Proposals for Enhancement of the Ada Programming Language:

A Software Engineering Perspective

November 1994

Author:

Mats Weber
Software Engineering Lab
Swiss Federal Institute of Technology
EPFL
CH-1015 Lausanne
Switzerland

e-mail: Mats.Weber@lglsun.epfl.ch

Thesis Committee:

Prof. Jean-Daniel Nicoud, chair,

Prof. Alfred Strohmeier, EPFL, thesis supervisor,

Prof. Hans-Heinrich Naegeli, University of Neuchâtel, external expert,

Prof. Erhard Ploedereder, University of Stuttgart, external expert,

Prof. Charles Rapin, EPFL, internal expert,

S. Tucker Taft, Intermetrics, external expert.

This document is provided as is, without any warranty.

Table of Contents

Abstract

This thesis is a critique of the Ada 83 programming language with emphasis on the construction of reusable components and their composition, and more generally on programming “in the large”.

Ada 83's deficiencies in that area are first described, and then solutions are proposed to most of the problems.

The main part of the thesis is a proposal for object-oriented extensions, making classes and objects with inheritance available through package and task types, as a very natural extension of Ada 83's task types.

These proposals can be viewed as an alternative to Ada 9X's tagged types, with which a comparison is made.

Résumé

Cette thèse est une critique du langage de programmation Ada 83, mettant l'accent sur la construction de composants réutilisables et leur composition, et plus généralement sur le développement de gros systèmes.

D'abord, les défauts d'Ada 83 dans ce domaine sont identifiés, puis des solutions sont proposées à la plupart des problèmes.

La partie principale de la thèse consiste en une proposition d'extension du langage dans le domaine de l'orienté objet. Les classes et objets sont fournis sous la forme de types paquetage et de types tâche, comme une extension très naturelle des types tâche d'Ada 83.

Ces propositions peuvent être considérées comme une alternative aux types étiquetés d'Ada 9X. Une comparaison avec ceux-ci est donnée.

Acknowledgments

I would like to thank Prof. Alfred Strohmeier for accepting to be the supervisor of my thesis.

Thanks to Prof. Hans-Heinrich Nägeli, Prof. Jean-Daniel Nicoud, Prof. Erhard Ploedereder, Prof. Charles Rapin, and Tucker Taft for accepting to be members of the Ph.D. examining board.

Thanks to Norbert Ebel for encouraging me to start writing this thesis.

Thanks to Magnus Kempe, Prof. Rolf Ingold, Catherine Jean and Stéphane Barbey for their careful review and comments on several versions of the text.

1. Introduction

Introduced in 1983 as a standard, the Ada programming language is the first attempt at creating a programming language from requirements and standardizing it before it is used. This process seems to have been successful but, ten years later, with the emergence of object-oriented programming and extensive use of abstraction in the construction of software systems, the language shows some weaknesses. This thesis presents these weaknesses along with extension proposals making Ada more powerful in specifying abstractions and extending existing modules to fulfill new requirements.

This text first presents some deficiencies of the Ada 83 programming language, with an accent on software engineering and programming in the large. Then, object-oriented extensions are introduced, solving some of the language's limitations. Finally, solutions to some of the remaining problems, not covered by object-oriented extensions, are given.

1.1. Structure of this Document

This document is divided into five chapters which are briefly described here:

1) Introduction.

2) Some Problems in Ada 83: describes weaknesses of Ada 83 with an accent on the development of reusable software components, but without giving solutions.

3) Higher Level Extensions: describes proposed object-oriented extensions for Ada, solving some of the problems exposed in chapter 2. A comparison with the current proposal for Ada 9X [Ada 9X 94] can be found in section 3.4.1.

4) Solutions to the Lower Level Problems: presents solutions to problems identified in chapter 2 but not covered by the object-oriented extensions.

5) Conclusion.

1.2. Terminology

The terms *class*, *object* and *method* will be used in their usual meaning in object-oriented programming, i.e. to denote abstract entities that are modeled with the corresponding language constructs, but not the constructs themselves. For instance integers with their operations are a class even if they are not implemented with the class construct of some programming language.

The term *Abstract Data Type* (ADT) should not be understood in a too formal sense, but rather as meaning “some encapsulated type along with operations on that type”.

1.3. Presentation of the Goals

I think that the most important weakness of Ada 83 (and, for that matter, of most other programming languages), lies in the expression of ADTs, their composability, extensibility and adaptability. Therefore the main goal of this document is to extend and improve the language in these directions.

The key mechanisms for achieving these goals are generics, inheritance and polymorphism. This is why they are given extensive treatment both in the critique of existing mechanisms and in the presentation of new constructs.

The necessity for object-oriented extensions, presented in chapter 3, comes not only from the fact that the object-oriented paradigm becomes more and more widespread, but also from the fact that they solve problems related with the composition and extension of ADTs.

1.3.1. Why Ada ?

I think that “pure” object-oriented languages, such as Eiffel, are not suitable for the general usage that Ada was designed for, because some mechanisms that are widely used in the classical procedural style of programming are not available. For instance, Eiffel has no module construct, which forces programmers to create special, stateless classes for things as simple as a library of mathematical functions.

Other languages, such as Classic-Ada, add object-oriented constructs through a special data type and notations, thus providing an “object subsystem” without changing the base language, very much like a “callable Smalltalk”. The main drawback of this approach is that the objects and classes are not well integrated into the base language.

I believe that Ada 83 provides the solid procedural base language that is needed in a general purpose language, and that it is also well suited for object-oriented extensions.

Following are a few comments on some other programming languages:

Modula-3

Modula-3 [Modula-3 89a] is probably one of the simplest languages that is able to fulfill requirements similar to those that lead to Ada. Following are some interesting characteristics of Modula-3:

- A *module* can implement more than one *interface*, which allows related abstractions to be specified separately but implemented together. Most other languages have a one-to-one correspondence between interfaces and modules.

- Incomplete types can be declared in interfaces, and then completed incrementally through several *revelations*, giving more details on how the type is implemented at each step.
- Modula-3 uses the same construct, namely interfaces, for specifying generic parameters and for exporting from modules: this guarantees the symmetry between declarable entities and entities that can be generic parameters.
- Methods are provided in the form of procedure components of objects. This shows that a very simple mechanism can be used to provide inheritance.

Modula-3's main weakness is in the expression of parallelism: only a very primitive Threads interface is offered, that is very far from the expressive power of Ada's tasks.

CLU

CLU [CLU 81] is certainly one of the best designed languages of its time. The way it defines objects and variables is very interesting: a variable is a *name* that *denotes* an object. This model enables a very clean definition of mechanisms such as parameter passing and constants. In most other programming language definitions, variables and objects are not as well distinguished, which sometimes leads to ambiguities.

CLU's iterator construct serves as a base for the proposal made in section 3.2.16.

Eiffel

Eiffel [Meyer 88] is one of the first pure object-oriented languages that was designed with compilation in mind. All type checks can be done at compile-time, thus allowing efficient implementations.

The fact that the class is the only mechanism for controlling visibility is sometimes awkward when programming in a classical, non-object-oriented, style. Eiffel's *once functions* are a proof that the language is lacking a module construct.

The object-oriented extensions presented in chapter 3 take many ideas from Eiffel, e.g. anchors (declaration by association) and constrained genericity.

C/C++

More than twenty years after the invention of modules and Backus-Naur notations for expressing the syntax of programming languages, the popularity of C and C++, with their file-based import and visibility system as well as the lack of a grammar for defining their syntax, is beyond my understanding.

On the positive side, the class construct offered in C++ provides the functionality that one expects from a modern object-oriented language.

Two key concepts, namely templates (generics) and exceptions, are almost defined for C++ but their availability in current implementations is not yet general enough to make them usable in a portable manner.

The (momentary) lack of a standard for C++ is a quite severe problem because many differences exist between different implementations.

2. Some Problems in Ada 83

In this chapter, some problems of the Ada 83 language are described. Here, as throughout this document, the emphasis is on the development of large programs, software engineering and composability/reusability of software components.

First, some problems with Ada's generics are presented. Since genericity is Ada's most important mechanism for the composition of software components, its treatment takes up a large part of this chapter (sections 2.1 to 2.7). Then, in section 2.11, problems specific to the expression of abstract data types are identified. The remainder of the chapter deals with miscellaneous problems such as the order of elaboration of library units, problems in developing subsystems, exceptions, visibility rules, etc.

Solutions for some of the problems presented are given in chapter 4; in such cases, a reference to the section containing the solution is given. Some problems are not solved in this document, for one of the following reasons:

- the change to the language would be too radical,
- the problem is solved indirectly by the object-oriented extensions presented in chapter 3,
- other solutions already exist, such as Ada 9X.

2.1. Lack of Symmetry between Declarable Entities and Generic Parameters

One of the main uses of generics is for providing software components without knowing the exact context of their eventual use. In this view, the generic parameters specify what the generic needs in order to implement the services it provides, and the specification of the generic specifies those services. There is some kind of contract between the generic and its user, where the generic's commitment is “if you give me <generic parameters>, then I will provide you with <generic specification>”.

In order for this paradigm to be generally applicable for the partition of a program into reusable and logically coherent modules, as well as for reusing existing component libraries, there must be an almost perfect symmetry between the set of entities that may be declared and the set of entities that can be imported as generic parameters.

In Ada 83, this symmetry is broken for the following entities, which can be declared but not passed as generic parameters:

- literals,

- named numbers,
- exceptions,
- subtypes,
- packages,
- tasks,
- generic units.

The rest of this section discusses the adverse effects that this asymmetry can have on program structure, often forcing programmers to hard-code components in-line when a generic formulation would have been more reusable and maintainable, as well as less error-prone.

2.1.1. Generics cannot be Generic Parameters

It is not possible to pass generics as generic parameters. This is a severe restriction because some abstractions which have a generic unit as a part of their specification cannot be passed as generic parameters. This situation is often encountered with iterators, as shown in the following example:

```
generic
   type Item_Type is private;
package Sets is

   type Set is limited private;

   procedure Add (Item : in Item_Type;
                  To   : in out Set);

   procedure Remove (Item : in Item_Type;
                     From : in out Set);

   function Is_Member (Item   : Item_Type;
                       Of_Set : Set) return Boolean;

   generic
      with procedure Action (On_Item : in Item_Type);
   procedure Iterate (Through : in Set);

end Sets;
```

The generic package Sets defines an abstract data type Set with an (intentionally) minimal set of operations. In order to extend this set of operations in a way that enables extensions to be used for all instances of Sets, one must make these extensions generic. It is very likely that such extensions will need the generic procedure Iterate in order to implement their operations, as shown in the example below:

```
generic
   type Item_Type is private;
   with function "<" (Left, Right : Item_Type)
           return Boolean is <>;
```

```
      type Set is limited private;
      with
         generic              -- not possible in Ada 83
            with procedure Action (On_Item : in Item_Type);
         procedure Iterate (Through : in Set);
   package Set_Operations is

      function Minimum (Of_Set : Set) return Item_Type;
      function Maximum (Of_Set : Set) return Item_Type;

      procedure Unite (First_Set, Second_Set : in Set;
                       Into                  : in out Set);

      procedure Intersect (First_Set, Second_Set : in Set;
                           Into                  : in out Set);

      Empty_Set : exception;  -- raised by Minimum and Maximum

   end Set_Operations;
```

The body of the procedure Minimum, for instance, needs to instantiate the generic procedure Iterate:

```
   package body Set_Operations is

      function Minimum (Of_Set : Set) return Item_Type is

         Result      : Item_Type;
         Initialized : Boolean := False;

         procedure Update_Result (With_Item : in Item_Type) is
         begin
            if Initialized then
               if With_Item < Result then
                  Result := With_Item;
               end if;
            else
               Result := With_Item;
               Initialized := True;
            end if;
         end Update_Result;

         procedure Find_Minimum is
            new Iterate(Action => Update_Result);

      begin
         Find_Minimum(Of_Set);
         if Initialized then
            return Result;
         else
            raise Empty_Set;
         end if;
      end Minimum;

      ...

   end Set_Operations;
```

To overcome this restriction of Ada 83, Sets must be instantiated within the specification of Set_Operations, which makes Set_Operations much less general: consider having two set packages instead of one:

- Discrete_Sets (implemented as boolean arrays) for sets of discrete items,
- General_Sets (implemented as AVL trees) for sets of any items;

Set_Operations must then be duplicated for each variant of the set ADT package.

Further discussion on why this restriction greatly limits the expressive power of generics may be found in [Hosch 90].

2.1.2. Exceptions cannot be Generic Parameters

Exceptions are another kind of entity that can be exported from abstractions but cannot be parameters of a generic unit[1]. This is quite a severe restriction if one considers the set of exceptions that a subprogram may raise as being part of that subprogram's profile, as is done in [Modula-3 89a], where this set must be included in all subprogram declarations.

For instance, there is no way of importing the abstraction of a file of Items in which the Read procedure raises End_Error when an attempt is made to read past the end of the file:

```
generic
   type Item is private;
   type File_Type is limited private;
   with procedure Open (The_File : in out File_Type) is <>;
   with procedure Read (From_File : in out File_Type;
                        The_Item  : out Item) is <>;
           -- raises End_Error at the end of the file
   End_Error : exception;   -- not possible in Ada 83
```

Instead, one has to write:

```
generic
   type Item is private;
   type File_Type is limited private;
   with procedure Open (The_File : in out File_Type) is <>;
   with procedure Read (From_File : in out File_Type;
                        The_Item  : out Item) is <>;
   with function End_Of_File (On_File : File_Type)
           return Boolean is <>;
```

and ensure that End_Of_File is always checked before a call to Read; failure to do so would potentially raise an exception that can be handled only with an **others** choice within the generic.

[1] In fact, the raising of exceptions can be imported into generic units by importing a subprogram that raises a given exception, but such imported exceptions cannot be handled inside generics as there is no way of naming them.

2.1.3. Tasks cannot be Passed as a Whole

It is not possible to pass a task along with its entry profile as a generic parameter, which might be very useful when writing a generic unit whose instantiations must communicate with different tasks having the same profile.

For example,

```
generic
   task T is
      entry Put (X : in Character);
      entry Get (X : out Character);
   end T;
```

must be written

```
generic
   type T is limited private;

   with procedure Put (Into : in out T; X : in Character);
   with procedure Get (From : in out T; X : out Character);
```

which implies that T cannot be a mere task but must be a task type, hides the fact that T is a task inside the generic, loses the T.Put(...) notation and obliges the programmer to construct wrappers around all tasks that are to be passed as generic actual parameters in order to provide for Put and Get operations that match the generic formal part. Moreover, the possibility of calling the task's entries using timed entry calls is lost unless additional parameters are added to the Put and Get procedures in the generic formal part above.

The templates discussed below (2.11.6) and the proposal for constrained genericity (3.2.15) will address this shortcoming.

2.1.4. Subtypes cannot be Passed as Generic Parameters

It is impossible to pass a subtype as a generic parameter, which would sometimes be useful, as in the following example:

```
generic
   type Item_Type is private;
   type Index_Base is (<>);
   subtype Index is Index_Base range <>;   -- not possible
                                           -- in Ada 83
   type Item_Array is array (Index range <>) of Item_Type;
procedure Do_Something (On_Array : in out Item_Array);

procedure Do_Something (On_Array : in out Item_Array) is

   Last_Processed : Index_Base := Index'Pred(On_Array'First);

begin
   ...
end;
```

where Do_Something implements an algorithm that needs values of the type Index_Base that are outside the subtype Index. Here is a typical instantiation of Do_Something:

```
type Table is array (Positive range <>) of Float;

procedure Process is new Do_Something(Item_Type  => Float,
                                      Index_Base => Natural,
                                      Index      => Positive,
                                      Item_Array => Table);
```

Instead, inside Do_Something, one must use another type to index Item_Arrays, and explicitly convert it to Index for every indexed component.

This restriction is somewhat alleviated in [Ada 9X 94], where Index'Base is allowed as the type mark of an object declaration. There are, however, situations where a finer degree of control is necessary.

2.2. Staticness is Cut at Generic Boundaries

In Ada 83, it is not possible to have a generic depend on static entities, that is, entities that are static outside the generic lose that property inside the generic when they are passed as generic parameters. Following are two situations where this causes problems:

```
generic
   type Argument_Real is digits <>;
   type Result_Real is digits <>;
function Sine (X : Argument_Real) return Result_Real;
```

One would like to implement that generic function in a way such that Constraint_Error is never raised if Argument_Real and Result_Real cover the ranges of arguments and results for which it will be used. One would also like calculations to be done with a floating point type that has 'Digits >= Max(Argument_Real'Digits, Result_Real'Digits).

Unfortunately, this is impossible in the current state of the language because the 'Digits attribute of a generic formal floating point type is not static, and also because it is not possible to write static functions such as Max.

Similarly problems exist with integer types: the 'Base attribute of an integer type cannot be used to declare objects and the 'Base'First and 'Base'Last attributes of generic formal integer types are non-static.

Another situation where this causes problems is with task priorities: if a generic unit has tasks inside it (whether visible or hidden to users of the generic), then it is not possible to make the priorities of these tasks depend on a generic parameter because values given in the pragma priority must be static.

2.3. Naming of Generics does not Allow Overloading

Ada 83's overloading is only defined for subprograms and enumeration literals, but not for other entities that might take advantage of it. One such example is generics, as shown in the following (illegal Ada 83) example:

```
generic
   type Key_Type is private;
   type Item_Type is private;
package Tables is

   type Table is limited private;

   ...

   generic
      procedure Action (Key : in Key_Type);
   procedure Traversal (On_Table : in Table);

   generic
      procedure Action (Key  : in Key_Type;
                        Item : in Item_Type);
   procedure Traversal (On_Table : in Table);

end Tables;
```

The two iterators above, one for iterating over all keys in a table, and the other for iterating over all (key, item) pairs in a table, cannot have the same name. A different name must therefore be invented for each variant of an operation that must be expressed as a generic.

No solution will be proposed to this shortcoming as it does not severely impair the expressive power of the language. Furthermore, it might be hard for compilers to resolve overloading in such situations.

2.4. Generics are not Derivable Operations

Ada 83's derived type mechanism was introduced as an ability to duplicate an abstraction and its behavior, which is very useful when creating an ADT based on another, as in the following example:

```
package Integer_Lists is

   type List is limited private;

   procedure Append  (Item : in Integer; To : in out List);
   procedure Prepend (Item : in Integer; To : in out List);

   procedure Unappend  (From_List : in out List);
   procedure Unprepend (From_List : in out List);

   function First (Of_List : List) return Integer;
   function Last  (Of_List : List) return Integer;
```

```
      generic
         with procedure Action (Item : in Integer);
      procedure Traversal (The_List : in List);

   private

      type List is new integer;

   end Integer_Lists;
```

In the following package, the type List is used as the base type for creating the type Stack:

```
   with Integer_Lists;

   package Integer_Stacks is

      type Stack is limited private;

      procedure Push (Item : in Integer; On : in out Stack);
      procedure Pop  (Item : out Integer; From : in out Stack);

      function Top (Of_Stack : Stack) return Integer;

      generic
         with procedure Action (Item : in Integer);
      procedure Traversal (The_Stack : in Stack);

   private

      type Stack is new Integer_Lists.List;

   end Integer_Stacks;
```

All operations of the type List are available for the type Stack within the body of the package Integer_Stacks, except the generic procedure Traversal. This makes the bodies of Push, Pop and Top on the one hand, and that of Traversal on the other hand different: the former can be implemented with derived operations whereas the latter must use type conversions in order to call the procedure Traversal defined in Integer_Lists:

```
   package body Integer_Stacks is

      procedure Push (Item : in Integer; On : in out Stack) is
      begin
         Prepend(Item, To => On);
            -- call of derived subprogram Prepend
      end Push;

      ...

      procedure Traversal (The_Stack : in Stack) is

         procedure Traverse is
            new Integer_Lists.Traversal(Action);
               -- Traversal is not derived, so that
               -- Integer_Lists's Traversal must
               -- be used
      begin
         Traverse(The_List => Integer_Lists.List(The_Stack));
            -- type conversion required because the parameter
            -- of the instantiation Traverse is of type
            -- Integer_Lists.List
      end Traversal;

   end Integer_Stacks;
```

Ada 83's special treatment of generics makes it difficult to change a non-generic operation into a generic one when this is necessary. Such lacks of orthogonality create difficulties when composing ADTs; besides, they make the language unnecessarily hard to learn.

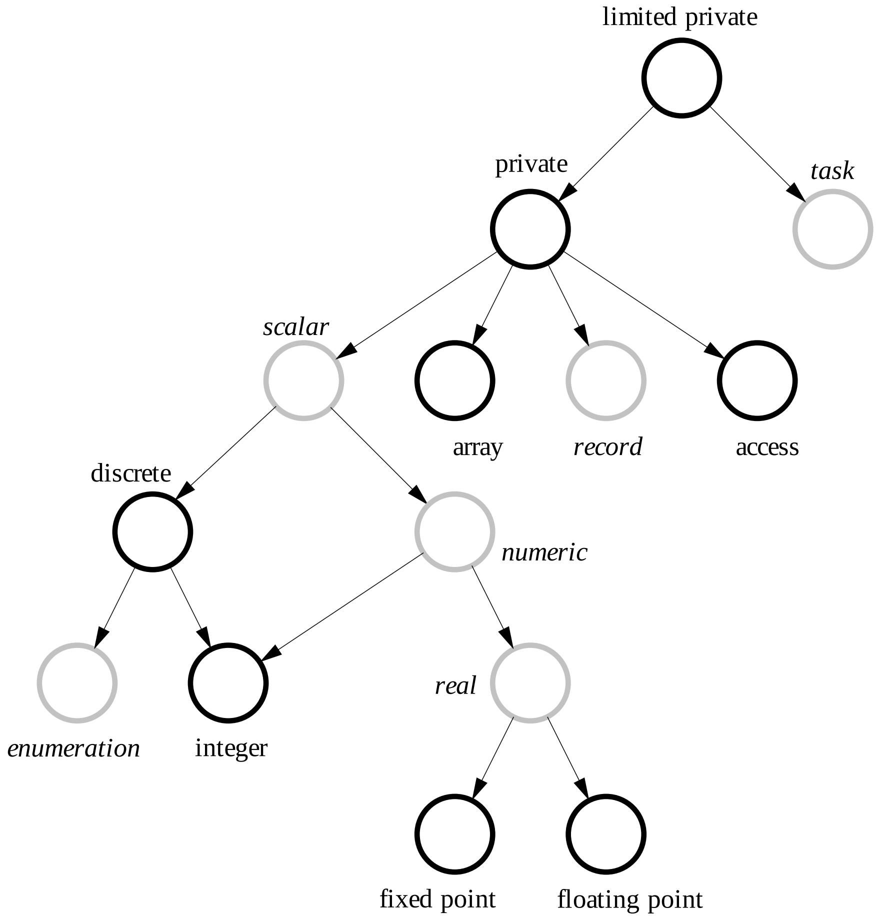
2.5. Incompleteness of the Type Class Graph

Ada 83 defines several ways of constraining generic formal types, corresponding to some type classes of the language:

- (<>); matches any enumeration or integer type and corresponds to the type class discrete.
- **range** <>; matches any integer type.
- **delta** <>; matches any fixed point type.
- **digits** <>; matches any floating point type.
- **array** (...) **of** ...; matches an array type for which both "..." also match.
- **access** ...; matches any access type for which "..." also matches.
- **private**; matches any nonlimited type.
- **limited private**; matches any type.

One can observe that these type classes are interrelated through inclusion relations such as **limited private** ⊃ **private** ⊃ (<>) ⊃ **range** <> and that the graph of this relation is acyclic.

In the following figure, the black circles represent the type classes that can be expressed as generic formal types, and the dimmed circles represent type classes that stem from the existing ones but are not part of Ada 83:



Ada's type class graph

This exhibits other interesting type classes, such as:

- **scalar**, which would contain all types that have attributes 'First and 'Last and relational operators "<", ">", "<=" and ">=".
- **real**, which would contain all fixed and floating point types. This class would be useful when it is irrelevant whether fixed or floating point numbers are used.
- **numeric**, which would contain all numeric types.
- **task**, which would contain all task types, or possibly a constrained subset thereof (for instance all task types with two entries named Seize and Release).

The great number of such classes that could be added (one could think of ordered, totally ordered, etc.) and the fact that non-predefined types might very well fit into language predefined classes (for instance a custom implementation of real, complex or rational numbers or extended precision integers) suggest that these type classes should be an extensible language mechanism[1].

2.6. Generic Parameters needed only in the Implementation

All generic formal parameters must appear in the specification of a generic unit, even if some of the parameters are needed only in the body of the generic. This may require modifying the specification in later stages of program development, namely when it is determined that additional generic parameters are required, which may not be obvious when the specification of the generic is first written.

One typical example of this situation is a generic package implementing associative tables:

```
generic
   type Key_Type is private;
   type Item_Type is private;
package Tables is

   type Table is limited private;

   procedure Insert (Key  : in Key_Type;
                     Item : in Item_Type;
                     Into : in out Table);

   procedure Remove (Key  : in Key_Type;
                     From : in out Table);

   function Search (Key : Key_Type; Within : Table)
      return Item_Type;

end Tables;
```

The above specification is complete with regard to the abstraction provided by the package. When writing the body of the package, however, one needs additional operations on Key_Type, such as a hashing function (if tables is to be implemented with hash tables) or an ordering function (if Tables is to be implemented with AVL trees).

This implies that the generic formal part of Tables must be extended, and this makes it impossible for two different implementations of Tables to share the same specification:

```
generic
   type Key_Type is private;
   type Item_Type is private;
   type Hash_Code is (<>);
   with function Hash_Of (Key : Key_Type) return Hash_Code;
```

[1] The fact that type class relations form a directed acyclic graph and not a tree complicates matters when thinking of language extensions (multiple inheritance, etc.). The extensions to be proposed later (see section 3.2) will not take these complications into account.

```
generic
   type Key_Type is private;
   type Item_Type is private;
   with function "<" (Left, Right : Key_Type)
      return Boolean;
```

Another example, below, is a generic package for complex numbers:

```
generic
   type Real is digits <>;
package Complex_Numbers is

   type Complex is
      record
         Re, Im : Real;
      end record;

   function "abs" (X : Complex) return Real;

   function Argument (X : Complex) return Real;

   function To_Complex (Radius, Argument : Real) return Complex;

   ...

end Complex_Numbers;
```

The specification of this package does not need any operations of the type Real, but its body needs to compute square roots, arc tangents, sines and cosines in order to implement "abs", Argument and To_Complex. Thus, possibly later in the development, the generic formal part of Complex_Numbers must be changed to

```
generic
   type Real is digits <>;
   with function Sqrt   (X : Real) return Real;
   with function Sin    (X : Real) return Real;
   with function Cos    (X : Real) return Real;
   with function ArcTan (X : Real) return Real;
```

These functions have nothing to do with the abstraction of complex numbers provided by the package. It would be much cleaner to be able to add these generic parameters elsewhere, perhaps to the beginning of the body of Complex_Numbers, which would raise the problem that the information found in the specification of a generic unit is not sufficient for instantiating it.

2.7. Elaboration of Library Units

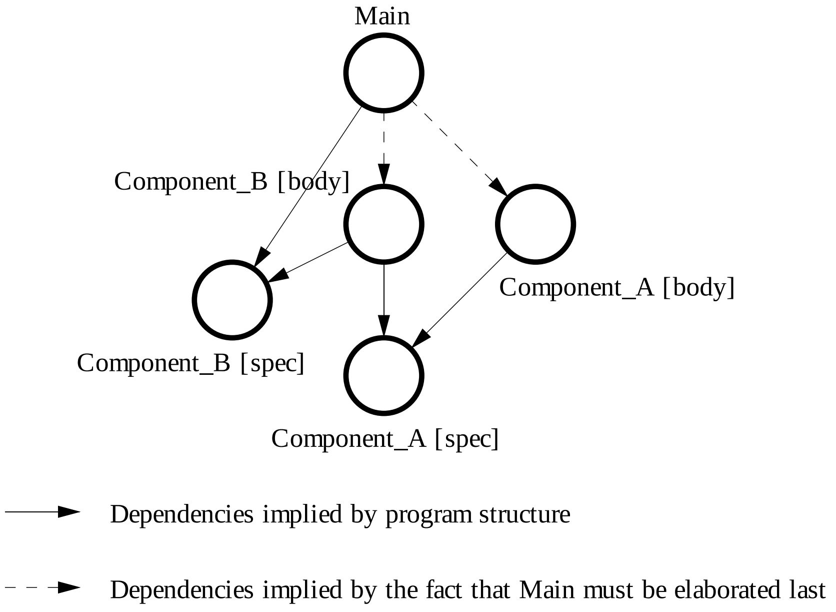
In [Ada 80], the rules governing the order of elaboration of library units left everything up to the language translator. This would be fine if it were feasible, but it isn't: during the review of the Ada 80 document, it was discovered that deciding whether or not an Ada program could be elaborated led to solving undecidable problems.

As a consequence, the approach adopted in Ada 83 left nearly all the work to the programmer (introducing pragma Elaborate). It will be shown here that an approach somewhere between Ada 80 and Ada 83 is feasible at reasonable cost, easing the maintenance and development of large programs in a substantial way.

2.7.1. Why pragma Elaborate Messes Up Programs

The following example shows why the use of pragma Elaborate is “contagious”, that is, if it is needed for one library unit, then it must be added to many others in order to achieve the desired effect. This is obviously a problem in large programs, especially when generic component libraries are used.

```
generic
package Component_A is
   ...
end;

package body Component_A is
   ...
end;

generic
package Component_B is
   ...
end;

with Component_A;
package body Component_B is
   package Instance_A is new Component_A;
end;

with Component_B;
procedure Main is
   package Instance_B is new Component_B;
begin
   ...
end Main;
```


Main
Component_B [body]
Component_A [body]
Component_B [spec]
Component_A [spec]
Dependencies implied by program structure
Dependencies implied by the fact that Main must be elaborated last

Elaboration order dependencies

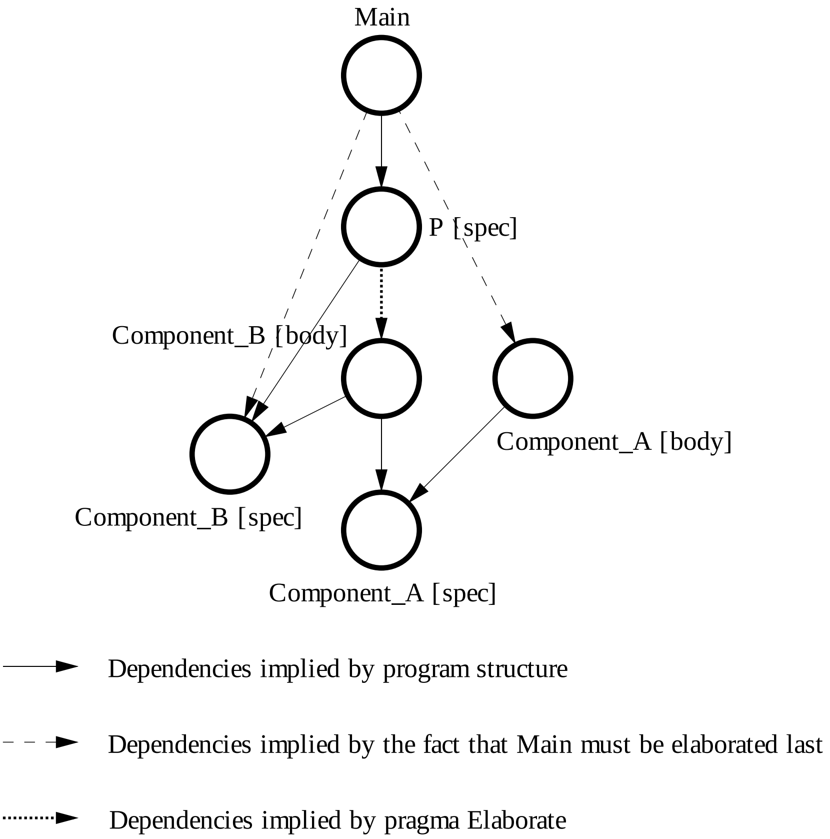
No pragma Elaborate is needed because only Main requires the bodies of Component_A and Component_B to be elaborated, and Ada 83 requires that all library units and bodies must be elaborated before the main program (Note that the dashed arrows would not exist if Main were not the main program).

Note that Ada 83 does not require the body of Component_A to be elaborated before the body of Component_B (see [Ada 83 10.5]).

Now suppose that for some reason, Instance_B is moved from Main into a library package P. This requires that the body of Component_B be elaborated before P's body (generic bodies must be elaborated before they are instantiated):

```
with Component_B;
pragma Elaborate(Component_B);
package P is
   package Instance_B is new Component_B;
end P;

with P;
procedure Main is
   ...
begin
   ...
end Main;
```

The following diagram shows the elaboration order dependencies for the modified program:


Main
P [spec]
Component_B [body]
Component_A [body]
Component_B [spec]
Component_A [spec]
Dependencies implied by program structure
Dependencies implied by the fact that Main must be elaborated last
Dependencies implied by pragma Elaborate

Elaboration order dependencies after modification

The problem is that Component_B instantiates Component_A in its body, and thus Component_A's body must be elaborated before P as well (P did not inherit Main's property of being the main program). As a consequence, a pragma Elaborate(Component_A) is required in one of two places:

- In the context clause of Component_B's body, which requires modifying Component_B, which might be part of a library of reusable components that cannot be modified.
- In the context clause of P's specification, which requires that Component_A appear in P's context clause and thus that Component_A be visible within P.

Of course, the above reasoning also holds for more complex cases, in which Component_B depends on more than one generic unit. This situation can cause major maintenance problems because:

- One must look at the bodies of many units to find out the elaboration order constraints for a given program. Some of these bodies may be inaccessible or highly subject to change, for instance if they come from a library of reusable components.
- There is no way of ensuring that a program is correct with respect to elaboration order rules (i.e. that it will always elaborate without raising Program_Error) because Ada 83 does not require implementations to perform any verifications. The only requirement is that library units be elaborated “in an order that is consistent with the partial ordering defined by the with clauses” [Ada 83 10.5(2)].

A complete solution to the elaboration order problem is given in section 4.5.

2.8. Erroneous Execution

Ada uses the concepts of *erroneous execution* and *incorrect order dependence* for describing situations where a program is incorrect in a way that cannot be checked by the compiler because it would be too hard, or even impossible. The behavior of a program in the presence of erroneous execution is left totally undefined.

In some cases, such as using the value of an uninitialized variable to index an array, the effect of the program is indeed unpredictable and not much can be done against it (except automatically initializing all variables).

In other cases, however, Ada 83 goes too far: for instance the problem of detecting incorrect order dependencies is known to be undecidable. Moreover, there are many cases where the order of execution of a construct is unimportant, as in the following example:

```
Random_Numbers : constant array (1 .. 100) of Float :=
                 (others => Random.Uniform);
```

where Random.Uniform is a function that returns a pseudo-random number. [Ada 83 1.6(8-10)] allows (but does not require) the compiler to generate code that raises Program_Error because the effect of the aggregate depends on the order of evaluation of its components.

This introduces an unnecessary portability problem, as well as a loss in ease of expression, without providing much advantage.

The rules on incorrect order dependencies are in terms of “the effect of the execution” of constructs of the language (see [Ada 83 1.6(9)]). But this term is defined nowhere and is quite ambiguous, as the following example illustrates:

```
Last : Boolean := False;

function Alternate return Boolean is
   -- alternatively returns True and False
begin
   Last := not Last;
   return Last;
end;

A : array (Boolean) of Integer := (5, 7);

Sum : Integer := A(Alternate) + A(Alternate);
```

The order of evaluation of the left and right arguments of "+" are not defined by the language, so that Sum can be initialized with any of the two values A(True) + A(False) or A(False) + A(True). As "+" is commutative, the result will always be 12. But is the fact that "+" is called with (Left => 7, Right => 5) or (Left => 5, Right => 7) part of the “the effect of the execution” of the program ?

2.9. Lack of Specification in the LRM

A few characteristics of the Ada language are left unspecified in the reference manual, leaving too much freedom to implementations as to how they are dealt with, thus reducing the portability of programs. Of course, some characteristics such as the order of evaluation of expressions or the task scheduling policy must be left unspecified in order for the language to be able to take advantage of particularities of a variety of implementation techniques and operating systems. However, I feel that Ada 83 lacks specification in some critical aspects, some of which are discussed here.

- Ada 83 allows, but does not require, implementations to provide a garbage collector. The predefined generic procedure Unchecked_Deallocation is required by the standard, but is not required to do any actual storage reclamation: the only required behavior is to it set its actual parameter to **null**.

  This situation makes it impossible to develop portable applications that allocate storage dynamically. There should at least be a requirement that either garbage collection is provided, or that Unchecked_Deallocation actually reclaims the storage.

- The memory allocation scheme for objects whose storage size depends on a discriminant is not specified by the language. Consider the following example:

  ```
  type Matrix is array (Positive range <>,
                        Positive range <>) of Float;

  type Square_Matrix (Dimension : Natural := 0) is
     record
        Element : Matrix(1 .. Dimension, 1 .. Dimension);
     end record;
  ```

  implementations are free to either always allocate the maximum size for objects of type Square_Matrix (which will almost certainly raise Program_Error in this case), or to dynamically allocate storage depending on the value of the discriminant (which requires reallocation when the discriminant changes).

  This freedom left to implementors is very harmful for portability.

Both problems shown above are at a level of specification that the Ada 83 reference manual misses: the language definition is at a higher level of abstraction, at which such things as a memory leakage have no meaning.

Ada 9X alleviates this problem by giving the programmer control over the storage allocation/deallocation mechanisms through *storage pools* [Ada 9X 94 13.11].

2.10. Incorrect Link Between Types and Operations

Because of the existence of derived types, Ada 83 had to define, for each type, which were the operations “owned” by that type and to be made available as operations of the derived type. Unfortunately, this was done only for subprograms and specific type features defined by predefined type constructors (e.g. enumeration literals and attributes), but not for other operations defined for the parent type, such as generic units or operations defined in nested packages.

Moreover, the language is polluted with rules such as “a type cannot be derived within the package specification that defines it” [Ada 83 3.4(15) and 7.4.1(4)]. This suggests, as pointed out in [Hilfinger 83], that derived types are fine for specifying operations of numeric types through rewriting rules, but not for user-defined types, which usually have a more complex set of operations.

Besides, some types are often linked to subprograms that operate on them in a way that is not intended by the programmer. The following examples show situations where Ada 83's rules on derived types are inappropriate:

```
package Graph_Handler is
   type Vertex is private;
   type Arc is private;

   function New_Vertex return Vertex;
   function New_Arc (Initial, Final : Vertex) return Arc;
```

```
      function Initial (Of_Arc : Arc) return Vertex;
      function Final   (Of_Arc : Arc) return Vertex;

      type Graph is limited private;

      procedure Add (The_Vertex : in Vertex; To : in out Graph);
      procedure Add (The_Arc    : in Arc;    To : in out Graph);
   end Graph_Handler;
```

In this example, the types Vertex and Arc are equally “important” to users of the package, but there is no way of deriving them together in order to create a new pair of related types with a corresponding set of operations. Writing

```
   type Vertex_2 is new Graph_Handler.Vertex;
   type Arc_2    is new Graph_Handler.Arc;
```

yields the following set of operations, which is useless in most cases:

```
   function New_Vertex return Vertex_2;
   function New_Arc (Initial, Final : Vertex_2) return Arc;
   function New_Arc (Initial, Final : Vertex)   return Arc_2;

   function Initial (Of_Arc : Arc) return Vertex_2;
   function Final   (Of_Arc : Arc) return Vertex_2;

   function Initial (Of_Arc : Arc_2) return Vertex;
   function Final   (Of_Arc : Arc_2) return Vertex;
```

Whereas the desired set of operations, i.e. the one that would be useful in most cases, is:

```
   function New_Vertex return Vertex_2;
   function New_Arc (Initial, Final : Vertex_2) return Arc_2;

   function Initial (Of_Arc : Arc_2) return Vertex_2;
   function Final   (Of_Arc : Arc_2) return Vertex_2;
```

The following example is based on the predefined package Text_IO and shows what happens when a type that does not represent the main abstraction of a package is derived:

```
   package Text_IO is
      type File_Type is limited private;
      type File_Mode is (In_File, Out_File);

      type Count is range 0 .. <Implementation Defined>;
      subtype Positive_Count is Count range 1 .. Count'Last;
      Unbounded : constant Count := 0;

      ...

      procedure Set_Line_Length (File : in File_Type; To : in Count);
      procedure Set_Line_Length (To : in Count);

      function Line_Length (File : in File_Type) return Count;
      function Line_Length return Count;
      procedure New_Line (File    : in File_Type;
                          Spacing : in Positive_Count := 1);
      procedure New_Line (Spacing : in Positive_Count := 1);

      ...

   end Text_IO;
```

After the derived type declaration:

```
      type New_Count is new Text_IO.Count;
```

the following derived subprograms are implicitly declared, which can be quite surprising to a programmer who just wanted a type with the same range as Text_IO.Count:

```
      procedure Set_Line_Length (File : in Text_IO.File_Type;
                                 To : in New_Count);
      procedure Set_Line_Length (To : in New_Count);

      function Line_Length (File : in Text_IO.File_Type)
         return New_Count;
      function Line_Length return New_Count;

      subtype New_Positive_Count is New_Count   -- this declaration
         range 1 .. New_Count'Last;             -- is not visible

      procedure New_Line (File    : in Text_IO.File_Type;
                          Spacing : in New_Positive_Count := 1);
      procedure New_Line (Spacing : in New_Positive_Count := 1);
```

The problem here is that the type File_Type, which is the principal abstraction exported by Text_IO, cannot be distinguished from “secondary” types such as File_Mode and Count.

This may be viewed as a more general problem: packages, unlike class constructs found in most object-oriented programming languages, *export* types but *are* not types. This sometimes makes it difficult to identify the main abstraction exported by a package and adds somewhat unnatural language rules that link packages with types defined in their visible part.

In [9X Mapping 91b], a proposal is made for adding a new kind of visibility for operations of a type defined within a package specification: such operations are said to be *primitively visible* and may be used as if they were directly visible in any scope where the type on which they operate is visible, whether directly or as a selected component. I think, however, that this concept further complicates Ada's visibility rules, and that a simpler construct resembling Eiffel's classes [Meyer 88] would be easier both in terms of language specification and usage.

2.11. Problems with the Implementation of ADTs

Ada's packages with private and limited private types seems to be a good mechanism for expressing ADTs. But, as will be shown in this section, the desired degree of composability and cohesion between a type and its operations cannot always be attained.

2.11.1. Private versus Limited Private is too Harsh

In some circumstances, the distinction between private and limited private types is too harsh, i.e. it is difficult to decide whether some ADT should be made limited or not, as can be seen in the following example:

```
package Varying_Text is

   type Text (Max_Length : Natural) is private;

   ...

private

   type Text (Max_Length : Natural) is
      record
         Current_Length : Natural := 0;
         Value          : String(1 .. Max_Length);
      end record;

end Varying_Text;
```

Assignment for type Text makes perfect sense here (except that it will raise Constraint_Error if it is attempted on two objects with different Max_Length values), but equality does not because the value of an object T of type text is made up (logically) only of T.Value(1 .. T.Current_Length); even worse, using "=" is erroneous if some components of Value have not been initialized (which typically will hold for T.Value(T.Current_Length + 1 .. T.Value'Last))[1].

As "=" cannot be redefined for non-limited types in Ada 83, there are three possibilities to work around the problem:

- Make the type Text limited, which is almost unacceptable in this case (varying length strings are often needed as subcomponents of records, which in turn would become limited).
- Give the type Text a default initial value, and explicitly pad the unused part of the Value component for each operation, so that "=" at least works for values of type Text with the same Max_Length:

```
Padding : constant Character := ASCII.Nul;

type Text (Max_Length : Natural) is
   record
      Current_Length : Natural := 0;
      Value          : String(1 .. Max_Length) :=
                       (others => Padding);
   end record;
```

[1] Ada 83 does not consider copying undefined subcomponents of composite types erroneous; an execution only becomes erroneous when undefined subcomponents are read (for instance as part of an equality test).

```
      procedure Assign (Object : out Text; Value : in String) is
      begin
         Object.Value(1 .. Value'Length) := Value;
         Object.Value(Value'Length + 1 .. Object.Value'Last) :=
            (others => Padding);
         Object.Current_Length := Value'Length;
      end;
```

Note that implementing Text as an access type (beside the memory leakage problems) would not work, again because "=" cannot be redefined.

Another example where the restrictions on the redefinition of equality get into the way is when an ADT is implemented with pointers, for instance in the following specification for extended precision integers implemented as pointers to arrays of integers:

```
   package Large_Integers is

      type Large_Integer is private;

      Zero, One : constant Large_Integer;

      function "+" (R : Large_Integer) return Large_Integer;
      function "-" (R : Large_Integer) return Large_Integer;

      function "+"   (L, R : Large_Integer) return Large_Integer;
      function "-"   (L, R : Large_Integer) return Large_Integer;
      function "*"   (L, R : Large_Integer) return Large_Integer;
      function "/"   (L, R : Large_Integer) return Large_Integer;
      function "mod" (L, R : Large_Integer) return Large_Integer;
      function "rem" (L, R : Large_Integer) return Large_Integer;

   private

      type Slice is range 0 .. 2**15 - 1;

      type Slice_Array is array (Natural range <>) of Slice;

      type Large_Integer_Rec (Size : Natural) is
         record
            Sign   : Integer range -1 .. 1;
            Slices : Slice_Array(0 .. Size);
         end record;

      type Large_Integer is access Large_Integer_Rec;

      Zero : constant Large_Integer :=
             new Large_Integer_Rec'(Size   => 0,
                                    Sign   => 0,
                                    Slices => (0 => 0));

      One  : constant Large_Integer :=
             new Large_Integer_Rec'(Size   => 0,
                                    Sign   => 1,
                                    Slices => (0 => 1));

   end Large_Integers;
```

In the presence of a garbage collector (or of an adequate finalization construct), the preceding specification works perfectly except for equality, which cannot be redefined to compare the denoted objects instead of the access values. In Ada 83, one is obliged to make the type Large_Integer limited, which greatly reduces its ease of use.

Note that implementing Large_Integer as a Large_Integer_Rec (making the discriminant Size visible to users) would not work because "=" would take Size into account when comparing, which is clearly not intended here. Furthermore, it would oblige users of the package to always devise an upper bound on the size of the Large_Integers to be manipulated.

I think that giving "=" a special status over "<", "<=", etc. (except perhaps for the automatic redefinition of "/=") is not appropriate as it often gets into the way when creating abstractions. The fact that it is possible to define "<" and ">" such that A < B is not equivalent to B > A can be just as dangerous, but Ada 83 does not prohibit such inconsistencies (for instance, there could be a rule that makes ">", "<=" and ">=" available as soon as "=" and "<" are defined).

2.11.2. The Reemergence of Hidden Predefined Operations

Predefined operations have some kind of a privileged status that makes them available in some situations (namely within generic instantiations) even if they are hidden within the declarative part containing the type that implicitly declares them. This problem is a consequence of the rules on generic formal types, which state that predefined operations of a generic formal type are available within the generic unit (see [Ada 83 12.1.2]).

As an example, consider the following package specification for modular arithmetic:

```
package Modular_Arithmetic is

   Modulus : constant := 12;

   type Number is range 0 .. Modulus - 1;

   function "+" (Right : Number) return Number;
   function "-" (Right : Number) return Number;

   function "+"   (Left, Right : Number) return Number;
   function "-"   (Left, Right : Number) return Number;
   function "*"   (Left, Right : Number) return Number;
   function "/"   (Left, Right : Number) return Number;
   function "mod" (Left, Right : Number) return Number;
   function "rem" (Left, Right : Number) return Number;

   function "**" (Left : Number; Right : Integer)
      return Number;

end Modular_Arithmetic;
```

The function declarations for "+", "-", "*", "/", "mod", "rem" and "**" hide the predefined operators with the same names that are implicitly declared after the type declaration of Number.

One may think that these predefined operators are made invisible by these overloaded declarations, but this is not true; they will reappear in instances of generic units having

```
type Int is range <>;
```

in their generic formal part.

If such a generic unit is instantiated with Int => Modular_Arithmetic.Number, then inside the instance, only the predefined versions of the arithmetic operators on the type Number are visible, not the ones redefined in Modular_Arithmetic. Avoiding this requires writing

```
generic
   type Int is range <>;

   with function "+" (Right : Int) return Int is <>;
   with function "-" (Right : Int) return Int is <>;

   with function "+"   (Left, Right : Int) return Int is <>;
   with function "-"   (Left, Right : Int) return Int is <>;
   with function "*"   (Left, Right : Int) return Int is <>;
   with function "/"   (Left, Right : Int) return Int is <>;
   with function "mod" (Left, Right : Int) return Int is <>;
   with function "rem" (Left, Right : Int) return Int is <>;

   with function "**" (Left : Int; Right : Integer)
           return Int is <>;
```

Packages are Ada 83's way of linking a type with its operations; I think that the fact that these links are not automatically carried along with the type in all situations is a mistake in Ada 83's design.

A solution to this problem is presented in section 4.2.2.

2.11.3. Overloading "=" would not hide predefined "="

One may think that the problems presented in 2.11.1 may be solved by allowing "=" to be overloaded for private types, but the problem is not as simple as it looks, because of the way predefined operations are linked to generic actual types (see 2.11.2).

The problem presented in 2.11.2 is not very severe in the case of arithmetic operators, but it is when transposed to the equality operator, if overloading of "=" for nonlimited types were permitted:

```
generic
   type ADT is private;
```

and

```
generic
   type ADT is private;
   with function "=" (Left, Right : ADT) return Boolean is <>;
```

would have different meanings if the generic actual type for ADT had an overloaded "=" operator. The former would use the predefined "=" inside the generic whereas the latter would use the overloaded "=" operator.

This is a severe problem because it would enable generic units to violate abstractions by using an inadequate "=" operator. It strongly suggests that the link between generic actual types and their operations should be revised.

2.11.4. The Lack of Automatic Perpetuation of Operations

In Ada 83, predefined and user-defined operations are not treated the same way with respect to composition. For instance, if a record or array type has components of a limited type for which "=" has been redefined, then the composite type does not automatically inherit an equality test operation that compares matching components, as is the case with non-limited types[1].

Example of "=" not being perpetuated:

```
package P is

   type T is limited private;
   function "=" (Left, Right : T) return Boolean;

   type R is           -- no "=" is available on type R,
      record           -- even though it is defined for all
         I : Integer;  -- of R's components
         C : T;
      end record;

private

   ...

end P;
```

A solution to this problem is presented in section 4.2.3.

2.11.5. Generics Should Allow the Creation of New Type Constructors

Ada's array, record and access type declarations may be viewed as *type constructors*, taking one or more types as “arguments” and “returning” a new, composite, type.

Array, record and access are *primitive* type constructors in that they cannot be made up from others (because they introduce new notations such as selected and indexed components, and also because they need to be implemented

[1] Ada 83 (probably deliberately) takes an approach such that whenever predefined equality is available for a type, then a bitwise comparison can be used to compute it. This is valid even for real operands.

efficiently). The set constructor, on the other hand, is not primitive because it can be constructed with boolean arrays or linked lists (using access types).

A generic stack package, for example, may be viewed as a user-defined type constructor taking one type as argument and returning a new type with operations Push and Pop:

```
generic
   type Item_Type is private;
package Stacks is
   type Stack is limited private;

   procedure Push (Item : in Item_Type; On : in out Stack);
   procedure Pop (From : in out Stack; Item : out Item_Type);

   Underflow : exception;
end Stacks;
```

Unfortunately, such user-defined type constructors are not as versatile as the predefined ones:

- They yield packages instead of types. Operations are exported from the package instead of being linked to the type (Ada 83 attempts to repair this with derived subprograms). This problem is addressed in section 3.2.11, where generic classes are presented.
- They cannot redefine notations such as indexed or selected components, or **for** loop iteration schemes. A partial solution to this shortcoming is given in section 3.2.16 on iterators[1].

2.11.6. The Loose Link Between a Type and Its Operations

Through packages and private types, Ada 83 provides a good mechanism for writing encapsulations, but types and subprograms are still expressed as separate entities, so that there is no way of naming an abstraction (a type together with its operations) as a whole.

It is often necessary, when writing generic components, to have several types and their operations as generic parameters. The generic formal part then looks like an unstructured list of types and subprograms, causing redundancy between package specifications and generic parameter specifications: if some abstraction needs to be changed, the change must be carried out in its package specification as well as in all generics that have this abstraction as a parameter.

[1] Full uniform reference, which would allow to define the meaning of selected components of user-defined types, is not proposed.

For instance, the following generic procedure creates a list of all prime numbers in a given interval; it is designed to be usable with any representation of integers and also any representation of a list of integers:

```
generic
   type Number is private;
   Zero, One : in Number;
   with function "+"   (Left, Right : Number) return Number is <>;
   with function "-"   (Left, Right : Number) return Number is <>;
   with function "*"   (Left, Right : Number) return Number is <>;
   with function "/"   (Left, Right : Number) return Number is <>;
   with function "mod" (Left, Right : Number) return Number is <>;

   type List_Of_Numbers is limited private;
   with procedure Insert_At_Tail (Item : in Number;
                                  Into : in out List_Of_Numbers);
   with procedure Destroy (The_List : in out List_Of_Numbers);
procedure List_Prime_Numbers (From, To : in Number;
                              Into     : in out List_Of_Numbers);
```

Unfortunately, there is no way of grouping the generic formal subprograms that operate on the type Number and those that operate on the type List_Of_Numbers into separate constructs. It is also impossible to write a generic parameter that is "a type with two values Zero and One along with operations "+", "-", "*", "/" and "mod"".

What one would like here is some sort of *template*[1] that would group types and operations together, for instance:

```
template Integer_Number is
   Zero, One : constant Integer_Number;

   function "+"   (Left, Right : Integer_Number)
      return Integer_Number;
   function "-"   (Left, Right : Integer_Number)
      return Integer_Number;

   function "*"   (Left, Right : Integer_Number)
      return Integer_Number;
   function "/"   (Left, Right : Integer_Number)
      return Integer_Number;
   function "mod" (Left, Right : Integer_Number)
      return Integer_Number;
end Integer_Number;

generic
   type Item_Type is private;
template List is
   procedure Insert_At_Tail (Item : in Item_Type;
                             Into : in out List);
   procedure Insert_At_Head (Item : in Item_Type;
                             Into : in out List);
```

[1] The templates presented here are not proposed as a language change, but rather as an example feature that solves the problem at hand in this section. The proposed solution will be in terms of abstract classes and constrained genericity (see 3.2.14 and 3.2.15).

```
      function Head (Of_List : List) return Item_Type;
      function Tail (Of_List : List) return Item_Type;

      function Length (Of_List : List) return Natural;
      function Is_Empty (The_List : List) return Boolean;

      procedure Destroy (The_List : in out List);
   end List;
```

The procedure specification List_Prime_Numbers could then be written as:

```
   generic
      type Number is Integer_Number;
      type List_Of_Numbers is new List(Number);
   procedure List_Prime_Numbers (From, To : in Number;
                                 Into     : in out List_Of_Numbers);
```

This change has several advantages:

- Abstractions are always imported as a whole: this means that inside the body of List_Prime_Numbers, all operations defined in the templates Integer_Number and List are always available for Numbers and List_Of_Numbers. If the algorithm that implements List_Prime_Numbers suddenly needs the operation Insert_At_Head, no change needs to be made to List_Prime_Numbers' specification or to any of its instances.

- Less duplicated information: all the features available for objects of type Number need not be repeated in all generic formal parts that need types conforming to that template. Thus if one decides that it is necessary to add some feature to a template (for example a unary "-" operator for Number), then this needs only to be done in one place, namely in the template.

- More abstract way of speaking of types as classes, like deferred classes in Eiffel [Meyer 88].

Note that this template facility exists in a very restricted form in Ada 83: language predefined type constructors may be used to constrain generic actual types to be of some type class, thereby making predefined operations specific to that type class available inside the generic:

- `(<>);`
- **`range`** `<>;`
- **`delta`** `<>;`
- **`digits`** `<>;`
- **`array`** `(...)` **`of`** `...;`
- **`access`** `...;`

A generic formal part such as

```
   generic
      type Int is range <>;
```

closely resembles the above Integer_Number template except for attributes and integer literals, and

```
generic
   type Index is (<>);
   type Item_Type is private;
   type Item_Array is array (Index range <>) of Item_Type;
```

is very much like

```
generic
   type Index is (<>);
   type Item_Type is private;
   type Item_Array is new Array_Type(Index, Item_Type);
```

where Array_Type is defined as

```
generic
   type Index_Type is (<>);
   type Item_Type is private;
template Array_Type is
   type Array (First, Last : Index_Type) is private;

   function Component (Of_Array : Array;
                       At_Index : Index_Type) return Item_Type;

   procedure Set_Component (Of_Array : in out Array;
                            At_Index : in Index_Type;
                            Value    : in Item_Type);
end Array_Type;
```

if only functionality is taken into account. Of course the indexed component notation is not available as with language-defined array types.

The functionality of the templates presented above is given by generic abstract classes (sections 3.2.11 and 3.2.14) and constrained genericity (section 3.2.15).

2.11.7. Problems with Composing ADTs

Ada 83 has severe shortcomings with respect to the composability of ADTs: *composite ADTs* (i.e. ADTs whose values are made up of the values of their components, such as sets, stacks, arrays, tables, etc.) are generally written so that they take generic formal private types as parameters and export limited private types, which makes it impossible to compose them directly because limited types cannot match generic formal private types.

There are two possible approaches for composition in this case:

- Have all composite ADTs in two flavors: one taking private types as parameters, and the other taking limited types as generic parameters, e.g.

  ```
  generic
     type Item_Type is private;
  package Stacks_Of_Static_Items is ...
  ```

  and

```
generic
   type Item_Type is limited private;
   with procedure Assign (Object : in out Item_Type;
                          Value  : in Item_Type);
   with procedure Destroy (Object : in out Item_Type);
package Stacks_Of_Dynamic_Items is ...
```

- Instantiate the outer ADT with an access type pointing to the inner one, e.g.

```
generic
   type Item_Type is private;
package Sets is

   type Set is limited private;

   ...

end Sets;

generic
   type Item_Type is private;
package Stacks is

   type Stack is limited private;

   ...

end Stacks;

package Integer_Sets is new Sets(Item_Type => Integer);

type Integer_Set_Access is access Integer_Sets.Set;

package Integer_Set_Stacks is
   new Stacks(Item_Type => Integer_Set_Access);
```

None of these approaches is satisfactory: the former requires maintaining the two flavors, generally implemented with the same algorithms but with minor differences scattered all over the code. The latter requires allocating new objects and, worse, always deallocating them when they are not needed anymore. It also introduces the risk of unwanted aliasing between objects because references, not values, are manipulated by the ADT.

An extensive discussion of the first approach can be found in [Genillard 89].

2.11.8. Semantics of Parameter Modes

When a type is defined using access types, the modes of subprogram formal parameters apply only to the first level of the underlying structure of the type, as illustrated by the following example:

```
type Cell;

type Link is access Cell;
type Cell is
   record
      Value : Item_Type;
      Next  : Link;
   end record;

type List is
   record
      First : Link;
      Card  : Natural := 0;
   end record;

procedure Change_First (New_Value : in Item_Type;
                        In_List   : in List) is   -- note mode in
begin
   In_List.First.Value := New_Value;
end Change_First;
```

The procedure Change_First, above, can change In_List.First.Value, but not In_List.First or In_List.Card, in spite of the fact that In_List has mode **in**. It is, of course, very bad style to write this procedure with mode **in**, but it shows a (slight) shortcoming of the language: parameter modes applied to types defined using access types are often documentary instead of conveying their real meaning.

The reason why it is bad style to give In_List the mode **in** is that if the implementation of List is changed to a version without access types, then it would have to be changed to **in out**. Moreover, for the sake of composition of ADTs, it is useful to have matching specifications for different implementations of the same ADT. Otherwise, generics such as the following will not be usable to their intended extent:

```
generic
   type Item_Type is private;

   type Item_List is limited private;
   with procedure Change_First (New_Value : in Item_Type;
                                In_List   : in out Item_List);
```

By the way, Text_IO, as defined in [Ada 83], violates this principle in that it forces an implementation to use an access type somewhere in the definition of File_Type, because the mode of File is **in** in many operations that change the underlying file:

```
procedure Put (File : in File_Type; Item : in Character);
```

No solution will be proposed to this problem because it would change Ada 83 so much that it would not be Ada anymore. Furthermore, it would make too many Ada 83 programs illegal.

2.12. Parameter Passing Mechanisms

Ada 83 requires that parameters of scalar and access types be passed by copy, and allows parameters of other types to be passed either by copy or by reference. These rules can cause problems in some situations:

- Copy and reference mechanisms do not interact the same way with exceptions, as the following example illustrates:

  ```
  type Link is access ...;

  type List is
     record
        Length : Natural := 0;
        First  : Link;
     end record;

  procedure Destroy (The_List : in out List);
     -- Destroys The_List, reclaiming any storage
     -- allocated to it.

  procedure Build_List (New_List  : in out List;
                        From_File : in String) is
  begin
     Destroy(New_List);   -- first make sure that the list is empty
     ...   -- read From_File, inserting
           -- elements into New_List
  exception
     when others =>
        Destroy(New_List);
        raise;
  end Build_List;

  ...

  L : List;

  ...

  Build_List(L, From_File => "datafile");
  ```

  If L is passed to Build_List by reference, then L remains in a consistent state even if Build_List returns with an exception (the call to Destroy(New_List) directly operates on the object L).

  On the other hand, if L is passed by copy and Build_List returns with an exception, then L has the value it had before the call. This means that L.First points to deallocated storage if L was non-empty before the call.

  This example shows that it is very hard to write code that does not depend on parameter passing mechanisms; this observation is further substantiated in [Gonzales 90]. Moreover, if the representation of some type is changed from an access type to a record type, all code using that type must be reviewed for dependence on the parameter passing mechanism, which is a serious problem for the maintainability of programs.

- Within a generic instance, the constraints on parameter passing mechanisms are imported along with generic actual types. This can make code sharing between instantiations of the same generic very hard.

Ada 9X partially solves this problem by requiring that tagged types be passed by reference.

In order to reduce the differences between the copy and reference mechanisms, maybe it would be a good idea to force copy-out of **in out** and **out** parameters when returning from a subprogram with an exception (but this would certainly cause too many compatibility problems for existing programs).

2.13. The Flat Name Space of the Program Library

All library units in a program are (logically) part of a single declarative region, namely that of the predefined package Standard. Moreover, only package, subprogram and task *bodies* may be made separate units. This may be a problem when developing large programs, in which separate programmer teams create library units which must later be merged into a single library, with possible name clashes.

In [Booch 87], a partitioning of large programs into so called subsystems is suggested. The nearest Ada 83 construct for expressing subsystems is nested packages, which satisfy the goal of showing the logical decomposition of the system, but are not workable because a change in any subpackage makes all units depending on the library package containing that subpackage obsolete; besides, with clauses cannot apply to subpackages, but only to library packages and it is thus impossible to have selective visibility on a subpackage of a subsystem.

No proposal will be made to correct this problem, as the child library units of [Ada 9X 94] are a perfect solution.

2.14. Limitations in Tasking

Given a set of tasks that must communicate, it is often hard to determine which synchronization constraints will exist between them at the same time as their interfaces are designed. As Ada 83 only allows selective waits on a set of accept statements, one is sometimes obliged to move entries from one task to another, inverting the parameter modes. This often leads to changing the interface of tasks in a way that makes them less readable and may be very expensive in terms of reprogramming if tasks are part of the specification of a much used package.

As an example, suppose that two tasks provide streams of items that must be blended by a third task into a new stream:

```
task type Stream_Provider is
   ...
   entry Get (Item : out Item_Type);
end Stream_Provider;

Stream_1, Stream_2 : Stream_Provider;

task Blender is
   entry Get (Item : out Item_Type);
end Blender;

task body Blender is
   Next : Item_Type;
begin
   loop
      select
         Stream_1.Get(Item => Next);  -- not possible
      or                              -- in Ada 83
         Stream_2.Get(Item => Next);
      or
         terminate;
      end select;

      select
         accept Get (Item : out Item_Type) do
            Item := Next;
         end Get;
      or
         terminate;
      end select;
   end loop;
end Blender;
```

The first **select** statement above is not legal Ada 83: select alternatives must be **accept** statements. This makes it impossible to wait for the first item deliverable by either Stream_1 or Stream_2.

To overcome this restriction, Stream_Provider must be modified in order to deliver its items by calling another task instead of delivering them through an entry:

```
task type Stream_Provider;

Stream_1, Stream_2 : Stream_Provider;

task Blender is
   entry Get (Item : out Item_Type);
   entry Put (Item : in Item_Type);
end Blender;

task body Stream_Provider is
begin
   loop
      ...
      Blender.Put(...);
      ...
   end loop;
end Stream_Provider;
```

```
task body Blender is
   Next : Item_Type;
begin
   loop
      select
         accept Put (Item : in Item_Type) do
            Next := Item;
         end;
      or
         terminate;
      end select;

      select
         accept Get (Item : out Item_Type) do
            Item := Next;
         end Get;
      or
         terminate;
      end select;
   end loop;
end Blender;
```

In order to be able to wait for multiple providers, the Get entry calls in Blender were changed to an accept statement and a corresponding entry was added to Blender's specification. Symmetrically, in the Stream_Provider, the accept statement that delivered items was changed into an entry call.

The amount of change is quite small in this example, but it could become huge if Stream_Providers are used at many places in the program to be modified.

In the current state of the language, the design phase of a tasking program is unnecessarily constrained: for any pair of tasks that must communicate, one must decide very early which one gets the accept statement and which one gets the entry call. This implies that the arity of relations between tasks must be known beforehand, which is seldom the case.

Allowing selective waits on all tasking events (i.e. accept statements, entry calls, delays and terminate alternatives) would make the Ada rendezvous completely symmetric except for the naming of tasks (the caller knows the identity of the called task but the reverse is not true), which is aesthetically pleasing and has been proved to be feasible (see [Riese 89] and references of this paper). It would also remove the distinction between the selective wait and the timed and conditional entry calls.

One argument against this generalization is that it complicates the implementation of Ada on distributed systems. I think that this is not too severe because Ada's model of intertask communication nearly implies shared memory (because access to global variables from tasks is permitted, and access values can be passed among tasks). [Ada 9X 94] further substantiates this model in that a completely separate proposal is made for distributed systems.

2.15. Exceptions

Ada 83's exceptions are often too restrictive when used to describe error situations in some abstractions:

- They cannot be grouped together, for instance, there is no way of referring to all exceptions in Text_IO without explicitly enumerating them. The only way of grouping exceptions is an **others** choice in an exception handler, which denotes all possible exceptions declared in the program, including Constraint_Error and Storage_Error, which deserve special treatment most of the time.

- Subsystems whose error situations are expressed with exceptions are hard to extend: if new exceptions are added, then all code using that subsystem must be reviewed in order to add that new exception in some handlers. As an example, suppose Text_IO must be extended in order to include a new exception Non_ASCII_Character in order to accommodate for 8-bit characters. Then, all places within a program where all read errors are handled must be modified in order to include Non_ASCII_Character:

```
declare

   Ch : Character;

begin
   ...
   Text_IO.Get(Ch);
   ...
exception
   when Text_IO.Data_Error |      -- this handler must catch
        Text_IO.End_Error |       -- all exceptions that can
        Text_IO.Mode_Error |      -- be raised by Text_IO.Get
        Text_IO.Device_Error =>
      Handle_Read_Error;
end;
```

- Some systems, such as the ISO standard GKS Ada binding or the POSIX 1003.5 Ada binding, have error situations that are hard to map onto exceptions and as a result, an approach like the following is taken:

```
package Binding is

   procedure P (X : in Arg);

   Error : exception;

   type Error_Code is ...;

   function Last_Error return Error_Code;

end Binding;
```

where users of Binding are supposed to call Last_Error when handling Error. This approach does either not work at all if Binding is called from different tasks, or implementing Binding requires access to the Ada run-time system in order to make Last_Error local to each task. Moreover, there is no language-level link between the exception Error and the function Last_Error. There

> should be some mechanism for linking exceptions with data types representing additional information pertaining to the exception[1].

Extensions to the exception mechanism are proposed in section 4.3.

2.16. Control of Direct Visibility (use clauses)

Ada 83's **use** clause has been the object of much debate (dangerous, counterintuitive, making programs difficult to read, etc.). On the contrary, I think that its semantics have been very carefully designed, in particular the fact that already visible declarations cannot be hidden by a **use** clause (see [Ada 83 8.4]).

Potentially visible declarations with the same name are not actually made directly visible unless all of them are overloadable entities (i.e. subprograms or enumerals). This leads to a style of programming with **use** clauses and a convention that non-overloadable entities are always written as expanded names, as shown in the following example:

```
package Sets is
   type Collection is private;
      -- duplicate items not cumulated

   procedure Add (Item : in Integer; To : in out Collection);
end Sets;

package Bags is
   type Collection is private;
      -- duplicate items allowed

   procedure Add (Item : in Integer; To : in out Collection);
end Bags;

with Sets, Bags;
use Sets, Bags;
procedure P is

   S : Sets.Collection;   -- selected component notation
   B : Bags.Collection;   -- necessary for Collection

begin
   Add(Item => 4, To => S);   -- overloadable entities
   Add(Item => 3, To => B);   -- are directly visible
end;
```

This technique is quite maintenance-safe because adding a **use** clause for another package with an operation named Add cannot invalidate the code already written (Of course, this does not hold for parameterless procedures, which are not really overloadable entities anyway).

[1] [Modula-3 89a] has parameterized exceptions which, together with inheritance, offer a mechanism for linking data to exceptional situations in a way that enables extension of the exception's associated data type at the same time the abstraction containing the exception is extended.

Unfortunately, this convention cannot be enforced using constructs of Ada 83[1]. For instance, there is no way of ensuring that a type is never referred to by its direct name. The above program would also be rendered illegal by adding a use clause for a package containing a non-overloadable entity named Add, but this should seldom happen if names are well chosen.

A solution to this problem is presented in section 4.4.

[1] Some authors recommend never using **use** and replace it with renaming declarations when direct visibility on a subprogram is desired. This is a very dangerous practice when combined with the cut and paste features of modern text editors. Can you rapidly find the error in the following program fragment ?

```
function "+" (L, R : Some_Package.Some_Type) return Some_Package.Some_Type
   renames Some_Package."+";
function "-" (L, R : Some_Package.Some_Type) return Some_Package.Some_Type
   renames Some_Package."+";
```

Furthermore, such practice can make programs difficult to maintain because renaming declarations such as the one above will almost surely not be removed by maintainers once they are not needed anymore and will uselessly clutter up the program.

3. Higher Level Extensions

This chapter describes the proposed extensions that go further than just repairing "language bugs". Their main motivation is to enhance the expressiveness of the language at a high level of abstraction. The central ideas behind these extensions are:

- Uniformizing the concepts of package and task as encapsulations, and introduction of package types as a parallel to task types.
- Introduction of objects and classes with the same structure as tasks and packages, together with inheritance and polymorphism. Inheritance is achieved through subtyping, which I think is the most natural way of providing polymorphism.
- Abstract classes and constrained genericity, which offer the functionality of the templates presented in section 2.11.6, as well as type constructors based on generics (see section 2.11.5).

3.1. Requirements for writing ADTs

This section will attempt to define a set of language-level requirements for the expression of ADTs. These requirements, together with the existing Ada 83 language definition and previously proposed extensions such as [Cohen 89], will be used for the design of the proposed extensions.

3.1.1. Encapsulation

There must be a mechanism for expressing encapsulation. This means that it must be possible to wrap together a type, its operations and related exceptions into a single, named construct which can then be used in many places in a program, such that a change to the definition of the encapsulation is automatically carried over to all its occurrences in the program.

Ada 83's support for encapsulation through private and limited private types is quite good. It has some weaknesses, though:

- Encapsulations must be defined inside the visible part of some package, which is also supposed to contain the operations of the encapsulation, but this is not required. The definition of the type of the encapsulation is too much separated from the definition of its operations. There should be a language construct for defining encapsulations with a tighter link between the encapsulation's underlying type and its operations.

3.1.2. Data Abstraction

Data abstraction must be supported, which means that one must be able to write an ADT without having full knowledge of what data objects it is going to deal with. This requires that program entities be parameterizable with respect to types or encapsulations, which suggests that there must be a symmetry between what can be exported from an encapsulation and what can be required from a parameter of a parameterized program unit.

Ada 83's generics are almost satisfactory for that purpose, although not completely, as shown in section 2.11.

3.1.3. Composability

ADTs should be fully composable in a straightforward manner, that is, for instance, if Stack_ADT and Set_ADT are two generic ADTs, it should be easy to create a stack of sets of integers.

As in the previous section, this requires that anything that can be exported from an encapsulation can serve as an actual parameter of another, parameterized, encapsulation.

This requirement is not entirely fulfilled by Ada 83, as explained in section 2.11.7.

3.1.4. Extensibility

Once an abstraction is available, it must be possible to extend it, which means either adding operations and data components to it, or slightly modifying the behavior of some operations without changing or duplicating the original abstraction.

It is very important not to textually duplicate the original abstraction when extending it because source code duplication can mean bug duplication. This can have adverse effects on the reuse of components (because the extended components will not benefit from enhancements in the reused ones) and in early stages of the development (when the components to be extended are not yet fully implemented and/or debugged).

3.1.5. Multiple Implementations

[Booch 87] describes a taxonomy of reusable software components where each abstraction has a set of different implementations, each of which is expressed in Ada 83 as a package specification and body. Unfortunately, the fact that the specifications of two packages implementing the same abstraction are (nearly) the same is not captured by the language and must be expressed through source code duplication.

There should be a way of writing components following such a taxonomy such that all components implementing the same abstraction share the same specification, but this must not be too strict a conformance: for instance, it must be possible to add discriminants, generic parameters or additional operations applicable only to some variants.

The example below shows Ada 83 specifications of different variants of a Set ADT:

- The first one, Sets, is the most general. It makes as few assumptions as possible on the type of items it can hold and on its underlying implementation (so few, in fact, that it is impossible to implement efficiently). The operations it supplies are applicable to all kinds of Sets:

```
generic
   type Item_Type is private;
package Sets is

   type Set is limited private;

   procedure Add (Item : in Item_Type;
                  To   : in out Set);

   procedure Remove (Item : in Item_Type;
                     From : in out Set);

   function Is_Member (Item   : Item_Type;
                       Of_Set : Set) return Boolean;

   generic
      with procedure Action (On_Item : in Item_Type);
   procedure Iterate (Through : in Set);

end Sets;
```

- Discrete_Sets makes the assumption that the items it can hold are of a discrete type, which makes it possible for the Set type to be non-limited (as can be seen in the private part). Object of this Set type can thus be easily manipulated, which makes the declaration of operators and constants useful. Unfortunately, the fact that Discrete_Sets is a specialization of Sets cannot be expressed in Ada 83 and must be implemented through source code duplication.

```
generic
   type Item_Type is (<>);
package Discrete_Sets is

   type Set is private;

   ...   -- Same as in Sets

   function "+" (Left, Right : Set) return Set;
   function "-" (Left, Right : Set) return Set;
   function "*" (Left, Right : Set) return Set;

   Full_Set, Empty_Set : constant Set;

private

   type Set is array (Item_Type) of Boolean;
   Full_Set  : constant Set := (others => True);
   Empty_Set : constant Set := (others => False);

end Discrete_Sets;
```

- Hash_Table_Sets is another specialization of Sets, requiring a hash function for items to be inserted into the sets. As for Discrete_Sets, operators can be provided because the type can be non-limited if the hash table is implemented as an array[1]. A discriminant specifying the maximum number of elements has also been added, along with the function Is_Full:

```
generic
   type Item_Type is private;

   type Hash_Code is range <>;
   with function Hash_Of (Item : Item_Type) return Hash_Code;
package Hash_Table_Sets is

   type Set (Max_Size : Natural) is private;

   ...   -- Same as in Discrete_Sets, but
         -- without the constant Full_Set

   function Is_Full (The_Set : Set) return Boolean;

end Hash_Table_Sets;
```

- AVL_Tree_Sets is exactly the same as Sets except that it requires the items it can contain to be totally ordered by a relation to be supplied as a generic parameter:

```
generic
   type Item_Type is private;
   with function "<" (Left, Right : Item_Type)
           return Boolean is <>;
package AVL_Tree_Sets is

   type Set is limited private;

   ...   -- Same as in Sets

end AVL_Tree_Sets;
```

It is unfortunate that the specification of the base package Sets cannot be shared by its specializations. This sharing would bring many advantages:

- Operation specifications added to the base package (or a package anywhere in a specialization tree) are automatically added to all its specializations. In the Sets example, the function Is_Empty has been forgotten and it must be added to all four variants.
- By looking only at one ADT specification, a user can know which operations he will find in all of its variants. He can thus write a prototype using the base ADT and later change variants for efficiency or other reasons (for instance, in the Sets example, when an item type cannot be totally ordered anymore).

[1] Except that "=" would certainly not work, but let's just leave this problem aside for now, or have a look at sections 2.11.1 and 4.2.1 to 4.2.3.

- Some specialization may make very little changes to their parent, so that not all operation bodies need to be recoded in the presence of adequate language constructs.

Of course, the above benefits can be achieved through disciplined programming methodologies, but a language-level enforcement of specialization dependencies is much better.

Abstract class types, to be presented below (3.2.14), support multiple implementations of the same specification in a way that makes it possible to decide on the implementation of an object as late as run time.

3.1.6. Concurrent ADTs

One important dimension of the taxonomy presented in [Booch 87] is concurrency, i.e. the fact that objects of some class may be accessed concurrently by multiple tasks in a way that guarantees internal consistency of the objects.

Unfortunately, Ada 83 suffers a few shortcomings with respect to the expression of such concurrent ADTs:

- The task construct is not powerful enough to fully encapsulate the implementation of an ADT[1] and one often has to resort to implementing the ADT outside of a task and adding a hidden semaphore[2] to protect its access, which is a very error-prone technique.
- It is too awkward to transform a non-concurrent ADT into a concurrent one: it requires textually duplicating the specification of the ADT and either surrounding all its operations with calls to a semaphore or embedding the ADT into a task and making all operations of the ADT call entries of that task.

[1] Two severe limitations are that it is hard to write iterators (entries cannot be generic) and that accept statements may not appear within a task's local procedures, which makes it unnecessarily hard to encapsulate recursive structures into a task type.

[2] There are two solutions for this:

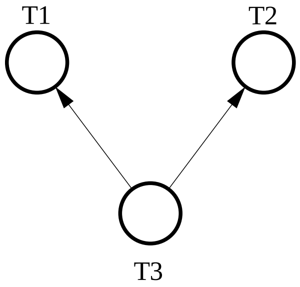
- with visible Seize and Release operations, requiring users to always ensure that they lock and unlock objects before and after using them;
- with no visible locking operations, where the ADT locks the object before operating on it and unlocks it afterwards.

Both solutions have severe pitfalls: the former requires users to be very careful always to unlock objects after using them, especially when exception can be raised, and the latter can easily lead to unexpected deadlock situations when using operations that involve more than one object of the ADT.

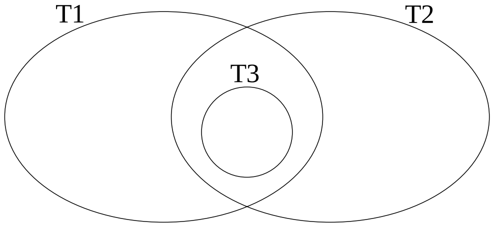
3.1.7. Multiple Inheritance

Multiple inheritance will not be part of the object oriented extensions to be proposed, for the following reasons:

- There is no generally accepted semantic definition of multiple inheritance: should it be a strict Cartesian product, duplicating common ancestors, or should common ancestors be inherited only once ?
- Multiple inheritance makes programming languages more complex because, among other things, of the need for name clash resolution mechanisms. Furthermore, many languages, such as Modula-3, do not provide multiple inheritance either.
- In the proposed solution, inheritance is based on subtyping. As pointed out in [Cohen 89], this is a problem for multiple inheritance, as shown in the following diagram, where T3 inherits from both T1 and T2:



If T1 and T2 are distinct types, then the relations T3 ⊆ T1 and T3 ⊆ T2 (implied by Ada's subtype model) imply that either T3 is empty or that T1 and T2 have a non-empty intersection, which is in contradiction with Ada's type model.



If T1 and T2 are subtypes of the same type (i.e. the classes T1 and T2 have a common ancestor), then multiple inheritance would be feasible. But then, the programmer would always have to think beforehand, when designing inheritance hierarchies, which classes are likely to be combined later using multiple inheritance. Because of these restrictions, I decided to drop multiple inheritance altogether.

3.2. Proposed Solution

3.2.1. Uniformizing the Concepts of Package and Task

In Ada 83 packages and tasks are constructs that both can export “procedural” entities (procedures for packages, entries for tasks). At an abstract level, procedures and entries may be viewed as the same concept. Unfortunately, this is where the similarity ends; many features are proper to packages or to tasks, among which:

- Packages can export functions, types, exceptions, objects, tasks, generic units.
- Tasks can be types (thus providing the <object>.<operation> dotted notation).

The aim here is to make packages and tasks exactly the same at the interface level, except for concurrency, so that both may be used to encapsulate implementations of ADTs.

There are good reasons to enhance the expressive power of tasks and task types. One of them may be found in [Gonzales 90], where it is shown that approaches to the implementation of ADTs using tasks as mere semaphores, as in [Booch 87], are inappropriate because they lead to erroneous programs. It should thus be possible to encapsulate the state of ADTs inside tasks, instead of just using tasking to achieve synchronization.

3.2.2. Objects as References versus Value Containers

A major question when designing the object-oriented extensions proposed here is whether objects should be references or value containers. Both approaches have their advantages and drawbacks.

In favor of objects as references:

- They are better in terms of efficiency when ADTs are composed; for instance, when building Stacks of Sets of Integers by composing ADTs, if objects are references then inserting a set into a stack will not copy the set's whole data structure but just a reference (of course causing aliasing but I think that this is seldom a problem). On the other hand, performance problems have been observed with composed ADTs when deep copy semantics were used.
- Object data is copied only when really needed (presumably through an explicit call to a method named Clone returning a new object), avoiding duplication of potentially large memory areas when this is not necessary. Moreover, shallow copying of objects that are value containers is often not enough and may introduce hidden aliases or force programmers to declare objects as limited types.

- Objects must be explicitly created when new ones are needed, through a construct like an allocator, thus making it clear when new objects of some class are created.
- Objects as references approach CLU's very clean definition of what is an object and what is a variable (objects exist in the ether and variables are names that permit to access them). (See [CLU 81]).
- Objects as value containers are more difficult to implement: their storage size is not always the same because it may depend on discriminants. Moreover, sharing generic instances among different types can be difficult when their representations differ.

Against objects as references:

- After an assignment A := B, where A and B are variables, A and B are in fact names for the same object and thus all modifications to A do the same to B. This may not always be a disadvantage but would require users to become used to this style of programming: manipulating references instead of values.
- In Ada 83, everything that is not explicitly of an access type behaves like a scalar type: assignment copies values and not references. Objects as references would break this rule and force programmers to learn a new style. Furthermore, many existing generic components would not work anymore if instantiated with object types that are references.
- Ada 83's task creation and termination semantics treat task objects as actual objects: tasks are activated when they are declared and must be terminated before their scope is left. Making objects references, and thus probably making task types non-limited, would be contrary to Ada 83's semantics and would require extensive modifications to existing programs.
- In some circumstances, objects that are subcomponents of other objects should be explicitly non-shareable, as in this example (expressed in a free syntax):

  ```
  class Truck is
     Its_Engine : Engine;
        -- should not be shareable with other Trucks

     Its_Driver : Person;
        -- may be shared with other Trucks

     ...

  end Truck;
  ```

  such constraints, however, may be expressed through initial values of objects: new Trucks will always be initialized with a new instance of Engine, which would certainly not be the right behavior for Its_Driver.

Of course, objects as references can be constructed using objects as value containers and pointers (access types), but the reverse isn't true. Because Ada 83 objects[1] are value containers except where access types are used explicitly, the extensions proposed here will stick to that principle in order to preserve the uniformity of the language. In many situations, though, reference semantics will be necessary and constructs to support this will be given.

3.2.3. Class Types

Package and task types will be made as similar as possible, concurrency remaining the only difference at the specification level.

The syntax for class type declarations is[2]

```
class_type_declaration :==
   package_type_declaration
 | task_type_declaration

package_type_declaration ::=
  [limited] package type identifier [discriminant_part] [is
     {class_specific_declaration}
  [private
     {class_specific_declaration}]
   end [package_type_simple_name]];

task_type_declaration ::=
  [limited] task type identifier [discriminant_part] [is
     {class_specific_declaration}
  [private
     {class_specific_declaration}]
   end [task_type_simple_name]];

class_specific_declaration ::=
   constant_object_declaration
 | subprogram_declaration
 | procedure new [formal_part]
 | entry_declaration
 | generic_subprogram_declaration
 | deferred_constant_declaration
```

Example of a package type:

```
package type Window is
   X_Position, Y_Position : Integer;
   X_Size, Y_Size : Integer;

   procedure Hide;
   procedure Show;
   procedure Redraw;

private
   Is_Hidden : Boolean := False;
end Window;
```

[1] in the sense of [Ada 83 3.2].

[2] The usual “a type declaration does not define a type but really a first named subtype” stuff applies here, of course, but I will leave these details aside from now on and happily mix types and first named subtypes, clarifying only when necessary.

```
package body Window is

   procedure Hide is
   begin
      Is_Hidden := True;
   end Hide;

   procedure Show is
   begin
      Is_Hidden := False;
      Redraw;
   end Show;

   procedure Redraw is
   begin
      ...
   end Redraw;

end Window;

...

W1, W2 : Window;      -- objects can be declared just as
                      -- with regular types

W2.X_Size := W1.X_Size;    -- each Window has its own
                           -- copy of variables declared
                           -- in the package type

W1.Hide;      -- procedures of package objects are
W2.Redraw;    -- invoked like Ada 83 task entries

if W1.Is_Hidden then      -- illegal: Is_Hidden is not visible
   ...                    -- outside the package type because it
end if;                   -- is declared in the private part.
```

Package and task types or subtypes will hereafter collectively be called *class types* or *class subtypes* in describing situations where either of them, but not ordinary types, may be used. Objects of class types will be called *class objects*.

The reserved word **entry** is made obsolete and is replaced by **procedure**. Entry declarations are not allowed in package types (in order to discourage the use of **entry** instead of **procedure**). For upward compatibility, entries are still allowed in task types, but they are equivalent to procedures. For package types, subprogram bodies are provided as usual, whereas for task types, they are provided in the form of accept statements. This implies that a task can have functions, and thus that accept statements can return values[1].

From now on, any reference to the terms subprogram, procedure or function in the context of class types will also apply to entries if the corresponding class type is a task type.

[1] Note that Ada 83's definition of the return statement [Ada 83 5.8] is adequate for these extensions: a return statement is defined to leave the innermost subprogram or accept statement.

A task type is always limited, whether or not the reserved word **limited** appears in its declaration. **limited** is allowed in a task type declaration for symmetry with package types, but it is not mandatory in order to preserve compatibility with Ada 83.

In Ada 83 types and exceptions are always *statically namable*, that is, they always belong to a construct that is either a subprogram, a package or a task body, but never to an object in which it is visible as a subcomponent. It is thus impossible to create types dynamically or to have situations where dynamic type checking is necessary. Note that this restriction is maintained by the above definition of class types[1].

Class type bodies are provided with the same syntax as package or task bodies. The restriction imposed upon the declarations that can appear within a class type's specification are not necessary in their bodies, as is the case with Ada 83's task types and task bodies. This is because a non-statically namable declaration such as a type or an exception can never “escape” from a package or task body, and thus there is no dynamic creation of types or exceptions.

Within the declarative region associated with a class type, the usual visibility rules of Ada 83 apply:

- Discriminants are directly visible.
- Declarations occurring within the class type's declarative region are directly visible in that region from the point of their declaration.

Class types have value semantics, just like Ada 83 records:

- Predefined equality is defined as the equality of matching components (including components of the private part and body of the class type), as in [Ada 83 4.5.2]. Assignment copies matching components, or raises Constraint_Error if the expression on the right-hand side of the assignment does not belong to the subtype of the target class object. Assignment and predefined equality are not available for objects of limited class types.
- The selected component notation can be used on objects of class types for accessing the discriminants and the declarative items in the visible part of the class type declaration[2].

The private part and body of class types are not visible to the outside world, just like for regular packages. But within the body of a class type, the declarative items of the private parts and bodies of other objects of the same type are visible as selected components. Section 3.2.6 shows an example where this is useful.

[1] Except for generic subprogram declarations within class types, which will be explained later in section 3.2.12.

[2] As a result of the definition of the syntax for names [Ada 83 4.1], it is not possible to call the procedure **new** (see 3.2.5) of a class object other than by creating a new class object.

Within the specification and body of class types, the new reserved word **self** denotes the current class object, which means:

- within the body of a subprogram declared within a package type specification or body: the object that owns the subprogram that is being called, i.e. the object denoted by the prefix of the selected component used to call the subprogram;
- in the specification or body of a class type: the object that is being elaborated or executed.

Ada 83's convention of using the task type's name for this purpose cannot be applied here because that name may be needed to refer to the type instead of the current instance, which is not possible in Ada 83[1].

Use clauses are not permitted to name objects of package types (there is no way of getting direct visibility of items that can be accessed as selected components of class objects).

The class specific declaration "**procedure new** [formal_part]" is explained in section 3.2.5.

3.2.4. Class Types and Access Types

As stated earlier, reference semantics will often be useful for class types. In order to better support this, some extensions are proposed to Ada 83's access types.

In Ada 83, there is no way to subtype an access type so that the subtype can only denote some subtype of the type denoted by the parent access type. The following extension is proposed in order to overcome this limitation:

```
constraint ::=
   range_constraint
 | floating_point_constraint
 | fixed_point_constraint
 | index_constraint
 | discriminant_constraint
 | access_constraint

access_constraint ::= access subtype_indication
```

The subtype indication in an access constraint must be a subtype of the subtype denoted by the access type to which the constraint applies.

Example:

```
type Integer_Access is access Integer;

subtype Positive_Access is Integer_Access access Positive;
```

[1] In order to preserve compatibility, the possibility of a context-sensitive interpretation of the name of the class type was considered, but rejected because of too much complication and confusion for programmers.

When using class types, access types are often so important that many class types will need an associated access type. In order to avoid the repetitive declaration of such access types (or subtypes), each class type has its own predefined access type (or subtype), denoted by T'Access where T is a class type.

If U is a subclass of T, then U'Access is a subtype of T'Access (see section 3.2.7 for a definition of subclasses).

In order to support this new construct, the syntax of a type mark given in [Ada 83 3.3.2] must be extended:

```
type_mark ::= type_name | subtype_name | access_attribute
```

This feature has a great advantage with generics that have formal class types: they won't need to have an explicit formal access type denoting the formal class type.

The access attribute may be useful even for types other than class types, but this will not be discussed in this document.

3.2.5. Class Objects

Class types may be parameterized in two ways:

- with discriminants (specified as discriminant_part in the above syntax), which behave like any other Ada 83 discriminated type, i.e. each different set of discriminant values is a different subtype of the class type;
- with *class parameters*, by declaring a possibly overloaded procedure **new** in the visible part of the class type; this allows users of class types to give objects some initial data at their creation. Unlike discriminants, class parameters are not part of the value of a class object.

The simple names of formal parameters of a procedure **new** and those of discriminants of the class type it belongs to must not overlap. A procedure **new** may only have parameters of mode **in** (implicitly or explicitly, as for functions in Ada 83).

Objects of class types are created either by an object declaration with a subtype indication of the form

```
class_type_mark [(class_creation_parameter {,
                  class_creation_parameter})]

class_creation_parameter ::= discriminant_association
                           | parameter_association
```

or by an allocator of the form

```
class_allocator ::=
   new class_type_mark['(class_creation_parameter {,
                         class_creation_parameter})]
```

Each object created by such an object declaration or allocator has its own copy of the items declared in the declarative region of the class type and its body.

An object of a class type is constrained by the class subtype appearing in its declaration or allocation. As a result, a value belonging to one of its strict class subtypes cannot be assigned to it, or Constraint_Error will be raised. Access types denoting class types must be used when this is a problem.

Values of non-limited package types can be written as qualified expressions of the following form:

```
class_type_mark['(class_creation_parameter {,
                  class_creation_parameter})]
```

Each class creation parameter is either a discriminant of the class type or a parameter of a **new** procedure declared in the visible part of the class type. If one discriminant appears in the list, then all of them must be present; discriminants may be omitted only if they have default values. Usual rules apply for positional/named associations; if positional associations are used, then discriminants are before **new** procedure parameters.

If a **new** procedure is declared in the visible part of the class type, then class creation parameters must be supplied for it at object creation. It is possible to declare more than one **new** procedure in a class type, thus allowing several different profiles for the creation of objects. In this case, the usual overloading rules are used on the class allocator parameters that are not discriminants. Note that class creation parameters can be made optional either by declaring a parameterless **new** procedure or by using default parameter values.

The elaboration of a class object creation proceeds as follows: first, the class creation parameters are evaluated in some order that is not defined by the language, except that discriminants are evaluated first. Then the new class object is created and

- for package types, the declarative region of the package type is elaborated and its sequence of statements (if one is present) is executed. Then, if the type has a **new** procedure, the one determined by the overloading rules is called;
- for task types, the declarative region of the task type specification is elaborated, but the task object is activated later, as defined in [Ada 83 9.3]. If the type has a **new** procedure, then the call of the corresponding task entry is postponed until after the activation of the task object (note that this implies a rendez-vous);

finally, in the case of an allocator, an access value denoting the new object is returned. If an exception is raised before the allocator returns, then it is propagated at the place where the allocator is evaluated[1].

3.2.6. Subprograms as Methods

The term *method* will be used hereafter to denote subprograms occurring immediately within the declarative region of a class type. This choice seems

[1] Except for task types, where Tasking_Error is raised instead, as defined in [Ada 83 9.3(7)].

adequate as it matches the way this term is used in the object-oriented community.

Methods always have an implicit parameter whose type is the class type in which they appear. This parameter is accessed with the reserved word **self** and is of mode **in out**. It would be possible to add syntax to specify the mode of method parameters, but in view of the problems exposed in section 2.11.8, I did not consider such a feature as very useful.

The following example shows how visibility into the private part and body of other objects of the same class type (as defined in section 3.2.3) can be useful:

```
type Item_Type is range 1 .. 10;

package type Set is

   procedure Add (Item : in Item_Type);
   procedure Remove (Item : in Item_Type);

   function Intersection (Other : Set) return Set;

private

   Contents : array (Item_Type) of Boolean := (others => False);

end Set;

package body Set is

   ...

   function Intersection (Other : Set) return Set is

      Result : Set;

   begin
      for I in Item_Type loop
         Result.Contents(I) := Contents(I) and Other.Contents(I);
      end loop;
      return Result;
   end Intersection;

end Set;
```

Note that **self** has not been used in this example, although it could: the statement within the loop could be written:

```
Result.Contents(I) := self.Contents(I) and Other.Contents(I);
```

Section 3.2.7, where inheritance is introduced, shows where **self** really becomes useful.

The above example could also be written with Contents declared in the body of the package type Set.

3.2.7. Subtyping provides Inheritance

Class types can be extended or specialized through subtyping, which corresponds to inheritance in other object-oriented languages. Subtyping was chosen over another mechanism (e.g. type derivation as in Ada 9X) because Ada 83's rules on types and subtypes are quite appropriate for class inheritance hierarchies, as shown in [Cohen 89].

Following is the syntax for class subtype declarations:

```
package_subtype_declaration ::=
  [limited] package subtype identifier [discriminant_part] is
      parent_class_subtype_name
  [with
     {class_specific_declaration}
  [private
     {class_specific_declaration}]
   end [package_subtype_simple_name]];

task_subtype_declaration ::=
  [limited] task subtype identifier [discriminant_part] is
      parent_class_subtype_name
  [with
     {class_specific_declaration}
  [private
     {class_specific_declaration}]
   end [task_subtype_simple_name]];
```

The *parent_class_subtype_*name after the reserved word **is** specifies the single parent class (multiple inheritance is not available).

The declarative part between **with** and **end** is used to add new items to the heir class type, or override existing items of the parent class. Items declared in the visible part of the parent class type are automatically inherited and implicitly available from objects of the heir class type, unless they are overridden.

In particular, no modification is made to the subtypes of the formal parameters or function results of the inherited methods, even when the type of these formal parameters is the class type being subtyped (see section 3.3.2 for an explanation). The following example illustrates this:

```
package type Parent is
   procedure P (Other : in Parent);
   function F (Other : Parent) return Parent;
end Parent;

package subtype Heir is Parent;
   -- inherited operations
   -- procedure P (Other : in Parent);
   -- function F (Other : Parent) return Parent;
```

Class subtype bodies are provided with the same syntax as package or task bodies. The visibility rules within class subtypes are the following:

- Within the visible part of a class subtype, the visible part of the parent class and, recursively, the visible parts of all ancestors, are visible.
- Within the private part and body of a class subtype, the visible and private parts of the parent class and, recursively, the visible and private parts of all

ancestors, are visible. Note that the body of the parent class is not visible within the body of the heir.

This enables a very selective export of declarations appearing in classes: it is possible to make them available everywhere (when declared in the visible part), only to heirs of the class (when declared in the private part) or only to the class itself (when declared in the body).

Also, the visible and private parts of a class subtype are considered an extension of the visible and private parts of the ancestors. This means that a declaration within the visible or private part of a subclass type must not be a homograph of a declaration in the visible and private parts of its ancestors, unless both declarations are subprogram declarations (i.e. it is allowed to override the subtype profile of methods). As an example, the following subtype of the class Parent shown earlier is legal:

```
package subtype Heir is Parent with
   procedure P (Other : in Heir);
   function F (Other : Parent) return Heir;
end Heir;
```

Note that Heir.P and Heir.F have the same parameter and result type profile as, respectively, Parent.P and Parent.F, because Heir is a subtype of Parent, not a different type.

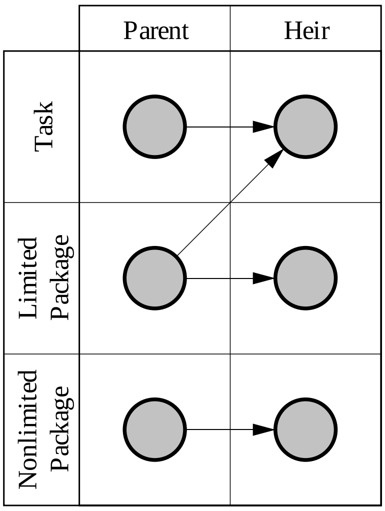
Discriminants of the ancestor classes are also inherited, and the discriminants appearing in the class subtype declaration are added to the ancestors' discriminants. The names of new discriminants must not clash with those of the ancestors'.

The new attribute *'Subtype*, when applied to an object of a class type, yields the actual subtype of the object. It can be used to constrain formal parameters and results of methods, and also to declare objects of class types, as in the following example:

```
package type Part is
   function Clone return self'Subtype;
end Part;

Last_Serial_Number : Natural := 0;

package body Part is

   Serial_Number  : Positive;
   X              : Integer;

   function Clone return self'Subtype is
      Result : self'Subtype;
   begin
      Last_Serial_Number := Last_Serial_Number + 1;
      Result.Serial_Number := Last_Serial_Number;
      Result.X := X;
      return Result;
   end Clone;
end Part;
```

Then, in all heirs of Part, Clone will always return a value of the same subtype as the object of which it is called. Only heirs of Part that add data to their representation will have to write a new body for Clone, as shown in section 3.2.9.

The following diagram shows which transitions are possible from a parent to an heir class type with respect to “taskness” and “limitedness”:



Possible Inheritance Transitions

A limited class type is not allowed to inherit from a nonlimited one because limitedness is a property of types, not of subtypes. This limitation may be quite constraining (for instance, it is impossible to make a nonlimited package type concurrent, or to add a limited component to a nonlimited class type), but removing it would lead to very complicated rules in order to forbid copying objects of limited subtypes.

To overcome this limitation, it is recommended that class types for which limited heirs are foreseen be made limited, and to use access to class types (reference semantics).

A package type is not allowed to inherit from a task type because this could potentially violate the mutual exclusion that is expected from a task type (and of its heirs).

Following are the rules that apply when the class subtype being defined is a task type:

- If the parent type is a package type and if no method is redefined in the heir task type, and no task body is given, then, for a package type of the form

  ```
  limited package type T is
     procedure P;
     function F return Some_Type;
     ...
  end T;
  ```

  for which an heir of the form

  ```
  limited task subtype U is T;
  ```

is created, a default task body of the following form is automatically provided:

```
task body U is
begin
   loop
      select
         accept P do
            T'(self).P;
         end P;
      or
         accept F return Some_Type do
            return T'(self).F;
         end F;
      or
         ...
      or
         terminate;
      end select;
   end loop;
end U;
```

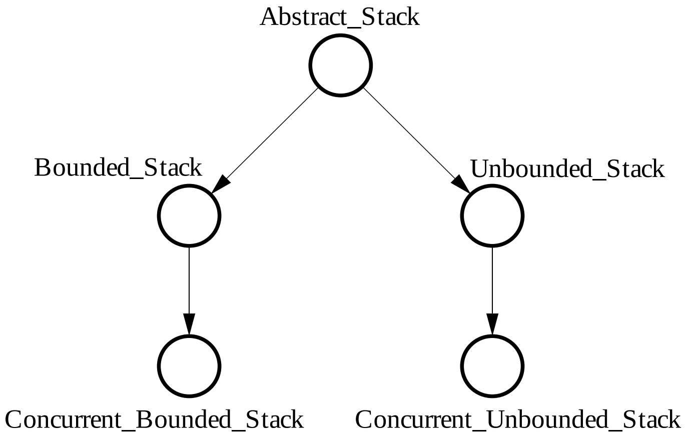
The rules behind the notation T'(**self**).P are explained in section 3.2.9. The previous rule is designed to make it as easy as possible to create a concurrent version of an ADT when no particular synchronization or entry call ordering is necessary, which should often be the case.

- If the parent type is already a task type, then the whole body of the heir task type must be rewritten, and the bodies of operations of the parent are not inherited; their specifications, however, are inherited in the normal way[1].

[1] Further study of this subject might come up with a better solution for reusing task bodies when inheriting. For instance, it might be possible to inherit everything from the parent except the accept statements, so that the heir would only have to redefine the bodies of the accept statements, while preserving the general control structure of the parent task. Such a solution is complicated because accept statements can be nested within declare blocks, whereas procedures exported from a package (type) are always at the same level.

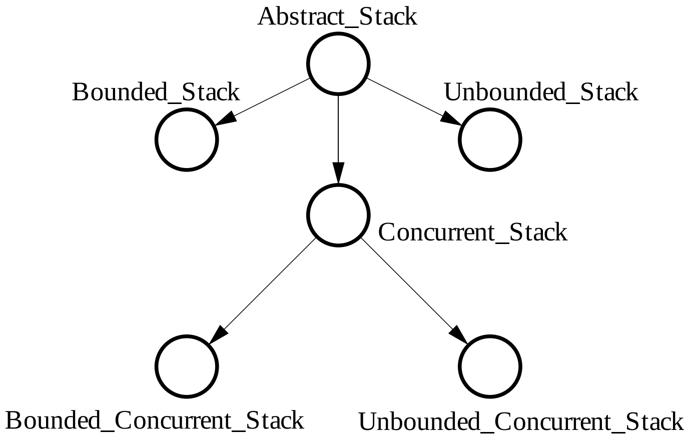
A Note on Concurrent Classes

The above rules suggest that “taskness” should be added as late as possible when creating inheritance hierarchies, e.g. an inheritance tree like the following:



Adding “taskness” latest

is to be preferred to one like this:



Adding “taskness” too early

because in the first case, the implementation of the bounded and unbounded variants need only be written once and can be inherited in the concurrent versions, whereas in the second case, these implementations must be written twice: once for the non-concurrent variant and once for the concurrent one.

3.2.8. Class Aggregates

Values of non-limited class types can be written using *class aggregates*, which are very similar to Ada 83 record aggregates.

Following are the syntax rules for class aggregates:

```
class_aggregate ::=
   aggregate |
   expression with [aggregate | (null)];
```

Class aggregates can appear only in class type bodies[1]. The reason for this restriction is that class type bodies may declare data components, and such components are visible only within the body that declares them. Thus, the body of a class type is the only place where the completeness of a class aggregate can be checked. Note that this restriction can be overcome by careful design of the constructors of the class type (using **new** procedures, described in section 3.2.5).

The above restriction implies that the subtype of a class aggregate is determinable from the context (because class types cannot be nested). Therefore, qualification is not necessary.

The first variant of class aggregate can be used only in root class types (i.e. class types declared as **package type**, not **package subtype**). They must contain values for all the components of the class type, i.e. discriminants and components declared in the visible part, in the private part and in the body of the class type.

The second variant is useful for extending the value of the parent of a class subtype. The expression before the reserved work **with** must be belong to the subtype of the parent class. The aggregate must contain values for all components added in the current class subtype; (**null**) is used when no components are added.

Examples of class aggregates:

- The body of the package type Set in section 3.2.6 can be rewritten as follows:

  ```
  package body Set is

     ...

     function Intersection (Other : Set) return Set is
     begin
        return (Contents => Contents and Other.Contents);
     end Intersection;

  end Set;
  ```

- Here is a useful extension of Set:

  ```
  package subtype Counting_Set is Set with

     function Intersection (Other : Set) return Counting_Set;
        -- the subtype of the result is narrowed

  private

     Card : Natural := 0;

  end Counting_Set;
  ```

[1] Class aggregates are not the same as the qualified expressions yielding class type values presented in section 3.2.5. The former are used outside class type bodies to provide values of class types, and their parameters are class creation parameters consisting of discriminants and **new** procedure parameters. The latter are used inside class type bodies and specify the values of all the class type's data components.

```
package body Counting_Set is

   ...   -- provide new bodies for Add and Remove
         -- to update Card as needed.

   function Intersection (Other : Set) return Counting_Set is

      Uncounted_Result : constant Set :=
                         Set'(self).Intersect(Other);
                            -- call the parent's Intersect method

      Count : Natural := 0;

   begin
      for I in Uncounted_Result.Contents'Range loop
         if Uncounted_Result.Contents(I) then
            Count := Count + 1;
         end if;
      end loop;
      return Uncounted_Result with (Card => Count);
         -- Uncounted_Result, of subtype Set, is
         -- extended with the new component Card
   end Intersection;

end Counting_Set;
```

3.2.9. Redefining Methods

Methods can be redefined even if they are not redeclared in the specification of the class subtype. In this case, the redefined method must conform to the specification given in the youngest ancestor in which it appears, according to the rules given in [Ada 83 6.3.1].

All methods of class types are dynamically bound, that is, the subprogram body that is called is the one that appears in the body of the actual subtype of the class object. In many cases, the compiler will be able to tell at compile-time which subclass an object will be in, and thus bind statically in these cases. I felt that adding a special construct for providing static binding added too much complexity[1]. Moreover, methods would be "annotated" with binding mechanisms, which generally have nothing to do with the problem domain.

Subprograms defined in the body of class types, on the other hand, need not be dynamically bound, because they are invisible to heirs and to the outside world, so that their only direct callers are subprograms in the same class subtype.

When redefining a method, it often happens that its behavior must be only slightly modified. Therefore, the corresponding method in the parent class type can often be reused. This requires a way of viewing **self** (and also other objects of the class) as if they were instances of their parent class. This is done by using

[1] C++ provides both static and dynamic binding of methods, even in classes that can be inherited from. All dynamically bound methods must be marked as *virtual*. I think that this is very dangerous because a non-virtual method could be redefined without noticing its non-virtuality in the parent class.

Ada 83's qualified expressions and by extending the syntax of names to be more permissive with prefixes ([Ada 83 4.1.(2)]):

```
prefix ::= name | function_call | qualified_expression
```

An occurrence of T'(Object) means Object interpreted as an instance of the class subtype T. T must be a subtype of the base type of Object, and Constraint_Error is raised if Object is not in T.

The following example shows how this can be used to extend the behavior of the class type Part in section 3.2.7:

```
package subtype Extended_Part is Part with
private
   Y : Integer;
end Extended_Part;

package body Extended_Part is
   function Clone return self'Subtype is
   begin
      return Part'(self).Clone with (Y => self.Y);
   end Clone;
end Extended_Part;
```

3.2.10. Operators on Class Types

So far, the proposed class type extensions do not accommodate Ada 83's operator syntax very well. One way of making operators available for class types would be to include them inside the class type declaration, but this would have the disadvantage of requiring that the left operand always be the selector for the corresponding method, as in C++ [C++ 90]. This approach has been rejected because it makes it impossible (or at least syntactically ugly) to have a class type as the right operand of one of its operators.

Instead, operators must be declared outside the class declarations they operate on (possibly in the immediately enclosing block or in a separate package). Operator subprogram bodies will thus not have access to the inside view of the classes they operate on and will have to call methods of those classes in order to perform their function, as in the following example:

```
package Large_Integer_Class is

   package type Large_Integer is
      -- Operations used to implement operators are
      -- declared to have names that are identifiers

      function Right_Multiply (Right : Large_Integer)
         return Large_Integer;

      function Left_Multiply (Left : Integer)
         return Large_Integer;

      function Unary_Minus return Large_Integer;

      ...

   end Large_Integer;
```

```
      -- Declaration of the operators

      function "*" (Left, Right : Large_Integer)
         return Large_Integer;

      function "*" (Left : Integer; Right : Large_Integer)
         return Large_Integer;

      function "-" (Right : Large_Integer) return Large_Integer;

      ...

   end Large_Integer_Class;

   package body Large_Integer_Class is

      package body Large_Integer is

         ...

      end Large_Integer;

      -- The operators are implemented as method calls

      function "*" (Left, Right : Large_Integer)
         return Large_Integer is
      begin
         return Left.Right_Multiply(Right);
      end "*";

      function "*" (Left : Integer; Right : Large_Integer)
         return Large_Integer is
      begin
         return Right.Left_Multiply(Left);
      end "*";

      function "-" (Right : Large_Integer) return Large_Integer is
      begin
         return Right.Unary_Minus;
      end "-";

   end Large_Integer_Class;
```

Operators defined using this scheme behave well with respect to inheritance: as methods are dynamically bound to objects, the right version of the method implementing the operator will always be called.

One can see that this way of declaring operators also works well in conjunction with constrained genericity (see section 3.2.15) and also provides a way of writing templates (see 2.11.6) that include operators, as the following example shows:

```
   package Integer_Number is

      package type Number is   -- abstract (see section 3.2.14)

         procedure new (Initial_Value : in Integer := 0);
         -- Methods necessary for implementing operators
         function Sum (Right : Number) return Number;
         function Product (Right : Number) return Number;
         function Quotient (Right : Number) return Number;
         function Remainder (Right : Number) return Number;
         function Negated return Number;

      end Number;

      function "+" (Right : Number) return Number;
      function "-" (Right : Number) return Number;

      function "+"   (Left, Right : Number) return Number;
      function "-"   (Left, Right : Number) return Number;
      function "*"   (Left, Right : Number) return Number;
      function "/"   (Left, Right : Number) return Number;
      function "mod" (Left, Right : Number) return Number;

   end Integer_Number;
```

All heirs of the class type Integer_Number.Number can be used as a generic actual parameter of an instantiation of a generic unit starting with:

```
   with Integer_Number;

   generic
      with package type Some_Number is Integer_Number.Number with <>;
```

Within such a generic, all operations of Integer_Number.Number are available, and as a result of dynamic binding, the methods implementing the operators will be those redefined in the generic actual parameter.

3.2.11. Generic Classes

Class types can be generic. The syntax for their generic part is as usual, as well as the semantics: generic class types cannot be used directly to create objects; they must be instantiated and each instance is a new type that can be used just as an explicitly declared class type.

Class subtypes can also be generic. A subtype of a generic class must also be generic; even if no generic parameters are added, the reserved word **generic** must still appear in its declaration.

When inheriting from a generic class, the generic formal part of the parent is also inherited (and must not be repeated). The names of generic formal parameters of a generic class subtype must not clash with those of its ancestors.

Following is the syntax for instantiations of generic classes:

```
   package_type_instantiation ::=
      package type identifier is
         new generic_package_type_name [generic_actual_part];

   package_subtype_instantiation ::=
      package subtype identifier is parent_package_type_name with
         new generic_package_type_name [generic_actual_part];
```

```
task_type_instantiation ::=
   task type identifier is
      new generic_task_type_name [generic_actual_part];

task_subtype_instantiation ::=
   task subtype identifier is parent_task_type_name with
      new generic_task_type_name [generic_actual_part];
```

There are two ways of instantiating a generic class type or subtype:

- using a *class type instantiation* (package_type_instantiation or task_type_instantiation in the syntax), thus creating a new class type from a generic class type or subtype. In the case of a generic class subtype, generic actual parameters must be given (explicitly or implicitly) for all generic formal parameters of the generic class subtype, and of all its ancestors;
- using a *class subtype instantiation* (package_subtype_instantiation or task_subtype_instantiation in the syntax), thus creating an heir of the parent class, which is itself an instance of some strict ancestor of the generic class subtype being instantiated. For the instantiation

  ```
  package subtype B is A with
     new GB(<params_B>);
  ```

  to be legal, there must be an ancestor GA of GB such that A is an instance of GA, and <params_B> must provide generic actual parameters for all generic formal parameters of GB and all its ancestors up to, but not including, GA.

  This is similar to the rules on instantiating child units of generic units in Ada 9X: a child of a generic can only be instantiated as the child of an instance of its parent (see [Ada 9X 94 10.1.1(19)]).

A complete example using this mechanism can be found in 3.4.1, along with a comparison to the corresponding feature in Ada 9X.

3.2.12. Classes Exporting Generics

It is often useful for ADTs to export generic subprograms (the only generics allowed in class types are subprograms, see syntax in section 3.2.3); a very common example is an iterator on a composite structure. In Ada 83, a stack ADT for objects of type Item_Type would be written:

```
package Stacks is
   type Stack is limited private;

   procedure Push (Item : in Item_Type; On : in out Stack);
   procedure Pop (Item : out Item_Type; From : in out Stack);

   generic
      with procedure Action (Item : in Item_Type);
   procedure Iterate (On_Stack : in Stack);
end Stacks;

A, B : Stacks.Stack;
```

Which can be transformed into the following using the package type construct presented in section 3.2.3:

```
limited package type Stack is
   procedure Push (Item : in Item_Type);
   procedure Pop (Item : out Item_Type);

   generic
      with procedure Action (Item : in Item_Type);
   procedure Iterate;
end Stack;

A, B : Stack;
```

There are two possibilities for making instances of Iterate available to clients of the package type Stack:

- link instances of Iterate to the package object from which they were instantiated, which implies that the generic procedure Iterate is visible as a selected component of objects of the type Stack (this would be the most natural way of treating instances of Iterate, because it is exactly what would happen if A and B were directly declared as packages instead of package objects);
- link instances of Iterate to the package type Stack, so that they can be used on all objects of type Stack. This implies that Iterate should be visible as a selected component of Stack itself, as if Stack were a package instead of a package type (all generics defined within package types would be visible as selected components in this way, but no other entities).

In the following discussion, it is assumed that a procedure

```
procedure Put_Item (Item : in Item_Type);
```

is visible and that its effect is to write a human-readable representation of Item to Text_IO.Current_Output.

- If Iterate is visible as a selected component of A and B, which would be the most consistent with usual Ada visibility rules, then an instance of the iterator would only be usable on one Stack object:

  ```
  procedure Put_Contents is
     new A.Iterate(Action => Put_Item);
  ```

  Put_Contents is only usable for writing the items in A, and another instance of Iterate would be required for writing the elements in B. Moreover, in the absence of consistent naming conventions, it is not directly apparent, from an isolated call to Put_Contents, which Stack object it will operate on.

  Also, objects of type Stack may be dynamically allocated and thus disappear before some instance of Iterate that still “points” to them, as in

  ```
  X : Stack'Access := new Stack;

  procedure Put_Contents_Of_X is
     new X.Iterate(Action => Put_Item);

  ...
  ```

```
X := null;
Put_Contents_Of_X;    -- call to a “dangling” subprogram
```

- If Iterate is visible within Stack, which introduces a dissymmetry between usual packages and objects of package types, then an instance of Iterate such as

  ```
  procedure Put_Contents is
     new Stack.Iterate(Action => Put_Item);
  ```

  can be used for writing the items in several Stack objects. This would be written in dotted notation, as if Put_Contents had been added to the visible part of Stack:

  ```
  A.Put_Contents;
  B.Put_Contents;
  ```

From the preceding discussion, it seems preferable to adopt the second solution for the following reasons:

- the objects which are operated on are easily identified;
- instantiating generics defined within class types extends the functionality of the type, not only of one object;
- the dangling subprogram problem does not exist.

Note that if Ada had subprograms types, this problem would not exist, except that the action to be iterated would probably have to be a global subprogram[1].

An instance of a generic declared in a class type is added to the visible part of that class type. Such an instance is visible in the intersection of the scopes of the class type and that of the generic instantiation. An instantiation of a generic declared in a class type is illegal if it is a homograph of a method of a visible class subtype of the class type. This situation is illustrated below:

```
package type Stack is
   ...   -- same as above
end Stack;

package subtype Stack_2 is

   procedure Put_Contents;

end Stack_2;

procedure Put_Contents is
   new Stack.Iterate(Action => Put_Item);   -- illegal

S : Stack_2;
```

[1] Precisely because of such restrictions, no proposal for subprogram types are made. I believe that there are very few situations in which a type consisting of a single subprogram component is useful, and that it is much more often the case that some data must be associated with the subprogram. In this case, a class type with a single method is appropriate. Exactly this situation arises with X-Windows callbacks, which are one of the reasons motivating the requirement for having access-to-subprogram types in Ada-9X.

```
...

S.Put_Contents;   -- would be ambiguous
```

It is also illegal to declare a method of a class subtype that is a homograph of a visible instance of a generic of an ancestor of the class subtype.

Instances of generic subprograms of class types are dynamically bound, just as regular subprograms. This means that when a generic subprogram of a class type is instantiated, all redefinitions of this subprogram, in all heirs of the class, must be automatically instantiated. When a call to such an instance is made, dynamic binding is used to select the subprogram body that is executed.

3.2.13. Classes with Discriminants

As the syntax given in section 3.2.3 shows, class types can have discriminants. A class type with discriminants behaves in essentially the same way as a discriminated record type.

From inside a class type, discriminants are directly visible as constants. From outside, they are visible as selected components, also as constants.

There are two ways of creating subtypes of class types that have discriminants: by creating subclasses, as explained in section 3.2.7, and by constraining the discriminants, in the usual Ada 83 way. These two aspects are orthogonal[1].

The following example shows how both subtyping mechanisms can be used:

```
limited package type Bag (Duplicates_Allowed : Boolean) is
   procedure Insert (Item : in Item_Type);
   procedure Remove (Item : in Item_Type);
end Bag;

subtype Set is Bag(Duplicates_Allowed => False);

limited task subtype Concurrent_Bag is Bag;
   -- no constraint on the discriminant Duplicates_Allowed

limited task subtype Concurrent_Set is Set;
   -- Duplicates_Allowed is constrained to False
```

[1] It is somewhat unfortunate that the two subtyping methods have so different notations. This could be corrected by using a syntax like the following for package and task types:

```
type T is                                 type T is
   package                                   task
      procedure P;                              procedure P;
   end T;                                    end T;

subtype U is T with                       subtype U is T with
   package                                   task
      function F return Boolean;                function F return Boolean;
   end U;                                    end U;
```

I rejected this idea because I didn't want to become incompatible with Ada 83's notation for task types.

3.2.14. Abstract Classes

An *abstract* class type is a class type for which some methods do not have a body. Creating instances of abstract classes is allowed, but this will generally not be very useful.

Calling an unimplemented method of an abstract package object raises Program_Error, and calling a procedure (entry) of an abstract task object for which there is no accept statement makes the caller wait until the task terminates and raises Tasking_Error[1].

Another variant here would be to explicitly mark classes as abstract, perhaps with a syntax like

```
abstract package type T is
   ...
end T;
```

and require bodies for all methods in non-abstract classes. This was considered too complicated because abstractness would have to be part of generic formal class types (see next section), which would complicate the syntax too much.

3.2.15. Constrained Genericity

Constrained genericity is an additional way of constraining generic formal types, specifying that corresponding generic actual types must be class types that inherit directly or indirectly from some (possibly abstract) class type.

New kinds of generic parameter declarations are thus introduced, with the following syntax:

```
generic_parameter_declaration ::=
   identifier_list : [in [out]] type_mark [:= expression];
 | type identifier is generic_type_definition;
 | private_type_declaration
 | with subprogram_specification [is name];
 | with subprogram_specification [is <>];
 | with [limited] package type identifier [is
                     class_type_name with <>];
 | with [limited] task type identifier [is
                     class_type_name with <>];
 | with [limited] package type identifier is
       new generic_package_type_name [generic_actual_part] with <>;
 | with [limited] task type identifier is
       new generic_task_name [generic_actual_part] with <>;
 | with generic_package_or_task_instantiation

generic_instantiation ::=
   generic_package_or_task_instantiation
 | generic_class_type_instantiation
 | generic_subprogram_instantiation
```

[1] This is to maintain compatibility with Ada 83: an entry is not required to have a corresponding accept statement, and the caller waits until the task reaches such an accept statement, which may never happen.

```
generic_package_or_task_instantiation ::=
   package identifier is
      new generic_package_name [generic_actual_part];
 | task identifier is
      new generic_task_name [generic_actual_part];

generic_class_type_instantiation ::=
   package type identifier is
      new generic_package_type_name [generic_actual_part];
 | task type identifier is
      new generic_task_type_name [generic_actual_part];

generic_subprogram_instantiation ::=
   procedure identifier is
      new generic_procedure_name [generic_actual_part];
 | function designator is
      new generic_function_name [generic_actual_part];
```

Note that the above extensions imply that tasks can be generic units. The full syntax for generic tasks is not given here as it is straightforward.

Following are the matching rules for the new kinds of generic parameters:

- A generic parameter of the form

  ```
  with [limited] package type T;
  ```

  is matched by any class type. Within instances, all occurrences of T stand for the subtype of the generic formal type given as the corresponding generic actual type.

  A generic formal limited package type can be matched by an actual task subtype.

  This construct is similar to Ada-83's generic formal private types, but it additionally requires that the actual type be a class type. This can be useful when some class and one if its heirs must be imported into a generic, as in the following example:

  ```
  generic
     with package type Base;
     with package type Heir is Base with <>;
  ```

- A generic parameter of the form

  ```
  with [limited] package type T is Base with <>;
  ```

  is matched by any (direct or indirect) heir of the class type Base. Within the generic unit, T and objects of type T may be used like Base, but in instances, all occurrences of T stand for the subtype of the generic formal type given as the corresponding generic actual type.

  A generic formal limited package type can be matched by an actual task subtype.

  Note that the syntax says package *type*, but actually, it is a *subtype* that is imported into the instance. This is for symmetry with other generic formal types, which are also seen as subtypes within the generic unit.

- A generic parameter of the form

  ```
  with [limited] task type T is Base with <>;
  ```

  obeys the same rules as the preceding form, but with tasks instead of packages. Moreover, the full declaration of the type Base is allowed to be a limited package type, in order to be able to constrain the generic actual type to a tasking heir of some nontasking class type.

- A generic parameter of the form

  ```
  with [limited] package type T is
                    new Base(Parameter, ...) with <>;
  ```

  is matched by any heir of an instance of any heir of the class type Base with the same generic actual parameters.

  This kind of generic parameter is especially useful in situations like the following, where Objects and Stacks of Objects are needed:

  ```
  generic
     type Item is private;
  package type Stack is

     procedure Push (X : in Item);
     procedure Pop (X : out Item);

  end Stack;

  generic
     type Object is private;
     with package type Object_Stack is new Stack(Item => Object)
        with <>;
  ```

  This kind of generic parameterization can be considered as an extension of Ada 83's generic array type mechanism to structures other than arrays.

- A generic parameter of the form

  ```
  with [limited] task type T is
                    new Base(Parameter, ...) with <>;
  ```

  obeys the same rules as the preceding form, but with tasks instead of packages.

- A generic parameter of the form

  ```
  with package P is new GP(Parameter, ...);
  ```

  or

  ```
  with task P is new GP(Parameter, ...);
  ```

  is matched by any instance of GP with the same generic actual parameters.

In the above definitions, the phrase *with the same generic actual parameters* means that generic actual parameters corresponding to the same generic formal parameter must conform according to the same rules as given in [Ada 83 6.3.1] for the conformance of multiple occurrences of the specification of the same subprogram.

3.2.16. Iterators

As discussed earlier, iterators are a very important feature of ADTs. They may be expressed in Ada 83 with generic procedures taking a procedure as generic parameter.

In regard to the semantic difficulties with task types that export generics, and also as an extension to the **for** loop construct as an array iterator, it may be reasonable to make iterators a language construct, much along the lines of what has been done in [CLU 81].

The following example shows how an iterator construct simplifies the traversal of a composite structure while also improving the readability of the code:

```
generic
   type Index is private;
   with function "<" (Left, Right : Index) return Boolean;
           -- total order on indices to define iteration order.

   type Item is private;
package type Dynamic_Array is

   procedure Add (At_Index : in Index; Value : in Item);

   function Value_At (The_Index : Index) return Item;

   generic
      with procedure Action (The_Index : in Index;
                             The_Item  : in out Item);
   procedure Iterate (Descending : in Boolean := False);

end Dynamic_Array;

subtype Name is String(1 .. 10);

package type Occurrence_Table is
   new Dynamic_Array(Index => Name,
                     Value => Natural);

procedure Increment_All (A : in Occurrence_Table) is

   procedure Increment_One (I : in Name; X : in out Natural) is
   begin
      X := X + 1;
   end;

   procedure Iterate_Increment_One is
      new Occurrence_Table.Iterate(Increment_One);

begin
   A.Iterate_Increment_One;
end Increment_All;
```

Now consider the same example written with the iterator construct to be proposed:

```
generic
   ...   -- same as above
package type Dynamic_Array is

   ...   -- same as above

   iterator All_Items (Descending : in Boolean := False)
      yield (The_Index : out Index;
             The_Item  : out in Item);

end Dynamic_Array;

subtype Name is String(1 .. 10);

package type Occurrence_Table is
   new Dynamic_Array(Index => Name,
                     Value => Natural);

procedure Increment_All (A : in out Occurrence_Table) is
begin
   for The_Index => I, The_Item => X in A.All_Items loop
      X := X + 1;
   end loop;
end Increment_All;
```

This second version has several advantages over the first one:

- Using a **for** loop is a more common way of iterating over a composite structure.
- The code is more readable due to the fact that actions to be performed at each iteration are included inside a loop and do not have to be factored out as a procedure. Furthermore it is not necessary to invent new names for the iterated action or for the instance of Iterate.
- Nested loops are much more readable if expressed with an iterator construct than with instantiations of generic subprograms.

Here are the rules governing the iterator construct:

- Iterator declarations:

```
iterator_specification ::=
   iterator identifier [formal_part] [yield yield_part]

iterator_declaration ::= iterator_specification;

yield_part ::=
   (iterator_parameter_specification {;
    iterator_parameter_specification})

iterator_parameter_specification ::=
   identifier_list : iterator_mode type_mark

iterator_mode ::= [out] | out in | in
```

- Iterator calls:

  ```
  loop_parameter_specification ::=
     identifier in [reverse] discrete_range   -- like Ada 83
   | [parameter_association {, parameter_association}] in
        iterator_name [actual_parameter_part]
  ```

- Iterator bodies:

  ```
  iterator_body ::=
     iterator_specification is
       [declarative_part]
     begin
        sequence_of_statments
    [exception
        exception_handler
       {exception_handler}]
     end [iterator_simple_name];

  yield_statement ::= yield [actual_parameter_part];
  ```

The modes of the parameters in a yield part are reversed because they are expressed with respect to the iterator, just as formal parameter modes of a subprogram are expressed with respect to the subprogram. In the above example, values of The_Index are given by the iterator to its caller, so the mode is **out**, and values of The_Item are first given by the iterator to its caller, and then read back by the iterator, so the mode is **out in**.

Iterators can be overloaded like subprograms. The signature that is used to resolve overloading consists of the parameters of the yield part together with those of the formal part of the iterator.

The effect of a yield statement is to transfer control to the sequence of statements of the **for** loop calling the iterator (the *iterated sequence of statements*). The execution of the iterator is then resumed after the yield statement. Exceptions and parameters are treated as if the iterated sequence of statements were a procedure called by the iterator. A yield statement is only allowed within the body of an iterator.

An iterator can be declared in a task. In this case, an iterator body can appear in the corresponding task body in place of an accept statement. The iterated sequence of statements is executed by the task containing the iterator[1].

Iterators declared in class types are dynamically bound, with similar implications as for generic subprograms exported from class types (see end of section 3.2.12).

3.2.17. Finalization

The expression of classes as package types allows a very clean extension for finalizing objects when they become inaccessible. I do not make a complete

[1] Another possibility would be to have the caller execute the iterated sequence of statements, thus requiring a rendez-vous at each iteration. I chose execution by the iterator owner because it is closer to what happens with generic procedures exported from tasks. Further study of the subject should be made in order to determine the advantages of both options.

treatment of the mechanisms that lead to terminating objects, as this has already been done in [Ada 9X 94], but I make a proposal on how the finalization of class objects can be expressed. This proposal is very similar to Newton's **inner** construct [Newton 90].

The syntax rules for bodies of class types (and also of regular packages and tasks) are extended as follows:

```
package_body ::=
   package body package_simple_name is
     [declarative_part]
  [begin
      sequence_of_statements
  [final
      sequence_of_statements]
  [exception
      exception_handler
     {exception_handler}]
   end [package_simple_name]];

task_body ::=
   task body task_simple_name is
     [declarative_part]
   begin
      sequence_of_statements
  [final
      sequence_of_statements]
  [exception
      exception_handler
     {exception_handler}]
   end [task_simple_name];
```

The sequence of statements after the reserved word **final** (the *final sequence of statements*) is executed when an object is terminated. For tasks, this execution can happen after a **terminate** alternative has been selected. In this case, it is the responsibility of the programmer to make sure that the final sequence of statements cannot block the task.

The exception handlers in a package or task body apply to both sequences of statements. Nested blocks can be used if more specific handlers are needed.

For the finalization of an object of a class subtype, the final part of the actual subtype of the object is first executed, and then the final parts of all heirs of that subtype are executed in the reverse inheritance order.

For nested packages or tasks, the final parts of the most nested constructs are executed first. For packages or tasks at the same nesting level, the final parts are executed in the reverse elaboration order.

3.2.18. A Complete Example

The following example, first presented in section 2.11.6 using templates, illustrates the use of abstract classes and constrained genericity to provide the functionality of templates.

```
package Integers is

   package type Number is   -- probably abstract

      procedure new (Initial_Value : in Standard.Integer := 0);
      procedure new (Initial_Value : in self'Subtype);

      procedure Set_Value (New_Value : in Standard.Integer);

      function Equal     (Other : self'Subtype) return Boolean;
      function Less_Than (Other : self'Subtype) return Boolean;

      function Plus   (Other : self'Subtype) return self'Subtype;
      function Minus  (Other : self'Subtype) return self'Subtype;
      function Times  (Other : self'Subtype) return self'Subtype;
      function Divide (Other : self'Subtype) return self'Subtype;
      function Modulo (Other : self'Subtype) return self'Subtype;

      procedure Increment (By : in Standard.Integer := 1);

   end Number;

   -- operators on Number, as shown in section 3.2.10

   function "+" (Right : Number) return Number;
   function "-" (Right : Number) return Number;

   function "+" (Left, Right : Number) return Number;
   ...

   function "=" (Left, Right : Number) return Boolean;
   function "=" (Left  : Number;
                 Right : Standard.Integer) return Boolean;
   function "<" (Left, Right : Number) return Boolean;
   ...

end Integers;


package Lists is

   generic
      type Item_Type is private;
   limited package type List is

      procedure Insert_At_Tail (Item : in Item_Type);
      procedure Insert_At_Head (Item : in Item_Type);

      function Head return Item_Type;
      function Tail return Item_Type;

      function Length return Natural;
      function Is_Empty return Boolean;
      procedure Destroy;

   end List;

end Lists;


with Integers,
     Lists;

generic
   with package type Integer_Number is Integers.Number with <>;
   with limited package type List_Of_Numbers is
           new Lists.List(Item_Type => Integer_Number) with <>;
procedure List_Prime_Numbers (From, To : in Integer_Number;
                              Into     : in out List_Of_Numbers);
```

All objects of type Integer_Number created inside List_Of_Numbers will be constrained to their initial subtype. Thus, if List_Prime_Numbers is called with From or To belonging to a strict subtype of Integer_Number, Constraint_Error will be raised. To avoid this, one would specify List_Of_Numbers as

```
   with limited package type List_Of_Numbers is
           new Lists.List
                  (Item_Type => Integer_Number'Access) with <>;
```

thus inserting access values instead of actual objects into the number lists.

```
procedure List_Prime_Numbers (From, To : in Integer_Number;
                              Into     : in out List_Of_Numbers) is

   Current  : Integer_Number := From;
   Divisor  : Integer_Number;
   Is_Prime : Boolean;

begin
   Into.Destroy;   -- make sure Into is empty
   while Current <= To loop
      Divisor.Set_Value(2);
      Is_Prime := True;

      while Divisor < Current loop
         if Current mod Divisor = 0 then
            Is_Prime := False;
            exit;
         end if;
         Divisor.Increment;
      end loop;

      if Is_Prime then
         Into.Insert_At_Tail(Item => Current);
      end if;

      Current.Increment;
   end loop;
end List_Prime_Numbers;
```

The following shows how List_Prime_Numbers can be used with an actual implementation of the abstract type Integers.Number:

```
with Integers;

package Long_Integers is

   type Slice is range 0 .. 9;

   package subtype Number (Max_Digits : Natural) is
      Integers.Number with
      -- no method specifications need to be overridden
   private
      Value : array (0 .. Max_Digits) of Slice;
              -- in private part (instead of body) so that
              -- siblings have visibility on each other's Value
   end Number;

end Long_Integers;


package body Long_Integers is

   package body Number is

      procedure new (Initial_Value : in Standard.Integer := 0) is
      begin
         for I in Value'Range loop
            begin
               Value(I) := Slice(Initial_Value / 10 ** I mod 10);
            exception
               when Constraint_Error =>
                  exit;
            end;
         end loop;
      end new;

      function Plus (Other : self'Subtype) return self'Subtype;

         Result : self'Subtype
                     (Max_Digits => Max(Max_Digits,
                                        Other.Max_Digits);

         Carry : Slice := 0;

      begin
         for I in Result.Value'Range loop
            Result.Value(I) :=
               (Value(I) + Other.Value(I) + Carry) mod 10;
            Carry := (Value(I) + Other.Value(I) + Carry) / 10;
         end loop;

         if Carry > 0 then
            raise Constraint_Error;
         end if;
      end Plus;

      ...

   end Number;

end Long_Integers;
```

```
with Long_Integers,
     Lists,
     List_Prime_Numbers;

package Long_Integer_Primes is

   subtype Long_Int is Long_Integers.Number(Max_Digits => 50);

   package type List_Of_Long_Integer_Numbers is
      new Lists.List(Item_Type => Long_Int);

   procedure List_Long_Prime_Numbers is
      new List_Prime_Numbers
             (Integer_Number  => Long_Int,
              List_Of_Numbers => List_Of_Long_Integer_Numbers);

end Long_Integer_Primes;
```

3.2.19. Implementation Notes

This section shows a few interesting aspects of the implementation of the proposals made in this chapter. No attempt is made to completely cover the subject, or to prove that every proposal can be implemented at a reasonable cost.

Dynamic binding for package an task types can be done in the classical way: the first element in the representation of each object is a pointer to a subtype descriptor. This pointer serves as a tag, uniquely identifying each class. The subtype descriptor contains general information such as a pointer to the parent subtype's descriptor, whether the subtype is a task, and the addresses of all methods of the subtype.

When calling a method of some class, the constraint checks on the actual parameters of the method must be done after determination of which subprogram will be called. This is because the subtypes of method parameters can change when methods are redefined. This necessitates additional information (subtype descriptors) to be stored along with method addresses in subtype descriptors. It may also add some overhead due to the fact that the checks cannot be made inline, just before the call, as for normal subprogram calls.

When a generic subprogram declared in a class type is instantiated, then the possible redefinitions of this subprogram in heirs of the class must also be instantiated with the same generic parameters. This is because such generic subprograms, like other subprograms in class types, are dynamically bound and it is not always possible to determine before run-time which body will be called. This problem can be simplified by sharing the code of generic instances when feasible.

In section 3.2.7, automatic generation of a task type body from a package type body is presented. The goal is simply to provide mutual exclusion on objects of the type; therefore, the actual mechanism for achieving it can be simpler than a task body. It could, for instance, be done by adding a semaphore to the type's representation.

3.2.20. Summary of Incompatibilities with Ada 83

This proposal introduces four new reserved words: **self**, **yield**, **iterator**, and **final**. Existing Ada 83 programs that use these as identifiers must be modified.

As explained in section 3.2.3, the name of a task type can no longer be used to denote the current task object. Existing Ada 83 programs that use this convention must be modified to use **self** instead. Within generic units, the convention that the name of the generic denotes the current instance is maintained.

3.3. Discussion of the Proposal

This section discusses some of the consequences of the proposal presented in section 3.2. First, some implications of using subtypes as subclasses are shown, with points of comparison to other languages. Then, a few notes on program verification suggest that Ada could benefit from rules that cannot be checked during compilation, much like Eiffel 3's system validity rules.

3.3.1. Comments on Subtyping as the Inheritance Mechanism

My proposal for classes in Ada is based on subtyping as the inheritance mechanism. It is heavily based on Eiffel, with the difference that I do not attempt to offer a solution where compile-time type (class) checking is possible. This section explains the rationale for this choice and shows some of its consequences.

I wanted to provide object-oriented extensions to Ada that would allow maximum flexibility, permitting the use of inheritance for specialization (generally requiring narrowing the subtypes of method parameters) as well as for generalization (generally requiring widening of method parameters). There were three options:

- Provide class types in a new type system, as other object-oriented languages do, with rules for implicit conversions within an inheritance hierarchy. This is the approach taken in [Eiffel 92] and [Modula-3 89a].
- Use derived types for classes, as done in Ada 9X. I rejected this idea because I hadn't thought of class wide types, which are absolutely crucial in providing polymorphism.
- Provide classes as types and subclasses as subtypes: Ada 83 has a quite rich type/subtype model which, at least at first sight, seems appropriate. It is the approach taken in [Cohen 89], and also the one I chose[1].

[1] The main difference is that [Cohen 89] uses records as containers, where I use packages and tasks. The difference is mainly notational.

The main (negative) consequence of applying the subtype model to classes is that many class mismatches are (potentially) not detected until run-time. For instance, the following program fragment is legal:

```
package type T is ... end T;

package subtype T1 is T with ... end T1;

procedure P (X : T1);

...

X : T := T'(...);

...

P(X);   -- raises Constraint_Error
```

Note that Ada 83 has this behavior built into the subtype model. The following declaration, for instance, is legal but will raise Constraint_Error:

```
X : Natural := -1;
```

Compilers are required to generate code for such cases, even though they know that an exception will be raised[1]. This keeps the model simple, with a clear separation of compile-time and run-time errors. Accepting the program as legal even though it is known that Constraint_Error will be raised does not mean that the error *must* go undetected at compile-time: if a warning is issued by the compiler, the programmer can then react accordingly.

The same holds for class subtypes: some subtype mismatches can be detected before run-time, but they don't have to be and they don't make the program illegal. As explained in section 3.3.2 below, fully checking a program for class mismatches is not feasible without drastic restrictions to the inheritance mechanism.

Another consequence of the subtype model is that it does not, like Ada 9X, allow a mixed style of programming, where polymorphic entities can coexist with constrained ones: in my proposal, writing

```
procedure P (X : T);
```

where T is a class type, allows an object of any subclass of T as an actual parameter. Whereas in Ada 9X, one can force a parameter to be exactly of class T, or one of its subclasses:

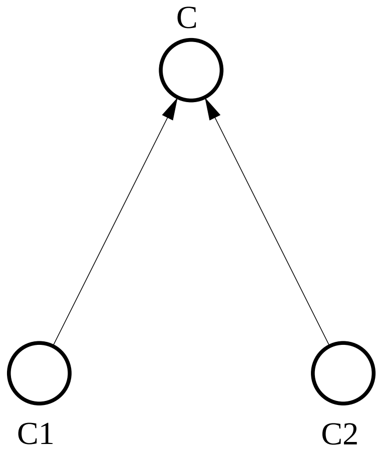
```
procedure P (X : T);          -- X must be exactly T
procedure P (X : T'Class);    -- X in any subclass of T
```

[1] They can, however, replace the assignment with code that raises the exception directly. Requiring compilers to detect such cases at compile-time would be very hard to specify: must they be detected for static expressions ? Within generic instantiations (easy if using macro-expansion, harder otherwise) ?

Comparison with Other Languages

Most other object-oriented languages (Eiffel, Modula-3, C++) use neither subtypes nor derived types as the underlying model for compatibility between classes.

The following three figures show the compatibility relations between the types resulting from a very simple class inheritance hierarchy made up of a base class C with two direct heirs C1 and C2. An arrow from T to U means "an actual parameter of type T matches as a formal parameter of type U".



Eiffel and Modula-3

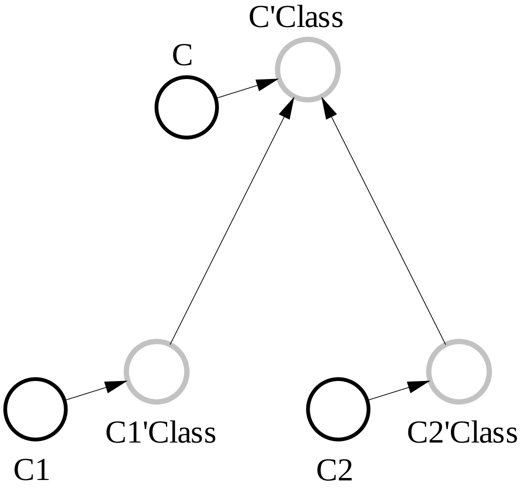
Eiffel treats all classes as distinct types, and defines the relation of conformance between them, so that subclasses conform to their superclass(es), and an instance of a subclass can always be used where an instance of the base class is required[1]. This model fits the object-oriented paradigm much better than the subtype model.

Also, as demonstrated by Eiffel, this model facilitates the provision of multiple inheritance because, as far as only type conformance is concerned, multiple inheritance simply allows making a type conformant with more than one parent type. This is in contrast to the subtype model, where, as pointed out in [Cohen 89], multiple inheritance faces the problem described in section 3.1.7 and thus is infeasible.

Modula-3 also treats all classes as distinct types. All types are ordered[2] by the "**<:**" relation, which means "is a subtype of", and determines when an object of a given type matches a parameter of another type. Modula-3's concept of subtype is very different from Ada's: it is a relation between types whereas in Ada a subtype is a subset of the set of values of a type.

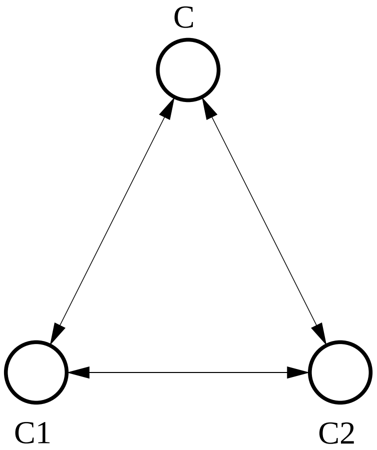
[1] Actually the Eiffel model is more complex because of the possibility of restricting exports in heirs.

[2] In fact, "**<:**" is reflexive and transitive, but not antisymmetric; but this has no effect on classes.



Ada 9X

For each class, Ada 9X implicitly creates a class wide type, and the inheritance hierarchy is built on the class wide types. Polymorphism is made possible by appropriate implicit conversion rules.



Subtypes

In the case of classes as subtypes, there is only one type for each base class. As subtypes are not determining for the legality of parameter associations, all combinations are possible; most of them will raise Constraint_Error.

Differences between Class Subtypes and “Normal” Ada 83 Subtypes

Class subtypes are different from usual (Ada 83) subtypes in that an object of a class subtype T cannot be assigned a value of subtype T1 if T1 is a strict subtype of T, because objects of class subtypes are constrained to their initial subtype, as defined in section 3.2.5. The following example illustrates this:

```
package type T is ... end T;

package subtype T1 is T with ... end T1;
```

```
X : T;
Y : T1 := ...;
Z : T := Y;      -- OK, Z's subtype becomes, and remains, T1

...

X := Y;   -- raises Constraint_Error
```

This may seem strange, but there are obvious implementation constraints behind this rule: if X were allowed to take any value of any subtype of T, then the memory allocated for T would have to be the size of T's largest subtype. This quantity can be hard to determine, as subtypes of T can be defined in other compilation units.

Moreover, if discriminants are added to heirs of T, then this maximum allocation size can become almost unbounded, as in the following example:

```
package subtype T2 (Name_Length : Natural := 0) is T with

   Name : String (1 .. Name_Length);

end T2;
```

These limitations can be overcome by using access types denoting class types, and allocating new objects when a class change is necessary, as shown below:

```
X : T'Access;
Y : T1;

procedure Free is new Unchecked_Deallocation(T, T'Access);

Free(X);
X := new T1'(Y);
```

Note that the above anomaly cannot happen if T is a limited class type, which I believe is the most frequent case. The limited case is exactly what happens in the first version of the Eiffel language (without expanded classes), because in Eiffel, assignment has reference semantics, and the user has explicit control over copying through the Clone method.

The most important subtyping rules, namely those regarding parameter passing, are the same for class subtypes as for usual Ada 83 subtypes.

Ada 9X has the same restriction on class wide types: objects of class wide types have their tag constrained to its initial value.

There is another difference, related to visibility rules: the visibility of selected components of objects of class (sub)types is determined by the declared subtype of the object, as shown in this example:

```
package type T is
   procedure P;
end T;

package subtype T1 is T with
   procedure Q;
end T1;
```

```
X : T;
Y : T1;

...

X.Q;   -- illegal, even though X might be in T1
Y.Q;   -- OK
```

Ada 83 has no such rules where visibility depends on the subtype of an object.

Problem when Modifying an Inheritance Tree at its Root

The following problem can arise with classes expressed as subtypes: consider the following pair of classes:

```
package type C1 is ... end C1;

package type C2 is

   procedure P (X : in C1);
   procedure P (X : in C2);

end C2;
```

Now, if for some reason, C1 and C2 are made to inherit from a common ancestor C, the code is rewritten as:

```
package type C is ... end C;

package subtype C1 is C with ... end C1;

package subtype C2 is C with

   procedure P (X : in C1);
   procedure P (X : in C2);

end C2;
```

Suddenly, the overloading of procedure P becomes illegal, because now C1 and C2 are subtypes of the same type, namely C, and both declarations of P are homographs with the same parameter and result type profile. The problem can be overcome by using a derived type, as follows:

```
package type C is ... end C;

type NC is new C;

package subtype C1 is C with ... end C1;

package subtype C2 is NC with

   procedure P (X : in C1);   -- X of type C
   procedure P (X : in C2);   -- X of type NC

end C2;
```

In practice, the problem will probably be more severe than in the very simple example above, as the inheritance chains between C and C1, respectively C and C2, may be quite long and thus involve many classes.

Ada 9X does not suffer this problem because it uses derived types instead of subtypes to achieve inheritance. Here is the same example written in Ada 9X (using 'Class for closest possible equivalence with the previous example):

```
package P1 is
   type C1 is tagged private;
end P1;

package P2 is
   type C2 is tagged private;

   procedure P (Self : in out C2; X : in P1.C1'Class);
   procedure P (Self : in out C2; X : in C2'Class);
end P2;
```

Then, C1 and C2 are made heirs of the same class C:

```
package P is
   type C is tagged private;
end P;

package P1 is
   type C1 is new P.C with private;
end P1;

package P2 is
   type C2 is new P.C with private;

   procedure P (Self : in out C2; X : in P1.C1'Class);
   procedure P (Self : in out C2; X : in C2'Class);
end P2;
```

The overloading of P remains legal because C1'Class and C2'Class are distinct types.

3.3.2. On Program Verification

The study of problems like covariance/contravariance of method parameters and the order of elaboration of library units leads to the following observation: some verifications of program correctness cannot be carried out in a complete way during compilation.

In the *classic compilation model* (used in Ada 83 as well as in most other languages supporting separate compilation), the compiler “sees” the unit currently being compiled, and the visible parts only of all units imported by that unit[1]. All compile-time errors that are required to be detected in Ada 83 fit into this model.

The two next sections discuss the implications of this model.

[1] There are a few exceptions to this rule, as for instance pragma Inline, but they do not change the language semantics.

Covariance/Contravariance of Method Parameters

In the first example given in section 3.2.7, it would probably be more useful, in most cases, to redefine F as

```
procedure F (Other : Parent) return Heir;
```

applying the principle of *covariance* (see [Cook 89]) to function results and formal parameters of mode **out**, and *contravariance* to formal parameters of mode **in** or **in out**.

The automatic covariance/contravariance of formal parameters when inheriting was rejected because I did not want to make it mandatory to rewrite the bodies of subprograms which have covariant parameters, as there are situations where covariance may not be desired. If covariance is required, then the body must be rewritten anyway, so it was considered to be a minor constraint to force rewriting of the specification as well.

The problems associated with covariance/contravariance are too complicated to be automated, because in general more than one class is involved, as illustrated by the following example:

```
package type T;

package type U is
   function F return T;
   procedure P (X : in T);
end U;
```

Then, in a specialization of T and U, both classes may need to be modified in parallel:

```
package subtype HT is T;

package subtype HU is U with
   function F return HT;
   procedure P (X : in HT);
end HU;
```

Note that HU.F and HU.P are narrowed, so that an instance of HU cannot always be used in a context where an instance of U is expected.

The possibility of narrowing the subtype (subclass) of a method parameter (i.e. covariance) is clearly a very useful feature. As shown in [Cook 89], requiring compile-time checking of programs that use covariance is not feasible within the classic compilation model.

This is why Eiffel 3 (see [Eiffel 92]) divides type conformance rules into two categories: class conformance and system conformance. A method call X.M(P) is *class-valid* if M appears in the declared (sub)class of X and the parameters conform. A method call with prefix X is *system-valid* if it is class-valid for all classes in the dynamic class set of X[1].

[1] This is a somewhat simplified version of the call validity rule. The full version takes into account the fact that some class attributes can have export restrictions in heirs.

System validity is a check that cannot be carried out during the compilation process, because calculating the dynamic class set requires a global view of the program (this is done with a fixed point algorithm, see [Eiffel 92] for full details).

Eiffel's system validity rule is stronger than needed, in the following sense: some programs whose execution will never make an invalid call are nevertheless declared illegal. This is because the general problem of determining whether or not a program will execute an invalid method call is undecidable. Therefore, the definition of the dynamic class set of an object starts by making all conditional statements non-conditional (note the similarity with elaboration order in section 4.5).

The same approach could be applied to classes as subtypes in Ada: the language rules can remain the same (i.e. raise Constraint_Error on a class mismatch) but there could be a post-processor which, based on information extracted from the program library, would indicate places in the program where a class mismatch is likely to occur.

Notes on the Compilation Process

The classic compilation model is enough for type checking, but other checks require visibility on the whole program, for instance:

- detection of some cases of erroneous execution, such as reading an uninitialized variable;
- determining an order of elaboration that will not raise Program_Error, if one exists;
- checking the system-validity of method calls in Eiffel.

Should an Ada system do this kind of checks ? I think that the language rules should be in terms of the classic compilation model. Additional tools, using a wider view than the compiler's, can then be developed for performing more elaborate checks[1], such as:

- when feasible, show places in the program where it is certain that Constraint_Error will be raised as the result of a constraint check[2];
- show places in the program where there is a risk of reading an uninitialized variable;
- show places in the program where a system-invalid method call might be performed;
- in Ada 9X, detect dispatching calls where all controlling operands might not all have the same tag (see [Ada 9X 94 3.9.2(15)]);
- etc.

[1] Some of them might be required by the standard, but this is a side issue to this discussion.

[2] Many compilers do this already, but only based on knowledge extracted from the unit currently being compiled and the specifications of imported units.

Note that an absolute verification of such properties is not possible, because they are undecidable.

3.4. Comparison with other Proposals for Object-Oriented Ada

Some object-oriented extensions to Ada have been proposed by a variety of authors. A few of them are briefly discussed in the following paragraphs.

3.4.1. Ada 9X [Ada 9X 94]

Tagged types in Ada 9X offer object-oriented features through type derivation instead of subtyping; all classes in a hierarchy are distinct types. For each tagged type T, there is an associated *class-wide type* T'Class; the set of values of T'Class consists of the discriminated union of the sets of values of all types derived directly or indirectly from T.

Tagged types offer essentially the same expressive power as Eiffel's classes, Modula-3's object types, or C++'s classes. But the approach behind tagged types is quite different in some respects:

- Each subclass in an inheritance hierarchy is a distinct type, instead of a subtype in other languages;
- The glue between classes and their methods is the encapsulating package, instead of a parenthesized construct as in other languages;
- Most object-oriented languages use the dot notation for method calls, where the selector's actual class determines the subprogram body to be called. Ada 9X uses one of the parameters of the method (the *controlling operand*) to achieve dynamic binding (dispatching).

These differences clearly make tagged types a “non-classic” approach, which might hinder Ada 9X's acceptance.

Here are a few aspects of tagged types, compared with the proposals made in section 3.2:

- For a tagged type T, the distinction between T and T'Class, while interesting is some cases (e.g. when it is known beforehand that dynamic binding is not necessary), adds unnecessary complexity to the language.
- Primitive operations of tagged types cannot have parameters of different classes in the same inheritance hierarchy, as the following example shows:

```
type Set is abstract tagged null record;

procedure Intersect (S1, S2 : in Set;
                     Result : in out Set) is abstract;
```

when specializing Set, e.g. to implement it using AVL trees, one would like to write the following:

```
type AVL_Set is new Set with private;

procedure Intersect (S1, S2 : in AVL_Set;
                     Result : in out AVL_Set);
```

The problem here is that Constraint_Error is raised if Intersect is called with S1, S2 and Result of type Set'Class and all three parameters do not have the same tag. To overcome this, one has to elect one of the parameters of Intersect as the controlling operand and make the other parameters class-wide, as in:

```
procedure Intersect (S1, S2 : in Set'Class;
                     Result : in out Set) is abstract;
   -- Result is the controlling operand
```

but then, in all heirs of Set, the S1 and S2 parameters of Intersect are forced to be of the type Set'Class, and can never be narrowed (see [Ada 9X 94 3.9.2(10)]). For instance, it is not possible to redefine Intersect as[1]:

```
procedure Intersect (S1, S2 : in AVL_Set'Class;
                     Result : in out AVL_Set);
```

These problems must be thought of when designing the top-most class in an inheritance hierarchy, because once the subtypes of non-controlling parameters are fixed, they apply to the whole hierarchy.

On the other hand, my proposal allows for much more freedom in redefining methods, but with some added implementation cost: parameter subtype checking can only be performed after dynamic binding has determined which subprogram is to be called.

- Generic operations of tagged types are not inherited (derived). This is a quite severe restriction that generally forces programmers to use class-wide types with generics. It is, however, alleviated by the existence of access-to-subprogram types, which can replace generics in some cases (e.g. iterators).
- Operations of tagged types are attached to their type only by the fact that they are in the same package. I think that it is unfortunate that no closer, syntactic, link exists between types and their operations.
- The fact that tagged types are essentially incrementally defined record types makes their use less verbose in many simple situations (e.g. aggregates can be used in many situations where package types require the use of **new** procedures, package bodies and generally heavier notation, as shown in the example below).

[1] More precisely, this declaration of Intersect is legal but, as its profile is different from the parent type's Intersect, it is not considered as a redefinition of the operation, and will therefore never be selected in a dispatching call to Intersect with Result in Set'Class, even if the tag of Result is AVL_Set.

The following example, taken from [Ada 9X 94 3.9], shows the syntactic differences between the class constructs of both languages:

- Formulation using Ada 9X tagged types:

```
type Point is tagged
   record
      X, Y : Real := 0.0;
   end record;

type Painted_Point is new Point with
   record
      Paint : Color := White;
   end record;

Origin : constant Painted_Point := (X | Y => 0.0, Paint => Black);



type Expression is tagged null record;

type Literal is new Expression with
   record
      Value : Real;
   end record;

type Expr_Ptr is access all Expression'Class;

type Binary_Operation is new Expression with
   record
      Left, Right : Expr_Ptr;
   end record;

type Addition is new Binary_Operation with null record;

type Subtraction is new Binary_Operation with null record;

Tree : Expr_Ptr :=
       new Addition'
          (Left  => new Literal'(Value => 5),
           Right => new Subtraction'
                        (Left  => new Literal'(Value => 13),
                         Right => new Literal'(Value => 7)));
```

- Formulation using package types:

```
package type Point is
   procedure new (X, Y : Real := 0);
   X, Y : Real := 0.0;
end;   -- requires a body (because of procedure new)

package subtype Painted_Point is Point with
   procedure new (X, Y : Real := 0; Paint : Color := White);
   Paint : Color := White;
end;

Origin : constant Painted_Point (X | Y => 0.0, Paint => Black);



package type Expression;
package subtype Literal is Expression with
   procedure new (Value : Real);
   Value : Real;
end;

package subtype Binary_Operation is Expression with
   procedure new (Left, Right : Expression'Access);
   Left, Right : Expression'Access;
end;

package subtype Addition is Binary_Operation;

package subtype Subtraction is Binary_Operation;

Tree : Expression'Access :=
       new Addition'
          (Left  => new Literal'(Value => 5),
           Right => new Subtraction'
                        (Left  => new Literal'(Value => 13),
                         Right => new Literal'(Value => 7)));
```

The next example provides comparison on a very important feature: the combination of generics and inheritance:

Example: Generic Sets with Multiple Implementations

As explained in section 3.1.5, ADTs with multiple implementations, sharing the same specification, are a very important requirement for my proposed extensions. Following is an example of a Set ADT with two different implementations (bounded and unbounded), first expressed using class subtypes, and then using Ada 9X tagged types.

Using Class Subtypes

The base class in the hierarchy is written as a limited package type. This type is abstract, but this can be expressed only as a comment:

```
generic
   type Element_Type is private;
limited package type Set is   -- abstract

   procedure Add (Item : in Element_Type);

   procedure Remove (Item : in Element_Type);

   function Is_Member (Item   : Element_Type;
                       Of_Set : Set) return Boolean;

   procedure Intersect (Other : Set);

end Set;
```

The unbounded variant inherits the specification without changes. It adds a "<" function to the generic formal part, required for implementation as an AVL tree:

```
generic
   with function "<" (Left, Right : Element_Type) return Boolean;
limited package subtype Unbounded_Set is Set with

   -- no redefinitions necessary

private

   ...

end Unbounded_Set;
```

The bounded variant needs a discrete representation of set elements:

```
generic
   type Discrete_Element is (<>);
   with function To_Discrete (E : Element) return Discrete_Element;
limited package subtype Discrete_Set is Set with

private

   Contents : array (Discrete_Element) of Boolean :=
              (others => False);

end Discrete_Set;
```

Note that if a more efficient version of Intersect were desired, then its parameter could be narrowed as follows:

```
   procedure Intersect (Other : Discrete_Set);
```

but of course, this removes the possibility of intersecting a discrete set with another kind of set.

As explained in section 3.2.11, the above generic class hierarchy can be instantiated in two ways:

- By instantiating the base class, and then its heirs, thus reproducing the generic inheritance hierarchy as a hierarchy of instances:

```
type Int is range 3 .. 12;

package type Int_Set is new Set(Element_Type => Int);

package subtype Int_Unbounded_Set is Int_Set with
   new Unbounded_Set("<");

function Identity (X : Int) return Int is begin return X end;

package subtype Int_Discrete_Set is Int_Set with
   new Discrete_Set(Discrete_Element => Int,
                    To_Discrete      => Identity);
```

- By instantiating the generic class subtypes directly, for cases where the base classes are not useful:

```
package type Int_Unbounded_Set is
   new Unbounded_Set(Element_Type => Int,
                     "<"          => "<");
```

```
package type Int_Discrete_Set is
   new Discrete_Set(Element_Type     => Int,
                    Discrete_Element => Int,
                    To_Discrete      => Identity);
```

In the first case, Int_Unbounded_Set and Int_Discrete_Set are subtypes of the same type Int_Set, so that they can be combined (for instance using Intersect). In the second case, Int_Unbounded_Set and Int_Discrete_Set are separate types (their respective base types are anonymous).

Using Ada 9X Tagged Types

In Ada 9X, the combination of child library units and generic formal packages allows several interesting variations. Note the use of a class wide parameter in the Intersect procedure:

```
generic
   type Element_Type is private;
package Sets is

   type Set is abstract tagged limited private;

   procedure Add (Item : in Element_Type;
                  To   : in out Set);

   procedure Remove (Item : in Element_Type;
                     From : in out Set);

   function Is_Member (Item   : Element_Type;
                       Of_Set : Set) return Boolean;

   procedure Intersect (The_Set : in out Set;
                        Other   : in Set'Class);

end Sets;
```

The simplest version, as a child of the library package Sets:

```
generic
   with function "<" (Left, Right : Element_Type) return Boolean;
package Sets.Unbounded is

   type Set is new Sets.Set with private;

   procedure Add (Item : in Element_Type;
                  To   : in out Set);

   ...

   procedure Intersect (The_Set : in out Set;
                        Other   : in Set'Class);

end Sets.Unbounded;
```

The next version, using a formal instance of Sets, is applicable even if Sets is not a library package:

```
with Sets;

generic
   with package Some_Sets is new Sets(<>);
   with function "<" (Left, Right : Some_Sets.Element_Type)
           return Boolean;
package Unbounded_Sets is

   type Set is new Some_Sets.Set with private;

   procedure Add (Item : in Some_Sets.Element_Type;
                  To   : in out Set);

   ...

   procedure Intersect (The_Set : in out Set;
                        Other   : in Some_Sets.Set'Class);

end Unbounded_Sets;
```

The last version can be used to extend any extension of any instance of Sets:

```
with Sets;

generic
   with package Some_Sets is new Sets(<>);
   type Set_Type is abstract new Some_Sets.Set with private;
   with function "<" (Left, Right : Some_Sets.Element_Type)
           return Boolean;
package Unbounded_Sets is

   type Set is new Set_Type with private;

   ...

end Unbounded_Sets;
```

Discrete variant omitted, as it does not exhibit any interesting problems.

Following are example instantiations of the above packages:

- Instance of the base class:

```
type Int is range 3 .. 12;

package Int_Sets is new Sets (Element_Type => Int);
```

- Instances of the three variants of subclasses:

```
package Int_Sets.Unbounded is new Sets.Unbounded("<");

package Int_Unbounded_Sets is
   new Unbounded_Sets(Some_Sets => Int_Sets,
                      "<"       => "<");

package Int_Unbounded_Sets is
   new Unbounded_Sets(Some_Sets => Int_Sets,
                      Set_Type  => Int_Sets.Set,
                      "<"       => "<");
```

3.4.2. Classic-Ada™[1] [Classic-Ada 89]

Classic-Ada defines objects as all having the same type Object_Id and uses a preprocessor to translate Classic-Ada programs into Ada. Subtyping is not used for expressing the inheritance hierarchy and there is no possible compile-time check for the existence of methods, which makes many program correctness problems detectable only when the program is executed, as in Smalltalk.

The notation chosen for method invocation does not have the syntax of a subprogram call, which makes the conversion between the class notation and the package notation difficult. Moreover, overloading of method names does not follow the same rules as in Ada 83.

The preceding comments suggest that Classic-Ada is merely a callable object management system, rather than an object-oriented extension to the Ada language.

3.4.3. Ada Subtypes as Subclasses [Cohen 89]

[Cohen 89] describes an approach that comes very close to the requirements expressed above. It is very well integrated into the Ada language.

I think that extendable types have the following disadvantages:

- Iterators or constructs involving generics cannot be specified correctly in a way such that they are inherited by subclasses. This is a severe restriction because iterator-like operations are very common and essential in the specification of an ADT.
- Inheritance-constrained genericity is available, but using structural matching rules on the generic parameters, instead of a template consisting of an abstract class specification. This approach implies that the definition of an abstraction must be duplicated every time it is used in a generic formal part.

[1] Classic-Ada™ is a trademark of Software Productivity Solutions, Inc.

4. Solutions to the Lower Level Problems

This chapter proposes solutions to some of the problems identified in chapter 2, and for which the object-oriented extensions of chapter 3 do not provide solutions.

4.1. Improvements to the Generic Facility

No specific improvements to generics are proposed in addition to those in chapter 3 and section 4.2.2. Following are some shortcomings of Ada 83 generics that are overcome by the class types extensions:

- Generics cannot be passed directly as generic parameters (see section 2.1.1), but class types (possibly abstract) can. As a class type can have generic subprograms, these can be passed as generic parameters, along with the class type they belong to. The example in section 2.1.1 can be rewritten as follows:

```
generic
   type Item_Type is private;
limited package type Abstract_Set is

   procedure Add (Item : in Item_Type);

   ...

   generic
      with procedure Action (On_Item : in Item_Type);
   procedure Iterate;

end Abstract_Set;

generic
   type Item_Type is private;
   with function "<" (Left, Right : Item_Type)
           return Boolean is <>;

   with limited package type Set is
           new Abstract_Set(Item_Type) with <>;
package Set_Operations is

   function Minimum (Of_Set : Set) return Item_Type;

   ...

   procedure Unite (First_Set, Second_Set : in Set;
                    Into                  : in out Set);
   ...
end Set_Operations;
```

Within Set_Operations, Set.Iterate is available as a generic procedure and can be instantiated, and then called as a selected component of an object of type Set (see section 3.2.12).

One could imagine a solution for having generic items as explicit generic parameters, but this would require very complicated matching rules. Moreover, such generics are often needed as operations of some type, and thus are better imported as a component of a class type.

- Using constrained genericity (see 3.2.15) and abstract task types (see 3.2.14), tasks can be passed as generic parameters, thus correcting the limitation identified in section 2.1.3.
- Generic parameters that are needed only in the implementation of the generic, presented in section 2.6, are not provided directly[1]; but generic classes, combined with inheritance, allow to extend the generic parameters of a class without modifying the original. The following example shows how this can be applied to the Tables package given in section 2.6:

```
generic
   type Key_Type is private;
   type Item_Type is private;
package type Table is

   procedure Insert (Key : in Key_Type; Item : in Item_Type);
   procedure Remove (Key : in Key_Type);

   function Search (Key : Key_Type) return Item_Type;

end Table;

generic
   -- Table's generic parameters are inherited,
   -- as explained in section 3.2.11.
   type Hash_Code is (<>);
   with function Hash_Of (Key : Key_Type) return Hash_Code;
package subtype Hashed_Table is Table;
   -- no methods need to be redeclared, but
   -- method bodies will be provided in the
   -- body of Hashed_Table.
```

4.2. Solutions to the Problems with ADTs

Many of the problems regarding the implementation of ADTs are solved by the higher level extensions (chapter 3):

- The templates, introduced in section 2.11.6 for illustration of the underlying problem, are available through the combination of abstract classes,

[1] Allowing generic parameters of generic bodies would introduce anomalies in the visibility rules: one would have to look at the body of a generic in order to be able to instantiate it.

inheritance and constrained genericity. An example of this can be found in section 3.2.18.

- Generics as type constructors (see section 2.11.5) are available through generic classes and iterators. Full uniform reference is not available (for instance, it is not possible to define indexed components) because this would change the language too much.

The following sections address problems related to the expression of ADTs and not covered by the extensions in chapter 3.

4.2.1. Allowing "=" to be Redefined for All Types

In view of the problems presented in section 2.11.1, "=" should be allowed to be redefined for all types, thus removing the special treatment of this operator.

I suggest replacing paragraphs 6.7(4-5) in [Ada 83] with the following:

> If an overloading of equality delivers a result of the predefined type Boolean, then it also implicitly overloads the inequality operator "/=", so that this still gives the complementary result to the equality operator. Explicit overloading of an inequality operator returning the predefined type Boolean is not allowed.

Allowing "=" to return types other than Boolean can be useful, as Tucker Taft pointed out in comp.lang.ada a few years ago. For instance, one could make it return an array of booleans as the result of comparing two arrays of integers, where each component of the result would be true if and only if the matching components of the parameters are equal.

Removing the restriction that "=" must have the same type for its left and right operands is also useful, as can be seen in the following example:

```
type Varying_Text (Max_Length : Natural) is private;

function "=" (Left : Varying_Text; Right : String) return Boolean;
function "=" (Left : String; Right : Varying_Text) return Boolean;
```

4.2.2. Eliminating the Reemergence of Predefined Operations

As explained in section 2.11.2, the way generic subprogram parameters are passed to instantiations can make hidden predefined operations of a generic actual type reemerge unexpectedly. The solution proposed here will make this problem disappear by changing the way operations of generic formal types are passed, thus changing the semantics of some Ada 83 programs. I think that it is more important to remove this language “bug” (especially when applied to equality, see section 4.2.1) than to preserve upward compatibility for a few programs relying on dubious features of the language.

Removing the anomaly amounts to changing paragraph 12.3(15) in [Ada 83], replacing it with the following (changes appear in italic):

> [beginning of paragraph does not change]. In addition, if within the generic unit a predefined operator or basic operation of a formal type is used, then within the instance the corresponding occurrence refers to the corresponding predefined operation of the actual type associated with the formal type, *except if the actual type is declared immediately within the visible part of a package, and this predefined operation has been redefined in this package specification, in which case the redefined version is used*[1].

It is possible that the above change reduces the opportunity to share code among generic instances, but I think that it is much more important to have solid ADTs than small executables. Moreover, situations where the code will not be as easily shareable are the seldom cases where, for instance, "+" has been redefined for an integer type, which deserves special treatment anyway.

[Ada 9X 94] does not eliminate the reemergence of predefined operations for non-tagged types, as proposed above, on the ground of preserving upward compatibility. This may in fact have the opposite consequence: consider all existing Ada 83 generic components which have

```
generic
   type T is private;
```

in their generic formal part; then all these components will become invalid when instantiated with actual types for which "=" has been redefined (which [Ada 9X 94] allows for all types). Furthermore, the dissymmetry in the treatment of tagged and non-tagged types is very likely to cause much confusion.

4.2.3. Automatic Perpetuation of Operations

In order to have an "=" operation on the type R given in the example of section 2.11.4, the programmer has to write

```
function "=" (Left, Right : R) return Boolean is
begin
   return Left.I = Right.I and Left.C = Right.C;
end "=";
```

which the compiler could very well do automatically (it is much less complicated than many other constructs in the language).

Proposed change to [Ada 83]:

- Remove paragraph 4.5.2(8), which explicitly states that if "=" is defined for a limited type, then it does not extend to composite types,

[1] In Ada 83, the derivable operations of a type are those declared in the same package visible part as the type on which they operate. If this rule changes (as will probably be the case in Ada 9X), then the rule should be extended to reflect this.

- Change paragraph 4.5.2(5) to the following:

  For two array values or two record values of the same type, the left operand is equal to the right operand if and only if for each component of the left operand there is a matching component of the right operand and vice versa; and the values of matching components are equal, *as given by the equality operator defined for the component type. This equality operator is the one that is visible within the same declarative part as the component type (i.e. if it has been redefined, then the predefined equality does not reemerge[1]). If one or more of the component types do not have a defined equality operator, then "=" is not defined for the composite type*.

  In particular, two null arrays of the same type are always equal; two null records of the same type are always equal.

Here again, I think that it is more important to preserve the consistency of ADTs with respect to composition, than favoring efficiency or ease of sharing code between generic instances. The simple cases, where predefined equality applies to all subcomponents of a composite type, can still be optimized by the compiler.

4.3. Extensions to the Exception Mechanism

The extensions proposed here to the exception mechanism are very much like Modula-3's exception mechanism: an exception declaration may optionally include a type mark, and a value of that type must be included in each raise statement for that exception. I think that this enhancement is simpler than including exceptions into Ada's type model, as is done in [9X Mapping 91b], while providing approximately the same functionality and expressive power.

Here is the new syntax for exception declarations, raise statements and exception handlers:

```
exception_declaration ::=
   identifier_list : exception [formal_part];

exception_handler ::=
   when exception_choice {| exception_choice} [formal_part] =>
      sequence_of_statements

raise_statement ::=
   raise [exception_name [actual_parameter_part]];
```

The formal part of an exception declaration may include only parameters with mode **in**, whether explicit or implicit.

The formal part of an exception handler must conform to the one in the corresponding exception declaration according to the conformance rules given

[1] Reemergence of predefined equality as the result of type composition does not exist in Ada 83, because equality is extended to composite types only if all components are non-limited, so that no component can have a redefined "=" operator.

for subprograms in [Ada 83 6.3.1], except that default parameter values must be omitted. If more than one exception choice is given, then the formal parts of corresponding exception declarations must conform to each other.

Parameters in the formal part of an exception handler are visible within the sequence of statements of that handler.

If a formal part is given in the declaration of an exception, then a matching actual part must be given in raise statements for that exception, except if all parameters have default values. A raise statement without an exception name re-raises the current exception with the same parameters.

Of course, this proposal raises the question of what happens when the evaluation of some actual parameter of an exception itself raises an exception, as in the following example:

```
E : exception (X : Positive);

...

raise E(-1);   -- Constraint_Error is raised instead
```

It is the responsibility of the programmer to make sure that such cases do not happen, or are dealt with properly.

4.4. Control of Direct Visibility (use clauses)

In order to solve the problems identified in section 2.16, the following extension to the use clause is proposed, based on the idea that overloadable entities are more useful to **use** then others:

```
use_clause ::= use package_name[.<>] {, package_name[.<>]};
```

where **use** P.<> means that all subprogram declarations, and all renaming declarations or generic instantiations that declare subprograms, occurring immediately within P's visible part, are made potentially visible. The rest of the definition of the use clause remains as is.

A finer control of direct visibility might be achieved with a syntax like **use** P.**function** or **use** P.**procedure**, where one could make a use clause apply selectively to procedures or functions.

I think that the above proposal is more natural than the **use type** clause in [Ada 9X 94], because it is in terms of packages, which are the main construct for controlling visibility in Ada, instead of types, which have no direct relationship with visibility.

4.5. Order of Elaboration of Library Units

The solution to be presented here will guarantee that any program with "reasonable" elaboration order dependencies between library units will be elaborated automatically without raising Program_Error. A few correct Ada 83 programs will be made illegal by this language change, but, as will be seen below, only questionable programming style can lead to such programs.

The Ada language has some constructs that can be defined in two steps, first the declaration and then the body. These constructs are

- subprograms,
- tasks,
- generic units (subprograms and packages),
- private types,
- incomplete types,
- deferred constants.

As the atomic entities to be dealt with are compilation units, private types and deferred constants are not relevant here because their declaration and "body" are always inside the same compilation unit (which is always a package specification). Incomplete types are even simpler: the restrictions placed on the use of an incomplete type (see [Ada 83 3.8.1]) are such that they cannot have any impact on elaboration order constraints. The other three constructs are very similar:

- subprogram bodies must be elaborated before they can be called,
- task bodies must be elaborated before task objects of the type are activated,
- generic unit bodies must be elaborated before they can be instantiated.

The above three rules will be used to define a partial ordering of compilation units, such that any order of elaboration following that partial ordering will guarantee proper elaboration, without Program_Error being raised because of non-elaborated entities.

At some time between the last compilation and the execution of the program, its elaboration order constraints must be determined. The set of Ada entities that are relevant for this calculation are:

- library units,
- library unit bodies,
- subunits,
- subprogram bodies,
- task bodies,
- generic bodies,

- generic instantiations.

These entities will hereafter be called *elaboration units*.

Generic instantiations must be expanded inline for the following calculations. All elaboration units created by this generic expansion must be taken into account[1]. Within a generic instantiation, a call to a generic formal subprogram counts as a call to the corresponding generic actual subprogram.

Subunits are also dealt with as if they were expanded inline: the body stubs are replaced with the corresponding separate bodies[2], and the with clauses of subunits are added to their parents' with clauses, in a transitive manner.

Library units and their bodies must be distinguished: the specification and body of library unit A are written A[spec] and A[body] respectively. Other specification/body pairs need not be distinguished because only the bodies of program units that are not library units need to be considered.

Library subprograms that have no separate specification are treated as if there were one, i.e. for a library subprogram P, two compilation units P[spec] and P[body] are used in the calculations. P's context clause applies to P[spec] in this case. This is necessary in order to distinguish calls to P (applying to P[body]) from occurrences of P in a with clause (applying to P[spec]).

Given any two elaboration units X and Y, the relation X **needs** Y is true if and only if one or more of the following conditions are met:

(a) Y is a subprogram and X potentially calls Y,

(b) Y is a generic body and X potentially instantiates Y,

(c) Y is a task type body and X potentially activates a task of type Y,

(d) Y is a library unit body or specification and X is in Y,

(e) X is the body of Y,

(f) Y is a library unit specification and Y appears in X's context clause.

Potentially calls, instantiates or activates means calls, instantiates or activates when all control structures (**if**, **case**, **loop**, exception handlers, etc.) are removed. As an example, consider the following compilation units:

```
package B is
   procedure P;
end B;
```

[1] The algorithm presented here is expressed in terms of program units with all generic units expanded. This does not mean that generics must be implemented via macro expansion. The important thing is the construction of the contracted call graph, which can certainly be optimized with respect to generic units.

[2] This is consistent with the way Ada 83 treats elaboration of body stubs (see [Ada 83 10.2(7)]: "The effect of the elaboration of a body stub is to elaborate the proper body of the subunit").

```
package C is
   procedure P;
end C;

package A is
   ...
end A;

with B, C;
package body A is

   ...

begin
   if False then
      B.P;
   else
      C.P;
   end if;
end A;
```

then it will be said that A[body] potentially calls B.P and C.P, even though False is static. Thus, in this case, A[body] **needs** B.P and A[body] **needs** C.P are both true.

The **needs** relation defines a directed graph on the set of all elaboration units in a program. This graph is called the *elaboration order graph*.

The elaboration order graph of a legal program may have circuits (for instance in the presence of mutually recursive subprograms).

The *contracted elaboration order graph* is constructed as follows:

- all arcs between compilation unit nodes are kept,
- for any two nodes X and Y in the contracted elaboration order graph, an arc from X to Y is added if and only if there exists a path from X to Y going only through non-compilation unit nodes in the elaboration order graph.
- all nodes in the elaboration order graph that are not compilation units are removed,

The order of elaboration of the compilation units of the program must be consistent with the partial ordering defined by the contracted elaboration order graph. If the contracted elaboration order graph contains circuits involving more than one node, then the program is illegal.

The cost of implementing this algorithm at link time is reasonable; although it has not actually been done, here are a few points of interest:

- The time and space complexity of the algorithm is of the same order as the number of calls to elaboration units present in a program. Other typical link-time operations such as code relocation have the same complexity.
- All the information needed for calculating the **needs** relation exists in the compiler at some time, and this information is small compared to other program library information.

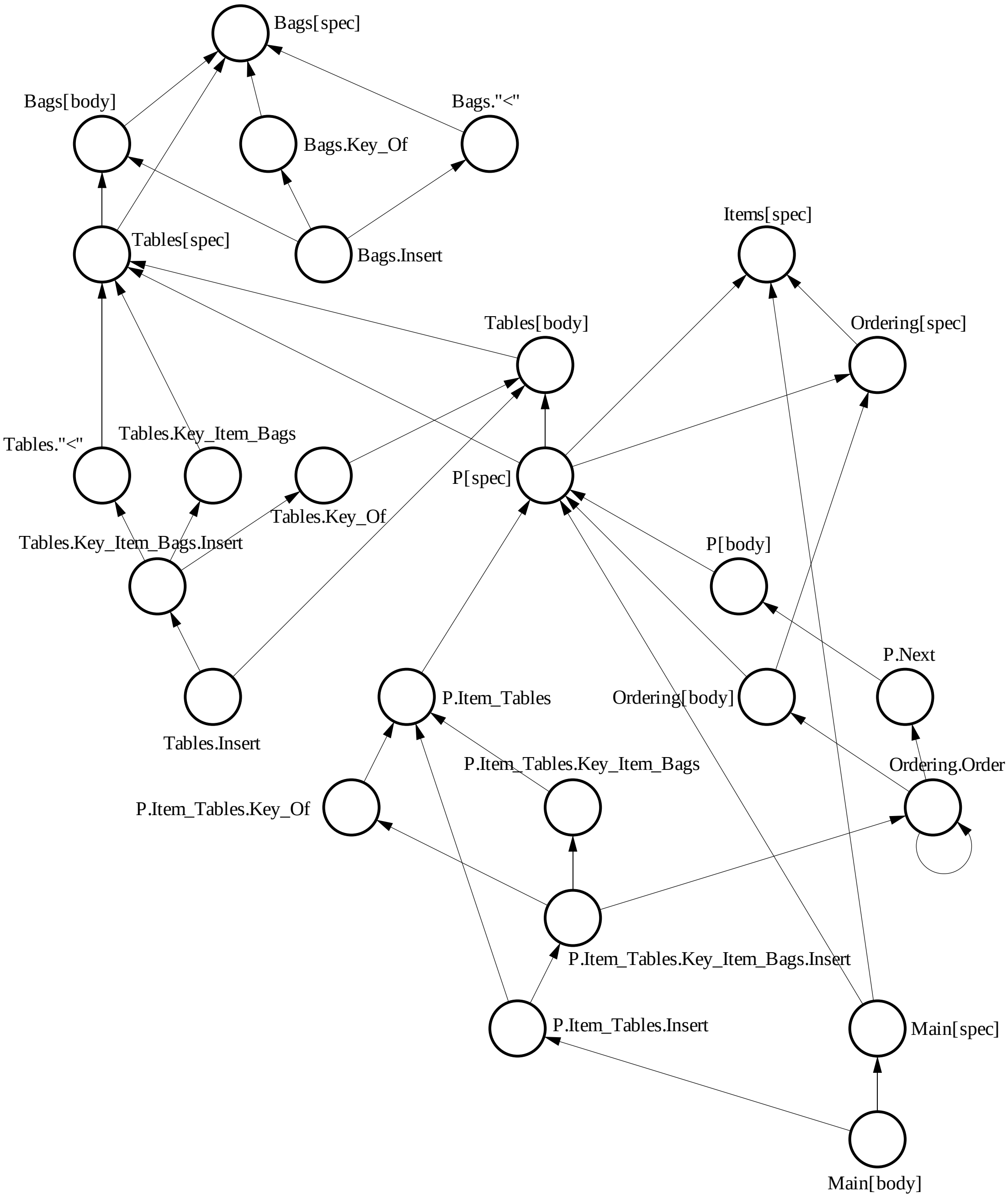
4.5.1. Example of Elaboration Order Calculation

The following example illustrates the algorithm presented above:

```
generic
   type Key_Type is limited private;
   type Item_Type is private;
   with function Key_Of (Item : in Item_Type) return Key_Type;
   with function "<" (X, Y : Key_Type) return Boolean is <>;
package Bags is
   type Bag is limited private;

   procedure Insert (Item : in Item_Type; Into : in out Bag);

   procedure Remove (Key : in Key_Type; From : in out Bag);

   function Search (Key : Key_Type; Within : Bag)
      return Item_Type;
end Bags;

package body Bags is
   procedure Insert (Item : in Item_Type; Into : in out Bag) is
   begin
      if Key_Of(Item) < ... then
         ...
      else
         ...
      end if;
   end Insert;

   ...  -- does not call "<" or Key_Of during elaboration
end Bags;

with Bags;
generic
   type Key_Type is private;
   type Item_Type is private;
   with function "<" (A, B : Key_Type) return Boolean is <>;
package Tables is
   type Table is limited private;

   procedure Insert (Key  : in Key_Type;
                     Item : in Item_Type;
                     Into : in out Table);

   procedure Remove (Key : in Key_Type; From : in out Table);

   function Search (Key : Key_Type; Within : Table)
      return Item_Type;

private
   type Key_Item is
      record
         The_Key  : Key_Type;
         The_Item : Item_Type;
      end record;

   function Key_Of (The_Key_Item : Key_Item) return Key_Type;

   package Key_Item_Bags is new Bags (Key_Type  => Key_Type,
                                      Item_Type => Key_Item,
                                      Key_Of    => Key_Of);
   type Table is new Key_Item_Bags.Bag;
end Tables;

package body Tables is
   function Key_Of (The_Key_Item : Key_Item) return Key_Type is
   begin
      return The_Key_Item.The_Key;
   end Key_Of;

   procedure Insert (Key  : in Key_Type;
                     Item : in Item_Type;
                     Into : in out Table) is
   begin
      Insert(Item => Key_Item'(Key, Item),
             Into => Into);
   end Insert;

   ...  -- does not call "<" during elaboration
end Tables;

package Items is
   type Item is private;
end Items;

with Items;
package Ordering is
   function Order (X, Y : Items.Item) return Boolean;
end Ordering;

with Items, Ordering, Tables;
package P is
   function Next (X : Items.Item) return Items.Item;

   type Occurence_Count is new Natural;

   package Item_Tables is
      new Tables(Key_Type  => Items.Item,
                 Item_Type => Occurence_Count,
                 "<"       => Ordering.Order);
end P;

with P;
package body Ordering is
   function Order (X, Y : Items.Item) return Boolean is
      use Items;
   begin
      return X /= Y and then
             (P.Next(X) = Y or else
              Order(P.Next(X), Y));
   end Order;
end Ordering;

package body P is
   function Next (X : Items.Item) return Items.Item is
   begin
      ...
   end;
end P;
```

```
with Items, P;
procedure Main is
   X : Items.Item;
   T : P.Item_Tables.Table;
begin
   P.Item_Tables.Insert(Key => X, Item => 12, Into => T);
end Main;
```

Expansion of all generics and calculation of the **needs** relation yields the following elaboration order graph:

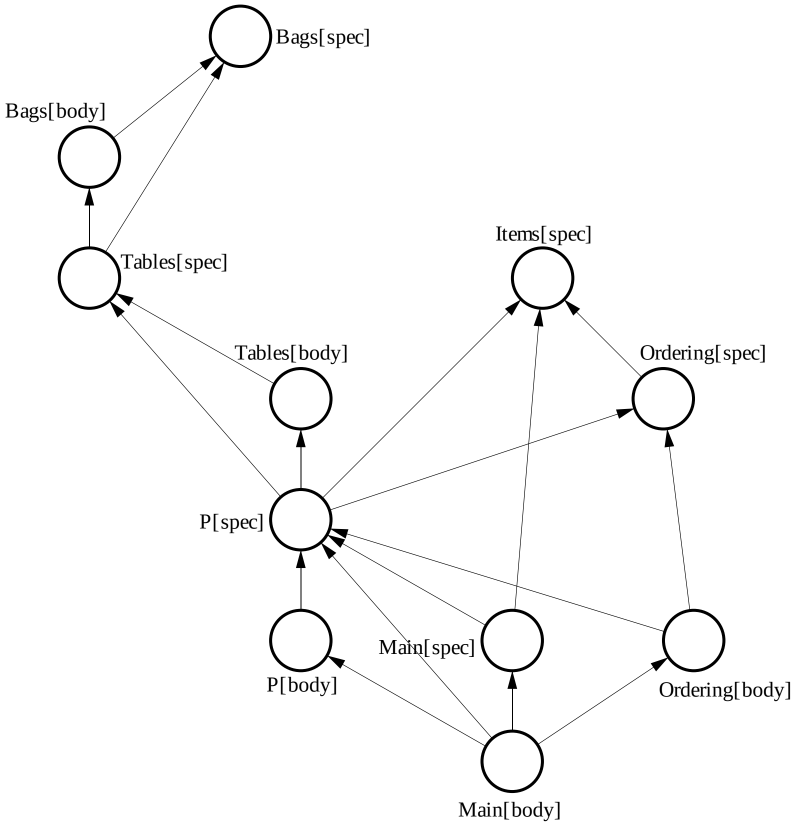


Elaboration Order Graph

Following are some examples of the **needs** relation in the above graph and the rules they refer to:

- Ordering.Order → P.Next (a),
- P[spec] → Tables[body] (b),
- P.Next → P[body] (d),
- Ordering[body] → Ordering[spec] (e),
- Ordering[body] → P[spec] (f).
- P.Item_Tables.Key_Item_Bags.Insert → Ordering.Order (a)
  because
  - Bags.Insert → Bags."<" (in Bags[body]),
  - Bags."<" → Tables."<" (instantiation of Bags in Tables[spec]),
  - Tables."<" → Ordering.Order (instantiation of Tables in P[spec]).

The contracted elaboration graph shows the elaboration order constraints. Note that the constraint that Bags[body] and Tables[body] must be elaborated before P[spec] is not implied by the Ada 83 elaboration order rules and would have required two Elaborate pragmas:



Contracted Elaboration Order Graph

Note: some of the constraints in the above graph are not necessary for proper elaboration of the program. For instance, Tables[spec] **needs** Bags[body] could be replaced with P[spec] **needs** Bags[body], which is weaker but still sufficient. One advantage of the stronger constraint is that it is easier to move from a generic version of the package Tables to a non-generic one (in which case the constraint would be necessary).

On the other hand, too many existing programs might become illegal because of such constraints (no extensive study has been made). The above rules could then be changed so that calls from generic bodies do not count for the **needs** relation, but are propagated to the instantiations instead.

4.5.2. Comparison with Current Ada 9X Proposals

Following is a discussion of the proposals on elaboration of library units made by the Ada 9X Mapping/Revision Team:

- In [9X Mapping 91a], a proposal is made to force elaboration of library unit bodies right after their specifications[1]. I think that the current proposal is better because elaboration order is determined by real dependencies instead of potential ones, and is thus closer to language semantics.

- [9X Mapping 91b] proposes, in addition to the preceding rule, to make pragma Elaborate recursive, i.e. to make it apply to the closure of the library unit which it names. This approach is dangerous because it can make the use of some library units erroneous except if another library unit is also present in the program, thus making the maintenance of component libraries difficult. Here is an example of such a situation:

  ```
  with Devices;
  package Low_Level_IO is
     ...
  end Low_Level_IO;

  with Low_Level_IO;
  pragma Elaborate(Low_Level_IO);
  package Text_IO is
     ...
  end Text_IO;

  with Low_Level_IO,
       Text_IO;
  package A is
     ...
  end A;
  ```

  In the above program fragment, the bodies of Low_Level_IO and packages used by Low_Level_IO are always elaborated before A's specification if pragma Elaborate is recursive. When the program is debugged and ready to be compiled for some embedded system, Text_IO might be removed from A's context clause, thus removing the constraint that Low_Level_IO's body must be elaborated before A.

[1] This proposal has been withdrawn in [9X Mapping 91b].

- In [9X Mapping 91c], the rule introduced in [9X Mapping 91a] (elaborate bodies as soon as possible after their specifications) is removed, and only the change to the effect of pragma Elaborate is left. I think that this is unacceptable because it is only a partial solution with severe drawbacks. Making a program elaborate properly still requires tedious verification and pragma additions, which is shown here to be avoidable.

5. Conclusion

I believe to have shown that making Ada 83 a true object-oriented language is possible without fundamentally altering the language's basic principles and syntax.

The proposal in chapter 3 offers an alternative to Ada 9X's tagged types, while providing similar expressive power. I think that my proposal is less complex than Ada 9X's, and closer to other object-oriented languages.

The goal of enhancing Ada 83 for the expression of ADTs, their composition and extension is achieved mainly through the following constructs:

- package types, which provide more cohesion between types and their operations,
- constrained genericity, allowing the signature of an abstraction to be used for parameterizing components,
- inheritance, enabling the specialization and extension of ADTs with maximum reuse of existing components.

The main deficiency of the proposal is probably the use of a subtype model for expressing inheritance. Comparison with inheritance mechanisms in other languages leads to the following observations:

- Compared with Ada 9X and Eiffel, too many subclass mismatches are detected too late (i.e. at run time). This is counterbalanced by added flexibility, for instance in the redefinition of method parameters that are class types.
- Many languages or compilation systems would benefit from post-compilation checkers for verifying certain aspects of program correctness that cannot be verified within the classic compilation model.
- Building a type system that supports classes and inheritance is better done from scratch than by adapting an existing type system. Eiffel is a very good illustration of this.

In retrospect, I would have chosen a mechanism other than subtyping for expressing inheritance, making each class a separate type, but with appropriate implicit conversions of subclasses toward their parent class. The syntax would be basically the same otherwise, except that class subtypes would not exist.

The extensions presented in this text certainly contain inconsistencies, which could only be detected through careful review by a design team.

No implementation of the proposed extensions has been carried out, for the following reasons:

- availability of modifiable compilers was very limited (hopefully this will change with GNU Ada);
- a preprocessor approach would be difficult to achieve because, among other things, of the visibility rules for class types, private types and the need for overload resolution;
- lack of time.

Implementing the extensions proposed in this thesis would be a very interesting further development from a technical point of view; but using these extensions in actual development would probably be a bad idea because one of the strengths of Ada is that it is a standard, with no allowed language subsets or supersets.

6. Bibliography

[Ada 83] "ANSI/MIL-STD-1815A, Ada® Programming Language", 22 January 1983.

[Ada 80] "Ada Reference Manual", Proposed Standard Document, United States Department of Defense, July 1980 (Reprinted November 1980).

[Ada 9X 94] "Ada 9X Reference Manual (Draft), Proposed ANSI/ISO Standard", Version 5.0, Ada 9X Mapping/Revision Team, Intermetrics, Inc., 1 June 1994.

[9X Rationale 94] "Ada 9X Rationale (Draft)", Version 5.0, Ada 9X Mapping/Revision Team, Intermetrics, Inc., 1 June 1994.

[9X Mapping 91c] Draft Ada 9X Project Report, "Ada 9X Mapping Document Volume II, Mapping Specification", Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, December 1991.

[9X Mapping 91b] Draft Ada 9X Project Report, "Ada 9X Mapping Document Volume II, Mapping Specification", Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, August 1991.

[9X Rationale 91b] Draft Ada 9X Project Report, "Ada 9X Mapping Document Volume I, Mapping Rationale", Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, August 1991.

[9X Mapping 91a] Ada 9X Project Report, "DRAFT Mapping Document", Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, February 1991.

[9X Rationale 91a] Ada 9X Project Report, "DRAFT Mapping Rationale Document", Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, February 1991.

[Ada++ 89] J. P. Forestier, C. Fornarino, P. Franchi-Zannettacci, "Ada++: A Class and Inheritance Extension for Ada", Proceedings of the Ada-Europe International Conference, Madrid, Pages 3-15, 13-15 June 1989.

[Alphard 81] Mary Shaw (editor) "Alphard: Form and Content", Springer Verlag, 1981.

[Booch 87] Grady Booch, “Software Components with Ada - Structures, Tools, and Subsystems”, Benjamin/Cummings Publishing Co., 1987.

[C++ 90] Margaret A. Ellis, Bjarne Stroustrup, “The Annotated C++ Reference Manual”, Addison-Wesley, 1990.

[C++ 91] Bjarne Stroustrup, “The C++ Programming Language, Second Edition”, Addison-Wesley, 1991.

[Classic-Ada 89] C. M. Donaldson, “Dynamic Binding and Inheritance in an Object-Oriented Ada Design”, Proceedings of the Ada-Europe International Conference, Madrid, Pages 16-25, 13-15 June 1989.

[CLU 81] Barbara Liskov, Russel Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Roby Scheifler, Alan Snyder, “CLU Reference Manual”, Lecture Notes in Computer Science #114, Springer Verlag, 1981.

[Cohen 89] Norman H. Cohen, Research Report “Ada Subtypes as Subclasses (Version 1)”, 9/8/89, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.

[Cook 89] W. R. Cook, “A Proposal for Making Eiffel Type-Safe”, The Computer Journal Vol. 32, No. 4, August 1989, Pages 305-311.

[DRAGOON 89] A. Di Maio, C. Cardigno, R. Bayan, C. Destombes, C. Atkinson, “DRAGOON: An Ada-Based Object Oriented Language for Concurrent, Real-Time, Distributed Systems”, Proceedings of the Ada-Europe International Conference, Madrid, Pages 39-48, 13-15 June 1989.

[Eiffel 92] Bertrand Meyer, “Eiffel, the Language”, Prentice Hall, 1992.

[Gautier 90] “Software Reuse with Ada”, Edited by Bob Gautier and Peter Wallis, IEE Computing Series 16, 1990.

[Gehani 84] N. H. Gehani and T. A. Cargill, “Concurrent Programming in the Ada Language: the Polling Bias”, Software Practice and Experience Vol. 14, No. 5, pages 413-427, May 1984.

[Genillard 89] C. Genillard, N. Ebel, A. Strohmeier, “Rationale for the Design of Reusable Abstract Data Types Implemented in Ada”, Ada Letters Vol. IX, No. 2, March/April 1989.

[Gonzales 90] Dean W. Gonzales, “Multitasking Software Components”, Ada Letters Vol. X, No. 2, January/February 1990.

[Gonzales 91] Dean W. Gonzales, “"=" Considered Harmful”, Ada Letters Vol. XI, No. 2, March/April 1991.

[Grogono 91] Peter Grogono, “Issues in the Design of an Object-Oriented Programming Language”, Structured Programming Vol. 12, No. 1, 1991.

[Hilfinger 83] Paul N. Hilfinger, “Abstraction Mechanisms and Language Design”, MIT Press, 1983.

[Hosch 90] Frederick A. Hosch, “Generic Instantiations as Closures”, Ada Letters January/February 1990.

[Lennon 92] Hugh John Lennon, “Polymorph Objects, A Step Further Than Multiple Inheritance ?”, February 1991.

[MAINSAIL 87] “MAINSAIL®[1] Overview”, 25 July 1987, XIDAK, Inc., Menlo Park, California.

[Meyer 88] Bertrand Meyer, “Object Oriented Software Construction”, Prentice Hall, 1988.

[Modula-3 89a] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson, “Modula-3 Report (revised)”, November 1, 1989, DEC Systems Research Center Report 52.

[Modula-3 89b] Mark R. Brown and Greg Nelson, “IO Streams: Abstract Types, Real Programs”, November 15, 1989, DEC Systems Research Center Report 53.

[Newton 90] Prof. Charles Rapin, “Paranewton Report”, Ecole Polytechnique Fédérale de Lausanne, Laboratoire de Compilation, September 1990.

[Oberon-2 91a] Hanspeter Mössenböck, Niklaus Wirth, “Differences between Oberon and Oberon-2”, Structured Programming Vol. 12, No. 4, 1991, Pages 175-177.

[Oberon-2 91b] Hanspeter Mössenböck, Niklaus Wirth, “The Programming Language Oberon-2”, Structured Programming Vol. 12, No. 4, 1991, Pages 179-195.

[Oberon-2 91c] Josef Templ, “Design and Implementation of SPARC-Oberon”, Structured Programming Vol. 12, No. 4, 1991, Pages 197-205.

[Requirements 90] Ada 9X Project Report, “Ada 9X Requirements”, Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, December 1990.

[Reuse 90] Ada 9X Project Report, “Ada Support for Software Reuse”, Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, October 1990.

[1] MAINSAIL is a registered trademark of XIDAK, Inc.

[Riese 89] Marc Riese and Giovanni Conti, “Drawbacks of Ada's Synchronization Mechanism and a Simple Solution”, 3rd International Workshop on Real-Time Ada Issues, 1989.

[Rovner 86] Paul Rovner, “Extending Modula-2 to build Large, Integrated Systems”, IEEE Software Vol. 3, No. 6, November 1986.

[Schäffer 90] Bruno Schäffer, “Chiron — A Homogeneous Object-Oriented Language”, Structured Programming Vol. 11, No. 4, 1990.

[Seidewitz 91] Ed Seidewitz, “Object-Oriented Programming Through Type Extension in Ada 9X”, Ada Letters Vol. XI, No. 2, March/April 1991.

[Tartan 78] Mary Shaw, Paul Hilfinger, Wm. A. Wulf, “TARTAN Language Design for the Ironman Requirement: Reference Manual, Notes and Examples”, SIGPLAN Notices Vol. 13, No. 9, September 1978, Pages 36-75.

Curriculum Vitae

Je suis né le 2 juillet 1964 à Lausanne. J'y ai suivi ma scolarité jusqu'en 1980; j'ai obtenu mon baccalauréat de type C au gymnase du Belvédère.

J'ai ensuite étudié à l'Ecole Polytechnique Fédérale de Lausanne (EPFL), où j'ai obtenu mon diplôme de mathématicien en 1987.

J'ai travaillé à l'EPFL de 1988 à 1991 comme assistant du Professeur Giovanni Coray, puis du Professeur Alfred Strohmeier.

Depuis août 1991, je travaille comme ingénieur de développement à Matrix SA (Lausanne), petite société d'informatique spécialisée dans l'archivage de documents.