DRAFT
Rapide 1.0 Types Language Reference Manual

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

July 17, 1997

Preface

This section has not been written yet.

Contents

Chapter 1

Introduction

This reference manual defines the RAPIDE 1.0 Types Language. The reference manual is divided into the following parts:

1. Chapter 2 defines the syntactic and lexical issues of the RAPIDE 1.0 Types Language.

2. The next three chapters, Chapter 3 through Chapter 5 defines the types of the RAPIDE 1.0 Types Language. Chapter 3 defines the general concepts of the RAPIDE 1.0 Types Language types. Chapter 4 defines interface types while chapter 5 defines function types.

3. Following the description of the types in RAPIDE 1.0 Types Language, Chapter 6 defines the type constructors of RAPIDE 1.0 Types Language and the ways in which they may be instantiated to obtain types.

4. Most of the visibility rules are not within the scope of the RAPIDE 1.0 Types Language. However, Chapter 7 defines all such rules that are part of the RAPIDE 1.0 Types Language.

5. Finally, there are appendices that presents a glossary of keywords used in the reference manual, give an overview of the RAPIDE 1.0 Types Language syntax, and define the assumptions that the RAPIDE 1.0 Types Language makes of the RAPIDE 1.0 language framework.

The official reference manual describing RAPIDE 1.0 types starts from Chapter 2 and ends at Chapter 7. The examples and notes in these chapters are not part of the official reference manual. The preface, this introduction, and the appendices are also not part of the official reference manual.

Chapter 2

Syntax and Lexical Elements

2.1 Syntax

The syntax of the RAPIDE 1.0 Types Language along with the other sublanguages within the same language framework (i.e., the pattern, executable, architecture, and constraint languages) is a context free grammar. Each of the above mentioned languages define a portion of this grammar.

The following grammar non-terminals are defined in the RAPIDE 1.0 Types Language and may be used by the other sublanguages within the same language framework:

type_declaration
type_expression
type_constructor_declaration
type_constructor_expression

The following grammar non-terminals are used by the RAPIDE 1.0 Types Language and must be defined in the executable language within the same language framework:

executable_language_type_expression
executable_language_name_declaration
executable_language_type_constructor_expression
executable_language_type_constructor_declaration
object_expression

The following grammar non-terminals are used by the RAPIDE 1.0 Types Language and must be defined in the constraint language within the same language framework:

interface_constraint
function_constraint
type_constructor_constraint

The following grammar non-terminal is used by the RAPIDE 1.0 Types Language and must be defined in the architecture language within the same language framework:

architecture_language_interface_declarative_region

The above non-terminals (defined in the executable, constraint and architecture languages) appear *italicized* throughout the RAPIDE 1.0 Types Language Reference Manual.

The grammar described in the RAPIDE 1.0 Types Language Reference Manual is unambiguous in itself. If ambiguities are introduced with respect to productions introduced in one of the other languages in the same language framework, then these other languages must also describe how these ambiguities are resolved.

2.2 Lexical Elements

The lexical elements of the RAPIDE 1.0 Types Language are divided into three categories — *identifiers*, *delimiters*, and *reserved words*. The RAPIDE 1.0 Types Language places no requirements on the concrete representation of these lexical elements. The executable language within the same language framework must define the concrete representation of these lexical elements, as well as any other general notions of lexical structure such as *white space* and *comments*.

The mnemonics used to represent the lexical elements in the RAPIDE 1.0 Types Language have been chosen solely based on convenience, and in no way do they indicate the concrete representation of the corresponding lexical elements.

Identifiers

Identifiers are represented in the RAPIDE 1.0 Types Language as the lexical token

identifier

An identifier is a lexical token that can usually have multiple concrete representations, and there is a comparison operation defined on these concrete representations. So two identifier's may either be considered "same" or "distinct" based on applying this comparison operation on their concrete representations. The executable language within the same language framework must define this comparison operation.

Delimiters

The delimiters of the RAPIDE 1.0 Types Language are listed below.

‘(’ ‘)’ ‘,’ ‘:’ ‘;’ ‘<:’

Reserved Words

The reserved words of the RAPIDE 1.0 Types Language are listed below:

action	**end**	**except**	**function**	**include**	**interface**	**is**
only	**private**	**provides**	**replace**	**requires**	**return**	**to**
type						

Chapter 3

Types and Type Declarations

This chapter and the next two (Chapters 4 and 5) describes the types of the RAPIDE 1.0 Types Language. Types describe sets of "values". In RAPIDE 1.0 these "values" are referred to as *objects*.

Terminology. If an object is a member of a particular type T, the object is referred to as being *of type* T.

The RAPIDE 1.0 Types Language defines two kinds of types — the *interface type* and the *function type*.

Interface types describe encapsulations of system components. They describe the interface of the system component with other system components. Objects of interface types are referred to as *modules*. Chapter 4 describes interface types.

Function types describe functions — objects of function types are *functions*. The function type describes the parameter and result types of its member functions. Chapter 5 describes function types.

The executable language within the same language framework must define syntactic constructs that denote objects (functions and modules). These constructs are referred to as *object expressions*. Their syntax must be defined (in the executable language) as expansions of the non terminal *object_expression*.

3.1 Type Expressions

The syntactic construct used to *denote* types is the *type expression*.

3.1.1 Syntax

type_expression ::=
  interface_type_expression
| *function_type_expression*
| *type_constructor_application*
| '(' *type_expression* ')'
| *identifier*
| *FRAMEWORK_type_expression*

3.1.2 Semantics

The semantics for interface type expressions and function type expressions are described in Chapters 4 and 5 respectively. The semantics for type constructor applications are described in Chapter 6.

Type expressions may be parenthesized for purposes of clarity — a parenthesized type expression denotes the same type as denoted by the type expression within the parentheses.

Type declarations (see Section 3.2) and type name declarations (see Chapters 4 and 5) cause identifiers to denote types. In certain contexts, as described in Chapter 7, these identifiers may be used as type expressions. The type denoted by the type expression is then the same as the type denoted by the identifier according to these declarations.

Finally, the executable language within the same language framework must provide syntactic constructs that denote types — these constructs may also be used as type expressions. One of these syntactic constructs must be the selection operation (see Section 4.5.1).

Terminology. The phrase "*(the) type denoted by* E", where E is any type expression, may be replaced by the phrase "*(the) type* E".

Example 1: *A function type expression*

> **function**(X, Y : Integer) **return** Integer

> This function type expression denotes a function type with two arguments of type Integer and result type Integer. Functions of this type in the RAPIDE 1.0 Executable Language may be called with two Integer objects as parameters which causes the body of the function to be executed to return an integer object as the result.

□

Example 2: *An interface type expression*

```
interface
   provides
      Current_Value : Real;
      Change_Value : function(New_Value:Real);
end interface
```

This interface type expression denotes an interface type which describes modules that provide the two constituents listed in the expression. Modules of this type in the RAPIDE 1.0 Executable Language will contain an object Current_Value of type Real, and will define a function object Change_Value that accepts an argument of type Real.

□

3.2 Type Declarations

The RAPIDE 1.0 Types Language defines two kinds of declarations — *type declarations* and type constructor declarations. A type declaration introduces an identifier and gives it a type as its denotation. Type constructor declarations are described in Chapter 6.

3.2.1 Syntax

type_declaration ::=
    type *identifier* **is** *type_expression* ';'

3.2.2 Semantics

The type declaration introduces the identifier after the keyword **type** and gives as its denotation the type denoted by the type expression after the keyword **is**.

Example 3: *A Type Declaration*

```
type Point is interface
provides
    X_Val : Integer;
    Y_Val : Integer;
    Distance : function(P : Point) return Integer;
end interface;
```

This is an example of a type declaration that introduces the identifier Point and gives as its denotation the interface type denoted by the interface type expression to the right of **is**.

□

3.3 Subtyping and Substitutivity

The RAPIDE 1.0 Types Language defines the *subtype* relationship between two types which is elaborated on in Chapters 4 and 5.

Suppose that T1 is a subtype of T2 for some types T1 and T2 and suppose that O1 and O2 are two object expressions of types T1 and T2 respectively.

The other sublanguages within the same language framework must guarantee that if a particular use of O2 in a RAPIDE 1.0 program does not violate any of the static semantic rules of these respective languages, then replacing O2 by O1 will not violate any of these static semantic rules either. This rule is called the *substitutivity rule*.

Also, O1 must also be an object of type T2. This rule is called the *membership rule*.

There is one exception to the applicability of the substitutivity rule: If T2 is an interface type with a type constituent (see Chapter 4), then the substitutivity rule need not hold, however the membership rule still has to hold.

3.4 Equality between Types

Two types are equal to each other (*i.e.*, the same type) if each is a subtype of the other.

Chapter 4

Interface Types

An interface type declares a set of *constituents*. Each constituent consists of two parts — an identifier and a *denotation specification*.

An interface type has four kinds of declarative region — *(1)* a *provides interface*, *(2)* a *requires interface*, *(3)* an *action interface*, and *(4)* the *private interface*.

The constituents of a provides interface may be used by any component in the system subject to the rules stated in Section 4.5.1. The constituents of a requires interface are object names that modules of this interface type assume they have visibility to (in addition to any other visible declarations). The constituents of an action interface may be used by any component in the system subject to the rules stated the RAPIDE 1.0 Architectures Reference Manual. The constituents of the private interface of an interface type may be used only within modules of this interface type (*the private interface rule*). The notion of "within modules" in the previous sentence must be defined in the executable language used within the same language framework.

For each provides and private constituent in a module's interface type, the module must define a denotation for the identifier of the constituent. For each requires constituent of a module's interface type, the architecture language within the same language framework must provide a means of defining a denotation for the identifier of the declaration.

These denotations must be consistent (defined in Section 4.2) with their denotation specification in the interface type.

Commentary

> Interface types have similar counterparts in other languages. They are similar to class type declarations in C++ and package specifications in Ada. The identifiers of the constituents of an interface type need not be distinct — *i.e.*, the constituent names may be "overloaded".

4.1 Interface Type Expressions

The syntactic construct used to denote interface types is the *interface type expression.*

4.1.1 Syntax

```
interface_type_expression ::=
        { interface_derivation_declaration }
        interface
               {interface_declarative_item}
               {interface_declarative_region}
               [ private {interface_declarative_region} ]
        end [ interface ]


interface_declarative_region ::=
        interface_provides_part
|       interface_requires_part
|       interface_action_part
|       architecture_language_interface_declarative_region


provides_part ::=
        provides { interface_declarative_item }


requires_part ::=
        requires { interface_declarative_item }


action_part ::=
        action { action_declaration }


interface_declarative_item ::=
|       name_declaration
|       interface_derivation_declaration
|       constraint_language_constraint


action_declaration ::=
        in { action_name_declaration } ';'
|       out { action_name_declaration } ';'
```

4.1.2 Semantics

An interface type expression contains a list of interface derivation declarations followed (without any delimiter) by an interface indicated by the keyword, **interface**. An interface contains a list of interface declarative regions, each of which introduces a set of provides name decalarations, requires name decalarations, action name declarations, or private name declarations.

The provides interface is defined by the set of provides declarative regions, the requires interface is defined by the set of requires declarative regions, the action interface is defined by the set of action declarative regions, and the private interface is defined by a single private declarative region. These interface declarative regions begin with the keyword **provides**, **requires**,

action and **private**, respectively, and are followed by a list of interface declarative items. For convenience, if the first declarative region has no keyword, the **provides** keyword is assumed.

An interface declaration is either *(1)* a *name declaration* which defines constituents of the interface type, or *(2)* an *interface derivation declaration* which is an inheritance mechanism that expands into a sequence of name declarations. Name declarations are described in Section 4.2 and interface derivation declarations are described in Section 4.3.

Other interface declarative regions may be defined by the RAPIDE 1.0 Architecture Language-within the same language framework. The *constraint* construct must be defined in the constraint language within the same language framework. Its syntax must distinguish it from the rest of the interface declaration list.

Example 1: *A Point Interface*

```
type Point is interface
provides
    X_Val : Integer;
    Y_Val : Integer;
    Distance : function(P : Point) return Integer;
end interface;
```

The interface type Point contains three name declarations. Modules of type Point define actual objects with these names. Note that this interface has no requires or private interface.

□

Example 2: *A Set Interface*

```
type IntegerSetType is interface
provides
    Insert : function(N : Integer);
    Delete : function(N : Integer);
    Union : function(S : IntegerSetType);
    Is_Member : function(N : Integer) return Boolean;
private
    List_of_Members : function() return Integer_List;
end interface;
```

This example illustrates the use of *private interfaces*. Modules of type IntegerSetType export the functions listed in the provides interface part. These modules also export the constituents of the private interface to other modules of the same type — IntegerSetType. This facilitates the implementation of binary operations such as Union above. The Union operation in a module of type IntegerSetType works on sets implemented by other modules by using the List_of_Members function provided by that module.

□

Example 3: *The interface of a timer*

```
type Timer is interface
provides
    Start : function();
    Stop : function();
    Reset : function();
    Time_Elapsed : function() return Time;
requires
    Clk : Clock;
end interface;
```

This example illustrates the use of *requires* declarations. The interface type above describes timers. A timer can provide various timing utilities as long as it has access to some clock. This may be specified by using the requires declaration to require that a clock be made available to every instance of the type Timer.

□

Example 4: *The interface of an electrical device*

```
type Gadget is interface
provides
    On : function();
    Off : function();
    Increase : function(Amount : Real);
    Decrease : function(Amount : Real);
    Reset : function();
requires
    VS : Voltage_Supply;
    Ground : Earth_Connection;
private
    Current_Setting : Real;
    Max_Setting : Real;
    Change_Setting : function(New_Value : Real);
end interface;
```

This example illustrates the different parts of an interface – (1) The public interface lists functions that can be invoked by any user of such devices. (2) Devices of this type might have certain requirements in order to function properly. So, for instance, they might need a voltage supply and a ground connection. Such requirements can be specified with requires declarations. (3) Devices of this type might be connected to similar devices to increase their functionality. For this, the device might provide certains constituents that can be accessed only by a device of the same type. These would be part of the private interface.

□

Commentary

Interface expressions are RAPIDE 1.0's way of defining data abstractions and sets of objects, hence they provide the capability for both object-oriented programming as well as programming with abstract data types.

The provides interface specifies a list of constituents for which implementations must be provided by all modules (objects) of this type. The requires interface specifies a list of constituents that all modules of this type can assume are supplied by other modules. The mechanism for connecting requires constituents to other modules is defined in the RAPIDE 1.0 Architecture Language. The action interface specifies an ability of modules of this type to asynchronously receive (**in**) and generate (**out**) events of particular types, as described in the RAPIDE 1.0 Architecture Language.

The provides interface constituents and **in** actions can be referred to by all users of modules of the interface type. However, it is sometimes necessary for similar modules to have access to more facilities than those provided to all users. The private interface is for this purpose. Code within any module of an interface type may refer to private interface constituents of any other module of the same type. This is usually useful when different modules of the same type have different representations, and they wish to interoperate. For example, a module (object) that implements sparse matrices may wish to multiply itself with one that implements dense matrices. Sparse matrices and dense matrices may be represented using different data structures and some extra conversion routines may be required to interoperate between these data structures.

4.2 Name Declarations and Denotation Specifications

Every constituent of an interface type consists of an identifier and a denotation specification. The syntactic construct that denotes constituents of interface types is the name declaration.

4.2.1 Syntax

name_declaration ::=
    type_name_declaration
  | *type_constructor_name_declaration*
  | *object_name_declaration*
  | *action_name_declaration*
  | *FRAMEWORK_name_declaration*

type_name_declaration ::=
    type *identifier* ';'
  | **type** *identifier* '<:' *type_expression* ';'
  | **type** *identifier* **is** *type_expression* ';'

type_constructor_name_declaration ::=
    type *identifier* '('[*formal_parameter_declaration_list*] ')' ';'
  | **type** *identifier* '('[*formal_parameter_declaration_list*] ')' '<:' *type_expression* ';'
  | **type** *identifier* '('[*formal_parameter_declaration_list*] ')' **is** *type_expression* ';'

object_name_declaration ::=
    identifier_list ':' *type_expression* ';'

action_name_declaration ::=
    identifier_list '('[*formal_parameter_declaration_list*] ')' ';'

identifier_list ::=
    identifier { ',' *identifier*}

4.2.2 Semantics

There are four kinds of name declarations: *type name declarations*, *type constructor name declarations*, *object name declarations*, and *action name declarations*. The executable and architecture languages within the same language framework may define alternate syntax for these name declarations by defining expansions for the non terminals *executable_language_name_declaration* and *architecture_language_name_declaration* respectively.

Type name declarations and type constructor name declarations may not appear in a requires declarations part of an interface type expression. Action name declarations may appear only in an action declarations part.

A name declaration defines constituents of an interface type and is characterized by the denotation specification it defines. Type name declarations define *type denotation specifications*, type constructor name declarations define *type constructor denotation specifications*, object name declarations define *object denotation specifications*, and action name declarations define *event type denotation specifications*.

The different kinds of denotation specifications and the consistency rules in each case are listed below. The consistency rules define the set of valid denotations for each denotation specification.

Type denotation specifications require their corresponding denotations to be types. Constituents containing type denotation specifications are introduced using type name declarations. Each type name declaration introduces one constituent whose identifier appears after the keyword **type**. There are three different forms of type name declarations each of which define different type denotation specifications:

1. The simplest form of type name declaration consists of the keyword **type** followed by an identifier and a semicolon. For this form of type name declaration, the corresponding denotation may be any type.

2. In the second form, the delimiter '<:' followed by a type expression appears after the identifier of the type name declaration. For this form of type name declaration, the corresponding denotation must be a type that is a subtype of the type denoted by the type expression after the delimiter '<:'.

3. In the third form, the keyword **is** followed by a type expression appears after the identifier of the type name declaration. For this form of type name declaration, the corresponding denotation must be the same type as that denoted by the type expression after the keyword **is**.

Type constructor denotation specifications require their corresponding denotations to be type constructors. Constituents containing type constructor denotation specifications are introduced using type constructor name declarations. Each type constructor name declaration introduces one constituent whose identifier appears after the keyword **type**. There are three different forms of type constructor name declarations each of which define different type constructor denotation specifications:

1. The simplest form of type constructor name declaration consists of the keyword **type** followed by an identifier, a formal parameter list, and a semicolon. For this form of type constructor name declaration, the corresponding denotation may be any conforming type constructor. Conformance is described in Chapter 6.

2. In the second form, the delimiter '<:' followed by a type expression appears after the formal parameter list of the type constructor name declaration. For this form of type constructor name declaration, the corresponding denotation must be a conforming type constructor, each application of which is a subtype of the type denoted by the type expression (with any identifiers from the formal parameter list being replaced by the respective parameters used in the application) after the delimiter '<:'.

3. In the third form, the keyword **is** followed by a type expression appears after the formal parameter list of the type constructor name declaration. This type constructor name declaration is then identical to a type constructor declaration. For this form of type constructor name declaration, the corresponding denotation must be equal to the type constructor name declaration (interpreted as a type constructor declaration). The notion of "same type constructors" is defined in Chapter 6.

Object denotation specifications require their corresponding denotations to be objects of a particular type. Provides, requires and private constituents containing object denotation specifications are introduced using object name declarations. An object name declaration contains a list of identifiers and introduces a constituent corresponding to each of these identifiers. All constituents are given the same object denotation specification, which is the type expression to the right of the delimiter ':'. The corresponding denotation can be any object of the type denoted by the type expression.

Action name declarations define event type denotation specifications as defined in the RAPIDE 1.0 Architectures Reference Manual. An action name constituent denotes that modules of the interface type can receive events (**in**) or generate events (**out**) of the denoted event type. The semantics of action name interface constituents is defined in terms of the subtyping rules in Section 4.4 and the architecture conformance rules in RAPIDE 1.0 Architectures Reference Manual.

The identifiers of constituents with type denotation specifications and the identifiers of constituents with type constructor denotation specifications must be distinct from each other and from the identifiers of all other constituents of the interface type. In other words, the overloading of type names is disallowed. However, identifiers of constituents with object denotation specifications are not thus constrained. Hence, object names can be overloaded. The executable language is required to resolve any conflicts arising from such overloading.

Example 5: *Type Name Declarations in Interface Types*

```
type Stack_Package is interface
provides
    type Stack;
    ...
end interface;

type Work_Force is interface
    type Employee <: interface
                          Name: String;
                          Salary : Float;
                      end interface;
```

```
    ...
end interface;

type Electrical_Components is interface
provides
    type Switch is interface
                        Current_State : Boolean;
                        Toggle : function() return Switch;
                    end interface;
    ...
end interface;
```

The first interface type Stack_Package contains a type name declaration which introduces the identifier Stack and gives it a type as its denotation specification. Any type is consistent with this denotation specification.

The second interface type Work_Force contains a type name declaration which introduces the identifier Employee and gives it a type as its denotation specification. It also specifies that the corresponding denotation in modules must be a subtype of the bound provided. This bound specifies that the denotation of Employee is guaranteed to have a Name and an Salary object. Note that since there is no declarative region keyword in either of these interface type expressions, the keyword **provides** is assumed.

The last type introduces the identifier Switch, gives it a type as its denotation specification, and specifies exactly the denotation of the type in modules of the type Electrical_Components.

□

Example 6: *Type Constructor Name Declarations*

```
type Collections is interface
    type Deque(type Element; Size : Integer);
    type List(type T) <: interface
                            Item : T;
                            Next : List(T);
                        end interface;
    ...
end interface;
```

The interface type Collections contains two type constructor name declarations.

The first declaration introduces the identifier Deque and gives it a type constructor as its denotation specification. Corresponding modules must provide type constructors consistent with this denotation specification.

The second declaration introduces the identifier List and gives it a type constructor as its denotation specification. It also places a restriction on applications of the corresponding denotations within modules. These have to be subtypes of the interface type expression after the symbol "<:" (after replacing all occurrences of T with the parameter used in the application).

□

Example 7: *Object Name Declarations*

```
type Employee is interface
provides
    Name, Occupation : String;
    Salary : Float;
end interface;
```

The definition of type Employee declares three constituents — Name, Occupation, and Salary — using two object name declarations. All modules of this type must supply objects with the same identifiers and the same types.

□

Example 8: *An example with overloading*

```
type Collection is interface
provides
    ...
    Max : function() return ElementType;
    ...
    Max : function() return Integer;
    ...
end interface;
```

This is an example of an interface type with an overloaded object name Max. The first Max is used to denote the type of the function that returns the biggest element in the collection, while the second Max is the type of the function that returns the maximum capacity of the collection – in other words, the maximum number of elements that it can store. The executable language is required to guarantee that all uses of Max will be disambiguated.

□

Example 9: *An example with actions*

```
type InputOutput is interface
action
    in  MouseEvent (B : Buttons; X, Y : Integer);
    out MoveCursor (X, Y : Integer);
    ...
end interface;
```

This is an example of an interface type containing asynchronous communication actions. Action name declarations denote that the modules of the type can receive (**in**) or generate (**out**) events of particular event types which are instances of the predefined event type constructor in the RAPIDE 1.0 Predefined Types and Objects Reference Manual. The **in** action MouseEvent denotes that modules of type InputOutput must be able to receive events with that action name and parameters. The **out** action MoveCursor denotes that modules of type inputOutput can generate events with that action name and parameters.

In the above example, modules of type InputOutput may communicate with other modules by means of MoveCursor, while the other modules may communicate with modules of type InputOutput by means of MouseEvent.

□

Commentary

The ability to define type names in interfaces and to leave the full definition of the types to the corresponding modules provides the capability to define abstract data types in RAPIDE 1.0. The details of the type structure is hidden outside the module and the only means to access the type is through the operations provided in the interface. By bounding the types in the interface, some more information on the types is provided to users of the module. This gives the users the capability to use the type with a little more flexibility and also constrains modules of the interface to define constituent type names subject to the bounds constraint.

Commentary

The **provides**, **requires**, and **action** parts of an interface define a *contract* between the modules of a type and their users. A provided function name is a promise by the module to receive function calls and to compute return objects. A required function name signifies that the module has an ability to generate function calls and assumes that a means to compute return objects will be connected to it. Connections are assumed to be type correct, as explained in the RAPIDE 1.0 Architectures Reference Manual. An **in** action promises a means to receive events. An **out** action signifies an ability to generate events and assumes they will be connected to an observer. The notion of "assumption" is somewhat weaker in the asynchronous event case. If a required function is not connected, calls to it will result in an error or non-termination. If an **out** event is not observed, the generating module simply will not know.

4.3 Interface Derivation Declarations

Interface derivation declarations are used to copy portions of already existing interface types expressions into another interface type expression. This declaration is the inheritance mechanism of the RAPIDE 1.0 Types Language. The semantics of interface derivation declarations is very similar to that achieved by performing a textual copy (using a text editor) of portions of one interface type expression into another.

4.3.1 Syntax

interface_derivation_declaration ::=
    include *type_expression*
        [*include_modifier* '(' *identifier_list* ')']
        [**replace** '(' *replacement_list* ')'] ';'

include_modifier ::=
    only
  | **except**

replacement_list ::=
    replacement { ',' *replacement* }

replacement ::=
    identifier **to** *identifier*

4.3.2 Semantics

An interface derivation declaration copies a set of name declarations from a *source* interface type expression to a *target* interface type expression. The target interface type expression is the one which the interface derivation declaration immediately precedes or within which it appears. The source interface type expression is obtained by *normalizing* the type expression after the keyword **include**. A normalized type expression is an interface type expression that contains only name declarations. The process of normalization is described in Section 4.3.3.

The type expression after the keyword **include** may not contain references to any type expression within which the interface derivation declaration is present at any level of nesting.

If the derivation declaration immediately precedes the target interface type expression then all name declarations of the source interface are considered for copying into the target interface. If the derivation declaration appears within the target interface, then only name declarations in the same kind of interface declarative region in the source interface as the interface declarative region in the target interface containing the interface derivation declaration are considered for copying. This set of name declarations may be further restricted according to the following rules:

- If the interface derivation declaration contains the keyword **only** followed by an identifier list, only those name declarations whose identifiers are in this list are considered for copying.
- If the interface derivation declaration contains the keyword **except** followed by an identifier list, only those name declarations whose identifiers are not in this list are considered for copying.

If the interface derivation declaration does not contain the keyword **replace**, then the portion of the source interface type expression considered for copying is copied over to the position of the interface derivation declaration.

If the interface derivation declaration contains the keyword **replace** followed by a list of replacements, then each name declaration considered for copying that introduces the same identifier as that to the left of the keyword **to** in a replacement is modified by replacing the identifier it introduces with the identifier to the right of the same keyword **to**. Such a replacement is performed on a name declaration at most once. Furthermore, if any of the name declarations considered for copying contained references to any of the replaced identifiers, then these references are also replaced by the new identifiers. Once these replacements are performed, the name declarations considered for copying are copied over to the position of the interface derivation declaration. The above replacements do not change the source interface type expression.

Two replacements in the same replacement list may not contain the same identifier to the left of the keyword **to**.

The semantics of the copied name declarations are determined *after* the copying is complete — *i.e.*, the interface derivation declaration has an effect similar to textual copying.

4.3.3 Normalization of Type Expressions

The normalization of a type expression results in an interface type expression that contains only name declarations. Normalization is only defined for type expressions that denote interface types.

Normalization is performed as follows:

Step 1: Every type expression that denotes an interface type is either itself an interface type expression or is a reference to another interface type expression that defines the interface type. In the latter case, the type expression is replaced by the interface type expression it refers to.

Step 2: In the second step, any interface derivation declaration present in the interface type expression obtained in Step 1 is replaced by a list of name declarations according to the rules of Section 4.3.2. Note that this description is recursive — interface derivation makes use of normalized interface type expressions and vice versa.

Step 3: Any object name declaration that contains a list of more than one identifier is replaced by a list of object name declarations one for each identifier in the original object name declaration. Each of these new object name declarations contain exactly this identifier in its identifier list and its type expression is the same as that of the original object name declaration.

Example 10: *Deriving Colored Points from Points*

```
type Colored_Point is interface
provides
    include Point;
    Color : function() return Color_Map;
end interface;
```

The interface derivation declaration above has the effect of copying all the constituent declarations of interface Point into Colored_Point. The above interface is therefore equivalent to:

```
type Colored_Point is interface
provides
    X_Val : Integer;
    Y_Val : Integer;
    Distance : function(P : Point) return Integer;
    Color : function() return Color_Map;
end interface;
```

Note that the copied declarations are interpreted according to the scope and visibility rules of Colored_Point. Therefore, global references such as Integer and Color_Map may have different denotations within Colored_Point.

□

Example 11: *Modification to Previous Example*

```
type Colored_Point is interface
provides
    include Point except (Distance);
    Distance : function(P : Colored_Point) return Integer;
    Color : function() return Color_Map;
end interface;
```

In the earlier example, the parameter of the copied function Distance was of type Point. Suppose it was desired that this parameter be of type Colored_Point. This is achieved in the above example by copying all constituents of Point except Distance, and then declaring a new function Distance with the desired parameter type.

Note here that it might be desirable to be able to say:

```
type Colored_Point is interface
provides
    include Point replace (Point by Colored_Point);
    Color : function() return Color_Map;
end interface;
```

In other words, it might be useful to allow renaming of the type being included. This is not permitted but it shall be shown later (see page 41) how type constructors can be used to handle such situations.

□

4.4 Subtyping in Interface Types

This section describes the conditions under which an interface type I1 is a subtype of another interface type I2.

The subtyping rules use the notion of *characteristic function types* of type constructors. Characteristic function types are described in the next section, following which the subtyping rules are described.

4.4.1 Characteristic Function Types of Type Constructors

The characteristic function type of a type constructor (and similarly that of a type constructor denotation specification) is the type denoted by the function type expression constructed by making the formal parameter list of the function type the same as that of the type constructor. The characteristic function type does not have a result type (see Chapter 5).

Example 12: *A Characteristic Function Type*

```
function (type Element; Size : Integer);
```

This function type expression is the characteristic function type of the type constructor Deque in the example of 4.2 (see page 16).

□

4.4.2 The Subtyping Rules for Interfaces

For the purpose of this section, I1 and I2 refer to two arbitrary interface types. This section characterizes the situations in which I1 is a subtype of I2.

I1 is a subtype of I2 if the following hold:

1. All identifiers of constituents of the provides interface of I2 are also identifiers of some constituent(s) of the provides interface of I1.
2. All identifiers of constituents of the private interface of I2 are also identifiers of some constituent(s) of either the provides or the private interface of I1.
3. All identifiers of constituents of the requires interface of I1 are also identifiers of constituents of the requires interface of I2.
4. All identifiers of action declarations of I2 are also identifiers of some action declaration(s) of I1.
5. Every identifier, T, that has a type denotation specification in I2 also has a type denotation specification in I1.

   If the type denotation specification of T in I2 constrains the possible denotations in modules to be a subtype of some type T2, then either:

   (a) The type denotation specification of T in I1 constrains the possible denotations in modules to be a subtype of some type T1 such that T1 is a subtype of T2; or
   (b) The type denotation specification of T in I1 constrains the denotation in modules to be some fixed type T1′ such that T1′ is a subtype of T2.

   If the type denotation specification of T in I2 constrains the denotation in modules to be some fixed type, then the type denotation specification of T in I1 must also constrain the denotation in modules to be the same type.
6. Every identifier, T, that has a type constructor denotation specification in I2 also has a type constructor denotation specification in I1 such that the characteristic function type of the denotation specification of T in I1 is a subtype of the characteristic function type of the denotation specification of T in I2.

   If the type constructor denotation specification of T in I2 constrains the applications of (type constructor) denotations in modules to be a subtype of some type T2 (with formal parameters replaced by parameters from the application), then either:

   (a) The type constructor denotation specification of T in I1 constrains the applications of (type constructor) denotations in modules to be a subtype of some type T1 (with formal parameters replaced by parameters from the application). Furthermore, for each possible application to a (type constructor) denotation of T in I1, substitution of the parameters of the application for formal parameters in T1 and T2 results in T1 (after these substitutions) being a subtype of T2 (after these substitutions).
   (b) The type constructor denotation specification of T in I1 takes the form of a type constructor declaration such that for each possible application of the type constructor introduced by the declaration resulting in the type T1, substitution of the parameters of the application for formal parameters in T2 results in T1 being a subtype of T2 (after these substitutions).

   If the type constructor denotation specification of T in I2 takes the form of a type constructor declaration, then the type constructor denotation specification of T in I1 also takes the form of a type constructor declaration such that the two type constructors introduced are the same. The notion of "same type constructors" is defined in Chapter 6.

7. For each provides or private constituent in I2 with an object denotation specification, there is a corresponding constituent in I1 with an object denotation specification that introduces the same identifier, and such that the type of the denotation specification in I1 is a subtype of the type of the denotation specification in I2. Furthermore if the constituent in I2 is in its provides interface, then the corresponding constituent in I1 may not be in its private interface.

8. For each requires constituent in I1 with an object denotation specification, there is a corresponding requires constituent in I2 with an object denotation specification such that the type of the denotation specification in I2 is a subtype of the type of the denotation specification in I1.

9. For each action declaration in I2, there is a corresponding action declaration in I1 of the same mode (**in** or **out**) such that the denoted event type of the action declaration in I1 is a subtype of the denoted event type of the action declaration in I2.

10. Any subtype relationships that involves types bounded in I2 and that are required to hold due to the above rules need hold only under the assumption that this type is a subtype of its bound in I2.

11. Similarly, any subtype relationships that involves type constructors bounded in I2 and that are required to hold due to the above rules need hold only under the assumption that any application of the type constructor will be a subtype of its bound, where any type constructor parameters that occur in the bound are replaced by the actual values used in the application.

It may sometimes be the case (*e.g.*, with recursive types) that while determining whether I1 is a subtype of I2, one needs to determine this same result again as a subgoal; *i.e.*, there is a circularity in the subtype determination. Whenever such a circularity arises, the subgoal may be assumed to be true.

Example 13: *Points and Colored Points*

```
type Point is interface
provides
    X_Val : Integer;
    Y_Val : Integer;
    Distance : function(P : Point) return Integer;
end interface;

type Colored_Point_a is interface
provides
    include Point;
    Color : function() return Color_Map;
end interface;

type Colored_Point_b is interface
provides
    X_Val : Integer;
    Y_Val : Integer;
    Distance : function(P : Point) return Integer;
    Color : function() return Color_Map;
end interface;
```

```
type Colored_Point_c is interface
provides
    include Point except (Distance);
    Distance : function(P : Colored_Point_c) return Integer;
    Color : function() return Color_Map;
end interface;
```

The type Colored_Point_a is a subtype of type Point. This is because Colored_Point_a has all the constituents of Point and one more constituent. Colored_Point_b and Colored_Point_a are equal types as a result of the interface derivation rules. Therefore, Colored_Point_b is also a subtype of Point. However, Colored_Point_c is not related by subtyping to any of the other types in this example. This is because of the change in the function Distance. For Colored_Point_c to be a subtype of Point, the type of Distance in Colored_Point_c must be a subtype of the type of Distance in Point. This in turn requires that the parameter type of Distance in Point be a subtype of the parameter type of Distance in Colored_Point_c — *i.e.*, that Point is a subtype of Colored_Point_c. This is not the case.

□

Example 14: *Subtyping in the presence of Bounds*

```
type Payroll is interface
provides
    type Employee;
    Print_Salary : function(E : Employee) return Float;
end interface;

type Inc_Payroll is interface
provides
    type Employee;
    Print_Salary : function(E : Employee) return Float;
    Give_Raise : function(E : Employee) return Employee;
end interface;

type Payroll_With_Names is interface
provides
    type Employee <: interface
                         Name : String;
                         Salary : Float;
                     end interface;
    Print_Salary : function(E : Employee) return Float;
end interface;
```

```
type Inc_Payroll_With_Names is interface
provides
    type Employee <: interface
                        Name : String;
                        Salary : Float;
                     end interface;
    Print_Salary : function(E : Employee) return Float;
    Give_Raise : function(E : Employee) return Employee;
end interface;

type Safe_Payroll is interface
provides
    type Employee <: interface
                        Salary : Float;
                     end interface;
    Print_Salary : function(E : Employee) return Float;
end interface;

type Payroll_With_Ages is interface
provides
    type Salary_Type is Float;
    type Age_Type is Integer;
    type Employee is interface
                        Salary : Salary_Type;
                        Age : Age_Type;
                     end interface;
    Print_Salary : function(E : Employee) return Salary_Type;
    Print_Age : function(E : Employee) return Age_Type;
end interface;
```

Inc_Payroll is a subtype of Payroll because Inc_Payroll has the same constituents as Payroll and one more.

Payroll_With_Names and Payroll differ in only one respect — the type Employee in Payroll_With_Names has a bound. The presence of this bound restricts the denotations that Employee may be given in modules of type Payroll_With_Names. Hence Payroll_With_Names is a subtype of Payroll.

Note that to determine subtyping in detail in the above case, it is necessary to determine whether the type of Print_Salary in Payroll_With_Names is a subtype of the type of Print_Salary in Payroll. Since these types involve a bounded type, namely Employee, the subtyping relationship between the types of Print_Salary needs to hold only under the assumption that Employee is a subtype of its bound in Payroll_With_Names (which is the case).

Also note that Inc_Payroll is not a subtype of Payroll_With_Names since Inc_Payroll allows its modules to define Employee more liberally.

However, Inc_Payroll_With_Names is a subtype of Payroll_With_Names since they both impose the same restrictions on Employee, and Inc_Payroll_With_Names has an additional operation.

Now compare Safe_Payroll and Payroll_With_Names. The only difference between them is that their respective bounds for type Employee are different. However, the bound in Payroll_With_Names is a subtype of the bound in Safe_Payroll — *i.e.*, Safe_Payroll is more liberal in the kind of denotations that may be given to Employee. Hence, Payroll_With_Names is a subtype of Safe_Payroll.

The last type in this example is Payroll_With_Ages. In this type, constituent types are declared using full type declarations. The subtyping rules make Payroll_With_Ages a subtype of Payroll and Safe_Payroll. This is because the Payroll_With_Ages has all the constituents of Payroll and Safe_Payroll, and the denotation it gives Employee satisfies the bounds restrictions in Payroll and Safe_Payroll. Note however, that Payroll_With_Ages is not a subtype of Payroll_With_Names since the denotation of Employee Payroll_With_Ages does not satisfy the bounds restrictions in Payroll_With_Names.

□

Example 15: *Subtyping in the presence of Recursion*

```
type Point_a is interface
provides
    X_Val : Integer;
    Y_Val : Integer;
    Make_Point : function(X, Y : Integer) return Point_a;
end interface;

type Point_b is interface
provides
    X_Val : Integer;
    Y_Val : Integer;
    Dist_From_Origin : function() return Float;
    Make_Point : function(X, Y : Integer) return Point_b;
end interface;
```

To determine whether Point_b is a subtype of Point_a, it is first noted that Point_b has the constituents X_Val and Y_Val with exactly the same types as in Point_a. However, the constituent Make_Point has to be inspected in more detail. The parameters of Make_Point in Point_a and Point_b are the same, but their result types are different. Therefore, for the type of Make_Point in Point_b to be a subtype of the type of Make_Point in Point_a, Point_b has to be a subtype of Point_a. At this point there is a circularity. However, the RAPIDE 1.0 language rule for such circularity guarantees that modules of type Point_b will be substitutable for modules of type Point_a. Hence, Point_b is determined to be a subtype of Point_a.

□

Example 16: *Subtyping in the presence of Private Interfaces*

```
type Gadget_1 is interface
provides
    Start : function();
    Current_Value : Real;
    Increase : function(Amount : Real);
private
    Reset : function();
end interface;
```

```
type Gadget_2 is interface
provides
      Start : function();
      Current_Value : Real;
      Increase : function(Amount : Real);
      Decrease : function(Amount : Real);
      Reset : function();
private
      Shutdown : function();
end interface;
```

For Gadget_2 to be a subtype of Gadget_1, it is required that all the provides constituents of Gadget_1 be present in the provides interface of Gadget_2, and all the private constituents of Gadget_1 be present in either the provides or the private interfaces of Gadget_2. This is the case, and Gadget_2 is determined to be a subtype of Gadget_1.

□

Example 17: *Subtyping in the presence of Overloading*

```
type Viewer is interface
provides
      Display : function(T:Text);
      -- Show text
      Display : function(F:Figure);
      -- Show figure
      Display : function(M:DisplayMonitor);
      -- Change the monitor on which displaying is being done
end interface;

type Versatile_Viewer is interface
provides
      Display : function(D:Displayable);
      -- Show any displayable object
      Display : function(M:DisplayMonitor);
      -- Change the monitor on which displaying is being done
end interface;
```

Assume that Text and Figure are both subtypes of Displayable. In that case, modules of type Versatile_Viewer can be used as modules of type Viewer. This is because the first Display constituent of Versatile_Viewer can be the constituent that corresponds to both the first and second Display constituents of Viewer.

□

Example 18: *Subtyping with actions*

```
type MultiMediaIO is interface
action
      in    MouseEvent (B : Buttons; X, Y : Integer);
```

```
    in   FaxModem (B : BitVector);
    out  MoveCursor (X, Y : Integer);
    out  Audio (S : Signal);
    ...
end interface;
```

The MultimediaIO type is a subtype of the InputOutput type on page 17. It declares action names corresponding to the action names in InputOutput and additional **in** and **out** actions. Intuitively a module of Multimedia type should be usable in place of a module of InputOutput type. That is, it should be able to comunicate with other modules by means of events of the type denoted by MoveCursor, while the other modules may communicate with it by means of events of the type denoted by MouseEvent. So the environment should be unaware of the replacement. In addition it can receive or send comunication by means of additional events. Subtyping interprets asynchronous communication as meaning the module will operate even if it does not receive FaxModem events or other modules do not receive the Audio events that it generates.

□

Commentary

Subtyping in Rapide is a limited kind of structural subtyping, requiring name identification at the leaf nodes of the subtype subgoal search to reduce the size of the search. Subtyping does not take account of any semantic constraints, which is true of subtyping in all current programming languages. It distinguishes between functions and actions in that fewer requires functions (all other constituents being the same) lead to a subtype, whereas more **out** actions lead to a subtype. The intuition in doing this is that a module that calls a requires function must wait for the call to terminate, and so is making a demand on its environment, whereas a module generating an **out** event does not necessarily wait for, or require, a response. These distinctions between synchronous and asynchronous constituents are somewhat crude and indicate a general need to incorporate semantic constraints into subtyping.

4.5 The Selection Operation

The executable and architecture languages within the same language framework must provide a binary *selection operation* one parameter of which is an object expression and the other is an identifier.

4.5.1 Semantics

If α is an object expression that denotes a module of interface type I, and if C is the identifier of a constituent of I, then the selection operation applied to α and C denotes:

- a type if C has a type denotation specification;
- a type constructor if C has a type constructor denotation specification;

- an object if C has an object denotation specification.
- and an event type if C has an event type denotation specification.

In all cases, the denotation of the application of the selection operation to α and C must be consistent with the denotation specification of C in I.

Furthermore the use of the selection operation is constrained by the rules for private interfaces described in Page 9.

Chapter 5

Function Types

A function type contains a list of *formal parameters* and a *result type*. The formal parameters are similar to constituents of interface types in that they contain an identifier and a denotation specification. The result type is a type.

Within a function type, the identifiers introduced by its formal parameters must all be distinct.

5.1 Function Type Expressions

The syntactic construct that introduces function types is the *function type expression.*

5.1.1 Syntax

function_type_expression ::=
  function '(' [*formal_parameter_declaration_list*] ')' [**return** *type_expression*]
    [*FRAMEWORK_function_constraint*]

formal_parameter_declaration_list ::=
  formal_parameter_declaration { ';' *formal_parameter_declaration* }

5.1.2 Semantics

A function type expression contains a formal parameter declaration list within parentheses which describes the list of formal parameters of the function type denoted by the function type expression. If the formal parameter declaration list is omitted, the list of formal parameters of the function type is empty. The details of formal parameters are described in Section 5.2.

A function type expression may also include the keyword **return** followed by a type expression. If this is present, then the type denoted by the type expression after the keyword **return** is the result type of the function type denoted by the function type expression. If the keyword **return** is not present, then the result type is the interface type with no constituents.

The *function_constraint* construct must be defined in the constraint language within the same language framework.

Example 1: *Some Simple Function Types*

> **type** Projector_Function **is** **function**() **return** Integer;
>
> **type** Distance_Function **is** **function**(P : Point) **return** Integer;
>
> **type** Change_Setting **is** **function**(New_Value : Real);
>
> **type** Three_Way_Adder **is** **function**(X, Y, Z : Integer **is** 0) **return** Integer;

The first type, Projector_Function, describes parameterless functions that return integers. This is the type of the function constituents X_Val and Y_Val of the interface type Point in 4.1.

The second type, Distance_Function, describes functions that take one parameter of type Point and return an integer.

The third type, Change_Setting, describes functions that take one parameter of type Real. The result type is not specified and hence is implicitly set to the empty interface type.

The fourth type, Three_Way_Adder, describes functions of three integer parameters that return integer results. The three parameters have default denotations of 0. "0" is assumed to be a valid object expression of the executable language within the same language framework.

□

5.2 Formal Parameters

There are two kinds of formal parameters — the *formal type parameter* and the *formal object parameter*. The syntactic construct that introduces formal parameters is the *formal parameter declaration*.

5.2.1 Syntax

formal_parameter_declaration ::=
    formal_type_parameter_declaration
  | *formal_object_parameter_declaration*

formal_type_parameter_declaration ::=
    type *identifier* ['<:' *type_expression*]

formal_object_parameter_declaration ::=
    identifier_list ':' *type_expression* [**is** *FRAMEWORK_object_expression*]

5.2.2 Semantics

A formal type parameter is a formal parameter that has a denotation specification which requires the corresponding denotation in function calls (see Section 5.4) to be a type. The syntactic construct that introduces formal type parameters is the *formal type parameter declaration.* There are two forms of formal type parameters:

1. The simplest form of formal type parameter consists of the keyword **type** followed by an identifier. For this form of formal type parameter, the corresponding denotation in function calls may be any type.

2. In the second form, the delimiter '<:' followed by a type expression appears after the identifier of the formal type parameter. For this form of formal type parameter, the corresponding denotation in function calls must be a type that is a subtype of the type denoted by the type expression after the delimiter '<:'. Subtyping rules are described in Section 4.4 and Section 5.3.

A formal object parameter is a formal parameter that has a denotation specification which requires the corresponding denotation in function calls (see Section 5.4) to be an object of a specified type. The syntactic construct that introduces formal object parameters is the *formal object parameter declaration.* Each formal object parameter declaration contains an identifier list and introduces as many formal object parameters as there are identifiers in the identifier list. The corresponding (object) denotations in function calls must be of the type denoted by the type expression after the delimiter ':'.

A formal object parameter may be provided a *default denotation.* In the formal object parameter declaration, the default denotation is the object denoted by the object expression that appears after the keyword **is**. If a default denotation is present, it must be consistent with the denotation specification of the formal object parameter. Also, a denotation need not be provided for the corresponding parameter in a function call (see Section 5.4 for details).

Example 2: *Binary and Relational Operators*

> **type** Binary_Op(**type** T) **is** **function**(X,Y : T) **return** T;
>
> **type** Relational_Op(**type** T) **is** **function**(X,Y : T) **return** Boolean;
>
> Binary_Op has been defined to be a type constructor, which when applied to an actual parameter T, results in a function type which describes binary functions on T. For example, Binary_Op(Integer) is a function type that describes binary operations on integers. Similarly, Relational_Op(T) describes relations of two arguments on T.

□

Example 3: *Function Type describing Min and Max Functions*

> **type** Min_Max_Functions **is** **function**(**type** T;
Ordered : Relational_Op(T);
X, Y : T) **return** T;
>
> This example describes the type of functions such as Min and Max of objects of a type T that has an ordering relation Ordered associated with it. Unlike the previous examples (binary and relational operators), the type T is now a parameter of the function type, as opposed to a parameter of the type constructor which was visible in the function type expression. Given this difference, functions of type

Min_Max_Functions are polymorphic with respect to type T — *i.e.*, they will work for any type T that has a function Ordered defined for it.

Suppose a more specialized type was required that restricted the kinds of types which could be denoted by T. This could be done by placing a bound in the formal type parameter declaration of T. For example:

```
type Min_Max_Functions_a is function(type T <: interface
                                                  provides
                                                    Index : Integer;
                                                end interface;
                                     Ordered : Relational_Op(T);
                                     X, Y : T) return T;
```

Functions of this type expect the type parameter T to be an interface type that contains at least the constituent Index of type Integer.

□

Example 4: *Some Personnel Operations*

```
type Simple_Employee is interface
                          provides
                            Name : String;
                            Salary : Real;
                        end interface;

type Salary_Info is interface
                      provides
                        Salary : Real;
                    end interface;

type Simplify is function(type T <: Simple_Employee;
                          E : T) return Simple_Employee;

type Give_Raise is function(type T <: Salary_Info;
                            E : T) return T;
```

It this example, two types are declared that describe certain kinds of employee records. The first, Simple_Employee, is an interface that describes what a simple employee record looks like. The second, Salary_Info, describes the minimal amount of information required in a record to perform operations related to salaries of employees. Two function types follow. The first, Simplify, takes as parameters an employee type and an employee of that type, which must therefore have at least all the constituents of a simple employee record, and returns a simple employee record. The second, Give_Raise, describes functions that operate on records that contain at least a salary field and return a similar record. An example of a function of this type is one that takes an arbitrary employee type that contains a salary field, and an employee record of that type, adds a certain amount to the salary field and returns the resulting record with the other fields unchanged.

□

5.3 Subtyping Rules for Function Types

A function type F1 is a subtype of a function type F2 if the following hold:

(Some of the rules that follow refer to the i^{th} parameter of the function types. These rules have to hold for all values of i from 1 through n, where n is the number of parameters of the function type with less number of parameters.)

1. If F1 has more parameters than F2 (F2 has n parameters), then all parameters of F1 beyond its n^{th} parameter have default denotations.
2. The identifier associated with the i^{th} parameter of F1 and the identifier associated with the i^{th} parameter of F2 are the same.
3. If the i^{th} parameter of F1 is a type parameter, then the i^{th} parameter of F2 is also a type parameter. In this case, if the denotation specification of the i^{th} parameter of F1 requires corresponding denotations to be subtypes of a type T1, then the denotation specification of the i^{th} parameter of F2 also requires corresponding denotations to be subtypes of a type T2 such that T2 is a subtype of T1.
4. If the i^{th} parameter of F1 is an object parameter, then the i^{th} parameter of F2 is also an object parameter. In this case, the type of the i^{th} parameter of F2 is a subtype of the type of the i^{th} parameter of F1.
5. The result type of F1 must be a subtype of the result type of F2.
6. Any subtype relationships that involves types bounded in F2 and that are required to hold due to the above rules need to hold only under the assumption that this type is a subtype of its bound in F2.

The following examples are based on other examples that were presented earlier in this chapter.

1. Consider the types T1 defined as Relational_Op(Simple_Employee) and T2 defined as Relational_Op(Salary_Info) as described in the previous section. T2 is a subtype of T1 because the result types of the two functions are the same, they have the same number of parameters, and the parameter type of T1, namely Simple_Employee, is a subtype of the parameter type of T2, namely Salary_Info.
2. However, the types Binary_Op(Simple_Employee) and Binary_Op(Salary_Info) are not related to each other by subtyping. For one function type to be a subtype of another, its argument types must be supertypes and its result type must be a subtype of the argument types and result type respectively of the other type. In this case, both the argument and result types are related in similar ways.
3. The type Min_Max_Functions is a subtype of the type Min_Max_Functions_a. This is because of the bound present in the formal type parameter of Min_Max_Functions_a. Functions of type Min_Max_Functions may be substituted by functions of type Min_Max_Functions_a without typing errors since the latter functions accept a less restricted class of type denotations for T.
4. The type Three_Way_Adder is a subtype of the types Projector_Function and Binary_Op(Integer). Functions of type Three_Way_Adder may be used as functions of the other types, the extra parameters are simply given their default denotations. Also, Projector_Function is a subtype of both Distance_Function and Three_Way_Adder since Projector_Function has fewer number of parameters than the others do. However, Distance_Function is not a subtype of Three_Way_Adder since the names of their first parameters are different.

5. Note that according to the rules of type equality, the types Projector_Function and Three_Way_Adder are equal.

6. The type Give_Raise is a subtype of the type Simplify. They both have a formal type parameter T, and the bound for T in Simplify is a subtype of the bound for T in Give_Raise. They both have a formal object parameter E. The types of this parameter are the same in both cases, namely the type parameter T. The result types of the two functions have to be compared now. This comparison has to be made under the assumption that T is a subtype of its bound in Simplify, namely, Simple_Employee. Hence, it follows that the result type of Give_Raise is a subtype of the result type of Simplify.

5.4 The Function Call Operation

The executable language within the same language framework must provide a *function call operation* that takes an object expression and a list of identifier–denotation pairs as parameters.

When the object expression denotes a function and the list of identifier–denotation pairs is such that:

1. The identifiers in this list are all distinct;

2. All identifiers in this list appear in some formal parameter in the type of the above-mentioned function and the corresponding denotation is consistent with the denotation specification of the formal parameter; and

3. An identifier–denotation pair exists for all formal parameters that do not have a default denotation.

then the function call operation must be an object expression that denotes an object whose type is the result type of the (function) type of the above-mentioned function.

Chapter 6

Type Constructors

Type constructors are templates for types. The templates contain *slots* into which type expressions and object expressions may be placed to obtain a type expression. Type constructors are defined using *type constructor declarations* and their "instantiation" to type expressions is called *type constructor application*.

6.1 Type Constructor Declarations

A type constructor declaration introduces an identifier and gives it a type constructor as its denotation.

6.1.1 Syntax

type_constructor_declaration ::=
    type *identifier* '(' [*formal_parameter_declaration_list*] ')' [*FRAMEWORK_type_constructor_constraint*]
        is *type_expression* ';'

6.1.2 Semantics

The formal parameter declaration list introduces a list of formal parameters each with its own identifier. Its semantics is identical to the formal parameter declaration list of function types. The type expression is the template of the type constructor and may contain identifiers of formal parameters in which case these references are the slots of the template.

The identifiers that define the slots in a type constructor may only appear *(1)* where a type expression is expected, in which case the identifier must have a type denotation specification in the formal parameter list; or *(2)* where an object expression is expected, in which case the identifier must have an object denotation specification in the formal parameter list.

The *type_constructor_constraint* construct must be defined in the constraint language within the same language framework.

Example 1: *A list type constructor*

```
type List(type T) is interface
provides
    First : T;
    Rest : function() return List(T);
end interface;
```

The type constructor List has a type parameter T. The type List(T) is used in the body of the type constructor List.

The application of the type constructor List to any type S produces an interface type where the actual parameter S replaces the formal parameter T in the body of the constructor List.

□

Example 2: *Generic Set Interfaces*

```
type SetType(type Elem) is interface
provides
        Insert :  function(E: Elem);
        Delete : function(E: Elem);
        Union : function(S: SetType(Elem));
        Is_Member : function(E: Elem) return Boolean;
private
    List_of_Members : function() return List(Elem);
end interface;
```

This example illustrates the use of type constructors to define sets of an arbitrary element type. SetType is a type constructor. It takes one type parameter Elem which can be replaced by any type during application.

□

Example 3: *A type constructor with function Less_Than*

```
type Less_Than_Mixin(type T) is interface
provides
     Less_Than : function(X : T) return Boolean;
end interface;
```

Application of the type constructor Less_Than_Mixin to any type S produces an interface where the formal parameter T is replaced by the actual parameter S in the body of the constructor Less_Than_Mixin. Application of type constructors will be discussed in the next section.

The type constructor Less_Than_Mixin can be used to place constraints on parameters of other type constructors. The next example illustrates this.

Less_Than_Mixin(I) can be included in another interface I. The result of this inclusion is that the interface I "gets" the less than function as a constituent.

□

Example 4: *Using Less_Than_Mixin in a subtype constraint*

```
type Sorting_Functions(type T <: Less_Than_Mixin(T)) is interface
provides
    function Quick_Sort(A : List(T)) ... ;
    function Heap_Sort(A : List(T)) ... ;
    ...
end interface;
```

The type constructor Sorting_Functions has an interface with sorting functions specific to lists with elements of type S whenever it is applied to any type S.

The constraint T <: Less_Than_Mixin(T) on the formal parameter T above expresses the fact that the actual parameter must have a constituent function Less_Than of the appropriate type.

Recursive constraints are useful in practice to express constraints on parameters of type constructors, as demonstrated above.

□

6.2 Type Constructor Expressions

The syntactic construct that denotes a type constructor is the *type constructor expression*.

6.2.1 Syntax

type_constructor_expression ::=
    identifier
  | *FRAMEWORK_type_constructor_expression*

6.2.2 Semantics

Type constructor declarations cause identifiers to denote type constructors. In certain contexts as described in Chapter 7 these identifiers may be used as type constructor expressions. The type constructor denoted by the type constructor expression is then the same as the type constructor denoted by the identifier according to the type constructor declarations.

The executable language within the same language framework may provide syntactic constructs that denote type constructors — these constructs may also be used as type constructor expressions. If any of these constructs defined by the executable language have an identical syntax to the other type constructor expressions, then they must also have the same semantics.

6.3 Type Constructor Applications

A type constructor application replaces the slots in a type constructor with type expressions and object expressions to result in a type expression.

6.3.1 Syntax

type_constructor_application ::=
    type_constructor_expression '(' [*actual_parameter_list*] ')'
  | *FRAMEWORK_type_constructor_declaration*

actual_parameter_list ::=
    actual_parameter { ',' *actual_parameter*}

actual_parameter ::=
    type_expression
  | *FRAMEWORK_object_expression*

6.3.2 Semantics

A type constructor application consists of a type constructor expression and a list of *actual parameters* Each actual parameter corresponds to a particular formal parameter of the type constructor denoted by the type constructor expression. The actual parameters and formal parameters are matched by position — the i^{th} actual parameter corresponds to the i^{th} formal parameter. Each actual parameter must denote a type or an object such that this denotation is consistent with the denotation specification of the corresponding formal parameter. There may be more actual parameters than formal parameters in which case the extra actual parameters are ignored. Also, if there are more formal parameters than actual parameters, all the extra formal parameters must have default denotations, and these default denotations are used as the denotations of the actual parameters.

Each slot of the template of the type constructor is replaced by the type expression or object expression of the corresponding formal parameter. This expression is either the actual parameter or the default denotation as the case may be. Once this replacement has been performed, the template is interpreted as a type expression at the location of the type constructor application.

The executable language within the same language framework may provide alternate syntax for type constructor applications by defining expansions for the non terminal *executable_language_type_constructor_application*.

Example 5: *Application of type constructors*

> **type** Integer_List **is** List(Integer);
>
> **type** IntegerSetType **is** SetType(Integer);

Applying the list and set type constructors to the type Integer results in interface definitions for lists and sets of integers respectively.

□

Example 6: *Renaming effect with type constructors*

```
type Point_Generator(type Point_Type) is interface
provides
    X_Val : Integer;
    Y_Val : Integer;
    Distance : function(P : Point_Type) return Integer;
end interface;

type Point is Point_Generator(Point);

type Colored_Point is interface
provides
    include Point_Generator(Colored_Point);
    Color : function() return Color_Map;
end interface;
```

Earlier (on page 20), it was mentioned that type constructors could be used in situations where one felt a need to rename the type being included. In this example, it is shown how the same effect is achieved.

□

Example 7: *Examples of Inheritance with Type Constructors*

```
type Linear_Collection(type Collection_Name, type Element_Type) is interface
provides
    New_Collection : function() return Collection_Name;
    Insert_At_Top : function(E:Element_Type) return Collection_Name;
    Delete_And_Return_Top : function() return Element_Type;
    Insert_At_Bottom : function(E:Element_Type) return Collection_Name;
    Delete_And_Return_Bottom : function() return Element_Type;
    Is_Empty : function() return Boolean;
end interface;

type Double_Ended_Queue(type Element_Type) is interface
provides
    include Linear_Collection(Double_Ended_Queue, Element_Type)
        replace (New_Collection by New_Dequeue,
                 Insert_At_Top by Enqueue1,
                 Delete_And_Return_Top by Dequeue1,
                 Insert_At_Bottom by Enqueue2,
                 Delete_And_Return_Bottom by Dequeue2);
end interface;

type Stack(type Element_Type) is interface
provides
    include Linear_Collection(Stack, Element_Type)
```

```
        except (Insert_At_Bottom, Delete_And_Return_Bottom)
        replace (New_Collection by Empty_Stack,
                 Insert_At_Top by Push,
                 Delete_And_Return_Top by Pop);
end interface;

type Queue(type Element_Type) is interface
provides
    include Linear_Collection(Queue, Element_Type)
        except (Insert_At_Top, Delete_And_Return_Bottom)
        replace (New_Collection by Empty_Queue,
                 Delete_And_Return_Top by Dequeue;
                 Insert_At_Bottom by Enqueue);
end interface;

type Stream(type Element_Type) is interface
provides
    include Linear_Collection(Stream, Element_Type)
        except (Insert_At_Top, Delete_And_Return_Bottom)
        replace (New_Collection by New_Stream,
                 Delete_And_Return_Top by Get,
                 Insert_At_Bottom by Put);
end interface;

type Integer_Dequeue is Double_Ended_Queue(Integer);
type Integer_Stack is Stack(Integer);
type Integer_Queue is Queue(Integer);
type Integer_Stream is Stream(Integer);
```

The first interface type Linear_Collection defines the generic operations one might need on a simple list of items. It has two type parameters – the first is to achieve the renaming of recursive constituents (as in the previous example) and the second is the type of the elements that will be put in the collection.

The next four declarations illustrate how some common collections can be defined be reusing the definition of the type Linear_Collection. A two-ended queue is essentially just a linear collection with a standard naming of operations. Stacks and queues are obtainable by straightforward restrictions of two-ended queues. The interface for streams is identical to that of queues modulo renaming of the identifiers.

The final set of declarations define the various kinds of collections for integers.

□

6.4 Conformance and Equality

A type constructor declaration *conforms* to a type constructor name declaration if and only if the characteristic function of the type constructor declaration is a subtype of the characteristic function of the type constructor name declaration.

Two type constructors are equal (*i.e.*, are the same) if they have identical formal parameter lists and for every possible type constructor application to these type constructors, the resulting types are equal.

Chapter 7

Naming Rules

This chapter lists the legal uses of identifiers as expansions of *(1)* type expressions, and *(2)* type constructor expressions.

The Use of Constituents

Any identifier given a type denotation specification or a type constructor denotation specification within an interface type expression may be used anywhere within the interface expression after its own declaration except where it is hidden within nested declarations. This use of the identifier denotes a type or a type constructor as defined in modules of the interface type containing this identifier as a constituent.

The Use of Formal Parameters

Any identifier given a type denotation specification within a function type expression (type constructor declaration) may be used anywhere within the function type expression (type constructor declaration) after its own declaration except where it is hidden within nested declarations. This use of the identifier denotes a type as defined in function calls (type constructor applications) of the function type (type constructor) containing the identifier as a formal parameter.

Type and Type Constructor Declarations

The identifier introduced by a type declaration or a type constructor declaration may be used within the type expression after the keyword **is** of the same declaration to denote the entire type except where it is hidden within nested declarations.

The executable language within the same language framework may also define *visibility rules* that characterize additional valid uses of identifiers as type expressions and type constructor expressions.

Appendix A

Syntax Summary

type_expression ::=
    interface_type_expression
  | *function_type_expression*
  | *type_constructor_application*
  | '(' *type_expression* ')'
  | *identifier*
  | *FRAMEWORK_type_expression*

type_declaration ::=
    type *identifier* **is** *type_expression* ';'

interface_type_expression ::=
    {*interface_derivation_declaration*}
    interface
        {*interface_declarative_item*}
        {*interface_declarative_region*}
        [**private** {*interface_declarative_region*}]
    end [**interface**]

interface_declarative_region ::=
    interface_provides_part
  | *interface_requires_part*
  | *interface_action_part*
  | *FRAMEWORK_declarative_region*

provides_part ::=
    provides
        {*interface_declaration_item*}

requires_part ::=
    requires
        {*interface_declaration_item*}

action_part ::=
    action
        {*action_declaration*}

interface_declarative_item ::=
  name_declaration
  interface_derivation_declaration
  FRAMEWORK_interface_constraint

action_declaration ::=
  in { *action_name_declaration* } ';'
| **out** { *action_name_declaration* } ';'
| *interface_derivation_declaration*

name_declaration ::=
  type_name_declaration
| *type_constructor_name_declaration*
| *object_name_declaration*
| *FRAMEWORK_name_declaration*

type_name_declaration ::=
  type *identifier* ';'
| **type** *identifier* '<:' *type_expression* ';'
| **type** *identifier* **is** *type_expression* ';'

type_constructor_name_declaration ::=
  type *identifier* '('[*formal_parameter_declaration_list*]')' ';'
| **type** *identifier* '('[*formal_parameter_declaration_list*]')' '<:' *type_expression* ';'
| **type** *identifier* '('[*formal_parameter_declaration_list*]')' **is** *type_expression* ';'

object_name_declaration ::=
  identifier_list ':' *type_expression* ';'

action_name_declaration ::=
  identifier_list '('[*formal_parameter_declaration_list*]')' ';'

identifier_list ::=
  identifier { ',' *identifier* }

interface_derivation_declaration ::=
  include *type_expression*
    [*include_modifier* '(' *identifier_list* ')']
    [**replace** '(' *replacement_list* ')'] ';'

include_modifier ::=
  only
| **except**

replacement_list ::=
  replacement { ',' *replacement* }

replacement ::=
  identifier **to** *identifier*

function_type_expression ::=
    function '(' [*formal_parameter_declaration_list*] ')' [**return** *type_expression*]
        [*FRAMEWORK_function_constraint*]

formal_parameter_declaration_list ::=
    formal_parameter_declaration { ';' *formal_parameter_declaration* }

formal_parameter_declaration ::=
    formal_type_parameter_declaration
  | *formal_object_parameter_declaration*

formal_type_parameter_declaration ::=
    type *identifier* ['<:' *type_expression*]

formal_object_parameter_declaration ::=
    identifier_list ':' *type_expression* [**is** *FRAMEWORK_object_expression*]

type_constructor_declaration ::=
    type *identifier* '(' [*formal_parameter_declaration_list*] ')' [*FRAMEWORK_type_constructor_constraint*]
        is *type_expression* ';'

type_constructor_expression ::=
    identifier
  | *FRAMEWORK_type_constructor_expression*

type_constructor_application ::=
    type_constructor_expression '(' [*actual_parameter_list*] ')'
  | *FRAMEWORK_type_constructor_declaration*

actual_parameter_list ::=
    actual_parameter { ',' *actual_parameter* }

actual_parameter ::=
    type_expression
  | *FRAMEWORK_object_expression*

Glossary

action interface: A portion of the interface type that is visible to the entire system. See page 9.

action name declaration: The syntactic construct that, within an action delcaration, introduces a list of names and associates it with a formal parameter list. See page 14.

actual parameter: The types or objects that are used in type constructor applications to replace the slots of the type constructor template. See page 38.

characteristic function type: A representation of certain aspects of type constructors and type constructor name declarations as function types. See page 21.

conformance: A relationship that must hold between a type constructor denotation specification and corresponding type constructor declarations defined in modules. See page 40.

constituent: The "things" that comprise interface types. See page 9.

default denotation: The object denoted by this is used as the parameter in a function call expression if the function call does not provide its own object. See page 31.

denotation specification: The constraint part of an interface declaration which specifies the set of legal denotations that modules of that interface type may provide or require of the identifier of the constituent declaration. See page 9.

denotation: The meaning of a syntactic construct. For example, an interface type expression (a syntactic construct) "denotes" an interface type (a semantic entity). See page 5.

event type denotation specification: A denotation specification that requires its corresponding denotations to be event types. See page 14.

formal object parameter declaration: The syntactic construct that introduces formal object parameters. See page 31.

formal object parameter: This is a kind of formal parameter that requires denotations of actual parameters in function call expressions to be objects. See page 30.

formal parameter declaration: The syntactic construct that introduces formal parameters. See page 30.

formal parameter: The part of a function type that describes the parameter types of a function of the type. See page 29.

formal type parameter declaration: The syntactic construct that introduces formal type parameters. See page 31.

formal type parameter: This is a kind of formal parameter that requires denotations of actual parameters in function call expressions to be types. See page 30.

function call operation: An operation on functions and actual parameters of its (function) type that must be provided by the executable language within the same language framework. See page 34.

function type expression: The syntactic construct that introduces function types. See page 29.

function type: They describe functions by specifying their parameter and result types. See page 5.

function: Members of function types. See page 5.

interface derivation declaration: An inheritance mechanism for interface types. These expand into name declarations. See page 11.

interface type expression: The syntactic construct used to denote interface types. See page 9.

interface type: They describe encapsulations of system components. They describe the interface of the system component with other system components. See page 5.

module: Members of interface types. See page 5.

name declaration: The fundamental kind of declaration of an interface type. See page 11.

normalization: The process of representing an interface type using an interface type expression that contains only name declarations. See page 18.

object denotation specification: A denotation specification that requires its corresponding denotations to be objects. See page 14.

object expression: A syntactic construct defined in the executable language within the same language framework. These denote objects. See page 5.

object name declaration: The syntactic construct that introduces a list of names and associates it with an object denotation specification. See page 14.

object: Members of types. See page 5.

private interface: The portion of the interface type that is visible only within modules of the same type. See page 9.

provides interface: A portion of an interface type that declares constituents that are visible to the entire system. See page 9.

requires interface: , See page 9.

result type: The part of a function type that describes the type of a function call expression. This is therefore also the “return type” of functions. See page 29.

selection operation: An operation on modules and the constituent identifiers of its (interface) type that must be provided by the executable language within the same language framework. See page 27.

slot: The portion of the type constructor template that may be replaced by types or objects to obtain a type. See page 35.

subtype: A relationship between two types. When one type is a subtype of the other, objects of the first type may be used as if they are objects of the second type. See page 7.

type constructor application: The process of replacing the slots in type constructors by types or objects to obtain a type. See page 35.

type constructor declaration: The syntactic construct that introduces an identifier and gives it a type constructor as its denotation. See page 35.

type constructor denotation specification: A denotation specification that requires its corresponding denotations to be type constructors. See page 14.

type constructor expression: The syntactic construct that denotes a type constructor. See page 37.

type constructor name declaration: The syntactic construct that introduces a name and associates it with a type constructor denotation specification. See page 14.

type constructor: These are parameterized templates of types from which types may be constructed by replacing the parameters by types or objects. See page 35.

type declaration: A syntactic construct that introduces an identifier and gives it a type as its denotation. See page 7.

type denotation specification: A denotation specification that requires its corresponding denotations to be types. See page 14.

type expression: The syntactic construct that denotes types. See page 5.

type name declaration: The syntactic construct that introduces a name and associates it with a type denotation specification. See page 14.

Index