-SET
  Feature-list[0-9]* -> FEATURE-LIST
  Undefine[0-9]* -> UNDEFINE
  Redefine[0-9]* -> REDEFINE
end module Parent-Syntax
G.23.2 Parent-Validity

module Parent-Validity
imports Parents-Syntax Type-Validity
exports
context-free syntax
vc-inheritance "[" INHERITANCE "]" -> CONTEXT-UPDATE
vc-parents "[" {PARENT ":",}"* "]" -> CONTEXT-UPDATE
vc-parent "[" PARENT "]" -> CONTEXT-UPDATE
vc-feature-adoption "[" FEATURE-ADOPTION "]" -> CONTEXT-UPDATE
vc-rename "[" RENAME "]" -> CONTEXT-UPDATE
vc-rename-list "[" RENAME-LIST "]" -> CONTEXT-UPDATE
vc-rename-pair "[" RENAME-PAIR "]" -> CONTEXT-UPDATE
vc-new-exports "[" NEW-EXPORTS "]" -> CONTEXT-UPDATE
vc-new-export-list "[" NEW-EXPORT-LIST "]" -> CONTEXT-UPDATE
vc-new-export-item "[" NEW-EXPORT-ITEM "]" -> CONTEXT-UPDATE
vc-feature-set "[" FEATURE-SET "]" -> CONTEXT-UPDATE
vc-undefine "[" UNDEFINE "]" -> CONTEXT-UPDATE
vc-redefine "[" REDefINE "]" -> CONTEXT-UPDATE
vc-select "[" SELECT "]" -> CONTEXT-UPDATE
hiddens
context-free syntax
init-parent-features"(" FEATURE-TABLE ")" -> FEATURE-TABLE
init-parent-features"(" FEATURE-TABLE "," CONTEXT ")" -> FEATURE-TABLE
inherit-parent-features"(" FEATURE-TABLE ")" -> CONTEXT-UPDATE
inherit-parent-features"(" FEATURE-TABLE "," CONTEXT ")" -> CONTEXT-UPDATE

equations
[inheritance-1]
vc-inheritance [ ] = vc-inheritance [ inherit ANY ]
[inheritance-2]
vc-inheritance [ inherit ] = vc-inheritance [ inherit ANY ]
[inheritance-3]
vc-inheritance [ inherit Parent* ] = vc-parents [ Parent* ]
[parents-0]
vc-parents [ ] = id-context
[parents-n]
vc-parents [ Parent; Parent* ]
begin
vc-parent [ Parent ];
vc-parents [ Parent* ];
end

/* validity VHPR, item 1 forbids cycles in inheritance graph;
systems with such cycles are already rejected by vc-system. */

[parent]
vc-parent [ .id Actual-Generics Feature-adaptation ]
begin
vc-type [ .id Actual-Generics ];
if empty? @.cp | "@.type-stack.top.base == ANY
then
enter-parent(@.type-stack.top);
if @.cc.parent-table ? @.cp.parent-name
% .base == .base
then
/*
This parent was already inherited once.
Conflict or Repeated Inheritance

/*
error "Warning: Repeated Inheritance or Conflict?"
else
  cp.parent-features
  := init-parent-features(
      \$.cs.class-table.(\$.cp.parent-name.base).class-features
      subst make-substitution(\$.cp.parent-name));
  vc-feature-adaptation ![ Feature-adaptation ]
end;
leave-parent;
end
end

[init-pf] init-parent-features(Feature-Table) in C
  = init-parent-features(Feature-Table in C, C)
[init-pf-0] init-parent-features(, C) = []
[init-pf-n]
  init-parent-features([FC*, C]
  = _FC:is-renamed := false
  :feature-clients := {} 
  :to-be-redefined := false
  :is-undefined := false
  :origin := C.cp.parent-name
  ++ init-parent-features([FC*], C)

[inherit-pf] inherit-parent-features(Feature-Table) in C
  = inherit-parent-features(Feature-Table in C, C)
[inherit-pf-0] inherit-parent-features(, C) = C
[inherit-pf-n]
  inherit-parent-features([FC*, C]
  = inherit-parent-features([FC*], C)
  with C do
    if \$.cc.class-features?(_FC.feature-name)
      then
        cc.class-features := _FC -> \$.cc.class-features
      else
        error _FC.feature-name:"Nameclash or ???"
end)
[inherit-feature-adaptation-1]
  vc-feature-adaptation ![ ] = inherit-parent-features(\$.cp.parent-features)
[inherit-feature-adaptation-2]
  vc-feature-adaptation ![ Rename New-Exports Undefine Redefine Select end ]
  = begin
    vc-rename ![ Rename ];
    vc-new-exports ![ New-Exports ];
    vc-undefine ![ Undefined ];
    vc-redefine ![ Redefine ];
    vc-select ![ Select ];
    cc.class-features := \$.cc.class-features <- \$.cp.parent-features
  end

[rename-0] vc-rename ![ ] = id-context
[rename-n] vc-rename ![ rename Renaming* ] = vc-rename-list ![ Renaming* ]
[rename-list-0] vc-rename-list ![ ] = id-context
[rename-list-n]
  vc-rename-list ![ Feature-Name as Feature-Name’, Renaming* ]
\begin{verbatim}
= begin
  vc-rename-pair |[ normal(Feature-Name) as normal(Feature-Name') ]|;
  vc-rename-list |[ Renaming* ]|
end

[rename-pair]
vc-rename-pair |[ Feature-Name as Feature-Name' ]|
= if ~ cp.parent-features?Feature-Name then
  error Feature-Name:validity(VHRC, 81, 1)
else
  cf := cp.parent-features.Feature-Name;
  if @.cf.is-renamed then
    error Feature-Name:validity(VHRC, 81, 1)
  else
    /* [May92] says nothing about clashes caused by renaming ?
    Does this follow from rule about disjunct feature names?
    What about f as g, g as h?
    */
    cf.feature-name := Feature-Name';
    cf.is-renamed := true;
    cp.parent-features
    := @.cf -> (@.cp.parent-features - Feature-Name)
end
end

[new-exports-0] vc-new-exports |[ ]| = id-context
[new-exports-n] vc-new-exports |[ export New-export-list ]|
  = vc-new-export-list |[ New-export-list ]|
[new-export-list-0] vc-new-export-list |[ ]| = id-context
[new-export-list-1]
vc-new-export-list |[ Clients all; Clients Feature-list; New-export-list ]|
  = vc-new-export-list |[ Clients Feature-list; Clients all; New-export-list ]|
[new-export-list-2]
vc-new-export-list |[ Clients all; Clients' all; New-export-list ]|
  = begin
    error validity(VLEL, 102, 1);
    vc-new-export-list |[ Clients all; New-export-list ]|
end
[new-export-list-3]
vc-new-export-list |[ Clients all ]| = vc-new-export-item |[ Clients all ]|
[new-export-list-4]
vc-new-export-list |[ Clients Feature-list; New-export-list ]|
  = begin
    vc-new-export-item |[ Clients Feature-list ]|;
    vc-new-export-list |[ New-export-list ]|
end
[new-export-item]
vc-new-export-item |[ Clients Feature-set]|
  = begin
    vc-clients |[ Clients ]|;
    vc-feature-set |[ Feature-set ]|
end
[feature-set-1]
vc-feature-set |[ all ]|
  = begin
    -- all cp.parent-features with {} as clients field
    -- get c-clients as clients
end
\end{verbatim}
end

[feature-set-2] vc-feature-set |[ ]| = id-context
[feature-set-3]
vc-feature-set |[ Feature-Name, Feature-Name* ]|
= if " @.cp.parent-features?normal(Feature-Name)
then
  error normal(Feature-Name):validity(VLEL, 102, 2)
else
  cf := @.cp.parent-features.normal(Feature-Name);
  if " empty? @.cf.clients then
    error normal(Feature-Name):validity(VLEL, 102, 3)
  else
    cf.clients := @.c-clients;
    cp.parent-features := @.cf -> @.cp.parent-features
end

[undefine-1] vc-undefine |[ ]| = id-context
[undefine-2] vc-undefine |[ undefine ]| = id-context
[undefine-3]
vc-undefine |[ undefine Feature-Name, Feature-Name* ]|
= begin
  if @.cp.parent-features?normal(Feature-Name) then
    error Validity(VDUS, 160, 1)
  else
    cf := @.cp.parent-features.normal(Feature-Name);
    case
      @.cf.is-frozen | @.cf.is-attribute
      error validity(VDUS, 160, 2);
      " @.cf.is-effective
      error validity(VDUS, 160, 3);
      @.cf.is-undefined
      -- ????
      error validity(VDUS, 160, 4);
    otherwise
      @.cf.is-deferred := true
      cp.parent-features := @.cf -> @.cp.parent-features
    end
  end;
  vc-undefine |[ undefine Feature-Name* ]|
end

[redefine-1] vc-redefine |[ ]| = id-context
[redefine-2] vc-redefine |[ redefine ]| = id-context
[redefine-3]
vc-redefine |[ redefine Feature-Name, Feature-Name* ]|
= begin
  if @.cp.parent-features?normal(Feature-Name) then
    error Validity(VDUS, 159, 1)
  else
    cf := @.cp.parent-features.normal(Feature-Name);
    case
      @.cf.is-frozen | @.cf.is-const-attribute
      error normal(Feature-Name):validity(VDUS, 159, 1)
      @.cf.to-be-redefined
      -- ????
      error normal(Feature-Name):validity(VDUS, 159, 3)
    otherwise
113
@@.cf.to-be-redefined := true;
cp.parent-features := @.cf -> @.cp.parent-features
end
end;
vc-redefine [[ redefine Feature-Name* ]] end

[select-1] vc-select [[ ]] = id-context
[select-2]
  vc-select [[ select Feature-Name, Feature-Name*]]
  = begin
    error "Warning: vc-select not specified"
  end
end module Parent-Validity

G.24 Formal Generics

G.24.1 Formal-Generic-Syntax

module Formal-Generic-Syntax
  imports Type-Syntax
  exports
    sorts FORMAL-GENERICS FORMAL-GENERIC-LIST FORMAL-GENERIC CONSTRAINT
    context-free syntax
      "[" FORMAL-GENERICS "]" -> FORMAL-GENERICS
      { FORMAL-GENERIC ","} -> FORMAL-GENERIC-LIST
      ID CONSTRAINT -> FORMAL-GENERIC
      -> CONSTRAINT
      ":->" CLASS-TYPE -> CONSTRAINT
    variables
      Formal-generics[0-9]* -> FORMAL-GENERICS
      Formal-generic-list[0-9]* -> FORMAL-GENERIC-LIST
      Formal-generic[*][0-9]* -> FORMAL-GENERIC-LIST
      Formal-generic[0-9]* -> FORMAL-GENERIC
      Constraint[0-9]* -> CONSTRAINT
  end module Formal-Generic-Syntax

G.24.2 Formal-Generic-Validity

module Formal-Generic-Validity
  imports Type-Validity
  exports
    context-free syntax
      vc-generics "[" FORMAL-GENERICS "]" -> CONTEXT-UPDATE
      vc-constraint "[" CONSTRAINT "]" -> CONTEXT-UPDATE
    equations
      [constraint-1] vc-constraint [[ ]] = type-stack.push(class-type(ANY,[[])])
      [constraint-2] vc-constraint [[ -> Type ]] = vc-type [[ Type ]] end
      [generics-1] vc-generics [[ ]] = cc.formal-generics := []
      [generics-2] vc-generics [[ ]] = cc.formal-generics := []
      [generics-3] vc-generics [[ _id Constraint, FG* ]] = begin
        vc-generics [[ [FG* ]]];
case
   formal-generics1 == @.pass
   require "@.cs.class-table ? upper-name(_id)
   else validity(VCFG, 52, 1);
   require "@.cc.formal-generics ? upper-name(_id)
   else validity(VCFG, 52, 2);
   cc.formal-generics := (upper-name(_id), class-type(ANY,[]))
     -> @.cc.formal-generics

formal-generics2 == @.pass
vc-constraint ![ Constraint ];
cc.formal-generics := (upper-name(_id), @.type-stack.top)
     -> @.cc.formal-generics
end
end
end module Formal-Generics-Validity

G.25 Indexing

module Indexing-Syntax
imports Identifier-Syntax Constant-Syntax
exports
sorts INDEXING INDEX-LIST INDEX-CLAUSE INDEX INDEX-TERMS INDEX-VALUE
context-free syntax
   indexing INDEX-LIST -> INDEXING
   {INDEX-CLAUSE ";"}+ -> INDEX-LIST
   INDEX INDEX-TERMS -> INDEX-CLAUSE
   -> INDEX
   ID ";" -> INDEX
   {INDEX-VALUE ","}+ -> INDEX-TERMS
   ID -> INDEX-VALUE
   MANIFEST-CONSTANT -> INDEX-VALUE

end module Indexing-Syntax

G.26 Obsolete

module Obsolete-Syntax
imports String-Char-Syntax
exports
sorts OBSOLETE MESSAGE
context-free syntax
   obsolete MESSAGE -> OBSOLETE
   MANIFEST-STRING -> MESSAGE
variables
   Obsolete[0-9]* -> OBSOLETE

end module Obsolete-Syntax

G.27 Creators
G.27.1 Creators-Syntax

module Creators-Syntax
imports Client-Syntax Feature-Name-Syntax
exports
sorts CREATORS CREATION-CLAUSE
context-free syntax
CREATION-CLAUSE* -> CREATORS
creation CLIENTS HEADER-COMMENT FEATURE-LIST -> CREATION-CLAUSE
variables
Creators[0-9]* -> CREATORS Creation-Clause[0-9]* -> CREATION-CLAUSE
end module Creators-Syntax

G.27.2 Creators-Validity

module Creators-Validity
imports Creators-Syntax Clients-Validity Feature-Name-Validity % % ??
exports
context-free syntax
vc-creators " | [ " CREATORS " ] | " -> CONTEXT-UPDATE
vc-creation-clause " | [ " CREATION-CLAUSE " ] | " -> CONTEXT-UPDATE
vc-feature-creators " | [ " FEATURE-LIST " ] | " -> CONTEXT-UPDATE
vc-feature-creator " | [ " FEATURE-NAME " ] | " -> CONTEXT-UPDATE
equations
collectors-0] vc-creators | [] | = id-context
collectors-n]
vc-creators | [ Creation-Clause Creators ] |
begin
vc-creation-clause | [ Creation-Clause ] |
vc-creators | [ Creators ] |
end
creation-clause]
vc-creation-clause | [ creation Clients Feature-List ] |
begin
require ` @ .cc.is-deferred
  else validity(VGCCP, 285, 1);
vc-clients | [ Clients ] |
vc-feature-creators | [ Feature-List ] |
end
[feature-creators-0] vc-feature-creators | [ ] | = id-context
[feature-creators-n]
vc-feature-creators | [ Feature-Name, Feature-Name* ] |
begin
vc-feature-creator | [ Feature-Name ] |
vc-feature-creators | [ Feature-Name* ] |
end
[feature-creator-1]
vc-feature-creator | [ Binary ] | = vc-feature-creator | [ binary(Binary) ] |
[feature-creator-2]
vc-feature-creator | [ Unary ] | = vc-feature-creator | [ unary(Unary) ] |
[feature-creator-3]
vc-feature-creator | [ _id ] |
= if ` cc.class-features?upper-name(_id) then
  error validity(VGCCP, 285, 2)
  else
  enter-feature(upper-name(_id));
  require ` @ .cf.is-creator
  else validity(VGCCP, 285, 3);
  require @ .cc.is-expanded <-> @ .cf.signature == ([],[])
116
else validty(WGCP, 285, 4);
cf.is-creator := true;
cf.clients := 0.cf.clients + c-clients;  %% ??
leave-feature
end

module Creators-Validity

G.28 Class Header

G.28.1 Class-Header-Syntax

module Class-Header-Syntax
imports Identifier-Syntax
exports
sorts CLASS-HEADER HEADER-MARK
context-free syntax
   HEADER-MARK class ID  -> CLASS-HEADER
   deferred             -> HEADER-MARK
   expanded             -> HEADER-MARK
variables
   Class-header[0-9]*    -> CLASS-HEADER
   Header-mark[0-9]*     -> HEADER-MARK
end module Class-Header-Syntax

G.28.2 Class-Header-Validity

module Class-Header-Validity
imports Class-Header-Syntax
exports
context-free syntax
   vc-header-mark "|" [" HEADER-MARK "] | " -> CONTEXT-UPDATE
equations
[0] vc-header-mark [ ]
    = begin
        @.cc.is-deferred := false;
        @.cc.is-expanded := false
    end

[1] vc-header-mark [ deferred ]
    = begin
        @.cc.is-deferred := true;
        @.cc.is-expanded := false
    end

    = begin
        @.cc.is-deferred := false;
        @.cc.is-expanded := true
    end

end module Class-Header-Validity
G.29 Class

G.29.1 Class-Syntax

module Class-Syntax
imports
  Indexing-Syntax Obsolete-Syntax Formal-Generics-Syntax
  Parent-Syntax Feature-Clause-Syntax Class-Header-Syntax
  Creators-Syntax
exports
  sorts CLASS-DECLARATION
context-free syntax

INDEXING
CLASS-HEADER FORMAL-GENERICS
  OBSOLETE
  INHERITANCE
  CREATORS
  FEATURES
  INVARIANT
end
  -> CLASS-DECLARATION

variables
  Class[0-9]* -> CLASS-DECLARATION

end module Class-Syntax

G.29.2 Class-Strip

module Class-Strip
imports Class-Syntax
equations
  % Strip Indexing and Obsolete
  [00] Indexing Class-Header Formal-generics Obsolete Inheritance Creators Features Invariant end
      =
          Class-Header Formal-generics          Inheritance Creators Features Invariant end

  % normalize formal generics
  [01] Class-Header [FG*1, _id, FG*2] Inheritance Creators Features Invariant end
      = Class-Header [FG*1, _id -> ANY, FG*2] Inheritance Creators Features Invariant end

  % normalize inheritance
  [02] Class-Header Formal-generics Creators Features Invariant end
      = Class-Header Formal-generics inherit ANY Creators Features Invariant end

  [03] Class-Header Formal-generics inherit Creators Features Invariant end
      = Class-Header Formal-generics inherit ANY Creators Features Invariant end

end module Class-Strip

G.29.3 Class-Validity

module Class-Validity
imports Feature-Validity Class-Strip
exports
  context-free syntax
  vc-class "|"[" CLASS "]|" -> CONTEXT-UPDATE
equations
[class]  
vc-class |[ Header-mark class _id Formal-generics  
  Inheritance  
  Creators  
  Features  
  Invariant  
  end  
}|  
=  
begin  
  enter-class(upper-name(_id));  
case  
  class-names == @.pass  
  vc-header-mark |[ Header-mark ];  
  require ~ @.cs.class-table?upper-name(_id)  
  else validity(VSCN, 38) & validity(VCCH, 51, 1)  
  formal-generics1 == @.pass  
  vc-generics |[ Formal-generics ];  
  formal-generics2 == @.pass  
  vc-generics |[ Formal-generics ];  
  feature-signatures == @.pass  
  @.cc.check-deferred := false;  
  vc-inheritance |[ Inheritance ];  
  vc-feature-clauses |[ Features ];  
  vc-creators |[ Creators ];  
  feature-bodies == @.pass  
  vc-feature-clauses |[ Features ];  
  vc-invariant |[ Invariant ];  
  require cc.is-deferred <-> cc.check-deferred  
  else validity VCCH p.51 item 2;  
end;  
leave-class  
end -- vc-class  
end module Class-Validity

G.30 LACE
G.30.1 LACE-Syntax

module LACE-Syntax
  imports Identifier-Syntax
  exports
  sorts ACE
  context-free syntax
    system ID root ID end -> ACE
  variables
    Ace[0-9]* -> ACE

end module LACE-Syntax

G.30.2 LACE-Validity

module LACE-Validity
  imports CML Lace-Syntax
  exports
  context-free syntax
    vc-ace "|" ACE "|" -> CONTEXT-UPDATE
end module LACE-Validity

G.31 System

G.31.1 System-Syntax

module System-Syntax
imports LACE-Syntax Class-Syntax
exports
sorts SYSTEM
context-free syntax
ACE CLASS* -> SYSTEM
variables
System[0-9]* -> SYSTEM
Class[*][0-9]* -> CLASS*
end module System-Syntax

G.31.2 System-Validity

module System-Validity
imports System-Syntax LACE-Validity Class-Validity
exports
context-free syntax
vc-system "[" SYSTEM "]" -> CONTEXT-UPDATE
hiddens
context-free syntax
parent-names "[" INHERITANCE "]" -> ID-SET
class-name "[" CLASS "]" -> ID
inheritance-graph "[" CLASS* "]" -> INHERITANCE-GRAPH
insert CONTEXT "[" CLASS "]" "[" CLASS* "]" "[" CLASS* "]" -> CLASS*
CLASS* in CONTEXT -> CLASS*
sort "[" CLASS* "]" -> CLASS*
vc-classes1 "[" CLASS* "]" -> CONTEXT-UPDATE
vc-classes "[" CLASS* "]" -> CONTEXT-UPDATE
check-root-creators(" FEATURE-TABLE ")'" -> CONTEXT-UPDATE
check-root-creators(" FEATURE-TABLE "," CONTEXT ")'" -> CONTEXT

equations
[00] parent-names [[ inherit ]] = {}
[01] parent-names [[ inherit _id Actual-Generics Feature-Adoptation Parent* ]] = upper-name(_id) + parent-names [[ inherit Parent* ]]

[02] class-name [[ Header-mark class _id Formal-generics
    Inheritance Creators Features Invariant end ]]
    = upper-name(_id)

[03] inheritance-graph [[ Header-mark class _id Formal-generics
    Inheritance Creators Features Invariant end
    Class* ]] = (upper-name(_id), parent-names [[ Inheritance ]])
+ inheritance-graph |[ Class* ]|

[04] insert C |[ Class ]| | ]| = Class
[05] class-name[Class]| in C.ig-trans.class-name|[Class'].parents = true

insert C |[ Class ]| | Class' Class* |

[06] class-name[Class]| in C.ig-trans.class-name|[Class'].parents = false

insert C |[ Class ]| | Class' Class* |

[07] sort |[ ]| in C =

[08] sort |[ Class Class* ]| in C = insert C |[ Class ]| | [ sort |[ Class* ]| in C ]|

[09] vc-classes1 |[ ]| = id-context
[10] vc-classes1 |[ Class Class* ]|
   = begin vc-class |[ Class ]|; vc-classes1 |[ Class* ]| end


vc-classes |[ Class* ]| in C
   = with C do
      begin
         pass := class-names;
         vc-classes1 |[ Class*’ ]|;
         if empty? @.cs.errors then
            pass := formal-generics;
            vc-classes1 |[ Class*’ ]|;
            pass := feature-signatures;
            vc-classes1 |[ Class*’ ]|;
            pass := feature-bodies;
            vc-classes1 |[ Class*’ ]|
      end
   end

[12] check-root-creators(_Feature-Table) in C = check-root-creators(_Feature-Table in C, C)
[14] check-root-creators([_FC, _FC*], C)
   = check-root-creators([_FC*],
      with C do
         if _FC.is-creator then
            require
               _FC.signature == ([],[])
               _FC.signature
                  == ([class-type(ARRAY,[class-type(STRING,[])])],[])
            else
               _FC.feature-name:validity(VSRC,36,2)
         end)

[15] vc-system |[ Ace Class* ]|
   = begin
      cs.ig := inheritance-graph |[ Class* ]|;
      cs.ig-trans := trans(@.ig);
      if "irrefl?(@.ig-trans) then
         error "Cycle in inheritance graph"
      else
         vc-ace |[ Ace ]|;
         vc-classes |[ Class* ]|;
      end

      -- check root class rule
      if "@.cs.class-table?(@.cs.root-name) then
error "root class not in system"
else
    require # (@.cs.class-table.@.cs.root-name.formal-generics) == 0
    else validity(VSRC,36,1);
    check-root-creators(@.cs.class-table.@.cs.root-name.class-features)
end
end
end
end module System-Validity
# Contents

1 Introduction ................................................. 2
   1.1 General ............................................. 2
   1.2 Preliminaries ......................................... 2
      1.2.1 Eiffel ........................................... 2
      1.2.2 ASF ............................................... 3
      1.2.3 SDF ............................................... 3
      1.2.4 ASF+SDF ........................................ 4
   1.3 Related Work ........................................... 4
   1.4 Outline of the Paper .................................... 4

2 Checking the Static Semantics of Eiffel ....................... 4

3 Overview of the Specification—The Tools .................... 6
   3.1 Abstract Datatype Modules ............................. 6
   3.2 Reusing Modules ....................................... 8
      3.2.1 The Lambda and Merge Operators ................... 8
      3.2.2 Abbreviations .................................... 8
      3.2.3 Implementation ................................... 9
      3.2.4 What Parameter Mechanism to Choose? .............. 9
   3.3 The Symbol Table or Context ............................ 9
      3.3.1 Components of the Context ........................ 10
      3.3.2 Examples of Usage ................................ 10
   3.4 A Context Modification Language ....................... 13
      3.4.1 Syntax of CML .................................... 13
      3.4.2 Semantics of CML ................................ 13
      3.4.3 The Context as Global Environment ................. 14
      3.4.4 Reusability of the Contexts Specification ......... 15

4 Overview of the Specification—Operations on Types .......... 15
   4.1 Type Substitutions .................................... 15
   4.2 Bitsize of Types ...................................... 16
   4.3 Conformance ......................................... 16

5 Overview of the Specification—the Validity Checker ....... 16
   5.1 Outline of the Validity Checking Process ............... 16
   5.2 Conformance to the Definition ........................ 17
   5.3 Future Work ......................................... 17

6 Discussion .................................................. 18

7 Conclusions .................................................. 19

References ..................................................... 20

A Import Graphs ................................................. 22

B Library Modules ............................................... 25
   B.1 Layout ............................................... 25
   B.2 Booleans ............................................. 25
   B.3 Integers .............................................. 26
E.2.1 Module Bool-in-Context .................................... 51
E.2.2 Module Identifier-in-Context ............................. 51
E.2.3 Module Integers-in-Context ............................... 52
E.2.4 Module String-in-Context ................................. 52
E.2.5 Module Sets-in-Context .................................. 52
E.2.6 Module Tuples2-in-Context .............................. 53
E.2.7 Module Sequences-in-Context ........................... 53
E.2.8 Module Tables-in-Context ............................... 54
E.2.9 Module Graphs-in-Context .............................. 54
E.2.10 Abbreviations for in-Context Modules ................. 54
E.2.11 Declaration of Module In-Context ..................... 56
E.3 A Context Modification Language ......................... 62

F Eiffel—Operations on Types ................................. 66
F.1 Type Substitutions .......................................... 66
F.2 Bitsize of Types ............................................ 67
F.3 Conformance of Types, Type sequences and Signatures . 68

G Eiffel—Syntax and Static Semantics ......................... 72
G.1 Layout ......................................................... 72
G.2 Identifiers ................................................... 72
G.2.1 Identifier-Syntax ...................................... 72
G.2.2 Identifier-Equality .................................. 72
G.2.3 Identifier-Validity ................................... 75
G.3 Operators ................................................... 75
G.3.1 Operator-Syntax ...................................... 75
G.3.2 Operator-Equality ................................... 76
G.4 Feature-Names ............................................. 76
G.4.1 Feature-Name-Syntax .................................. 76
G.4.2 Feature-Name-Equality ................................ 76
G.5 Bit-Sequences ............................................. 77
G.5.1 Bit-Sequence-Syntax ................................ 77
G.5.2 Bit-Sequence-Validity ................................ 77
G.6 Booleans .................................................... 78
G.6.1 Boolean-Syntax ...................................... 78
G.6.2 Boolean-Validity ................................... 78
G.7 Integers ..................................................... 78
G.7.1 Integer-Syntax ...................................... 78
G.7.2 Integer-Validity ................................... 78
G.8 Reals ........................................................ 79
G.8.1 Real-Syntax ........................................... 79
G.8.2 Real-Validity ......................................... 79
G.9 Characters and Strings .................................... 79
G.9.1 String-Char-Syntax .................................. 79
G.9.2 String-Char-Validity ................................ 80
G.10 Constants ................................................... 80
G.10.1 Constant-Syntax .................................... 80
G.10.2 Constant-Validity .................................. 81
G.11 Types ....................................................... 81
G.11.1 Type-Syntax .......................................... 81
G.11.2 Type-Validity ....................................... 82
G.12 Formal Arguments and Local Entities .................... 85
G.12.1 Formals-Locals-Syntax ................................ 85
G.12.2 Formals-Locals-Validity ............................. 86

125
G.13 Expressions .................................................. 88
  G.13.1 Expression-Syntax .................................... 88
  G.13.2 Expression-Validity ................................... 89
G.14 Assertions .................................................. 93
  G.14.1 Assertion-Syntax ...................................... 93
  G.14.2 Assertion-Validity .................................... 94
G.15 Assignment ................................................ 96
  G.15.1 Assignment-Syntax .................................... 96
  G.15.2 Assignment-Validity ................................... 96
G.16 Instructions ............................................... 98
  G.16.1 Instruction-Syntax .................................... 98
  G.16.2 Instruction-Validity .................................. 99
G.17 External .................................................. 101
  G.17.1 External-Syntax ...................................... 101
  G.17.2 External-Validity .................................... 101
G.18 Obsolete Messages ....................................... 102
G.19 Features ................................................ 102
  G.19.1 Routine-Syntax ....................................... 102
  G.19.2 Routine-Validity ..................................... 102
G.20 Features ................................................ 103
  G.20.1 Feature-Syntax ....................................... 103
  G.20.2 Feature-Validity ..................................... 104
G.21 Clients ................................................ 107
  G.21.1 Client-Syntax ......................................... 107
  G.21.2 Client-Validity ....................................... 107
G.22 Feature-Clauses ......................................... 108
  G.22.1 Feature-Clause-Syntax ................................ 108
  G.22.2 Feature-Clause-Validity ................................ 108
G.23 Inheritance ............................................. 109
  G.23.1 Parent-Syntax ......................................... 109
  G.23.2 Parent-Validity ....................................... 110
G.24 Formal Generics ......................................... 114
  G.24.1 Formal-Generics-Syntax ................................ 114
  G.24.2 Formal-Generics-Validity ................................ 114
G.25 Indexing ................................................ 115
G.26 Obsolete ................................................ 115
G.27 Creators ................................................ 115
  G.27.1 Creators-Syntax ....................................... 115
  G.27.2 Creators-Validity ..................................... 116
G.28 Class Header ............................................ 117
  G.28.1 Class-Header-Syntax .................................. 117
  G.28.2 Class-Header-Validity ................................ 117
G.29 Class .................................................. 118
  G.29.1 Class-Syntax ......................................... 118
  G.29.2 Class-Strip ........................................... 118
  G.29.3 Class-Validity ........................................ 118
G.30 LACE ................................................... 119
  G.30.1 LACE-Syntax .......................................... 119
  G.30.2 LACE-Validity .......................................... 119
G.31 System .................................................. 120
  G.31.1 System-Syntax ......................................... 120
  G.31.2 System-Validity ....................................... 120