DRAFT
Rapide 1.0 Predefined Types and Objects Reference Manual

RAPIDE Design Team
Program Analysis and Verification Group
Computer Systems Lab
Stanford University

July 17, 1997

Preface

This manual is part of a suite of manuals on RAPIDE-1.0. The complete set of manuals is:

- Guide to the RAPIDE 1.0 Language Reference Manuals
- RAPIDE 1.0 Types Language Reference Manual
- RAPIDE 1.0 Architectures Reference Manual
- RAPIDE 1.0 Executable Language Reference Manual
- RAPIDE 1.0 Event Pattern Language Reference Manual
- RAPIDE 1.0 Constraint Language Reference Manual
- RAPIDE 1.0 Predefined Types and Objects Reference Manual

This manual describes the standard predefined types and object that are available in the Stanford RAPIDE toolset. The syntax and semantics of each of these predefined units are described. A few things to remember:

- For each predefined type, the manual provides a RAPIDE interface or function type that defines precisely and completely the operations on that predefined type. This "equivalent" type, and the list of constraints associated with it, can usually be used as a source of documentation of all the operations provided on each of the predefined types. The constraints are a list of the assumptions that can be made by a user of the predefined type about its behavior, and these must be satisfied by any implementor of the type.

- [redacted][1] s omething similar about objects.

- In the case of certain types, the default syntax for using a predefined type seems uncomfortably dissimilar to more conventional usage. In such cases, RAPIDE provides some extra user-friendly syntax.

- It should be remembered that the RAPIDE 1.0 Executable Language automatically provides *infix* syntax for all operators defined in interfaces with a single argument. Similarly, it provides *prefix* syntax for those with zero arguments. We will often use this syntax in the interest of increasing readability. See the Executable LRM, Section 4.3.5 for details.

- Certain types and type constructors are simply used as a convenient repository of operations (and constraints) that are defined for objects of more than one type. Hence the RAPIDE tool-set does not provide any objects specifically associated with these types.

The manual proper consists of the syntax and semantics sections of each chapter. Examples and commentaries are for explanatory purposes only.

Contents

1 Introduction **1**

1.1 Notational Conventions 1

2 Boolean **3**

2.1 Boolean type expressions 3

2.1.1 Syntax 3

2.1.2 Semantics 3

2.2 Boolean object expressions 4

2.2.1 Syntax 4

2.2.2 Examples 4

3 Equality **7**

3.1 Purpose 7

3.2 Equality type expressions 7

3.2.1 Syntax 7

3.2.2 Semantics 7

3.2.3 Examples 8

3.3 Equality object expressions 8

4 Order **9**

4.1 Purpose 9

4.2 Order type expressions 9

4.2.1 Syntax 9

4.2.2 Semantics 9

4.2.3 Examples 10

4.3 Order object expressions 10

5 Triv **11**

5.1 Purpose 11

5.2 Triv type expressions 11

5.2.1 Syntax 11

5.2.2 Semantics 11

5.3 Triv object expressions 11

6 Numeric types **13**

6.1 Numeric type expressions 13

6.1.1 Syntax 13

6.1.2 Semantics 13

6.1.3 Subtyping 15

6.2 Numeric object expressions 15

6.2.1 Syntax 15

7 Character **17**

7.1 Character type expressions 17

7.1.1 Syntax 17

7.1.2 Semantics 17

7.2 Character object expressions 18

7.2.1 Syntax 18

8 String **19**

8.1 String type expressions 19

8.1.1 Syntax 19

8.1.2 Semantics 19

8.2 String object expressions 20

8.2.1 Syntax 20

8.2.2 Examples 21

9 Record **23**

9.1 Record type expressions 23

9.1.1 Syntax 23

9.1.2 Semantics 23

9.1.3 Examples 24

9.1.4 Subtyping 24

9.2 Record object expressions 26

9.2.1 Syntax 26

9.2.2 Semantics 26

9.2.3 Examples 26

10 Union 27

10.1 Union type expressions 27

10.1.1 Syntax 27

10.1.2 Semantics 27

10.1.3 Examples 28

10.1.4 Subtyping for Union types 28

10.2 Union object expressions 30

10.2.1 Syntax 30

10.2.2 Examples 31

10.3 Reduction of union types to function types 33

11 Enumeration 37

11.1 Enumeration type expressions 37

11.1.1 Syntax 37

11.1.2 Semantics 37

11.1.3 Examples 38

11.1.4 Subtyping 38

11.2 Enumeration object expressions 38

11.2.1 Syntax 38

12 Iteration 39

12.1 Iteration type expressions 39

12.1.1 Syntax 39

12.1.2 Semantics 39

12.1.3 Examples 40

12.1.4 Subtyping 40

12.2 Iteration object expressions 40

13 Range Type 41

13.1 Range 41

13.2 Range type expressions 41

13.2.1 Syntax 41

13.2.2 Semantics 41

13.2.3 Subtyping 42

13.3 Range object expressions 42

13.3.1 Syntax 42

13.3.2 Examples 42

14 Array **45**

14.1 Array type expressions 45

14.1.1 Syntax 45

14.1.2 Semantics 45

14.1.3 Examples 46

14.1.4 Subtyping 46

14.2 Array object expressions 46

14.2.1 Syntax 46

14.2.2 Examples 48

15 Set **49**

15.1 Set type expressions 49

15.1.1 Syntax 49

15.1.2 Semantics 49

15.1.3 Examples 50

15.1.4 Subtyping 50

15.2 Set object expressions 50

15.2.1 Syntax 50

16 Ref **51**

16.1 Ref type expressions 51

16.1.1 Syntax 51

16.1.2 Semantics 51

16.1.3 Examples 52

16.1.4 Subtyping 52

16.2 Ref object expressions 52

16.3 References and Causality 53

17 Clock **55**

17.1 Clock type expressions 55

17.1.1 Syntax 55

17.1.2 Semantics 55

17.1.3 Examples 57

17.1.4 Subtyping 58

17.2 Clock object expressions 58

18 Predefined Pattern Macros **59**

18.1 Introduction 59

18.2 Syntax 59

18.3 Semantics 60

18.4 Special Syntax 60

19 Event **61**

19.1 Event type expressions 61

19.1.1 Syntax 61

19.1.2 Semantics 61

19.1.3 Examples 62

19.1.4 Subtyping 62

19.2 Event object expressions 62

20 Input and Output **63**

20.1 Introduction 63

20.2 Syntax 63

20.3 Semantics 65

20.4 Transformation Functions 65

20.4.1 Syntax 65

20.4.2 Semantics 65

20.5 Examples 66

21 Link **69**

21.1 Introduction 69

21.2 Syntax 69

21.3 Semantics 69

22 Identity **71**

22.1 Introduction 71

22.2 Syntax 71

22.3 Semantics 71

A Predefined Types Syntax : Quick Reference **73**

Glossary **78**

Index **79**

Chapter 1

Introduction

This manual defines the predefined language components that are provided by the RAPIDE toolset. In RAPIDE, familiar types such as integers, arrays, and enumeration types are all definable as interface types; predefined types have no special place in the language. These types are supplied in a predefined library that is accessible via separate compilation.

1.1 Notational Conventions

- Grammar productions are given in the form:

  ```
  non_terminal ::=
          rhs1
  |       rhs2
  |       ...
  |       rhsn
  ```

  The rhs_i (*rhs* stands for right hand side) are a sequence of terminals, non-terminals and the following constructs:

`[ rhs ]`	zero or one occurrences of `rhs`
`{ rhs }`	zero or more occurrences of `rhs`
`( rhs )`	grouping
`rhs1 | ...| rhsn`	choice

  Keywords are written in **boldface**. Terminals (other than keywords) are written in all UPPERCASE.

  Some nonterminals are defined in other Language Reference Manuals. Nonterminals starting with one of the words "types_" "exec_" "predef_" "architecture_" or "constraint_" indicates that the symbol is defined in another manual (the RAPIDE 1.0 Types Language Reference Manual, RAPIDE 1.0 Executable Language Reference Manual, this manual, the RAPIDE 1.0 Architectures Reference Manual and the RAPIDE 1.0 Constraint Language Reference Manual, respectively).

  E.g. the nonterminal *types_object_name_declaration* indicates that it is defined as the meaning of *object_name_declaration*in the RAPIDE 1.0 Types Language Reference Manual.

- The types defined in this document are mostly defined as *subtypes* of other types in order to restrict the programmer's ability to define her own implementation of predefined types.

Example: The type integer (see 6) is defined as a subtype fo a an interface type:
type integer <: **interface function** Pred() **return** integer; ... **end**;

The interface identifies the attributes of objects of type integer that are available to the programmer, without allowing her to define her own integer objects except by means of the integer objects supplied by the Preamble module.

[2] **D** o we still want this? There is no way to define it in the language itself.

- The RAPIDE 1.0 Executable Language Reference Manual provides infix and prefix notation for binary, unary and nullary functions. This document will make use of that notational convention. Thus the expression

  e1 f e2

  where e1 and e2 are expressions and f is an identifier represents one of the expressions

  e1.f(e2)

  and

  f(e1, e2).

  The context determines which is appropriate.

  The expression

  f e

  where where f is an identifier and e is an expression may represent the expression

  e.f()

  where appropriate.

  Example: For integers 1 and 2 (see Chapter 6), the expression 1 + 2 represents the expression 1.+(2).

  For the boolean (see Chapter 2) True, the expression **not** True represents the expression True.Not()

[3] **Big Issue: Right now all the formal definitions of the types use FOL, and define the axioms separate from the types themselves. Can we use the Anna-like constraint language and put these axioms in the types themselves?**

[4] **Should eventually list all the predefined exceptions that are raised when the operations aren't well-defined.**

[5] **Should we use prefix/infix notation for constraints, or the more accurate and illustrative "dot" notation?**

[6] **One thing that needs to be checked is the subtyping relationship sections. I think many are wrong (e.g. for arrays and sets).**

[7] **There are a few things the current implementation has that are not included in this document (you can find them in /usr/rapide/examples/predef/*.tex). We need to decide if we want them in this document. They are: arguments.tex, list.tex, new.tex, root.tex.**

Chapter 2

Boolean

2.1 Boolean type expressions

2.1.1 Syntax

boolean_type_expression ::= Boolean

2.1.2 Semantics

The operations that can be performed on booleans are described by the following interface:

```
type Boolean <: interface
      function Not() return Boolean;
      function And(rhs : Boolean) return Boolean;
      function Or(rhs : Boolean) return Boolean;
      function Xor(rhs : Boolean) return Boolean;
      function =(rhs : Boolean) return Boolean;
      function /=(rhs : Boolean) return Boolean;
end interface;

False : Boolean is implementation_defined;
True : Boolean is implementation_defined;
```

The following constraints are imposed on the behavior of the operations on booleans:

1. There are two distinct predefined boolean modules, True and False.
2. Every boolean has to be either equal to True or False.
3. Not True returns False.
   Not False returns True.
4. True And True returns True.
   True And False returns False.

False And True returns False.
False And False returns False.

5. True Or True returns True.
True Or False returns True.
False Or True returns True.
False Or False returns False.

6. True Xor True returns False.
True Xor False returns True.
False Xor True returns True.
False Xor False returns False.

7. True = True returns True.
True = False returns False.
False = True returns False.
False = False returns True.

8. True /= True returns False.
True /= False returns True.
False /= True returns True.
False /= False returns False.

2.2 Boolean object expressions

2.2.1 Syntax

Boolean literals

boolean_literal ::= True | False

Boolean operations

All boolean operations can be invoked using the standard syntax corresponding to the interface expression for Boolean.

2.2.2 Examples

Example:
Pred : Boolean **is** True;
Commentary:

This declares the name Pred to denote the Boolean object True.

□

Example:
Verify_Check : Boolean **is** Verify_Signature And (Not Overdrawn);
Commentary:

This declares Verify_Check to be a boolean whose value is true iff Verify_Signature is true and Overdrawn is false.

□

Chapter 3

Equality

3.1 Purpose

Most predefined types have equality, inequality and other equivalence and ordering functions defined on their objects. These common operations are encapsulated in the type constructors Equality, defined in this chapter, and Order (see Chapter 4).

3.2 Equality type expressions

3.2.1 Syntax

equality_type_expression ::= Equality '(' *type_expression* ')'

3.2.2 Semantics

The following type constructor, when applied to any type, generates an interface with the correct type for operations to check for equality and inequality of elements of that type.

```
type Equality(type T) is interface
    function =(rhs : T) return Boolean;
    function /=(rhs : T) return Boolean;
end Equality;
```

Elements of any type T such that T <: Equality(T) satisfy the following properties:

1. X = X returns True for any X. *(reflexive)*
2. X = Y and Y = X always return the same value. *(symmetric)*
3. If X = Y and Y = Z both return True, then X = Z must return True as well. *(transitive)*
4. X /= Y returns the same value as Not (X = Y).

Note that = identifies only an equivalence relation and not necessarily a congruence.

3.2.3 Examples

Example: *Set constructor*

```
type Set(type Element <: Equality(Element)) is interface
    ...
    function Member(E : Element) return Boolean;
    ...
end Set;
```

Commentary:

Consider the definition of a set type constructor. Since sets have a membership operation, it is appropriate to require the element type of the set to define equality. This can be enforced using the Equality type constructor.

□

3.3 Equality object expressions

There are no predefined modules specifically associated with Equality.

Chapter 4

Order

4.1 Purpose

Most predefined types have equality, inequality and other equivalence and ordering functions defined on their objects. These common operations are encapsulated in the type constructors Order, defined in this chapter, and Equality (see Chapter 3).

4.2 Order type expressions

4.2.1 Syntax

order_type_expression ::= Order'('*type_expression*')'

4.2.2 Semantics

The following type constructor provides the types for the usual ordering operations.

```
type Order(type T) is
    include Equality(T);
interface
    function <=(rhs : T) return Boolean;
    function <(rhs : T) return Boolean;
    function >(rhs : T) return Boolean;
    function >=(rhs : T) return Boolean;
end interface;
```

Since the type constructor Order has functions which are also defined in Equality, we reuse those functions by including Equality(T) in Order.

Elements of any type T such that T <: Order(T) satisfy all the equality constraints. In addition, they must satisfy the following as well:

1. X <= X returns True for any X. *(reflexive)*
2. If X <= Y and Y <= Z both return True, then X <= Z must return True as well. *(transitive)*
3. If both X <= Y and Y <= X return True, then X = Y must return True. *(anti-symmetric)*
4. If X = Y returns True, then X <= Y must return True.
5. X < Y returns True iff X <= Y returns True and X = Y returns False.
6. X > Y and Y < X always return the same value.
7. X >= Y and Y <= X always return the same value.

Note that these constraints impose a partial order. One could add a constraint of the form

- One of X <= Y and Y <= X must return True.

to require a total order.

4.2.3 Examples

Example: *Polymorphic sorting routine*

function Polymorphic_Sort(**type** T <: Order(T); L : List(T))
return List(T);

Commentary:

Consider the type of a polymorphic sorting routine, which takes a list of elements of some type and returns the sorted list. The list can be sorted only if the elements have an ordering on them. This can be expressed using the Order type constructor as a subtype constraint on the type parameter.

□

4.3 Order object expressions

There are no predefined modules specifically associated with Order.

Chapter 5

Triv

5.1 Purpose

Triv is the trivial type, with only one element, unit. Triv is used when one is not interested in the *values* of an object. Triv is typically used in the definition of recursive union and record types, such as finite trees (reftree-example).

[8] **What is the subtype relationship of Triv to other interface types? Is there a Triv for functions? Also, Root seems to be missing from the current document. Should we change this chapter to deal with all maxima and minima of the subtype tree?**

5.2 Triv type expressions

5.2.1 Syntax

triv_type_expression ::= Triv

5.2.2 Semantics

```
type Triv <: interface
    include Equality(Triv);
end Triv;
```

The following constraint is imposed on the behavior of the operations on Triv objects:

o = Unit returns True for all Triv objects o.

5.3 Triv object expressions

Triv literals

triv_literal ::= Unit

Unit is the only object of type Triv.

Chapter 6

Numeric types

The numeric predefined types are Integer, Float, Natural and Positive. The type Numeric defines the functionality common to floating-point numbers and integers.

6.1 Numeric type expressions

6.1.1 Syntax

numeric_type_expression ::= Integer | Float | Natural | Positive

6.1.2 Semantics

The operations on the numeric types are defined by the following series of interfaces and sets of constraints.

The interface Numeric defines the set of operations that are common to integers and floating-point numbers.

```
type Numeric (type T) is interface
    include Order(T);
    function +() return T;
    function −() return T;
    function +(rhs : T) return T;
    function −(rhs : T) return T;
    function ∗(rhs : T) return T;
    function ∗∗(rhs : T) return T;
    function /(rhs : T) return T;
    function mod(rhs : T) return T;
    function Abs() return T;
end Numeric;
```

The interface for integers is defined by using the Numeric constructor and adding integer-specific operations. The environment also defines the smallest and largest integer that it permits.

```
type Integer <: interface
      include Numeric(Integer);
      function Pred() return Integer;
      function Succ() return Integer;
      function Float() return Float;
end Integer;

Max_Integer : Integer;
Min_Integer : Integer;
```

The constraints on the integers (provided all integer computations are restricted to not exceed Max_Integer or be less than Min_Integer) are:

- Succ(Pred(I)) returns I.
  Pred(Succ(I)) returns I.
- I < Succ(I) returns True.
  Pred(I) < I returns True.
- I + 0 returns I.
- I + Succ(J) = Succ(I + J) returns True.
  I + Pred(J) = Pred(I + J) returns True.
- (I1 − I2 = I3) returns True if and only if (I1 = I2 + I3) does too.
- I * 0 returns 0.
- (I * Succ(J)) = (I * J) + I returns True.
  (I * Pred(J)) = (I * J) − I returns True.
- I ** 0 returns 1.
- (I ** Succ(N)) = (I * (I ** N)) returns True.
- If N1 > N2 returns True, then (N1 / N2) = ((N1 − N2) / N2) + 1 returns True, else (N1 / N2) = 0 returns True.
- ((N1 mod N2) = (N1 − (N2 * (N1 / N2)))) returns True.
- + I returns I.
- (− I = (0 − I)) returns True.
- If I < 0 returns True then **abs** I = − I returns True else **abs** I = I returns True.
- Max_Integer and Min_Integer are implementation-dependent.

Floating-point numerals are defined as:

```
type Float <: interface
      include Numeric(Float);
      function Floor() return Integer;
end interface;
```

[9] **What should be the contraints on floats?**

Naturals are simply

type Natural **is** Integer

with the constraint

- N >= 0 for all naturals N.

Similarly, the type of positive integers is

type Positive **is** Natural

with the constraint

- P >= 1 for all positives P.

6.1.3 Subtyping

- Positive, Natural and Integer are only distinct because of the constraints imposed, and are hence indistinguishable by the type system.
- There is no subtyping relationship between Integer and Float.

6.2 Numeric object expressions

6.2.1 Syntax

Numeric literals

numeric_literal	::=	*integer_literal* | *float_literal*
integer_literal	::=	{ *digit* }
float_literal	::=	*integer_literal* '.' *integer_literal* [*exponent*]
		| *integer_literal* '#' { *hex_digit* } '.' *integer_literal* '#' [*exponent*]
digit	::=	'0' | '1' | ... | '9'
hex_digit	::=	*digit* | 'a' | ... | 'f' | 'A' | ... | 'F'
exponent	::=	('e' | 'E') *integer_literal*

[10] **This isn't quite right. You can suck in the correct version from** `/usr/rapide/lrm/syntax/lexical.tex`

Numeric operations

All numeric operations can be invoked using the standard syntax corresponding to the interface expressions for Numeric.

Chapter 7

Character

7.1 Character type expressions

7.1.1 Syntax

character_type_expression ::= Character

7.1.2 Semantics

The definition for the Character predefined type includes an interface for characters and a function that maps the numeric character codes to the character they represent.

```
type Character is interface
    include Order(Character);
    function Code() return Integer;
end Character;

function Code_To_Char(i : Integer) return Character;
```

The constraints on characters are:

- The Code and Code_To_Char functions are one-to-one and inverses.
- Two characters are equal iff their numeric Code values are equal.
- All the ordering relations on characters are identical to the numeric ordering on their Code values.
- The Code function is such that the lower-case English alphabet, the upper-case English alphabet and the numeric digits are ordered in the standard alphabetic ordering of English. Beyond this, no other ordering is required (and in particular, the ASCII ordering need not hold).

7.2 Character object expressions

7.2.1 Syntax

Character literals

character_literal	::=	'’' *unquoted_character_literal* '’'
unquoted_character_literal	::=	'a' | ... | 'z' | 'A' | ... | 'Z' |
		'\n' | '\\' | '\''
		'*integer_literal*'

The escape characters on the second line represent, respectively, the characters for new line, backslash and single quote. Literals of the form `'\N'`, where N is an integer, simply denote the character returned by Code_To_Char(N).

Character operations

All character operations can be invoked using the standard syntax corresponding to the interface expression for Character.

Chapter 8

String

Strings are sequences of Character objects.

8.1 String type expressions

8.1.1 Syntax

string_type_expression ::= String

8.1.2 Semantics

The following interface and function types defines the set of operations on strings:

```
type String <: interface
     include Order(String);
     function Is_Null() return Boolean;
     function Length() return Integer;
     function [ ](Position : Integer) return Character;
     function &(S : String) return String;
     function Append(C : Character) return String;
     function Prepend(C : Character) return String;
     function Slice(Lbound, Ubound : Integer) return String;
     function String_To_Integer(S : String) return Integer;
end String;

function "''" return String;
function &(S : String, C : Character) return String;
function &(C : Character, S : String) return String;

type Caseless_String <: String;
```

[11] **In the current implementation, [ ] is overloaded to return prefixes, suffixes or substrings, based on the argument and return types.**

The constraints on strings are:

- (S.Prepend(C))[1] = C
  (S.Prepend(C))[N] = S[N-1] for $N > 1$
- (S.Append(C))[N] = S[N] if $N \leq$ S.Length()
  (S.Append(C))[S.Length+1] = C
- (S1 & S2)[N] = S1[N] if $N \leq$ S1.Length()
  (S1 & S2)[N] = S2[N - S1.Length()] otherwise
- "".Length() = 0.
- (S.Prepend(C)).Length() = S.Length() + 1
- (S.Append(C)).Length() = S.Length() + 1
- (S1 & S2).Length() = S1.Length() + S2.Length()
- (S1 = S2) if S1.Length() = S2.Length() and for $1 \leq i \leq$ S1.Length(), S1[i] = S2[i].
- (S1 < S2) if S1 = "" and S2 /= ""
  (S1 < S2) if S1[1] < S2[1] and S1.Slice(2, S1.Length()) < S2.Slice(2, S2.Length())
- S.Is_Null = (S = "")
- &(S,C) = S.Append(C)
- &(C,S) = S.Prepend(C)

The type Caseless_String differs from String in that the implementations of its functions are case-insensitive.

8.2 String object expressions

8.2.1 Syntax

String literals

string_literal ::= '"' { *unquoted_character_literal* } '"'

String operations

All String operations can be invoked using the standard syntax corresponding to the interface expressions for String.

Special syntax is provided for calling the [ ] and Slice operations.

A string can be indexed using the expression

expression '[' *expression* ']'

Thus, one could write S[I] instead of S.[ ](I).

The Slice function may be called using square backets enclosing two expressions, and using '..' to separate the expressions.

expression '[' *expression* '..' *expression* ']'

For example, S.Slice(I,J) could be written as S[I..J].

8.2.2 Examples

Example:

S1 : String **is** "Charlie";
S2 : String **is** "Chaplin";
X : String **is** S1 & S2;
Y : String **is** S1 & ' ' & S2;
Z : Character **is** X[4];

Commentary:

X is the string "CharlieChaplin" while Y is "Charlie Chaplin".

Z denotes the character 'r'.

□

Chapter 9

Record

A record is a finite, heterogeneous collection of labeled objects. Another way to describe records, which are familiar from Pascal and many subsequent languages, is as tuples whose components are selected by component name rather than position in the collection.

9.1 Record type expressions

9.1.1 Syntax

A Record type expression lists the component, or field, names and their types.

record_type_expression ::=
    record
        [*field_declarations*]
    end [**record**] [*identifier*]

field_declaration_list ::= *field_declaration* { *field_declaration* }
field_declaration ::=
    types_object_name_declaration |
    interface_derivation

All the field names must be distinct.

Record types may also be defined from other Record types. This works as for interface types, but the type(s) listed in the interface derivation(s) must be record type(s).

9.1.2 Semantics

A Record type:

```
record
    include ...;
    F1 : T1;
    F2 : T2;
    ...
    Fn : Tn;
```

```
    end record;
```

where Fi are identifiers and Ti are types, denotes the type

```
    interface
include ...;
        F1 : T1;
        F2 : T2;
        ...
        Fn : Tn;
    end interface;
```

9.1.3 Examples

1. Objects of the following Employee_Record have a Name component with a String object, and a Salary component with an Integer object.

```
type Employee_Record is
record
   Name : String;
   Salary : Integer;
end record;
```

2. The record type for managers can be defined by extending that for employees.

```
type Manager_Record is
record
   include Employee_Record;
   Dept : String;
end record;
```

   This type is equal to the following one:

```
type Manager_Record_Also is
record
   Salary : Integer;
   Dept, Name : String;
end record;
```

   Note that the reordering of the fields does not affect type equality.

9.1.4 Subtyping

The subtyping rules follow directly from those of interface types in general (see the RAPIDE 1.0 Types Language Reference Manual, Section 4.4):

A Record type R1 is a subtype of a Record type R2 if the set of fields of R2 is a subset of the set of fields of R1, and for every field F of R2, the type of F in R1 is a subtype of the type of F in R2. This may be viewed as two separate rules for subtyping — adding fields and replacing types of fields by subtypes. Subtyping for recursive record types is determined similar to that for interface types.

Example: *Record Subtyping*

```
type Employee_Record is
record
    Name : String;
    Salary : Integer;
end record;

type Teaching_Employee_Record is
record
    Name : String;
    Salary : Integer;
    Class : String;
end record;
```

Commentary:

The type Teaching_Employee_Record is a subtype of Employee_Record since Teaching_Employee_Record has all the fields of Employee_Record and some more fields.

Therefore, a Record of type Teaching_Employee_Record is also of type Employee_Record. In other words, a teaching employee is always an employee.

□

Example: *Record Subtyping by Replacing Field Types by Subtypes*

```
type Simple_Traffic_Light is enum Green, Red end enum;
type Traffic_Light is enum Green, Yellow, Red end enum;

type Simple_Intersection is
record
    Light1 : Simple_Traffic_Light;
    Light2 : Traffic_Light;
end record;

type Intersection is
record
    Light1, Light2 : Traffic_Light;
end record;
```

Commentary:

Since type Simple_Traffic_Light is a subtype of Traffic_Light, Simple_Intersection is a subtype of Intersection. Any object of type Simple_Intersection is also a object of type Intersection.

□

9.2 Record object expressions

9.2.1 Syntax

Record literals

A Record literal is a list of field name, field object pairs written in the following manner

record_literal ::= '(' [*field_object_list*] ')'
field_object_list::= *executable_named_association* { ',' *executable_named_association* }

Record operations

The only operation on Record objects is component selection.

9.2.2 Semantics

A record literal of the form (F1 **is** V1, ... Fn **is** Vn) is evaluated as

```
module
    F1 : T1 is V1;
    ...
    Fn : Tn is Vn;
end module;
```

Its type is a supertype of

```
type R is
interface
    F1 : T1;
    ...
    Fn : Tn;
end interface;
```

where T1, ... Tn are the types of V1, ..., Vn, respectively, as determined by the context.

9.2.3 Examples

1. (Name **is** "Jack", Salary **is** 45000)
   This is a Record object of type Employee_Record defined earlier.

[12] **Should we have equality for record types all of whose components have equality?**

Chapter 10

Union

Union types are *tagged unions*, (sometimes called *discriminated unions*) of one or more types. If U is the tagged union of S and T and o is an object of type U, then it is possible to extract from o an object of type S or type T, depending on the *tag* of o.

The types S and T are called the *member types* of U.

Objects of union types are called *union objects*.

The operations associated with union types are *literals, extraction* of member type objects, *extraction* of the tag of a union object, and *discrimination* based on tags.

There is also a polymorphic type constructor which for a given union type constructs an enumeration type with one member for each of the tags (see Chapter 11).

10.1 Union type expressions

10.1.1 Syntax

union_type_expression ::= **union**
    { *types_object_name_declaration* }
  end [**union**]

A type expression for a Union type gives a list of member types, each prefixed by an identifier called its *tag*.

A member type may occur more than once in a Union type but a given *tag* may occur only once. For each tag-type pair, the type is said to be the one *associated* with the tag.

10.1.2 Semantics

A union type object has two components, a tag and an object of the member type associated with that tag. The member type may be obtained by applying the extraction operation to the object, and a conditional expression based upon the tag of the object can be constructed using discrimination.

Union types, as all other predefined types, can be expressed in terms of interfaces and function types. For union types, however, this reduction is fairly subtle, and the interested reader is referred to Section 10.3

10.1.3 Examples

An example of a Union type of Integer and Boolean:

Example: *Union of Integer and Boolean Types*

```
type Int_or_Bool is
union
    Int : Integer;
    Bool : Boolean;
end union;
```

Commentary:

This Union type has two member types — Integer and Boolean. The tag of Boolean is Bool and the tag of Integer is Int. An object of type Int_or_Bool may be created from an object of either type, and the tag will identify from which it was created.

□

Example: *Building Variant Records from Union Types*

```
type Marital_Status is
union
    Spouse : Name_Type;
    No_Spouse : Zodiac_Sign;
end union;
type Person is record
    N : Name_Type;
    S : Marital_Status;
end record;
```

Commentary:

This use of a union type in a record type allows us to define variant record types as in Ada or other languages.

Person records would be constructed so that the S field corresponds with the marital status of the N field. Depending on whether a given person is married or not S would either be a sign from the Zodiac (when unmarried) or identify the spouse (when married). When given an object of type Person one could check whether the person is married or not by checking the tag of the S field (using the discrimination operation) and then act appropriately, making use of the Zodiac sign or the name of the spouse, as the case may be.

□

10.1.4 Subtyping for Union types

A union type U1 is a subtype of a union type U2 if the set of tags of U1 is a subset of the set of tags of U2, and for every tag T of U1, the type associated with T in U1 is a subtype of the type associated with T in U2.

Example: *Union Subtyping by Removing Field Declarations*

```
type Char_Int_or_Bool is
union
      Int : Integer;
      Bool : Boolean;
      Char : Character;
end union;

type Int_or_Bool is
union
      Int : Integer;
      Bool : Boolean;
end union;
```

Commentary:

The type Int_or_Bool is a subtype of Char_Int_or_Bool since Int_or_Bool is made up of some the member types of Char_Int_or_Bool. Therefore, a union object of type Int_or_Bool is also of type Char_Int_or_Bool.

□

Example: *Union Subtyping by Replacing Member Types by Subtypes*

```
type Employee_Id;
type Manager_Id <: Employee_Id;

type Employee_Key is
union
      Name : String;
      Id : Employee_Id;
end union;

type Special_Employee_Key is
union
      Name : String;
      Id : Manager_Id;
end union;
```

Commentary:

Information about employees may be identified by a name or some identification key (whose structure need not concern us here). Managers are identified by special identification keys, a subtype of the general identification keys. Since Manager_Id is a subtype of Employee_Id, the union type Special_Employee_Key is a subtype of Employee_Key.

In other words, a Special_Employee_Key object is also an object of type Employee_Key.

□

10.2 Union object expressions

Objects of union types can be constructed from a tag and an object of the associated member type. One can perform the following operations on objects of union types – *extraction* of a member type object from a union type object using a tag; *discrimination*, which allows construction of a conditional expression based upon the tag of a union type object; a polymorphic type constructor *tags* which for a union type constructs an enumeration type with one object for each tag, and a *tag* operation which extracts the tag of a union object. There are also predefined *equality* operators (= and /=) for union types whose member types have an equality operation.

10.2.1 Syntax

Union Literals

A union type literal consists of a tag and an object of the type associated with the tag.

union_literal ::= '(' *identifier* ',' *expression* ')'

The identifier must be a tag of the union type of the literal. This type is determined by the context. The *expression* must be of the type associated with the tag.

Union operations

The representation of union types as polymorphic function types provides syntax for the use of union types. However, since this syntax isn't very natural, special syntax is provided for the operations on union types.

Object extraction: The member type of a union object may be extracted using the tag. The tag is applied to the union object with the dot notation.

union_extraction ::= *union_object_expression* '.' *tag_identifier*

The *union_object_expression* must be of a union type having the *tag_identifier* among its tags, otherwise the extraction is semantically invalid and results in a compile time error. If *tag_identifier* is the tag of the *union_object_expression* then the extraction denotes its member type, otherwise the extraction raises the exception UNION_EXTRACTION.

[13] **The whole exception mechanism and naming scheme is still unresolved.**

Discrimination: A discrimination is a conditional expression of a union object which depends upon the tag of the object.

union_discrimination ::=
  case ([*identifier* ':' *union_type_expression* **is**] *union_object_expression* ','
    tag_alternative
    {',' *tag_alternative*}
    [',' **default is** *expression*]
)

tag_alternative ::= *executable_named_association*

The *union_object_expression* must evaluate to a union type object of the *union_type_expression* type (if given). If the optional identifier and *union_type_expression* are not given then the type is determined from the type of the *union_object_expression*.

Each tag of the union type must occur in only one *tag_alternative*, and must occur in an alternative unless there is a **default** *alternative*; all tag alternative expressions must be of the same type, which is then the type of the *union_discrimination* expression.

If T is a union type with tags $\mathsf{C}\Delta 1, \ldots,\ \mathsf{C}\Delta n$ and *obj* is a union object of type T with tag $\mathsf{C}\Delta i$, then **case** (*obj*, $\mathsf{C}\Delta 1$ **is** $f\Delta 1$, ..., $\mathsf{C}\Delta n$ **is** $f\Delta n$, **default is** $f\Delta e$), is evaluated by evaluating the expression $f\Delta i$. In case there is no alternative for $\mathsf{C}\Delta i$, the expression f_e is evaluated.

Tag extraction: [14] **Do we still have this?**

For each union type U with tags TAG1, TAG2,..., TAGn, there exists a predefined

function Tagof(p: U) **return enum** TAG1, TAG2,..., TAGn **end enum**

which extracts the tag of a union type object.

Equality: [15] **Is this true?**

If equality is defined for all the member types of a union type then equality is implicitly defined for the union type.

For two union type objects o1 and o2 the expression o1=o2 evaluates to true if (1) the tag of o1 is identical to the tag of o2 and then (2) the object used when creating o1 is equal to the object used when creating o2.

10.2.2 Examples

1. The following example illustrates the use of unions.

```
type Employee_Key is
union
      Name : String;
      Id : Employee_Id;
end union;
...
x1: Employee_Key is (Name, "Jane Doe");
x2: Employee_Key is (Id, Get_Id());
...
function Lookup(P: Employee_Key) return Person is
begin
   return(
      case (P,
              Name is Retrieve_From_Name(P.Name),
              Id    is Retrieve_From_Id(P.Id)
           ));
end Lookup;
...
```

Employees may be identified by name or by some identification key.

Assume the function Retrieve_From_Name retrieves information from a personel database on the basis of a given string, and Retrieve_From_Id does so on the basis of a given Id. Lookup(x1) will execute the Name alternative resulting in a retrieval on the record identified by the string "Jane Doe". Lookup(x2) will execute the Id alternative resulting in a retrieval based on an id.

Both retrieval functions must return objects of type Person, which could then be subjected to further tag discrimination. The last two expressions will print the marital status of the person, depending on whether the S field contains an object tagged with the Spouse or No_Spouse tag.

2. The following example illustrates the use of union types used to define recursive data structures.

```
type Tree(type T) is
union
        Empty : Triv;
        Node : record
                        V : T;
                        Left, Right: Tree(T);
                  end record;
end union
```

Any object of type Tree(T) for some type T is either an empty tree, or it is a node with a value, and two subtrees.

Objects of this type could be

```
Empty_Tree: Tree(integer) is (Empty, Unit);

Some_Tree: Tree(integer) is
(Node,
     (V is 1,
      Left is (Node,
                   (V is 2,
                    Left is Empty_Tree,
                    Right is Empty_Tree)),
      Right is (Node,
                    V( is 3,
                    Left is (Node,
                                 (V is 4,
                                  Left is Empty_Tree,
                                  Right is Empty_Tree))
                    Right is Empty_Tree))))
```

[16] **Some picture guru: Please draw the tree here.**

3. The following example illustrates the use of equality on union types.

```
type String_or_Number is
union
       S : String;
       N1 : Integer;
       N2 : Integer;
end union;

o1: String_or_Number is (S, "Jane Doe");
```

```
o2: String_or_Number is (N1, 42);
o3: String_or_Number is (N2, 42);
o4: String_or_Number is (N1, 42);
...
... o1 = o2 ... -- False, tags mismatch
... o2 = o3 ... -- False, tags mismatch
... o2 = o4 ... -- True, tags match, and the extracted objects are equal
```

Since equality is defined for both strings and integers, it is also defined for the type String_or_Number. The expression o1=o2 is true if both are tagged with S, and where created from equal strings, or both are tagged with N1, and where created from equal integers, or both are tagged with N2, and where created from equal integers. Equality of integer values is not sufficient, if the tags are distinct.

10.3 Reduction of union types to function types

Semantically, union types may be defined in terms of function types. This well known reduction implies the subtyping rules described above. Below we outline how this reduction provides the correct subtyping for union types, and how it defines the creation, extraction, discrimination, equality and tag extraction operations.

The union types as function types

A union type declaration:

```
type U is
union
     TAG1 : T1;
     TAG2 : T2;
     ...
     TAGn : Tn;
end union;
```

where TAGi are identifiers and Ti are type expressions, is shorthand for the declaration of a polymorphic function type with two parameters:

```
type U is function ( -- two parameters: A type and a record of functions
          type T;
          record
               TAG1 : function (o : T1) return T;
               TAG2 : function (o : T2) return T;
               ...
               TAGn : function (o : Tn) return T;
          end record ) return T;
```

An object *obj* of a union type U is thus interpreted as a function taking a type and a record parameter, where the record has as components a set of functions corresponding to the tags. Each function maps objects of the member type associated with the tag to objects of the type T. The union subtyping follows from the general rules for subtyping of function types.

Union literals as function literals

A union type literal denotes a particular function. If e is of type Ti, then the evaluation of

```
U'(TAGi, e)
```

results in an evaluation of e returning some object o and then returns the function object

```
function (type T;
          P: record
                 TAGi: function(o : Ti) return T
             end record) return T is
begin
      return P.TAGi(o);
end
```

The type of the formal parameter P is a supertype of the corresponding formal parameter of the function type U:

```
    record
          TAG1 : function (o : T1) return T;
          TAG2 : function (o : T2) return T;
          ...
          TAGn : function (o : Tn) return T;
    end record
```

Consequently, the literal denotes a function of a subtype of U, which makes it an object of type U.

Extraction

Extraction has the semantics of a call to an extraction function, one for each member type of the union type.

The expression *obj*.TAGi where *obj* is of a union type with TAGi as one of its tags whose associated type is Ti, corresponds to the expression

```
obj(Ti,
   (T1 is function f1(m: T1) return Ti is
          begin raise Union_Error; end;
     ...
    Ti is function fi(m: Ti) return Ti is
          begin return(m) end;
     ...
   )
  )
```

i.e., when evaluating *obj*.TAGi, if *obj* does not have the tag TAGi then the function associated with the tag will raise the exception. If its tag *is* TAGi then the object of type Ti used when creating *obj* will be returned.

Discrimination

Assume given the expression

```
case(x: U is e,
        TAG1 is e1(x),
        TAG2 is e2(x),
        ...
        TAGn is en(x),
    default is ee(x)
    )
```

with the type of the expressions being T. This evaluates to the value of the expression

```
e(T,
    (TAG1 is function (x: T1) return T is
              begin return e1(x); end;
      ...
     TAGn is function (x: Tn) return T is
              begin return en(x); end;
     TAGm is function (x: Tm) return T is
              begin return ee(U'(TAGm,x)); end;
      ...
     TAGp is function (x: Tp) return T is
              begin return ee(U'(TAGp,x)); end;
    )
  )
```

where TAGm through TAGp are the tags of the type U not explicitly listed in the case expression. Since each of the expressions e1 through en of TAG1 through TAGn expect the identifier x to denote an object of the type associated with the tag we need not qualify x. However, the expression associated with the **default** alternative expects x to be of type U. Consequently we need to replace occurrences of x with the union type literal corresponding to the particular tag.

The Tagof function

The function Tagof evaluates to an object of an enumeration type, corresponding to the tag of the actual parameter. It is simply defined:

```
function Tagof(p: U) return enum TAG1, ..., TAGn end enum is
begin
    return
        (case (p,
                TAG1 is Tags(U)'TAG1;
              ...
                TAGn is Tags(U)'TAGn;
              ));
end;
```

Equality

Given the type

```
type U is
union
    TAG1 : T1;
    TAG2 : T2;
    ...
    TAGn : Tn;
end union;
```

If equality is defined for all the member types of U, then equality for U is defined by:

```
function "="(o1, o2: U) return Boolean is
begin
    return(case(o1,
                TAG1 is case(o2,
                            TAG1 is o1 = o2, -- T1 equality
                            default is false,
                            )
                ...
                TAGn is case(o2,
                            TAGn is o1 = o2, -- Tn equality
                            default is false,
                            )
                ));
end;
```

Chapter 11

Enumeration

An enumeration type is a type that is defined by explicitly listing its member objects.

[17] **Double check: we're all happy with unordered enumeration types?**

11.1 Enumeration type expressions

11.1.1 Syntax

An enumeration type expression lists all of the objects of the enumeration type, each identified by the identifier given.

enumeration_type_expression ::= **enum** *identifier* {',' *identifier* } **end** [**enum**]

Identifiers must not be repeated in an enumeration type expression.

11.1.2 Semantics

An enumeration type expression is shorthand for a corresponding union type expression. The declaration

type E **is** **enum** I1, I2, ..., In **end** **enum**

represents the type declaration

type E **is** **union** I1, I2,..., In: **triv**; **end** **union**

with the constraint that the list of identifiers represents an *exhaustive* list of the objects of the type.

11.1.3 Examples

1. A simple example of enumeration types is the type of traffic lights.

type Traffic_Lights **is** **enum** Red, Yellow, Green **end** **enum**;

11.1.4 Subtyping

Since an enumeration type expression is shorthand for a union type expression in which all the tags (enumeration identifiers) are associated with type `triv`, it follows that an enumeration type E1 is a subtype of an enumeration type E2 if the set of identifiers listed in E1 is a subset of the identifiers listed in E2.

11.2 Enumeration object expressions

11.2.1 Syntax

Enumeration literals

Enumeration_literal ::= *identifier*

The type of the literal is determined by the context (as defined by the executable language), and the identifier must belong to this enumeration type.

The denotation of an identifier “`n`” as an enumeration object expression is “`(n, Unit)`.

[18] **This definition strikes me as bogus. There should be a nice way to introduce the enumeration literal using the usual visibility rules of the language. One way to do this is to create for each literal an implicit declaration when an enumeration type is declared. This is what the current implementation does, but there's a problem: it doesn't work for anonymous enumeration types.**

[19] **Should include a discussion of how to define an iterator over an enumeration type (especially if there's no order).**

Chapter 12

Iteration

This chapter defines a set of type constructors that are meant for inclusion in the definition of other interface types to add standard operations for iteration.

[20] **Make sure this definition is adequate for the uses of iteration (i.e. to define iteration of enumerations, that ranges are a syntax shorthand for an iterator over a discrete type, etc.**

12.1 Iteration type expressions

12.1.1 Syntax

iteration_type_expression ::=
    Iterator'('*type_expression*')' |
    Iterable'('*type_expression*')' |
    Discrete'('*type_expression*')'

12.1.2 Semantics

```
type Iterator (type T) is interface
   Item  : function () return T;
   More  : function () return Boolean;
end Iterator;

type Iterable (type T) is interface
   Iterator : module () return Iterator(T);
end Iterable;

type Discrete (type T) is interface
   <     : function (rhs : T) return Boolean;
   =     : function (rhs : T) return Boolean;
   Succ  : function () return T;
end Discrete;
```

Type T is said to be *iterable* if T <: Iterable(T) and *discrete* if T <: Discrete(T).

12.1.3 Examples

1. Type Integer is discrete, while enumeration types and type Float are not.

12.1.4 Subtyping

The subtypingrules follow directly from the general subtyping rules for interface types (see the RAPIDE 1.0 Types Language Reference Manual, 4.4):

- Iterator(T1) is a subtype of Iterator(T2) iff T1 is a subtype of T2.
- Iterable(T1) is a subtype of Iterable(T2) iff T1 is a subtype of T2.
- Discrete(T1) is a subtype of Discrete(T2) iff T1 is equal to T2.

12.2 Iteration object expressions

There are no predefined modules specifically associated with Iterator, Iterable and Discrete.

Chapter 13

Range Type

13.1 Range

Range types are used for defining subranges of discrete types. A typical usage of ranges is to define indices of arrays.

[21] **We don't want Range types anymore. Ranges are iterators. This chapter should be rewritten and moved to a subsection of the Iteration chapter.**

13.2 Range type expressions

13.2.1 Syntax

Ranges are defined by the following interface type and module generator declaration:

range_type_expression ::= Range '(' *types_type_expression* ')'

13.2.2 Semantics

The Range type constructor is defined as

```
type Range(type T <: discrete(T)) is
      first, last : T;
      function Iterator() return Iterator(T);
end;
```

The type constructor Range creates a new range type for a discrete type. Range is a subtype of iterable:

Range(T) <: Iterable(T)

13.2.3 Subtyping

The RAPIDE type system does not permit distinguishing types based on values (objects). That is, any two types created by applying a type constructor parameterized by an object cannot be distinguished from each other regardless of the objects used as parameters. Thus there is now way to distinguish the *types* for different subranges of integer.

Thus all subranges of integer have the same type, and range objects are used to denote specific subranges. This can seem cumbersome to an Ada programmer, but it is the price we pay for complete separation of interface and implementation.

Also, it does tend to result in a greater degree of reusability. For example, it tends to encourage functions that operate on ranges to be written in such a way as to operate on any subrange of the appropriate type, and not just a particular subrange. As we'll see in defining arrays, in Ada terms this means that functions operating on Arrays are generally forced to be written such that they will work for any size array (unconstrained arrays). In Pascal, Ada, or C writing functions that work with arbitrary sized arrays can require careful though, whereas it is the natural way to write the function in RAPIDE.

13.3 Range object expressions

13.3.1 Syntax

module Range(type T <: discrete(T); mfirst, mlast : T) return Range(T);

The module generator creates a new range object. Range objects store the actual upper and lower bounds of the range. The value of the bounds are NOT part of the type as is common with languages like Ada.

Range literals

range_literal ::= *executable_expression* '..' *executable_expression*

The Range literal `min .. max` denotes the range object `Range(T, min, max)` where thetype `T` is determined by the types of the expressions `min` and `max` and the context of the literal.

Range operations

All Range operations can be invoked using the standard syntax corresponding to the interface expressions for `Range`.

13.3.2 Examples

Example:
int1to10 : Range(Integer) IS Range(Integer,1,10);
Commentary:

This defines an integer range 1 to 10. A for loop interating over this range is written:

```
for i : integer in int1to10 do
    ...
end;
```

□

Chapter 14

Array

An array type declares a mapping from objects of its *index* type to objects of the array's *element* type.

14.1 Array type expressions

14.1.1 Syntax

array_type_expression ::=
    array '['*type_expression* {',' *type_expression* }']' **of** *type_expression*

14.1.2 Semantics

The types between the brackets are the *index types* of the array type.

array [T1, T2, ...] **of** T is equivalent to **array** [T1] **of** **array** [T2] **of** ... **of** T.

In the most general case, the type expression
    array [Index_Type] **of** Elem_Type
is equivalent to the expression
    Array(Index_Type, Elem_Type)
where the Array type constructor is defined as:

```
type Array (type Index; type Element) is interface
    include Iterable(Index);
    [ ] : function (I : Index) return Element;
end Array;
```

For a *discrete* Index_Type (i.e., Index_Type <: Discrete(Index_Type)), the type expression
    array [Index_Type] **of** Elem_Type
is equivalent to
    Discrete_Array(Index_Type, Elem_Type)
where the Discrete_Array type constructor is defined as:

```
type Discrete_Array (type Index <: Discrete(Index);
                     type Element) is interface
    include Array(Index, Element);
    Slice       : function (First, Last : Index) return Discrete_Array(Index, Element);
    =, /=       : function (rhs : Discrete_Array(Index, Element)) return Boolean;
    First, Last : function () return Index;
    Length      : function () return Integer;
end Discrete_Array;
```

The Iterator and Discrete type constructors are declared in Chapter 12.

14.1.3 Examples

Example:

```
type A1 is array [Integer] of Float;
type A2 is array [Integer, Integer] of Boolean;
type A3 is array [Integer, Float] of Boolean;
```

Commentary:

> Type A1 is an integer-indexed array of floating-point numbers. Since integers are discrete, this is equivalent to Discrete_Array(Integer, Float).
>
> A2 is the type of a boolean matrix, and is equivalent to Discrete_Array(Integer, Discrete_Array(Integer, Boole
>
> A3 is the type of a two-dimensional matrix of booleans, one dimension of which is indexed by integers and the other by floating-point numbers. This is equivalent to Discrete_Array(Integer, Array(Float, Boolean)).

□

14.1.4 Subtyping

The subtyping of arrays follows directly from the general subtyping rules of interface types, i.e. an array type A1 is a subtype of array type A2 if the index type of A2 is a subtype of the index type of A1, and the element type of A1 is a subtype of the element type of A2.

```
array [S1] of T1 <: array [S2] of T2
                 iff
      S2 <: S1 and T1 <: T2
```

14.2 Array object expressions

14.2.1 Syntax

Array literals

Array literals are written using an aggregate notation with the following syntax:

array_literal ::= '(' [*init_range*] *association* { ',' *association* } ')'
init_range ::= *expression*
association ::= *positional_association* | *named_association* | *others_association*

positional_association ::= *expression*
named_association ::= *expression* **is** *expression*
others_association ::= **default is** *expression*

The type of the array object is determined by the context of the aggregate. (The qualified expression form of an aggregate can be used to fix the type.) [22] **put reference here.** The *init_range* must be of type Iterable(I) where I is the index type. If an *init_range* is not given then only named associations and "default" associations are allowed.

"Default" association is only allowed as at most one association of an aggregate, and only as the last association of the aggregate.

The expresson on the right side of a "default" association is evaluated once per index it associates. If an *init_range* is given then the "default" expression is evaluated for the indices returned by the iterator as part of the evaluation of the aggregate. Hence, the following two aggregates would be equivalent.

(1 .. 3, **default is** F())
(1 **is** F(), 2 **is** F(), 3 **is** F(), **default is** F())

For indices not identified by the *init_range*, the expression is evaluated the first time the indexing function ([ ]) of the array object is called with an index for which a value hasn't been associated by name or position. [23] **I think this is at odds with the current implementation (and possibly what we want), where the init_range defines the full range of the array, and references outside the range cause an error.**

Example:

A : **array** [Character] **of** Integer **is** (**default is** F());
I : Integer **is** A['b']; -- Evaluates F().
J : Integer **is** A['b']; -- Does not evaluate F().
K : Integer **is** A['c']; -- Evaluates F().

□

Array operations

All array operations can be invoked using the standard syntax corresponding to the interface expressions for the Array and Discrete_Array type constructors.

Special syntax is provided for calling the [ ] and Slice functions.

An array can be indexed using the expression

expression '[' *expression* { ',' *expression* } ']'

Thus, one could write A[I1,I2] instead of A.[ ](I1).[ ](I2) and so on.

The Slice function may be called using square backets enclosing two expressions, and using '..' to separate the expressions.

expression '[' *expression* '..' *expression* ']'

For example, A.Slice(L,U) could be written as A[L .. U].

Assignment is not supported on arrays. Programmers wishing to allow assignment on individual elements of an array should make these elements Ref types. Programmers wishing to allow assignment on entire arrays should use the type Ref(*array_type*).

14.2.2 Examples

Example:
Names : Array(Integer, String) **is** (1 .. 3, "Bill", "Mary", "George");
Commentary:

This declares Names to be an array of the three given strings.

□

Example: *Array of a Ref type*
Names : **array** [Integer] **of** Ref(String) **is** (1 .. 3, **default is** Nil(String));
Commentary:

This declares Names to be an assignable array of strings, all initialized to a nil reference of type String. One could then say, for instance,
Names[3] := "Sue";

□

Chapter 15

Set

15.1 Set type expressions

15.1.1 Syntax

set_type_expression ::= Set'('*type_expression*')'

15.1.2 Semantics

The Set type constructor is:

```
type Set(type Element <: Equality(Element)) is interface
    include Equality(Set(Element));
    function Is_Member(E : Element) return Boolean;
    function +(E : Element) return Set(Element);
    function &(S : Set(Element)) return Set(Element);
    function −(E : Element) return Set(Element);
    function −(S : Set(Element)) return Set(Element);
    function <(S : Set(Element)) return Boolean;
    function Cardinality() return Natural;
end Set;

function Empty_Set(type Element <: Equality(Element)) return  Set(Element);
```

[24] **We could use the Order type constructor to get all the subset operations.**

Sets are only defined for element types with equality.

The following constraints are imposed on sets:

- { }.Is_Member(E) = False.
- (S + E1).Is_Member(E2) = True iff S.Is_Member(E2) = True or E1 = E2.
- (S1 & S2).Is_Member(E) = True iff S1.Is_Member(E) = True or S2.Is_Member(E) = True.

- (S − E1).Is_Member(E2) = True iff S.Is_Member(E2) = True and E1 /= E2.
- (S1 − S2).Is_Member(E) = True iff (S1.Is_Member(E) = True) and (S2.Is_Member(E) = False).
- S1 < S2 iff S1.Is_Member(E) = True implies S2.Is_Member(E) = True for all E of the element type.
- S1 = S2 iff S1.Is_Member(E) = S2.Is_Member(E) for all E of the element type.
- { }.Cardinality() = 0.
- (S + E).Cardinality() = S.Cardinality() if S.Is_Member(E) = True
  (S + E).Cardinality() = S.Cardinality() + 1 otherwise.
- If S1 = S2 then S1.Cardinality() = S2.Cardinality()

Note that the type of the expression { } is determined by the context of the expression.

15.1.3 Examples

Example:
type Integer_Set **is** Set(Integer);
Commentary:

This defines the type of sets of integers by applying the Set type constructor to the type Integer.

□

15.1.4 Subtyping

Set(E1) is a subtype of Set(E2) iff E1 is type equivalent to E2.

15.2 Set object expressions

15.2.1 Syntax

Set literals

set_literal ::= '{' '}' | '{' *expression* {',' *expression*} '}'

All expressions must be of the type of the set. This type is determined by the context, as defined by the executable language.

A literal of the form {A,B,...} is equivalent to { } + A + B + ...

Set operations

All Set operations can be invoked using the standard syntax corresponding to the interface expressions for Set.

Chapter 16

Ref

References *refer to* objects. References are objects of reference types. A reference type expression R must mention the type of the referenced objects.

[25] **There is currently no way to make a ref object Nil once it has been assigned to.**

16.1 Ref type expressions

16.1.1 Syntax

ref_type_expression ::= Ref'('*type_expression*')'
| **var** *type_expression*

16.1.2 Semantics

The operations on references are defined by the following interface and function types.

```
type Ref(type T) is interface
    function :=(rhs : T) return Ref(T);
    function $() return T;
    function Is_Nil() return Boolean;
end Ref;

module Ref_To(type T; rhs : T) return Ref(T);

function Nil(type T) return Ref(T);
```

The operation R:=X makes the reference R refer to X. The operation $R returns the object that R refers to. The operation R.Is_Nil() checks whether R refers to an object.

The call Ref_To(T,X) creates a reference of type Ref(T) which refers to X.

The call Nil(T) creates a reference of type Ref(T).

References satisfy the following constraints:

- $(R := X) = X
- $(Ref_To(T,X)) = X
- $(Nil(T)) returns an exception. [26] **Which one?**
- (R := X).Is_Nil() = False
- Ref_To(T,X).Is_Nil() = False
- Nil(T).Is_Nil() = True

The var declaration provides alternative syntax for objects declared using it. The declaration "R : **var** T" is equivalent to the declaration "R : Ref(T)", but within the scope of this declaration, the expression &R is interpreted simply as R, and all occurences of R are reinterpreted as $R.

The use of references introduces certain causalities in the default executable language. This is desribed in detail in Section 16.3.

16.1.3 Examples

Example:

```
type References_To_Integers is Ref(Integer);
X : References_To_Integers is Ref_To(Integer,0);
...
X := $X + 1;
```

Commentary:

The type References_To_Integers is the type of references to Integer objects. One can then define objects of this type and "assign" them values.

□

Example: *Var syntax*

```
X : var Integer := 0;
...
&X := X + 1;
```

Commentary:

Using the var syntax we equivalently express the first example.

□

16.1.4 Subtyping

Ref(T1) $<:$ Ref(T2) if and only if T1 and T2 are type-equivalent.

16.2 Ref object expressions

The only way to create Ref objects is by calls to the Ref_To and Nil functions.

16.3 References and Causality

Operations on reference objects impose certain causality orderings on the processes invoking these operations, as described below.

All $ and := functions on a reference object R are executed in some sequential order, even though the calls may be executed concurrently. The return from any $ function invoked on R is causally dependent on the return from the most recent := function in that sequential order.

Note, however, that such a call to := need not always exist. The following is an example of where it doesn't:

```
X : Ref(Integer) is Ref_To(Integer, 12);
I : Integer is $X;
```

In this case, the return from the $ function is causally dependent on the return from the function call (Ref_To or Nil) that creates the reference.

Chapter 17

Clock

Clocks are monotonically increasing counters which provide a notion of simulation time. Events are timed with respect to (one or more) clocks and have start and finish times with respect to each of these clocks. [27] **Need a reference back to the Executable LRM where clocks are discussed.**

17.1 Clock type expressions

17.1.1 Syntax

clock_type_expression ::= Clock | Synchronous_Clock | Regular_Clock | Slaved_Clock

17.1.2 Semantics

The types and constraints that define the various kinds of clocks are described below.

Each clock defines a type Ticks, which are values of the clock's counter.

```
type Clock is interface
    type Ticks <: Natural;
    function Now() return Ticks;
    function Start(E : event()) return Ticks;
    function Finish(E : event()) return Ticks;
    function Length(E : event()) return Ticks;
end Clock;

module Make_Clock() return Clock;
```

The constraints on clocks are:

- C.Start(E) <= C.Finish(E)
- C.Length(E) = C.Finish(E) − C.Start(E)

- If E1 < E2 then C.Finish(E1) <= C.Start(E2)

Synchronous clocks relate the time of a clock to a common metric called the GST (Global Standard Time).

```
type GST <: Float;

type Synchronous_Clock is interface
    include Clock;
    function GST(T : Ticks) return GST;
    function Distance(T1, T2 : Ticks) return GST;
    function Ticks(D : GST) return Ticks;
end Synchronous_Clock;

module Make_Synchronous_Clock() return Synchronous_Clock;
```

The behavior of synchronous clocks is described by the following constraints:

- C.Distance(T1,T2) = Abs(C.GST(T1) − C.GST(T2))
- C.Ticks(C.GST(T)) = T
- Abs(C.GST(C.Ticks(G)) − G) <= Abs(C.GST(T) − G) for all T
  (the Ticks function makes the closest mapping to GST)
- If T <= C.Now() then C.GST(T) = C.Distance(0,T)
- If T1.<=(T2) and T2.<=(T3),
  then C.Distance(T1,T2) + C.Distance(T2,T3) = C.Distance(T1,T3)
- If T1.<(T2) then C.GST(T1) < C.GST(T2)
- If C.GST(T1) = C.GST(T2) then T1.=(T2)

Note that the function Ticks is not required to be one to one. Also, distances are not required to be linearly increasing with ticks.

Regular clocks are synchronous clocks with a specified period and accuracy. The period relates the time of the clock with that of the GST. Accuracy can be defined either in terms of an interval measure or a ratio - in either case, 1.0 implies perfectly accurate and 0.0 indicates complete inaccuracy.

```
type Accuracy is record
    Kind : enum Interval, Ratio end enum;
    Measure : Float range 0.0 .. 1.0;
end Accuracy;

type Regular_Clock is interface
    include Synchronous_Clock;
    Period : Float;
    Precision : Accuracy;
end Regular_Clock;
```

```
module Make_Regular_Clock(Period : Float;
                          Precision : Accuracy) return Regular_Clock;
```

The constraints on regular clocks are:

- If C.Precision.Kind = Interval then for distinct T1, T2
  ```
  C.Precision.Measure <= (C.GST(T1) - C.GST(T2)) / C.Period*(T1 - T2)
                      <= 2.0 - C.Precision.Measure
  ```
- If C.Precision.Kind = Ratio and C.Precision.Measure /= 0.0 then for distinct T1, T2
  ```
  C.Precision.Measure <= (C.GST(T1) - C.GST(T2)) / C.Period*(T1 - T2)
                      <= 1.0 / C.Precision.Measure
  ```

Slaved clocks relate their ticks to those of another clock (called the Master).

```
type Slaved_Clock is interface
    include Synchronous_Clock;
    Master : Synchronous_Clock;
    Period : Float;
    Precision : Accuracy;
end Slaved_Clock;

module Make_Slaved_Clock(Master : Synchronous_Clock;
                         Period : Float;
                         Precision : Accuracy) return Slaved_Clock;
```

The constraints on slaved clocks are:

- If C.Precision.Kind = Interval then for distinct T1, T2
  ```
  C.Precision.Measure <= (C.Master.GST(T1) - C.Master.GST(T2)) /
                          C.Period*(C.GST(T1) - C.GST(T2))
                      <= 2.0 - C.Precision.Measure
  ```
- If C.Precision.Kind = Ratio and C.Precision.Measure /= 0.0 then for distinct T1, T2
  ```
  C.Precision.Measure <= (C.Master.GST(T1) - C.Master.GST(T2)) /
                          C.Period*(C.GST(T1) - C.GST(T2))
                      <= 1.0 / C.Precision.Measure
  ```

17.1.3 Examples

1. The following example shows a slaved clock and its master.

   ```
   Exact : Accuracy is (Interval, 1.0);
   M : Regular_Clock is Make_Regular_Clock(2.0, Exact);
   S : Slaved_Clock is Make_Slaved_Clock(M, 2.0, Exact);
   ```

   M is perfectly accurate, with a period of 2.0 relative to GST. S is a slave of M, is also perfectly accurate. M is twice as fast as S.

17.1.4 Subtyping

- Slaved_Clock $<:$ Regular_Clock $<:$ Synchronous_Clock $<:$ Clock

17.2 Clock object expressions

Clock objects are generated by calls to the clock generator functions Make_Clock, Make_Synchronous_Clock, Make_Regular_Clock and Make_Slaved_Clock.

Chapter 18

Predefined Pattern Macros

18.1 Introduction

TBD. [28] **What about dropping the last parameter in implementations that support only global clocks?**

18.2 Syntax

```
pattern Conflict( pattern P;  Var : Ref(Root) ) is
    P || Var.:='Return;

pattern at( pattern P; Time : Integer; C : Clock ) is
    timing(P,Time,Time,C);

pattern before( pattern P; Time : Integer; C : Clock ) is
    (?First, ?Last in Integer)
        timing(P,?First,?Last,C) where ?Last <= Time;

pattern after( pattern P; Time : Integer; C : Clock ) is
    (?First, ?Last in Integer)
        timing(P,?First,?Last,C) where ?First >= Time;

pattern within( pattern P; Time : Integer; C : Clock ) is
    (?First, ?Last in Integer)
        timing(P,?First,?Last,C) where (?Last-?First <= Time);

pattern within( pattern P; Time1, Time2 : Integer; C : Clock ) is
    (?First, ?Last in Integer)
        timing(P,?First,?Last,C) where Time1 <= ?Last-?First <= Time2);

pattern "<"( pattern P1, P2; C : Clock ) is
    (?First1, ?First2, ?Last1, ?Last2 in Integer)
    ( timing(P1,?First1,?Last1,C) ~
      timing(P2,?First2,?Last2,C) ) where ?Last1 < ?First2;
```

18.3 Semantics

TBD

18.4 Special Syntax

TBD

Chapter 19

Event

Events are record-like structures that are generated by modules as part of the action-based communication primitives of RAPIDE. For more details, refer to the RAPIDE 1.0 Executable Language Reference Manual.

19.1 Event type expressions

19.1.1 Syntax

event_type_expression	::=	**event** '(' *param_list* ')'
param_list	::=	*types_object_name_declaration* { ';' *types_object_name_declaration* }

19.1.2 Semantics

Every event type is defined in terms of its *parameter record type* and a set of operations available on all events.

For every event type `E` of the form **event**(`F1:T1, ... Fn:Tn`), the associated parameter record type `E_Rec` is defined as:

```
type E_Rec is record
    F1 : T1;
    F2 : T2;
    ...
    Fn : Tn;
end record;
```

Also, all events have a default set of operations defined on them. These allow an event to be compared with another and allow the originating module and action to be determined. These operations are described in the following interface.

type Event_Operations **is interface**
    Name : Caseless_String;
    Owner : **interface end**;
    Performer : **interface end**;
    =, /=, <, >, <=, >=, || : **function** (rhs : **event**()) **return** Boolean;
end interface;

The event type E of the form **event**(F1:T1, ... Fn:Tn) is now defined as:

type E **is interface**
    include E_Rec;
    include Event_Operations;
end interface;

Event types can be defined from other event types in the same way that record types are (see Section 9).

19.1.3 Examples

Example: *Simple event declarations*
    event()
    event(i : Integer; s : String)
□

19.1.4 Subtyping

Subtyping of the event types follow directly from the subtyping rules of interface types in general:

- Every event type is a subtype of its parameter record type.
- For two event types E1 and E2, E1 is a subtype of E2 iff E1's parameter record type is a subtype of E2's parameter record type.

19.2 Event object expressions

The only way to generate events is by calling actions. For details, refer to the RAPIDE 1.0 Executable Language Reference Manual. ▬[29] **Similar remark about the relationship between patterns and events.**

Chapter 20

Input and Output

20.1 Introduction

TBD

[30] **I just yanked this from the old IO proposal doc. Needs work.**

20.2 Syntax

```
type File_Mode is enum In_File, Out_File, InOut_File end enum;

type IO_Error is enum No_Error, File_Not_Open, File_Already_Open,
                      Unable_To_Open, Mode_Error, End_Of_Input,
                      Unknown_Error
               end enum;


type IO_Type is interface
        type File_Type;
        type Position is Positive;

        -- Routines for manipulating standard I/O streams.
        function Set_Input( F : File_Type );
        -- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
        -- Mode(in F) == Out_File -> Error_Of(out F) = Mode_Error;

        function Set_Output( F : File_Type );
        -- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open;
        -- Mode(in F) == In_File -> Error_Of(out F) = Mode_Error;

        function Set_Error( F : File_Type );
        -- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open;
        -- Mode(in F) == In_File -> Error_Of(out F) = Mode_Error;

        function Current_Input() return File_Type;
```

```
function Current_Output() return File_Type;
function Current_Error() return File_Type;
function Standard_Input() return File_Type;
function Standard_Output() return File_Type;
function Standard_Error() return File_Type;

-- Routines for managing files.
function Is_Open( F : File_Type ) return Boolean;
function Error_Of( F : File_Type ) return IO_Error;

function Name_Of( F : File_Type ) return String;
-- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
-- Error_Of(out F) = Error_Of(in F);

function Mode_Of( F : File_Type ) return File_Mode;
-- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
-- Error_Of(out F) = Error_Of(in F);

function Length( F : File_Type ) return Natural;
-- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
-- Error_Of(out F) = Error_Of(in F);

function Loc( F : File_Type ) return Position;
-- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
-- Error_Of(out F) = Error_Of(in F);

function Separator( F : File_Type ) return Character;
-- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
-- Error_Of(out F) = Error_Of(in F);

function Open( Name : String; Mode : File_Mode; F : File_Type;
               Field_Separator : Character is '\n' );
-- Is_Open(in F) -> Error_Of(out F) = File_Already_Open,
-- Out-files always create new files; use In_Out to append
-- Mode = Out_File -> Length(out F) = 0;

function Close( F : File_Type );
-- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open;

function Reset( F : File_Type; Mode : File_Mode is In_File );
-- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
-- out( Mode(F) = Mode ),
-- out( if Mode = Out_File then Loc(F) = Length(F) + 1
--      else Loc(F) = 1 end if );

-- Routines for moving around files.
function Move_To( F : File_Type; P : Position );
-- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
-- P > Length(in F) + 1 -> Error_Of(out F) = End_Of_Input,
-- P <= Length(in F) + 1 -> Loc(out F) = P;

-- Put and Get routines
function Put( S : String;
```

```
                    F : File_Type is Current_Output() );
    -- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
    -- Mode_Of(F) = In_File -> Error_Of(out F) = Mode_Error,
    -- in( Loc(F) = Length(F) + 1 ),
    -- Loc(out F) = Length(out F) + 1 = Loc(in F) + 1;

    function Get( F : File_Type is Current_Input() ) return String;
    -- not Is_Open(in F) -> Error_Of(out F) = File_Not_Open,
    -- Mode_Of(F) = Out_File -> Error_Of(out F) = Mode_Error,
    -- Loc(F) = Length(F) + 1 -> Error_Of(out F) = End_Of_Input,
    -- Loc(out F) = Loc(in F) + 1;

constraint
  -- For out_files, Loc() is always Length() + 1.
  axiom for all F : File_Type; =>
           if Is_Open(F) and Mode_Of(F) = Out_File
           then Loc(F) = Length(F) + 1
           else True end if;

  axiom Mode_Of( Standard_Input() ) = In_File,
        Mode_Of( Standard_Output() ) = Out_File,
        Mode_Of( Standard_Error() ) = Out_File;
end interface;

IO : IO_Type;
```

20.3 Semantics

TBD

20.4 Transformation Functions

20.4.1 Syntax

```
function Image (type T; V : T) return String;
function Is_Image (type T; S : String) return Boolean;
function Value (type T; S : String) return T;
```

20.4.2 Semantics

Three functions are provided for converting between basic types and String. The first is Image, which converts a type to its String equivalent. Then the routine Is_Image is defined to test if a String is the String representation of a particular type. Finally Value converts a String to an object of the predefined type. For safety, it is advised to call Is_Image on the String before calling the appropriate Value function.

20.5 Examples

Example: *Using IO*

```
type Mytype is interface
requires
    My_IO : IO_Type;
end;

module M() return Mytype is
    F : My_IO.File_Type;
    S : var String;
    action A( I : Integer );
initial
    -- Writing to files: a file consisting of an integer and a string.
    My_IO.Open( "my_file", Out_file, F );
    assert My_IO.Error_Of(F) = No_Error;
    My_IO.Put( Image(Integer, 10), F );
    My_IO.Put( "a record.", F );

    -- Reading from files.
    My_IO.Reset( F, In_File );
    My_IO.Move_To( F, 2 );          -- Skip the integer record
    S := My_IO.Get( F );
    My_IO.Put( $S );                -- Put to standard out: "another record."
    My_IO.Move_To( F, 1 );          -- Now go get the integer
    A( Value(Integer, My_IO.Get(F)) );
    My_IO.Close(F);
end;

architecture Top_Level() return Root is
    Local_IO : IO_Type is IO;
    M_Object : Mytype is M();
connect
    M_Object.My_IO to Local_IO;
end;
```

Commentary:

In this example, we define the architecture called Top_Level, which has a component called Local_IO, defined to be an alias of the global IO object. The required object My_IO has been connected to Local_IO. This means that when M_Object uses its component My_IO, it is actually using the global IO object; thus communication integrity is assured.

[31] **By the current connection rules, the connection in Top_Level is illegal. Connections may only be made between interface constituents of an architecture and**

interface constituents of a component. But components themselves may not be connected as a whole to anything. We will need to study this issue more.

□

Chapter 21

Link

21.1 Introduction

TBD.

21.2 Syntax

function Link(Src, Dest : Root) **return** Root;
function Link(Src : Root) **return** Root;

function Unlink(Src, Dest : Root);
function Unlink(Src : Root);

21.3 Semantics

When X is linked to Y, mode-out events generated by X are available to Y. The call Link(X,Y) links X to Y; the call Link(X) links X to the enclosing module (i.e. the module in which the call to Link was made). Link(X) may only be called where there is an enclosing module. Linking X to Y has no affect if X is already linked to Y.

A call to Link returns the Src parameter of the call: Link(X) returns X.

After a call Unlink(X,Y), X is not linked to Y. After a call Unlink(X), X is not linked to the enclosing module that called Unlink.

Every module is initially linked to its parent (creator), if it has a parent; otherwise it is not linked to any module. A module may be linked to zero or more modules.

[32] **In the above function signatures, the intent is that Root is the supertype of all module types. Rapide doesn't have such a type because of the subtyping rules w.r.t. required constituents. The intent is to allow linking of any module to any other module, so Link and Unlink may be primitives whose signatures cannot be defined in the language.**

[33] **Someone (I think Frank) suggested that Link have a "stack" semantics: to unlink two objects, you have to call Unlink as many times as Link has been called for them.**

[34] **Sigurd wants a polymorphic link.**

Chapter 22

Identity

22.1 Introduction

Identity functions yield whether two names denote the same module.

[35] **This is another construct that fudges the idea of a grand interface supertype.**

22.2 Syntax

"==", "/==" : **function** (Right : T) **return** Boolean;

22.3 Semantics

Each call to a module generator, module literal evaluation, and call to an allocator function yields a new module. An identity function "==" (and its complement "/==") is predefined for all interface types. The identity function of module m yields whether m and the function parameter were created by the same generator call, literal evaluation, or allocator call. Module definitions may not provide a function (i.e., object declaration) for "==" or "/=="; tool-supplied functions are always used.

For any module generator G and allocator function New:

G()	/==	G()	-- *module generator calls*
Root'(**module end**)	/==	Root'(**module end**)	-- *literals*
New()	/==	New()	-- *allocator function calls*

Appendix A

Predefined Types Syntax : Quick Reference

Boolean

true	*boolean literal*
false	*boolean literal*
not B	*negation*
B1 and B2	*conjunction*
B1 or B2	*disjunction*
B1 xor B2	*exclusive disjunction*
B1 = B2	*equality*
B1 /= B2	*inequality*

Integer, Float, Natural, Positive

max_integer	*maximum integer literal*
min_integer	*minimum integer literal*
+ N	*unary plus*
− N	*unary minus*
N1 + N2	*addition*
N1 − N2	*subtraction*
N1 * N2	*multiplication*
N1 ** N2	*exponentiation*
N1 / N2	*division*
N1 % N2	*remainder*
abs N	*absolute value*
pred N	*predecessor (not for Float)*
succ N	*successor (not for Float)*
floor N	*integer floor (for Float only)*
N1 = N2	*equality*
N1 /= N2	*inequality*

N1 <= N2	*less than or equal*
N1 < N2	*less than*
N1 >= N2	*greater than or equal*
N1 > N2	*greater than*

Character

code C	*integer code of character C*
code_to_char N	*character with integer code N*
\n	*character literal for newline*
\\	*character literal for backslash*
\'	*character literal for single quote*
\N	*character literal for character with integer code N*
C1 = C2	*equality*
C1 /= C2	*inequality*
C1 <= C2	*less than or equal*
C1 < C2	*less than*
C1 >= C2	*greater than or equal*
C1 > C2	*greater than*

String

""	*null string*
is_null S	*empty*
length S	*length of string*
S[N]	*N^{th} character of string*
S[N..M]	*substring*
S1 & S2	*concatenation*
S & C	*add character to end of string*
C & S	*add character to beginning of string*
String_to_Integer S	*convert string to integer*
S1 = S2	*equality*
S1 /= S2	*inequality*
S1 <= S2	*less than or equal*
S1 < S2	*less than*
S1 >= S2	*greater than or equal*
S1 > S2	*greater than*

Record

(F1 **is** V1, ... Fn **is** Vn)	*record with value Vi for field Fi*
R.F	*record selection*

Union

(l,X)	*union object with tag l and value X*
U.l	*value selection based on tag*
case([x:T is] U, l1 **is** E1, l2 **is** E2, ... , **default is** E)	
	discrimination
tagof U	*enumeration of tags in the union object*
U1 = U2	*equality (only if all member types have equality)*

Array

A[N]	*indexing*
iterator A	*return the iterator for A*

For discrete arrays only:

(X..Y, E1, E2,..., **default is** E)	*array literal*
A[N..M]	*slicing*
A1 = A2	*equality*
A1 /= A2	*inequality*
first A	*first index of the array*
last A	*last index of the array*
length A	*length of the array*

Set

{ }	*empty set*
S is_member E	*membership*
S + E	*insert an element*
S − E	*delete an element*
S1 & S2	*set union*
S1 − S2	*set difference*
S1 < S2	*subset*
cardinality S	*size of set*
S1 = S2	*equality*
S1 /= S2	*inequality*

Ref

ref_to(T,X)	*create reference to X*
nil(T)	*create nil reference*

R := X	*assignment*
*R	*dereference*
is_nil R	*check whether R refers to an object*
&R	*alternate syntax for "var" declaration*

Clock

make_clock()	*create a clock*
make_synchronous_clock()	*create a synchronous clock*
make_regular_clock(...)	*create a regular clock*
make_slaved_clock(...)	*create a slaved clock*
now C	*current clock counter*
C start E	*counter for start of event E*
C finish E	*counter for finish of event E*
C length E	*counter interval for event E*

For synchronous, regular and slaved clocks only:

C GST T	*convert counter to common metric*
C.distance(T1,T2)	*convert counter interval to common metric*
C ticks G	*closest common metric*

For regular and slaved clocks only:

period C	*time period of synchronization*
precision C	*measure of accuracy*

For slaved clocks only:

master C	*the master clock*

Event

from_module E	*module that generated the event*
action_name E	*action that generated the event*
E1 = E2	*causally identical*
E1 /= E2	*causally distinct*
E1 < E2	*causally precedes*
E1 <= E2	*causally precedes or identical*
E1 > E2	*causally follows*
E1 >= E2	*causally follows or identical*
E1 || E2	*causally independent*

I/O

file_open S	*open the file corresponding to string*
std_in	*the standard input file*
std_out	*the standard output file*
std_err	*the standard error file*
set_input F	*set std_in to F*
set_output F	*set std_out to F*
set_error F	*set std_err to F*
go_to_start F	*beginning of file*
go_to_end F	*end of file*
new_line F	*add a new line*
next_line F	*go to next line*
at_end F	*check whether end of file*
read_bool F	*read a boolean*
read_int F	*read an integer*
read_float F	*read a float*
read_char F	*read a char*
read_string F	*read a string*
F put_bool B	*write a boolean*
F put_int N	*write an integer*
F put_float F	*write a float*
F put_char C	*write a character*
F put_string S	*write a string*

Glossary

natural A type equivalent to Integer, with a constraint requiring its elements to be non-negative. See page 14.

Index

array, 45–48
array, discrete, 45

boolean, 3–5

character, 17–18
character, backslash, 18
character, newline, 18
character, quote, 18
clock, 55–58
clock, regular, 56
clock, slaved, 57
clock, synchronous, 56

enumeration, 37–38
equality, 7–8
event, 61–62

false, 3
float, 14

I/O, 63
identity, 71
identity function, *71*
index type, 45
inequality, 7–8
input, 63–67
integer, 13–14
IO, 67
iteration, 39–40

link, 69

max_integer, 13
min_integer, 13

natural, 14
numeric, 13–15

order, 9–10
order, partial, 10
order, total, 10
output, 63–67

positive, 15

range, 41–43
record, 23–26
ref, 51–53

set, 49–50
string, 19–21

tool supplied
    functions, 71
Triv, 11–12
true, 3