Rationale for Ada 2005: Introduction

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This is the first of a number of papers describing the rationale for Ada 2005. In due course it is anticipated that the papers will be combined (after appropriate reformatting and editing) into a single volume for formal publication.

This first paper covers the background to the development of Ada 2005 and gives a brief overview of the main changes from Ada 95. Other papers will then look at the changes in more detail.

These papers are being published in the Ada User Journal. An earlier version of this first paper appeared in the Ada User Journal, Vol. 25, Number 4, December 2004. Other papers in this series will be found in later issues of the Journal or elsewhere on this website. The full series is expected to be

0 Introduction

1 Object oriented model

2 Access types

3 Structure and visibility

4 Tasking and Real-Time

5 Exceptions, generics etc

6 Predefined library

6a Containers

7 Epilogue

Keywords: rationale, Ada 2005.

1 Revision process

Readers will recall that the development of Ada 95 from Ada 83 was an extensive process funded by the USDoD. Formal requirements were established after comprehensive surveys of user needs and competitive proposals were then submitted resulting in the selection of Intermetrics as the developer under the devoted leadership of Tucker Taft. The whole technical development process was then comprehensively monitored by a distinct body of Distinguished Reviewers. Of course, the process was also monitored by the ISO committee concerned and the new language finally became an ISO standard in 1995.

The development of Ada 2005 from Ada 95 has been (and continues to be) on a more modest scale. The work has almost entirely been by voluntary effort with support from within the industry itself through bodies such as the Ada Resource Association and Ada-Europe.

The development is being performed under the guidance of ISO/IEC JTC1/SC22 WG9 (hereinafter just called WG9) chaired adroitly by James Moore whose deep knowledge leads us safely through

the minefield of ISO procedures. This committee has included national representatives of many nations including Belgium, Canada, France, Germany, Italy, Japan, Sweden, Switzerland, the UK and the USA. WG9 developed guidelines [1] for a revision to Ada 95 which were then used by the Ada Rapporteur Group (the ARG) in drafting the revised standard.

The ARG is a team of experts nominated by the national bodies represented on WG9 and the two liaison organizations, ACM SIGAda and Ada-Europe. The ARG was originally led with Teutonic precision by Erhard Plödereder and is currently led with Transalpine Gallic flair by Pascal Leroy. The editor, who at the end of the day actually writes the words of the standard, is the indefatigable Randy (fingers) Brukardt.

Suggestions for the revised standard have come from a number of sources such as individuals on the ARG, national bodies on WG9, users via email discussions on Ada-Comment and so on.

At the time of writing (June 2005), the revision process is essentially finished. The details of all individual changes are now clear and they have been integrated to form a new version of the Annotated Ada Reference Manual. This is currently being reviewed and the final approved standard should emerge in the first half of 2006.

There has been much discussion on whether the language should be called Ada 2005 or Ada 2006. For various reasons the WG9 meeting in York in June 2005 decided that the vernacular name should be Ada 2005.

2 Scope of revision

The changes from Ada 83 to Ada 95 were large. They included several major new items such as

- polymorphism through tagged types, class-wide types and dispatching,
- the hierarchical library system including public and private child packages,
- protected objects for better real-time control,
- more comprehensive predefined library, especially for character and string handling,
- specialized annexes such as those for system programming, real-time, and numerics.

By contrast the changes from Ada 95 to Ada 2005 are relatively modest. Ada 95 was almost a new language which happened to be compatible with Ada 83. However, a new language always brings surprises and despite very careful design things do not always turn out quite as expected when used in earnest.

Indeed, a number of errors in the Ada 95 standard were corrected in the Corrigendum issued in 2001 [2] and then incorporated into the Consolidated Ada Reference Manual [3]. But it was still essentially the same language and further improvement needed to be done.

Technically, Ada 2005 is defined as an Amendment to rather than a Revision of the Ada 95 standard and this captures the flavour of the changes as not being very extensive.

In a sense we can think of Ada 2005 as rounding out the rough edges in Ada 95 rather than making major leaps forward. This is perhaps not quite true of the Real-Time Systems annex which includes much new material of an optional nature. Nevertheless I am sure that the changes will bring big benefits to users at hopefully not too much cost to implementors.

The scope of the Amendment was guided by a document issued by WG9 to the ARG in September 2002 [1]. The key paragraph is:

"The main purpose of the Amendment is to address identified problems in Ada that are interfering with Ada's usage or adoption, especially in its major application areas (such as high-reliability, long-

lived real-time and/or embedded applications and very large complex systems). The resulting changes may range from relatively minor, to more substantial."

Note that by saying "identified problems" it implicitly rejects a major redesign such as occurred with Ada 95. The phrase in parentheses draws attention to the areas where Ada has a major market presence. Ada has carved an important niche in the safety-critical areas which almost inevitably are of a real-time and/or embedded nature. But Ada is also in successful use in very large systems where the inherent reliability and composition features are extremely valuable. So changes should aim to help in those areas. And the final sentence is really an exhortation to steer a middle course between too much change and not enough.

The document then identifies two specific worthwhile changes, namely, inclusion of the Ravenscar profile [4] (for predictable real-time) and a solution to the problem of mutually dependent types across two packages (see Section 3.3 below).

The ARG is then requested to pay particular attention to

A Improvements that will maintain or improve Ada's advantages, especially in those user domains where safety and criticality are prime concerns. Within this area it cites as high priority, improvements in the real-time features and improvements in the high integrity features. Of lesser priority are features that increase static error checking. Improvements in interfacing to other languages are also mentioned.

B Improvements that will remedy shortcomings in Ada. It cites in particular improvements in OO features, specifically, adding a Java-like interface feature and improved interfacing to other OO languages.

So the ARG is asked to improve both OO and real-time with a strong emphasis on real-time and high integrity features. It is interesting that WG9 rejected the thought that "design by contract" features should be added to the above general categories on the grounds that they would not be static.

The ARG is also asked to consider the following factors in selecting features for inclusion:

_ Implementability. Can the feature be implemented at reasonable cost?

_ Need. Do users actually need it? [A good one!]

_ Language stability. Would it appear disturbing to current users?

_ Competition and popularity. Does it help to improve the perception of Ada and make it more competitive?

_ Interoperability. Does it ease problems of interfacing with other languages and systems? [That's the third mention of interfacing.]

_ Language consistency. Is it syntactically and semantically consistent with the language's current structure and design philosophy?

An important further statement is that "In order to produce a technically superior result, it is permitted to compromise backwards compatibility when the impact on users is judged to be acceptable." In other words don't be paranoid about compatibility.

Finally, there is a warning about secondary standards. Its essence is don't use secondary standards if you can get the material into the RM itself. And please put the stuff on vectors and matrices from ISO/IEC 13813 [5] into the language itself. The reason for this exhortation is that secondary standards have proved themselves to be almost invisible and hence virtually useless.

The guidelines conclude with the target schedule. This includes WG9 approval of the scope of the amendment in June 2004 which was achieved and submission to ISO/IEC JTC1 in late 2005.

3 Overview of changes

It would be tedious to give a section by section review of the changes as seen by the Reference Manual language lawyer. Instead, the changes will be presented by areas as seen by the user. There can be considered to be six areas:

1 Improvements to the OO model. These include a more traditional notation for invoking an operation of an object without needing to know precisely where the operation is declared (the Obj.Op(...) or prefixed style), Java-like multiple inheritance using the concept of interfaces, the introduction of null procedures as a category of operation rather like an abstract operation, and the ability to do type extension at a more nested level than that of the parent type. There are also explicit features for overcoming nasty bugs that arise from confusion between overloading and overriding.

2 More flexible access types. Ada 95 access types have a hair-shirt flavour compared with other languages because of the general need for explicit conversions with named access types. This is alleviated by permitting anonymous access types in more contexts. It is also possible to indicate whether an access type is an access to a constant and whether a null value is permitted. Anonymous access-to-subprogram types are also introduced thus permitting so-called downward closures.

3 Enhanced structure and visibility control. The most important change here is the introduction of limited with clauses which allow types in two packages to refer to each other (the mutual dependence problem referred to in the WG9 guidelines). This is done by extending the concept of incomplete types (and introducing tagged incomplete types). There are also private with clauses just providing access from a private part. And there are significant changes to limited types to make them more useful; these include initialization using limited aggregates and composition using a new form of return statement.

4 Tasking and real-time improvements. Almost all of the changes are in the Real-Time Systems annex. They include the introduction of the Ravenscar profile (as explicitly mentioned in the WG9 guidelines) and a number of new scheduling and dispatching policies. There are also new predefined packages for controlling execution time clocks and execution time budgets and for the notification of task termination and similar matters. A change related to the OO model is the introduction of protected and task interfaces thereby drawing the OO and tasking aspects of the language closer together.

5 Improvements to exceptions, numerics, generics etc. There are some minor improvements in the exception area, namely, neater ways of testing for null occurrence and raising an exception with a message. Two small but vital numeric changes are a Mod attribute to solve problems of mixing signed and unsigned integers and a fix to the fixed-fixed multiplication problem (which has kept some users locked into Ada 83). There are also a number of new pragmas such as: Unsuppress to complement the Suppress pragma, Assert which was already offered by most vendors, Preelaborable_Initialization which works with the existing pragma Preelaborate, No_Return which indicates that a procedure never returns normally, and Unchecked_Union to ease interfacing to unchecked unions in C. There is also the ability to have more control of partial parameters of generic formal packages to improve package composition.

6 Extensions to the standard library. New packages include a comprehensive Container library, mechanisms for directory operations and access to environment variables, further operations on times and dates, the vectors and matrices material from ISO/IEC 13813 (as directed in the WG9 guidelines) plus commonly required simple linear algebra algorithms. There are also wide-wide character types and operations for 32-bit characters, the ability to use more characters in identifiers, and improvements and extensions to the existing string packages.

Of course, the areas mentioned above interact greatly and much of 2 and 3 could be classified as improvements to the OO model. There are also a number of changes not mentioned which will mostly be of interest to experts in various areas. These cover topics such as streams, object factory functions, subtle aspects of the overload resolution rules, and the categorization of packages with pragmas Pure and Preelaborate.

The reader might feel that the changes are quite extensive but each has an important role to play in making Ada more useful. Indeed many other changes were rejected as really unnecessary. These include old chestnuts such as **in out** and **out** parameters for functions (ugh), extensible enumeration types (a slippery slope), defaults for all generic parameters (would lead one astray), and user-defined operator symbols (a nightmare).

Before looking at the six areas in a little more detail it is perhaps worth saying a few words about compatibility with Ada 95. The guidelines gave the ARG freedom to be sensible in this area. Of course, the worst incompatibilities are those where a valid program in Ada 95 continues to be valid in Ada 2005 but does something different. It is believed that serious incompatibilities of this nature will never arise. There are however, a very few minor and benign such incompatibilities concerning the raising of exceptions such as that with access parameters discussed in Section 3.2.

However, incompatibilities whereby a valid Ada 95 program fails to compile in Ada 2005 are tolerable provided they are infrequent. A few such incompatibilities are possible. The most obvious cause is the introduction of three more reserved words: **interface**, **overriding**, and **synchronized**. Thus if an existing Ada 95 program uses any of these as an identifier then it will need modification. The introduction of a new category of unreserved keywords was considered for these so that incompatibilities would not arise. However, it was felt that this was ugly, confusing, and prone to introducing nasty errors. In any event the identifiers Overriding and Synchronized are likely to be rare and although Interface is clearly a likely identifier nevertheless to have it both as an identifier and as a keyword in the same program would be nasty. Note also that the pragma Interface which many compilers still support from Ada 83 (although not mentioned by Ada 95 at all) is being put into Annex J for obsolescent features.

3.1 The object oriented model

The Ada 95 object oriented model has been criticized as not following the true spirit of the OO paradigm in that the notation for applying subprograms to objects is still dominated by the subprogram and not by the object concerned.

It is claimed that real OO people always give the object first and then the method (subprogram). Thus given

```
package P is
  type T is tagged ... ;

  procedure Op(X: T; ... );
  ...
end P;
```

then assuming that some variable Y is declared of type T, in Ada 95 we have to write

```
P.Op(Y, ... );
```

in order to apply the procedure Op to the object Y whereas a real OO person would expect to write something like

```
Y.Op( ... );
```

where the object Y comes first and only any auxiliary parameters are given in the parentheses.

A real irritation with the Ada 95 style is that the package P containing the declaration of Op has to be mentioned as well. (This assumes that use clauses are not being employed as is often the case.) However, given an object, from its type we can find its primitive operations and it is illogical to require the mention of the package P. Moreover, in some cases involving a complicated type hierarchy, it is not always obvious to the programmer just which package contains the relevant operation.

The prefixed notation giving the object first is now permitted in Ada 2005. The essential rules are that a subprogram call of the form P.Op(Y, ...); can be replaced by Y.Op(...); provided that

- T is a tagged type,
- Op is a primitive (dispatching) or class wide operation of T,
- Y is the first parameter of Op.

The new prefixed notation has other advantages in unifying the notation for calling a function and reading a component of a tagged type. Thus consider the following geometrical example which is based on that in a (hopefully familiar) textbook [6]

```
package Geometry is
  type Object is abstract tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;

  function Area(O: Object) return Float is abstract;
  function MI(O: Object) return Float is abstract;
end;
```

The type Object has two components and two primitive operations Area and MI (Area is the area of an object and MI is its moment of inertia but the fine details of Newtonian mechanics need not concern us). The key point is that with the new notation we can access the coordinates and the area in a unified way. For example, suppose we derive a concrete type Circle thus

```
package Geometry.Circle is
  type Circle is new Object with
    record
      Radius: Float;
    end record;

  function Area(C: Circle) return Float;
  function MI(C: Circle) return Float;
end;
```

where we have provided concrete operations for Area and MI. Then in Ada 2005 we can access both the coordinates and area in the same way

```
X:= A_Circle.X_Coord;
A:= A_Circle.Area;          -- call of function Area
```

Note that since Area just has one parameter (A_Circle) there are no parentheses required in the call. This uniformity is well illustrated by the body of MI which can be written as

```
  function MI(C: Circle) is
  begin
    return 0.5 * C.Area * C.Radius**2;
  end MI;
```

whereas in Ada 95 we had to write

```
return 0.5 * Area(C) * C.Radius**2;
```

which is perhaps a bit untidy.

A related advantage concerns dereferencing. If we have an access type such as

```
type Pointer is access all Object'Class;
...
This_One: Pointer := A_Circle'Access;
```

and suppose we wish to print out the coordinates and area then in Ada 2005 we can uniformly write

```
Put(This_One.X_Coord); ...
Put(This_One.Y_Coord); ...
Put(This_One.Area); ...          -- Ada 2005
```

whereas in Ada 95 we have to write

```
Put(This_One.X_Coord); ...
Put(This_One.Y_Coord); ...
Put(Area(This_One.all)); ...     -- Ada 95
```

In Ada 2005 the dereferencing is all implicit whereas in Ada 95 some dereferencing has to be explicit which is ugly.

The reader might feel that this is all syntactic sugar for the novice and of no help to real macho programmers. So we shall turn to the topic of multiple inheritance. In Ada 95, multiple inheritance is hard. It can sometimes be done using generics and/or access discriminants (not my favourite topic) but it is hard work and often not possible at all. So it is a great pleasure to be able to say that Ada 2005 introduces real multiple inheritance in the style of Java.

The problem with multiple inheritance in the most general case is clashes between the parents. Assuming just two parents, what happens if both parents have the same component (possibly inherited from a common ancestor)? Do we get two copies? And what happens if both parents have the same operation but with different implementations? These and related problems are overcome by placing firm restrictions on the possible properties of parents. This is done by introducing the notion of an interface.

An interface can be thought of as an abstract type with no components – but it can of course have abstract operations. It has also proved useful to introduce the idea of a null procedure as an operation of a tagged type; we don't have to provide an actual body for such a null procedure (and indeed cannot) but it behaves as if it has a body consisting of just a null statement. So we might have

```
package P1 is
  type Int1 is interface;
  procedure Op1(X: Int1) is abstract;
  procedure N(X: Int1) is null;
end P1;
```

Note carefully that **interface** is a new reserved word. We could now derive a concrete type from the interface Int1 by

```
  type DT is new Int1 with record ... end record;
  procedure Op1(NX: DT);
```

We can provide some components for DT as shown (although this is optional). We must provide a concrete procedure for Op1 (we wouldn't if we had declared DT itself as abstract). But we do not

have to provide an overriding of N since it behaves as if it has a concrete null body anyway (but we could override N if we wanted to).

We can in fact derive a type from several interfaces plus possibly one conventional tagged type. In other words we can derive a tagged type from several other types (the ancestor types) but only one of these can be a normal tagged type (it has to be written first). We refer to the first as the parent (so the parent can be an interface or a normal tagged type) and any others as progenitors (and these have to be interfaces).

So assuming that Int2 is another interface type and that T1 is a normal tagged type then all of the following are permitted

```
type DT1 is new T1 and Int1 with null record;

type DT2 is new Int1 and Int2 with
  record ... end record;

type DT3 is new T1 and Int1 and Int2 with ...
```

It is also possible to compose interfaces to create further interfaces thus

```
type Int3 is interface and Int1;
...
type Int4 is interface and Int1 and Int2 and Int3;
```

Note carefully that **new** is not used in this construction. Such composed interfaces have all the operations of all their ancestors and further operations can be added in the usual way but of course these must be abstract or null.

There are a number of simple rules to resolve what happens if two ancestor interfaces have the same operation. Thus a null procedure overrides an abstract one but otherwise repeated operations have to have the same profile.

Interfaces can also be marked as limited.

```
type LI is limited interface;
```

An important rule is that a descendant of a nonlimited interface must be nonlimited. But the reverse is not true.

Some more extensive examples of the use of interfaces will be given in a later paper.

Incidentally, the newly introduced null procedures are not just for interfaces. We can give a null procedure as a specification whatever its profile and no body is then required or allowed. But they are clearly of most value with tagged types and inheritance. Note in particular that the package Ada.Finalization in Ada 2005 is

```
package Ada.Finalization is
  pragma Preelaborate(Finalization);
  pragma Remote_Types(Finalization);

  type Controlled is abstract tagged private;
  pragma Preeleborable_Initialization(Controlled);
  procedure Initialize(Object: in out Controlled) is null;
  procedure Adjust(Object: in out Controlled) is null;
  procedure Finalize(Object: in out Controlled) is null;

  -- similarly for Limited_Controlled
  ...
end Ada.Finalization;
```

The procedures Initialize, Adjust, and Finalize are now explicitly given as null procedures. This is only a cosmetic change since the Ada 95 RM states that the default implementations have no effect. However, this neatly clarifies the situation and removes ad hoc semantic rules. (The pragma Preelaborable_Initialization will be explained in a later paper.)

Another important change is the ability to do type extension at a level more nested than that of the parent type. This means that controlled types can now be declared at any level whereas in Ada 95, since the package Ada.Finalization is at the library level, controlled types could only be declared at the library level. There are similar advantages in generics since currently many generics can only be instantiated at the library level.

The final change in the OO area to be described here is the ability to (optionally) state explicitly whether a new operation overrides an existing one or not.

At the moment, in Ada 95, small careless errors in subprogram profiles can result in unfortunate consequences whose cause is often difficult to determine. This is very much against the design goal of Ada to encourage the writing of correct programs and to detect errors at compilation time whenever possible. Consider

```
with Ada.Finalization;  use Ada.Finalization;
package Root is
  type T is new Controlled with ... ;
  procedure Op(Obj: in out T; Data: in Integer);
  procedure Finalise(Obj: in out T);
end Root;
```

Here we have a controlled type plus an operation Op of that type. Moreover, we intended to override the automatically inherited null procedure Finalize of Controlled but, being foolish, we have spelt it Finalise. So our new procedure does not override Finalize at all but merely provides another operation. Assuming that we wrote Finalise to do something useful then we will find that nothing happens when an object of the type T is automatically finalized at the end of a block because the inherited null procedure is called rather than our own code. This sort of error can be very difficult to track down.

In Ada 2005 we can protect against such errors since it is possible to mark overridding operations as such thus

```
  overriding
  procedure Finalize(Obj: in out T);
```

And now if we spell Finalize incorrectly then the compiler will detect the error. Note that **overriding** is another new reserved word. However, partly for reasons of compatibility, the use of overriding indicators is optional; there are also deeper reasons concerning private types and generics which will be discussed in a later paper.

Similar problems can arise if we get the profile wrong. Suppose we derive a new type from T and attempt to override Op thus

```
package Root.Leaf is
  type NT is new T with null record;
  procedure Op(Obj: in out NT; Data: in String);
end Root.Leaf;
```

In this case we have given the identifier Op correctly but the profile is different because the parameter Data has inadvertently been declared as of type String rather than Integer. So this new version of Op will simply be an overloading rather than an overriding. Again we can guard against this sort of error by writing

```
overriding
procedure Op(Obj: in out NT; Data: in Integer);
```

On the other hand maybe we truly did want to provide a new operation. In this case we can write **not overriding** and the compiler will then ensure that the new operation is indeed not an overriding of an existing one thus

```
not overriding
procedure Op(Obj: in out NT; Data: in String);
```

The use of these overriding indicators prevents errors during maintenance. Thus if later we add a further parameter to Op for the root type T then the use of the indicators will ensure that we modify all the derived types appropriately.

3.2 Access types

It has been said that playing with pointers is like playing with fire – properly used all is well but carelessness can lead to disaster. In order to avoid disasters, Ada 95 takes a stern view regarding the naming of access types and their conversion. However, experience has shown that the Ada 95 view is perhaps unnecessarily stern and leads to tedious programming.

We will first consider the question of giving names to access types. In Ada 95 all access types are named except for access parameters and access discriminants. Thus we might have

```
type Animal is tagged
  record Legs: Integer; ... end record;

type Acc_Animal is access Animal;                -- named

procedure P(Beast: access Animal; ... );         -- anonymous
```

Moreover, there is a complete lack of symmetry between named access types and access parameters. In the case of named access types, they all have a null value (and this is the default on declaration if no initial value be given). But in the case of access parameters, a null value is not permitted as an actual parameter. Furthermore, named access types can be restricted to be access to constant types such as

```
type Rigid_Animal is access constant Animal;
```

which means that we cannot change the value of the Animal referred to. But in the case of access parameters, we cannot say

```
procedure P(Beast: access constant Animal);      -- not 95
```

In Ada 2005 almost all these various restrictions are swept away in the interests of flexibility and uniformity.

First of all we can explicitly specify whether an access type (strictly subtype) has a null value. We can write

```
type Acc_Animal is not null access all Animal'Class;
```

This means that we are guaranteed that an object of type Acc_Animal cannot refer to a null animal. Therefore, on declaration such an object should be initialized as in the following sequence

```
type Pig is new Animal with ... ;
Empress_Of_Blandings: aliased Pig := ... ;

My_Animal: Acc_Animal := Empress_Of_Blandings'Access;        -- must initialize
```

(The Empress of Blandings is a famous pig in the novels concerning Lord Emsworth by the late P G Wodehouse.) If we forget to initialize My_Animal then Constraint_Error is raised; technically the

underlying type still has a null value but Acc_Animal does not. We can also write **not null access constant** of course.

The advantage of using a null exclusion is that when we come to do a dereference

```
Number_of_Legs: Integer := My_Animal.Legs;
```

then no check is required to ensure that we do not dereference a null pointer. This makes the code faster.

The same freedom to add **constant** and **not null** also applies to access parameters. Thus we can write all of the following in Ada 2005

```
procedure P(Beast: access Animal);
procedure P(Beast: access constant Animal);

procedure P(Beast: not null access Animal);
procedure P(Beast: not null access constant Animal);
```

Note that **all** is not permitted in this context since access parameters always are general (that is, they can refer to declared objects as well as to allocated ones).

Note what is in practice a minor incompatibility, the first of the above now permits a null value as actual parameter in Ada 2005 whereas it was forbidden in Ada 95. This is actually a variation at runtime which is normally considered abhorrent. But in this case it just means that any check that will still raise Constraint_Error will be in a different place – and in any event the program was presumably incorrect.

Another change in Ada 2005 is that we can use anonymous access types other than just as parameters (and discriminants). We can in fact also use anonymous access types in

- the declaration of stand-alone objects and components of arrays and records,
- a renaming declaration,
- a function return type.

Thus we can extend our farmyard example

```
type Horse is new Animal with ... ;

type Acc_Horse is access all Horse;
type Acc_Pig is access all Pig;

Napoleon, Snowball: Acc_Pig := ... ;

Boxer, Clover: Acc_Horse := ... ;
```

and now we can declare an array of animals

```
Animal_Farm: constant array (Positive range <>) of access Animal'Class :=
                                                (Napoleon, Snowball, Boxer, Clover);
```

(With acknowledgments to George Orwell.) Note that the components of the array are of an anonymous access type. We can also have record components of an anonymous type

```
type Ark is
  record
    Stallion, Mare: access Horse;
    Boar, Sow: access Pig;
    Cockerel, Hen: access Chicken;
    Ram, Ewe: access Sheep;
```

```
    ...
  end record;

Noahs_Ark: Ark := (Boxer, Clover, ... );
```

This is not a very good example since I am sure that Noah took care to take actual animals into the Ark and not merely their addresses.

A more useful example is given by the classic linked list. In Ada 95 (and Ada 83) we have

```
type Cell;
type Cell_Ptr is access Cell;

type Cell is
  record
    Next: Cell_Ptr;
    Value: Integer;
  end record;
```

In Ada 2005, we do not have to declare the type Cell_Ptr in order to declare the type Cell and so we do not need to use the incomplete declaration to break the circularity. We can simply write

```
type Cell is
  record
    Next: access Cell;
    Value: Integer;
  end record;
```

Here we have an example of the use of the type name Cell within its own declaration. In some cases this is interpreted as referring to the current instance of the type (for example, in a task body) but the rule has been changed to permit its usage as here.

We can also use an anonymous access type for a single variable such as

```
List: access Cell := ... ;
```

An example of the use of an anonymous access type for a function result might be in another animal function such as

```
function Mate_Of(A: access Animal'Class) return access Animal'Class;
```

We could then perhaps write

```
if Mate_Of(Noahs_Ark.Ram) /= Noahs_Ark.Ewe then
  ... -- better get Noah to sort things out
end if;
```

Anonymous access types can also be used in a renaming declaration. This and other detailed points on matters such as accessibility will be discussed in a later paper.

The final important change in access types concerns access to subprogram types. Access to subprogram types were introduced into Ada 95 largely for the implementation of callback. But important applications of such types in other languages (going back to Pascal and even Algol 60) are for mathematical applications such as integration where a function to be manipulated is passed as a parameter. The Ada 83 and Ada 95 approach has always been to say "use generics". But this can be clumsy and so a direct alternative is now provided.

Recall that in Ada 95 we can write

```
type Integrand is access function(X: Float) return Float;
```

```
function Integrate(Fn: Integrand; Lo, Hi: Float) return Float;
```

The idea is that the function Integrate finds the value of the integral of the function passed as parameter Fn between the limits Lo and Hi. This works fine in Ada 95 for simple cases such as where the function is declared at library level. Thus to evaluate

$$\int_0^1 \sqrt{x}\, dx$$

we can write

```
Result := Integrate(Sqrt'Access, 0.0, 1.0);
```

where the function Sqrt is from the library package Ada.Numerics.Elementary_Functions.

However, if the function to be integrated is more elaborate then we run into difficulties in Ada 95 if we attempt to use access to subprogram types. Consider the following example which aims to compute the integral of the expression xy over the square region $0 \le x, y \le 1$.

```
with Integrate;
procedure Main is
  function G(X: Float) return Float is
    function F(Y: Float) return Float is
    begin
      return X*Y;
    end F;
  begin
    return Integrate(F'Access, 0.0, 1.0);      -- illegal in 95
  end G;

  Result: Float;
begin
  Result:= Integrate(G'Access, 0.0, 1.0);      -- illegal in 95
  ...
end Main;
```

But this is illegal in Ada 95 because of the accessibility rules necessary with named access types in order to prevent dangling references. Thus we need to prevent the possibility of storing a pointer to a local subprogram in a global structure. This means that both F'Access and G'Access are illegal in the above.

Note that although we could make the outer function G global so that G'Access would be allowed nevertheless the function F has to be nested inside G in order to gain access to the parameter X of G. It is typical of functions being integrated that they have to have information passed globally – the number of parameters of course is fixed by the profile used by the function Integrate.

The solution in Ada 2005 is to introduce anonymous access to subprogram types by analogy with anonymous access to object types. Thus the function Integrate becomes

```
function Integrate(Fn: access function(X: Float) return Float;
                   Lo, Hi: Float) return Float;
```

Note that the parameter Fn has an anonymous type defined by the profile so that we get a nesting of profiles. This may seem a bit convoluted but is much the same as in Pascal.

The nested example above is now valid and no accessibility problems arise. (The reader will recall that accessibility problems with anonymous access to object types are prevented by a runtime check; in the case of anonymous access to subprogram types the corresponding problems are prevented by

decreeing that the accessibility level is infinite – actually the RM says larger than that of any master which comes to the same thing.)

Anonymous access to subprogram types are also useful in many other applications such as iterators as will be illustrated later.

Note that we can also prefix all access to subprogram types, both named and anonymous, by **constant** and **not null** in the same way as for access to object types.

3.3 Structure, visibility, and limited types

Structure is vital for controlling visibility and thus abstraction. There were huge changes in Ada 95. The whole of the hierarchical child unit mechanism was introduced with both public and private children. It was hoped that this would provide sufficient flexibility for the future.

But one problem has remained. Suppose we have two types where each wishes to refer to the other. Both need to come first! Basically we solve the difficulty by using incomplete types. We might have a drawing package concerning points and lines in a symmetric way. Each line contains a list or array of the points on it and similarly each point contains a list or array of the lines through it. We can imagine that they are both derived from some root type containing printing information such as color. In Ada 95 we might write

```
type Object is abstract tagged
  record
    Its_Color: Color;
    ...
  end record;

type Point;
type Line;
type Acc_Point is access all Point;
type Acc_Line is access all Line;

subtype Index is Integer range 0 .. Max;
type Acc_Line_Array is array (1 .. Max) of Acc_Line;
type Acc_Point_Array is array (1 .. Max) of Acc_Point;

type Point is new Object with
  record
    No_Of_Lines: Index;
    LL: Acc_Line_Array;
    ...
  end record;

type Line is new Object with
  record
    No_Of_Points: Index;
    PP: Acc_Point_Array;
    ...
  end record;
```

This is very crude since it assumes a maximum number Max of points on a line and vice versa and declares the arrays accordingly. The reader can flesh it out more flexibly. Well this is all very well but if the individual types get elaborate and each has a series of operations, we might want to declare them in distinct packages (perhaps child packages of that containing the root type). In Ada 95 we cannot do this because both the incomplete declaration and its completion have to be in the same package.

The net outcome is that we end up with giant cumbersome packages.

What we need therefore is some way of logically enabling the incomplete view and the completion to be in different packages. The elderly might remember that in the 1980 version of Ada the situation was even worse – the completion had to be in the same list of declarations as the incomplete declaration. Ada 83 relaxed this (the so-called Taft Amendment) and permits the private part and body to be treated as one list – the same rule applies in Ada 95. We now go one step further.

Ada 2005 solves the problem by introducing a variation on the with clause – the limited with clause. The idea is that a library package (and subprogram) can refer to another library package that has not yet been declared and can refer to the types in that package but only as if they were incomplete types. Thus we might have a root package Geometry containing the declarations of Object, Max, Index, and so on and then

```
limited with Geometry.Lines;
package Geometry.Points is

  type Acc_Line_Array is array (1 .. Max) of access Lines.Line;

  type Point is new Object with
    record
      No_Of_Lines: Index;
      LL: Acc_Line_Array;
      ...
    end record;

  ...
end Geometry.Points;
```

The package Geometry.Lines is declared in a similar way. Note especially that we are using the anonymous access type facility discussed in Section 3.2 and so we do not even have to declare named access types such as Acc_Line in order to declare Acc_Line_Array.

By writing **limited with** Geometry.Lines; we get access to all the types visible in the specification of Geometry.Lines but as if they were declared as incomplete. In other words we get an incomplete view of the types. We can then do all the things we can normally do with incomplete types such as use them to declare access types. (Of course the implementation checks later that Geometry.Lines does actually have a type Line.)

Not only is the absence of the need for a named type Acc_Line a handy shorthand, it also prevents the proliferation of named access types. If we did want to use a named type Acc_Line in both packages then we would have to declare a distinct type in each package. This is because from the point of view of the package Points, the Acc_Line in Lines would only be an incomplete type (remember each package only has a limited view of the other) and thus would be essentially unusable. The net result would be many named access types and wretched type conversions all over the place.

There are also some related changes to the notation for incomplete types. We can now write

```
type T is tagged;
```

and we are then guaranteed that the full declaration will reveal T to be a tagged type. The advantage is that we also know that, being tagged, objects of the type T will be passed by reference. Consequently we can use the type T for parameters before seeing its full declaration. In the example of points and lines above, since Line is visibly tagged in the package Geometry.Lines we will thus get an incomplete tagged view of Lines.

The introduction of tagged incomplete types clarifies the ability to write

```
type T_Ptr is access all T'Class;
```

This was allowed in Ada 95 even though we had not declared T as tagged at this point. Of course it implied that T would be tagged. In Ada 2005 this is frowned upon since we should now declare that T is tagged incomplete if we wish to declare a class wide access type. For compatibility the old feature has been retained but banished to Annex J for obsolescent features.

Further examples of the use of limited with clauses will be given in a later paper.

Another enhancement in this area is the introduction of private with clauses which overcome a problem with private child packages.

Private child packages were introduced to enable the details of the implementation of part of a system to be decomposed and yet not be visible to the external world. However, it is often convenient to have public packages that use these details but do not make them visible to the user. In Ada 95 a parent or sibling body can have a with clause for a private child. But the specifications cannot. These rules are designed to ensure that information does not leak out via the visible part of a specification. But there is no logical reason why the private part of a package should not have access to a private child. Ada 2005 overcomes this by introducing private with clauses. We can write

```
private package App.Secret_Details is
  type Inner is ...
  ...          -- various operations on Inner etc
end App.Secret_Details;

private with App.Secret_Details;
package App.User_View is

  type Outer is private;
  ...          -- various operations on Outer visible to the user

   -- type Inner is not visible here
private
   -- type Inner is visible here

   type Outer is
     record
       X: Secret_Details.Inner;
       ...
     end record;
  ...
end App.User_View;
```

thus the private part of the public child has access to the type Inner but it is still hidden from the external user.

Note that the public child and private child might have mutually declared types as well in which case they might also wish to use the limited with facility. In this case the public child would have a limited private with clause for the private child written thus

```
limited private with App.Secret_Details;
package App.User_View is ...
```

In the case of a parent package, its specification cannot have a with clause for a child – logically the specification cannot know about the child because the parent must be declared (that is put into the program library) first. Similarly a parent cannot have a private with clause for a private child. But it

can have a limited with clause for any child (thereby breaking the circularity) and in particular it can have a limited private with clause for a private child. So we might also have

```
limited private with App.Secret_Details;
package App is ...
```

The final topic in this section is limited types. The reader will recall that the general idea of a limited type is to restrict the operations that the user can perform on a type to just those provided by the developer of the type and in particular to prevent the user from doing assignment and thus making copies of an object of the type.

However, limited types have never quite come up to expectation both in Ada 83 and Ada 95. Ada 95 brought significant improvements by disentangling the concept of a limited type from a private type but problems have remained.

The key problem is that Ada 95 does not allow the initialization of limited types because of the view that initialization requires assignment and thus copying. A consequence is that we cannot declare constants of a limited type either. Ada 2005 overcomes this problem by allowing initialization by aggregates.

As a simple example, consider

```
type T is limited
  record
    A: Integer;
    B: Boolean;
    C: Float;
  end record;
```

in which the type as a whole is limited but the components are not. If we declare an object of type T in Ada 95 then we have to initialize the components (by assigning to them) individually thus

```
  X: T;
begin
  X.A := 10;  X.B := True;  X.C := 45.7;
```

Not only is this annoying but it is prone to errors as well. If we add a further component D to the record type T then we might forget to initialize it. One of the advantages of aggregates is that we have to supply all the components (allowing automatic so-called full coverage analysis, a key benefit of Ada).

Ada 2005 allows the initialization with aggregates thus

```
X: T := (A => 10,  B => True,  C => 45.7);
```

Technically, Ada 2005 just recognizes properly that initialization is not assignment. Thus we should think of the individual components as being initialized individually *in situ* – an actual aggregated value is not created and then assigned. (Much the same happens when initializing controlled types with an aggregate.)

Sometimes a limited type has components where an initial value cannot be given. This happens with task and protected types. For example

```
protected type Semaphore is ... ;

type PT is
  record
    Guard: Semaphore;
    Count: Integer;
```

```
   Finished: Boolean := False;
end record;
```

Remember that a protected type is inherently limited. This means that the type PT is limited because a type with a limited component is itself limited. It is good practice to explicitly put **limited** on the type PT in such cases but it has been omitted here for illustration. Now we cannot give an explicit initial value for a Semaphore but we would still like to use an aggregate to get the coverage check. In such cases we can use the box symbol <> to mean use the default value for the type (if any). So we can write

```
X: PT := (Guard => <>, Count => 0, Finished => <>);
```

Note that the ability to use <> in an aggregate for a default value is not restricted to the initialization of limited types. It is a new feature applicable to aggregates in general. But, in order to avoid confusion, it is only permitted with named notation.

Limited aggregates are also allowed in other similar contexts where copying is not involved including as actual parameters of mode **in**.

There are also problems with returning results of a limited type from a function. This is overcome in Ada 2005 by the introduction of an extended form of return statement. This will be described in detail in a later paper.

3.4 Tasking and real-time facilities

Unless mentioned otherwise all the changes in this section concern the Real-Time Systems annex.

First, the well-established Ravenscar profile is included in Ada 2005 as directed by WG9. A profile is a mode of operation and is specified by the pragma Profile which defines the particular profile to be used. Thus to ensure that a program conforms to the Ravenscar profile we write

```
pragma Profile(Ravenscar);
```

The purpose of Ravenscar is to restrict the use of many of the tasking facilities so that the effect of the program is predictable. This is very important for real-time safety-critical systems. In the case of Ravenscar the pragma is equivalent to the joint effect of the following pragmas

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;
```

plus a **pragma** Restrictions with a host of arguments such as No_Abort_Statements and No_Dynamic_Priorities.

The pragma Detect_Blocking plus many of the Restrictions identifiers are new to Ada 2005. Further details will be given in a later paper.

Ada 95 allows the priority of a task to be changed but does not permit the ceiling priority of a protected object to be changed. This is rectified in Ada 2005 by the introduction of an attribute Priority for protected objects and the ability to change it by a simple assignment such as

```
My_PO'Priority := P;
```

inside a protected operation of the object My_PO. The change takes effect at the end of the protected operation.

The monitoring and control of execution time naturally are important for real-time programs. Ada 2005 includes packages for three different aspects of this

Ada.Execution_Time – this is the root package and enables the monitoring of execution time of individual tasks.

Ada.Execution_Time.Timers – this provides facilities for defining and enabling timers and for establishing a handler which is called by the run time system when the execution time of the task reaches a given value.

Ada.Execution_Time.Group_Budgets – this allows several tasks to share a budget and provides means whereby action can be taken when the budget expires.

The execution time of a task or CPU time, as it is commonly called, is the time spent by the system executing the task and services on its behalf. CPU times are represented by the private type CPU_Time. The CPU time of a particular task is obtained by calling the following function Clock in the package Ada.Execution_Time

```
function Clock(T: Task_Id := Current_Task) return CPU_Time;
```

A value of type CPU_Time can be converted to a Seconds_Count plus residual Time_Span by a procedure Split similar to that in the package Ada.Real_Time. Incidentally we are guaranteed that the granularity of CPU times is no greater than one millisecond and that the range is at least 50 years.

In order to find out when a task reaches a particular CPU time we use the facilities of the child package Ada.Execution_Time.Timers. This includes a discriminated type Timer and a type Handler thus

```
type Timer(T: not null access constant Task_Id) is tagged limited private;
type Handler is access protected procedure (TM: in out Timer);
```

Note how the access discriminant illustrates the use of both **not null** and **constant**.

We can then set the timer to expire at some absolute time by

```
Set_Handler(My_Timer, Time_Limit, My_Handler'Access);
```

and then when the CPU time of the task reaches Time_Limit (of type CPU_Time), the protected procedure My_Handler is executed. Note how the timer object incorporates the information regarding the task concerned using an access discriminant and that this is passed to the handler via its parameter. Another version of Set_Handler enables the timer to be triggered after a given interval (of type Time_Span).

In order to program various aperiodic servers it is necessary for tasks to share a CPU budget. This can be done using the child package Ada.Execution_Time.Group_Budgets. In this case we have

```
type Group Budget is tagged limited private;
type Handler is access protected procedure (GB: in out Group_Budget);
```

The type Group_Budget both identifies the group of tasks it belongs to and the size of the budget. Various subprograms enable tasks to be added to and removed from a group budget. Other procedures enable the budget to be set and replenished.

A procedure Set_Handler associates a particular handler with a budget.

```
Set_Handler(GB => My_Group_Budget, Handler => My_Handler'Access);
```

When the group budget expires the associated protected procedure is executed.

A somewhat related topic is that of low level timing events. The facilities are provided by the package Ada.Real_Time.Timing_Events. In this case we have

```
type Timing_Event is tagged limited private;
type Timing_Event_Handler is access protected procedure (Event: in out Timing_Event);
```

The idea here is that a protected procedure can be nominated to be executed at some time in the future. Thus to ring a pinger when our egg is boiled after four minutes we might have a protected procedure

```
protected body Egg is
  procedure Is_Done(Event: in out Timing_Event) is
  begin
    Ring_The_Pinger;
  end Is_Done;
end Egg;
```

and then

```
Egg_Done: Timing_Event;
Four_Min: Time_Span := Minutes(4);
...
Put_Egg_In_Water;
Set_Handler(Event => Egg_Done, In_Time => Four_Min, Handler => Egg.Is_Done'Access);
--  now read newspaper whilst waiting for egg
```

This facility is of course very low level and does not involve Ada tasks at all. Note that we can set the event to occur at some absolute time as well as at a relative time as above. Incidentally, the function Minutes is a new function added to the parent package Ada.Real_Time. Otherwise we would have had to write something revolting such as 4*60*Milliseconds(1000). A similar function Seconds has also been added.

There is a minor flaw in the above example. If we are interrupted by the telephone between putting the egg in the water and setting the handler then our egg will be overdone. We will see how to cure this in a later paper.

Readers will recall the old problem of how tasks can have a silent death. If something in a task goes wrong in Ada 95 and an exception is raised which is not handled by the task, then it is propagated into thin air and just vanishes. It was always deemed impossible for the exception to be handled by the enclosing unit because of the inherent asynchronous nature of the event.

This is overcome in Ada 2005 by the package Ada.Task_Termination which provides facilities for associating a protected procedure with a task. The protected procedure is invoked when the task terminates with an indication of the reason. Thus we might declare a protected object Grim_Reaper

```
protected Grim_Reaper is
  procedure Last_Gasp(C: Cause_Of_Termination; T: Task_Id; X: Exception_Occurrence);
end Grim_Reaper;
```

We can then nominate Last_Gasp as the protected procedure to be called when task T dies by

```
Set_Specific_Handler(T'Identity, Grim_Reaper.Last_Gasp'Access);
```

The body of the protected procedure Last_Gasp might then output various diagnostic messages

```
procedure Last_Gasp(C: Cause_Of_Termination; T: Task_Id; X: Exception_Occurrence) is
begin
  case C is
    when Normal => null;
    when Abnormal =>
      Put("Something nasty happened"); ...
    when Unhandled_Exception =>
      Put("Unhandled exception occurred"); ...
```

```
   end case;
end Last_Gasp;
```

There are three possible reasons for termination, it could be normal, abnormal, or caused by an unhandled exception. In the last case the parameter X gives details of the exception occurrence.

Another area of increased flexibility in Ada 2005 is that of task dispatching policies. In Ada 95, the only predefined policy is FIFO_Within_Priorities although other policies are permitted. Ada 2005 provides further pragmas, policies and packages which facilitate many different mechanisms such as non-preemption within priorities, the familiar Round Robin using timeslicing, and the more recently acclaimed Earliest Deadline First (EDF) policy. Moreover, it is possible to mix different policies according to priority level within a partition.

Various facilities are provided by the package Ada.Dispatching plus two child packages

Ada.Dispatching – this is the root package and simply declares an exception Dispatching_Policy_Error.

Ada.Dispatching.Round_Robin – this enables the setting of the time quanta for time slicing within one or more priority levels.

Ada.Dispatching.EDF – this enables the setting of the deadlines for various tasks.

A policy can be selected for a whole partition by one of

```
pragma Task_Dispatching_Policy(Non_Preemptive_FIFO_Within_Priorities);

pragma Task_Dispatching_Policy(Round_Robin_Within_Priorities);

pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

In order to mix different policies across different priority levels we use the pragma Priority_Specific_Dispatching with various policy identifiers thus

```
pragma Priority_Specific_Dispatching(Round_Robin_Within_Priorities, 1, 1);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 10);
pragma Priority_Specific_Dispatching(FIFO_Within_Priorities, 11, 24);
```

This sets Round Robin at priority level 1, EDF at levels 2 to 10, and FIFO at levels 11 to 24.

The final topic in this section concerns the core language and not the Real-Time Systems annex. Ada 2005 introduces a means whereby object oriented and real-time features can be closely linked together through inheritance.

Recall from Section 3.1 that we can declare an interface to be limited thus

```
type LI is limited interface;
```

We can also declare an interface to be synchronized, task, or protected thus

```
type SI is synchronized interface;
type TI is task interface;
type PI is protected interface;
```

A task interface or protected interface has to be implemented by a task type or protected type respectively. However, a synchronized interface can be implemented by either a task type or a protected type. These interfaces can also be composed with certain restrictions. Detailed examples will be given in a later paper.

3.5 Exceptions, numerics, generics etc

As well as the major features discussed above there are also a number of improvements in various other areas.

There are two small changes concerning exceptions. One is that we can give a message with a raise statement, thus

```
raise Some_Error with "A message";
```

This is a lot neater than having to write (as in Ada 95)

```
Ada.Exceptions.Raise_Exception(Some_Error'Identity, "A message");
```

The other change concerns the detection of a null exception occurrence which might be useful in a package analysing a log of exceptions. The problem is that exception occurrences are of a limited private type and so we cannot compare an occurrence with Null_Occurrence to see if they are equal. In Ada 95 applying the function Exception_Identity to a null occurrence unhelpfully raises Constraint_Error. This has been changed in Ada 2005 to return Null_Id so that we can now write

```
procedure Process_Ex(X: Exception_Occurrence) is
begin
  if Exception_Identity(X) = Null_Id then
    -- process the case of a Null_Occurrence
  ...
end Process_Ex;
```

Ada 95 introduced modular types which are of course unsigned integers. However it has in certain cases proved very difficult to get unsigned integers and signed integers to work together. This is a trivial matter in fragile languages such as C but in Ada the type model has proved obstructive. The basic problem is converting a value of a signed type which happens to be negative to an unsigned type. Thus suppose we want to add a signed offset to an unsigned address value, we might have

```
type Offset_Type is range –(2**31) .. 2**31–1;
type Address_Type is mod 2**32;

Offset: Offset_Type;
Address: Address_Type;
```

We cannot just add Offset to Address because they are of different types. If we convert the Offset to the address type then we might get Constraint_Error and so on. The solution in Ada 2005 is to use a new functional attribute S'Mod which applies to any modular subtype S and converts a universal integer value to the modular type using the corresponding mathematical mod operation. So we can now write

```
Address := Address + Address_Type'Mod(Offset);
```

Another new attribute is Machine_Rounding. This enables high-performance conversions from floating point types to integer types when the exact rounding does not matter.

The third numeric change concerns fixed point types. It was common practice for some Ada 83 programs to define their own multiply and divide operations, perhaps to obtain saturation arithmetic. These programs ran afoul of the Ada 95 rules that introduced universal fixed operations and resulted in ambiguities. Without going into details, this problem has been fixed in Ada 2005 so that user-defined operations can now be used.

Ada 2005 has several new pragmas. The first is

```
pragma Unsuppress(Identifier);
```

where the identifier is that of a check such as Range_Check. The general idea is to ensure that checks are performed in a declarative region irrespective of the use of a corresponding pragma Suppress. Thus we might have a type My_Int that behaves as a saturated type. Writing

```
function "*" (Left, Right: My_Int) return My_Int is
  pragma Unsuppress(Overflow_Check);
begin
  return Integer(Left) * Integer(Right);
exception
  when Constraint_Error =>
    if (Left>0 and Right>0) or (Left<0 and Right<0) then
      return My_Int'Last;
    else
      return My_Int'First;
    end if;
end "*";
```

ensures that the code always works as intended even if checks are suppressed in the program as a whole. Incidentally the On parameter of pragma Suppress which never worked well has been banished to Annex J.

Many implementations of Ada 95 support a pragma Assert and this is now consolidated into Ada 2005. The general idea is that we can write pragmas such as

```
pragma Assert(X >50);

pragma Assert(not Buffer_Full, "buffer is full");
```

The first parameter is a Boolean expression and the second optional parameter is a string. If at the point of the pragma at execution time, the expression is False then action can be taken. The action is controlled by another pragma Assertion_Policy which can switch the assertion mechanism on and off by one of

```
pragma Assertion_Policy(Check);

pragma Assertion_Policy(Ignore);
```

If the policy is to check then the exception Assertion_Error is raised with the message, if any. This exception is declared in the predefined package Ada.Assertions. There are some other facilities as well.

The pragma No_Return is also related to exceptions. It can be applied to a procedure (not to a function) and indicates that the procedure never returns normally but only by propagating an exception. Thus we might have

```
procedure Fatal_Error(Message: in String);
pragma No_Return(Fatal_Error);
```

And now whenever we call Fatal_Error the compiler is assured that control is not returned and this might enable some optimization or better diagnostic messages.

Note that this pragma applies to the predefined procedure Ada.Exceptions.Raise_Exception.

Another new pragma is Preelaborable_Initialization. This is used with private types and indicates that the full type will have preelaborable initialization. A number of examples occur with the predefined packages such as

```
pragma Preelaborable_Initialization(Controlled);
```

in Ada.Finalization.

Finally, there is the pragma Unchecked_Union. This is useful for interfacing to programs written in C that use the concept of unions. Unions in C correspond to variant types in Ada but do not store

any discriminant which is entirely in the mind of the C programmer. The pragma enables a C union to be mapped to an Ada variant record type by omitting the storage for the discriminant.

If the C program has

```
union {
  double spvalue;
  struct {
    int length;
    double* first;
    } mpvalue;
} number;
```

then this can be mapped in the Ada program by

```
type Number(Kind: Precision) is
  record
    case Kind is
      when Single_Precision =>
        SP_Value: Long_Float;
      when Multiple_Precision =>
        MP_Value_Length: Integer;
        MP_Value_First: access Long_Float;
    end case;
  end record;
pragma Unchecked_Union(Number);
```

One problem with pragmas (and attributes) is that many implementations have added implementation defined ones (as they are indeed permitted to do). However, this can impede portability from one implementation to another. To overcome this there are further Restrictions identifiers so we can write

```
pragma Restrictions(No_Implementation_Pragmas, No_Implementation_Attributes);
```

Observe that one of the goals of Ada 2005 has been to standardize as many of the implementation defined attributes and pragmas as possible.

Readers might care to consider the paradox that GNAT has an (implementation-defined) restrictions identifier No_Implementation_Restrictions.

Another new restrictions identifier prevents us from inadvertently using features in Annex J thus

```
pragma Restrictions(No_Obsolescent_Features);
```

Similarly we can use the restrictions identifier No_Dependence to state that a program does not depend on a given library unit. Thus we might write

```
pragma Restrictions(No_Dependence => Ada.Command_Line);
```

Note that the unit mentioned might be a predefined library unit as in the above example but it can also be used with any library unit.

The final new general feature concerns formal generic package parameters. Ada 95 introduced the ability to have formal packages as parameters of generic units. This greatly reduced the need for long generic parameter lists since the formal package encapsulated them.

Sometimes it is necessary for a generic unit to have two (or more) formal packages. When this happens it is often the case that some of the actual parameters of one formal package must be

identical to those of the other. In order to permit this there are two forms of generic parameters. One possibility is

```
generic
  with package P is new Q(<>);
package Gen is ...
```

and then the package Gen can be instantiated with any package that is an instantiation of Q. On the other hand we can have

```
generic
  with package R is new S(P1, P2, ... );
package Gen is ...
```

and then the package Gen can only be instantiated with a package that is an instantiation of S with the given actual parameters P1, P2 etc.

These mechanisms are often used together as in

```
generic
  with package P is new Q(<>);
  with package R is new S(P.F1);
package Gen is ...
```

This ensures that the instantiation of S has the same actual parameter (assumed only one in this example) as the parameter F1 of Q used in the instantiation of Q to create the actual package corresponding to P.

There is an example of this in one of the packages for vectors and matrices in ISO/IEC 13813 which is now incorporated into Ada 2005 (see Section 3.6). The generic package for complex arrays has two package parameters. One is the corresponding package for real arrays and the other is the package Generic_Complex_Types from the existing Numerics annex. Both of these packages have a floating type as their single formal parameter and it is important that both instantiations use the same floating type (eg both Float and not one Float and one Long_Float) otherwise a terrible mess will occur. This is assured by writing (using some abbreviations)

```
with ... ;
generic
  with package Real_Arrays is new Generic_Real_Arrays(<>);
  with package Complex_Types is new Generic_Complex_Types(Real_Arrays.Real);
package Generic_Complex_Arrays is ...
```

Well this works fine in simple cases (the reader may wonder whether this example is simple anyway) but in more elaborate situations it is a pain. The trouble is that we have to give all the parameters for the formal package or none at all in Ada 95.

Ada 2005 permits only some of the parameters to be specified, and any not specified can be indicated using the box. So we can write any of

```
with package Q is new R(P1, P2, F3 => <>);
with package Q is new R(P1, others => <>);
with package Q is new R(F1 => <>, F2 => P2, F3 => P3);
```

Note that the existing form (<>) is now deemed to be a shorthand for (**others** => <>). As with aggregates, the form <> is only permitted with named notation.

Examples using this new facility will be given in a later paper.

3.6 Standard library

There are significant improvements to the standard library in Ada 2005. One of the strengths of Java is the huge library that comes with it. Ada has tended to take the esoteric view that it is a language for constructing programs from components and has in the past rather assumed that the components would spring up by magic from the user community. There has also perhaps been a reluctance to specify standard components in case that preempted the development of better ones. However, it is now recognized that standardizing useful stuff is a good thing. And moreover, secondary ISO standards are not very helpful because they are almost invisible. Ada 95 added quite a lot to the predefined library and Ada 2005 adds more.

First, there are packages for manipulating vectors and matrices already mentioned in Section 3.5 when discussing formal package parameters. There are two packages, Ada.Numerics.Generic_Real_Arrays for real vectors and matrices and Ada.Numerics.Generic_Complex_Arrays for complex vectors and matrices. They can be instantiated according to the underlying floating point type used. There are also nongeneric versions as usual.

These packages export types for declaring vectors and matrices and many operations for manipulating them. Thus if we have an expression in mathematical notation such as

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{z}$$

where $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ are vectors and $\boldsymbol{A}$ is a square matrix, then this calculation can be simply programmed as

```
X, Y, Z: Real_Vector(1 .. N);
A: Real_Matrix(1 .. N, 1 .. N);
...
Y := A * X + Z;
```

and the appropriate operations will be invoked. The packages also include subprograms for the most useful linear algebra computations, namely, the solution of linear equations, matrix inversion and determinant evaluation, plus the determination of eigenvalues and eigenvectors for symmetric matrices (Hermitian in the complex case). Thus to determine X given Y, Z and A in the above example we can write

```
X := Solve(A, Y – Z);
```

It should not be thought that these Ada packages in any way compete with the very comprehensive BLAS (Basic Linear Algebra Subprograms). The purpose of the Ada packages is to provide simple implementations of very commonly used algorithms (perhaps for small embedded systems or for prototyping) and to provide a solid framework for developing bindings to the BLAS for more demanding situations. Incidentally, they are in the Numerics annex.

Another (but very trivial) change to the Numerics annex is that nongeneric versions of Ada.Text_IO.Complex_IO have been added in line with the standard principle of providing nongeneric versions of generic predefined packages for convenience. Their omission from Ada 95 was an oversight.

There is a new predefined package in Annex A for accessing tree-structured file systems. The scope is perhaps indicated by this fragment of its specification

```
with ...
package Ada.Directories is
  -- Directory and file operations
  function Current_Directory return String;
  procedure Set_Directory(Directory: in String);
  ...
```

```
   -- File and directory name operations
   function Full_Name(Name: in String) return String;
   function Simple_Name(Name: in String) return String;
   ...
   -- File and directory queries
   type File_Kind is (Directory, Ordinary_File, Special_File);
   type File_Size is range 0 .. implementation-defined;
   function Exists(Name: in String) return Boolean;
   ...
   -- Directory searching
   type Directory_Entry_Type is limited private;
   type Filter_Type is array (File_Kind) of Boolean;
   ...
   -- Operations on directory entries
   ...
end Ada.Directories;
```

The package contains facilities which will be useful on any Unix or Windows system. However, it has to be recognized that like Ada.Command_Line it might not be supportable on every environment.

There is also a package Ada.Environment_Variables for accessing the environment variables that occur in most operating systems.

A number of additional subprograms have been added to the existing string handling packages. There are several problems with the Ada 95 packages. One is that conversion between bounded and unbounded strings and the raw type String is required rather a lot and is both ugly and inefficient. For example, searching only part of a bounded or unbounded string can only be done by converting it to a String and then searching the appropriate slice (or by making a truncated copy first).

In brief the additional subprograms are as follows

- Three further versions of function Index with an additional parameter From indicating the start of the search are added to each of Strings.Fixed, Strings.Bounded and Strings.Unbounded.
- A further version of function Index_Non_Blank is similarly added to all three packages.
- A procedure Set_Bounded_String with similar behaviour to the function To_Bounded_String is added to Strings.Bounded. This avoids the overhead of using a function. A similar procedure Set_Unbounded_String is added to Strings.Unbounded.
- A function and procedure Bounded_Slice are added to Strings.Bounded. These avoid conversions from type String. A similar function and procedure Unbounded_Slice are added to Strings.Unbounded.

As well as these additions there is a new package Ada.Text_IO.Unbounded_IO for the input and output of unbounded strings. This again avoids unnecessary conversion to the type String. Similarly, there is a generic package Ada.Text_IO.Bounded_IO; this is generic because the package Strings.Bounded has an inner generic package which is parameterized by the maximum string length.

Finally, two functions Get_Line are added to Ada.Text_IO itself. These avoid difficulties with the length of the string which occurs with the existing procedures Get_Line.

In Ada 83, program identifiers used the 7-bit ASCII set. In Ada 95 this was extended to the 8-bit Latin-1 set. In Ada 2005 this is extended yet again to the entire ISO/IEC 10646:2003 character

repertoire. This means that identifiers can now use Cyrillic and Greek characters. Thus we could extend the animal example by

```
_т____: access Pig renames Napoleon;
Πεγασυς: Horse;
```

In order to encourage us to write our mathematical programs nicely the additional constant

```
π: constant := Pi;
```

has been added to the package Ada.Numerics in Ada 2005.

In a similar way types Wide_String and Wide_Character were added to Ada 95. In Ada 2005 this process is also extended and a set of wide-wide types and packages for 32-bit characters are added. Thus we have types Wide_Wide_Character and Wide_Wide_String and so on.

A major addition to the predefined library is the package Ada.Containers and its children plus some auxiliary child functions of Ada.Strings. These are very important and considerable additions to the predefined capability of Ada and bring the best in standard data structure manipulation to the fingers of every Ada programmer. The scope is perhaps best illustrated by listing the units involved.

Ada.Containers – this is the root package and just declares types Hash_Type and Count_Type which are an implementation-defined modular and integer type respectively.

Ada.Strings.Hash – this function hashes a string into the type Hash_Type. There are also versions for bounded and unbounded strrings.

Ada.Containers.Vectors – this is a generic package with parameters giving the index type and element type of a vector plus "=" for the element type. This package declares types and operations for manipulating vectors. (These are vectors in the sense of flexible arrays and not the mathematical vectors used for linear algebra as in the vectors and matrices packages mentioned earlier.) As well as subprograms for adding, moving and removing elements there are also generic subprograms for searching, sorting and iterating over vectors.

Ada.Containers.Doubly_Linked_Lists – this is a generic package with parameters giving the element type and "=" for the element type. This package declares types and operations for manipulating doubly-linked lists. It has similar functionality to the vectors package. Thus, as well as subprograms for adding, moving and removing elements there are also generic subprograms for searching, sorting and iterating over lists.

Ada.Containers.Hashed_Maps – this is a generic package with parameters giving a key type and an element type plus a hash function for the key, a function to test for equality between keys and "=" for the element type. It declares types and operations for manipulating hashed maps.

Ada.Containers.Ordered_Maps – this is a similar generic package for ordered maps with parameters giving a key type and an element type and "<" for the key type and "=" for the element type.

Ada.Containers.Hashed_Sets – this is a generic package with parameters giving the element type plus a hash function for the elements and a function to test for equality between elements. It declares types and operations for manipulating hashed sets.

Ada.Containers.Ordered_Sets – this is a similar generic package for ordered sets with parameters giving the element type and "<" and "=" for the element type.

There are then another six packages with similar functionality but for indefinite types with corresponding names such as Ada.Containers.Indefinite_Vectors.

Ada.Containers.Generic_Array_Sort – this is a generic procedure for sorting arrays. The generic parameters give the index type, the element type, the array type and "<" for the element type. The array type is unconstrained.

Finally there is a very similar generic procedure Ada.Containers.Generic_Constrained_Array_Sort but for constrained array types.

It is hoped that the above list gives a flavour of the capability of the package Containers. Some examples of the use of the facilities will be given in a later paper.

Finally, there are further packages for manipulating times (that is of type Ada.Calendar.Time and not Ada.Real_Time.Time and thus more appropriate in a discussion of the predefined library than the real-time features). The package Ada.Calendar has a number of obvious omissions and in order to rectify this the following packages are added.

Ada.Calendar.Time_Zones – this declares a type Time_Offset describing in minutes the difference between two time zones and a function UTC_Time_Offset which given a time returns the difference between the time zone of Calendar at that time and UTC (Coordinated Universal Time which is close to Greenwich Mean Time). It also has an exception which is raised if the time zone of Calendar is not known (maybe the clock is broken).

Ada.Calendar.Arithmetic – this declares various types and operations for coping with leap seconds.

Ada.Calendar.Formatting – this declares further types and operations for dealing with formatting and related matters.

Most of the new calendar features are clearly only for the chronological addict but the need for them does illustrate that this is a tricky area. However, a feature that all will appreciate is that the package Ada.Calendar.Formatting includes the following declarations

```
type Day_Name is (Monday, Tuesday, Wednesday,
                  Thursday, Friday, Saturday, Sunday);

function Day_Of_Week(Date: Time) return Day_Name;
```

There is also a small change in the parent package Ada.Calendar itself. The subtype Year_Number is now

```
subtype Year_Number is Integer range 1901 .. 2399;
```

This reveals confidence in the future of Ada by adding another three hundred years to the range of dates.

4 Conclusions

This overview of Ada 2005 should have given the reader an appreciation of the important new features in Ada 2005. Some quite promising features failed to be included partly because the need for them was not clear and also because a conclusive design proved elusive. We might think of them as Forthcoming Attractions for any further revision!

Some esoteric topics have been omitted in this overview; they concern features such as: streams, object factory functions, the partition control system in distributed systems, partition elaboration policy for high integrity systems, a subtlety regarding overload resolution, the title of Annex H, quirks of access subtypes, rules for pragma Pure, and the classification of various units as pure or preelaborable.

Further papers will expand on the six major topics of this overview in more detail.

It is worth briefly reviewing the guidelines (see Section 2 above) to see whether Ada 2005 meets them. Certainly the Ravenscar profile has been added and the problem of mutually dependent types across packages has been solved.

The group A items were about real-time and high-integrity, static error checking and interfacing. Clearly there are major improvements in the real-time area. And high-integrity and static error checking are addressed by features such as the **overriding** prefix, various pragmas such as Unsuppress and Assert and additional Restrictions identifiers. Better interfacing is provided by the pragma Unchecked_Union and the Mod attribute.

The group B items were about improvements to the OO model, the need for a Java-like interface feature and better interfacing to other OO languages. Major improvements to the OO model are brought by the prefixed (Obj.Op) notation and more flexible access types. The Java-like interface feature has been added and this provides better interfacing.

The final direct instruction was to incorporate the vectors and matrices stuff and this has been done. There are also many other improvements to the predefined library as we have seen.

It seems clear from this brief check that indeed Ada 2005 does meet the objectives set for it.

Finally, I need to thank all those who have helped in the preparation of this paper. First I must acknowledge the financial support of Ada-Europe and the Ada Resource Association. And then I must thank those who reviewed earlier versions. There are almost too many to name, but I must give special thanks to Randy Brukardt, Pascal Leroy and Tucker Taft of the ARG, to my colleagues on the UK Ada Panel (BSI/IST/5/–/9), and to James Moore of WG9. I am especially grateful for a brilliant suggestion of Randy Brukardt which must be preserved for the pleasure of future generations. He suggests that this document when complete be called the Ada Language Enhancement Guide. This means that if combined with the final Ada Reference Manual, the whole document can then be referred to as the ARM and ALEG. Thanks Randy.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] ISO/IEC 8652:1995/COR 1:2001, *Ada Reference Manual – Technical Corrigendum 1*.

[3] S. T. Taft et al (eds) (2001) *Consolidated Ada Reference Manual*, LNCS 2219, Springer-Verlag.

[4] ISO/IEC TR 24718:2004 (2004) *Guide for the use of the Ada Ravenscar Profile in high integrity systems*. This is based on University of York Technical Report YCS-2003-348 (2003).

[5] ISO/IEC 13813:1997 (1997) *Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types)*.

[6] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

Rationale for Ada 2005: 1 Object oriented model

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various important improvements to the object oriented model for Ada 2005.

First an alternative more traditional prefixed notation for calling operations has been introduced. A major improvement is that Java-like interfaces are introduced thereby permitting simple multiple inheritance; null procedures have also been introduced as a category of operation. Greater general flexibility is provided by allowing type extension at a more nested level than that of the parent.

There are also explicit features for overcoming nasty bugs which arise from confusion between overloading and overriding.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 1, March 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] identifies very large complex systems as a major application area for Ada. It says

"The main purpose of the Amendment is to address identified problems in Ada that are interfering with Ada's usage or adoption, especially in its major application areas (such as high-reliability, long-lived real-time and/or embedded applications and very large complex systems). The resulting changes may range from relatively minor, to more substantial."

Object oriented techniques are of course important in very large systems in providing flexibility and extensibility. The document later asks the ARG to pay particular attention to

> Improvements that will remedy shortcomings in Ada. It cites in particular improvements in OO features, specifically, adding a Java-like interface feature and improved interfacing to other OO languages.

Ada 2005 does indeed make many improvements in the object oriented area. The following Ada Issues cover the relevant changes and are described in detail in this paper:

218 Accidental overloading when overriding

251 Abstract interfaces to provide multiple inheritance

252 Object.Operator notation

260 Abstract formal subprograms & dispatching constructors

284 New reserved words

310 Ignore abstract nondispatching ops during overloading

344 Allow nested type extensions

348 Null procedures

391 Functions with controlling results on null extension

396 The "no hidden interfaces" rule

400 Wide and wide-wide images

401 Terminology for interfaces

405 Progenitors and Ada.Tags

407 Terminology and semantics for prefix names

411 Equality for types derived from interfaces

417 Lower bound of functions in Ada.Tags etc

419 Limitedness of derived types

These changes can be grouped as follows.

First we discuss the fact that Ada 2005 has three new reserved words, **interface**, **overriding**, and **synchronized**. It so happens that these are all used in different aspects of the OO model and so we discuss them in this paper (284).

Then there is the introduction of the Obj.Op or prefixed notation used by many other languages (252, 407). This should make Ada easier to use, improve its image, and improve interfacing to other languages.

A huge improvement is the addition of Java-like interfaces which allow proper multiple inheritance (251, 396, 401, 411, 419). A related change is the introduction of null procedures as a category of operation somewhat like abstract operations (348).

Type extension is now permitted at a more nested level than that of the parent type (344). An important consequence is that controlled types no longer need to be declared at library level.

An interesting development is the introduction of generic functions for the dynamic creation of objects of any type of a class (260, 400, 405, 417). These are sometimes called object factory functions or just object factories.

Additional syntax permits the user to say whether an operation is expected to be overriding or not (218). This detects certain unfortunate errors during compilation which otherwise can be difficult to find at execution time. A small change to the overriding rules is that a function with a controlling result does not "go abstract" if an extension is in fact null (391). Finally, we discuss a minor but useful change to the overloading rules; in a sense this is not about OO at all since it concerns the rules for nondispatching operations but it is convenient to discuss it here (310).

There are in fact many other OO related improvements in Ada 2005 concerning matters such as access types, visibility, and generics. They will be described in later papers.

2 Reserved words

Ada 2005 has three further reserved words namely **interface**, **overriding**, and **synchronized**. Readers may recall that Ada 95 had six more reserved words than Ada 83 and the fact that this meant that some programs were incompatible and thus had to be rewritten loomed large in the minds of many commentators.

When new syntax for the introduction of interfaces was being discussed it was strongly felt that incompatibilities should be avoided and that any new syntax words should be unreserved. It was also noted that Interface was a popular identifier and that making it a reserved word would cause many programs to have to be rewritten.

However, it was soon realised that treating Interface as unreserved would have permitted sequences such as

```
type T is interface;
subtype Interface is T;
```

in which Interface is a subtype of the interface T. This would have been total madness. Some reviewers also had memories of PL/I in which words such as IF were not reserved so that one could write IF IF ... where the first IF is a syntax word and the second is a user identifier.

Accordingly it was decided that the new words would have to be reserved. No sensible alternative to **interface** could be thought of although it would be irritating for users who had packages called Interface – actually a brief survey revealed that most such packages had longer names such as Radar_Interface so that the problem was more apparent than real. The other new reserved words **overriding** and **synchronized** clearly present less of a problem since they are less likely to have been used as identifiers.

3 The prefixed notation

As mentioned in the Introduction, the Ada 95 object oriented model has been criticized for not being really OO since the notation for applying a subprogram (method) to an object emphasizes the subprogram and not the object. Thus given

```
package P is
  type T is tagged ... ;

  procedure Op(X: T; ... );
  ...
end P;
```

then we usually have to write

```
P.Op(Y, ... );                    -- subprogram first
```

in order to apply the operation to an object Y of type T whereas an OO person would expect to write

```
Y.Op( ... );                      -- object first
```

Some hard line OO languages such as Smalltalk take the view that everything is an object and that all activities are operations upon some object. Thus adding 2 and 3 can be seen as sending a message to 2 instructing 3 to be added to it. This is clearly an extreme view.

Older languages take the view that subprograms are dominant and that they act upon parameters which might be raw numbers such as 2 or denote objects such as a circle. Ada 95 primarily takes this view which reflects its Pascal foundation over 20 years ago. Thus if Area is a function which returns the area of a circle then we write

```
A := Area(A_Circle);
```

However, when we come to tasks and protected objects Ada takes the OO view in which the identity of the object comes first. Thus given a task Actor with an entry Start we call the entry by writing

```
Actor.Start( ... );
```

So Ada 95 already uses the object notation although it only applies to concurrent objects such as tasks. Other objects and, in particular, objects of tagged types have to use the subprogram notation.

A major irritation of the subprogram notation is that it is usually necessary to name the package containing the declaration of the subprogram thus

```
P.Op(Y, ... );                    -- package P mentioned
```

There are two situations when P need not be mentioned – one is where the procedure call is actually inside the package P, the other is where we have a use clause for P (and even that sometimes does not give the required visibility). But these are special cases.

In Ada 2005 we can replace P.Op(Y, ...); by the so-called prefixed notation

```
Y.Op( ... );                      -- package P never mentioned
```

provided that

- T is a tagged type,
- Op is a primitive (dispatching) or class wide operation of T,
- Y is the first parameter of Op.

The reason there is never any need to mention the package is that, by starting from the object, we can identify its type and thus the primitive operations of the type. Note that a class wide operation can be called in this way only if it is declared at the same place as the primitive operations of T (or one of its ancestors).

There are many advantages of the prefixed notation as we shall see but perhaps the most important is ease of maintenance from not having to mention the package containing the declaration of the operation. Having to name the package is often tricky because in complicated situations involving several levels of inheritance it may not be obvious where the operation is declared. This happens especially when operations are declared implicitly and when class-wide operations are involved. Moreover if we change the structure for some reason then operations might move.

As a simple example consider a hierarchy of plane geometrical object types. All objects have a position given by the two coordinates *x* and *y* (this is the position of the centre of gravity of the object). There will be other specific properties according to the type such as the radius of a circle. In addition there might be general properties such as the area of the object, its distance from the origin and moment of inertia about it centre.

There are a number of ways in which such a hierarchy might be structured. We might have a package declaring a root abstract type and then another package with several derived types.

```
package Root is
  type Object is abstract tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;

  function Area(O: Object) return Float is abstract;
  function MI(O: Object) return Float is abstract;
  function Distance(O: Object) return Float;
end Root;

package body Root is
  function Distance(O: Object) return Float is
  begin
```

```
    return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
  end Distance;
end Root;
```

This package declares the root type and two abstract operations Area and MI (moment of inertia) and a concrete operation Distance. We might then have

```
with Root;
package Shapes is
  type Circle is new Root.Object with
    record
      Radius: Float;
    end record;

  function Area(C: Circle) return Float;
  function MI(C: Circle) return Float;

  type Triangle is new Root.Object with
    record
      A, B, C: Float;                -- lengths of sides
    end record;

  function Area(T: Triangle) return Float;
  function MI(T: Triangle) return Float;

-- and so on for other types such as Square

end Shapes;
```

(In the following discussion we will assume that use clauses are not being used. This is quite realistic because many projects forbid use clauses.)

Having declared some objects such as A_Circle and A_Triangle we can then apply the operations Area, Distance, and MI. In Ada 95 we write

```
A := Shapes.Area(A_Circle);
D := Shapes.Distance(A_Triangle);
M := Shapes.MI(A_Square);
```

Observe that the operation Distance is inherited and so is implicitly declared in the package Shapes for all types even though there is no mention of it in the text of the package Shapes. However, if we were using Ada 2005 and the prefixed notation then we could simply write

```
A := A_Circle.Area;
D := A_Triangle.Distance;
M := A_Square.MI;
```

and there is no mention of the package Shapes at all.

A clever friend then points out that by its nature Distance is the same for all types so it would be safer to avoid the risk of it getting changed by making it class wide. So we change the declaration of Distance in the package Root thus

```
  function Distance(O: Object'Class) return Float;
```

and recompile our program. But the Ada 95 version won't recompile. Why? Because class wide operations are not inherited. So there is only one function Distance and it is declared in the package Root. So all our calls of Distance have to be changed to

```
D := Root.Distance(A_Triangle);
```

However, if we had been using the prefixed notation then there would have been nothing to change.

Our manager might then read about the virtues of child packages and tell us to restructure the whole thing as follows

```
package Geometry is
  type Object is abstract ...

  ... -- functions Area, MI, Distance
end Geometry;

package Geometry.Circles is
  type Circle is new Object with
    record
      Radius: Float;
    end record;

  ... -- functions Area, MI
end Geometry.Circles;

package Geometry.Triangles is
  type Triangle is new Object with
    record
      A, B, C: Float;
    end record;

  ... -- functions Area, MI
end Geometry.Triangles;

-- and so on
```

This is of course a much more beautiful structure and avoids having to write Root.Object when doing the extensions. But, horrors, our assignments in Ada 95 now have to be changed to

```
A := Geometry.Circles.Area(A_Circle);
D := Geometry.Distance(A_Triangle);
M := Geometry.Squares.MI(A_Square);
```

But the lucky programmer using Ada 2005 can still write

```
A := A_Circle.Area;
D := A_Triangle.Distance;
M := A_Square.MI;
```

and have a refreshing coffee (or a relaxing martini) while we are toiling with the editor.

Some time later the program might be extended to accommodate triangles that are specialized to be equilateral. This might be done by

```
package Geometry.Triangles.Equilateral is
  type Equilateral_Triangle is new Triangle with private;
  ...
private
  ...
end;
```

This type of course inherits all the operations of the type Triangle. We might now realize that the object A_Triangle of type Triangle was equilateral anyway and so it would be better to change it to be of type Equilateral_Triangle. The lucky Ada 2005 programmer will only have to change the

declaration of the object but the poor Ada 95 programmer will have to change the calls on all its primitive operations such as

```
A := Geometry.Triangles.Area(A_Triangle);
```

to the corresponding

```
A := Geometry.Triangles.Equilateral.Area(A_Triangle);
```

Other advantages of the prefixed notation were mentioned in the Introduction. One is that it unifies the notation for calling a function with a single parameter and directly reading a component of the object. Thus we can write uniformly

```
X := A_Circle.X_Coord;
A := A_Circle.Area;
```

Of course if we were foolish and had a *visible* component Area as well as a function Area then we could not call the function in this way.

But now suppose we decide to make the root type private so that the coordinates cannot be changed inadvertently. Moreover we decide to provide functions to read them. So we have

```
package Geometry is
  type Object is abstract tagged private;
  function Area(O: Object) return Float is abstract;
  function MI(O: Object) return Float is abstract;
  function Distance(O: Object'Class) return Float;

  function X_Coord(O: Object'Class) return Float;
  function Y_Coord(O: Object'Class) return Float;

private
  type Object is tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;

end Geometry;
```

Using Ada 95 we would now have to change statements such as

```
X := A_Triangle.X_Coord;
Y := A_Triangle.Y_Coord;
```

into

```
X := Geometry.X_Coord(A_Triangle);
Y := Geometry.Y_Coord(A_Triangle);
```

or (if we had not been wise enough to make the functions class wide) perhaps even

```
X := Geometry.Triangles.Equilateral.X_Coord(A_Triangle);
Y := Geometry.Triangles.Equilateral.Y_Coord(A_Triangle);
```

whereas in Ada 2005 we do not have to make any changes at all.

Another advantage mentioned in the Introduction is that when using access types explicit dereferencing is not necessary. Suppose we have

```
type Pointer is access all Geometry.Object'Class;
...
This_One: Pointer := A_Circle'Access;
```

In Ada 95 (assuming that X_Coord is a visible component) we have to write

```
Put(This_One.X_Coord); ...
Put(This_One.Y_Coord); ...
Put(Geometry.Area(This_One.all));
```

whereas in Ada 2005 we can uniformly write

```
Put(This_One.X_Coord); ...
Put(This_One.Y_Coord); ...
Put(This_One.Area);
```

and of course this remains unchanged if we make the coordinates into functions whereas the Ada 95 statements will need to be changed.

There are other structural changes that can occur during program development which are much easier to cope with using the prefix notation. For example, a class wide operation might be moved. And in the case of multiple interfaces to be described in the next section an operation might be moved from one interface to another.

It is clear that the prefixed notation has significant benefits both in terms of program clarity and for program maintenance.

Other variations on the rules for the use of the notation were considered. One was that the mechanism should apply to untagged types as well but this was rejected on the grounds that it might add to rather than reduce confusion in some cases. In any event, untagged types do not have class wide types so they are intrinsically simpler.

It is of course important to note that the first parameter of an operation plays a special role since in order to take advantage of the prefixed notation we have to ensure that the first parameter is a controlling parameter. Treating the first parameter specially can appear odd in some circumstances such as when there is symmetry among the parameters. Thus suppose we have a set package for creating and manipulating sets of integers

```
package Sets is
  type Set is tagged private;
  function Empty return Set;
  function Unit(N: Integer) return Set;
  function Union(S, T: Set) return Set;
  function Intersection(S, T: Set) return Set;
  function Size(S: Set) return Integer;
  ...
end Sets;
```

then we can apply the function Union in the traditional way

```
A, B, C: Set;
...
C := Sets.Union(A, B);
```

The object oriented addict can also write

```
C := A.Union(B);
```

but this destroys the obvious symmetry and is rather like sending 3 to be added to 2 mentioned at the beginning of this discussion.

Hopefully the mature programmer will use the OO notation wisely. Maybe its existence will encourage a more uniform style in which the first parameter is always a controlling operand wherever possible. Of course it cannot be used for functions which are tag indeterminate such as

```
function Empty return Set;
function Unit(N: Integer) return Set;
```

since there are no controlling parameters. If a subprogram has just one parameter (which is controlling) such as Size then the call just becomes X.Size and no parentheses are necessary.

Note that the prefix does not have to be simply the name of an object such as X, it could be a function call so we might write

```
N := Sets.Empty.Size;                -- N = 0
M := Sets.Unit(99).Size;             -- M = 1
```

with the obvious results as indicated.

4 Interfaces

In Ada 95, a derived type can really only have one immediate ancestor. This means that true multiple inheritance is not possible although curious techniques involving discriminants and generics can be used in some circumstances

General multiple inheritance has problems. Suppose that we have a type T with some components and operations. Perhaps

```
type T is tagged
  record
    A: Integer;
    B: Boolean;
  end record;

procedure Op1(X: T);
procedure Op2(X: T);
```

Now suppose we derive two new types from T thus

```
type T1 is new T with
  record
    C: Character;
  end record;

procedure Op3(X: T1);

-- Op1 and Op2 inherited, Op3 added

type T2 is new T with
  record
    C: Colour;
  end record;

procedure Op1(X: T2);
procedure Op4(X: T2);

-- Op1 overridden, Op2 inherited, Op4 added
```

Now suppose that we were able to derive a further type from both T1 and T2 by perhaps writing

```
type TT is new T1 and T2 with null record;          -- illegal
```

This is about the simplest example one could imagine. We have added no further components or operations. But what would TT have inherited from its two parents?

There is a general rule that a record cannot have two components with the same identifier so presumably it has just one component A and one component B. But what about C? Does it inherit the character or the colour? Or is it illegal because of the clash? Suppose T2 had a component D instead of C. Would that be OK? Would TT then have four components?

And then consider the operations. Presumably it has both Op1 and Op2. But which implementation of Op1? Is it the original Op1 inherited from T via T1 or the overridden version inherited from T2? Clearly it cannot have both. But there is no reason why it cannot have both Op3 and Op4, one inherited from each parent.

The problems arise when inheriting components from more than one parent and inheriting different *implementations* of the same operation from more than one parent. There is no problem with inheriting the same specification of an operation from two parents.

These observations provide the essence of the solution. At most one parent can have components and at most one parent can have concrete operations – for simplicity we make them the same parent. But abstract operations can be inherited from several parents. This can be phrased as saying that this kind of multiple inheritance is about merging contracts to be satisfied rather than merging algorithms or state.

So Ada 2005 introduces the concept of an interface which is a tagged type with no components and no concrete operations. The idea of a null procedure as an operation of a tagged type is also introduced; this has no body but behaves as if it has a null body. Interfaces are only permitted to have abstract subprograms and null procedures as operations.

We will outline the ways in which interfaces can be declared and composed in a symbolic way and then conclude with a more practical example.

We might declare a package Pi1 containing an interface Int1 thus

```
package Pi1 is
  type Int1 is interface;
  procedure Op1(X: Int1) is abstract;
  procedure N1(X: Int1) is null;
end Pi1;
```

Note the syntax. It uses the new reserved word **interface**. It does not say **tagged** although all interface types are tagged. The abstract procedure Op1 has to be explicitly stated to be abstract as usual. The null procedure N1 uses new syntax as well. Remember that a null procedure behaves as if its body comprises a single null statement; but it doesn't actually have a concrete body.

The main type derivation rule then becomes that a tagged type can be derived from zero or one conventional tagged types plus zero or more interface types. Thus

```
type NT is new T and Int1 and Int2 with ... ;
```

where Int1 and Int2 are interface types. The normal tagged type if any has to be given first in the declaration. The first type is known as the parent so the parent could be a normal tagged type or an interface. The other types are known as progenitors. Additional components and operations are allowed in the usual way.

The term progenitors may seem strange but the term ancestors in this context was confusing and so a new term was necessary. Progenitors comes from the Latin progignere, to beget, and so is very appropriate.

It might have been thought that it would be quite feasible to avoid the formal introduction of the concept of an interface by simply saying that multiple parents are allowed provided only the first has components and concrete operations. However, there would have been implementation complexities with the risk of violating privacy and distributed overheads. Moreover, it would have caused maintenance problems since simply adding a component to a type or making one of its abstract operations concrete would cause errors elsewhere in the system if it was being used as a secondary parent. It is thus much better to treat interfaces as a fundamentally new concept. Another advantage is that this provides a new class of generic parameter rather neatly without complex rules for instantiations.

If the normal tagged type T is in a package Pt with operations Opt1, Opt2 and so on we could now write

```
with Pi1, Pt;
package PNT is
  type NT is new Pt.T and Pi1.Int1 with ... ;
  procedure Op1(X: NT);                    -- concrete procedure
  --  possibly other ops of NT
end PNT;
```

We must of course provide a concrete procedure for Op1 inherited from the interface Int1 since we have declared NT as a concrete type. We could also provide an overriding for N1 but if we do not then we simply inherit the null procedure of Int1. We could also override the inherited operations Opt1 and Opt2 from T in the usual way.

Interfaces can be composed from other interfaces thus

```
type Int2 is interface;
...
type Int3 is interface and Int1;
...
type Int4 is interface and Int1 and Int2;
...
```

Note the syntax. A tagged type declaration always has just one of **interface**, **tagged** and **with** (it doesn't have any if it is not a tagged type). When we derive interfaces in this way we can add new operations so that the new interface such as Int4 will have all the operations of both Int1 and Int2 plus possibly some others declared specifically as operations of Int4. All these operations must be abstract or null and there are fairly obvious rules regarding what happens if two or more of the ancestor interfaces have the same operation. Thus a null procedure overrides an abstract one but otherwise repeated operations must have profiles that are type conformant and have the same convention.

We refer to all the interfaces in an interface list as progenitors. So Int1 and Int2 are progenitors of Int4. The first one is not a parent – that term is only used when deriving a type as opposed to composing an interface.

Note that the term ancestor covers all generations whereas parent and progenitors are first generation only.

Similar rules apply when a tagged type is derived from another type plus one or more interfaces as in the case of the type NT which was

```
type NT is new T and Int1 and Int2 with ... ;
```

In this case it might be that T already has some of the operations of Int1 and/or Int2. If so then the operations of T must match those of Int1 or Int2 (be type conformant etc).

We informally speak of a specific tagged type as implementing an interface from which it is derived (directly or indirectly). The phrase "implementing an interface" is not used formally in the definition of Ada 2005 but it is useful for purposes of discussion.

Thus in the above example the tagged type NT must implement all the operations of the interfaces Int1 and Int2. If the type T already implements some of the operations then the type NT will automatically implement them because it will inherit the implementations from T. It could of course override such inherited operations in the usual way.

The normal "going abstract" rules apply in the case of functions. Thus if one operation is a function F thus

```
package Pi2 is
  type Int2 is interface;
  function F(Y: Int2) return Int2 is abstract;
end Pi2;
```

and T already has such a conforming operation

```
package PT is
  type T is tagged record ...
  function F(X: T) return T;
end PT;
```

then in this case the type NT must provide a concrete function F. See however the discussion at the end of this paper for the case when the type NT has a null extension.

Class wide types also apply to interface types. The class wide type Int1'Class covers all the types derived from the interface Int1 (both other interfaces as well as normal tagged types). We can then dispatch using an object of a concrete tagged type in that class in the usual way since we know that any abstract operation of Int1 will have been overridden. So we might have

```
type Int1_Ref is access all Int1'Class;
NT_Var: aliased NT;
Ref: Int1_Ref := NT_Var'Access;
```

Observe that conversion is permitted between the access to class wide type Int1_Ref and any access type that designates a type derived from the interface type Int1.

Interfaces can also be used in private extensions and as generic parameters.

Thus

```
  type PT is new T and Int2 and Int3 with private;
  ...
private
  type PT is new T and Int2 and Int3 with null record;
```

An important rule regarding private extensions is that the full view and the partial view must agree with respect to the set of interfaces they implement. Thus although the parent in the full view need not be T but can be any type derived from T, the same is not true of the interfaces which must be such that they both implement the same set exactly. This rule is important in order to prevent a client type from overriding private operations of the parent if the client implements an interface added in the private part.

Generic parameters take the form

```
generic
  type FI is interface and Int1 and Int2;
package ...
```

and then the actual parameter must be an interface which implements all the ancestors Int1, Int2 etc. The formal could also just be **type** FI **is interface**; in which case the actual parameter can be any interface. There might be subprograms passed as further parameters which would require that the actual has certain operations. The interfaces Int1 and Int2 might themselves be formal parameters occurring earlier in the parameter list.

Interfaces (and formal interfaces) can also be limited thus

```
type LI is limited interface;
```

We can compose mixtures of limited and nonlimited interfaces but if any one of them is nonlimited then the resulting interface must not be specified as limited. This is because it must implement the equality and assignment operations implied by the nonlimited interface. Similar rules apply to types which implement one or more interfaces. We will come back to this topic in a moment.

There are other forms of interfaces, namely synchronized interfaces, task interfaces, and protected interfaces. These bring support for polymorphic, class wide object oriented programming to the real time programming arena. They will be described in a later paper.

Having described the general ideas in somewhat symbolic terms, we will now discuss a more concrete example.

Before doing so it is important to emphasize that interfaces cannot have components and therefore if we are to perform multiple inheritance then we should think in terms of abstract operations to read and write components rather than the components themselves. This is standard OO thinking anyway because it preserves abstraction by hiding implementation details.

Thus rather than having a component such as Comp it is better to have a pair of operations. The function to read the component can simply be called Comp. A procedure to update the component might be Set_Comp. We will generally use this convention although it is not always appropriate to treat the components as unrelated entities.

Suppose now that we want to print images of the geometrical objects. We will assume that the root type is declared as

```
package Geometry is
  type Object is abstract tagged private;
  procedure Move(O: in out Object'Class; X, Y: Float);
  ...
private
  type Object is abstract tagged
    record
      X_Coord: Float := 0.0;
      Y_Coord: Float := 0.0;
    end record;
  ...
end;
```

The type Object is private and by default both coordinates have the value of zero. The procedure Move, which is class wide, enables any object to be moved to the location specified by the parameters.

Suppose also that we have a line drawing package with the following specification

```
package Line_Draw is
   type Printable is interface;
   type Colour is ... ;
   type Points is ... ;
   procedure Set_Hue(P: in out Printable; C: in Colour) is abstract;
   function Hue(P: Printable) return Colour is abstract;
   procedure Set_Width(P: in out Printable; W: in Points) is abstract;
   function Width(P: Printable) return Points is abstract;

   type Line is ... ;
   type Line_Set is ... ;

   function To_Lines(P: Printable) return Line_Set is abstract;

   procedure Print(P: in Printable'Class);

private
   procedure Draw_It(L: Line; C: Colour; W: Points);

end Line_Draw;
```

The idea of this package is that it enables the drawing of an image as a set of lines. The attributes of the image are the hue and the width of the lines and there are pairs of subprograms to set and read these properties of any object of the interface Printable and its descendants. These operations are of course abstract.

In order to prepare an object in a form that can be printed it has to be converted to a set of lines. The function To_Lines converts an object of the type Printable into a set of lines; again it is abstract. The details of various types such as Line and Line_Set are not shown.

Finally the package Line_Draw declares a concrete procedure Print which takes an object of type Printable'Class and does the actual drawing using the slave procedure Draw_It declared in the private part. Note that Print is class wide and is concrete. This is an important point. Although all primitive operations of an interface must be abstract this does not apply to class wide operations since these are not primitive.

The body of the procedure Print could take the form

```
procedure Print(P: in Printable'Class) is
   L: Line_Set := To_Lines(P);
   A_Line: Line;
begin
   loop
      -- iterate over the Line_Set and extract each line
      A_Line := ...
      Draw_It(A_Line, Hue(P), Width(P));
   end loop;
end Print;
```

but this is all hidden from the user. Note that the procedure Draw_It is declared in the private part since it need not be visible to the user.

One reason why the user has to provide To_Lines is that only the user knows about the details of how best to represent the object. For example the poor circle will have to be represented crudely as a polygon of many sides, perhaps a hectogon of 100 sides.

We can now take at least two different approaches. We can for example write

```
with Geometry, Line_Draw;
package Printable_Geometry is
  type Printable_Object is
                    abstract new Geometry.Object and Line_Draw.Printable with private;
  procedure Set_Hue(P: in out Printable_Object; C: in Colour);
  function Hue(P: Printable_Object) return Colour;
  procedure Set_Width(P: in out Printable_Object; W: in Points);
  function Width(P: Printable_Object) return Points;
  function To_Lines(P: Printable_Object) return Line_Set is abstract;

private
  ...
end Printable_Geometry;
```

The type Printable_Object is a descendant of both Object and Printable and all concrete types descended from Printable_Object will therefore have all the operations of both Object and Printable. Note carefully that we have to put Object first in the declaration of Printable_Object and that the following would be illegal

```
type Printable_Object is
                abstract new Line_Draw.Printable and Geometry.Object with private;   --illegal
```

This is because of the rule that only the first type in the list can be a normal tagged type; any others must be interfaces. Remember that the first type is always known as the parent type and so the parent type in this case is Object.

The type Printable_Object is declared as abstract because we do not want to implement To_Lines at this stage. Nevertheless we can provide concrete subprograms for all the other operations of the interface Printable. We have given the type a private extension and so in the private part of its containing package we might have

```
private
  type Printable_Object is abstract new Geometry.Object and Line_Draw.Printable with
    record
      Hue: Colour := Black;
      Width: Points := 1;
    end record;
end Printable_Geometry;
```

Just for way of illustration, the components have been given default values. In the package body the operations such as the function Hue are simply

```
  function Hue(P: Printable_Object) return Colour is
  begin
    return P.Hue;
  end;
```

Luckily the visibility rules are such that this does not do an infinite recursion!

Note that the information containing the style components is in the record structure following the geometrical properties. This is a simple linear structure since interfaces cannot add components. However, since the type Printable_Object has all the operations of both an Object and a Printable, this adds a small amount of complexity to the arrangement of dispatch tables. But this detail is hidden from the user.

The key point is that we can now pass any object of the type Printable_Object or its descendants to the procedure

```
procedure Print(P: in Printable'Class);
```

and then (as outlined above) within Print we can find the colour to be used by calling the function Hue and the line width to use by calling the function Width and we can convert the object into a set of lines by calling the function To_Lines.

And now we can declare the various types Circle, Triangle, Square and so on by making them descendants of the type Printable_Object and in each case we have to implement the function To_Lines.

The unfortunate aspect of this approach is that we have to move the geometry hierarchy. For example the triangle package might now be

```
package Printable_Geometry.Triangles is
  type Printable_Triangle is new Printable_Object with
    record
      A, B, C: Float;
    end record;
  ... -- functions Area, To_Lines etc
end;
```

We can now declare a Printable_Triangle thus

```
A_Triangle: Printable_Triangle := (Printable_Object with A => 4.0, B => 4.0, C => 4.0);
```

This declares an equilateral triangle with sides of length 4.0. Its private Hue and Width components are set by default. Its coordinates which are also private are by default set to zero so that it is located at the origin. (The reader can improve the example by making the components A, B and C private as well.)

We can conveniently move it to wherever we want by using the procedure Move which being class wide applies to all types derived from Object. So we can write

```
A_Triangle.Move(1.0, 2.0);
```

And now we can make a red sign

```
Sign: Printable_Triangle := A_Triangle;
```

Having declared the object Sign, we can give it width and hue and print it

```
Sign.Set_Hue(Red);
Sign.Set_Width(3);
Sign.Print;                                    -- print thick red triangle
```

As we observed earlier this approach has the disadvantage that we had to move the geometry hierarchy. A different approach which avoids this is to declare printable objects of just the kinds we want as and when we want them.

So assume now that we have the package Line_Draw as before and the original package Geometry and its child packages. Suppose we want to make printable triangles and circles. We could write

```
with Geometry, Line_Draw;  use Geometry;
package Printable_Objects is
  type Printable_Triangle is new Triangles.Triangle and Line_Draw.Printable with private;
  type Printable_Circle is new Circles.Circle and Line_Draw.Printable with private;

  procedure Set_Hue(P: in out Printable_Triangle; C: in Colour);
  function Hue(P: Printable_Triangle return Colour;
  procedure Set_Width(P: in out Printable_Triangle; W: in Points);
```

```
    function Width(P: Printable_Triangle) return Points;
    function To_Lines(T: Printable_Triangle) return Line_Set;

    procedure Set_Hue(P: in out Printable_Circle; C: in Colour);
    function Hue(P: Printable_Circle) return Colour;
    procedure Set_Width(P: in out Printable_Circle; W: in Points);
    function Width(P: Printable_Circle) return Points;
    function To_Lines(C: Printable_Circle) return Line_Set;
  private

    type Printable_Triangle is new Triangles.Triangle and Line_Draw.Printable with
      record
        Hue: Colour := Black;
        Width: Points := 1;
      end record;

    type Printable_Circle is new Circles.Circle and Line_Draw.Printable with
      record
        Hue: Colour := Black;
        Width: Points := 1;
      end record;
  end Printable_Objects;
```

and the body of the package will provide the various subprogram bodies.

Now suppose we already have a normal triangle thus

```
A_Triangle: Geometry.Triangles.Triangle := ... ;
```

In order to print A_Triangle we first have to declare a printable triangle thus

```
Sign: Printable_Triangle;
```

and now we can set the triangle components of it using a view conversion thus

```
Triangle(Sign) := A_Triangle;
```

And then as before we write

```
Sign.Set_Hue(Red);
Sign.Set_Width(3);
Sign.Print_It;                          -- print thick red triangle
```

This second approach is probably better since it does not require changing the geometry hierarchy. The downside is that we have to declare the boring hue and width subprograms repeatedly. We can make this much easier by declaring a generic package thus

```
with Line_Draw;  use Line_Draw;
generic
  type T is abstract tagged private;
package Make_Printable is
  type Printable_T is abstract new T and Printable with private;

  procedure Set_Hue(P: in out Printable_T; C: in Colour);
  function Hue(P: Printable_T) return Colour;
  procedure Set_Width(P: in out Printable_T; W: in Points);
  function Width(P: Printable_T) return Points;

private
  type Printable_T is abstract new T and Printable with
```

```
      record
        Hue: Colour := Black;
        Width: Points := 1;
      end record;
   end;
```

This generic can be used to make any type printable. We simply write

```
package P_Triangle is new Make_Printable(Triangle);
type Printable_Triangle is new P_Triangle.Printable_T with null record;
function To_Lines(T: Printable_Triangle) return Line_Set;
```

The instantiation of the package creates a type Printable_T which has all the hue and width operations and the required additional components. However, it simply inherits the abstract function To_Lines and so itself has to be an abstract type. Note that the function To_Lines has to be especially coded for each type anyway unlike the hue and width operations which can be the same.

We now do a further derivation largely in order to give the type Printable_T the required name Printable_Triangle and at this stage we provide the concrete function To_Lines.

We can then proceed as before. Thus the generic makes the whole process very easy – any type can be made printable by just writing three lines plus the body of the function To_Lines.

Hopefully this example has illustrated a number of important points about the use of interfaces. The key thing perhaps is that we can use the procedure Print to print anything that implements the interface Printable.

Earlier we stated that it was a common convention to provide pairs of operations to read and update properties such as Hue and Set_Hue and Width and Set_Width. This is not always appropriate. Thus if we have related components such as X_Coord and Y_Coord then although individual functions to read them might be sensible, it is undoubtedly better to update the two values together with a single procedure such as the procedure Move declared earlier. Thus if we wish to move an object from the origin (0.0, 0.0) to say (3.0, 4.0) and do it by two calls

```
Obj.Set_X_Coord(3.0);          -- first change X
Obj.Set_Y_Coord(4.0);          -- then change Y
```

then it seems as if it was transitorily at the point (3.0, 0.0). There are various other risks as well. We might forget to set one component or accidentally set the same component twice.

Finally, as discussed earlier, null procedures are a new kind of subprogram and the user-defined operations of an interface must be null procedures or abstract subprograms – there is of course no such thing as a null function.

(Nonlimited interfaces do have one concrete operation and that is predefined equality; it could even be overridden with an abstract one.)

Null procedures will be found useful for interfaces but are in fact applicable to any types. As an example the package Ada.Finalization now uses null procedures for Initialize, Adjust, and Finalize as described in the Introduction.

We conclude this section with a few further remarks on limitedness. We noted earlier that an interface can be explicitly stated to be limited so we might have

```
type LI is limited interface;      -- limited
type NLI is interface;             -- nonlimited
```

An interface is limited only if it says limited (or synchronized etc). As mentioned earlier, a descendant of a nonlimited interface must be nonlimited since it must implement assignment and

equality. So if an interface is composed from a mixture of limited and nonlimited interfaces it must be nonlimited

```
type I is interface and LI and NLI;              -- legal
type I is limited interface and LI and NLI;      -- illegal
```

In other words, limitedness is never inherited from an interface but has to be stated explicitly. This applies to both the composition of interfaces and type derivation. On the other hand, in the case of type derivation, limitedness is inherited from the parent provided it is not an interface. This is necessary for compatibility with Ada 95. So given

```
type LT is limited tagged ...
type NLT is tagged ...
```

then

```
type T is new NLT and LI with ...        -- legal, T not limited
type T is new NLT and NLI with ...       -- legal, T not limited
type T is new LT and LI with ...         -- legal, T limited
type T is new LT and NLI with ...        -- illegal
```

The last is illegal because T is expected to be limited because it is derived from the limited parent type LT and yet it is also a descendant of the nonlimited interface NLI.

In order to avoid certain curious difficulties, Ada 2005 permits **limited** to be stated explicitly on type derivation. (It would have been nice to insist on this always for clarity but such a change would have been too much of an incompatibility.) If we do state **limited** explicitly then the parent must be limited (whether it is a type or an interface).

Using **limited** is necessary if we wish to derive a limited type from a limited interface thus

```
type T is limited new LI with ...
```

These rules really all come down to the same thing. If a parent or progenitor (indeed any ancestor) is nonlimited then the descendant must be nonlimited. We can state that in reverse, if a type (including an interface) is limited then all its ancestors must be limited.

An earlier version of Ada 2005 ran into difficulties in this area because in the case of a type derived just from interfaces, the behaviour could depend upon the order of their appearance in the list (because the rules for parent and progenitors are a bit different). But in the final version of the language the order does not matter. So

```
type T is new NLI and LI with ...        -- legal, not limited
type T is new LI and NLI with ...        -- legal, not limited
```

But the following are of course illegal

```
type T is limited new NLI and LI with ...    -- illegal
type T is limited new LI and NLI with ...    -- illegal
```

There are also similar changes to generic formals and type extension – Ada 2005 permits **limited** to be given explicitly in both cases.

5 Nested type extension

In Ada 95 type extension of tagged types has to be at the same level as the parent type. This can be quite a problem. In particular it means that all controlled types must be declared at library level because the root types Controlled and Limited_Controlled are declared in the library level package Ada.Finalization. The same applies to storage pools and streams because again the root types Root_Storage_Pool and Root_Stream_Type are declared in library packages.

This has a cumulative effect since if we write a generic unit using any of these types then that package can itself only be instantiated at library level. This enforces a very flat level of programming and hinders abstraction.

The problems can actually be illustrated without having to use controlled types or generics. As a simple example consider the following which is adapted from a text book [3]. It manipulates lists of colours and we assume that the type Colour is declared somewhere.

```
package Lists is
  type List is limited private;
  type Iterator is abstract tagged null record;
  procedure Iterate(IC: in Iterator'Class; L: in List);
  procedure Action(It: in out Iterator; C: in out Colour) is abstract;
private
  ...
end;
```

The idea is that a call of Iterate calls Action (by dispatching) on each object of the list and thereby gives access to the colour of that object. The user has to declare an extension of Iterator and a specific procedure Action to do whatever is required on each object.

Some readers may find this sort of topic confusing. It might be easier to understand if we look at the private part and body of the package Lists which might be

```
private
  type Cell is
    record
      Next: access Cell;                    -- anonymous type
      C: Colour;
    end record;

  type List is access Cell;
end;

package body Lists is
  procedure Iterate(IC: in Iterator'Class; L: in List) is
    This: access Cell := L;
  begin
    while This /= null loop
      Action(IC, This.C);                   -- dispatching call
                                            -- or IC.Action(This.C);
      This := This.Next;
    end loop;
  end Iterate;
end Lists;
```

Note the use of the anonymous access types which avoid the need to have an incomplete declaration of Cell in the private part.

Now suppose we wish to change the colour of every green object to red. We write (in some library level package)

```
type GTR_It is new Iterator with null record;

procedure Action(It: in out GTR_It; C: in out Colour) is
begin
```

```
    if C = Green then C := Red; end if;
  end Action;

  procedure Green_To_Red(L: in List) is
    It: GTR_It;
  begin
    Iterate(It, L);                          -- or It.Iterate(L);
  end Green_To_Red;
```

This works but is not ideal. The type GTR_It and the procedure Action should not be declared outside the procedure Green_To_Red since they are really only part of its internal workings. But we cannot declare the type GTR_It inside the procedure in Ada 95 because that would be an extension at an inner level.

The extra facilities of the predefined library in Ada 2005 and especially the introduction of containers which are naturally implemented as generic units forced a reconsideration of the reasons for restricting type extension in Ada 95. The danger of nested extension of course is that values of objects could violate the accessibility rules and outlive their type declaration. It was concluded that type extension could be permitted at nested levels with the addition of just a few checks to ensure that the accessibility rules were not violated.

So in Ada 2005 the procedure Green_To_Red can be written as

```
  procedure Green_To_Red(L: in List) is
    type GTR_It is new Iterator with null record;

    procedure Action(It: in out GTR_It; C: in out Colour) is
    begin
      if C = Green then C := Red; end if;
    end Action;

    It: GTR_It;
  begin
    Iterate(It, L);                          -- or It.Iterate(L);
  end Green_To_Red;
```

and all the workings are now wrapped up within the procedure as they should be.

Note incidentally that we can use the notation It.Iterate(L); even though the type GTR_It is not declared in a package in this case. Remember that although we cannot add new dispatching operations to a type unless it is declared in a package specification, nevertheless we can always override existing ones such as Action.

This example is all quite harmless and nothing can go wrong despite the fact that we have performed the extension at an inner level. This is because the value It does not outlive the execution of the procedure Action.

But suppose we have a class wide object Global_It as in the following

```
with Lists; use Lists;
package body P is

  function Dodgy return Iterator'Class is
    type Bad_It is new Iterator with null record;

    procedure Action(It: in out Bad_It; C: in out Colour) is
    begin
      ...
    end Action;
```

```
    It: Bad_It;
  begin
    return It;
  end Dodgy;

  Global_It: Iterator'Class := Dodgy;
begin
  Global_It.Action(Red_For_Danger);        -- dispatches
end P;
```

Now we are in deep trouble. We have returned a value of the local type Bad_It, assigned it as the initial value to Global_It and then dispatched on it to the procedure Action. But the procedure Action that will be called is the one inside Dodgy and this does not exist anymore since we have left the function Dodgy. So this must not be allowed to happen.

So various accessibility checks are required. There is a check on the return from a function with a class wide result that the value being returned does not have the tag of a type at a deeper level than that of the function itself. So in this example there is a check on the return from the function Dodgy; this fails and raises Program_Error so all is well.

There are similar checks on class wide allocators and when using T'Class'Input or T'Class'Output. Some of these can be carried out at compile time but others have to be checked at run time and they also raise Program_Error if they fail.

Moreover, in order to implement the checks associated with T'Class'Input and T'Class'Output two additional functions are declared in the package Ada.Tags; these are

```
function Descendant_Tag(External: String; Ancestor: Tag) return Tag;

function Is_Descendant_At_Same_Level (Descendant, Ancestor: Tag) return Boolean;
```

The use of these will be outlined in the next section.

6 Object factory functions

The Ada 95 Rationale (Section 4.4.1) [2] says "We also note that object oriented programming requires thought especially if variant programming is to be avoided. There is a general difficulty in finding out what is coming which is particularly obvious with input–output; it is easy to write dispatching output operations but generally impossible for input." In this context, variant programming means messing about with case statements and so on.

The point about input–output is that it is easy to write a heterogeneous file but not so easy to read it. In the simple case of a text file we can just do a series of calls of Put thus

```
Put ("John is ");  Put(21, 0);  Put(" years old.");
```

But text input is not so easy unless we know the order of the items in the file. If we don't know the order then we really have to read the wretched thing a line at a time and then analyse the lines.

Ada 95 includes a mechanism for doing this relatively easily in the case of tagged types and stream input–output. Suppose we have a class of tagged types rooted at Root with various derived specific types T1, T2 and so on. We can then output a sequence of values X1, X2, X3 of a variety of these types to a file identified by the stream access value S by writing

```
Root'Class'Output(S, X1);
Root'Class'Output(S, X2);
Root'Class'Output(S, X3);
...
```

The various calls first write the tag of the specific type and then the value of the type. The tag corresponding to the type T1 is the string External_Tag(T1'Tag). Remember that External_Tag is a function in the predefined package Ada.Tags.

On input we can reverse the process by writing something like

```
declare
   X: Root'Class := Root'Class'Input(S);
begin
   Process(X);                              -- now process the object in X
```

The call of Root'Class'Input first reads the external tag and then dispatches to the appropriate function Tn'Input according to the value of the tag. The function reads the value and this is now assigned as the initial value to the class wide variable X. We can then do whatever we want with X by perhaps dispatching to a procedure Process which deals with it according to its specific type.

This works in Ada 95 but it is all magic and done by smoke and mirrors inside the implementation. The underlying techniques are unfortunately not available to the user.

This means that if we want to devise our own stream protocol or maybe just process some values in circumstances where we cannot directly use dispatching then we have to do it all ourselves with if statements or case statements. Thus we might be given a tag value and separately some information from which we can create the values of the particular type. In Ada 95 we typically have to do something like

```
The_Tag: Ada.Tags.Tag;
A_T1: T1;                                   -- series of objects of each
A_T2: T2;                                   -- specific type
A_T3: T3;
...
The_Tag := Get_Tag( ... );                  -- get the tag value
if The_Tag = T1'Tag then
   A_T1 := Get_T( ... );                    -- get value of specific type
   Process(A_T1);                           -- process the object
elsif The_Tag = T2'Tag then
   A_T2 := Get_T( ... );                    -- get value of specific type
   Process(A_T2);                           -- process the object
elsif
   ...
end if;
```

We assume that Get_T is a primitive function of the class rooted at Root. There is therefore a function for each specific type and the selection in the if statements is made at compile time by the normal overload rules. Similarly Process is also a primitive subprogram of the class of types.

This is all very tedious and needs careful maintenance if we add further types to the class.

Ada 2005 overcomes this problem by providing a generic object constructor function. Its specification is

```
generic
   type T (<>) is abstract tagged limited private;
   type Parameters (<>) is limited private;
   with function Constructor(Params: access Parameters) return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
                              (The_Tag: Tag; Params: access Parameters) return T'Class;
```

```
pragma Preelaborate(Generic_Dispatching_Constructor);
pragma Convention(Intrinsic, Generic_Dispatching_Constructor);
```

This generic function works for both limited and nonlimited types. Remember that a nonlimited type is allowed as an actual generic parameter corresponding to a limited formal generic type. The generic function Generic_Dispatching_Constructor is Pure and has convention Intrinsic.

Note carefully the formal function Constructor. This is an example of a new kind of formal generic parameter introduced in Ada 2005. The distinctive feature is the use of **is abstract** in its specification. The interpretation is that the actual function must be a dispatching operation of a tagged type uniquely identified by the profile of the formal function. The actual operation can be concrete or abstract. Remember that the overriding rules ensure that the specific operation for any concrete type will always have a concrete body. Note also that since the operation is abstract it can only be called through dispatching.

In this example, it therefore has to be a dispatching operation of the type T since that is the only tagged type involved in the profile of Constructor. We say that T is the controlling type. In the general case, the controlling type does not itself have to be a formal parameter of the generic unit but usually will be as here. Moreover, note that although the operation has to be a dispatching operation, it is not primitive and so if we derive from the type T, it will not be inherited.

Formal abstract subprograms can of course be procedures as well as functions. It is important that there is exactly one controlling type in the profile. Thus given that TT1 and TT2 are tagged types then the following would both be illegal

```
with procedure Do_This(X1: TT1; X2: TT2) is abstract;      -- illegal
with function Fn(X: Float) return Float is abstract;       -- illegal
```

The procedure Do_This is illegal because it has two controlling types TT1 and TT2. Remember that we can declare a subprogram with parameters of more than one tagged type but it can only be a dispatching operation of one tagged type. The function Fn is illegal because it doesn't have any controlling types at all (and so could never be called in a dispatching call anyway).

The formal function Constructor is legal because only T is tagged; the type Parameters which also occurs in its profile is not tagged.

And now to return to the dispatching constructor. The idea is that we instantiate the generic function with a (root) tagged type T, some type Parameters and the dispatching function Constructor. The type Parameters provides a means whereby auxiliary information can be passed to the function Constructor.

The generic function Generic_Dispatching_Constructor takes two parameters, one is the tag of the type of the object to be created and the other is the auxiliary information to be passed to the dispatching function Constructor.

Note that the type Parameters is used as an access parameter in both the generic function and the formal function Constructor. This is so that it can be matched by the profile of the attribute Input whose specification is

```
function T'Input(Stream: access Root_Stream_Type'Class) return T;
```

Suppose we instantiate Generic_Dispatching_Constructor to give a function Make_T. A call of Make_T takes a tag value, dispatches to the appropriate Constructor which creates a value of the specific tagged type corresponding to the tag and this is finally returned as the value of the class wide type T'Class as the result of Make_T. It's still magic but anyone can use the magic and not just the magician implementing stream input–output.

We can now do our abstract problem as follows

```
function Make_T is
   new Generic_Dispatching_Constructor(Root, Params, Get_T);

...
declare
  Aux: aliased Params := ... ;
  A_T: Root'Class:= Make_T(Get_Tag( ... ), Aux'Access);
begin
  Process(A_T);                                   -- dispatch to process the object
end;
```

We no longer have the tedious sequence of if statements and the calls of Get_T and Process are dispatching calls.

The previously magic function T'Class'Input can now be implemented in a very natural way by something like

```
function Dispatching_Input is
   new Generic_Dispatching_Constructor(T, Root_Stream_Type'Class, T'Input);

function T_Class_Input(S: access Root_Stream_Type'Class) return T'Class is
  The_String: String := String'Input(S);                -- read tag as string from stream
  The_Tag: Tag := Descendant_Tag(The_String, T'Tag);    -- convert to a tag
 begin
  -- now dispatch to the appropriate function Input
  return Dispatching_Input(The_Tag, S);
end T_Class_Input;

for T'Class'Input use T_Class_Input;
```

The body could of course be written as one giant statement

```
return Dispatching_Input(Descendant_Tag(String'Input(S), T'Tag), S);
```

but breaking it down hopefully clarifies what is happening.

Note the use of Descendant_Tag rather than Internal_Tag. Descendant_Tag is one of a few new functions introduced into the package Ada.Tags in Ada 2005. Streams did not work very well for nested tagged types in Ada 95 because of the possibility of multiple elaboration of declarations (as a result of tasking and recursion); this meant that two descendant types could have the same external tag value and Internal_Tag could not distinguish them. This is not an important problem in Ada 95 as nested tagged types are rarely used. In Ada 2005 the situation is potentially made worse because of the possibility of nested type extension.

The goal in Ada 2005 is simply to ensure that streams do work with types declared at the same level and to prevent erroneous behaviour otherwise. The goal is not to permit streams to work with the nested extensions introduced in Ada 2005. Any attempt to do so will result in Tag_Error being raised.

Note that we cannot actually declare an attribute function such as T'Class'Input by directly using the attribute name. We have to use some other identifier such as T_Class_Input and then use an attribute definition clause as shown above.

Observe that T'Class'Output can be implemented as

```
procedure T_Class_Output(S: access Root_Stream_Type'Class; X: in T'Class) is
begin
  if not Is_Descendant_At_Same_Level(X'Tag, T'Tag) then
    raise Tag_Error;
```

```
   end if;
   String'Output(S, External_Tag(X'Tag));
   T'Output(S, X);
end T_Class_Output;

for T'Class'Output use T_Class_Output;
```

Remember that streams are designed to work only with types declared at the same accessibility level as the parent type T. The call of Is_Descendant_At_Same_Level, which is another new function in Ada 2005, ensures this.

We can use the generic constructor to create our own stream protocol. We could in fact replace T'Class'Input and T'Class'Output or just create our own distinct subsystem. One reason why we might want to use a different protocol is when the external protocol is already given such as in the case of XML.

Note that it will sometimes be the case that there is no need to pass any auxiliary parameters to the constructor function in which case we can declare

```
type Params is null record;
Aux: aliased Params := (null record);
```

Another example can be based on part of the program Magic Moments in [3]. This reads in the values necessary to create various geometrical objects such as a Circle, Triangle, or Square which are derived from an abstract type Object. The values are preceded by a letter C, T or S as appropriate. The essence of the code is

```
Get(Code_Letter);
case Code_Letter is
   when 'C' => Object_Ptr := Get_Circle;
   when 'T' => Object_Ptr := Get_Triangle;
   when 'S' => Object_Ptr := Get_Square;
   ...
end case;
```

The types Circle, Triangle, and Square are derived from the root type Object and Object_Ptr is of the type **access** Object'Class. The function Get_Circle reads the value of the radius from the keyboard, the function Get_Triangle reads the values of the lengths of the three sides from the keyboard and so on.

The first thing to do is to change the various constructor functions such as Get_Circle into various specific overridings of a primitive operation Get_Object so that we can dispatch on it.

Rather than just read the code letter we could make the user type the external tag string and then we might have

```
function Make_Object is
   new Generic_Dispatching_Constructor(Object, Params, Get_Object);

...
S: String := Get_String;
...
Object_Ptr := new Object'(Make_Object(Internal_Tag(S), Aux'Access));
```

but this is very tedious because the user now has to type the external tag which will be an implementation defined mess of characters. Observe that the string produced by a call of Expanded_Name such as

```
OBJECTS.CIRCLE
```

cannot be used because it will not in general be unique and so there is no reverse function. (It is not generally unique because of tasking and recursion.) But Expanded_Name is useful for debugging purposes.

In these circumstances the best way to proceed is to invent some sort of registration system to make a map to convert the simple code letters into the tag. We might have a package

```
with Ada.Tags; use Ada.Tags;
package Tag_Registration is
  procedure Register(The_Tag: Tag; Code: Character);
  function Decode(Code: Character) return Tag;
end;
```

and then we can write

```
Register(Circle'Tag, 'C');
Register(Triangle'Tag, 'T');
Register(Square'Tag, 'S');
```

And now the program to read the code and then make the object becomes simply

```
Get(Code_Letter);
Object_Ptr := new Object'(Make_Object(Decode(Code_Letter), Aux'Access));
```

and there are no case statements to maintain.

The really important point about this example is that if we decide at a later date to add more types such as 'P' for Pentagon and 'H' for Hexagon then all we have to do is register the new code letters thus

```
Register(Pentagon'Tag, 'P');
Register(Hexagon'Tag, 'H');
```

and nothing else needs changing. This registration can conveniently be done when the types are declared.

The package Tag_Registration could be implemented trivially as follows by

```
package body Tag_Registration is
  Table: array (Character range 'A' .. 'Z') of Tag := (others => No_Tag);
  procedure Register(The_Tag: Tag; Code: Character) is
  begin
    Table(Code) := The_Tag;
  end Register;

  function Decode(Code: Character) return Tag is
  begin
    return Table(Code);
  end Decode;
end Tag_Registration;
```

The constant No_Tag is a value of the type Tag which does not represent an actual tag. If we forget to register a type then No_Tag will be returned by Decode and this will cause Make_Object to raise Tag_Error.

A more elegant registration system could be easily implemented using the container library which will be described in a later paper.

Note that any instance of Generic_Dispatching_Constructor checks that the tag passed as parameter is indeed that of a type descended from the root type T and raises Tag_Error if it is not.

In simple cases we could in fact perform that check for ourselves by writing something like

```
   Trial_Tag: Tag := The_Tag;
loop
   if Trial_Tag = T'Tag then exit; end if;
   Trial_Tag := Parent_Tag(Trial_Tag);
   if Trial_Tag = No_Tag then raise Tag_Error; end if;
end loop;
```

The function Parent_Tag and the constant No_Tag are further items in the package Ada.Tags whose specification in Ada 2005 is

```
package Ada.Tags is
   pragma Preelaborate(Tags);

   type Tag is private;
   No_Tag: constant Tag;

   function Expanded_Name(T: Tag) return String;
   ...                    -- also Wide and Wide_Wide versions
   function External_Tag(T: Tag) return String;
   function Internal_Tag(External: String) return Tag;
   function Descendant_Tag(External: String; Ancestor: Tag) return Tag;
   function Is_Descendant_At_Same_Level(Descendant, Ancestor: Tag) return Boolean;
   function Parent_Tag(T: Tag) return Tag;

   type Tag_Array is (Positive range <>) of Tag;
   function Interface_Ancestor_Tags(T: Tag) return Tag_Array;

   Tag_Error: exception;
private
   ...
end Ada.Tags;
```

The function Parent_Tag returns No_Tag if the parameter T of type Tag has no parent which will be the case if it is the ultimate root type of the class. As mentioned earlier, two other new functions Descendant_Tag and Is_Descendant_At_Same_Level are necessary to prevent the misuse of streams with types not all declared at the same level.

There is also a function Interface_Ancestor_Tags which returns the tags of all those interfaces which are ancestors of T as an array. This includes the parent if it is an interface, any progenitors and all their ancestors which are interfaces as well – but it excludes the type T itself.

Finally note that the introduction of 16- and 32-bit characters in identifiers means that functions also have to be provided to return the images of identifiers as a Wide_String or Wide_Wide_String. So we have functions Wide_Expanded_Name and Wide_Wide_Expanded_Name as well as Expanded_Name. The lower bound of the strings returned by these functions and by External_Tag is 1 – Ada 95 forgot to state this for External_Tag and Expanded_Name!

7 Overriding and overloading

One of the key goals in the design of Ada was to encourage the writing of correct programs. It was intended that the structure, strong typing, and so on should ensure that many errors which are not detected by most languages until run time should be caught at compile time in Ada. Unfortunately the introduction of type extension and overriding in Ada 95 produced a situation where careless errors in subprogram profiles lead to errors which are awkward to detect.

The Introduction described two typical examples. The first concerns the procedure Finalize. Consider

```
with Ada.Finalization;  use Ada.Finalization;
package Root is
  type T is new Controlled with ... ;
  procedure Op(Obj: in out T; Data: in Integer);
  procedure Finalise(Obj: in out T);
end Root;
```

We have inadvertently written Finalise rather than Finalize. This means that Finalize does not get overridden as expected and so the expected behaviour does not occur on finalization of objects of type T.

In Ada 2005 we can prefix the declaration with **overriding**

```
  overriding
  procedure Finalize(Obj: in out T);
```

And now if we inadvertently write Finalise then this will be detected during compilation.

Similar errors can occur in a profile. If we write

```
package Root.Leaf is
  type NT is new T with null record;
  overriding                                   -- overriding indicator
  procedure Op(Obj: in out NT; Data: in String);
end Root.Leaf;
```

then the compiler will detect that the new procedure Op has a parameter of type String rather than Integer.

However if we do want a new operation then we can write

```
  not overriding
  procedure Op(Obj: in out NT; Data: in String);
```

The overriding indicators can also be used with abstract subprograms, null procedures, renamings, instantiations, stubs, bodies and entries (we will deal with entries in the paper on tasking). So we can have

```
overriding
procedure Pap(X: TT) is abstract;

overriding
procedure Pep(X: TT) is null;

overriding
procedure Pip(Y: TT) renames Pop;

not overriding
procedure Poop is new Peep( ... );

overriding
procedure Pup(Z: TT) is separate;

overriding
procedure Pup(X: TT) is
begin ... end Pup;
```

We do not need to apply an overriding indicator to both a procedure specification and body but if we do then they naturally must not conflict. It is expected that overriding indicators will typically only be given on specifications but they would be appropriate in the case of a body standing alone as in the example of Action in the previous section. So we might have

```
procedure Green_To_Red(L: in List) is
  type GTR_It is new Iterator with null record;

  overriding
  procedure Action(It: in out GTR_It; C: in out Colour) is
  begin
    if C = Green then C := Red; end if;
  end Action;
...
```

The overriding indicators are optional for two reasons. One is simply for compatibility with Ada 95. The other concerns awkward problems with private types and generics.

Consider

```
package P is
  type NT is new T with private;
  procedure Op(X: T);
private
```

Now suppose the type T does not have an operation Op. Then clearly it would be wrong to write

```
package P is
  type NT is new T with private;          -- T has no Op
  overriding                              -- illegal
  procedure Op(X: T);
private
```

because that would violate the information known in the partial view.

But suppose that in fact it turns out that in the private part the type NT is actually derived from TT (itself derived from T) and that TT does have an operation Op.

```
private
  type NT is new TT with ...              -- TT has Op
end P;
```

In such a case it turns out in the end that Op is in fact overriding after all. We can then put an overriding indicator on the body of Op since at that point we do know that it is overriding.

Equally of course we should not specify **not overriding** for Op in the visible part because that might not be true either (since it might be that TT does have Op). However if we did put **not overriding** on the partial view then that would not in itself be an error but would simply constrain the full view not to be overriding and thus ensure that TT does not have Op.

Of course if T itself has Op then we could and indeed should put an overriding indicator in the visible part since we know that to be the truth at that point.

The general rule is not to lie. But the rules are slightly different for **overriding** and **not overriding**. For **overriding** it must not lie at the point concerned. For **not overriding** it must not lie anywhere.

This asymmetry is a bit like presuming the prisoner is innocent until proved guilty. We sometimes start with a view in which an operation appears not to be overriding and then later on we find that it

is overriding after all. But the reverse never happens – we never start with a view in which it is overriding and then later discover that it was not. So the asymmetry is real and justified.

There are other similar but more complex problems with private types concerning implicit declarations where the implicit declaration turns up much later and is overriding but has no physical presence on which to hang the indicator. It was concluded that by far the best approach to these problems was just to say that the overriding indicator is always optional. We cannot expect to find all the bugs in a program through syntax and static semantics; the key goal here is to provide a simple way of finding most of them.

Similar problems arise with generics. As is usual with generics the rules are checked in the generic itself and then rechecked upon instantiation (in this case for uses within both the visible part and private part of the specification). Consider

```
generic
  type GT is tagged private;
package GP is
  type NT is new GT with private;
  overriding                              -- illegal, GT has no Op
  procedure Op(X: NT);
private
```

This has to be illegal because GT has no operation Op. Of course the actual type at instantiation might have Op but the check has to pass both in the generic and in the instantiation.

On the other hand saying **not overriding** is allowed

```
generic
  type GT is tagged private;
package GP is
  type NT is new GT with private;
  not overriding                          -- legal, GT has no Op
  procedure Op(X: NT);
private
```

However, in this case we cannot instantiate GP with a type that does have an operation Op because it would fail when checked on the instantiation. So in a sense this imposes a further contract on the generic. If we do not want to impose this restriction then we must not give an overriding indicator on the procedure Op for NT.

Another situation arises when the generic formal is derived

```
generic
  type GT is new T with private;
package GP is
  type NT is new GT with private;
  overriding                              -- legal if T has Op
  procedure Op(X: NT);
private
```

In this case it might be that the type T does have an operation Op in which case we can give the overriding indicator.

We might also try

```
generic
  type GT is tagged private;
  with procedure Op(X: GT);
```

```
package GP is
  type NT is new GT with private;
  overriding                                   -- illegal, Op not primitive
  procedure Op(X: NT);
private
```

But this is incorrect because although GT has to have an operation corresponding to Op as specified in the formal parameter list, nevertheless it does not have to be a primitive operation nor does it have to be called Op and thus it isn't inherited.

It should also be observed that overriding indicators can be used with untagged types although they have been introduced primarily to avoid problems with dispatching operations. Consider

```
package P is
  type T is private;
  function "+" (Left, Right: T) return T;
private
  type T is range 0 .. 100;                    -- "+" overrides
end P;
```

as opposed to

```
package P is
  type T is private;
  function "+" (Left, Right: T) return T;
private
  type T is (Red, White, Blue);                -- "+" does not override
end P;
```

The point is that the partial view does not reveal whether overriding occurs or not – nor should it since either implementation ought to be acceptable. We should therefore remain silent regarding overriding in the partial view. This is similar to the private extension and generic cases discussed earlier. Inserting **overriding** would be illegal on both examples, while **not overriding** would be allowed only on the second one (which would constrain the implementation as in the previous examples). Again, it is permissible to put an overriding indicator on the body of "+" to indicate whether or not it does override.

It is also possible for a subprogram to be primitive for more than one type (this cannot happen for more than one tagged type but it can happen for untagged types or one tagged type and some untagged types). It could then be overriding for some types and not overriding for others. In such a case it is considered to be overriding as a whole and any indicator should reflect this.

The possibility of having a pragma which would enforce the use of overriding indicators (so that they too could not be inadvertently omitted) was eventually abandoned largely because of the private type and generic problem which made the topic very complicated.

Note the recommended layout, an overriding indicator should be placed on the line before the subprogram specification and aligned with it. This avoids disturbing the layout of the specification.

It is hoped that programmers will use overriding indicators freely. As mentioned in the Introduction, they are very valuable for preventing nasty errors during maintenance. Thus if we add a further parameter to an operation such as Op for a root type and all type extensions have overriding indicators then the compiler will report an error if we do not modify the operators of all the derived types correctly.

We now turn to a minor change in the overriding rules for functions with controlling results.

The reader may recall the general rule in Ada 95 that a function that is a primitive operation of a tagged type and returns a value of the type, must always be overridden when the type is extended. This is because the function for the extended type must create values for the additional components. This rule is sometimes phrased as saying that the function "goes abstract" and so has to be overridden if the extended type is concrete. The irritating thing about the rule in Ada 95 is that it applies even if there are no additional components.

Thus consider a generic version of the set package of Section 3

```
generic
  type Element is private;
package Sets is
  type Set is tagged private;
  function Empty return Set;
  function Unit(E: Element) return Set;
  function Union(S, T: Set) return Set;
  function Intersection(S, T: Set) return Set;
  ...
end Sets;
```

Now suppose we declare an instantiation thus

```
package My_Sets is new Sets(My_Type);
```

This results in the type Set and all its operations being declared inside the package My_Sets. However, for various reasons we might wish to have the type and its operations at the current scope. One reason could just be for simplicity of naming so that we do not have to write My_Sets.Set and My_Sets.Union and so on. (We might be in a regime where use clauses are forbidden.) An obvious approach is to derive our own type locally so that we have

```
package My_Sets is new Sets(My_Type);
type My_Set is new My_Sets.Set with null record;
```

Another situation where we might need to do this is where we wish to use the type Set as the full type for a private type thus

```
  type My_Set is private;
private
  package My_Sets is new Sets(My_Type);
  type My_Set is new My_Sets.Set with null record;
```

But this doesn't work nicely in Ada 95 since all the functions have controlling results and so "go abstract" and therefore have to be overridden with wrappers thus

```
function Union(S, T: My_Set) return My_Set is
begin
  return My_Set(My_Sets.Union(My_Sets.Set(S), My_Sets.Set(T)));
end Union;
```

This is clearly a dreadful nuisance. Ada 2005 sensibly allows the functions to be inherited provided that the extension is visibly null (and that there is no new discriminant part) and so no overriding is required. This new facility will be much appreciated by users of the new container library in Ada 2005 which has just this style of generic packages which export tagged types.

The final topic to be discussed concerns a problem with overloading and untagged types. Remember that the concept of abstract subprograms was introduced into Ada 95 largely for the purpose of tagged types. However it can also be used with untagged types on derivation if we do not want an

operation to be inherited. This often happens with types representing physical measurements. Consider

```
type Length is new Float;
type Area is new Float;
```

These types inherit various undesirable operations such as multiplying a length by a length to give a length when of course we want an area. We can overcome this by overriding them with abstract operations. Thus

```
function "*" (L, R: Length) return Length is abstract;
function "*" (L, R: Area) return Area is abstract;
function "*" (L, R: Length) return Area;
```

We have also declared a function to multiply two lengths to give an area. So now we have two functions multiplying two lengths, one returns a length but is abstract and so can never be called and the other correctly returns an area.

Now suppose we want to print out some values of these types. We might declare a couple of functions delivering a string image thus

```
function Image(L: Length) return String;
function Image(L: Area) return String;
```

And then we decide to write

```
X: Length := 2.5;
...
Put_Line(Image(X * X));                    -- ambiguous in 95
```

This fails to compile in Ada 95 since it is ambiguous because both Image and "*" are overloaded. The problem is that although the function "*" returning a length is abstract it nevertheless is still there and is considered for overload resolution. So we don't know whether we are calling Image on a length or on an area because we don't know which "*" is involved.

So declaring the operation as abstract does not really get rid of the operation at all, it just prevents it from being called but its ghost lives on and is a nuisance.

In Ada 2005 this is overcome by a new rule that says "abstract nondispatching subprograms are ignored during overload resolution". So the abstract "*" is ignored and there is no ambiguity in Ada 2005.

Note that this rule does not apply to dispatching operations of tagged types since we might want to dispatch to a concrete operation of a descendant type. But it does apply to operations of a class-wide type.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] *Ada 95 Rationale* (1995) LNCS 1247, Springer-Verlag.

[3] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

Rationale for Ada 2005: 2 Access types

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements concerning access types for Ada 2005.

Ada 2005 permits all access types to be access to constant types and to indicate that null is not an allowed value in all contexts. Anonymous access types are permitted in more contexts than just as access parameters and discriminants; they can also be used for variables and all components of composite types. This further use of access types is of considerable value in object oriented programming by reducing the need for (unnecessary) explicit type conversions.

A further major improvement concerns access to subprogram types which are now allowed to be anonymous in line with access to object types. This permits so-called "downward closures" and allows the flexible use of procedures as parameters of subprograms and thereby avoids excessive use of generic units.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 2, June 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] does not specifically mention access types as an area needing attention. Access types are, of course, more of a tactical detail than a strategic issue and so this is not surprising.

However, the guidance document strongly emphasizes improvements to object oriented programming and the use of access types figures highly in that area. Indeed one of the motivations for changes was to reduce the number of explicit access type conversions required for OOP.

The guidance document also asks for "improvements that will remedy shortcomings in Ada". The introduction of anonymous access-to-subprogram types comes into that category in the minds of many users.

The following Ada issues cover the relevant changes and are described in detail in this paper:

230 Generalized use of anonymous access types

231 Access to constant parameters, null-excluding types

254 Anonymous access to subprogram types

318 Limited and anonymous access return types

363 Eliminating access subtype problems

382 Current instance rule and anonymous access types

384 Discriminated type conversion rules

385 Stand-alone objects of anonymous access types

392 Prohibit unsafe array conversions

402 Access discriminants of nonlimited types

404 Not null and all in access parameters and types

406 Aliased permitted with anonymous access types

409 Conformance with access to subprogram types

416 Access results, accessibility and return statements

420 Resolution of universal operations in Standard

423 Renaming, null exclusion and formal objects

These changes can be grouped as follows.

First, there is a general orthogonalization of the rules regarding whether the designated type is constant and whether the access subtype includes null (231, part of 404, part of 423).

A major change is the ability to use anonymous access types more widely (230, part of 318, 385, 392, part of 404, 406, part of 416, part of 420). This was found to require some redefinition of the rules regarding the use of a type name within its own definition (382). Access discriminants are now also permitted with nonlimited types (402).

The introduction of anonymous access-to-subprogram types enables local subprograms to be passed as parameters to other subprograms (254, 409). This has been a feature of many other programming languages for over 40 years and its omission from Ada has always been both surprising and irritating and forced the excessive use of generics.

Finally there are some corrections to the rules regarding changing discriminants which prevent attempting to access components of variants that do not exist (363). There is also a change to the rules concerning type conversions and discriminants to make them symmetric (384).

2 Null exclusion and constant

In Ada 95, anonymous access types and named access types have unnecessarily different properties. Furthermore anonymous access types only occur as access parameters and access discriminants.

Anonymous access types in Ada 95 never have null as a value whereas named access types always have null as a value. Suppose we have the following declarations

```
type T is
  record
    Component: Integer;
  end record;

type Ref_T is access T;
T_Ptr: Ref_T;
```

Note that T_Ptr by default will have the value **null**. Now suppose we have a procedure with an access parameter thus

```
procedure P(A: access T) is
  X: Integer;
begin
```

```
   X := A.Component;              -- read a component of A
                                  -- no check for null in 95
   ...
end P;
```

In Ada 95 an access parameter such as A can never have the value null and so there is no need to check for null when doing a dereference such as reading the component A.Component. This is assured by always performing a check when P is called. So calling P with an actual parameter whose value is null such as P(T_Ptr) causes Constraint_Error to be raised at the point of call. The idea was that within P we would have more efficient code for dereferencing and dispatching at the cost of just one check when the procedure is called. Such an access parameter we now refer to as being of a subtype that excludes null.

Ada 2005 extends this idea of access types that exclude null to named access types as well. Thus we can write

```
type Ref_NNT is not null access T;
```

In this case an object of the type Ref_NNT cannot have the value null. An immediate consequence is that all such objects should be explicitly initialized – they will otherwise be initialized to null by default and this will raise Constraint_Error.

Since the property of excluding null can now be given explicitly for named types, it was decided that for uniformity, anonymous access types should follow the same rule whenever possible. So, if we want an access parameter such as A to exclude null in Ada 2005 then we have to indicate this in the same way

```
procedure PNN(A: not null access T) is
   X: Integer;
begin
   X := A.Component;              -- read a component of A
                                  -- no check for null in 2005
   ...
end PNN;
```

This means of course that the original procedure

```
procedure P(A: access T) is
   X: Integer;
begin
   X := A.Component;              -- read a component of A
                                  -- check for null in 2005
   ...
end P;
```

behaves slightly differently in Ada 2005 since A is no longer of a type that excludes null. There now has to be a check when accessing the component of the record because null is now an allowed value of A. So in Ada 2005, calling P with a null parameter results in Constraint_Error being raised within P only when we attempt to do the dereference, whereas in Ada 95 it is always raised at the point of call.

This is of course technically an incompatibility of an unfortunate kind. Here we have a program that is legal in both Ada 95 and Ada 2005 but it behaves differently at execution time in that Constraint_Error is raised at a different place. But of course, in practice if such a program does raise Constraint_Error in this way then it clearly has a bug and so the difference does not really matter.

Various alternative approaches were considered in order to eliminate this incompatibility but they all seemed to be ugly and it was felt that it was best to do the proper thing rather than have a permanent wart.

However the situation regarding controlling access parameters is somewhat different. Remember that a controlling parameter is a parameter of a tagged type where the operation is primitive – that is declared alongside the tagged type in a package specification (or inherited of course). Thus consider

```
package PTT is
  type TT is tagged
    record
      Component: Integer;
    end record;

  procedure Op(X: access TT);              -- primitive operation
  ...
end PTT;
```

The type TT is tagged and the procedure Op is a primitive operation and so the access parameter X is a controlling parameter.

In this case the anonymous access (sub)type still excludes null as in Ada 95 and so null is not permitted as a parameter. The reason is that controlling parameters provide the tag for dispatching and null has no tag value. Remember that all controlling parameters have to have the same tag. We can add **not null** to the parameter specification if we wish but to require it explicitly for all controlling parameters was considered to be too much of an incompatibility. But in newly written programs, we should be encouraged to write **not null** explicitly in order to avoid confusion during maintenance.

Another rule regarding null exclusion is that a type derived from a type that excludes null also excludes null. Thus given

```
type Ref_NNT is not null access T;
type Another_Ref_NNT is new Ref_NNT;
```

then Another_Ref_NNT also excludes null. On the other hand if we start with an access type that does not exclude null then a derived type can exclude null or not thus

```
type Ref_T is access T;
type Another_Ref_T is new Ref_T;
type ANN_Ref_T is new not null Ref_T;
```

then Another_Ref_T does not exclude null but ANN_Ref_T does exclude null.

A technical point is that all access types including anonymous access types in Ada 2005 have null as a value whereas in Ada 95 the anonymous access types did not. It is only subtypes in Ada 2005 that do not always have null as a value. Remember that Ref_NNT is actually a first-named subtype.

An important advantage of all access types having null as a value is that it makes interfacing to C much easier. If a parameter in C has type *t then the corresponding parameter in Ada can have type **access** T and if the C routine needs null passed sometimes then all is well – this was a real pain in Ada 95.

An explicit null exclusion can also be used in object declarations much like a constraint. Thus we can have

```
type Ref_Int is access all Integer;
X: not null Ref_Int := Some_Integer'Access;
```

Note that we must initialize X otherwise the default initialization with **null** will raise Constraint_Error.

In some ways null exclusions have much in common with constraints. We should compare the above with

```
Y: Integer range 1 .. 10;
...
Y := 0;
```

Again Constraint_Error is raised because the value is not permitted for the subtype of Y. A difference however is that in the case of X the check is Access_Check whereas in the case of Y it is Range_Check.

The fact that a null exclusion is not actually classified as a constraint is seen by the syntax for subtype_indication which in Ada 2005 is

```
subtype_indication ::= [null_exclusion] subtype_mark [constraint]
```

An explicit null exclusion can also be used in subprogram declarations thus

```
function F(X: not null Ref_Int) return not null Ref_Int;
procedure P(X: in not null Ref_Int);
procedure Q(X: in out not null Ref_Int);
```

But a difference between null exclusions and constraints is that although we can use a null exclusion in a parameter specification we cannot use a constraint in a parameter specification. Thus

```
procedure P(X: in not null Ref_Int);          -- legal
procedure Q(X: in Integer range 1 .. N);      -- illegal
```

But null exclusions are like constraints in that they are both used in defining subtype conformance and static matching.

We can also use a null exclusion with access-to-subprogram types including protected subprograms.

```
type F is access function (X: Float) return Float;
Fn: not null F := Sqrt'Access;
```

and so on.

A null exclusion can also be used in object and subprogram renamings. We will consider subprogram renamings here and object renamings in the next section when we discuss anonymous access types. This is an area where there is a significant difference between null exclusions and constraints.

Remember that if an entity is renamed then any constraints are unchanged. We might have

```
procedure P(X: Positive);
...
procedure Q(Y: Natural) renames P;
...
Q(0);                                   -- raises Constraint_Error
```

The call of Q raises Constraint_Error because zero is not an allowed value of Positive. The constraint Natural on the renaming is completely ignored (Ada has been like that since time immemorial).

We would have preferred that this sort of peculiar behaviour did not extend to null exclusions. However, we already have the problem that a controlling parameter always excludes null even if it does not say so. So the rule adopted generally with null exclusions is that "null exclusions never lie". In other words, if we give a null exclusion then the entity must exclude null; however, if no null

exclusion is given then the entity might nevertheless exclude null for other reasons (as in the case of a controlling parameter).

So consider

```
procedure P(X: not null access T);
...
procedure Q(Y: access T) renames P;                    -- OK
...
Q(null);          -- raises Constraint_Error
```

The call of Q raises Constraint_Error because the parameter excludes null even though there is no explicit null exclusion in the renaming. On the other hand (we assume that X is not a controlling parameter)

```
procedure P(X: access T);
...
procedure Q(Y: not null access T) renames P;           -- NO
```

is illegal because the null exclusion in the renaming is a lie.

However, if P had been a primitive operation of T so that X was a controlling parameter then the renaming with the null exclusion would be permitted.

Care needs to be taken when a renaming itself is used as a primitive operation. Consider

```
package P is
  type T is tagged ...
  procedure One(X: access T);                  -- excludes null

  package Inner is
    procedure Deux(X: access T);               -- includes null
    procedure Trois(X: not null access T);     -- excludes null
  end Inner;

  use Inner;

  procedure Two(X: access T) renames Deux;           -- NO
  procedure Three(X: access T) renames Trois;        -- OK
  ...
```

The procedure One is a primitive operation of T and its parameter X is therefore a controlling parameter and so excludes null even though this is not explicitly stated. However, the declaration of Two is illegal. It is trying to be a dispatching operation of T and therefore its controlling parameter X has to exclude null. But Two is a renaming of Deux whose corresponding parameter does not exclude null and so the renaming is illegal. On the other hand the declaration of Three is permitted because the parameter of Trois does exclude null.

The other area that needed unification concerned **constant**. In Ada 95 a named access type can be an access to constant type rather than an access to variable type thus

```
type Ref_CT is access constant T;
```

Remember that this means that we cannot change the value of an object of type T via the access type.

Remember also that Ada 95 introduced more general access types whereas in Ada 83 all access types were pool specific and could only access values created by an allocator. An access type in Ada 95 can also refer to any object marked **aliased** provided that the access type is declared with **all** thus

```
type Ref_VT is access all T;
X: aliased T;
R: Ref_VT := X'Access;
```

So in summary, Ada 95 has three kinds of named access types

```
access T;                -- pool specific only, read & write
access all T             -- general, read & write
access constant T        -- general, read only
```

But in Ada 95, the distinction between variable and constant access parameters is not permitted. Ada 2005 rectifies this by permitting **constant** with access parameters. So we can write

```
procedure P(X: access constant T);       -- legal 2005
procedure P(X: access T);
```

Observe however, that **all** is not permitted with access parameters. Ordinary objects can be constant or variable thus

```
C: constant Integer := 99;
V: Integer;
```

and access parameters follow this pattern. It is named access types that are anomalous because of the need to distinguish pool specific types for compatibility with Ada 83 and the subsequent need to introduce **all**.

In summary, Ada 2005 access parameters can take the following four forms

```
procedure P1(X: access T);
procedure P2(X: access constant T);
procedure P3(X: not null access T);
procedure P4(X: not null access constant T);
```

Moreover, as mentioned above, controlling parameters always exclude null even if this is not stated and so in that case P1 and P3 are equivalent. Controlling parameters can also be constant in which case P2 and P4 are equivalent.

Similar rules apply to access discriminants; thus they can exclude null and/or be access to constant.

3 Anonymous access types

As just mentioned, Ada 95 permits anonymous access types only as access parameters and access discriminants. And in the latter case only for limited types. Ada 2005 sweeps away these restrictions and permits anonymous access types quite freely.

The main motivation for this change concerns type conversion. It often happens that we have a type T somewhere in a program and later discover that we need an access type referring to T in some other part of the program. So we introduce

```
type Ref_T is access all T;
```

And then we find that we also need a similar access type somewhere else and so declare another access type

```
type T_Ptr is access all T;
```

If the uses of these two access types overlap then we will find that we have explicit type conversions all over the place despite the fact that they are really the same type. Of course one might argue that planning ahead would help a lot but, as we know, programs often evolve in an unplanned way.

A more important example of the curse of explicit type conversion concerns object oriented programming. Access types feature quite widely in many styles of OO programming. We might have a hierarchy of geometrical object types starting with a root abstract type Object thus

```
type Object is abstract;
type Circle is new Object with ...

type Polygon is new Object with ...
type Pentagon is new Polygon with ...

type Triangle is new Polygon with ...
type Equilateral_Triangle is new Triangle with ...
```

then we might well find ourselves declaring named access types such as

```
type Ref_Object is access all Object'Class;
type Ref_Circle is access all Circle;
type Ref_Triangle is access all Triangle'Class;
type Ref_Equ_Triangle is access all Equilateral_Triangle;
```

Conversion between these clearly ought to be permitted in many cases. In some cases it can never go wrong and in others a run time check is required. Thus a conversion between a Ref_Circle and a Ref_Object is always possible because every value of Ref_Circle is also a value of Ref_Object but the reverse is not the case. So we might have

```
RC: Ref_Circle := A_Circle'Access;
RO: Ref_Object;
...
RO := Ref_Object(RC);          -- explicit conversion, no check
...
RC := Ref_Circle(RO);          -- needs a check
```

However, it is a rule of Ada 95 that type conversions between these named access types have to be explicit and give the type name. This is considered to be a nuisance by many programmers because such conversions are allowed without naming the type in other OO languages. It would not be quite so bad if the explicit conversion were only required in those cases where a run time check was necessary.

Moreover, these are trivial (view) conversions since they are all just pointers and no actual change of value takes place anyway; all that has to be done is to check that the value is a legal reference for the target type and in many cases this is clear at compilation. So requiring the type name is very annoying.

In fact the only conversions between named tagged types (and named access types) that are allowed implicitly in Ada are conversions to a class wide type when it is initialized or when it is a parameter (which is really the same thing).

It would have been nice to have been able to relax the rules in Ada 2005 perhaps by saying that a named conversion is only required when a run time check is required. However, such a change would have caused lots of existing programs to become ambiguous.

So, rather than meddle with the conversion rules, it was instead decided to permit the use of anonymous access types in more contexts in Ada 2005. Anonymous access types have the interesting property that they are anonymous and so necessarily do not have a name that could be used in a conversion. Thus we can have

```
RC: access Circle := A_Circle'Access;
RO: access Object'Class;          -- default null
...
RO := RC;                         -- implicit conversion, no check
```

On the other hand we cannot write

```
RC := RO;                         -- implicit conversion, needs a check
```

because the general rule is that if a check is required then the conversion must be explicit. So typically we will still need to introduce named access types for some conversions.

We can of course also use null exclusions with anonymous access types thus

```
RC: not null access Circle := A_Circle'Access;
RO: not null access Object'Class;          -- careful
```

The declaration of RO is unfortunate because no initial value is given and the default of null is not permitted and so it will raise Constraint_Error; a worthy compiler will detect this during compilation and give us a friendly warning.

Note that we never never write **all** with anonymous access types.

We can of course also use **constant** with anonymous access types. Note carefully the difference between the following

```
ACT: access constant T := T1'Access;
CAT: constant access T := T1'Access;
```

In the first case ACT is a variable and can be used to access different objects T1 and T2 of type T. But it cannot be used to change the value of those objects. In the second case CAT is a constant and can only refer to the object given in its initialization. But we can change the value of the object that CAT refers to. So we have

```
ACT := T2'Access;                 -- legal, can assign
ACT.all := T2;                    -- illegal, constant view
CAT := T2'Access;                 -- illegal, cannot assign
CAT.all := T2;                    -- legal, variable view
```

At first sight this may seem confusing and consideration was given to disallowing the use of constants such as CAT (but permitting ACT which is probably more useful since it protects the accessed value). But the lack of orthogonality was considered very undesirable. Moreover Ada is a left to right language and we are familiar with equivalent constructions such as

```
type CT is access constant T;
ACT: CT;
```

and

```
type AT is access T;
CAT: constant AT;
```

(although the alert reader will note that the latter is illegal because I have foolishly used the reserved word **at** as an identifier).

We can of course also write

```
CACT: constant access constant T := T1'Access;
```

The object CACT is then a constant and provides read-only access to the object T1 it refers to. It cannot be changed to refer to another object such as T2 nor can the value of T1 be changed via CACT.

An object of an anonymous access type, like other objects, can also be declared as aliased thus

```
X: aliased access T;
```

although such constructions are likely to be used rarely.

Anonymous access types can also be used as the components of arrays and records. In the Introduction we saw that rather than having to write

```
type Cell;
type Cell_Ptr is access Cell;

type Cell is
  record
    Next: Cell_Ptr;
    Value: Integer;
  end record;
```

we can simply write

```
type Cell is
  record
    Next: access Cell;
    Value: Integer;
  end record;
```

and this not only avoids have to declare the named access type Cell_Ptr but it also avoids the need for the incomplete type declaration of Cell.

Permitting this required some changes to a rule regarding the use of a type name within its own declaration – the so-called current instance rule.

The original current instance rule was that within a type declaration the type name did not refer to the type itself but to the current object of the type. The following task type declaration illustrates both a legal and illegal use of the task type name within its own declaration. It is essentially an extract from a program in Section 18.10 of [2] which finds prime numbers by a multitasking implementation of the Sieve of Eratosthenes. Each task of the type is associated with a prime number and is responsible for removing multiples of that number and for creating the next task when a new prime number is discovered. It is thus quite natural that the task should need to make a clone of itself.

```
task type TT (P: Integer) is
  ...
end;

type ATT is access TT;

task body TT is
  function Make_Clone(N: Integer) return ATT is
  begin
    return new TT(N);                -- illegal
  end Make_Clone;

  Ref_Clone: ATT;
  ...
```

```
begin
   ...
   Ref_Clone := Make_Clone(N);
   ...
   abort TT;                          -- legal
   ...
end TT;
```

The attempt to make a slave clone of the task in the function Make_Clone is illegal because within the task type its name refers to the current instance and not to the type. However, the abort statement is permitted and will abort the current instance of the task. In this example the solution is simply to move the function Make_Clone outside the task body.

However, this rule would have prevented the use of the type name Cell to declare the component Next within the type Cell and this would have been infuriating since the linked list paradigm is very common.

In order to permit this the current instance rule has been changed in Ada 2005 to allow the type name to denote the type itself within an anonymous access type declaration (but not a named access type declaration). So the type Cell is permitted.

Note however that in Ada 2005, the task TT still cannot contain the declaration of the function Make_Clone. Although we no longer need to declare the named type ATT since we can now declare Ref_Clone as

```
Ref_Clone: access TT;
```

and we can declare the function as

```
function Make_Clone(N: Integer) return access TT is
begin
   return new TT(N);
end Make_Clone;
```

where we have an anonymous result type, nevertheless the allocator **new** TT inside Make_Clone remains illegal if Make_Clone is declared within the task body TT. But such a use is unusual and declaring a distinct external function is hardly a burden.

To be honest we can simply declare a subtype of a different name outside the task

```
subtype XTT is TT;
```

and then we can write **new** XTT(N); in the function and keep the function hidden inside the task. Indeed we don't need the function anyway because we can just write

```
Ref_Clone := new XTT(N);
```

in the task body.

The introduction of the wider use of anonymous access types requires some revision to the rules concerning type comparisons and conversions. This is achieved by the introduction of a type *universal_access* by analogy with the types *universal_integer* and *universal_real*. Two new equality operators are defined in the package Standard thus

```
function "=" (Left, Right: universal_access) return Boolean;

function "/=" (Left, Right: universal_access) return Boolean;
```

The literal **null** is now deemed to be of type *universal_access* and appropriate conversions are defined as well. These new operations are only applied when at least one of the arguments is of an anonymous access types (not counting **null**).

Interesting problems arise if we define our own equality operation. For example, suppose we wish to do a deep comparison on two lists defined by the type Cell. We might decide to write a recursive function with specification

```
function "=" (L, R: access Cell) return Boolean;
```

Note that it is easier to use access parameters rather than parameters of type Cell itself because it then caters naturally for cases where null is used to represent an empty list. We might attempt to write the body as

```
function "=" (L, R: access Cell) return Boolean is
begin
  if L = null or R = null then              -- wrong =
    return L = R;                           -- wrong =
  elsif L.Value = R.Value then
    return L.Next = R.Next;                 -- recurses OK
  else
    return False;
  end if;
end "=" ;
```

But this doesn't work because the calls of "=" in the first two lines recursively call the function being declared whereas we want to call the predefined "=" in these cases.

The difficulty is overcome by writing Standard."=" thus

```
  if Standard."=" (L, null) or Standard."=" (R, null) then
    return Standard."=" (L, R);
```

The full rules regarding the use of the predefined equality are that it cannot be used if there is a user-defined primitive equality operation for either operand type unless we use the prefix Standard. A similar rule applies to fixed point types as we shall see in a later paper.

Another example of the use of the type Cell occurred in the previous paper when we were discussing type extension at nested levels. That example also illustrated that access types have to be named in some circumstances such as when they provide the full type for a private type. We had

```
package Lists is
  type List is limited private;             -- private type
  ...
private
  type Cell is
    record
      Next: access Cell;                    -- anonymous type
      C: Colour;
    end record;

  type List is access Cell;                 -- full type
end;

package body Lists is
  procedure Iterate(IC: in Iterator'Class; L: in List) is
    This: access Cell := L;                 -- anonymous type
  begin
```

```
      while This /= null loop
        IC.Action(This.C);                    -- dispatches
        This := This.Next;
      end loop;
    end Iterate;
  end Lists;
```

In this case we have to name the type List because it is a private type. Nevertheless it is convenient to use an anonymous access type to avoid an incomplete declaration of Cell.

In the procedure Iterate the local variable This is also of an anonymous type. It is interesting to observe that if This had been declared to be of the named type List then we would have needed an explicit conversion in

```
        This := List(This.Next);              -- explicit conversion
```

Remember that we *always* need an explicit conversion when converting to a named access type. There is clearly an art in using anonymous types to best advantage.

The Introduction showed a number of other uses of anonymous access types in arrays and records and as function results when discussing Noah's Ark and other animal situations. We will now turn to more weighty matters.

An important matter in the case of access types is accessibility. The accessibility rules are designed to prevent dangling references. The basic rule is that we cannot create an access value if the object referred to has a lesser lifetime than the access type.

However there are circumstances where the rule is unnecessarily severe and that was one reason for the introduction of access parameters. Perhaps some recapitulation of the problems would be helpful. Consider

```
type T is ...
Global: T;
type Ref_T is access all T;
Dodgy: Ref_T;

procedure P(Ptr: access T) is
begin
  ...
  Dodgy := Ref_T(Ptr);                        -- dynamic check
end P;

procedure Q(Ptr: Ref_T) is
begin
  ...
  Dodgy := Ptr;                               -- legal
end Q;
...
declare
  X: aliased T;
begin
  P(X'Access);                                -- legal
  Q(X'Access);                                -- illegal
end;
```

Here we have an object X with a short lifetime and we must not squirrel away an access referring to X in an object with a longer lifetime such as Dodgy. Nevertheless we want to manipulate X indirectly using a procedure such as P.

If the parameter were of a named type such as Ref_T as in the case of the procedure Q then the call would be illegal since within Q we could then assign to a variable such as Dodgy which would then retain the "address" of X after X had ceased to exist.

However, the procedure P which uses an access parameter permits the call. The reason is that access parameters carry dynamic accessibility information regarding the actual parameter. This extra information enables checks to be performed only if we attempt to do something foolish within the procedure such as make an assignment to Dodgy. The conversion to the type Ref_T in this assignment fails dynamically and disaster is avoided.

But note that if we had called P with

```
P(Global'Access);
```

where Global is declared at the same level as Ref_T then the assignment to Dodgy would be permitted.

The accessibility rules for the new uses of anonymous access types are very simple. The accessibility level is simply the level of the enclosing declaration and no dynamic information is involved. (The possibility of preserving dynamic information was considered but this would have led to inefficiencies at the points of use.)

In the case of a stand-alone variable such as

```
V: access Integer;
```

then this is essentially equivalent to

```
type anon is access all Integer;
V: anon;
```

A similar situation applies in the case of a component of a record or array type. Thus if we have

```
type R is
  record
    C: access Integer;
    ...
  end record;
```

then this is essentially equivalent to

```
type anon is access all Integer;
type R is
  record
    C: anon;
    ...
  end record;
```

Further if we now declare a derived type then there is no new physical access definition, and the accessibility level is that of the original declaration. Thus consider

```
procedure Proc is
  Local: aliased Integer;
  type D is new R;
  X: D := D'(C => Local'Access, ... );          -- illegal
begin
```

```
  ...
end Proc;
```

In this example the accessibility level of the component C of the derived type is the same as that of the parent type R and so the aggregate is illegal. This somewhat surprising rule is necessary to prevent some very strange problems which we will not explore in this paper.

One consequence of which users should be aware is that if we assign the value in an access parameter to a local variable of an anonymous access type then the dynamic accessibility of the actual parameter will not be held in the local variable. Thus consider again the example of the procedure P containing the assignment to Dodgy

```
procedure P(Ptr: access T) is
begin
  ...
  Dodgy := Ref_T(Ptr);                    -- dynamic check
end P;
```

and this variation in which we have introduced a local variable of an anonymous access type

```
procedure P1(Ptr: access T) is
  Local_Ptr: access T;
begin
  ...
  Local_Ptr := Ptr;                       -- implicit conversion
  Dodgy := Ref_T(Local_Ptr);              -- static check, illegal
end P1;
```

Here we have copied the value in the parameter to a local variable before attempting the assignment to Dodgy. (Actually it won't compile but let us analyze it in detail anyway.)

The conversion in P using the access parameter Ptr is dynamic and will only fail if the actual parameter has an accessibility level greater than that of the type Ref_T. So it will fail if the actual parameter is X and so raise Program_Error but will pass if it has the same level as the type Ref_T such as the variable Global.

In the case of P1, the assignment from Ptr to Local_Ptr involves an implicit conversion and static check which always passes. (Remember that implicit conversions are never allowed if they involve a dynamic check.) However, the conversion in the assignment to Dodgy in P1 is also static and will always fail no matter whether X or Global is passed as actual parameter.

So the effective behaviours of P and P1 are the same if the actual parameter is X (they both fail, although one dynamically and the other statically) but will be different if the actual parameter has the same level as the type Ref_T such as the variable Global. The assignment to Dodgy in P will work in the case of Global but the assignment to Dodgy in P1 never works.

This is perhaps surprising, an apparently innocuous intermediate assignment has a significant effect because of the implicit conversion and the consequent loss of the accessibility information. In practice this is very unlikely to be a problem. In any event programmers are aware that access parameters are special and carry dynamic information.

In this particular example the loss of the accessibility information through the use of the intermediate stand-alone variable is detected at compile time. More elaborate examples can be constructed whereby the problem only shows up at execution time. Thus suppose we introduce a third procedure Agent and modify P and P1 so that we have

```
procedure Agent(A: access T) is
begin
  Dodgy := Ref_T(A);                    -- dynamic check
end Agent;

procedure P(Ptr: access T) is
begin
  Agent(Ptr);                           -- may be OK
end P;

procedure P1(Ptr: access T) is
  Local_Ptr: access T;
begin
  Local_Ptr := Ptr;                     -- implicit conversion
  Agent(Local_Ptr);                     -- never OK
end P1;
```

Now we find that P works much as before. The accessibility level passed into P is passed to Agent which then carries out the assignment to Dodgy. If the parameter passed to P is the local X then Program_Error is raised in Agent and propagated to P. If the parameter passed is Global then all is well.

The procedure P1 now compiles whereas it did not before. However, because the accessibility of the original parameter is lost by the assignment to Local_Ptr, it is the accessibility level of Local_Ptr that is passed to Agent and this means that the assignment to Dodgy always fails and raises Program_Error irrespective of whether P1 was called with X or Global.

If we just want to use another name for some reason then we can avoid the loss of the accessibility level by using renaming. Thus we could have

```
procedure P2(Ptr: access T) is
  Local_Ptr: access T renames Ptr;
begin
  ...
  Dodgy := Ref_T(Local_Ptr);            -- dynamic check
end P2;
```

and this will behave exactly as the original procedure P.

As usual a renaming just provides another view of the same entity and thus preserves the accessibility information.

A renaming can also include **not null** thus

```
Local_Ptr: not null access T renames Ptr;
```

Remember that not null must never lie so this is only legal if Ptr is indeed of a type that excludes null (which it will be if Ptr is a controlling access parameter of the procedure P2).

A renaming might be useful when the accessed type T has components that we wish to refer to many times in the procedure. For example the accessed type might be the type Cell declared earlier in which case we might usefully have

```
Next: access Cell renames Ptr.Next;
```

and this will preserve the accessibility information.

Anonymous access types can also be used as the result of a function. In the Introduction we had

```
function Mate_Of(A: access Animal'Class) return access Animal'Class;
```

The accessibility level of the result in this case is the same as that of the declaration of the function itself.

We can also dispatch on the result of a function if the result is an access to a tagged type. Consider

```
function Unit return access T;
```

We can suppose that T is a tagged type representing some category of objects such as our geometrical objects and that Unit is a function returning a unit object such as a circle of unit radius or a triangle with unit side.

We might also have a function

```
function Is_Bigger(X, Y: access T) return Boolean;
```

and then

```
Thing: access T'Class := ... ;
...
Test: Boolean := Is_Bigger(Thing, Unit);
```

This will dispatch to the function Unit according to the tag of Thing and then of course dispatch to the appropriate function Is_Bigger.

The function Unit could also be used as a default value for a parameter thus

```
function Is_Bigger(X: access T; Y: access T := Unit) return Boolean;
```

Remember that a default used in such a construction has to be tag indeterminate.

Permitting anonymous access types as result types eliminates the need to define the concept of a "return by reference" type. This was a strange concept in Ada 95 and primarily concerned limited types (including task and protected types) which of course could not be copied. Enabling us to write **access** explicitly and thereby tell the truth removes much confusion. Limited types will be discussed in detail in a later paper.

Access return types can be a convenient way of getting a constant view of an object such as a table. We might have an array in a package body (or private part) and a function in the specification thus

```
package P is
  type Vector is array (Integer range <>) of Float;

  function Read_Vec return access constant Vector;
  ...
private

end;

package body P is

  The_Vector: aliased Vector :=   ;

  function Read_Vec return access constant Vector is
  begin
    return The_Vector'Access;
  end;
  ...
end P;
```

We can now write

```
X := Read_Vec(7);                    -- read element of array
```

This is strictly short for

```
X := Read_Vec.all(7);
```

Note that we cannot write

```
Read_Vec(7) := Y;                    -- illegal
```

although we could do so if we removed **constant** from the return type (in which case we should use a different name for the function).

The last new use of anonymous access types concerns discriminants. Remember that a discriminant can be of a named access type or an anonymous access type (as well as oher things). Discriminants of an anonymous access type are known as access discriminants. In Ada 95, access discriminants are only allowed with limited types. Discriminants of a named access type are just additional components with no special properties. But access discriminants of limited types are special. Since the type is limited, the object cannot be changed by a whole record assignment and so the discriminant cannot be changed even if it has defaults. Thus

```
type Minor is ...

type Major(M: access Minor) is limited
  record
    ...
  end record;

Small: aliased Minor;
Large: Major(Small'Access);
```

The objects Small and Large are now bound permanently together.

In Ada 2005, access discriminants are also allowed for nonlimited types. However, defaults are not permitted so that the discriminant cannot be changed so again the objects are bound permanently together. An interesting case arises when the discriminant is provided by an allocator thus

```
Larger: Major(new Minor( ... ));
```

In this case we say that the allocated object is a coextension of Larger. Coextensions have the same lifetime as the major object and so are finalized when it is finalized. There are various accessibility and other rules concerning objects which have coextensions which prevent difficulty when returning such objects from functions.

4 Downward closures

This section is really about access to subprogram types in general but the title downward closures has come to epitomize the topic.

The requirements for Ada 83, (Strawman .. Steelman) were strangely silent about whether parameters of subprograms could themselves be subprograms as was the case in Algol 60 and Pascal. Remember that Pascal was one of the languages on which the designs for the DoD language were to be based.

The predictability aspects of the requirements were interpreted as implying that all subprogram calls should be identified at compilation time on the grounds that if you didn't know what was being called than you couldn't know what the program was going to do. This was a particularly stupid attitude to take. The question of predictability (presumably in some safety or security context) really concerns the behaviour of particular programs rather than the universe of all programs that can be constructed in a language.

In any event the totality of subprograms that might be called in a program is finite and closed. It simply consists of the subprograms in the program. Languages such as Ada are not able to construct totally new subprograms out of lesser components in the way that they can create say floating point values.

So the world had to use generics for many applications that were natural for subprograms as parameters of other subprograms. Thankfully many implementers avoided the explosion that might occur with generics by clever code sharing which in a sense hid the parameterization behind the scenes.

The types of applications for which subprograms are natural as parameters are any where one subroutine is parameterized by another. They include many mathematical applications such as integration and maximization and more logical applications such as sorting and searching and iterating.

As outlined in the Introduction, the matter was partly improved in Ada 95 by the introduction of named access-to-subprogram types. This was essentially done to allow program call back to be implemented.

Program call back is when one program passes the "address" of a subprogram within it to another program so that this other program can later respond by calling back to the first program using the subprogram address supplied. This is often used for communication between an Ada application program and some other software such as an operating system which might even be written in another language such as C.

Named access to subprogram types certainly work for call back (especially with languages such as C that do not have nested subprograms) but the accessibility rules which followed those for general access to object types were restrictive. For example, suppose we have a general library level function for integration using a named access to subprogram type to pass the function to be integrated thus

```
type Integrand is access function(X: Float) return Float;

function Integrate(Fn: Integrand; Lo, Hi: Float) return Float;
```

then we cannot even do the simplest integration of our own function in a natural way. For example, suppose we wish to integrate a function such as Exp(X**2). We can try

```
with Integrate;
procedure Main is
  function F(X: Float) return Float is
  begin
    return Exp(X**2);
  end F;

  Result, L, H: Float;
begin
  ...                    -- set bounds in L and H say
  Result := Integrate(F'Access, L, H);        -- illegal in 95
  ...
end Main
```

But this is illegal because of the accessibility check necessary to prevent us from writing something like

```
Evil: Integrand;
X: Float;
...
declare
  Y: Float;
  function F(X: Float) return Float is
    ...
    Y := X;                          --assign to Y in local block
    ...
  end F;
begin
  Evil := F'Access: -- illegal
end;
  X := Evil(X);                      -- call function out of context
```

Here we have attempted to assign an access to the local function F in the global variable Evil. If this assignment had been permitted then the call of Evil would indirectly have called the function F when the context in which F was declared no longer existed; F would then have attempted to assign to the variable Y which no longer existed and whose storage space might now be used for something else. We can summarise this perhaps by saying that we are attempting to call F when it no longer exists.

Ada 2005 overcomes the problem by introducing anonymous access to subprogram types. This was actually considered during the design of Ada 95 but it was not done at the time for two main reasons. Firstly, the implementation problems for those who were using display vectors rather than static links were considered a hurdle. And secondly, a crafty technique was available using the newly introduced tagged types. And of course one could continue to use generics. But further thought showed that the implementation burden was not so great after all and nobody understood the tagged type technique which was really incredibly contorted. Moreover, the continued use of generics when other languages forty years ago had included a more natural mechanism was tiresome. So at long last Ada 2005 includes anonymous access to subprogram types.

We rewrite the integration function much as follows

```
function Integrate(Fn: access function(X: Float) return Float;
                       Lo, Hi: Float) return Float is
  Total: Float;
  N: constant Integer := ... ;          -- no of subdivisions
  Step: Float := (Hi – Lo) / Float(N);
  X: Float := Lo;                       -- current point
begin
  Total := 0.5 * Fn(Lo);                -- value at low bound
  for I in 1 .. N–1 loop
    X := X + Step;                      -- add values at
    Total := Total + Fn(X);             -- intermediate points
  end loop;
  Total := Total + 0.5 * Fn(Hi);        -- add final value
  return Total * Step;                  -- normalize
end Integrate;
```

The important thing to notice is the profile of Integrate in which the parameter Fn is of an anonymous access to subprogram type. We have also shown a simple body which uses the trapezium/trapezoid method and so calls the actual function corresponding to Fn at the two end points of the range and at a number of equally spaced intermediate points.

(NB It is time for a linguistic interlude. Roughly speaking English English trapezium equals US English trapezoid. They both originate from the Greek τραπεζα meaning a table (literally with four feet). Both originally meant a quadrilateral with no pairs of sides parallel. In the late 17th century, trapezium came to mean having one pair of sides parallel. In the 18th century trapezoid came to mean the same as trapezium but promptly faded out of use in England whereas in the US it continues in use. Meanwhile in the US, trapezium reverted to its original meaning of totally irregular. Trapezoid is rarely used in the UK but if used has reverted to its original meaning of totally irregular. A standard language would be useful. Anyway, the integration is using quadrilateral strips with one pair of sides parallel.)

With this new declaration of Integrate, the accessibility problems are overcome and we are allowed to write Integrate(F'Access, ...) just as we could write P(X'Access) in the example in the previous section where we discussed anonymous access to object types.

We still have to consider how a type conversion which would permit an assignment to a global variable is prevented. The following text illustrates both access to object and access to subprogram parameters.

```
type AOT is access all Integer;
type APT is access procedure (X: in out Float);

Evil_Obj: AOT;
Evil_Proc: APT;

procedure P(Objptr: access Integer;
            Procptr: access procedure (X: in out Float)) is
begin
  Evil_Obj := AOT(Objptr);              -- fails at run time
  Evil_Proc := APT(Procptr);            -- fails at compile time
end P;

declare
  An_Obj: aliased Integer;
  procedure A_Proc(X: in out Float) is
  begin ... end A_Proc;
begin
  P(An_Obj'Access, A_Proc'Access);      -- legal
end;

Evil_Obj.all := 0;                      -- assign to nowhere
Evil_Proc.all( ... );                   -- call nowhere
```

This repeats some of the structure of the previous section. The procedure P has an access to object parameter Objptr and an access to subprogram parameter Procptr; they are both of anonymous type. The call of P in the local block passes the addresses of a local object An_Obj and a local procedure A_Proc to P. This is permitted. We now attempt to assign the parameter values from within P to global objects Evil_Obj and Evil_Proc with the intent of assigning indirectly via Evil_Obj and calling indirectly via Evil_Proc after the object and procedure referred to no longer exist.

Both of these wicked deeds are prevented by the accessibility rules.

In the case of the object parameter Objptr it knows the accessibility level of the actual An_Obj and this is seen to be greater than that of the type AOT and so the conversion is prevented at run time and in fact Program_Error is raised. But if An_Obj had been declared at the same level as AOT and not within an inner block then the conversion would have been permitted.

However, somewhat different rules apply to anonymous access to subprogram parameters. They do not carry an indication of the accessibility level of the actual parameter but simply treat it as if it were infinite (strictly – deeper than anything else). This of course prevents the conversion to the type APT and all is well; this is detected at compile time. But note that if the procedure A_Proc had been declared at the same level as APT then the conversion would still have failed because the accessibility level is treated as infinite.

There are a number of reasons for the different treatment of anonymous access to subprogram types. A big problem is that named access to subprogram types are implemented in the same way as C *func in almost all compilers. Permitting the conversion from anonymous access to subprogram types to named ones would thus have caused problems because that model does not work especially for display based implementations. Carrying the accessibility level around would not have prevented these conversions. The key goal was simply to provide a facility corresponding to that in Pascal and not to encourage too much fooling about with access to subprogram types. Recall that the attribute Unchecked_Access is permitted for access to object types but was considered far too dangerous for access to subprogram types for similar reasons.

The reader may be feeling both tired and that there are other ways around the problems of accessibility anyway. Thus the double integration presented in the Introduction can easily be circumvented in many cases. We computed

$$\int_0^1 \int_0^1 xy \, dy \, dx$$

using the following program

```
with Integrate;
procedure Main is
  function G(X: Float) return Float is
    function F(Y: Float) return Float is
    begin
      return X*Y;
    end F;
  begin
    return Integrate(F'Access, 0.0, 1.0);
  end G;

  Result: Float;
begin
  Result:= Integrate(G'Access, 0.0, 1.0);
  ...
end Main;
```

The essence of the problem was that F had to be declared inside G because it needed access to the parameter X of G. But the astute reader will note that this example is not very convincing because the integrals can be separated and the functions both declared at library level thus

```
function F(Y: Float) return Float is
begin
  return Y;
end F;
```

```
function G(X: Float) return Float is
begin
  return X;
end G;

Result:= Integrate(F'Access, 0.0, 1.0) * Integrate(G'Access, 0.0, 1.0);
```

and so it all works using the Ada 95 version of Integrate anyway.

However, if the two integrals had been more convoluted or perhaps the region had not been square but triangular so that the bound of the inner integral depended on the outer variable as in

$$\int_0^1 \int_0^x xy \, dy \, dx$$

then nested functions would be vital.

We will now consider a more elegant example which illustrates how we might integrate an arbitrary function of two variables $F(x, y)$ over a rectangular region.

Assume that we have the function Integrate for one dimension as before

```
function Integrate(Fn: access function(X: Float) return Float;
                   Lo, Hi: Float) return Float;
```

Now consider

```
function Integrate(Fn: access function(X, Y: Float) return Float;
                   LoX, HiX: Float;
                   LoY, HiY: Float) return Float is
  function FnX(X: Float) return Float is
    function FnY(Y: Float) return Float is
    begin
      return Fn(X, Y);
    end FnY;
  begin
    return Integrate(FnY'Access, LoY, HiY);
  end FnX;
begin
  return Integrate(FnX'Access, LoX, HiX);
end integrate;
```

The new function Integrate for two dimensions overloads and uses the function Integrate for one dimension (a good example of overloading). With this generality it is again impossible to arrange the structure in a manner which is legal in Ada 95.

We might use the two-dimensional integration routine to solve the original trivial problem as follows

```
function F(X, Y: Float) return Float is
begin
  return X*Y;
end F;
...

Result := Integrate(F'Access, 0.0, 1.0, 0.0, 1.0);
```

As an exercise the reader might like to rewrite the two dimensional function to work on a non-rectangular domain. The trick is to pass the bounds of the inner integral also as functions. The profile then becomes

```
function Integrate(Fn: access function(X, Y: Float) return Float;
                   LoX, HiX: Float
                   LoY, HiY: access function(X: Float) return Float)
                                                        return Float;
```

In case the reader should think that this topic is all too mathematical it should be pointed out that anonymous access to subprogram parameters are widely used in the new container library thereby saving the unnecessary use of generics.

For example the package Ada.Containers.Vectors declares procedures such as

```
procedure Update_Element
  (Container: in Vector; Index: in Index_Type;
   Process: not null access procedure (Element: in out Element_Type));
```

This updates the element of the vector Container whose index is Index by calling the procedure Process with that element as parameter. Thus if we have a vector of integers V and we need to double the value of those with index in the range 5 to 10, then we would first declare a procedure such as

```
procedure Double(E: in out Integer) is
begin
  E := 2 * E;
end Double ;
```

and then write

```
for I in 5 .. 10 loop
  Update_Element(V, I, Double'Access);
end loop;
```

Further details of the use of access to subprogram types with containers will be found in a later paper.

Finally it should be noted that anonymous access to subprogram types can also be used in all those places where anonymous access to object types are allowed. That is as stand-alone objects, as components of arrays and records, as function results, in renamings, and in access discriminants.

The reader who likes long sequences of reserved words should realise by now that there is no limit in Ada 2005. This is because a function without parameters can return an access to function as its result and this in turn could be of a similar kind. So we would have

```
type FF is access function return access function return access function ...
```

Attempts to compile such an access to function type will inevitably lead to madness.

5 Access types and discriminants

This final topic concerns two matters. The first is about accessing components of discriminated types that might vanish or change mysteriously and the second is about type conversions.

Recall that we can have a mutable variant record such as

```
type Gender is (Male, Female, Neuter);
```

```
type Mutant(Sex: Gender := Neuter) is
  record
    Birth: Date;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Children: Integer;
      when Neuter =>
        null;
    end case;
  end record;
```

This represents a world in which there are three sexes, males which can have beards, females which can bear children, and neuters which are fairly useless. Note the default value for the discriminant. This means that if we declare an unconstrained object thus

```
The_Thing: Mutant;
```

then The_Thing is neuter by default but could have its sex changed by a whole record assignment thus

```
The_Thing := (Male, The_Thing.Birth, True);
```

It now is Male and has a beard but the date of birth retains its previous value.

The problem with this sort of object is that components can disappear. If it were changed to be Female then the beard would vanish and be replaced by children. Because of this ghostly behaviour certain operations on mutable objects are forbidden.

One obvious rule is that it is not permissible to rename components which might vanish. So

```
Hairy: Boolean renames The_Thing.Bearded;        -- illegal
```

is not permitted. This was an Ada 83 rule. It was probably the case that the rules were watertight in Ada 83. However, Ada 95 introduced many more possibilities. Objects and components could be marked as **aliased** and the Access attribute could be applied. Additional rules were then added to prevent creating references to things that could vanish.

However, it was then discovered that the rules in Ada 95 regarding access types were not watertight. Accordingly various attempts were made to fix them in a somewhat piecemeal fashion. The problems are subtle and do not seem worth describing in their entirety in this general presentation. We will content ourselves with just a couple of examples.

In Ada 95 we can declare types such as

```
type Mutant_Name is access all Mutant;
type Things_Name is access all Mutant(Neuter);
```

Naturally enough an object of type Things_Name can only be permitted to reference a Mutant whose Sex is Neuter.

```
Some_Thing: aliased Mutant;
Thing_Ptr: Things_Name := Some_Thing'Access;
```

Things would now go wrong if we allowed Some_Thing to have a sex change. Accordingly there is a rule in Ada 95 that says that an aliased object such as Some_Thing is considered to be constrained. So that is quite safe.

However, matters get more difficult when a type such as Mutant is used for a component of another type such as

```
type Monster is
  record
    Head: Mutant(Female);
    Tail: aliased Mutant;
  end record;
```

Here we are attempting to declare a nightmare monster whose head is a female but whose tail is deceivingly mutable. Those with a decent education might find that this reminds them of the Sirens who tempted Odysseus by their beautiful voices on his trip past the monster Scylla and the whirlpool Charybdis. Those with an indecent education can compare it to a pantomime theatre horse (or mare, maybe indeed a nightmare). We could then write

```
M: Monster;
Thing_Ptr := Monster.Tail'Access;
```

However, there is an Ada 95 rule that says that the Tail has to be constrained since it is aliased so the type Monster is not allowed. So far so good.

But now consider the following very nasty example

```
generic
  type T is private;
  Before, After: T;
  type Name is access all T;
  A_Name: in out Name;
procedure Sex_Change;

procedure Sex_Change is
  type Single is array (1..1) of aliased T;
  X: Single := (1 => Before);
begin
  A_Name := X(1)'Access;
  X := (1 => After);
end Sex_Change;
```

and then

```
A_Neuter: Mutant_Name(Neuter);          -- fixed neuter

procedure Surgery is new Sex_Change(
                    T => Mutant,
                    Before => (Sex => Neuter),
                    After => (Sex => Male, Bearded, True),
                    Name => Mutant_Name,
                    A_Name => A_Neuter);

Surgery;                                -- call of Surgery makes A_Neuter hairy
```

The problem here is that there are loopholes in the checks in the procedure Sex_Change. The object A_Name is assigned an access to the single component of the array X whose value is Before. When this is done there is a check that the component of the array has the correct subtype. However the subsequent assignment to the whole array changes the value of the component to After and this can change the subtype of X(1) surreptitiously and there is no check concerning A_Name. The key point is that the generic doesn't know that the type T is mutable; this information is not part of the generic contract.

So when we call Surgery, the object A_Neuter suddenly finds that it has grown a beard!

A similar difficulty occurs when private types are involved because the partial view and full view might disagree about whether the type is constrained or not. Consider

```
package Beings is
  type Mutant is private;
  type Mutant_Name is access Mutant;
  F, M: constant Mutant;
private
  type Mutant(Sex: Gender := Neuter) is
    record
      ...                        -- as above
    end record;

  F: constant Mutant := (Female, ... );
  M: constant Mutant := (Male, ... );
end Beings;
```

Now suppose some innocent user (who has not peeked at the private part) writes

```
Chris: Mutant_Name := new Mutant'(F);      -- OK
...
Chris.all := M;                            -- raises Constraint_Error
```

This is very surprising. The user cannot see that the type Mutant is mutable and in particular cannot see that M and F are different in some way. From the outside they just look like constants of the same type. The big trouble is that there is a rule in Ada 95 that says that an object created by an allocator is constrained. So the new object referred to by Chris is permanently Female and therefore the attempt to assign the value of M with its Bearded component to her is doomed.

Attempting to fix these and related problems with a number of minimal rules seemed fated not to succeed. In the end the approach has been taken of getting to the root of the matter in Ada 2005 and disallowing access subtypes for general access types that have defaults for their discriminants. So both the explicit Things_Name and also Mutant_Name(Neuter) are forbidden in Ada 2005.

Moreover we cannot even have an access type such as Mutant_Name when the access type completes a private view that has no discriminants.

By removing these nasty access subtypes it is now possible to say that heap objects are no longer considered constrained in this situation.

The other change in this area concerns type conversions. A variation on the gender theme is illustrated by the following

```
type Gender is (Male, Female);

type Person(Sex: Gender) is
  record
    Birth: Date;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Children: Integer;
    end case;
  end record;
```

Note that this type is not mutable so all persons are stuck with their sex from birth.

We might now declare some access types

```
type Person_Name is access all Person;
type Mans_Name is access all Person(Male);
type Womans_Name is access all Person(Female);
```

so that we can manipulate various names of people. We would naturally use Person_Name if we did not know the sex of the person and otherwise use Mans_Name or Womans_Name as appropriate. We might have

```
It: Person_Name := Chris'Access;
Him: Mans_Name := Jack'Access;
Her: Womans_Name := Jill'Access;
```

If we later discover that Chris is actually Christine then we might like to assign the value in It to a more appropriate variable such as Her. So we would like to write

```
Her := Womans_Name(It);
```

But curiously enough this is not permitted in Ada 95 although the reverse conversion

```
It := Person_Name(Her);
```

is permitted. The Ada 95 rule is that any constraints have to statically match or the conversion has to be to an unconstrained type. Presumably the reason was to avoid checks at run time. But this lack of symmetry is unpleasant and the rule has been changed in Ada 2005 to allow conversion in both directions with a run time check as necessary.

The above example is actually Exercise 19.8(1) in the textbook [2]. The poor student was invited to solve an impossible problem. But they will be successful in Ada 2005.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

Rationale for Ada 2005: 3 Structure and visibility

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements in the areas of structure and visibility for Ada 2005.

The most important improvement is perhaps the introduction of limited with clauses which permit types in two packages to refer to each other. A related addition to context clauses is the private with clause which just provides access from a private part.

There are also important improvements to limited types which make them much more useful; these include initialization with aggregates and composition using a new form of return statement.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 2, June 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] identifies the solution of the problem of mutually dependent types as one of the two specific issues that need to be addressed in devising Ada 2005.

Moreover the guidance document also emphasizes

> Improvements that will remedy shortcomings in Ada. It cites in particular improvements in OO features, specifically, adding a Java-like interface feature and improved interfacing to other OO languages.

OO is largely about structure and visibility and so further improvements and in particular those that remedy shortcomings are desirable.

The following Ada issues cover the relevant changes and are described in detail in this paper:

217 Mutually recursive types – limited with

262 Access to private units in the private part

287 Limited aggregates allowed

318 Limited and anonymous access return types

326 Tagged incomplete types

412 Subtypes and renamings of incomplete entities

These changes can be grouped as follows.

First there is the important solution to the problem of mutually dependent types across packages provided by the introduction of limited with clauses (217). Related changes are the introduction of

tagged incomplete types (326) and the ability to have subtypes and renamings of incomplete views (412).

Another improvement to the visibility rules is the introduction of private with clauses (262).

There are some changes to aggregates. These were triggered by problems with limited types but apply to aggregates in general (part of 287).

An important area is that of limited types which are somewhat confused in Ada 95. There are two changes which permit limited values to be built *in situ*. One is the use of aggregates for initialization and the other is a more elaborate return statement which enables the construction of limited values when returning from a function (287, 318).

2 Mutually dependent types

For many programmers the solution of the problem of mutually dependent types will be the single most important improvement introduced in Ada 2005.

This topic was discussed in the Introduction using an example of two mutually dependent types, Point and Line. Each type needed to refer to the other in its declaration and of course the solution to this problem is to use incomplete types. In Ada 95 there are three stages. We first declare the incomplete types

```
type Point;                    -- incomplete types
type Line;
```

Suppose for simplicity that we wish to study patterns of points and lines such that each point has exactly three lines through it and that each line has exactly three points on it. (This is not so stupid. The two most fundamental theorems of projective geometry, those of Pappus and Desargues, concern such structures and so does the simplest of finite geometries, the Fano plane.)

Using the incomplete types we can then declare

```
type Point_Ptr is access Point;    -- use incomplete types
type Line_Ptr is access Line;
```

and finally we can complete the type declarations thus

```
type Point is                      -- complete the types
  record
    L, M, N: Line_Ptr;
  end record;

type Line is
  record
    P, Q, R: Point_Ptr;
  end record;
```

Of course, in Ada 2005, as discussed in the previous paper, we can use anonymous access types more freely so that the second stage can be omitted in this example. As a consequence the complete declarations are simply

```
type Point is                      -- complete the types
  record
    L, M, N: access Line;
  end record;

type Line is
  record
```

```
    P, Q, R: access Point;
  end record;
```

This has the important advantage that we do not have to invent irritating identifiers such as Point_Ptr.

But we will stick to Ada 95 for the moment. In Ada 95 there are two rules

- the incomplete type can only be used in the definition of access types;
- the complete type declaration must be in the same declarative region as the incomplete type.

The first rule does actually permit

```
type T;
type A is access procedure (X: in out T);
```

Note that we are here using the incomplete type T for a parameter. This is not normally allowed, but in this case the procedure itself is being used in an access type. The additional level of indirection means that the fact that the parameter mechanism for T is not known yet does not matter.

Apart from this, it is not possible to use an incomplete type for a parameter in a subprogram in Ada 95 except in the case of an access parameter. Thus we cannot have

```
function Is_Point_On_Line(P: Point; L: Line) return Boolean;
```

before the complete type declarations.

It is also worth pointing out that the problem of mutually dependent types (within a single unit) can often be solved by using private types thus

```
  type Point is private;
  type Point_Ptr is access Point;
  type Line is private;
  type Line_Ptr is access Line;
private

  type Point is
    record
      L, M, N: Line_Ptr;
    end record;

  type Line is
    record
      P, Q, R: Point_Ptr;
    end record;
```

But we need to use incomplete types if we want the user to see the full view of a type so the situation is somewhat different.

As an aside, remember that if an incomplete type is declared in a private part then the complete type can be deferred to the body (this is the so-called Taft Amendment in Ada 83). In this case neither the user nor indeed the compiler can see the complete type and this is the main reason why we cannot have parameters of incomplete types whereas we can for private types.

We will now introduce what has become a canonical example for discussing this topic. This concerns employees and the departments of the organization in which they work. The information about employees needs to refer to the departments and the departments need to refer to the employees. We assume that the material regarding employees and departments is quite large so that

we naturally wish to declare the two types in distinct packages Employees and Departments. So we would like to say

```
with Departments;  use Departments;
package Employees is
  type Employee is private;
  procedure Assign_Employee(E: in out Employee; D: in out Department);
  type Dept_Ptr is access all Department;
  function Current_Department(E: Employee) return Dept_Ptr;
  ...
end Employees;
```

```
with Employees;  use Employees;
package Departments is
  type Department is private;
  procedure Choose_Manager(D: in out Department; M: in out Employee);
  ...
end Departments;
```

We cannot write this because each package has a with clause for the other and they cannot both be declared (or entered into the library) first.

We assume of course that the type Employee includes information about the Department for whom the Employee works and the type Department contains information regarding the manager of the department and presumably a list of the other employees as well – note that the manager is naturally also an Employee.

So in Ada 95 we are forced to put everything into one package thus

```
package Workplace is
  type Employee is private;
  type Department is private;
  procedure Assign_Employee(E: in out Employee; D: in out Department);
  type Dept_Ptr is access all Department;
  function Current_Department(E: Employee) return Dept_Ptr;
  procedure Choose_Manager(D: in out Department; M: in out Employee);
private
  ...
end Workplace;
```

Not only does this give rise to huge cumbersome packages but it also prevents us from using the proper abstractions. Thus the types Employee and Department have to be declared in the same private part and so are not protected from each other's operations.

Ada 2005 solves this by introducing a variation of the with clause – the limited with clause. A limited with clause enables a library unit to have an incomplete view of all the visible types in another package. We can now write

```
limited with Departments;
package Employees is
  type Employee is private;
  procedure Assign_Employee(E: in out Employee; D: access Departments.Department);
  type Dept_Ptr is access all Departments.Department;
  function Current_Department(E: Employee) return Dept_Ptr;
  ...
end Employees;
```

```
limited with Employees;
package Departments is
  type Department is private;
  procedure Choose_Manager(D: in out Department; M: access Employees.Employee);
  ...
end Departments;
```

It is important to understand that a limited with clause does not impose a dependence. Thus if a package A has a limited with clause for B, then A does not depend on B as it would with a normal with clause, and so B does not have to be compiled before A or placed into the library before A.

If we have a cycle of packages we only have to put **limited with** on one package since that is sufficient to break the cycle of dependences. However, for symmetry, in this example we have made them both have a limited view of each other.

Note the terminology: we say that we have a limited view of a package if the view is provided through a limited with clause. So a limited view of a package provides an incomplete view of its visible types. And by an incomplete view we mean as if they were incomplete types.

In the example, because an incomplete view of a type cannot generally be used as a parameter, we have had to change one parameter of each of Assign_Employee and Choose_Manager to be an access parameter.

There are a number of rules necessary to avoid problems. A natural one is that we cannot have both a limited with clause and a normal with clause for the same package in the same context clause (a normal with clause is now officially referred to as a nonlimited with clause). An important and perhaps unexpected rule is that we cannot have a use package clause with a limited view because severe surprises might happen.

To understand how this could be possible it is important to realise that a limited with clause provides a very restricted view of a package. It just makes visible

- the name of the package and packages nested within,
- an incomplete view of the types declared in the visible parts of the packages.

Nothing else is visible at all. Now consider

```
package A is
  X: Integer := 99;
end A;

package B is
  X: Integer := 111;
end B;

limited with A, B;
package P is
  ...                                   -- neither X visible here
end P;
```

Within package P we cannot access A.X or B.X because they are not types but objects. But we could declare a child package with its own with clause thus

```
with A;
package P.C is
  Y: Integer := A.X;
end P.C;
```

The nonlimited with clause on the child "overrides" the limited with clause on the parent so that A.X is visible.

Now suppose we were allowed to add a use package clause to the parent package; since a use clause on a parent applies to a child this means that we could refer to A.X as just X within the child so we would have

```
limited with A, B;
use A, B;                       -- illegal
package P is
  ...                           -- neither X visible here
end P;

with A;
package P.C is
  Y: Integer := X;              -- A.X now visible as just X
end P.C;
```

If we were now to change the with clause on the child to refer to B instead of A, then X would refer to B.X rather than A.X. This would not be at all obvious because the use clause that permits this is on the parent and we are not changing the context clause of the parent at all. This would clearly be unacceptable and so use package clauses are forbidden if we only have a limited view of the package.

Here is a reasonably complete list of the rules designed to prevent misadventure when using limited with clauses

- a use package clause cannot refer to a package with a limited view as illustrated above,

  ```
  limited with P;  use P;       -- illegal
  package Q is ...
  ```

  the rule also prevents

  ```
  limited with P;
  package Q is
    use P;                      -- illegal
  ```

- a limited with clause can only appear on a specification – it cannot appear on a body or a subunit,

  ```
  limited with P;               -- illegal
  package body Q is ...
  ```

- a limited with clause and a nonlimited with clause for the same package may not appear in the same context clause,

  ```
  limited with P; with P;       -- illegal
  ```

- a limited with clause and a use clause for the same package or one of its children may not appear in the same context clause,

  ```
  limited with P; use P.C;      -- illegal
  ```

- a limited with clause may not appear in the context clause applying to itself,

  ```
  limited with P;               -- illegal
  package P is ...
  ```

- a limited with clause may not appear on a child unit if a nonlimited with clause for the same package applies to its parent or grandparent etc,

```
with Q;
package P is ...

limited with Q;                    -- illegal
package P.C is ...
```

but note that the reverse is allowed as mentioned above

```
limited with Q;
package P is ...

with Q;                            -- OK
package P.C is ...
```

- a limited with clause may not appear in the scope of a use clause which names the unit or one of its children,

```
with A;
package P is
  package R renames A;
end P;

with P;
package Q is
  use P.R;                         -- applies to A
end Q;

limited with A;                    -- illegal
package Q.C is ...
```

without this specific rule, the use clause in Q which actually refers to A would clash with the limited with clause for A.

Finally note that a limited with clause can only refer to a package declaration and not to a subprogram, generic declaration or instantiation, or to a package renaming.

We will now return to the rules for incomplete types. As mentioned above the rules for incomplete types are quite strict in Ada 95 and apart from the curious case of an access to subprogram type it is not possible to use an incomplete type for a parameter other than in an access parameter.

Ada 2005 enables some relaxation of these rules by introducing tagged incomplete types. We can write

```
type T is tagged;
```

and then the complete type must be a tagged type. Of course the reverse does not hold. If we have just

```
type T;
```

then the complete type T might be tagged or not.

A curious feature of Ada 95 was mentioned in the Introduction. In Ada 95 we can write

```
type T;
...
type T_Ptr is access all T'Class;
```

By using the attribute Class, this promises in a rather sly way that the complete type T will be tagged. This is strictly obsolescent in Ada 2005 and moved to Annex J. In Ada 2005 we should write

```
type T is tagged;
...
type T_Ptr is access all T'Class;
```

The big advantage of introducing tagged incomplete types is that we know that tagged types are always passed by reference and so we are allowed to use tagged incomplete types for parameters.

This advantage extends to the incomplete view obtained from a limited with clause. If a type in a package is visibly tagged then the incomplete view obtained is tagged incomplete and so the type can then be used for parameters.

Returning to the packages Employees and Departments it probably makes sense to make both types tagged since it is likely that the types Employee and Department form a hierarchy. So we can write

```
limited with Departments;
package Employees is
  type Employee is tagged private;
  procedure Assign_Employee(E: in out Employee; D: in out Departments.Department'Class);
  type Dept_Ptr is access all Departments.Department'Class;
  function Current_Department(E: Employee) return Dept_Ptr;
  ...
end Employees;
```

```
limited with Employees;
package Departments is
  type Department is tagged private;
  procedure Choose_Manager(D: in out Department; M: in out Employees.Employee'Class);
  ...
end Departments;
```

The text is a bit cumbersome now with Class sprinkled liberally around but we can introduce some subtypes in order to shorten the names. We can also avoid the introduction of the type Dept_Ptr since we can use an anonymous access type for the function result as mentioned in the previous paper. So we get

```
limited with Departments;
package Employees is
  type Employee is tagged private;
  subtype Dept is Departments.Department;
  procedure Assign_Employee(E: in out Employee; D: in out Dept'Class);
  function Current_Department(E: Employee) return access Dept'Class;
  ...
end Employees;
```

```
limited with Employees;
package Departments is
  type Department is tagged private;
  subtype Empl is Employees.Employee;
  procedure Choose_Manager(D: in out Department; M: in out Empl'Class);
  ...
end Departments;
```

Observe that in Ada 2005 we can use a simple subtype as an abbreviation for an incomplete type thus

```
subtype Dept is Departments.Department;
```

but such a subtype cannot have a constraint or a null exclusion. In essence it is just a renaming. Remember that we cannot have a use clause with a limited view. Moreover, many projects forbid use clauses anyway but permit renamings and subtypes for local abbreviations. It would be a pain if such abbreviations were not also available when using a limited with clause.

It's a pity we cannot also write

```
subtype A_Dept is Departments.Department'Class;
```

but then you cannot have everything in life.

A similar situation arises with the names of nested packages. They can be renamed in order to provide an abbreviation.

The mechanism for breaking cycles of dependences by introducing limited with clauses does not mean that the implementation does not check everything thoroughly in a rigorous Ada way. It is just that some checks might have to be deferred. The details depend upon the implementation.

For the human reader it is very helpful that use clauses are not allowed in conjunction with limited with clauses since it eliminates any doubt about the location of types involved. It probably helps the poor compilers as well.

Readers might be interested to know that this topic was one of the most difficult to solve satisfactorily in the design of Ada 2005. Altogether seven different versions of AI-217 were developed. This chosen solution is on reflection by far the best and was in fact number 6.

A number of loopholes in Ada 95 regarding incomplete types are also closed in Ada 2005.

One such loophole is illustrated by the following (this is Ada 95)

```
package P is
  ...
private
  type T;                                -- an incomplete type
  type ATC is access all T'Class;        -- it must be tagged
  X: ATC;
  procedure Op(X: access T);             -- primitive operation
  ...
end P;
```

The incomplete type T is declared in the private part of the package P. The access type ACT is then declared and since it is class wide this implies that the type T must be tagged (the reader will recall from the discussion above that this odd feature is banished to Annex J in Ada 2005). The full type T is then declared in the body. We also declare a primitive operation Op of the type T in the private part.

However, before the body of P is declared, nothing in Ada 95 prevents us from writing a private child thus

```
private package P.C is
  procedure Naughty;
end P.C;

package body P.C is
  procedure Naughty is
  begin
    Op(X);                               -- a dispatching call
  end Naughty;
end P.C;
```

and the procedure Naughty can call the dispatching operation Op. The problem is that we are required to compile this call before the type T is completed and thus before the location of its tag is known.

This problem is prevented in Ada 2005 by a rule that if an incomplete type declared in a private part has primitive operations then the completion cannot be deferred to the body.

Similar problems arise with access to subprogram types. Thus, as mentioned above, Ada 95 permits

```
type T;
type A is access procedure (X: in out T);
```

In Ada 2005, the completion of T cannot be deferred to a body. Nor can we declare such an access to subprogram type if we only have an incomplete view of T arising from a limited with clause.

Another change in Ada 2005 can be illustrated by the Departments and Employees example. We can write

```
limited with Departments;
package Employees is
  type Employee is tagged private;
  procedure Assign_Employee(E: in out Employee; D: in out Departments.Department'Class);
  type Dept_Ptr is access all Departments.Department'Class;
  ...
end Employees;
```

```
with Employees;  use Employees;
procedure Recruit(D: Dept_Ptr; E: in out Employee) is
begin
  Assign_Employee(E, D.all);
end Recruit;
```

Ada 95 has a rule that says "thou shalt not dereference an incomplete type". This would prevent the call of Assign_Employee which is clearly harmless. It would be odd to require Recruit to have a nonlimited with clause for Departments to allow the call of Assign_Employee. Accordingly the rule is changed in Ada 2005 so that dereferencing an incomplete view is only forbidden when used as a prefix as, for example, in D'Size.

3 Visibility from private parts

Ada 95 introduced public and private child packages in order to enable subsystems to be decomposed in a structured manner. The general idea is that

- public children enable the decomposition of the view of a subsystem to the user of the subsystem,
- private children enable the decomposition of the implementation of a subsystem.

In turn both public and private children can themselves have children of both kinds. This has proved to work well in most cases but a difficulty has arisen regarding private parts.

Recall that the private part of a package really concerns the implementation of the package rather than specifying the facilities to the external user. Although it does not concern algorithmic aspects of the implementation it does concern the implementation of data abstraction. During the original design of Ada some thought was given to the idea that a package should truly be written and compiled as three distinct parts. Perhaps like this

```
with ...
package P is
```

```
   ...                          -- visible specification
end;

with ...
package private P is            -- just dreaming
   ...                          -- private part
end;

with ...
package body P is
   ...                          -- body
end;
```

Each part could even have had its own context clause as shown.

However, it was clear that this would be an administrative nightmare in many situations and so the two-part specification and body emerged with the private part lurking at the end of the visible part of the specification (and sharing its context clause).

This was undoubtedly the right decision in general. The division into just two parts supports separate compilation well and although the private part is not part of the logical interface to the user it does provide information about the physical interface and that is needed by the compiler.

The problem that has emerged is that the private part of a public package cannot access the information in private child packages. Private children are of course not visible to the user but there is no reason why they should not be visible to the private part of a public package provided that somehow the information does not leak out. Thus consider a hierarchy

```
package App is
   ...
private
   ...
end App;

package App.Pub is
   ...
private
   ...
end App.Pub;

private package App.Priv is
   ...
private
   ...
end App.Priv;
```

There is no reason why the private parts of App and App.Pub and the visible part of the specification of App.Priv should not share visibility (the private part of App.Priv logically belongs to the next layer of secrecy downwards). But this sharing is not possible in Ada 95.

The public package App.Pub is not permitted to have a with clause for the child package App.Priv since this would mean that the visible part of App.Pub would also have visibility of this information and by mechanisms such as renaming could pass it on to the external user.

The specification of the parent package App is also not permitted to have a with clause for App.Priv since this would break the dependence rules anyway. Any child has a dependence on its parent and so the parent specification has to be compiled or entered into the program library first.

Note that the private part of the public child App.Pub does automatically have visibility of the private part of the parent App. But the reverse cannot be true again because of the dependence rules.

Finally note that the private child App.Priv can have a with clause for its public sibling App.Pub (it creates a dependence of course) but that only gives the private child visibility of the visible part of the public child.

So the only visibility sharing among the three regions in Ada 95 is that the private part of the public child and the visible part of the private child can see the private part of the parent.

The practical consequence of this is that in large systems, information which should really be lower down the hierarchy has to be placed in the private part of the ultimate parent. This tends to mean that the parent package becomes very large thereby making maintenance more difficult and forcing frequent recompilations of the parent and thus the whole hierarchy of packages.

The situation is much alleviated in Ada 2005 by the introduction of private with clauses.

If a package P has a private with clause for a package Q thus

```
private with Q;
package P is ...
```

then the private part of P has visibility of the visible part of the package Q, whereas the visible part of P does not have visibility of Q and so visibility cannot be transmitted to a user of P. It is rather as if the with clause were attached to just the private part of P thus

```
package P is
  ...
with Q;                          -- we cannot write this
private
  ...
end P;
```

This echoes the three-part decomposition of a package discussed above.

A private with clause can be placed wherever a normal with clause for the units mentioned can be placed and in addition a private with clause which mentions a private unit can be placed on any of its parent's descendants.

So we can put a private with clause for App.Priv on App.Pub thereby permitting visibility of the private child from the private part of its public sibling. Thus

```
private with App.Priv;
package App.Pub is
  ...                            -- App.Priv not visible here
private
  ...                            -- App.Priv visible here
end App.Pub;
```

This works provided we don't run afoul of the dependence rules. The private with clause means that the public child has a dependence on the private child and therefore the private child must be compiled or entered into the program library first.

We might get a situation where there exists a mutual dependence between the public and private sibling in that each has a type that the other wants to access. In such a case we can use a limited private with clause thus

```
limited private with App.Priv;
package App.Pub is
```

```
   ...                              -- App.Priv not visible here
private
   ...                              -- limited view of App.Priv here
end App.Pub;
```

The child packages are both dependent on the parent package and so the parent cannot have with clauses for them. But a parent can have a limited with clause for a public child and a limited private with clause for a private child thus

```
limited with App.Pub;  limited private with App.Priv;
package App is
   ...                              -- limited view of App.Pub here
private
   ...                              -- limited view of App.Priv here
end App;
```

A simple example of the use of private with clauses was given in the Introduction. Here it is somewhat extended

```
limited with App.User_View;  limited private with App.Secret_Details;
package App is
   ...                              -- limited view of type Outer visible here
private
   ...                              -- limited view of type Inner visible here
end App;

private package App.Secret_Details is
   type Inner is ...
   ...                              -- various operations on Inner etc
end App.Secret_Details;

private with App.Secret_Details;
package App.User_View is

   type Outer is private;
   ...                              -- various operations on Outer visible to the user

                                    -- type Inner is not visible here
private
                                    -- type Inner is visible here

    type Outer is
      record
        X: Secret_Details.Inner;
        ...
      end record;
   ...
end App.User_View;
```

In the previous section we observed that there were problems with interactions between use clauses, nonlimited with clauses, and limited with clauses. Those rules also apply to private with clauses where a private with clause is treated as a nonlimited with clause and a limited private with clause is treated as a limited with clause. In other words private is ignored for the purpose of those rules.

Moreover, we cannot place a package use clause in the same context clause as a private with clause (limited or not). This is because we would then expect it to apply to the visible part as well which would be wrong. However, we can always put a use clause in the private part thus

```
private with Q;
package P is
  ...                          -- Q not visible here
private
  use Q;
  ...                          -- use visibility of Q here
end P;
```

At the risk of confusing the reader it might be worth pointing out that strictly speaking the rules regarding private with are treated as legality rules rather than visibility rules. Here is an example which illustrates this subtlety and the dangers it avoids

```
package P is
  function F return Integer;
end P;

function F return Integer;

with P;
private with F;
package Q is
  use P;
  X: Integer := F;             -- illegal
  Y: Integer := P.F;           -- legal
private
  Z: Integer := F;             -- legal, calls the library F
end Q;
```

If we treated the rules regarding private with as pure visibility rules then the call of F in the declaration of X in the visible part would be a call of P.F. So moving the declaration of X to the private part would silently change the F being called – this would be nasty. We can always write the call of F as P.F as shown in the declaration of Y.

So the rules regarding private with are written to make entities visible but unmentionable in the visible part. In practice programmers can just treat them as visibility rules so that the entities are not visible at all which is how we have described them above.

A useful consequence of the unmentionable rather than invisible approach is that we can use the name of a package mentioned in a private with clause in a pragma in the context clause thus

```
private with P;  pragma Elaborate(P);
package Q is ...
```

Private with clauses are in fact allowed on bodies as well, in which case they just behave as a normal with clause. Another minor point is that Ada has always permitted several with clauses for the same unit in one context clause thus

```
with P;  with P;  with P, P;
package Q is ...
```

To avoid complexity we similarly allow

```
with P;  private with P;
package Q is
```

and then the private with is ignored.

We have introduced private with clauses in this section as the solution to the problem of access to private children from the private part of the parent or public sibling. But they have other important uses. If we have

```
private with P;
package Q is ...
```

then we are assured that the package Q cannot inadvertently access P in the visible part and, in particular, pass on access to entities in P by renamings and so on. Thus writing **private with** provides additional documentation information which can be useful to both human reviewers and program analysis tools. So if we have a situation where a private with clause is all that is needed then we should use it rather than a normal with clause.

In summary, whereas in Ada 95 there is just one form of with clause, Ada 2005 provides four forms

```
with P;                    -- full view
limited with P;            -- limited view
private with P;            -- full view from private part
limited private with P;    -- limited view from private part
```

Finally, note that if a private with clause is given on a specification then it applies to the body as well as to the private part.

4 Aggregates

There are important changes to aggregates in Ada 2005 which are very useful in a number of contexts. These were triggered by the changes to the rules for limited types which are described in the next section, but it is convenient to first consider aggregates separately.

The main change is that the box notation <> is now permitted as the value in a named aggregate. The meaning is that the component of the aggregate takes the default value if there is one.

So if we have a record type such as

```
type RT is
  record
    A: Integer := 7;
    B: access Integer;
    C: Float;
  end record;
```

then if we write

```
X: RT := (A => <>, B => <>, C => <>);
```

then X.A has the value 7, X.B has the value **null** and X.C is undefined. So the default value is that given in the record type declaration or, in the absence of such an explicit default value, it is the default value for the type. If there is no explicit default value and the type does not have one either then the value is simply undefined as usual.

The above example could be abbreviated to

```
X: RT := (others => <>);
```

The obvious combinations are allowed

```
(A => <>, B => An_Integer'Access, C => 2.5)
(A => 3, others => <>)
(A => 3, B | C => <>)
```

The last two are the same. There is a rule in Ada 95 that if several record components in an aggregate are given the same expression using a | then they have to be of the same type. This does not apply in the case of <> because no typed expression is involved.

The <> notation is not permitted with positional notation. So we cannot write

```
(3, <>, 2.5)                    -- illegal
```

But we can mix named and positional notations in a record aggregate as usual provided the named components follow the positional ones, so the following are permitted

```
(3, B => <>, C => 2.5)
(3, others => <>)
```

A minor but important rule is that we cannot use <> for a component of an aggregate that is a discriminant if it does not have a default. Otherwise we could end up with an undefined discriminant.

The <> notation is also allowed with array aggregates. But in this case the situation is much simpler because it is not possible to give a default value for array components. Thus we might have

```
P: array (1.. 1000) of Integer := (1 => 2, others => <>);
```

The array P has its first component set to 2 and the rest undefined. (Maybe P is going to be used to hold the first 1000 prime numbers and we have a simple algorithm to generate them which requires the first prime to be provided.) The aggregate could also be written as

```
(2, others => <>)
```

Remember that **others** is permitted with a positional array aggregate provided it is at the end. But otherwise <> is not allowed with a positional array aggregate.

We can add **others** => <> even when there are no components left. This applies to both arrays and records.

The box notation is also useful with tasks and protected objects used as components. Consider

```
protected type Semaphore is ... ;

type PT is
  record
    Guard: Semaphore;
    Count: Integer;
    Finished: Boolean := False;
  end record;
```

As explained in the next section, we can now use an aggregate to initialize an object of a limited type. Although we cannot give an explicit initial value for a Semaphore we would still like to use an aggregate to get a coverage check. So we can write

```
X: PT := (Guard => <>, Count => 0, Finished => <>);
```

Note that although we can use <> to stand for the value of a component of a protected type in a record we cannot use it for a protected object standing alone.

```
Sema: Semaphore := <>;          -- illegal
```

The reason is that there is no need since we have no coverage check to concern us and there could be no other reason for doing it anyway.

Similarly we can use <> with a component of a private type as in

```
type Secret is private;

type Visible is
  record
    A: Integer;
    S: Secret;
  end record;

X: Visible := (A => 77; S => <>);
```

but not when standing alone

```
S: Secret := <>;                    -- illegal
```

It would not have any purpose because such a variable will take any default value anyway.

We conclude by mentioning a small point for the language lawyer. Consider

```
function F return Integer;

type T is
  record
    A: Integer := F;
    B: Integer := 3;
  end record;
```

Writing

```
X: T := (A => 5, others => <>);     -- does not call F
```

is not quite the same as

```
X: T;                               -- calls F
...
X.A := 5;  X.B := 3;
```

In the first case the function F is not called whereas in the second case it is called when X is declared in order to default initialize X.A. If it had a nasty side effect then this could matter. But then programmers should not use nasty side effects anyway.

5 Limited types and return statements

The general idea of a limited type is to restrict the operations that a user can do on the type to just those provided by the author of the type and in particular to prevent the user from doing assignment and thus making copies of objects of the type.

However, limited types have always been a problem. In Ada 83 the concept of limitedness was confused with that of private types. Thus in Ada 83 we only had limited private types (although task types were inherently limited).

Ada 95 brought significant improvement by two changes. It allowed limitedness to be separated from privateness. It also allowed the redefinition of equality for all types whereas Ada 83 forbade this for limited types. In Ada 95, the key property of a limited type is that assignment is not predefined and cannot be defined (equality is not predefined either but it can be defined). The general idea of course is that there are some types for which it would be wrong for the user to be

able to make copies of objects. This particularly applies to types involved in resource control and types implemented using access types.

However, although Ada 95 greatly improved the situation regarding limited types, nevertheless two major difficulties have remained. One concerns the initialization of objects and the other concerns the results of functions.

The first problem is that Ada 95 treats initialization as a process of assigning the initial value to the object concerned (hence the use of := unlike some Algol based languages which use = for initialization and := for assignment). And since initialization is treated as assignment it is forbidden for limited types. This means that we cannot initialize objects of a limited type nor can we declare constants of a limited type. We cannot declare constants because they have to be initialized and yet initialization is forbidden. This is more annoying in Ada 95 since we can make a type limited but not private.

The following example was discussed in the Introduction

```
type T is limited
  record
    A: Integer;
    B: Boolean;
    C: Float;
  end record;
```

Note that this type is explicitly limited (but not private) but its components are not limited. If we declare an object of type T in Ada 95 then we have to initialize the components (by assigning to them) individually thus

```
   X: T;
begin
   X.A := 10;  X.B := True;  X.C := 45.7;
```

Not only is this annoying but it is prone to errors as well. If we add a further component D to the type T then we might forget to initialize it. One of the advantages of aggregates is that we have to supply all the components which automatically provides full coverage analysis.

This problem did not arise in Ada 83 because we could not make a type limited without making it also private and so the individual components were not visible anyway.

Ada 2005 overcomes the difficulty by stating that initialization by an aggregate is not actually assignment even though depicted by the same symbol. This permits

```
X: T := (A => 10,  B => True,  C => 45.7);
```

We should think of the individual components as being initialized individually *in situ* – an actual aggregated value is not created and then assigned.

The reader might recall that the same thing happens when an aggregate is used to initialize a controlled type; this was not as Ada 95 was originally defined but it was corrected in AI-83 and consolidated in the 2001 Corrigendum [2].

We can now declare a constant of a limited type as expected

```
X: constant T := (A => 10,  B => True,  C => 45.7);
```

Limited aggregates can be used in a number of other contexts as well

- as the default expression in a component declaration,

so if we nest the type T inside some other type (which itself then is always limited – it could be explicitly limited but there is a general rule that a type is implicitly limited if it has a limited component) we might have

```
type Twrapper is
  record
    Tcomp: T := (0, False, 0.0);
  end record;
```

- as an expression in a record aggregate,

  so again using the type Twrapper as in

  ```
  XT: Twrapper := (Tcomp => (1, True, 1.0));
  ```

- as an expression in an array aggregate similarly,

  so we might have

  ```
  type Tarr is array (1 .. 5) of T;

  Xarr: Tarr := (1 .. 5 => (2, True, 2.0));
  ```

- as the expression for the ancestor part of an extension aggregate,

  so if TT were tagged as in

  ```
  type TT is tagged limited
    record
      A: Integer;
      B: Boolean;
      C: Float;
    end record;

  type TTplus is new TT with
    record
      D: Integer;
    end record;
  ...
  XTT: TTplus := ((1, True, 1.0) with 2);
  ```

- as the expression in an initialized allocator,

  so we might have

  ```
  type T_Ptr is access T;
  XT_Ptr: T_Ptr;
  ...
  XT_Ptr := new T'(3, False, 3.0);
  ```

- as the actual parameter for a subprogram parameter of a limited type of mode in

  ```
  procedure P(X: in T);
  ...
  P((4, True, 4.0));
  ```

- similarly as the default expression for a parameter

  ```
  procedure P(X: in T := (4, True, 4.0));
  ```

- as the result in a return statement

```
function F( ... ) return T is
begin
  ...
  return (5, False, 5.0);
end F;
```

this really concerns the other major change to limited types which we shall return to in a moment.

- as the actual parameter for a generic formal limited object parameter of mode in,

```
generic
  FT: in T;
package P is ...
...
package Q is new P(FT => (7, True, 7.0));
```

The last example is interesting. Limited generic parameters were not allowed in Ada 95 at all because there was no way of passing an actual parameter because the generic parameter mechanism for an in parameter is considered to be assignment. But now the actual parameter can be passed as an aggregate. An aggregate can also be used as a default value for the parameter thus

```
generic
  FT: in T := (0, False, 0.0);
package P is ...
```

Remember that there is a difference between subprogram and generic parameters. Subprogram parameters were always allowed to be of limited types since they are mostly implemented by reference and no copying happens anyway. The only exception to this is with limited private types where the full type is an elementary type.

The change in Ada 2005 is that an aggregate can be used as the actual parameter in the case of a subprogram parameter of mode **in** whereas that was not possible in Ada 95.

Sometimes a limited type has components where an initial value cannot be given as in

```
protected type Semaphore is ... ;

type PT is
  record
    Guard: Semaphore;
    Count: Integer;
    Finished: Boolean := False;
  end record;
```

Since a protected type is inherently limited the type PT is also limited because a type with a limited component is itself limited. Although we cannot give an explicit initial value for a Semaphore, we would still like to use an aggregate to get the coverage check. In such cases we can use the box symbol <> as described in the previous section to mean use the default value for the type (if any). So we can write

```
X: PT := (Guard => <>, Count => 0, Finished => <>);
```

The major rule that must always be obeyed is that values of limited types can never be copied. Consider nested limited types

```
type Inner is limited
  record
    L: Integer;
```

```
    M: Float;
  end record;

type Outer is limited
  record
    X: Inner;
    Y: Integer;
end record;
```

If we declare an object of type Inner

```
An_Inner: Inner := (L => 2, M => 2.0);
```

then we could not use An_Inner in an aggregate of type Outer

```
An_Outer: Outer := (X => An_Inner, Y => 3);          -- illegal
```

This is illegal because we would be copying the value. But we can use a nested aggregate as mentioned earlier

```
An_Outer: Outer := (X => (2, 2.0), Y => 3);
```

The other major change to limited types concerns returning values from functions.

We have seen that the ability to initialize an object of a limited type with an aggregate solves the problem of giving an initial value to a limited type provided that the type is not private.

Ada 2005 introduces a new approach to returning the results from functions which can be used to solve this and other problems.

We will first consider the case of a type that is limited such as

```
type T is limited
  record
    A: Integer;
    B: Boolean;
    C: Float;
  end record;
```

We can declare a function that returns a value of type T provided that the return does not involve any copying. For example we could have

```
function Init(X: Integer; Y: Boolean; Z: Float) return T is
begin
  return (X, Y, Z);
end Init;
```

This function builds the aggregate in place in the return expression and delivers it to the location specified where the function is called. Such a function can be called from precisely those places listed above where an aggregate can be used to build a limited value in place. For example

```
V: T := Init(2, True, 3.0);
```

So the function itself builds the value in the variable V when constructing the returned value. Hence the address of V is passed to the function as a sort of hidden parameter.

Of course if T is not private then this achieves no more than simply writing

```
V: T := (2, True, 3.0);
```

But the function Init can be used even if the type is private. It is in effect a constructor function for the type. Moreover, the function Init could be used to do some general calculation with the parameters before delivering the final value and this brings considerable flexibility.

We noted that such a function can be called in all the places where an aggregate can be used and this includes in a return expression of a similar function or even itself

```
function Init_True(X: Integer; Z: Float) return T is
begin
  return Init(X, True, Z);
end Init_True;
```

It could also be used within an aggregate. Suppose we have a function to return a value of the limited type Inner thus

```
function Make_Inner(X: Integer; Y: Float) return Inner is
begin
  return (X, Y);
end Make_Inner;
```

then not only could we use it to initialize an object of type Inner but we could use it in a declaration of an object of type Outer thus

```
An_Inner: Inner := Make_Inner(2, 2.0);
An_Outer: Outer := (X => Make_Inner(2, 2.0), Y => 3);
```

In the latter case the address of the component of An_Outer is passed as the hidden parameter to the function Make_Inner.

Being able to use a function in this way provides much flexibility but sometimes even more flexibility is required. New syntax permits the final returned object to be declared and then manipulated in a general way before finally returning from the function.

The basic structure is

```
function Make( ... ) return T is
begin
  ...
  return R: T do                    -- declare R to be returned
    ...                             -- here we can manipulate R in the usual way
    ...                             -- in a sequence of statements
  end return;
end Make;
```

The general idea is that the object R is declared and can then be manipulated in an arbitrary way before being finally returned. Note the use of the reserved word **do** to introduce the statements in much the same way as in an accept statement. The sequence ends with **end return** and at this point the function passes control back to where it was called. Note that if the function had been called in a construction such as the initialization of an object X of a limited type T thus

```
X: T := Make( ... );
```

then the variable R inside the function is actually the variable X being initialized. In other words the address of X is passed as a hidden parameter to the function Make in order to create the space for R. No copying is therefore ever performed.

The sequence of statements could have an exception handler

```
return R: T do
   ...                          -- statements
exception
   ...                          -- handlers
end return;
```

If we need local variables within an extended return statement then we can declare an inner block in the usual way

```
return R: T do
   declare
      ...                       -- local declarations
   begin
      ...                       -- statements
   end;
end return;
```

The declaration of R could have an initial value

```
return R: T := Init( ... ) do
   ...
end return;
```

Also, much as in an accept statement, the **do** ... **end return** part can be omitted, so we simply get

```
return R: T;
```

or

```
return R: T := Init( ... );
```

which is handy if we just want to return the object with its default or explicit initial value.

Observe that extended return statements cannot be nested but could have simple return statements inside

```
return R: T := Init( ... ) do
   if ... then
      ...
      return;                   -- result is R
   end if;
   ...
end return;
```

Note that simple return statements inside an extended return statement do not have an expression since the result returned is the object R declared in the extended return statement itself.

Although extended return statements cannot be nested there could nevertheless be several in a function, perhaps in branches of an if statement or case statement. This would be quite likely in the case of a type with discriminants

```
type Person(Sex: Gender) is ... ;

function F( ... ) return Person is
begin
   if ... then
      return R: Person(Sex => Male) do
         ...
      end return;
```

```
  else
    return R: Person(Sex => Female) do
      ...
    end return;
  end if;
end F;
```

This also illustrates the important point that although we introduced these extended return statements in the context of greater flexibility for limited types they can be used with any types at all such as the nonlimited type Person. The mechanism of passing a hidden parameter which is the address for the returned object of course only applies to limited types. In the case of nonlimited types, the result is simply delivered in the usual way.

We can also rename the result of a function call – even if it is limited.

The result type of a function can be constrained or unconstrained as in the case of the type Person but the actual object delivered must be of a definite subtype. For example suppose we have

```
type UA is array (Integer range <>) of Float;
subtype CA is UA(1 .. 10);
```

Then the type UA is unconstrained but the subtype CA is constrained. We can use both with extended return statements.

In the constrained case the subtype in the extended return statement has to statically match (typically it will be the same textually but need not) thus

```
function Make( ... ) return CA is
begin
  ...
  return R: UA(1 .. 10) do          -- statically matches
    ...
  end return;
end Make;
```

In the unconstrained case the result R has to be constrained either by its subtype or by its initial value. Thus

```
function Make( ... ) return UA is
begin
  ...
  return R: UA(1 .. N) do
    ...
  end return;
end Make;
```

or

```
function Make( ... ) return UA is
begin
  ...
  return R: UA := (1 .. N => 0.0) do
    ...
  end return;
end Make;
```

The other important change to the result of functions which was discussed in the previous paper is that the result type can be of an anonymous access type. So we can write a function such as

```
function Mate_Of(A: access Animal'Class) return access Animal'Class;
```

The introduction of explicit access types for the result means that Ada 2005 is able to dispense with the notion of returning by reference.

This does, however, introduce a noticeable incompatibility between Ada 95 and Ada 2005. We might for example have a pool of slave tasks acting as servers. Individual slave tasks might be busy or idle. We might have a manager task which allocates slave tasks to different jobs. The manager might declare the tasks as an array

```
Slaves: array (1 .. 10) of TT;          -- TT is some task type
```

and then have another array of properties of the tasks such as

```
type Task_Data is
  record
    Active: Boolean := False;
    Job_Code: ... ;
  end record;

Slave_Data: array (1 .. 10) of Task_Data;
```

We now need a function to find an available slave. In Ada 95 we write

```
function Get_Slave return TT is
begin
  ...                                   -- find index K of first idle slave
  return Slaves(K);                     -- in Ada 95, not in Ada 2005
end Get_Slave;
```

This is not permitted in Ada 2005. If the result type is limited (as in this case) then the expression in the return statement has to be an aggregate or function call and not an object such as Slaves(K).

In Ada 2005 the function has to be rewritten to honestly return an access value referring to the task type rather than invoking the mysterious concept of returning by reference.

So we have to write

```
function Get_Slave return access TT is
begin
  ...                                   -- find index K of first idle slave
  return Slaves(K)'Access;              -- in Ada 2005
end Get_Slave;
```

and all the calls of Get_Slave have to be changed to correspond as well.

This is perhaps the most serious incompatibility between Ada 95 and Ada 2005. But then, at the end of the day, honesty is the best policy.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] ISO/IEC 8652:1995/COR 1:2001, *Ada Reference Manual – Technical Corrigendum 1*.

[3] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

Rationale for Ada 2005: 4 Tasking and Real-Time

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements in the tasking and real-time areas for Ada 2005.

There are only a few changes to the core tasking model itself. One major extension, however, is the ability to combine the interface feature described in an earlier paper with the tasking model; this draws together the object-oriented and tasking models of Ada which previously were disjoint aspects of the language.

There are also many additional predefined packages in the Real-Time Systems annex concerning matters such as scheduling and timing; these form the major topic of this paper.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 3, September 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] identifies real-time systems as an important area. It says

"The main purpose of the Amendment is to address identified problems in Ada that are interfering with Ada's usage or adoption, especially in its major application areas (such as high-reliability, long-lived real-time and/or embedded applications and very large complex systems). The resulting changes may range from relatively minor, to more substantial."

It then identifies the inclusion of the Ravenscar profile [2] (for predictable real-time) as a worthwhile addition and then asks the ARG to pay particular attention to

> Improvements that will maintain or improve Ada's advantages, especially in those user domains where safety and criticality are prime concerns. Within this area it cites as high priority, improvements in the real-time features and improvements in the high integrity features.

Ada 2005 does indeed make many improvements in the real-time area and includes the Ravenscar profile as specifically mentioned. The following Ada issues cover the relevant changes and are described in detail in this paper:

249 Ravenscar profile for high-integrity systems

265 Partition elaboration policy for high-integrity systems

266 Task termination procedure

297 Timing events

298 Non-preemptive dispatching

305 New pragma and restrictions for real-time systems

307 Execution-time clocks

321 Definition of dispatching policies

327 Dynamic ceiling priorities

345 Protected and task interfaces

347 Title of Annex H

354 Group execution-time budgets

355 Priority dispatching including Round Robin

357 Earliest Deadline First scheduling

386 Further functions returning time-span values

394 Redundant Restrictions identifiers and Ravenscar

397 Conformance and overriding for procedures and entries

399 Single tasks and protected objects with interfaces

421 Sequential activation and attachment

These changes can be grouped as follows.

First there is the introduction of a mechanism for monitoring task termination (266).

A major innovation in the core language is the introduction of synchronized interfaces which provide a high degree of unification between the object-oriented and real-time aspects of Ada (345, 397, 399).

There is of course the introduction of the Ravenscar profile (249) plus associated restrictions (305, 394) in the Real-Time Systems annex (D).

There are major improvement to the scheduling and task dispatching mechanisms with the addition of further standard policies (298, 321, 327, 355, 357). These are also in Annex D.

A number of timing mechanisms are now provided. These concern stand-alone timers, timers for monitoring the CPU time of a single task, and timers for controlling the budgeting of time for groups of tasks (297, 307, 354, 386). Again these are in Annex D.

Finally, more control is provided over partition elaboration which is very relevant to real-time high-integrity systems (265, 421). This is in Annex H which is now entitled High Integrity Systems (347).

Note that further operations for the manipulation of time in child packages of Calendar (351) will be discussed with the predefined library in a later paper.

2 Task termination

In the Introduction we mentioned the problem of how tasks can have a silent death in Ada 95. This happens if a task raises an exception which is not handled by the task itself. Tasks may also terminate because of going abnormal as well as terminating normally. The detection of task termination and its causes can be monitored in Ada 2005 by the package Ada.Task_Termination whose specification is essentially

```
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Exceptions; use Ada.Exceptions;
package Ada.Task_Termination is
  pragma Preelaborable(Task_Termination);

  type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);
```

```
   type Termination_Handler is access protected
      procedure(Cause: in Cause_Of_Termination;
                  T: in Task_Id; X: in Exception_Occurrence);

   procedure Set_Dependents_Fallback_Handler (Handler: in Termination_Handler);
   function Current_Task_Fallback_Handler return Termination_Handler;

   procedure Set_Specific_Handler(T: in Task_Id; Handler: in Termination_Handler);
   function Specific_Handler(T: in Task_Id) return Termination_Handler;

end Ada.Task_Termination;
```

(Note that the above includes use clauses in order to simplify the presentation; the actual package does not have use clauses. We will use a similar approach for the other predefined packages described in this paper.)

The general idea is that we can associate a protected procedure with a task. The protected procedure is then invoked when the task terminates with an indication of the reason passed via its parameters. The protected procedure is identified by using the type Termination_Handler which is an access type referring to a protected procedure.

The association can be done in two ways. Thus (as in the Introduction) we might declare a protected object Grim_Reaper

```
protected Grim_Reaper is
  procedure Last_Gasp(C: Cause_Of_Termination; T: Task_Id; X: Exception_Occurrence);
end Grim_Reaper;
```

which contains the protected procedure Last_Gasp. Note that the parameters of Last_Gasp match those of the access type Termination_Handler.

We can then nominate Last_Gasp as the protected procedure to be called when the specific task T dies by

```
Set_Specific_Handler(T'Identity, Grim_Reaper.Last_Gasp'Access);
```

Alternatively we can nominate Last_Gasp as the protected procedure to be called when any of the tasks dependent on the current task becomes terminated by writing

```
Set_Dependents_Fallback_Handler(Grim_Reaper.Last_Gasp'Access);
```

Note that a task is not dependent upon itself and so this does not set a handler for the current task.

Thus a task can have two handlers. A fallback handler and a specific handler and either or both of these can be null. When a task terminates (that is after any finalization but just before it vanishes), the specific handler is invoked if it is not null. If the specific handler is null, then the fallback handler is invoked unless it too is null. If both are null then no handler is invoked.

The body of protected procedure Last_Gasp might then output various diagnostic messages

```
procedure Last_Gasp(C: Cause_Of_Termination; T: Task_Id; X: Exception_Occurrence) is
begin
  case C is
    when Normal => null;
    when Abnormal =>
      Put("Something nasty happened to task ");
      Put_Line(Image(T));
    when Unhandled_Exception =>
      Put("Unhandled exception occurred in task ");
      Put_Line(Image(T));
```

```
      Put(Exception_Information(X));
  end case;
end Last_Gasp;
```

There are three possible reasons for termination, it could be normal, abnormal (caused by abort), or because of propagation of an unhandled exception. In the last case the parameter X gives details of the exception occurrence whereas in the other cases X has the value Null_Occurrence.

Initially both specific and fallback handlers are null for all tasks. However, note that if a fallback handler has been set for all dependent tasks of T then the handler will also apply to any task subsequently created by T or one of its descendants. Thus a task can be born with a fallback handler already in place.

If a new handler is set then it replaces any existing handler of the appropriate kind. Calling either setting procedure with null for the handler naturally sets the appropriate handler to null.

The current handlers can be found by calling the functions Current_Task_Fallback_Handler or Specific_Handler; they return null if the handler is null.

It is important to realise that the fallback handlers for the tasks dependent on T need not all be the same since one of the dependent tasks of T might set a different handler for its own dependent tasks. Thus the fallback handlers for a tree of tasks can be different in various subtrees. This structure is reflected by the fact that the determination of the current fallback handler of a task is in fact done by searching recursively the tasks on which it depends.

Note that we cannot directly interrogate the fallback handler of a specific task but only that of the current task. Moreover, if a task sets a fallback handler for its dependents and then enquires of its own fallback handler it will not in general get the same answer because it is not one of its own dependents.

It is important to understand the situation regarding the environment task. This unnamed task is the task that elaborates the library units and then calls the main subprogram. Remember that library tasks (that is tasks declared at library level) are activated by the environment task before it calls the main subprogram.

Suppose the main subprogram calls the setting procedures as follows

```
procedure Main is

  protected RIP is
    protected procedure One( ... );
    protected procedure Two( ... );
  end;
  ...
begin
  Set_Dependents_Fallback_Handler(RIP.One'Access);
  Set_Specific_Handler(Current_Task, RIP.Two'Access);
  ...
end Main;
```

The specific handler for the environment task is then set to Two (because Current_Task is the environment task at this point) but the fallback handler for the environment task is null. On the other hand the fallback handler for all other tasks in the program including any library tasks is set to One. Note that it is not possible to set the fallback handler for the environment task.

The astute reader will note that there is actually a race condition here since a library task might have terminated before the handler gets set. We could overcome this by setting the handler as part of the elaboration code thus

```
package Start_Up is
  pragma Elaborate_Body;
end;

with Ada.Task_Termination; use Ada.Task_Termination;
package body Start_Up is
begin
  Set_Dependents_Fallback_Handler(RIP.One'Access);
end Start_Up;

with Start_Up;
pragma Elaborate(Start_Up);
package Library_Tasks is
  ...                                   -- declare library tasks here
end;
```

Note how the use of pragmas Elaborate_Body and Elaborate ensures that things get done in the correct order.

Some minor points are that if we try to set the specific handler for a task that has already terminated then Tasking_Error is raised. And if we try to set the specific handler for the null task, that is call Set_Specific_Handler with parameter T equal to Null_Task_Id, then Program_Error is raised. These exceptions are also raised by calls of the function Specific_Handler in similar circumstances.

3 Synchronized interfaces

We now turn to the most important improvement to the core tasking features introduced by Ada 2005. This concerns the coupling of object oriented and real-time features through inheritance.

Recall from the paper on the object oriented model that we can declare an interface thus

```
type Int is interface;
```

An interface is essentially an abstract tagged type that cannot have any components but can have abstract operations and null procedures. We can then derive other interfaces and tagged types by inheritance such as

```
type Another_Int is interface and Int1 and Int2;

type T is new Int1 and Int2;

type TT is new T and Int3 and Int4;
```

Remember that a tagged type can be derived from at most one other normal tagged type but can also be derived from several interfaces. In the list, the first is called the parent (it can be a normal tagged type or an interface) and any others (which can only be interfaces) are called progenitors.

Ada 2005 also introduces further categories of interfaces, namely synchronized, protected, and task interfaces. A synchronized interface can be implemented by either a task or protected type; a protected interface can only be implemented by a protected type and a task interface can only be implemented by a task type.

A nonlimited interface can only be implemented by a nonlimited type. However, an explicitly marked limited interface can be implemented by any tagged type (limited or not) or by a protected or task type. Remember that task and protected types are inherently limited. Note that we use the term limited interface to refer collectively to interfaces marked limited, synchronized, task or protected and we use explicitly limited to refer to those actually marked as limited.

So we can write

```
type LI is limited interface;                    -- similarly type LI2

type SI is synchronized interface;

type TI is task interface;

type PI is protected interface;
```

and we can of course provide operations which must be abstract or null. (Remember that **synchronized** is a new reserved word.)

We can compose these interfaces provided that no conflict arises. The following are all permitted:

```
type TI2 is task interface and LI and TI;

type LI3 is limited interface and LI and LI2;

type TI3 is task interface and LI and LI2;

type SI2 is synchronized interface and LI and SI;
```

The rule is simply that we can compose two or more interfaces provided that we do not mix task and protected interfaces and the resulting interface must be not earlier in the hierarchy: limited, synchronized, task/protected than any of the ancestor interfaces.

We can derive a real task type or protected type from one or more of the appropriate interfaces

```
task type TT is new TI with
  ...                          -- and here we give entries as usual
end TT;
```

or

```
protected type PT is new LI and SI with
  ...
end PT;
```

Unlike tagged record types we cannot derive a task or protected type from another task or protected type as well. So the derivation hierarchy can only be one level deep once we declare an actual task or protected type.

The operations of these various interfaces are declared in the usual way and an interface composed of several interfaces has the operations of all of them with the same rules regarding duplication and overriding of an abstract operation by a null one and so on as for normal tagged types.

When we declare an actual task or protected type then we must implement all of the operations of the interfaces concerned. This can be done in two ways, either by declaring an entry or protected operation in the specification of the task or protected object or by declaring a distinct subprogram in the same list of declarations (but not both). Of course, if an operation is null then it can be inherited or overridden as usual.

Thus the interface

```
package Pkg is
  type TI is task interface;
  procedure P(X: in TI) is abstract;
  procedure Q(X: in TI; I: in Integer) is null;
end Pkg;
```

could be implemented by

```
package PT1 is
  task type TT1 is new TI with
```

```
      entry P;                          -- P and Q implemented by entries
      entry Q(I: in Integer);
    end TT1;
  end PT1;
```

or by

```
  package PT2 is
    task type TT2 is new TI with
      entry P;                          -- P implemented by an entry
    end TT2;
                                        -- Q implemented by a procedure
    procedure Q(X: in TT2; I: in Integer);
  end PT2;
```

or even by

```
  package PT3 is
    task type TT3 is new TI with end;
                                        -- P implemented by a procedure
                                        -- Q inherited as a null procedure
    procedure P(X: in TT3);
  end PT3;
```

In this last case there are no entries and so we have the juxtaposition **with end** which is somewhat similar to the juxtaposition **is end** that occurs with generic packages used as signatures.

Observe how the first parameter which denotes the task is omitted if it is implemented by an entry. This echoes the new prefixed notation for calling operations of tagged types in general. Remember that rather than writing

```
Op(X, Y, Z, ...);
```

we can write

```
X.Op(Y, Z, ...);
```

provided certain conditions hold such as that X is of a tagged type and that Op is a primitive operation of that type.

In order for the implementation of an interface operation by an entry of a task type or a protected operation of a protected type to be possible some fairly obvious conditions must be satisfied.

In all cases the first parameter of the interface operation must be of the task type or protected type (it may be an access parameter).

In addition, in the case of a protected type, the first parameter of an operation implemented by a protected procedure or entry must have mode **out** or **in out** (and in the case of an access parameter it must be an access to variable parameter).

If the operation does not fit these rules then it has to be implemented as a subprogram. An important example is that a function has to be implemented as a function in the case of a task type because there is no such thing as a function entry. However, a function can often be directly implemented as a protected function in the case of a protected type.

Entries and protected operations which implement inherited operations may be in the visible part or private part of the task or protected type in the same way as for tagged record types.

It may seem rather odd that an operation can be implemented by a subprogram that is not part of the task or protected type itself – it seems as if it might not be task safe in some way. But a common

paradigm is where an operation as an abstraction has to be implemented by two or more entry calls. An example occurs in some implementations of the classic readers and writers problem as we shall see later.

Of course a task or protected type which implements an interface can have additional entries and operations as well just as a derived tagged type can have more operations than its parent.

The overriding indicators **overriding** and **not overriding** can be applied to entries as well as to procedures. Thus the package PT2 above could be written as

```
package PT2 is
  task type TT2 is new TI with
    overriding                    -- P implemented by an entry
    entry P;
  end TT2;

  overriding                      -- Q implemented by procedure
  procedure Q(X: in TT2; I: in Integer);
end PT2;
```

We will now explore a simple readers and writers example in order to illustrate various points. We start with the following interface

```
package RWP is
  type RW is limited interface;
  procedure Write(Obj: out RW; X: in Item) is abstract;
  procedure Read(Obj: in RW; X: out Item) is abstract;
end RWP;
```

The intention here is that the interface describes the abstraction of providing an encapsulation of a hidden location and a means of writing a value (of some type Item) to it and reading a value from it – very trivial.

We could implement this in a nonsynchronized manner thus

```
type Simple_RW is new RW with
  record
    V: Item;
  end record;

overriding
procedure Write(Obj: out Simple_RW; X: in Item);

overriding
procedure Read(Obj: in Simple_RW; X: out Item);

...

procedure Write(Obj: out Simple_RW; X: in Item) is
begin
  Obj.V := X;
end Write;

procedure Read(Obj: in Simple_RW; X: out Item) is
begin
  X := Obj.V;
end Read;
```

This implementation is of course not task safe (task safe is sometimes referred to as thread-safe). If a task calls Write and the type Item is a composite type and the writing task is interrupted part of the way through writing, then a task which calls Read might get a curious result consisting of part of the new value and part of the old value.

For illustration we could derive a synchronized interface

```
type Sync_RW is synchronized interface and RW;
```

This interface can only be implemented by a task or protected type. For a protected type we might have

```
protected type Prot_RW is new Sync_RW with
  overriding
  procedure Write(X: in Item);
  overriding
  procedure Read(X: out Item);
private
  V: Item;
end;

protected body Prot_RW is
  procedure Write(X: in Item) is
  begin
    V := X;
  end Write;

  procedure Read(X: out Item) is
  begin
    X := V;
  end Read;
end Prot_RW;
```

Again observe how the first parameter of the interface operations is omitted when they are implemented by protected operations.

This implementation is perfectly task safe. However, one of the characteristics of the readers and writers example is that it is quite safe to allow multiple readers since they cannot interfere with each other. But the type Prot_RW does not allow multiple readers because protected procedures can only be executed by one task at a time.

Now consider

```
protected type Multi_Prot_RW is new Sync_RW with
  overriding
  procedure Write(X: in Item);
  not overriding
  function Read return Item;
private
  V: Item;
end;

overriding
procedure Read(Obj: in Multi_Prot_RW; X: out Item);

...
```

```
protected body Multi_Prot_RW is
  procedure Write(X: in Item) is
  begin
    V := X;
  end Write;

  function Read return Item is
  begin
    return V;
  end Read;
end Multi_Prot_RW;

procedure Read(Obj: in Multi_Prot_RW; X: out Item) is
begin
  X := Obj.Read;
end Read;
```

In this implementation the procedure Read is implemented by a procedure outside the protected type and this procedure then calls the function Read within the protected type. This allows multiple readers because one of the characteristics of protected functions is that multiple execution is permitted (but of course calls of the protected procedure Write are locked out while any calls of the protected function are in progress). The structure is emphasized by the use of overriding indicators.

A simple tasking implementation might be as follows

```
task type Task_RW is new Sync_RW with
  overriding
  entry Write(X: in Item);
  overriding
  entry Read(X: out Item);
end;

task body Task_RW is
  V: Item;
begin
  loop
    select
      accept Write(X: in Item) do
        V := X;
      end Write;
    or
      accept Read(X: out Item) do
        X := V;
      end Read;
    or
      terminate;
    end select;
  end loop;
end Task_RW;
```

Finally, here is a tasking implementation which allows multiple readers and ensures that an initial value is set by only allowing a call of Write first. It is based on an example in that textbook [3].

```
task type Multi_Task_RW(V: access Item) is new Sync_RW with
  overriding
  entry Write(X: in Item);
```

```
   not overriding
   entry Start;
   not overriding
   entry Stop;
end;

overriding
procedure Read(Obj: in Multi_Task_RW; X: out Item);

...

task body Multi_Task_RW is
   Readers: Integer := 0;
begin
   accept Write(X: in Item) do
      V.all := X;
   end Write;
   loop
      select
         when Write'Count = 0 =>
         accept Start;
         Readers := Readers + 1;
      or
         accept Stop;
         Readers := Readers – 1;

      or
         when Readers = 0 =>
         accept Write(X: in Item) do
            V.all := X;
         end Write;
      or
         terminate;
      end select;
   end loop;
end Multi_Task_RW;

overriding
procedure Read(Obj: in Multi_Task_RW; X: out Item) is
begin
   Obj.Start;
   X := Obj.V.all;
   Obj.Stop;
end Read;
```

In this case the data being protected is accessed via the access discriminant of the task. It is structured this way so that the procedure Read can read the data directly. Note also that the procedure Read (which is the implementation of the procedure Read of the interface) calls two entries of the task.

It should be observed that this last example is by way of illustration only. As is well known, the Count attribute used in tasks (as opposed to protected objects) can be misleading if tasks are aborted or if entry calls are timed out. Moreover, it would be gruesomely slow.

So we have seen that a limited interface such as RW might be implemented by a normal tagged type (plus its various operations) and by a protected type and also by a task type. We could then dispatch

to the operations of any of these according to the tag of the type concerned. Observe that task and protected types are now other forms of tagged types and so we have to be careful to say tagged record type (or informally, normal tagged type) where appropriate.

In the above example, the types Simple_RW, Prot_RW, Multi_Prot_RW, Task_RW and Multi_Task_RW all implement the interface RW.

So we might have

```
RW_Ptr: access RW'Class := ...

...
RW_Ptr.Write(An_Item);                -- dispatches
```

and according to the value in RW_Ptr this might call the appropriate entry or procedure of an object of any of the types implementing the interface RW.

However if we have

```
Sync_RW_Ptr: access Sync_RW'Class := ...
```

then we know that any implementation of the synchronized interface Sync_RW will be task safe because it can only be implemented by a task or protected type. So the dispatching call

```
Sync_RW_Ptr.Write(An_Item);           -- task safe dispatching
```

will be task safe.

An interesting point is that because a dispatching call might be to an entry or to a procedure we now permit what appear to be procedure calls in timed entry calls if they might dispatch to an entry.

So we could have

```
select
  RW_Ptr.Read(An_Item);               -- dispatches
or
  delay Seconds(10);
end select;
```

Of course it might dispatch to the procedure Read if the type concerned turns out to be Simple_RW in which case a time out could not occur. But if it dispatched to the entry Read of the type Task_RW then it could time out.

On the other hand we are not allowed to use a timed call if it is statically known to be a procedure. So

```
A_Simple_Object: Simple_RW;
...
select
  A_Simple_Object.Read(An_Item);      -- illegal
or
  delay Seconds(10);
end select;
```

is not permitted.

A note of caution is in order. Remember that the time out is to when the call gets accepted. If it dispatches to Multi_Task_RW.Read then time out never happens because the Read itself is a procedure and gets called at once. However, behind the scenes it calls two entries and so could take a long time. But if we called the two entries directly with timed calls then we would get a time out if there were a lethargic writer in progress. So the wrapper distorts the abstraction. In a sense this is

not much worse than the problem we have anyway that a time out is to when a call is accepted and not to when it returns – it could hardly be otherwise.

The same rules apply to conditional entry calls and also to asynchronous select statements where the triggering statement can be a dispatching call.

In a similar way we also permit timed calls on entries renamed as procedures. But note that we do not allow timed calls on generic formal subprograms even though they might be implemented as entries.

Another important point to note is that we can as usual assume the common properties of the class concerned. Thus in the case of a task interface we know that it must be implemented by a task and so the operations such as **abort** and the attributes Identity, Callable and so on can be applied. If we know that an interface is synchronized then we do know that it has to be implemented by a task or a protected type and so is task safe.

Typically an interface is implemented by a task or protected type but it can also be implemented by a singleton task or protected object despite the fact that singletons have no type name. Thus we might have

```
protected An_RW is new Sync_RW with
  procedure Write(X: in Item);
  procedure Read(X: out Item);
end;
```

with the obvious body. However we could not declare a single protected object similar to the type Multi_Prot_RW above. This is because we need a type name in order to declare the overriding procedure Read outside the protected object. So singleton implementations are possible provided that the interface can be implemented directly by the task or protected object without external subprograms.

Here is another example

```
type Map is protected interface;
procedure Put(M: Map; K: Key; V: Value) is abstract;
```

can be implemented by

```
protected A_Map is new Map with
  procedure Put(K: Key; V: Value);
  ...
end A_Map;
```

There is a fairly obvious rule about private types and synchronized interfaces. Both partial and full view must be synchronized or not. Thus if we wrote

```
type SI is synchronized interface;
type T is new SI with private;
```

then the full type T has to be a task type or protected type or possibly a synchronized, protected or task interface.

We conclude this discussion on interfaces by saying a few words about the use of the word limited. (Much of this has already been explained in the paper on the object oriented model but it is worth repeating in the context of concurrent types.) We always explicitly insert limited, synchronized, task, or protected in the case of a limited interface in order to avoid confusion. So to derive a new explicitly limited interface from an existing limited interface LI we write

```
type LI2 is limited interface and LI;
```

whereas in the case of normal types we can write

```
type LT is limited ...
type LT2 is new LT and LI with ...          -- LT2 is limited
```

then LT2 is limited by the normal derivation rules. Types take their limitedness from their parent (the first one in the list, provided it is not a progenitor) and it does not have to be given explicitly on type derivation – although it can be in Ada 2005 thus

```
type LT2 is limited new LT and LI with ...
```

Remember the important rule that all descendants of a nonlimited interface have to be nonlimited because otherwise limited types could end up with an assignment operation.

This means that we cannot write

```
type NLI is interface;                      -- nonlimited
type LI is limited interface;               -- limited
task type TT is new NLI and LI with ...     -- illegal
```

This is illegal because the interface NLI in the declaration of the task type TT is not limited.

4 The Ravenscar profile

The purpose of the Ravenscar profile is to restrict the use of many tasking facilities so that the effect of the program is predictable. The profile was defined by the International Real-Time Ada Workshops which met twice at the remote village of Ravenscar on the coast of Yorkshire in North-East England. A general description of the principles and use of the profile in high integrity systems will be found in an ISO/IEC Technical Report [2] and so we shall not cover that material here.

Here is a historical interlude. It is reputed that the hotel in which the workshops were held was originally built as a retreat for King George III to keep a mistress. Another odd rumour is that he ordered all the natural trees to be removed and replaced by metallic ones whose metal leaves clattered in the wind. It also seems that Henry Bolingbroke landed at Ravenscar in July 1399 on his way to take the throne as Henry IV. Ravenscar is mentioned several times by Shakespeare in Act II of King Richard II; it is spelt Ravenspurg which is slightly confusing – maybe we need the ability to rename profile identifiers.

A profile is a mode of operation and is specified by the pragma Profile which defines the particular profile to be used. The syntax is

```
pragma Profile(profile_identifier [ , profile_argument_associations]);
```

where profile_argument_associations is simply a list of pragma argument associations separated by commas.

Thus to ensure that a program conforms to the Ravenscar profile we write

```
pragma Profile(Ravenscar);
```

The general idea is that a profile is equivalent to a set of configuration pragmas.

In the case of Ravenscar the pragma is equivalent to the joint effect of the following pragmas

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;

pragma Restrictions(
   No_Abort_Statements,
```

```
No_Dynamic_Attachment,
No_Dynamic_Priorities,
No_Implicit_Heap_Allocations,
No_Local_Protected_Objects,
No_Local_Timing_Events,
No_Protected_Type_Allocators,
No_Relative_Delay,
No_Requeue_Statements,
No_Select_Statements,
No_Specific_Termination_Handlers,
No_Task_Allocators,
No_Task_Hierarchy,
No_Task_Termination,
Simple_Barriers,
Max_Entry_Queue_Length => 1,
Max_Protected_Entries => 1,
Max_Task_Entries => 0,
No_Dependence => Ada.Asynchronous_Task_Control,
No_Dependence => Ada.Calendar,
No_Dependence => Ada.Execution_Time.Group_Budget,
No_Dependence => Ada.Execution_Time.Timers,
No_Dependence => Ada.Task_Attributes);
```

The pragma Detect_Blocking plus many of the Restrictions identifiers are new to Ada 2005. These will now be described.

The pragma Detect_Blocking, as its name implies, ensures that the implementation will detect a potentially blocking operation in a protected operation and raise Program_Error. Without this pragma the implementation is not required to detect blocking and so tasks might be locked out for an unbounded time and the program might even deadlock.

The identifier No_Dynamic_Attachment means that there are no calls of the operations in the package Ada.Interrupts.

The identifier No_Dynamic_Priorities means that there is no dependence on the package Ada.Priorities as well as no uses of the attribute Priority (this is a new attribute for protected objects as explained at the end of this section).

Note that the rules are that you cannot read as well as not write the priorities – this applies to both the procedure for reading task priorities and reading the attribute for protected objects.

The identifier No_Local_Protected_Objects means that protected objects can only be declared at library level and the identifier No_Protected_Type_Allocators means that there are no allocators for protected objects or objects containing components of protected types.

The identifier No_Local_Timing_Events means that objects of the type Timing_Event in the package Ada.Real_Time.Timing_Events can only be declared at library level. This package is described in Section 6 below.

The identifiers No_Relative_Delay, No_Requeue_Statements, and No_Select_Statements mean that there are no relative delay, requeue or select statements respectively.

The identifier No_Specific_Termination_Handlers means that there are no calls of the procedure Set_Specific_Handler or the function Specific_Handler in the package Task_Termination and the identifier No_Task_Termination means that all tasks should run for ever. Note that we are permitted to set a fallback handler so that if any task does attempt to terminate then it will be detected.

The identifier Simple_Barriers means that the Boolean expression in a barrier of an entry of a protected object shall be either a static expression (such as True) or a Boolean component of the protected object itself.

The Restrictions identifier Max_Entry_Queue_Length sets a limit on the number of calls permitted on an entry queue. It is an important property of the Ravenscar profile that only one call is permitted at a time on an entry queue of a protected object.

The identifier No_Dependence is not specific to the Real-Time Systems annex and is properly described in the next paper. In essence it indicates that the program does not depend upon the given language defined package. In this case it means that a program conforming to the Ravenscar profile cannot use any of the packages Asynchronous_Task_Control, Calendar, Execution_Time.Group_Budget, Execution_Time.Timers and Task_Attributes. Some of these packages are new and are described later in this paper.

Note that No_Dependence cannot be used for No_Dynamic_Attachment because that would prevent use of the child package Ada.Interrupts.Names.

All the other restrictions identifiers used by the Ravenscar profile were already defined in Ada 95. Note also that the identifier No_Asynchronous_Control has been moved to Annex J because it can now be replaced by the use of No_Dependence.

5 Scheduling and dispatching

Another area of increased flexibility in Ada 2005 is that of task dispatching policies. In Ada 95, the only predefined policy is FIFO_Within_Priorities although other policies are permitted. Ada 2005 provides further pragmas, policies and packages which facilitate many different mechanisms such as non-preemption within priorities, the familiar Round Robin using timeslicing, and the more recently acclaimed Earliest Deadline First (EDF) policy. Moreover it is possible to mix different policies according to priority level within a partition.

In order to accommodate these many changes, Section D.2 (Priority Scheduling) of the Reference Manual has been reorganized as follows

D.2.1 The Task Dispatching Model
D.2.2 Task Dispatching Pragmas
D.2.3 Preemptive Dispatching
D.2.4 Non-Preemptive Dispatching
D.2.5 Round Robin Dispatching
D.2.6 Earliest Deadline First Dispatching

Overall control is provided by two pragmas. They are

pragma Task_Dispatching_Policy(*policy*_identifier);

pragma Priority_Specific_Dispatching(*policy*_identifer,
*first_priority*_expression, *last_priority*_expression);

The pragma Task_Dispatching_Policy, which already exists in Ada 95, applies the same policy throughout a whole partition. The pragma Priority_Specific_Dispatching, which is new in Ada 2005, can be used to set different policies for different ranges of priority levels.

The full set of predefined policies in Ada 2005 is

FIFO_Within_Priorities – This already exists in Ada 95. Within each priority level to which it applies tasks are dealt with on a first-in-first-out basis. Moreover, a task may preempt a task of a lower priority.

Non_Preemptive_FIFO_Within_Priorities – This is new in Ada 2005. Within each priority level to which it applies tasks run to completion or until they are blocked or execute a delay statement. A task cannot be preempted by one of higher priority. This sort of policy is widely used in high integrity applications.

Round_Robin_Within_Priorities – This is new in Ada 2005. Within each priority level to which it applies tasks are timesliced with an interval that can be specified. This is a very traditional policy widely used since the earliest days of concurrent programming.

EDF_Across_Priorities – This is new in Ada 2005. This provides Earliest Deadline First dispatching. The general idea is that within a range of priority levels, each task has a deadline and that with the earliest deadline is processed. This is a fashionable new policy and has mathematically provable advantages with respect to efficiency.

For further details of these policies consult the forthcoming book by Alan Burns and Andy Wellings [4].

These various policies are controlled by the package Ada.Dispatching plus two child packages. The root package has specification

```
package Ada.Dispatching is
   pragma Pure(Dispatching);
   Dispatching_Policy_Error: exception;
end Ada.Dispatching;
```

As can be seen this root package simply declares the exception Dispatching_Policy_Error which is used by the child packages.

The child package Round_Robin enables the setting of the time quanta for time slicing within one or more priority levels. Its specification is

```
with System;  use System;
with Ada.Real_Time;  use Ada.Real_Time;
package Ada.Dispatching.Round_Robin is
   Default_Quantum: constant Time_Span := implementation-defined;
   procedure Set_Quantum(Pri: in Priority, Quantum: in Time_Span);
   procedure Set_Quantum(Low, High: in Priority; Quantum: in Time_Span);
   function Actual_Quantum(Pri: Priority) return Time_Span;
   function Is_Round_Robin(Pri: Priority) return Boolean;
end Ada.Dispatching.Round_Robin;
```

The procedures Set_Quantum enable the time quantum to be used for time slicing to be set for one or a range of priority levels. The default value is of course the constant Default_Quantum. The function Actual_Quantum enables us to find out the current value of the quantum being used for a particular priority level. Its identifier reflects the fact that the implementation may not be able to apply the exact actual value given in a call of Set_Quantum. The function Is_Round_Robin enables us to check whether the round robin policy has been applied to the given priority level. If we attempt to do something stupid such as set the quantum for a priority level to which the round robin policy does not apply then the exception Dispatching_Policy_Error is raised.

The other new policy concerns deadlines and is controlled by a new pragma Relative_Deadline and the child package Dispatching.EDF. The syntax of the pragma is

```
pragma Relative_Deadline(relative_deadline_expression);
```

The deadline of a task is a property similar to priority and both are used for scheduling. Every task has a priority of type Integer and every task has a deadline of type Ada.Real_Time.Time. Priorities can be set when a task is created by pragma Priority

```
task T is
  pragma Priority(P);
```

and deadlines can similarly be set by the pragma Relative_Deadline thus

```
task T is
  pragma Relative_Deadline(RD);
```

The expression RD has type Ada.Real_Time.Time_Span. Note carefully that the pragma sets the relative and not the absolute deadline. The initial absolute deadline of the task is

```
Ada.Real_Time.Clock + RD
```

where the call of Clock is made between task creation and the start of its activation.

Both pragmas Priority and Relative_Deadline can appear in the main subprogram and they then apply to the environment task. If they appear in any other subprogram then they are ignored. Both properties can also be set via a discriminant. In the case of priorities we can write

```
task type TT(P: Priority) is
  pragma Priority(P);
  ...
end;

High_Task: TT(13);
Low_Task: TT(7);
```

We cannot do the direct equivalent for deadlines because Time_Span is private and so not discrete. We have to use an access discriminant thus

```
task type TT(RD: access Timespan) is
  pragma Relative_Deadline(RD.all);
  ...
end;

One_Sec: aliased constant Time_Span := Seconds(1);
Ten_Mins: aliased constant Time_Span := Minutes(10);

Hot_Task: TT(One_Sec'Access);
Cool_Task: TT(Ten_Mins'Access);
```

Note incidentally that functions Seconds and Minutes have been added to the package Ada.Real_Time. Existing functions Nanoseconds, Microseconds and Milliseconds in Ada 95 enable the convenient specification of short real time intervals (values of type Time_Span). However, the specification of longer intervals such as four minutes meant writing something like Milliseconds(240_000) or perhaps 4*60*Milliseconds(1000). In view of the fact that EDF scheduling and timers (see Section 6) would be likely to require longer times the functions Seconds and Minutes are added in Ada 2005. There is no function Hours because the range of time spans is only guaranteed to be 3600 seconds anyway.

If a task is created and it does not have a pragma Priority then its initial priority is that of the task that created it. If a task does not have a pragma Relative_Deadline then its initial absolute deadline is the constant Default_Deadline in the package Ada.Dispatching.EDF; this constant has the value Ada.Real_Time.Time_Last (effectively the end of the universe).

Priorities can be dynamically manipulated by the subprograms in the package Ada.Dynamic_Priorities and deadlines can similarly be manipulated by the subprograms in the package Ada.Dispatching.EDF whose specification is

```
with Ada.Real_Time;  use Ada.Real_Time;
with Ada.Task_Identification; use Ada.Task_Identification;
package Ada.Dispatching.EDF is
   subtype Deadline is Ada.Real_Time.Time;
   Default_Deadline: constant Deadline := Time_Last;
   procedure Set_Deadline(D: in Deadline; T: in Task_Id := Current_Task);
   procedure Delay_Until_And_Set_Deadline (Delay_Until_Time: in Time;
                                           Deadline_Offset: in Time_Span);
   function Get_Deadline(T: Task_Id := Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

The subtype Deadline is just declared as a handy abbreviation. The constant Default_Deadline is set to the end of the universe as already mentioned. The procedure Set_Deadline sets the deadline of the task concerned to the value of the parameter D. The long-winded Delay_Until_And_Set_Deadline delays the task concerned until the value of Delay_Until_Time and sets its deadline to be the interval Deadline_Offset from that time – this is useful for periodic tasks. The function Get_Deadline enables us to find the current deadline of a task.

It is important to note that this package can be used to set and retrieve deadlines for tasks whether or not they are subject to EDF dispatching. We could for example use an ATC on a deadline overrun (ACT = Asynchronous Transfer of Control using a select statement). Hence there is no function Is_EDF corresponding to Is_Round_Robin and calls of the subprograms in this package can never raise the exception Dispatching_Policy_Error.

If we attempt to apply one of the subprograms in this package to a task that has already terminated then Tasking_Error is raised. If the task parameter is Null_Task_Id then Program_Error is raised.

As mentioned earlier, a policy can be selected for a whole partition by for example

```
pragma Task_Dispatching_Policy(Round_Robin_Within_Priorities);
```

whereas in order to mix different policies across different priority levels we can write

```
pragma Priority_Specific_Dispatching(Round_Robin_Within_Priority, 1, 1);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 10);
pragma Priority_Specific_Dispatching(FIFO_Within_Priority, 11, 24);
```

This sets Round Robin at priority level 1, EDF at levels 2 to 10, and FIFO at levels 11 to 24. This means for example that none of the EDF tasks can run if any of the FIFO ones can. In other words if any tasks in the highest group can run then they will do so and none in the other groups can run. The scheduling within a range takes over only if tasks in that range can go and none in the higher ranges can.

Note that if we write

```
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 5);
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 6, 10);
```

then this is not the same us

```
pragma Priority_Specific_Dispatching(EDF_Across_Priorities, 2, 10);
```

despite the fact that the two ranges in the first case are contiguous. This is because in the first case any task in the 6 to 10 range will take precedence over any task in the 2 to 5 range whatever the deadlines. If there is just one range then only the deadlines count in deciding which tasks are scheduled.

This is emphasized by the fact that the policy name uses Across rather than Within. For other policies such as Round_Robin_Within_Priority two contiguous ranges would be the same as a single range.

We conclude this section with a few words about ceiling priorities.

In Ada 95, the priority of a task can be changed but the ceiling priority of a protected object cannot be changed. It is permanently set when the object is created using the pragma Priority. This is often done using a discriminant so that at least different objects of a given protected type can have different priorities. Thus we might have

```
protected type PT(P: Priority) is
  pragma Priority(P);
  ...
end PT;

PO: PT(7);                          -- ceiling priority is 7
```

The fact that the ceiling priority of a protected object is static can be a nuisance in many applications especially when the priority of tasks can be dynamic. A common workaround is to give a protected object a higher ceiling than needed in all circumstances (often called "the ceiling of ceilings"). This results in tasks having a higher active priority than necessary when accessing the protected object and this can interfere with the processing of other tasks in the system and thus upset overall schedulability. Moreover, it means that a task of high priority can access an object when it should not (if a task with a priority higher than the ceiling priority of a protected object attempts to access the object then Program_Error is raised – if the object has an inflated priority then this check will pass when it should not).

This difficulty is overcome in Ada 2005 by allowing protected objects to change their priority. This is done through the introduction of an attribute Priority which applies just to protected objects. It can only be accessed within the body of the protected object concerned.

As an example a protected object might have a procedure to change its ceiling priority by a given amount. This could be written as follows

```
protected type PT is
  procedure Change_Priority(Change: in Integer);
  ...
end;

protected body PT is
  procedure Change_Priority(Change: in Integer) is
  begin
    ...                             -- PT'Priority has old value here
    PT'Priority := PT'Priority + Change;
    ...                             -- PT'Priority has new value here
    ...
  end Change_Priority;
  ...
end PT;
```

Changing the ceiling priority is thus done while mutual exclusion is in force. Although the value of the attribute itself is changed immediately the assignment is made, the actual ceiling priority of the protected object is only changed when the protected operation (in this case the call of Change_ Priority) is finished.

Note the unusual syntax. Here we permit an attribute as the destination of an assignment statement. This happens nowhere else in the language. Other forms of syntax were considered but this seemed the most expressive.

6 CPU clocks and timers

Ada 2005 introduces three different kinds of timers. Two are concerned with monitoring the CPU time of tasks – one applies to a single task and the other to groups of tasks. The third timer measures real time rather than execution time and can be used to trigger events at specific real times. We will look first at the CPU timers because that introduces more new concepts.

The execution time of one or more tasks can be monitored and controlled by the new package Ada.Execution_Time plus two child packages.

Ada.Execution_Time – this is the root package and enables the monitoring of execution time of individual tasks.

Ada.Execution_Time.Timers – this provides facilities for defining and enabling timers and for establishing a handler which is called by the run time system when the execution time of the task reaches a given value.

Ada.Execution_Time.Group_Budgets – this enables several tasks to share a budget and provides means whereby action can be taken when the budget expires.

The execution time of a task, or CPU time as it is commonly called, is the time spent by the system executing the task and services on its behalf. CPU times are represented by the private type CPU_Time. This type and various subprograms are declared in the root package Ada.Execution_Time whose specification is as follows (as before we have added some use clauses in order to ease the presentation)

```
with Ada.Task_Identification;  use Ada.Task_Identification;
with Ada.Real_Time;  use Ada.Real_Time;
package Ada.Execution_Time is

   type CPU_Time is private;
   CPU_Time_First: constant CPU_Time;
   CPU_Time_Last: constant CPU_Time;
   CPU_Time_Unit: constant := implementation-defined-real-number;
   CPU_Tick: constant Time_Span;

   function Clock(T: Task_Id := Current_Task) return CPU_Time;

   function "+" (Left: CPU_Time; Right: Time_Span) return CPU_Time;
   function "+" (Left: Time_Span; Right: CPU_Time) return CPU_Time;
   function "–" (Left: CPU_Time; Right: Time_Span) return CPU_Time;
   function "–" (Left: CPU_Time; Right: CPU_Time) return Time_Span;

   function "<" (Left, Right: CPU_Time) return Boolean;
   function "<=" (Left, Right: CPU_Time) return Boolean;
   function ">" (Left, Right: CPU_Time) return Boolean;
   function ">=" (Left, Right: CPU_Time) return Boolean;

   procedure Split(T: in CPU_Time; SC: out Seconds_Count; TS: out Time_Span);
   function Time_Of(SC: Seconds_Count; TS: Time_Span := Time_Span_Zero)
                                                          return CPU_Time;
```

```
private
   ... -- not specified by the language
end Ada.Execution_Time;
```

The CPU time of a particular task is obtained by calling the function Clock with the task as parameter. It is set to zero at task creation.

The constants CPU_Time_First and CPU_Time_Last give the range of values of CPU_Time. CPU_Tick gives the average interval during which successive calls of Clock give the same value and thus is a measure of the accuracy whereas CPU_Time_Unit gives the unit of time measured in seconds. We are assured that CPU_Tick is no greater than one millisecond and that the range of values of CPU_Time is at least 50 years (provided always of course that the implementation can cope).

The various subprograms perform obvious operations on the type CPU_Time and the type Time_Span of the package Ada.Real_Time.

A value of type CPU_Time can be converted to a Seconds_Count plus residual Time_Span by the function Split which is similar to that in the package Ada.Real_Time. The function Time_Of similarly works in the opposite direction. Note the default value of Time_Span_Zero for the second parameter – this enables times of exact numbers of seconds to be given more conveniently thus

```
Four_Secs: CPU_Time := Time_Of(4);
```

In order to find out when a task reaches a particular CPU time we can use the facilities of the child package Ada.Execution_Time.Timers whose specification is

```
with System;  use System;
package Ada.Execution_Time.Timers is

   type Timer(T: not null access constant Task_Id) is tagged limited private;
   type Timer_Handler is access protected procedure (TM: in out Timer);

   Min_Handler_Ceiling: constant Any_Priority := implementation-defined;

   procedure Set_Handler(TM: in out Timer; In_Time: Time_Span; Handler: Timer_Handler);
   procedure Set_Handler(TM: in out Timer; At_Time: CPU_Time; Handler: Timer_Handler);

   function Current_Handler(TM: Timer) return Timer_Handler;
   procedure Cancel_Handler(TM: in out Timer; Cancelled: out Boolean);
   function Time_Remaining(TM: Timer) return Time_Span;

   Timer_Resource_Error: exception;

private
   ...  -- not specified by the language
end Ada.Execution_Time.Timers;
```

The general idea is that we declare an object of type Timer whose discriminant identifies the task to be monitored – note the use of **not null** and **constant** in the discriminant. We also declare a protected procedure which takes the timer as its parameter and which performs the actions required when the CPU_Time of the task reaches some value. Thus to take some action (perhaps abort for example although that would be ruthless) when the CPU_Time of the task My_Task reaches 2.5 seconds we might first declare

```
My_Timer: Timer(My_Task'Identity'Access);
Time_Max: CPU_Time := Time_Of(2, Milliseconds(500));
```

and then

```
protected Control is
  procedure Alarm(TM: in out Timer);
end;

protected body Control is
  procedure Alarm(TM: in out Timer) is
  begin
    -- abort the task
    Abort_Task(TM.T.all);
  end Alarm;
end Control;
```

Finally we set the timer in motion by calling the procedure Set_Handler which takes the timer, the time value and (an access to) the protected procedure thus

```
Set_Handler(My_Timer, Time_Max, Control.Alarm'Access);
```

and then when the CPU time of the task reaches Time_Max, the protected procedure Control.Alarm is executed. Note how the timer object incorporates the information regarding the task concerned using an access discriminant T and that this is passed to the handler via its parameter TM.

Aborting the task is perhaps a little violent. Another possibility is simply to reduce its priority so that it is no longer troublesome, thus

```
    -- cool that task
    Set_Priority(Priority'First, TM.T.all);
```

Another version of Set_Handler enables the timer to be set for a given interval (of type Time_Span).

The handler associated with a timer can be found by calling the function Current_Handler. This returns null if the timer is not set in which case we say that the timer is clear.

When the timer expires, and just before calling the protected procedure, the timer is set to the clear state. One possible action of the handler, having perhaps made a note of the expiration of the timer, it to set the handler again or perhaps another handler. So we might have

```
protected body Control is
  procedure Alarm(TM: in out Timer) is
  begin
    Log_Overflow(TM);                -- note that timer had expired
    -- and then reset it for another 500 milliseconds
    Set_Handler(TM, Milliseconds(500), Kill'Access);
  end Alarm;

  procedure Kill(TM: in out Timer) is
  begin
    -- expired again so kill it
    Abort_Task(TM.T.all);
  end Kill;
end Control;
```

In this scenario we make a note of the fact that the task has overrun and then give it another 500 milliseconds but with the handler Control.Kill so that the second time is the last chance.

Setting the value of 500 milliseconds directly in the call is a bit crude. It might be better to parameterize the protected type thus

```
protected type Control(MS: Integer) is ...
...
My_Control: Control(500);
```

and then the call of Set_Handler in the protected procedure Alarm would be

```
Set_Handler(TM, Milliseconds(MS), Kill'Access);
```

Observe that overload resolution neatly distinguishes whether we are calling Set_Handler with an absolute time or a relative time.

The procedure Cancel_Handler can be used to clear a timer. The out parameter Cancelled is set to True if the timer was in fact set and False if it was clear. The function Time_Remaining returns Time_Span_Zero if the timer is not set and otherwise the time remaining.

Note also the constant Min_Handler_Ceiling. This is the minimum ceiling priority that the protected procedure should have to ensure that ceiling violation cannot occur.

This timer facility might be implemented on top of a POSIX system. There might be a limit on the number of timers that can be supported and an attempt to exceed this limit will raise Timer_Resource_Error.

We conclude by summarizing the general principles. A timer can be set or clear. If it is set then it has an associated (non-null) handler which will be called after the appropriate time. The key subprograms are Set_Handler, Cancel_Handler and Current_Handler. The protected procedure has a parameter which identifies the event for which it has been called. The same protected procedure can be the handler for many events. The same general structure applies to other kinds of timers which will now be described.

In order to program various so-called aperiodic servers it is necessary for tasks to share a CPU budget.

This can be done using the child package Ada.Execution_Time.Group_Budgets whose specification is

```
with System;  use System;
package Ada.Execution_Time.Group_Budgets is

  type Group_Budget is tagged limited private;
  type Group_Budget_Handler is access protected procedure (GB: in out Group_Budget);

  type Task_Array is array (Positive range <>) of Task_Id;

  Min_Handler_Ceiling: constant Any_Priority := implementation-defined;

  procedure Add_Task(GB: in out Group_Budget; T: in Task_Id);
  procedure Remove_Task(GB: in out Group_Budget; T: in Task_Id);
  function Is_Member(GB: Group_Budget; T: Task_Id) return Boolean;
  function Is_A_Group_Member(T: Task_Id) return Boolean;
  function Members(GB: Group_Budget) return Task_Array;

  procedure Replenish(GB: in out Group_Budget; To: in Time_Span);
  procedure Add(GB: in out Group_Budget; Interval: in Time_Span);

  function Budget_Has_Expired(GB: Group_Budget) return Boolean;
  function Budget_Remaining(GB: Group_Budget) return Time_Span;

  procedure Set_Handler(GB: in out Group_Budget; Handler: in Group_Budget_Handler);
  function Current_Handler(GB: Group_Budget) return Group_Budget_Handler;
  procedure Cancel_Handler(GB: in out Group_Budget; Cancelled: out Boolean);
```

```
   Group_Budget_Error: exception;

private
   ...  -- not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

This has much in common with its sibling package Timers but there are a number of important differences.

The first difference is that we are here considering a CPU budget shared among several tasks. The type Group_Budget both identifies the group of tasks it covers and the size of the budget.

Various subprograms enable tasks in a group to be manipulated. The procedures Add_Task and Remove_Task add or remove a task. The function Is_Member identifies whether a task belongs to a specific group whereas Is_A_Group_Member identifies whether a task belongs to any group. A task cannot be a member of more than one group. An attempt to add a task to more than one group or remove it from the wrong group and so on raises Group_Budget_Error. Finally the function Members returns all the members of a group as an array.

The value of the budget (initially Time_Span_Zero) can be loaded by the procedure Replenish and increased by the procedure Add. Whenever a budget is non-zero it is counted down as the tasks in the group execute and so consume CPU time. Whenever a budget goes to Time_Span_Zero it is said to have become exhausted and is not reduced further. Note that Add with a negative argument can reduce a budget – it can even cause it to become exhausted but not make it negative.

The function Budget_Remaining simply returns the amount left and Budget_Has_Expired returns True if the budget is exhausted and so has value Time_Span_Zero.

Whenever a budget *becomes* exhausted (that is when the value transitions to zero) a hander is called if one has been set. A handler is a protected procedure as before and procedures Set_Handler, Cancel_Handler, and function Current_Handler are much as expected. But a major difference is that Set_Handler does not set the time value of the budget since that is done by Replenish and Add. The setting of the budget and the setting of the handler are decoupled in this package. Indeed a handler can be set even though the budget is exhausted and the budget can be counting down even though no handler is set. The reason for the different approach simply reflects the usage paradigm for the feature.

So we could set up a mechanism to monitor the CPU time usage of a group of three tasks TA, TB, and TC by first declaring an object of type Group_Budget, adding the three tasks to the group and then setting an appropriate handler. Finally we call Replenish which sets the counting mechanism going. So we might write

```
ABC: Group_Budget;
...
Add_Task(ABC, TA'Identity);
Add_Task(ABC, TB'Identity);
Add_Task(ABC, TC'Identity);

Set_Handler(ABC, Control.Monitor'Access);
Replenish(ABC, Seconds(10));
```

Remember that functions Seconds and Minutes have been added to the package Ada.Real_Time.

The protected procedure might be

```
protected body Control is
   procedure Monitor(GB: in out Group_Budget) is
   begin
```

```
      Log_Budget;
      Add(GB, Seconds(10));                    -- add more time
    end Monitor;
  end Control;
```

The procedure Monitor logs the fact that the budget was exhausted and then adds a further 10 seconds to it. Remember that the handler remains set all the time in the case of group budgets whereas in the case of the single task timers it automatically becomes cleared and has to be set again if required.

If a task terminates then it is removed from the group as part of the finalization process.

Note that again there is the constant Min_Handler_Ceiling.

The final kind of timer concerns real time rather than CPU time and so is provided by a child package of Ada.Real_Time whereas the timers we have seen so far were provided by child packages of Ada.Execution_Time. The specification of the package Ada.Real_Time.Timing_Events is

```
package Ada.Real_Time.Timing_Events is

  type Timing_Event is tagged limited private;
  type Timing_Event_Handler is access protected procedure (Event: in out Timing_Event);

  procedure Set_Handler(Event: in out Timing_Event; At_Time: Time;
                                    Handler: Timing_Event_Handler);
  procedure Set_Handler(Event: in out Timing_Event; In_Time: Time_Span;
                                    Handler: Timing_Event_Handler);

  function Is_Handler_Set(Event: Timing_Event) return Boolean;
  function Current_Handler(Event: Timing_Event) return Timing_Event_Handler;
  procedure Cancel_Handler(Event: in out Timing_Event; Cancelled: out Boolean);

  function Time_Of_Event(Event: Timing_Event) return Time;

private
  ...  -- not specified by the language
end Ada.Real_Time.Timing_Events;
```

This package provides a very low level facility and does not involve Ada tasks at all. It has a very similar pattern to the package Execution_Time.Timers. A handler can be set by Set_Handler and again there are two versions one for a relative time and one for absolute time. There are also subprograms Current_Handler and Cancel_Handler. If no handler is set then Current_Handler returns null.

Set_Handler also specifies the protected procedure to be called when the time is reached. Times are of course specified using the type Real_Time rather than CPU_Time.

A minor difference is that this package has a function Time_Of_Event rather than Time_Remaining.

A simple example was given in the introductory paper. We repeat it here for convenience. The idea is that we wish to ring a pinger when our egg is boiled after four minutes. The protected procedure might be

```
protected body Egg is
  procedure Is_Done(Event: in out Timing_Event) is
  begin
    Ring_The_Pinger;
  end Is_Done;
end Egg;
```

and then

```
Egg_Done: Timing_Event;
Four_Min: Time_Span := Minutes(4);
...
Put_Egg_In_Water;
Set_Handler(Event => Egg_Done, In_Time => Four_Min, Handler => Egg.Is_Done'Access);
-- now read newspaper whilst waiting for egg
```

This is unreliable because if we are interrupted between the calls of Put_Egg_In_Water and Set_Handler then the egg will be boiled for too long. We can overcome this by adding a further procedure to the protected object so that it becomes

```
protected Egg is
  procedure Boil(For_Time: in Time_Span);
  procedure Is_Done(Event: in out Timing_Event);
end Egg;

protected body Egg is

  Egg_Done: Timing_Event;

  procedure Boil (For_Time: in Time_Span) is
  begin
    Put_Egg_In_Water;
    Set_Handler(Egg_Done, For_Time, Is_Done'Access);
  end Boil;

  procedure Is_Done (Event: in out Timing_Event) is
  begin
    Ring_The_Pinger;
  end Is_Done;
end Egg;
```

This is much better. The timing mechanism is now completely encapsulated in the protected object and the procedure Is_Done is no longer visible outside. So all we have to do is

```
Egg.Boil(Minutes(4));
-- now read newspaper whilst waiting for egg
```

Of course if the telephone rings as the pinger goes off and before we have a chance to eat the egg then it still gets overdone. One solution is to eat the egg within the protected procedure Is_Done as well. A gentleman would never let a telephone call disturb his breakfast.

One protected procedure could be used to respond to several events. In the case of the CPU timer the discriminant of the parameter identifies the task; in the case of the group and real-time timers, the parameter identifies the event.

If we want to use the same timer for several events then various techniques are possible. Note that the timers are limited so we cannot test for them directly. However, they are tagged and so can be extended. Moreover, we know that they are passed by reference and that the parameters are considered aliased.

Suppose we are boiling six eggs in one of those French breakfast things with a different coloured holder for each egg. We can write

```
type Colour is (Black, Blue, Red, Green, Yellow, Purple);

Eggs_Done: array (Colour) of aliased Timing_Event;
```

We can then set the handler for the egg in the red holder by something like

```
Set_Handler(Eggs_Done(Red), For_Time, Is_Done'Access);
```

and then the protected procedure might be

```
procedure Is_Done(E: in out Timing_Event) is
begin
  for C in Colour loop
    if E'Access = Eggs_Done(C)'Access then
                                -- egg in holder colour C is ready
      ...
      return;
    end if;
  end loop;
                                -- falls out of loop – unknown event!
  raise Not_An_Egg ;
end Is_Done;
```

Although this does work it is more than a little distasteful to compare access values in this way and moreover requires a loop to see which event occurred.

A much better approach is to use type extension and view conversions. First we extend the type Timing_Event to include additional information about the event (in this case the colour) so that we can identify the particular event from within the handler

```
type Egg_Event is new Timing_Event with
  record
    Event_Colour: Colour;
  end record;
```

We then declare an array of these extended events (they need not be aliased)

```
Eggs_Done: array (Colour) of Egg_Event;
```

We can now call Set_Handler for the egg in the red holder

```
Set_Handler(Eggs_Done(Red), For_Time, Is_Done'Access);
```

This is actually a call on the Set_Handler for the type Egg_Event inherited from Timing_Event. But it is the same code anyway.

Remember that values of tagged types are always passed by reference. This means that from within the procedure Is_Done we can recover the underlying type and so discover the information in the extension. This is done by using view conversions.

In fact we have to use two view conversions, first we convert to the class wide type Timing_Event'Class and then to the specific type Egg_Event. And then we can select the component Event_Colour. In fact we can do these operations in one statement thus

```
procedure Is_Done(E: in out Timing_Event) is
  C: constant Colour := Egg_Event(Timing_Event'Class(E)).Event_Colour;
begin
                                -- egg in holder colour C is ready
  ...
end Is_Done;
```

Note that there is a check on the conversion from the class wide type Timing_Event'Class to the specific type Egg_Event to ensure that the object passed as parameter is indeed of the type

Egg_Event (or a further extension of it). If this fails then Tag_Error is raised. In order to avoid this possibility we can use a membership test. For example

```
procedure Is_Done(E: in out Timing_Event) is
  C: Colour;
begin
  if Timing_Event'Class(E) in Egg_Event then
    C := Egg_Event(Timing_Event'Class(E)).Event_Colour;
                              -- egg in holder colour C is ready
    ...
  else
                              -- unknown event – not an egg event!
    raise Not_An_Egg;
  end if;
end Is_Done;
```

The membership test ensures that the event is of the specific type Egg_Event. We could avoid the double conversion to the class wide type by introducing an intermediate variable.

It is important to appreciate that no dispatching is involved in these operations at all – everything is static apart from the membership test.

Of course, it would have been a little more flexible if the various subprograms took a parameter of type Timing_Event'Class but this would have conflicted with the Restrictions identifier No_Dispatch. Note that Ravenscar itself does not impose No_Dispatch but the restriction is in the High-Integrity annex and thus might be imposed on some high-integrity applications which might nevertheless wish to use timers in a simple manner.

A few minor points of difference between the timers are worth summarizing.

The two CPU timers have a constant Min_Handler_Ceiling. This prevents ceiling violation. It is not necessary for the real-time timer because the call of the protected procedure is treated like an interrupt and thus is at interrupt ceiling level.

The group budget timer and the real-time timer do not have an exception corresponding to Timer_Resource_Error for the single task CPU timer. As mentioned above, it is anticipated that the single timer might be implemented on top of a POSIX system in which case there might be a limit to the number of timers especially since each task could be using several timers. In the group case, a task can only be in one group so the number of group timers is necessarily less than the number of tasks and no limit is likely to be exceeded. In the real-time case the events are simply placed on the delay queue and no other resources are required anyway.

It should also be noted that the group timer could be used to monitor the execution time of a single task. However, a task can only be in one group and so only one timer could be applied to a task that way whereas, as just mentioned, the single CPU timer is quite different since a given task could have several timers set for it to expire at different times. Thus both kinds of timers have their own distinct usage patterns.

7 High Integrity Systems annex

There are a few changes to this annex. The most noticeable is that its title has been changed from Safety and Security to High Integrity Systems. This reflects common practice in that high-integrity is now the accepted general term for systems such as safety-critical systems and security-critical systems.

There are some small changes to reflect the introduction of the Ravenscar profile. It is clarified that tasking is permitted in a high-integrity system provided that it is well controlled through, for example, the use of the Ravenscar profile.

A new pragma Partition_Elaboration_Policy is introduced. Its syntax is

```
pragma Partition_Elaboration_Policy(policy_identifier);
```

Two policy identifiers are predefined, namely, Concurrent and Sequential. The pragma is a configuration pragma and so applies throughout a partition. The default policy is Concurrent.

The normal behaviour in Ada when a program starts is that a task declared at library level is activated by the environment task and can begin to execute before all library level elaboration is completed and before the main subprogram is called by the environment task. Race conditions can arise especially when several library tasks are involved. Problems also arise with the attachment of interrupt handlers.

If the policy Sequential is specified then the rules are changed. The following things happen in sequence

- The elaboration of all library units takes place (this is done by the environment task) but library tasks are not activated (we say their activation is deferred). Similarly the attachment of interrupt handlers is deferred.
- The environment task then attaches the interrupts.
- The library tasks are then activated. While this is happening the environment task is suspended.
- Finally, the environment task then executes the main subprogram in parallel with the executing tasks.

Note that from the library tasks' point of view they go seamlessly from activation to execution. Moreover, they are assured that all library units will have been elaborated and all handlers attached before they execute.

If Sequential is specified then

```
pragma Restrictions(No_Task_Hierarchy);
```

must also be specified. This ensures that all tasks are at library level.

A final small point is that the Restrictions identifiers No_Unchecked_Conversion and No_Unchecked_Deallocation are now banished to Annex J because No_Dependence can be used instead.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] ISO/IEC TR 24718:2004 (2004) *Guide for the use of the Ada Ravenscar Profile in high integrity systems*. This is based on University of York Technical Report YCS-2003-348 (2003).

[3] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

[4] A. Burns and A. Wellings (2006) *Concurrent and Real-Time Programming In Ada 2005*, Cambridge University Press.

Rationale for Ada 2005: 5 Exceptions, generics etc

John Barnes
John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements in a number of general areas in Ada 2005.

There are some minor almost cosmetic improvements in the exceptions area which add to convenience rather than functionality. There are some important changes in the numerics area: one concerns mixing signed and unsigned integers and another concerns fixed point multiplication and division.

There are also a number of additional pragmas and Restrictions identifiers mostly of a safety-related nature.

Finally there are a number of improvements in the generics area such as better control of partial parameters of formal packages.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 3, September 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The areas mentioned in this paper are not specifically mentioned in the WG9 guidance document [1] other than under the request to remedy shortcomings and improve interfacing.

The following Ada Issues cover the relevant changes and are described in detail in this paper.

161 Preelaborable initialization

216 Unchecked unions – variants without discriminant

224 pragma Unsuppress

241 Testing for null occurrence

251 Abstract interfaces to provide multiple inheritance

257 Restrictions for implementation defined entities

260 Abstract formal subprograms & dispatching constructors

267 Fast float to integer conversion

286 Assert pragma

317 Partial parameter lists for formal packages

329 pragma No_Return – procedures that never return

340 Mod attribute

361 Raise with message

364 Fixed point multiply and divide

368 Restrictions for obsolescent features

381 New Restrictions identifier – No_Dependence

394 Redundant Restrictions identifiers and Ravenscar

398 Parameters of formal packages given at most once

400 Wide and wide-wide images

414 pragma No_Return for overriding procedures

417 Lower bound of functions in Ada.Exceptions etc

419 Limitedness of derived types

420 Resolution of universal operations in Standard

423 Renaming, null exclusion and formal objects

These changes can be grouped as follows.

First there are some minor changes to exception handling. There are neater means for testing for null occurrence and raising an exception with a message (241, 361) and also wide and wide-wide versions of some procedures (400, 417).

The numerics area has a number of small but important changes. They are the introduction of an attribute Mod to aid conversion between signed and unsigned integers (340); changes to the rules for fixed point multiplication and division which permit user-defined operations (364, 420); and an attribute Machine_Rounding which can be used to aid fast conversions from floating to integer types (267).

A number of new pragmas and Restrictions identifiers have been added. These generally make for more reliable programming. The pragmas are: Assert, No_Return, Preelaborable_Initialization, Unchecked_Union, and Unsuppress (161, 216, 224, 286, 329, 414). The restrictions identifiers are No_Dependence, No_Implementation_Pragmas, No_Implementation_Restrictions, and No_Obsolescent_Features (257, 368, 381). Note that there are also other new pragmas and new restrictions identifiers concerned with tasking as described in the previous paper. However, the introduction of No_Dependence means that the identifiers No_Asynchronous_Control, No_Unchecked_Conversion and No_Unchecked_Deallocation are now obsolescent (394).

Finally there are changes in generic units. There are changes in generic parameters which are consequences of changes in other areas such as the introduction of interfaces and dispatching constructors as described in the paper on the object oriented model (parts of 251 and 260); there are also changes to formal access and derived types (419, 423). Also, it is now possible to give just some parameters of a formal package in the generic formal part (317, 398).

2 Exceptions

There are two minor improvements in this area.

One concerns the detection of a null exception occurrence which might be useful in a routine for analysing a log of exceptions. This is tricky because although a constant Null_Occurrence is declared in the package Ada.Exceptions, the type Exception_Occurrence is limited and no equality is provided. So the obvious test cannot be performed.

We can however apply the function Exception_Identity to a value of the type Exception_Occurrence and this returns the corresponding Exception_Id. Thus we could check to see whether a particular occurrence X was caused by Program_Error by writing

```
if Exception_Identity(X) = Program_Error'Identity then
```

However, in Ada 95, applying Exception_Identity to the value Null_Occurrence raises Constraint_Error so we have to resort to a revolting trick such as declaring a function as follows

```
function Is_Null_Occurrence(X: Exception_Occurrence) return Boolean is
  Id: Exception_Id;
begin
  Id := Exception_Identity(X);
  return False;
exception
  when Constraint_Error => return True;
end Is_Null_Occurrence;
```

We can now write some general analysis routine as

```
procedure Process_Ex(X: in Exception_Occurrence) is
begin
  if Is_Null_Occurrence(X) then                    -- OK in Ada 95
    -- process the case of a null occurrence
  else
    -- process proper occurrences
  end if;
end Process_Ex;
```

But the detection of Constraint_Error in Is_Null_Occurrence is clearly bad practice since it would be all too easy to mask some other error by mistake. Accordingly, in Ada 2005, the behaviour of Exception_Identity is changed to return Null_Id when applied to Null_Occurrence. So we can now dispense with the dodgy function Is_Null_Occurrence and just write

```
procedure Process_Ex(X: in Exception_Occurrence) is
begin
  if Exception_Identity(X) = Null_Id then          -- OK in 2005
    -- process the case of a null occurrence
  else
    -- process proper occurrences
  end if;
end Process_Ex;
```

Beware that, technically, we now have an incompatibility between Ada 95 and Ada 2005 since the nasty function Is_Null_Occurrence will always return False in Ada 2005.

Observe that Constraint_Error is also raised if any of the three functions Exception_Name, Exception_Message, or Exception_Information are applied to the value Null_Occurrence so the similar behaviour with Exception_Identity in Ada 95 is perhaps understandable at first sight. However, it is believed that it was not the intention of the language designers but got in by mistake. Actually the change described here was originally classified as a correction to Ada 95 but later reclassified as an amendment in order to draw more attention to it because of the potential incompatibility.

The other change in the exception area concerns the raise statement. It is now possible (optionally of course) to supply a message thus

```
raise An_Error with "A message";
```

This is purely for convenience and is identical to writing

```
Raise_Exception(An_Error'Identity, "A message");
```

There is no change to the form of raise statement without an exception which simply reraises an existing occurrence.

Note the difference between

```
raise An_Error;                    -- message is implementation defined
```

and

```
raise An_Error with "";            -- message is null
```

In the first case a subsequent call of Exception_Message returns implementation defined information about the error whereas in the second case it simply returns the given message which in this example is a null string.

Some minor changes to the procedure Raise_Exception are mentioned in Section 4 below.

There are also additional functions in the package Ada.Exceptions to return the name of an exception as a Wide_String or Wide_Wide_String. They have identifiers Wide_Exception_Name and Wide_Wide_Exception_Name and are overloaded to take a parameter of type Exception_Id or Exception_Occurrence. The lower bound of the strings returned by these functions and by the existing functions Exception_Name, Exception_Message and Exception_Information is 1 (Ada 95 forgot to state this for the existing functions). The reader will recall that similar additional functions (and forgetfulness) in the package Ada.Tags were mentioned in the paper on the object oriented model.

3 Numerics

Although Ada 95 introduced unsigned integer types in the form of modular types, nevertheless, the strong typing rules of Ada have not made it easy to get unsigned and signed integers to work together. The following discussion using Ada 95 is based on that in AI-340.

Suppose we wish to implement a simulation of a typical machine which has addresses and offsets. We make it a generic

```
generic
  type Address_Type is mod <>;
  type Offset_Type is range <>;
  ...
package Simulator is
  function Calc_Address(Base_Add: Address_Type;
                        Offset: Offset_Type) return Address_Type;
  ...
end Simulator;
```

Addresses are represented as unsigned integers (a modular type), whereas offsets are signed integers. The function Calc_Address aims to add an offset to a base address and return an address. The offset could be negative.

Naïvely we might hope to write

```
function Calc_Address(Base_Add: Address_Type;
                      Offset: Offset_Type) return Address_Type is
begin
  return Base_Add + Offset;              -- illegal
end Calc_Address;
```

but this is plainly illegal because Base_Add and Offset are of different types.

We can try a type conversion thus

```
return Base_Add + Address_Type(Offset);
```

or perhaps, since Address_Type might have a constraint,

```
return Base_Add + Address_Type'Base(Offset);
```

but in any case the conversion is doomed to raise Constraint_Error if Offset is negative.

We then try to be clever and write

```
return Base_Add + Address_Type'Base(Offset mod
                                    Offset_Type'Base(Address_Type'Modulus));
```

but this raises Constraint_Error if Address_Type'Modulus > Offset_Type'Base'Last which it often will be. To see this consider for example a 32-bit machine with

```
type Offset_Type is range –(2**31) .. 2**31–1;
type Address_Type is mod 2**32;
```

in which case Address_Type'Modulus is 2**32 which is greater than Offset_Type'Base'Last which is 2**31–1.

So we try an explicit test for a negative offset

```
if Offset >= 0 then
  return Base_Add + Address_Type'Base(Offset);
else
  return Base_Add - Address_Type'Base(–Offset);
end if;
```

But if Address_Type'Base'Last < Offset_Type'Last then this will raise Constraint_Error for some values of Offset. Unlikely perhaps but this is a generic and so ought to work for all possible pairs of types.

If we attempt to overcome this then we run into problems in trying to compare these two values since they are of different types and converting one to the other can raise the Constraint_Error problem once more. One solution is to use a bigger type to do the test but this may not exist in some implementations. We could of course handle the Constraint_Error and then patch up the answer. The ruthless programmer might even think of Unchecked_Conversion but this has its own problems. And so on – 'tis a wearisome tale.

The problem is neatly overcome in Ada 2005 by the introduction of a new functional attribute

```
function S'Mod(Arg: universal_integer) return S'Base;
```

S'Mod applies to any modular subtype S and returns

```
Arg mod S'Modulus
```

In other words it converts a *universal_integer* value to the modular type using the corresponding mathematical mod operation. We can then happily write

```
function Calc_Address(Base_Add: Address_Type;
                      Offset: Offset_Type) return Address_Type is
begin
  return Base_Add + Address_Type'Mod(Offset);
end Calc_Address;
```

and this always works.

The next topic in the numerics area concerns rounding. One of the problems in the design of any programming language is getting the correct balance between performance and portability. This is particularly evident with numeric types where the computer has to implement only a crude approximation to the mathematician's integers and reals. The best performance is achieved by using types and operations that correspond exactly to the hardware. On the other hand, perfect portability requires using types with precisely identical characteristics on all implementations.

An interesting example of this problem arises with conversions from a floating point type to an integer type when the floating type value is midway between two integer values.

In Ada 83 the rounding in the midway case was not specified. This upset some people and so Ada 95 went the other way and decreed that such rounding was always away from zero. As well as this rule for conversion to integer types, Ada 95 also introduced a functional attribute to round a floating value. Thus for a subtype S of a floating point type T we have

```
function S'Rounding(X: T) return T;
```

This returns the nearest integral value and for midway values rounds away from zero.

Ada 95 also gives a bit more control for the benefit of the statistically minded by introducing

```
function S'Unbiased_Rounding(X: T) return T;
```

This returns the nearest integral value and for midway values rounds to the even value.

However, there are many applications where we don't care which value we get but would prefer the code to be fast. Implementers have reported problems with the elementary functions where table look-up is used to select a particular polynomial expansion. Either polynomial will do just as well when at the midpoint of some range. However on some popular hardware such as the Pentium, doing the exact rounding required by Ada 95 just wastes time and the resulting function is perhaps 20% slower. This is serious in any comparison with C.

This problem is overcome in Ada 2005 by the introduction of a further attribute

```
function S'Machine_Rounding(X: T) return T;
```

This does not specify which of the adjacent integral values is returned if X lies midway. Note that it is not implementation defined but deliberately unspecified. This should discourage users from depending upon the behaviour on a particular implementation and thus writing non-portable code.

Zerophiles will be pleased to note that if S'Signed_Zeros is true and the answer is zero then it has the same sign as X.

It should be noted that Machine_Rounding, like the other rounding functions, returns a value of the floating point type and not perhaps *universal_integer* as might be expected. So it will typically be used in a context such as

```
X: Some_Float;
Index: Integer;
...
Index := Integer(Some_Float'Machine_Rounding(X));
...                              -- now use Index for table look-up
```

Implementations are urged to detect this case in order to generate fast code.

The third improvement to the core language in the numerics area concerns fixed point arithmetic. This is a topic that concerns few people but those who do use it probably feel passionately about it.

The trouble with floating point is that it is rather machine dependent and of course integers are just integers. Many application areas have used some form of scaled integers for many decades and the

Ada fixed point facility is important in certain applications where rigorous error analysis is desirable.

The model of fixed point was changed somewhat from Ada 83 to Ada 95. One change was that the concepts of model and safe numbers were replaced by a much simpler model just based on the multiples of the number *small*. Thus consider the type

```
Del: constant := 2.0**(–15);
type Frac is delta Del range –1.0 .. 1.0;
```

In Ada 83 small was defined to be the largest power of 2 not greater than Del, and in this case is indeed 2.0**(–15). But in Ada 95, small can be chosen by the implementation to be any power of 2 not greater than Del provided of course that the full range of values is covered. In both languages an aspect clause can be used to specify small and it need not be a power of 2. (Remember that representation clauses are now known as aspect clauses.)

A more far reaching change introduced in Ada 95 concerns the introduction of operations on the type *universal_fixed* and type conversion.

A minor problem in Ada 83 was that explicit type conversion was required in places where it might have been considered quite unnecessary. Thus supposing we have variables F, G, H of the above type Frac, then in Ada 83 we could not write

```
H := F * G;                    -- illegal in Ada 83
```

but had to use an explicit conversion

```
H := Frac(F * G);              -- legal in Ada 83
```

In Ada 83, multiplication was defined between any two fixed point types and produced a result of the type *universal_fixed* and an explicit conversion was then required to convert this to the type Frac.

This explicit conversion was considered to be a nuisance so the rule was changed in Ada 95 to say that multiplication was only defined between *universal_fixed* operands and delivered a *universal_fixed* result. Implicit conversions were then allowed for both operands and result provided the type resolution rules identified no ambiguity. So since the expected type was Frac and no other interpretation was possible, the implicit conversion was allowed and so in Ada 95 we can simply write

```
H := F * G;                    -- legal in Ada 95
```

Similar rules apply to division in both Ada 83 and Ada 95.

Note however that

```
F := F * G * H;                -- illegal
```

is illegal in Ada 95 because of the existence of the pervasive type Duration defined in Standard. The intermediate result could be either Frac or Duration. So we have to add an explicit conversion somewhere.

One of the great things about Ada is the ability to define your own operations. And in Ada 83 many programmers wrote their own arithmetic operations for fixed point. These might be saturation operations in which the result is not allowed to overflow but just takes the extreme implemented value. Such operations often match the behaviour of some external device. So we might declare

```
function "*"(Left, Right: Frac) return Frac is
begin
  return Standard."*"(Left, Right);
```

```
exception
  when Constraint_Error =>
    if (Left>0.0 and Right>0.0) or (Left<0.0 and Right<0.0) then
      return Frac'Last;
    else
      return Frac'First;
    end if;
end "*";
```

and similar functions for addition, subtraction, and division (taking due care over division by zero and so on). This works fine in Ada 83 and all calculations can now use the new operations rather than the predefined ones in a natural manner.

Note however that

```
H := Frac(F * G);
```

is now ambiguous in Ada 83 since both our own new "*" and the predefined "*" are possible interpretations. However, if we simply write the more natural

```
H := F * G;
```

then there is no ambiguity. So we can program in Ada 83 without the explicit conversion.

However, in Ada 95 we run into a problem when we introduce our own operations since

```
H := F * G;
```

is ambiguous because both the predefined operation and our own operation are possible interpretations of "*" in this context. There is no cure for this in Ada 95 except for changing our own multiplying operations to be procedures with identifiers such as mul and div. This is a very tedious chore and prone to errors.

It has been reported that because of this difficulty many projects using fixed point have not moved from Ada 83 to Ada 95.

This problem is solved in Ada 2005 by changing the name resolution rules to forbid the use of the predefined multiplication (division) operation if there is a user-defined primitive multiplication (division) operation for either operand type unless there is an explicit conversion on the result or we write Standard."*" (or Standard."/").

This means that when there is no conversion as in

```
H := F * G;
```

then the predefined operation cannot apply if there is a primitive user-defined "*" for one of the operand types. So the ambiguity is resolved. Note that if there is a conversion then it is still ambiguous as in Ada 83.

If we absolutely need to have a conversion then we can always use a qualification as well or just instead. Thus we can write

```
F := Frac'(F * G) * H;
```

and this will unambiguously use our own operation.

On the other hand if we truly want to use the predefined operation then we can always write

```
H := Standard."*"(F, G);
```

Another example might be instructive. Suppose we declare three types TL, TA, TV representing lengths, areas, and volumes. We use centimetres as the basic unit with an accuracy of 0.1 cm

together with corresponding consistent units and accuracies for areas and volumes. We might declare

```
type TL is delta 0.1 range –100.0 .. 100.0;
type TA is delta 0.01 range –10_000.0 .. 10_000.0;
type TV is delta 0.001 range –1000_000.0 .. 1000_000.0;
for TL'Small use TL'Delta;
for TA'Small use TA'Delta;
for TV'Small use TV'Delta;

function "*"(Left: TL; Right: TL) return TA;
function "*"(Left: TL; Right: TA) return TV;
function "*"(Left: TA Right: TL) return TV;
function "/"(Left: TV; Right: TL) return TA;
function "/"(Left: TV; Right: TA) return TL;
function "/"(Left: TA; Right: TL) return TL;

XL, YL: TL;
XA, YA: TA;
XV, YV: TV;
```

These types have an explicit small equal to their delta and are such that no scaling is required to implement the appropriate multiplication and division operations. This absence of scaling is not really relevant to the discussion below but simply illustrates why we might have several fixed point types and operations between them.

Note that all three types have primitive user-defined multiplication and division operations even though in the case of multiplication, TV only appears as a result type. Thus the predefined multiplication or division with any of these types as operands can only be considered if the result has a type conversion.

As a consequence the following are legal

```
XV := XL * XA;          -- OK, volume = length × area
XL := XV / XA;          -- OK, length = volume ÷ area
```

but the following are not because they do not match the user-defined operations

```
XV := XL * XL;          -- no, volume ≠ length × length
XV := XL / XA;          -- no, volume ≠ length ÷ area
XL := XL * XL;          -- no, length ≠ length × length
```

But if we insist on multiplying two lengths together then we can use an explicit conversion thus

```
XL := TL(XL * XL);      -- legal, predefined operation
```

and this uses the predefined operation.

If we need to multiply three lengths to get a volume without storing an intermediate area then we can write

```
XV := XL * XL * XL;
```

and this is unambiguous since there are no explicit conversions and so the only relevant operations are those we have declared.

It is interesting to compare this with the corresponding solution using floating point where we would need to make the unwanted predefined operations abstract as discussed in an earlier paper.

It is hoped that the reader has not found this discussion to be too protracted. Although fixed point is a somewhat specialized area, it is important to those who find it useful and it is good to know that the problems with Ada 95 have been resolved.

There are a number of other improvements in the numerics area but these concern the Numerics annex and so will be discussed in a later paper.

4 Pragmas and Restrictions

Ada 2005 introduces a number of new pragmas and Restrictions identifiers. Many of these were described in the previous paper when discussing tasking and the Real-Time and High Integrity annexes. For convenience here is a complete list giving the annex if appropriate.

The new pragmas are

Assert	
Assertion_Policy	
Detect_Blocking	High-Integrity
No_Return	
Preelaborable_Initialization	
Profile	Real-Time
Relative_Deadline	Real-Time
Unchecked_Union	Interface
Unsuppress	

The new Restrictions identifiers are

Max_Entry_Queue_Length	Real-Time
No_Dependence	
No_Dynamic_Attachment	Real-Time
No_Implementation_Attributes	
No_Implementation_Pragmas	
No_Local_Protected_Objects	Real-Time
No_Obsolescent_Features	
No_Protected_Type_Allocators	Real-Time
No_Relative_Delay	Real-Time
No_Requeue_Statements	Real-Time
No_Select_Statements	Real-Time
No_Synchronous_Control	Real-Time
No_Task_Termination	Real-Time
Simple_Barriers	Real-Time

We will now discuss in detail the pragmas and Restrictions identifiers in the core language and so not discussed in the previous paper.

First there is the pragma Assert and the associated pragma Assertion_Policy. Their syntax is as follows

```
pragma Assert([Check =>] boolean_expression [, [Message =>] string_expression]);

pragma Assertion_Policy(policy_identifier);
```

The first parameter of Assert is thus a boolean expression and the second (and optional) parameter is a string. Remember that when we write Boolean we mean of the predefined type whereas boolean includes any type derived from Boolean as well.

The parameter of Assertion_Policy is an identifier which controls the behaviour of the pragma Assert. Two policies are defined by the language, namely, Check and Ignore. Further policies may be defined by the implementation.

There is also a package Ada.Assertions thus

```
package Ada.Assertions is
  pragma Pure(Assertions);

  Assertion_Error: exception;

  procedure Assert(Check: in Boolean);
  procedure Assert(Check: in Boolean; Message: in String);
end Ada.Assertions;
```

The pragma Assert can be used wherever a declaration or statement is allowed. Thus it might occur in a list of declarations such as

```
N: constant Integer := ... ;
pragma Assert(N > 1);
A: Real_Matrix(1 .. N, 1 .. N);
EV: Real_Vector(1 .. N);
```

and in a sequence of statements such as

```
pragma Assert(Transpose(A) = A, "A not symmetric");
EV := Eigenvalues(A);
```

If the policy set by Assertion_Policy is Check then the above pragmas are equivalent to

```
if not N > 1 then
  raise Assertion_Error;
end if;
```

and

```
if not Transpose(A) = A then
  raise Assertion_Error with "A not symmetric";
end if;
```

Remember from Section 2 that a raise statement without any explicit message is not the same as one with an explicit null message. In the former case a subsequent call of Exception_Message returns implementation defined information whereas in the latter case it returns a null string. This same behaviour thus occurs with the Assert pragma as well – providing no message is not the same as providing a null message.

If the policy set by Assertion_Policy is Ignore then the Assert pragma is ignored at execution time – but of course the syntax of the parameters is checked during compilation.

The two procedures Assert in the package Ada.Assertions have an identical effect to the corresponding Assert pragmas except that their behaviour does not depend upon the assertion policy. Thus the call

```
Assert(Some_Test);
```

is always equivalent to

```
if not Some_Test then
  raise Assertion_Error;
end if;
```

In other words we could define the behaviour of

```
pragma Assert(Some_Test);
```

as equivalent to

```
if policy_identifier = Check then
  Assert(Some_Test);          -- call of procedure Assert
end if;
```

Note again that there are two procedures Assert, one with and one without the message parameter. These correspond to raise statements with and without an explicit message.

The pragma Assertion_Policy is a configuration pragma and controls the behaviour of Assert throughout the units to which it applies. It is thus possible for different policies to be in effect in different parts of a partition.

An implementation could define other policies such as Assume which might mean that the compiler is free to do optimizations based on the assumption that the boolean expressions are true although there would be no code to check that they were true. Careless use of such a policy could lead to erroneous behaviour.

There was some concern that pragmas such as Assert might be misunderstood to imply that static analysis was being carried out. Thus in the SPARK language [2], the annotation

```
--# assert N /= 0
```

is indeed a static assertion and the appropriate tools can be used to verify this.

However, other languages such as Eiffel have used **assert** in a dynamic manner as now introduced into Ada 2005 and, moreover, many implementations of Ada have already provided a pragma Assert so it is expected that there will be no confusion with its incorporation into the standard.

Another pragma with a related flavour is No_Return. This can be applied to a procedure (not to a function) and asserts that the procedure never returns in the normal sense. Control can leave the procedure only by the propagation of an exception or it might loop forever (which is common among certain real-time programs). The syntax is

```
pragma No_Return(procedure_local_name {, procedure_local_name});
```

Thus we might have a procedure Fatal_Error which outputs some message and then propagates an exception which can be handled in the main subprogram. For example

```
procedure Fatal_Error(Msg: in String) is
  pragma No_Return(Fatal_Error);
begin
  Put_Line(Msg);
  ...                                  -- other last wishes
  raise Death;
end Fatal_Error;
...
procedure Main is
  ...
  ...
  Put_Line("Program terminated successfully");
exception
  when Death =>
    Put_Line("Program terminated: known error");
```

```
    when others =>
      Put_Line("Program terminated: unknown error");
  end Main;
```

There are two consequences of supplying a pragma No_Return.

- The implementation checks at compile time that the procedure concerned has no explicit return statements. There is also a check at run time that it does not attempt to run into the final end – Program_Error is raised if it does as in the case of running into the end of a function.
- The implementation is able to assume that calls of the procedure do not return and so various optimizations can be made.

We might then have a call of Fatal_Error as in

```
function Pop return Symbol is
begin
  if Top = 0 then
    Fatal_Error("Stack empty");              -- never returns
  elsif
    Top := Top – 1;
    return S(Top+1);
  end if;
end Pop;
```

If No_Return applies to Fatal_Error then the compiler should not compile a jump after the call of Fatal_Error and should not produce a warning that control might run into the final end of Pop.

The pragma No_Return now applies to the predefined procedure Raise_Exception. To enable this to be possible its behaviour with Null_Id has had to be changed. In Ada 95 writing

```
Raise_Exception(Null_Id, "Nothing");
```

does nothing at all (and so does return in that case) whereas in Ada 2005 it is defined to raise Constraint_Error and so now never returns.

We could restructure the procedure Fatal_Error to use Raise_Exception thus

```
procedure Fatal_Error(Msg: in String) is
  pragma No_Return(Fatal_Error);
begin
  ...                              -- other last wishes
  Raise_Exception(Death'Identity, Msg);
end Fatal_Error;
```

Since pragma No_Return applies to Fatal_Error it is important that we also know that Raise_Exception cannot return.

The exception handler for Death in the main subprogram can now use Exception_Message to print out the message.

Remember also from Section 2 above that we can now also write

```
raise Death with Msg;
```

rather than call Raise_Exception.

The pragma No_Return is a representation pragma. If a subprogram has no distinct specification then the pragma No_Return is placed inside the body (as shown above). If a subprogram has a distinct specification then the pragma must follow the specification in the same compilation or

declarative region. Thus one pragma No_Return could apply to several subprograms declared in the same package specification.

It is important that dispatching works correctly with procedures that do not return. A non-returning dispatching procedure can only be overridden by a non-returning procedure and so the overriding procedure must also have pragma No_Return thus

```
type T is tagged ...
procedure P(X: T; ... );
pragma No_Return(P);
...
type TT is new T with ...
overriding
procedure P(X: TT; ... );
pragma No_Return(P);
```

The reverse is not true of course. A procedure that does return can be overridden by one that does not.

It is possible to give a pragma No_Return for an abstract procedure, but obviously not for a null procedure. A pragma No_Return can also be given for a generic procedure. It then applies to all instances.

The next new pragma is Preelaborable_Initialization. The syntax is

```
pragma Preelaborable_Initialization(direct_name);
```

This pragma concerns the categorization of library units and is related to pragmas such as Pure and Preelaborate. It is used with a private type and promises that the full type given by the parameter will indeed have preelaborable initialization. The details of its use will be explained in the next paper.

Another new pragma is Unchecked_Union. The syntax is

```
pragma Unchecked_Union(first_subtype_local_name);
```

The parameter has to denote an unconstrained discriminated record subtype with a variant part. The purpose of the pragma is to permit interfacing to unions in C. The following example was given in the Introduction

```
type Number(Kind: Precision) is
  record
    case Kind is
      when Single_Precision =>
        SP_Value: Long_Float;
      when Multiple_Precision =>
        MP_Value_Length: Integer;
        MP_Value_First: access Long_Float;
    end case;
  end record;

pragma Unchecked_Union(Number);
```

Specifying the pragma Unchecked_Union ensures the following

- The representation of the type does not allow space for any discriminants.
- There is an implicit suppression of Discriminant_Check.
- There is an implicit **pragma** Convention(C).

The above Ada text provides a mapping of the following C union

```
union {
  double spvalue;
  struct {
    int length;
    double* first;
    } mpvalue;
} number;
```

The general idea is that the C programmer has created a type which can be used to represent a floating point number in one of two ways according to the precision required. One way is just as a double length value (a single item) and the other way is as a number of items considered juxtaposed to create a multiple precision value. This latter is represented as a structure consisting of an integer giving the number of items followed by a pointer to the first of them. These two different forms are the two alternatives of the union.

In the Ada mapping the choice of precision is governed by the discriminant Kind which is of an enumeration type as follows

```
type Precision is (Single_Precision, Multiple_Precision);
```

In the single precision case the component SP_Value of type Long_Float maps onto the C component spvalue of type double.

The multiple precision case is somewhat troublesome. The Ada component MP_Value_Length maps onto the C component length and the Ada component MP_Value_First of type **access** Long_Float maps onto the C component first of type double*.

In our Ada program we can declare a variable thus

```
X: Number(Multiple_Precision);
```

and we then obtain a value in X by calling some C subprogram. We can then declare an array and map it onto the C sequence of double length values thus

```
A: array (1 .. X.MP_Value_Length) of Long_Float;
for A'Address use X.MP_Value_First.all'Address;
pragma Import(C, A);
```

The elements of A are now the required values. Note that we don't use an Ada array in the declaration of Number because there might be problems with dope information.

The Ada type can also have a non-variant part preceding the variant part and variant parts can be nested. It may have several discriminants.

When an object of an unchecked union type is created, values must be supplied for all its discriminants even though they are not stored. This ensures that appropriate default values can be supplied and that an aggregate contains the correct components. However, since the discriminants are not stored, they cannot be read. So we can write

```
X: Number := (Single_Precision, 45.6);
Y: Number(Single_Precision);
...
Y.SP_Value := 55.7;
```

The variable Y is said to have an inferable discriminant whereas X does not. Although it is clear that playing with unchecked unions is potentially dangerous, nevertheless Ada 2005 imposes certain

rules that avoid some dangers. One rule is that predefined equality can only be used on operands with inferable discriminants; Program_Error is raised otherwise. So

```
if Y = 55.8 then                -- OK
if X = 45.5 then                -- raises Program_Error
if X = Y then                   -- raises Program_Error
```

It is important to be aware that unchecked union types are introduced in Ada 2005 for the sole purpose of interfacing to C programs and not for living dangerously. Thus consider

```
type T(Flag: Boolean := False) is
  record
    case Flag is
      when False =>
        F1: Float := 0.0;
      when True =>
        F2: Integer := 0;
    end case;
  end record;
pragma Unchecked_Union(T);
```

The type T can masquerade as either type Integer or Float. But we should not use unchecked union types as an alternative to unchecked conversion. Thus consider

```
X: T;                           -- Float by default
Y: Integer := X.F2;             -- erroneous
```

The object X has discriminant False by default and thus has the value zero of type Integer. In the absence of the pragma Unchecked_Union, the attempt to read X.F2 would raise Constraint_Error because of the discriminant check. The use of Unchecked_Union suppresses the discriminant check and so the assignment will occur. But note that the ARM clearly says (11.5(26)) that if a check is suppressed and the corresponding error situation arises then the program is erroneous.

However, assigning a Float value to an Integer object using Unchecked_Conversion is not erroneous providing certain conditions hold such as that Float'Size = Integer'Size.

The final pragma to be considered is Unsuppress. Its syntax is

```
pragma Unsuppress(identifier);
```

The identifier is that of a check or perhaps All_Checks. The pragma Unsuppress is essentially the opposite of the existing pragma Suppress and can be used in the same places with similar scoping rules.

Remember that pragma Suppress gives an implementation the permission to omit the checks but it does not require that the checks be omitted (they might be done by hardware). The pragma Unsuppress simply revokes this permission. One pragma can override the other in a nested manner. If both are given in the same region then they apply from the point where they are given and the later one thus overrides.

A likely scenario would be that Suppress applies to a large region of the program (perhaps all of it) and Unsuppress applies to a smaller region within. The reverse would also be possible but perhaps less likely.

Note that Unsuppress does not override the implicit Suppress of Discriminant_Check provided by the pragma Unchecked_Union just discussed.

A sensible application of Unsuppress would be in the fixed point operations mentioned in Section 3 thus

```
function "*"(Left, Right: Frac) return Frac is
  pragma Unsuppress(Overflow_Check);
begin
  return Standard."*"(Left, Right);
exception
  when Constraint_Error =>
    if (Left>0.0 and Right>0.0) or (Left<0.0 and Right<0.0) then
      return Frac'Last;
    else
      return Frac'First;
    end if;
end "*";
```

The use of Unsuppress ensures that the overflow check is not suppressed even if there is a global Suppress for the whole program (or the user has switched checks off through the compiler command line). So Constraint_Error will be raised as necessary and the code will work correctly.

In Ada 95 the pragma Suppress has the syntax

```
pragma Suppress(identifier [ , [On =>] name]);       -- Ada 95
```

The second and optional parameter gives the name of the entity to which the permission applies. There was never any clear agreement on what this meant and implementations varied. Accordingly, in Ada 2005 the second parameter is banished to Annex J so that the syntax in the core language is similar to Unsuppress thus

```
pragma Suppress(identifier);                         -- Ada 2005
```

For symmetry, Annex J actually allows an obsolete On parameter for Unsuppress. It might seem curious that a feature should be born obsolescent.

A number of new Restrictions identifiers are added in Ada 2005. The first is No_Dependence whose syntax is

```
pragma Restrictions(No_Dependence => name);
```

This indicates that there is no dependence on a library unit with the given name.

The name might be that of a predefined unit but it could in fact be any unit. For example, it might be helpful to know that there is no dependence on a particular implementation-defined unit such as a package Superstring thus

```
pragma Restrictions(No_Dependence => Superstring);
```

Care needs to be taken to spell the name correctly; if we write Supperstring by mistake then the compiler will not be able to help us.

The introduction of No_Dependence means that the existing Restrictions identifier No_Asynchronous_Control is moved to Annex J since we can now write

```
pragma Restrictions(No_Dependence => Ada.Asynchronous_Task_Control);
```

Similarly, the identifiers No_Unchecked_Conversion and No_Unchecked_Deallocation are also moved to Annex J.

Note that the identifier No_Dynamic_Attachment which refers to the use of the subprograms in the package Ada.Interrupts cannot be treated in this way because of the child package

Ada.Interrupts.Names. No dependence on Ada.Interrupts would exclude the use of the child package Names as well.

The restrictions identifier No_Dynamic_Priorities cannot be treated this way either for a rather different reason. In Ada 2005 this identifier is extended so that it also excludes the use of the attribute Priority and this would not be excluded by just saying no dependence on Ada.Dynamic_Priorities.

Two further Restrictions identifiers are introduced to encourage portability. We can write

```
pragma Restrictions(No_Implementation_Pragmas, No_Implementation_Attributes);
```

These do not apply to the whole partition but only to the compilation or environment concerned. This helps us to ensure that implementation dependent areas of a program are identified.

The final new restrictions identifier similarly prevents us from inadvertently using features in Annex J thus

```
pragma Restrictions(No_Obsolescent_Features);
```

Again this does not apply to the whole partition but only to the compilation or environment concerned. (It is of course not itself defined in Annex J.)

The reader will recall that in Ada 83 the predefined packages had names such as Text_IO whereas in Ada 95 they are Ada.Text_IO and so on. In order to ease transition from Ada 83, a number of renamings were declared in Annex J such as

```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```

A mild problem is that the user could write these renamings anyway and we do not want the No_Obsolescent_Features restriction to prevent this. Moreover, implementations might actually implement the renamings in Annex J by just compiling them and we don't want to force implementations to use some trickery to permit the user to do it but not the implementation. Accordingly, whether the No_Obsolescent_Features restriction applies to these renamings or not is implementation defined.

5 Generic units

There are a number of improvements in the area of generics many of which have already been outlined in earlier papers.

A first point concerns access types. The introduction of types that exclude null means that a formal access type parameter can take the form

```
generic
  ...
  type A is not null access T;
  ...
```

The actual type corresponding to A must then itself be an access type that excludes null. A similar rule applies in reverse – if the formal parameter excludes null then the actual parameter must also exclude null. If the two did not match in this respect then all sorts of difficulties could arise.

Similarly if the formal parameter is derived from an access type

```
generic
  ...
  type FA is new A;                -- A is an access type
  ...
```

then the actual type corresponding to FA must exclude null if A excludes null and vice versa. Half of this rule is automatically enforced since a type derived from a type that excludes null will automatically exclude null. But the reverse is not true as mentioned in an earlier paper when discussing access types. If A has the declaration

```
type A is access all Integer;          -- does not exclude null
```

then we can declare

```
type NA is new A;                      -- does not exclude null
type NNA is new not null A;            -- does exclude null
```

and then NA matches the formal parameter FA in the above generic but NNA does not.

There is also a change to formal derived types concerning limitedness. In line with the changes described in the paper on the object oriented model, the syntax now permits **limited** to be stated explicitly thus

```
generic
  type T is limited new LT;                -- untagged
  type TT is limited new TLT with private; -- tagged
```

However, this can be seen simply as a documentation aid since the actual types corresponding to T and TT must be derived from LT and TLT and so will be limited if LT and TLT are limited anyway.

Objects of anonymous access types are now also allowed as generic formal parameters so we can have

```
generic
  A: access T := null;
  AN: in out not null access T;
  F: access function (X: Float) return Float;
  FN: not null access function (X: Float) return Float;
```

If the subtype of the formal object excludes null (as in AN and FN) then the actual must also exclude null but not vice versa. This contrasts with the rule for formal access types discussed above in which case both the formal type and actual type have to exclude null or not. Note moreover that object parameters of anonymous access types can have mode **in out**.

If the subprogram profile itself has access parameters that exclude null as in

```
generic
  PN: access procedure (AN: not null access T);
```

then the actual subprogram must also have access parameters that exclude null and so on. The same rule applies to named formal subprogram parameters. If we have

```
generic
  with procedure P(AN: not null access T);
  with procedure Q(AN: access T);
```

then the actual corresponding to P must have a parameter that excludes null but the actual corresponding to Q might or might not. The rule is similar to renaming – "not null must never lie". Remember that the matching of object and subprogram generic parameters is defined in terms of renaming. Here is an example to illustrate why the asymmetry is important. Suppose we have

```
generic
  type T is private;
  with procedure P(Z: in T);
package G is
```

This can be matched by

```
type A is access ...;
procedure Q(Y: in not null A);
...
package NG is new G(T => A; P => Q);
```

Note that since the formal type T is not known to be an access type in the generic declaration, there is no mechanism for applying a null exclusion to it. Nevertheless there is no reason why the instantiation should not be permitted.

There are some other changes to existing named formal subprogram parameters. The reader will recall from the discussion on interfaces in an earlier paper that the concept of null procedures has been added in Ada 2005. A null procedure has no body but behaves as if it has a body comprising a null statement. It is now possible to use a null procedure as a possible form of default for a subprogram parameter. Thus there are now three possible forms of default as follows

```
with procedure P( ... ) is <>;          -- OK in 95
with procedure Q( ... ) is Some_Proc;   -- OK in 95
with procedure R( ... ) is null;        -- only in 2005
```

So if we have

```
generic
  type T is (<>);
  with procedure R(X: in Integer; Y: in out T) is null;
package PP ...
```

then an instantiation omitting the parameter for R such as

```
package NPP is new PP(T => Colour);
```

is equivalent to providing an actual procedure AR thus

```
procedure AR(X: in Integer; Y: in out Colour) is
begin
  null;
end AR;
```

Note that the profile of the actual procedure is conjured up to match the formal procedure.

Of course, there is no such thing as a null function and so null is not permitted as the default for a formal function.

A new kind of subprogram parameter was introduced in some detail when discussing object factory functions in the paper on the object oriented model. This is the abstract formal subprogram. The example given was the predefined generic function Generic_Dispatching_Constructor thus

```
generic
  type T (<>) is abstract tagged limited private;
  type Parameters (<>) is limited private;
  with function Constructor(Params: not null access Parameters) return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
        (The_Tag: Tag; Params: not null access Parameters) return T'Class;
```

The formal function Constructor is an example of an abstract formal subprogram. Remember that the interpretation is that the actual function must be a dispatching operation of a tagged type uniquely identified by the profile of the formal function. The actual operation can be concrete or

abstract. Formal abstract subprograms can of course be procedures as well as functions. It is important that there is exactly one controlling type in the profile.

Formal abstract subprograms can have defaults in much the same way that formal concrete subprograms can have defaults. We write

```
with procedure P(X: in out T) is abstract <>;
with function F return T is abstract Unit;
```

The first means of course that the default has to have identifier P and the second means that the default is some function Unit. It is not possible to give null as the default for an abstract parameter for various reasons. Defaults will probably be rarely used for abstract parameters.

The introduction of interfaces in Ada 2005 means that a new class of generic parameters is possible. Thus we might have

```
generic
  type F is interface;
```

The actual type could then be any interface. This is perhaps unlikely.

If we wanted to ensure that a formal interface had certain operations then we might first declare an interface A with the required operations

```
type A is interface;
procedure Op1(X: A; ... ) is abstract;
procedure N1(X: A; ... ) is null;
```

and then

```
generic
  type F is interface and A;
```

and then the actual interface must be descended from A and so have operations which match Op1 and N1.

A formal interface might specify several ancestors

```
generic
  type FAB is interface and A and B;
```

where A and B are themselves interfaces. And A and B or just some of them might themselves be further formal parameters as in

```
generic
  type A is interface;
  type FAB is interface and A and B;
```

These means that FAB must have both A and B as ancestors; it could of course have other ancestors as well.

The syntax for formal tagged types is also changed to take into account the possibility of interfaces. Thus we might have

```
generic
  type NT is new T and A and B with private;
```

in which case the actual type must be descended both from the tagged type T and the interfaces A and B. The parent type T itself might be an interface or a normal tagged type. Again some or all of T, A, and B might be earlier formal parameters. Also we can explicitly state **limited** in which case all of the ancestor types must also be limited.

An example of this sort of structure occurred when discussing printable geometric objects in the paper on the object oriented model. We had

```
generic
  type T is abstract tagged private;
package Make_Printable is
  type Printable_T is abstract new T and Printable with private;
  ...
end;
```

It might be that we have various interfaces all derived from Printable which serve different purposes (perhaps for different output devices, laser printer, card punch and so on). We would then want the generic package to take any of these interfaces thus

```
generic
  type T is abstract tagged private;
  type Any_Printable is interface and Printable;
package Make_Printable is
  type Printable_T is abstract new T and Any_Printable with private;
  ...
end;
```

A formal interface can also be marked as limited in which case the actual interface must also be limited and vice versa.

As discussed in the previous paper, interfaces can also be synchronized, task, or protected. Thus we might have

```
generic
  type T is task interface;
```

and then the actual interface must itself be a task interface. The correspondence must be exact. A formal synchronized interface can only be matched by an actual synchronized interface and so on. Remember from the discussion in the previous paper that a task interface can be composed from a synchronized interface. This flexibility does not extend to matching actual and formal generic parameters.

Another small change concerns object parameters of limited types. In Ada 95 the following is illegal

```
type LT is limited
  record
    A: Integer;
    B: Float;
  end record;                    -- a limited type

generic
  X: in LT;                      -- illegal in Ada 95
  ...
procedure P ...
```

It is illegal in Ada 95 because it is not possible to provide an actual parameter. This is because the parameter mechanism is one of initialization of the formal object parameter by the actual and this is treated as assignment and so is not permitted for limited types.

However, in Ada 2005, initialization of a limited object by an aggregate is allowed since the value is created *in situ* as discussed in an earlier paper. So an instantiation is possible thus

```
procedure Q is new P(X => (A => 1, B => 2.0), ... );
```

Remember that an initial value can also be provided by a function call and so the actual parameter could also be a function call returning a limited type.

The final improvement to the generic parameter mechanism concerns package parameters.

In Ada 95 package parameters take two forms. Given a generic package Q with formal parameters F1, F2, F3, then we can have

```
generic
  with package P is new Q(<>);
```

and then the actual package corresponding to the formal P can be any instantiation of Q. Alternatively

```
generic
  with package R is new Q(P1, P2, P3);
```

and then the actual package corresponding to R must be an instantiation of Q with the specified actual parameters P1, P2, P3.

As mentioned in the Introduction, a simple example of the use of these two forms occurs with the package Generic_Complex_Arrays which takes instantiations of Generic_Real_Arrays and Generic_Complex_Types which in turn both have the underlying floating type as their single parameter. It is vital that both packages use the same floating point type and this is assured by writing

```
generic
  with package Real_Arrays is new Generic_Real_Arrays(<>);
  with package Complex_Types is new Generic_Complex_Types(Real_Arrays.Real);
package Generic_Complex_Arrays is ...
```

However, the mechanism does not work very well when several parameters are involved as will now be illustrated with some examples.

The first example concerns using the new container library which will be discussed in some detail in a later paper. There are generic packages such as

```
generic
  type Index_Type is range <>;
  type Element_Type is private:
  with function "=" (Left, Right: Element_Type ) return Boolean is <>;
package Ada.Containers.Vectors is ...
```

and

```
generic
  type Key_Type is private;
  type Element_Type is private:
  with function Hash(Key: Key_Type) return Hash_Type;
  with function Equivalent_Keys(Left, Right: Key_Type) return Boolean;
  with function "=" (Left, Right: Element_Type ) return Boolean is <>;
package Ada.Containers.Hashed_Maps is ...
```

We might wish to pass instantiations of both of these to some other package with the proviso that both were instantiated with the same Element_Type. Otherwise the parameters can be unrelated.

It would be natural to make the vector package the first parameter and give it the (<>) form. But we then find that in Ada 95 we have to repeat all the parameters other than Element_Type for the maps package. So we have

```
with ... ; use Ada.Containers;
generic
  with package V is new Vectors(<>);
  type Key_Type is private;
  with function Hash(Key: Key_Type) return Hash_Type;
  with function Equivalent_Keys(Left, Right: Key_Type) return Boolean;
  with function "=" (Left, Right: Element_Type ) return Boolean is <>;
  with package HM is new Hashed_Maps(
                       Key_Type => Key_Type,
                       Element_Type => V.Element_Type,
                       Hash => Hash,
                       Equivalent_Keys => Equivalent_Keys,
                       "=" => "=");
package HMV is ...
```

This is a nuisance since when we instantiate HMV we have to provide all the parameters required by Hashed_Maps even though we must already have instantiated it elsewhere in the program. Suppose that instantiation was

```
package My_Hashed_Map is new Hashed_Maps(My_Key, Integer, Hash_It, Equiv, "=");
```

and suppose also that we have instantiated Vectors

```
package My_Vectors is new Vectors(Index, Integer, "=");
```

Now when we come to instantiate HMV we have to write

```
package My_HMV is
                       new HMV(My_Vectors, My_Key, Hash_It, Equiv, "=", My_Hashed_Maps);
```

This is very annoying. Not only do we have to repeat all the auxiliary parameters of Hashed_Maps but the situation regarding Vectors and Hashed_Maps is artificially made asymmetric. (Life would have been a bit easier if we had made Hashed_Maps the first package parameter but that just illustrates the asymmetry.) Of course we could more or less overcome the asymmetry by passing all the parameters of Vectors as well but then HMV would have even more parameters. This rather defeats the point of package parameters which were introduced into Ada 95 in order to avoid the huge parameter lists that had occurred in Ada 83.

Ada 2005 overcomes this problem by permitting just some of the actual parameters to be specified. Any omitted parameters are indicated using the <> notation thus

```
generic
  with package S is new Q(P1, F2 => <>, F3 => <>);
```

In this case the actual package corresponding to S can be any package which is an instantiation of Q where the first actual parameter is P1 but the other two parameters are left unspecified. We can also abbreviate this to

```
generic
  with package S is new Q(P1, others => <>);
```

Note that the <> notation can only be used with named parameters and also that (<>) is now considered to be a shorthand for (**others** => <>).

As another example

```
generic
  with package S is new Q(F1 => <>, F2 => P2, F3 => <>);
```

means that the actual package corresponding to S can be any package which is an instantiation of Q where the second actual parameter is P2 but the other two parameters are left unspecified. This can be abbreviated to

```
generic
  with package S is new Q(F2 => P2, others => <>);
```

Using this new notation, the package HMV can now simply be written as

```
with ... ; use Ada.Containers;
generic
  with package V is new Vectors(<>);
  with package HM is new Hashed_Maps
                    (Element_Type => V.Element_Type, others => <>);
package HMV is ...
```

and our instantiation of HMV becomes simply

```
package My_HMV is new HMV(My_Vectors, My_Hashed_Maps);
```

Some variations on this example are obviously possible. For example it is likely that the instantiation of Hashed_Maps must use the same definition of equality for the type Element_Type as Vectors. We can ensure this by writing

```
with ... ; use Ada.Containers;
generic
  with package V is new Vectors(<>);
  with package HM is new Hashed_Maps
                    (Element_Type => V.Element_Type, "=" => V."=", others => <>);
package HMV is ...
```

If this seems rather too hypothetical, a more concrete example might be a generic function which converts a vector into a list provided they have the same element type and equality. Note first that the specification of the container package for lists is

```
generic
  type Element_Type is private;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is ...
```

The specification of a generic function Convert might be

```
generic
  with package DLL is new Doubly_Linked_Lists(<>);
  with package V is new Vectors
                    (Index_Type => <>, Element_Type => DLL.Element_Type, "=" => DLL."=");
function Convert(The_Vector: V.Vector) return DLL.List;
```

On the other hand if we only care about the element types matching and not about equality then we could write

```
generic
  with package DLL is new Doubly_Linked_Lists(<>);
  with package V is new Vectors(Element_Type => DLL.Element_Type, others => <>);
function Convert(The_Vector: V.Vector) return DLL.List;
```

Note that if we had reversed the roles of the formal packages then we would not need the new <> notation if both equality and element type had to match but it would be necessary for the case where only the element type had to match.

Other examples might arise in the numerics area. Suppose we have two independently written generic packages Do_This and Do_That which both have a floating point type parameter and several other parameters as well. For example

```
generic
  type Real is digits <>;
  Accuracy: in Real;
  type Index is range <>;
  Max_Trials: in Index;
package Do_This is ...

generic
  type Floating is digits <>;
  Bounds: in Floating;
  Iterations: in Integer;
  Repeat: in Boolean;
package Do_That is ...
```

(This is typical of much numerical stuff. Authors are cautious and unable to make firm decisions about many aspects of their algorithms and therefore pass the buck back to the user in the form of a turgid list of auxiliary parameters.)

We now wish to write a package Super_Solver which takes instantiations of both Do_This and Do_That with the requirement that the floating type used for the instantiation is the same in each case but otherwise the parameters are unrelated. In Ada 95 we are again forced to repeat one set of parameters thus

```
generic
  with package This is new Do_This(<>);
  S_Bounds: in This.Real;
  S_Iterations: in Integer;
  S_Repeat: in Boolean;
  with package That is new Do_That(This.Real, S_Bounds, S_Iterations, S_Repeat);
package Super_Solver is ...
```

And when we come to instantiate Super_Solver we have to provide all the auxiliary parameters required by Do_That even though we must already have instantiated it elsewhere in the program. Suppose the instantiation was

```
package That_One is new Do_That(Float, 0.01, 7, False);
```

and suppose also that we have instantiated Do_This

```
package This_One is new Do_This( ... );
```

Now when we instantiate Super_Solver we have to write

```
package SS is new Super_Solver(This_One, 0.01, 7, False, That_One);
```

Just as with HMV we have all these duplicated parameters and an artificial asymmetry between This and That.

In Ada 2005 the package Super_Solver can be written as

```
generic
  with package This is new Do_This(<>);
  with package That is new Do_That(This.Real, others => <>);
package Super_Solver is ...
```

and the instantiation of Super_Solver becomes simply

```
package SS is new Super_Solver(This_One, That_One);
```

Other examples occur with signature packages. Remember that a signature package is one without a specification. It can be used to ensure that a group of entities are related in the correct way and an instantiation can then be used to identify the group as a whole. A trivial example might be

```
generic
  type Index is (<>);
  type item is private;
  type Vec is array (Index range <>) of Item;
package General_Vector is end;
```

An instantiation of General_Vector just asserts that the three types concerned have the appropriate relationship. Thus we might have

```
type My_Array is array (Integer range <>) of Float;
```

and then

```
package Vector is new General_Vector(Integer, Float, My_Array);
```

The package General_Vector could then be used as a parameter of other packages thereby reducing the number of parameters.

Another example might be the signature of a package for manipulating sets. Thus

```
generic
  type Element is private;
  type Set is private;
  with function Empty return Set;
  with function Unit(E: Element) return Set;
  with function Union(S, T: Set) return Set;
  with function Intersection(S, T: Set) return Set;
  ...
package Set_Signature is end;
```

We might then have some other generic package which takes an instantiation of this set signature. However, it is likely that we would need to specify the type of the elements but possibly not the set type and certainly not all the operations. So typically we would have

```
generic
  type My_Element is private;
  with package Sets is new Set_Signature(Element => My_Element, others => <>);
```

An example of this technique occurred when considering the possibility of including a system of units facility within Ada 2005. Although it was considered not appropriate to include it, the use of signature packages was almost essential to make the mechanism usable. The interested reader should consult AI-324.

We conclude by noting a small change to the syntax of a subprogram instantiation in that an overriding indicator can be supplied as mentioned in Section 7 of the paper on the object oriented model. Thus (in appropriate circumstances) we can write

```
overriding
procedure This is new That( ... );
```

This means that the instantiation must be an overriding operation for some type.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] J. G. P. Barnes (2003) *High Integrity Software – The SPARK Approach to Safety and Security*, Addison-Wesley.

Rationale for Ada 2005: 6 Predefined library

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various improvements to the predefined library in Ada 2005.

There are a number of important new core packages in Ada 2005. These include a number of packages for the manipulation of various types of containers, packages for directory operations and packages providing access to environment variables.

The entire ISO/IEC 10646:2003 character repertoire is now supported. Program text may now include other alphabets (such as Cyrillic and Greek) and wide-wide characters and strings are supported at run-time. There are also some improvements to the existing character, string and text input–output packages.

The Numerics annex now includes vector and matrix operations including those previously found in the secondary standard ISO/IEC 13813.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 4, December 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] says

"The main purpose of the Amendment is to address identified problems in Ada that are interfering with Ada's usage or adoption, especially in its major application areas (such as high-reliability, long-lived real-time and/or embedded applications and very large complex systems). The resulting changes may range from relatively minor, to more substantial."

Certainly one of the stated advantages of languages such as Java is that they come with a huge predefined library. By contrast the Ada library is somewhat Spartan and extensions to it should make Ada more accessible.

The guidance document also warns about secondary standards. Its essence is don't use secondary standards if you can get the material into the RM itself. And please put the stuff on vectors and matrices from ISO/IEC 13813 [2] into the RM. The reason for this exhortation is that secondary standards have proved themselves to be almost invisible and hence virtually useless.

We have already discussed the additional library packages in the area of tasking and real time in a previous paper. The following Ada issues cover the relevant changes in other areas and are described in detail in this paper:

161 Preelaborable initialization

248 Directory operations

270 Stream item size control

273 Use of PCS should not be normative

285 Support for 16-bit and 32-bit characters

296 Vector and matrix operations

301 Operations on language-defined strings

302 Container library

328 Non-generic version of Complex_IO

351 Time operations

362 Some predefined packages should be recategorized

366 More liberal rules for Pure units

370 Add standard interface for environment variables

388 Add Greek pi to Ada.Numerics

395 Clarifications concerning 16- and 32-bit characters

400 Wide and wide-wide images

418 Vector norm

427 Default parameters and Calendar operations

428 Input–output for bounded strings

441 Null streams

These changes can be grouped as follows.

First the container library is rather extensive and merits a whole paper alone (302). We only refer to it here for completeness.

New child packages of Calendar provide extra facilities for manipulating times and dates (351, 427).

There are additional packages in the core library providing access to aspects of the operational environment. These concern directory operations (248) and environment variables (370).

There are changes concerning characters both for writing program text itself and for handling characters and strings at run time. There is now support for 16- and 32-bit characters (285, 388, 395, 400), and there are additional operations in the string packages (301, 428).

The Numerics annex is enhanced by the addition of the vector and matrix material previously in ISO/IEC 13813 plus some commonly required linear algebra algorithms (296, 418) and a trivial addition concerning complex input–output (328).

The categorization of various predefined units has been changed in order to remove unnecessary restrictions on their use in distributed systems and similar applications (362, 366). The new pragma Preelaborable_Initialization is introduced as well for similar reasons (161). We can also group a minor change to the Distributed Systems annex here (273).

Finally there is new attribute Stream_Size in order to increase the portability of streams (270) and the parameter Stream of Read, Write etc now has a null exclusion (441).

2 The container library

This is a huge addition to the language and is described in a separate paper for convenience.

3 Times and dates

The first change to note is that the subtype Year_Number in the package Ada.Calendar in Ada 2005 is

```
subtype Year_Number is Integer range 1901 .. 2399;
```

In Ada 95 (and in Ada 83) the range is 1901 .. 2099. This avoids the leap year complexity caused by the 400 year rule at the expense of the use of dates in the far future. But, the end of the 21st century is perhaps not so far into the future, so it was decided that the 2.1k problem should be solved now rather than later. However, it was decided not to change the lower bound because some systems are known to have used that as a time datum. The upper bound was chosen in order to avoid difficulties for implementations. For example, with one nanosecond for Duration'Small, the type Time can just be squeezed into 64 bits.

Having grasped the nettle of doing leap years properly Ada 2005 dives in and deals with leap seconds, time zones and other such matters in pitiless detail.

There are three new child packages Calendar.Time_Zones, Calendar.Arithmetic and Calendar.Formatting. We will look at these in turn.

The specification of the first is

```
package Ada.Calendar.Time_Zones is

   -- Time zone manipulation:
   type Time_Offset is range –28*60 .. 28*60;
   Unknown_Zone_Error: exception;

   function UTC_Time_Offset(Date: Time := Clock) return Time_Offset;
end Ada.Calendar.Time_Zones;
```

Time zones are described in terms of the number of minutes different from UTC (which curiously is short for Coordinated Universal Time); this is close to but not quite the same as Greenwich Mean Time (GMT) and similarly does not suffer from leaping about in spring and falling about in the autumn. It might have seemed more natural to use hours but some places (for example India) have time zones which are not an integral number of hours different from UTC.

Time is an extraordinarily complex subject. The difference between GMT and UTC is never more than one second but at the moment of writing there is a difference of about 0.577 seconds. The BBC broadcast timesignals based on UTC but call them GMT and with digital broadcasting they turn up late anyway. The chronophile might find the website www.merlyn.demon.co.uk/misctime.htm# GMT of interest.

So the function UTC_Time_Offset applied in an Ada program in Paris to a value of type Time in summer should return a time offset of 120 (one hour for European Central Time plus one hour for daylight saving or heure d'été). Remember that the type Calendar.Time incorporates the date. To find the offset now (that is, at the time of the function call) we simply write

```
Offset := UTC_Time_Offset;
```

and then Clock is called by default.

To find what the offset was on Christmas Day 2000 we write

```
Offset := UTC_Time_Offset(Time_Of(2000, 12, 25));
```

and this should return 60 in Paris. So the poor function has to remember the whole history of local time changes since 1901 and predict them forward to 2399 – these Ada systems are pretty smart! In

reality the intent is to use whatever the underlying operating system provides. If the information is not known then it can raise Unknown_Zone_Error.

Note that we are assuming that the package Calendar is set to the local civil (or wall clock) time. It doesn't have to be but one expects that to be the normal situation. Of course it is possible for an Ada system running in California to have Calendar set to the local time in New Zealand but that would be unusual. Equally, Calendar doesn't have to adjust with daylight saving but we expect that it will. (No wonder that Ada.Real_Time was introduced for vital missions such as boiling an egg.)

A useful fact is that

```
Clock – Duration(UTC_Time_Offset*60)
```

gives UTC time – provided we don't do this just as daylight saving comes into effect in which case the call of Clock and that of UTC_Time_Offset might not be compatible.

More generally the type Time_Offset can be used to represent the difference between two time zones. If we want to work with the difference between New York and Paris then we could say

```
NY_Paris: Time_Offset := –360;
```

The time offset between two different places can be greater than 24 hours for two reasons. One is that the International Date Line weaves about somewhat and the other is that daylight saving time can extend the difference as well. Differences of 26 hours can easily occur and 27 hours is possible. Accordingly the range of the type Time_Offset allows for a generous 28 hours.

The package Calendar.Arithmetic provides some awkward arithmetic operations and also covers leap seconds. Its specification is

```
package Ada.Calendar.Arithmetic is

   -- Arithmetic on days:
   type Day_Count is range
     –366*(1+Year_Number'Last – Year_Number'First)
     ..
     +366*(1+Year_Number'Last – Year_Number'First);

   subtype Leap_Seconds_Count is Integer range –2047 .. 2047;

   procedure Difference(Left, Right: in Time;
                        Days: out Day_Count; Seconds: out Duration;
                        Leap_Seconds: out Leap_Seconds_Count);

   function "+" (Left: Time; Right: Day_Count) return Time;
   function "+" (Left: Day_Count; Right: Time) return Time;
   function "–" (Left: Time; Right: Day_Count) return Time;
   function "–" (Left, Right: Time) return Day_Count;

end Ada.Calendar.Arithmetic;
```

The range for Leap_Seconds_Count is generous. It allows for a leap second at least four time a year for the foreseeable future – the somewhat arbitrary range chosen allows the value to be accommodated in 12 bits. And the 366 in Day_Count is also a bit generous – but the true expression would be very unpleasant.

One of the problems with the old planet is that it is slowing down and a day as measured by the Earth's rotation is now a bit longer than 86,400 seconds. Naturally enough we have to keep the seconds uniform and so in order to keep worldly clocks synchronized with the natural day, an odd leap second has to be added from time to time. This is always added at midnight UTC (which means

it can pop up in the middle of the day in other time zones). The existence of leap seconds makes calculations with times somewhat tricky.

The basic trouble is that we want to have our cake and eat it. We want to have the invariant that a day has 86,400 seconds but unfortunately this is not always the case.

The procedure Difference operates on two values of type Time and gives the result in three parts, the number of days (an integer), the number of seconds as a Duration and the number of leap seconds (an integer). If Left is later then Right then all three numbers will be nonnegative; if earlier, then nonpositive.

Remember that Difference like all these other operations always works on local time as defined by the clock in Calendar (unless stated otherwise).

Suppose we wanted to find the difference between noon on June 1st 1982 and 2pm on July 1st 1985 according to a system set to UTC. We might write

```
Days: Day_Count;
Secs: Duration;
Leaps: Leap_Seconds_Count;
...
Difference(Time_Of(1985, 7, 1, 14*3600.0),
        Time_Of(1982, 6, 1, 12*3600.0), Days, Secs, Leaps);
```

The results should be

```
Days = 365+366+365+30 = 1126,
Secs = 7200.0,
Leaps = 2.
```

There were leap seconds on 30 June 1983 and 30 June 1985.

The functions "+" and "–" apply to values of type Time and Day_Count (whereas those in the parent Calendar apply only to Time and Duration and thus only work for intervals of a day or so). Note that the function "–" between two values of type Time in this child package produces the same value for the number of days as the corresponding call of the function Difference – leap seconds are completely ignored. Leap seconds are in fact ignored in all the operations "+" and "–" in the child package.

However, it should be noted that Calendar."–" counts the true seconds and so the expression

```
Calendar."–" (Time_Of(1985, 7, 1, 1*3600.0), Time_Of(1985, 6, 30, 23*3600.0))
```

has the Duration value 7201.0 and not 7200.0 because of the leap second at midnight that night. (We are assuming that our Ada system is running at UTC.) The same calculation in New York will produce 7200.0 because the leap second doesn't occur until 4 am in EST (with daylight saving).

Note also that

```
Calendar."–" (Time_Of(1985, 7, 1, 0.0), Time_Of(1985, 6, 30, 0.0))
```

in Paris where the leap second occurs at 10pm returns 86401.0 whereas the same calculation in New York will return 86400.0.

The third child package Calendar.Formatting has a variety of functions. Its specification is

```
with Ada.Calendar.Time_Zones;
use Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting is
```

```
-- Day of the week:
type Day_Name is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

function Day_Of_Week(Date: Time) return Day_Name;

-- Hours:Minutes:Seconds access:
subtype Hour_Number is Natural range 0 .. 23;
subtype Minute_Number is Natural range 0 .. 59;
subtype Second_Number is Natural range 0 .. 59;
subtype Second_Duration is Day_Duration range 0.0 .. 1.0;

function Year(Date: Time; Time_Zone: Time_Offset := 0) return Year_Number;

-- similarly functions Month, Day, Hour, Minute

function Second(Date: Time) return Second_Number;

function Sub_Second(Date: Time) return Second_Duration;

function Seconds_Of(Hour: Hour_Number;
        Minute: Minute_Number;
        Second: Second_Number := 0;
        Sub_Second: Second_Duration := 0.0) return Day_Duration;

procedure Split(Seconds: in Day_Duration;       -- (1)
        Hour: out Hour_Number;
        Minute: out Minute_Number;
        Second: out Second_Number;
        Sub_Second: out Second_Duration);

procedure Split(Date: in Time;                  -- (2)
        Year: out Year_Number;
        Month: out Month_Number;
        Day: out Day_Number;
        Hour: out Hour_Number;
        Minute: out Minute_Number;
        Second: out Second_Number;
        Sub_Second: out Second_Duration;
        Time_Zone: in Time_Offset := 0);

function Time_Of(Year: Year_Number;
        Month: Month_Number;
        Day: Day_Number;
        Hour: Hour_Number;
        Minute: Minute_Number;
        Second: Second_Number;
        Sub_Second: Second_Duration := 0.0;
        Leap_Second: Boolean := False;
        Time_Zone: Time_Offset := 0) return Time;

function Time_Of(Year: Year_Number;
        Month: Month_Number;
        Day: Day_Number;
        Seconds: Day_Duration;
        Leap_Second: Boolean := False;
        Time_Zone: Time_Offset := 0) return Time;
```

```
      procedure Split(Date: in Time;                    -- (3)
                ... -- as (2) but with additional parameter
                Leap_Second: out Boolean;
                Time_Zone: in Time_Offset := 0);

      procedure Split(Date: in Time;                    -- (4)
                ... -- as Calendar.Split
                ... -- but with additional parameters
                Leap_Second: out Boolean;
                Time_Zone: in Time_Offset := 0);

      -- Simple image and value:
      function Image(Date: Time;
                Include_Time_Fraction: Boolean := False;
                Time_Zone: Time_Offset := 0) return String;

      function Value(Date: String; Time_Zone: Time_Offset := 0) return Time;

      function Image (Elapsed_Time: Duration;
                Include_Time_Fraction: Boolean := False) return String;

      function Value(Elapsed_Time: String) return Duration;

   end Ada.Calendar.Formatting;
```

The function Day_Of_Week will be much appreciated. It is a nasty calculation.

Then there are functions Year, Month, Day, Hour, Minute, Second and Sub_Second which return the corresponding parts of a Time taking account of the time zone given as necessary. It is unfortunate that functions returning the parts of a time (as opposed to the parts of a date) were not provided in Calendar originally. All that Calendar provides is Seconds which gives the number of seconds from midnight and leaves users to chop it up for themselves. Note that Calendar.Second returns a Duration whereas the function in this child package is Seconds which returns an Integer. The fraction of a second is returned by Sub_Second.

Most of these functions have an optional parameter which is a time zone offset. Wherever in the world we are running, if we want to know the hour according to UTC then we write

```
Hour(Clock, UTC_Time_Offset)
```

If we are in New York and want to know the hour in Paris then we write

```
Hour(Clock, –360)
```

since New York is 6 hours (360 minutes) behind Paris.

Note that Second and Sub_Second do not have the optional Time_Offset parameter because offsets are an integral number of minutes and so the number of seconds does not depend upon the time zone.

The package also generously provides four procedures Split and two procedures Time_Of. These have the same general purpose as those in Calendar. There is also a function Seconds_Of. We will consider them in the order of declaration in the package specification above.

The function Seconds_Of creates a value of type Duration from components Hour, Minute, Second and Sub_Second. Note that we can use this together with Calendar.Time_Of to create a value of type Time. For example

```
T := Time_Of(2005, 4, 2, Seconds_Of(22, 4, 10, 0.5));
```

makes the time of the instant when I (originally) typed that last semicolon.

The first procedure Split is the reverse of Seconds_Of. It decomposes a value of type Duration into Hour, Minute, Second and Sub_Second. It is useful with the function Calendar.Split thus

```
Split(Some_Time, Y, M, D, Secs);  -- split time
Split(Secs, H, M, S, SS);         -- split secs
```

The next procedure Split (no 2) takes a Time and a Time_Offset (optional) and decomposes the time into its seven components. Note that the optional parameter is last for convenience. The normal rule for parameters of predefined procedures is that parameters of mode in are first and parameters of mode out are last. But this is a nuisance if parameters of mode in have defaults since this forces named notation if the default is used.

There are then two functions Time_Of which compose a Time from its various constituents and the Time_Offset (optional). One takes seven components (with individual Hour, Minute etc) whereas the other takes just four components (with Seconds in the whole day). An interesting feature of these two functions is that they also have a Boolean parameter Leap_Second which by default is False.

The purpose of this parameter needs to be understood carefully. Making up a typical time will have this parameter as False. But suppose we need to compose the time midway through the leap second that occurred on 30 June 1985 and assign it to a variable Magic_Moment. We will assume that our Calendar is in New York and set to EST with daylight saving (and so midnight UTC is 8pm in New York). We would write

```
Magic_Moment: Time := Time_Of(1985, 6, 30, 19, 59, 59, 0.5, True);
```

In a sense there were two 19:59:59 that day in New York. The proper one and then the leap one; the parameter distinguishes them. So the moment one second earlier is given by

```
Normal_Moment: Time := Time_Of(1985, 6, 30, 19, 59, 59, 0.5, False);
```

We could have followed ISO and used 23:59:60 UTC and so have subtype Second_Number **is** Natural **range** 0 .. 60; but this would have produced an incompatibility with Ada 95.

Note that if the parameter Leap_Second is True and the other parameters do not identify a time of a leap second then Time_Error is raised.

There are then two corresponding procedures Split (nos 3 and 4) with an out parameter Leap_Second. One produces seven components and the other just four. The difference between this seven-component procedure Split (no 3) and the earlier Split (no 2) is that this one has the out parameter Leap_Second whereas the other does not. Writing

```
Split(Magic_Moment, 0, Y, M, D, H, M, S, SS, Leap);
```

results in Leap being True whereas

```
Split(Normal_Moment, 0, Y, M, D, H, M, S, SS, Leap);
```

results in Leap being False but gives all the other out parameters (Y, ... , SS) exactly the same values.

On the other hand calling the version of Split (no 2) without the parameter Leap_Second thus

```
Split(Magic_Moment, 0, Y, M, D, H, M, S, SS);
Split(Normal_Moment, 0, Y, M, D, H, M, S, SS);
```

produces exactly the same results.

The reader might wonder why there are two Splits on Time with Leap_Second but only one without. This is because the parent package Calendar already has the other one (although without the time zone parameter). Another point is that in the case of Time_Of, we can default the Leap parameter being of mode in but in the case of Split the parameter has mode out and cannot be omitted. It would

be bad practice to encourage the use of a dummy parameter which is ignored and hence there have to be additional versions of Split.

Finally, there are two pairs of functions Image and Value. The first pair works with values of type Time. A call of Image returns a date and time value in the standard ISO 8601 format. Thus taking the Normal_Moment above

```
Image(Normal_Moment)
```

returns the following string

```
"1985-06-30 19:59:59"          -- in New York
```

If we set the optional parameter Include_Time_Fraction to True thus

```
Image(Normal_Moment, True)
```

then we get

```
"1985-06-30 19:59:59.50"
```

There is also the usual optional Time_Zone parameter so we could produce the time in Paris (from the program in New York) thus

```
Image(Normal_Moment, True, –360)
```

giving

```
"1985-07-01 02:59:59.50"       -- in Paris
```

The matching Value function works in reverse as expected.

We would expect to get exactly the same results with Magic_Moment. However, since some implementations might have an ISO function available in their operating system it is also allowed to produce

```
"1985-06-30 19:59:60"          -- in New York
```

The other Image and Value pair work on values of type Duration thus

```
Image(10_000.0)                -- "02:46:40"
```

with the optional parameter Include_Time_Fraction as before. Again the corresponding function Value works in reverse.

4 Operational environment

Two new packages are added to Ada 2005 in order to aid communication with the operational environment. They are Ada.Environment_Variables and Ada.Directories.

The package Ada.Environment_Variables has the following specification

```
package Ada.Environment_Variables is
   pragma Preelaborate(Environment_Variables);

   function Value(Name: String) return String;
   function Exists(Name: String) return Boolean;
   procedure Set(Name: in String; Value: in String);

   procedure Clear(Name: in String);
   procedure Clear;

   procedure Iterate(Process: not null access procedure (Name, Value: in String));

end Ada.Environment_Variables;
```

This package provides access to environment variables by name. What this means and whether it is supported depends upon the implementation. But most operating systems have environment variables of some sort. And if not, the implementation is encouraged to simulate them.

The values of the variable are also implementation defined and so simply represented by strings.

The behaviour is straightforward. We might check to see if there is a variable with the name "Ada" and then read and print her value and set it to 2005 if it is not, thus

```
if not Exists("Ada") then
   raise Horror;                    -- quel dommage!
end if;

Put("Current value of Ada is ");  Put_Line(Value("Ada"));

if Value("Ada") /= "2005" then
   Put_Line("Revitalizing Ada now");
   Set("Ada", "2005");
end if;
```

The procedure Clear with a parameter deletes the variable concerned. Thus Clear("Ada") eliminates her completely so that a subsequent call Exists("Ada") will return False. Note that Set actually clears the variable concerned and then defines a new one with the given name and value. The procedure Clear without a parameter clears all variables.

We can iterate over the variables using the procedure Iterate. For example we can print out the current state by

```
procedure Print_One(Name, Value: in String) is
begin
   Put_Line(Name & "=" & Value);
end Print_One;
...
Iterate(Print_One'Access);
```

The procedure Print_One prints the name and value of the variable passed as parameters. We then pass an access to this procedure as a parameter to the procedure Iterate and Iterate then calls Print_One for each variable in turn.

Note that the slave procedure has both Name and Value as parameters. It might be thought that this was unnecessary since the user can always call the function Value. However, real operating systems can sometimes have several variables with the same name; providing two parameters ensures that the name/value pairs are correctly matched.

Attempting to misuse the environment package such as reading a variable that doesn't exist raises Constraint_Error or Program_Error.

There are big dangers of race conditions because the environment variables are really globally shared. Moreover, they might be shared with the operating system itself as well as programs written in other languages.

A particular point is that we must not call the procedures Set or Clear within a procedure passed as a parameter to Iterate.

The other environment package is Ada.Directories. Its specification is

```
with Ada.IO_Exceptions;
with Ada.Calendar;
package Ada.Directories is
```

```
   -- Directory and file operations:
   function Current_Directory return String;
   procedure Set_Directory(Directory: in String);
   procedure Create_Directory(New_Directory: in String; Form: in String := "");
   procedure Delete_Directory(Directory: in String);
   procedure Create_Path(New_Directory: in String; Form: in String := "");
   procedure Delete_Tree(Directory: in String);
   procedure Delete_File(Name: in String);
   procedure Rename(Old_Name: in String; New_Name: in String);
   procedure Copy_File(Source_Name: in String; Target_Name: in String; Form: in String := "");

   -- File and directory name operations:
   function Full_Name(Name: String) return String;
   function Simple_Name(Name: String) return String;
   function Containing_Directory(Name: String) return String;
   function Extension(Name: String) return String;
   function Base_Name(Name: String) return String;
   function Compose(Containing_Directory: String := ""; Name: String; Extension: String := "")
                                                                        return String;

   -- File and directory queries:
   type File_Kind is (Directory, Ordinary_File, Special_File);
   type File_Size is range 0 .. implementation_defined;
   function Exists(Name: String) return Boolean;
   function Kind(Name: String) return File_Kind;
   function Size(Name: String) return File_Size;
   function Modification_Time(Name: String) return Ada.Calendar.Time;

   -- Directory searching:
   type Directory_Entry_Type is limited private;
   type Filter_Type is array (File_Kind) of Boolean;
   type Search_Type is limited private;
   procedure Start_Search(Search: in out Search_Type;
                          Directory: in String; Pattern: in String;
                          Filter: in Filter_Type := (others => True));
   procedure End_Search(Search: in out Search_Type);
   function More_Entries(Search: Search_Type) return Boolean;
   procedure Get_Next_Entry(Search: in out Search_Type;
                          Directory_Entry: out Directory_Entry_Type);
   procedure Search(Directory: in String;
             Pattern: in String;
             Filter: in Filter_Type := (others => True);
             Process: not null access procedure
                    (Directory_Entry: in Directory_Entry_Type));

   -- Operations on Directory Entries:
   function Simple_Name(Directory_Entry: Directory_Entry_Type) return String;
   function Full_Name(Directory_Entry: Directory_Entry_Type) return String;
   function Kind(Directory_Entry: Directory_Entry_Type) return File_Kind;
   function Size(Directory_Entry: Directory_Entry_Type) return File_Size;
   function Modification_Time(Directory_Entry: Directory_Entry_Type)
                                                          return Ada.Calendar.Time;
```

```
   Status_Error: exception renames Ada.IO_Exceptions.Status_Error;
   Name_Error: exception renames Ada.IO_Exceptions.Name_Error;
   Use_Error: exception renames Ada.IO_Exceptions.Use_Error;
   Device_Error: exception renames Ada.IO_Exceptions.Device_Error;
private
   -- Not specified by the language
end Ada.Directories;
```

Most operating systems have some sort of tree-structured filing system. The general idea of this package is that it allows the manipulation of file and directory names as far as is possible in a unified manner which is not too dependent on the implementation and operating system.

Files are classified as directories, special files and ordinary files. Special files are things like devices on Windows and soft links on Unix; these cannot be created or read by the predefined Ada input–output packages.

Files and directories are identified by strings in the usual way. The interpretation is implementation defined.

The full name of a file is a string such as

```
"c:\adastuff\rat\library.doc"
```

and the simple name is

```
"library.doc"
```

At least that is in good old DOS. In Windows XP it is something like

```
"C:\Documents and Settings\john.JBI3\My Documents\adastuff\rat\library.doc"
```

For the sake of illustration we will continue with the simple DOS example. The current directory is that set by the "cd" command. So assuming we have done

```
c:\>cd adastuff
c:\adastuff>
```

then the function Current_Directory will return the string "c:\adastuff". The procedure Set_Directory sets the current default directory. The procedures Create_Directory and Delete_Directory create and delete a single directory. We can either give the full name or just the part starting from the current default. Thus

```
Create_Directory("c:\adastuff\history");
Delete_Directory("history");
```

will cancel out.

The procedure Create_Path creates several nested directories as necessary. Thus starting from the situation above, if we write

```
Create_Path("c:\adastuff\books\old");
```

then it will first create a directory "books" in "c:\adastuff" and then a directory "old" in "books". On the other hand if we wrote Create_Path("c:\adastuff\rat"); then it would do nothing since the path already exists. The procedure Delete_Tree deletes a whole tree including subdirectories and files.

The procedures Delete_File, Rename and Copy_File behave as expected. Note in particular that Copy_File can be used to copy any file that could be copied using a normal input–output package such as Text_IO. For example, it is really tedious to use Text_IO to copy a file intact including all line and page terminators. It is a trivial matter using Copy_File.

Note also that the procedures Create_Directory, Create_Path and Copy_File have an optional Form parameter. Like similar parameters in the predefined input–output packages the meaning is implementation defined.

The next group of six functions, Full_Name, Simple_Name, Containing_Directory, Extension, Base_Name and Compose just manipulate strings representing file names and do not in any way interact with the actual external file system. Moreover, of these, only the behaviour of Full_Name depends upon the current directory.

The function Full_Name returns the full name of a file. Thus assuming the current directory is still "c:\adastuff"

```
Full_Name("rat\library.doc")
```

returns "c:\adastuff\rat\library.doc" and

```
Full_Name("library.doc")
```

returns "c:\adastuff\library.doc". The fact that such a file does not exist is irrelevant. We might be making up the name so that we can then create the file. If the string were malformed in some way (such as "66##77") so that the corresponding full name if returned would be nonsense then Name_Error is raised. But Name_Error is never raised just because the file does not exist.

On the other hand

```
Simple_Name("c:\adastuff\rat\library.doc")
```

returns "library.doc" and not "rat\library.doc". We can also apply Simple_Name to a string that does not go back to the root. Thus

```
Simple_Name("rat\library.doc");
```

is allowed and also returns "library.doc".

The function Containing_Directory_Name removes the simple name part of the parameter. We can even write

```
Containing_Directory_Name("..\rat\library.doc")
```

and this returns "..\rat"; note that it also removes the separator "\".

The functions Extension and Base_Name return the corresponding parts of a file name thus

```
Base_Name("rat\library.doc")        -- "library"
Extension("rat\library.doc")        -- "doc"
```

Note that they can be applied to a simple name or to a full name or, as here, to something midway between.

The function Compose can be used to put the various bits together, thus

```
Compose("rat", "library", "doc")
```

returns "rat\library.doc". The default parameters enable bits to be omitted. In fact if the third parameter is omitted then the second parameter is treated as a simple name rather than a base name. So we could equally write

```
Compose("rat","library.doc")
```

The next group of functions, Exists, Kind, Size and Modification_Time act on a file name (that is the name of a real external file) and return the obvious result. (The size is measured in stream elements – usually bytes.)

Various types and subprograms are provided to support searching over a directory structure for entities with appropriate properties. This can be done in two ways, either as a loop under the direct control of the programmer (sometimes called an active iterator) or via an access to subprogram parameter (often called a passive iterator). We will look at the active iterator approach first.

The procedures Start_Search, End_Search and Get_Next_Entry and the function More_Entries control the search loop. The general pattern is

```
Start_Search( ... );
while More_Entries( ... ) loop
  Get_Next_Entry( ... );
  ...                          -- do something with the entry found
end loop;
End_Search( ... );
```

Three types are involved. The type Directory_Entry_Type is limited private and acts as a sort of handle to the entries found. Valid values of this type can only be created by a call of Get_Next_Entry whose second parameter is an out parameter of the type Directory_Entry_Type. The type Search_Type is also limited private and contains the state of the search. The type Filter_Type provides a simple means of identifying the kinds of file to be found. It has three components corresponding to the three values of the enumeration type File_Kind and is used by the procedure Start_Search.

Suppose we want to look for all ordinary files with extension "doc" in the directory "c:\adastuff\rat". We could write

```
Rat_Search: Search_Type;
Item: Directory_Entry_Type;
Filter: Filter_Type := (Ordinary_File => True, others => False);
...
Start_Search(Rat_Search, "c:\adastuff\rat", "*.doc", Filter);
while More_Entries(Rat_Search) loop
  Get_Next_Entry(Rat_Search, Item);
  ...                          -- do something with Item
end loop;
End_Search(Rat_Search);
```

The third parameter of Start_Search (which is "*.doc" in the above example) represents a pattern for matching names and thus provides further filtering of the search. The interpretation is implementation defined except that a null string means match everything. However, we would expect that writing "*.doc" would mean search only for files with the extension "doc".

The alternative mechanism using a passive iterator is as follows. We first declare a subprogram such as

```
procedure Do_It(Item: in Directory_Entry_Type) is
begin
  ...                          -- do something with item
end Do_It;
```

and then declare a filter and call the procedure Search thus

```
Filter: Filter_Type := (Ordinary_File => True, others => False);
...
Search("c:\adastuff\rat", "*.doc", Filter, Do_It'Access);
```

The parameters of Search are the same as those of Start_Search except that the first parameter of type Search_Type is omitted and a final parameter which identifies the procedure Do_It is added. The variable Item which we declared in the active iterator is now the parameter Item of the procedure Do_It.

Each approach has its advantages. The passive iterator has the merit that we cannot make mistakes such as forget to call End_Search. But some find the active iterator easier to understand and it can be easier to use for parallel searches.

The final group of functions enables us to do useful things with the results of our search. Thus Simple_Name and Full_Name convert a value of Directory_Entry_Type to the corresponding simple or full file name. Having obtained the file name we can do everything we want but for convenience the functions Kind, Size and Modification_Time are provided which also directly take a parameter of Directory_Entry_Type.

So to complete this example we might print out a table of the files found giving their simple name, size and modification time. Using the active approach the loop might then become

```
while More_Entries(Rat_Search) loop
  Get_Next_Entry(Rat_Search, Item);
  Put(Simple_Name(Item));  Set_Col(15);
  Put(Size(Item/1000));  Put(" KB");  Set_Col(25);
  Put_Line(Image(Modification_Time(Item)));
end loop;
```

This might produce a table such as

```
access.doc      152 KB   2005-04-05  09:03:10
containers.doc  372 KB   2005-06-14  21:39:05
general.doc     181 KB   2005-03-03  08:43:15
intro.doc       173 KB   2004-11-25  15:52:20
library.doc     149 KB   2005-04-08  13:50:05
oop.doc         179 KB   2005-02-25  18:34:55
structure.doc   151 KB   2005-04-05  09:09:25
tasking.doc     174 KB   2005-03-31  11:16:40
```

Note that the function Image is from the package Ada.Calendar.Formatting discussed in the previous section.

Observe that the search is carried out on the directory given and does not look at subdirectories. If we want to do that then we can use the function Kind to identify subdirectories and then search recursively.

It has to be emphasized that the package Ada.Directories is very implementation dependent and indeed might not be supported by some implementations at all. Implementations are advised to provide any additional useful functions concerning retrieving other information about files (such as name of the owner or the original creation date) in a child package Ada.Directories.Information.

Finally, note that misuse of the various operations will raise one of the exceptions Status_Error, Name_Error, Use_Error or Device_Error from the package IO_Exceptions.

5 Characters and strings

An important improvement in Ada 2005 is the ability to deal with 16- and 32-bit characters both in the program text and in the executing program.

The fine detail of the changes to the program text are perhaps for the language lawyer. The purpose is to permit the use of all relevant characters of the entire ISO/IEC 10646:2003 repertoire. The most important effect is that we can write programs using Cyrillic, Greek and other character sets.

A good example is provided by the addition of the constant

```
π: constant := Pi;
```

to the package Ada.Numerics. This enables us to write mathematical programs in a more natural notation thus

```
Circumference: Float := 2.0 * π * Radius;
```

Other examples might be for describing polar coordinates thus

```
R: Float := Sqrt(X*X + Y*Y);
θ: Angle := Arctan(Y, X);
```

and of course in France we can now declare a decent set of ingredients for breakfast

```
type Breakfast_Stuff is (Croissant, Café, Œuf, Beurre);
```

Curiously, although the ligature æ is in Latin-1 and thus available in Ada 95 in identifiers, the ligature œ is not (for reasons we need not go into). However, in Ada 95, œ is a character of the type Wide_Character and so even in Ada 95 one can order breakfast thus

```
Put("Deux œufs easy-over avec jambon");          -- wide string
```

In order to manipulate 32-bit characters, Ada 2005 includes types Wide_Wide_Character and Wide_Wide_String in the package Standard and the appropriate operations to manipulate them in packages such as

```
Ada.Strings.Wide_Wide_Bounded
Ada.Strings.Wide_Wide_Fixed
Ada.Strings.Wide_Wide_Maps
Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants
Ada.Strings.Wide_Wide_Unbounded
Ada.Wide_Wide_Text_IO
Ada.Wide_Wide_Text_IO.Text_Streams
Ada.Wide_Wide_Text_IO.Complex_IO
Ada.Wide_Wide_Text_IO.Editing
```

There are also new attributes Wide_Wide_Image, Wide_Wide_Value and Wide_Wide_Width and so on.

The addition of wide-wide characters and strings introduces many additional possibilities for conversions. Just adding these directly to the existing package Ada.Characters.Handling could cause ambiguities in existing programs when using literals. So a new package Ada.Characters.Conversions has been added. This contains conversions in all combinations between Character, Wide_Character and Wide_Wide_Character and similarly for strings. The existing functions from Is_Character to To_Wide_String in Ada.Characters.Handling have been banished to Annex J.

The introduction of more complex writing systems makes the definition of the case insensitivity of identifiers, (the equivalence between upper and lower case), much more complicated.

In some systems, such as the ideographic system used by Chinese, Japanese and Korean, there is only one case, so things are easy. But in other systems, like the Latin, Greek and Cyrillic alphabets, upper and lower case characters have to be considered. Their equivalence is usually straightforward but there are some interesting exceptions such as

- Greek has two forms for lower case sigma (the normal form σ and the final form ς which is used at the end of a word). These both convert to the one upper case letter Σ.
- German has the lower case letter ß whose upper case form is made of two letters, namely SS.
- Slovenian has a grapheme LJ which is considered a single letter and has three forms: LJ, Lj and lj.

The Greek situation used to apply in English where the long s was used in the middle of words (where it looked like an f but without a cross stroke) and the familiar short s only at the end. To modern eyes this makes poetic lines such as "Where the bee sucks, there suck I" somewhat dubious. (This is sung by Ariel in Act V Scene I of The Tempest by William Shakespeare.)

The definition chosen for Ada 2005 closely follows those provided by ISO/IEC 10646:2003 and by the Unicode Consortium; this hopefully means that all users should find that the case insensitivity of identifiers works as expected in their own language.

Of interest to all users whatever their language is the addition of a few more subprograms in the string handling packages. As explained in the Introduction, Ada 95 requires rather too many conversions between bounded and unbounded strings and the raw type String and, moreover, multiple searching is inconvenient.

The additional subprograms in the packages are as follows.

In the package Ada.Strings.Fixed (assuming **use** Maps; for brevity)

```
function Index(Source: String; Pattern: String;
               From: Positive; Going: Direction := Forward;
               Mapping: Character_Mapping := Identity) return Natural;

function Index(Source: String; Pattern: String;
               From: Positive; Going: Direction := Forward;
               Mapping: Character_Mapping_Function) return Natural;

function Index(Source: String; Set: Character_Set;
               From: Positive; Test: Membership := Inside;
               Going: Direction := Forward) return Natural;

function Index_Non_Blank(Source: String;
               From: Positive; Going: Direction := Forward) return Natural;
```

The difference between these and the existing functions is that these have an additional parameter From. This makes it much easier to search for all the occurrences of some pattern in a string.

Similar functions are also added to the packages Ada.Strings.Bounded and Ada.Strings.Unbounded.

Thus suppose we want to find all the occurrences of "bar" in the string "barbara barnes" held in the variable BS of type Bounded_String. (I have put my wife into lower case for convenience.) There are 3 of course. The existing function Count can be used to determine this fact quite easily

```
N := Count(BS, "bar")                    -- is 3
```

But we really need to know where they are; we want the corresponding index values. The first is easy in Ada 95

```
I := Index(BS, "bar")                    -- is 1
```

But to find the next one in Ada 95 we have to do something such as take a slice by removing the first three characters and then search again. This would destroy the original string so we need to make a copy of at least part of it thus

```
Part := Delete(BS, I, I+2);                -- 2 is length "bar" – 1
I := Index(Part, "bar") + 3;               -- is 4
```

and so on in the not-so-obvious loop. (There are other ways such as making a complete copy first, this could either be in another bounded string or perhaps it is simplest just to copy it into a normal String first; but whatever we do it is messy.) In Ada 2005, having found the index of the first in I, we can find the second by writing

```
I := Index(BS, "bar", From => I+3);
```

and so on. This is clearly much easier.

The following are also added to Ada.Strings.Bounded

```
procedure Set_Bounded_String(Target: out Bounded_String;
          Source: in String; Drop: in Truncation := Error);

function Bounded_Slice(Source: Bounded_String;
          Low: Positive; High: Natural) return Bounded_String;

procedure Bounded_Slice(Source: in Bounded_String;
          Target: out Bounded_String;
          Low: in Positive; High: in Natural);
```

The procedure Set_Bounded_String is similar to the existing function To_Bounded_String. Thus rather than

```
BS := To_Bounded_String("A Bounded String");
```

we can equally write

```
Set_Bounded_String(BS, "A Bounded String");
```

The slice subprograms avoid conversion to and from the type String. Thus to extract the characters from 3 to 9 we can write

```
BS := Bounded_Slice(BS, 3, 9);                    -- "Bounded"
```

whereas in Ada 95 we have to write something like

```
BS := To_Bounded(Slice(BS, 3, 9));
```

Similar subprograms are added to Ada.Strings.Unbounded. These are even more valuable because unbounded strings are typically implemented with controlled types and the use of a procedure such as Set_Unbounded_String is much more efficient than the function To_Unbounded_String because it avoids assignment and thus calls of Adjust.

Input and output of bounded and unbounded strings in Ada 95 can only be done by converting to or from the type String. This is both slow and untidy. This problem is particularly acute with unbounded strings and so Ada 2005 provides the following additional package (we have added a use clause for brevity as usual)

```
with Ada.Strings.Unbounded;  use Ada.Strings.Unbounded;
package Ada.Text_IO.Unbounded_IO is

  procedure Put(File: in File_Type; Item: in Unbounded_String);
  procedure Put(Item: in Unbounded_String);

  procedure Put_Line(File: in File_Type; Item: in Unbounded_String);
  procedure Put_Line(Item: in Unbounded_String);

  function Get_Line(File: File_Type) return Unbounded_String;
  function Get_Line return Unbounded_String;
```

```
   procedure Get_Line(File: in File_Type; Item: out Unbounded_String);
   procedure Get_Line(Item: out Unbounded_String);

end Ada.Text_IO.Unbounded_IO;
```

The behaviour is as expected.

There is a similar package for bounded strings but it is generic. It has to be generic because the package Generic_Bounded_Length within Strings.Bounded is itself generic and has to be instantiated with the maximum string size. So the specification is

```
with Ada.Strings.Bounded;  use Ada.Strings.Bounded;
generic
  with package Bounded is new Generic_Bounded_Length(<>);
  use Bounded;
package Ada.Text_IO.Bounded_IO is

   procedure Put(File: in File_Type; Item: in Bounded_String);
   procedure Put(Item: in Bounded_String);

... -- etc as for Unbounded_IO

end Ada.Text_IO.Bounded_IO;
```

It will be noticed that these packages include functions Get_Line as well as procedures Put_Line and Get_Line corresponding to those in Text_IO. The reason is that procedures Get_Line are not entirely satisfactory.

If we do successive calls of the procedure Text_IO.Get_Line using a string of length 80 on a series of lines of length 80 (we are reading a nice old deck of punched cards), then it does not work as expected. Alternate calls return a line of characters and a null string (the history of this behaviour goes back to early Ada 83 days and is best left dormant).

Ada 2005 accordingly adds corresponding functions Get_Line to the package Ada.Text_IO itself thus

```
function Get_Line(File: File_Type) return String;
function Get_Line return String;
```

Successive calls of a function Get_Line then neatly return the text on the cards one by one without bother.

6 Numerics annex

When Ada 95 was being designed, the Numerics Rapporteur Group pontificated at length over what features should be included in Ada 95 itself, what should be placed in secondary standards, and what should be left to the creativeness of the user community.

A number of secondary standards had been developed for Ada 83. They were

11430 Generic package of elementary functions for Ada,

11729 Generic package of primitive functions for Ada,

13813 Generic package of real and complex type declarations and basic operations for Ada (including vector and matrix types),

13814 Generic package of complex elementary functions for Ada.

The first two, 11430 and 11729, were incorporated into the Ada 95 core language. The elementary functions, 11430, (Sqrt, Sin, Cos etc) became the package Ada.Numerics.Generic_Elementary_

Functions in A.5.1, and the primitive functions, 11729, became the various attributes such as Floor, Ceiling, Exponent and Fraction in A.5.3. The original standards were withdrawn long ago.

The other two standards, although originally developed as Ada 83 standards did not become finally approved until 1998.

In the case of 13814, the functionality was all incorporated into the Numerics annex of Ada 95 as the package Ada.Numerics.Generic_Complex_Elementary_Functions in G.1.2. Accordingly the original standard has now lapsed.

However, the situation regarding 13813 was not so clear. It covered four areas

1 a complex types package including various complex arithmetic operations,

2 a real arrays package covering both vectors and matrices,

3 a complex arrays package covering both vectors and matrices, and

4 a complex input–output package.

The first of these was incorporated into the Numerics annex of Ada 95 as the package Ada.Numerics.Generic_Complex_Types in G.1.1 and the last similarly became the package Ada.Text_IO.Complex_IO in G.1.3. However, the array packages, both real and complex, were not incorporated into Ada 95.

The reason for this omission is explained in Section G.1.1 of the Rationale for Ada 95 [3] which says

> A decision was made to abbreviate the Ada 95 packages by omitting the vector and matrix types and operations. One reason was that such types and operations were largely self-evident, so that little real help would be provided by defining them in the language. Another reason was that a future version of Ada might add enhancements for array manipulation and so it would be inappropriate to lock in such operations permanently.

The sort of enhancements that perhaps were being anticipated were facilities for manipulating arbitrary subpartitions of arrays such as were provided in Algol 68. These rather specialized facilities have not been added to Ada 2005 and indeed it seems most unlikely that they would ever be added. The second justification for omitting the vector and matrix facilities of 13813 thus disappears.

In order to overcome the objection that everything is self-evident we have taken the approach that we should further add some basic facilities that are commonly required, not completely trivial to implement, but on the other hand are mathematically well understood.

So the outcome is that Ada 2005 includes almost everything from 13813 plus subprograms for

- finding the norm of a vector,
- solving sets of linear equations,
- finding the inverse and determinant of a matrix,
- finding the eigenvalues and eigenvectors of a symmetric real or Hermitian matrix.

A small number of operations that were not related to linear algebra were removed (such as raising all elements of a matrix to a given power).

So Ada 2005 includes two new packages which are Ada.Numerics.Generic_Real_Arrays and Ada.Numerics.Generic_Complex_Arrays. It would take too much space to give the specifications of both in full so we give just an abbreviated form of the real package in which the specifications of the usual operators are omitted thus

```
generic
   type Real is digits <>;
package Ada.Numerics.Generic_Real_Arrays is
   pragma Pure(Generic_Real_Arrays);

   -- Types
   type Real_Vector is array (Integer range <>) of Real'Base;
   type Real_Matrix is array (Integer range <>, Integer range <>) of Real'Base;

   -- Real_Vector arithmetic operations
   ... -- unary and binary "+" and "–" giving a vector
   ... -- also inner product and two versions of "abs" – one returns a vector and the
   ... -- other a value of Real'Base

   -- Real_Vector scaling operations
   ... -- operations "*" and "/" to multiply a vector by a scalar and divide a vector by a scalar

   -- Other Real_Vector operations
   function Unit_Vector(Index: Integer; Order: Positive; First: Integer := 1) return Real_Vector;

   -- Real_Matrix arithmetic operations
   ... -- unary "+", "–", "abs", binary "+", "–" giving a matrix
   ... -- "*" on two matrices giving a matrix, on a vector and a matrix giving a vector,
   ... --  outer product of two vectors giving a matrix, and of course
   function Transpose(X: Real_Matrix) return Real_Matrix;

   -- Real_Matrix scaling operations
   ... -- operations "*" and "/" to multiply a matrix by a scalar and divide a matrix by a scalar

   -- Real_Matrix inversion and related operations
   function Solve(A: Real_Matrix; X: Real_Vector) return Real_Vector;
   function Solve(A, X: Real_Matrix) return Real_Matrix;
   function Inverse(A: Real_Matrix) return Real_Matrix;
   function Determinant(A: Real_Matrix) return Real'Base;

   -- Eigenvalues and vectors of a real symmetric matrix
   function Eigenvalues(A: Real_Matrix) return Real_Vector;
   procedure Eigensystem(A: in Real_Matrix;
                         Values: out Real_Vector; Vectors: out Real_Matrix);

   -- Other Real_Matrix operations
   function Unit_Matrix(Order: Positive; First_1, First_2: Integer := 1) return Real_Matrix;

end Ada.Numerics.Generic_Real_Arrays;
```

Many of these operations are quite self-evident. The general idea as far as the usual arithmetic operations are concerned is that we just write an expression in the normal way as illustrated in the Introduction. But the following points should be noted.

There are two operations "**abs**" applying to a Real_Vector thus

```
function "abs"(Right: Real_Vector) return Real_Vector;
function "abs"(Right: Real_Vector) return Real'Base;
```

One returns a vector each of whose elements is the absolute value of the corresponding element of the parameter (rather boring) and the other returns a scalar which is the so-called L2-norm of the vector. This is the square root of the inner product of the vector with itself or $\sqrt{(\Sigma x_i x_i)}$ – or just $\sqrt{(x_i x_i)}$ using the summation convention (which will be familiar to those who dabble in the relative world of

tensors). This is provided as a distinct operation in order to avoid any intermediate overflow that might occur if the user were to compute it directly using the inner product "*".

There are two functions Solve for solving one and several sets of linear equations respectively. Thus if we have the single set of *n* equations

$$\boldsymbol{Ax} = \boldsymbol{y}$$

then we might write

```
X, Y: Real_Vector(1 .. N);
A: Real_Matrix(1 .. N, 1 .. N);
...
Y := Solve(A, X);
```

and if we have *m* sets of *n* equations we might write

```
XX, YY: Real_Matrix(1 .. N, 1 .. M)
A: Real_Matrix(1 .. N, 1 .. N);
...
YY := Solve(A, XX);
```

The functions Inverse and Determinant are provided for completeness although they should be used with care. Remember that it is foolish to solve a set of equations by writing

```
Y := Inverse(A)*X;
```

because it is both slow and prone to errors. The main problem with Determinant is that it is liable to overflow or underflow even for moderate sized matrices. Thus if the elements are of the order of a thousand and the matrix has order 10, then the magnitude of the determinant will be of the order of 10^{30}. The user may therefore have to scale the data.

Two subprograms are provided for determining the eigenvalues and eigenvectors of a symmetric matrix. These are commonly required in many calculations in domains such as elasticity, moments of inertia, confidence regions and so on. The function Eigenvalues returns the eigenvalues (which will be non-negative) as a vector with them in decreasing order. The procedure Eigensystem computes both eigenvalues and vectors; the parameter Values is the same as that obtained by calling the function Eigenvalues and the parameter Vectors is a matrix whose columns are the corresponding eigenvectors in the same order. The eigenvectors are mutually orthonormal (that is, of unit length and mutually orthogonal) even when there are repeated eigenvalues. These subprograms apply only to symmetric matrices and if the matrix is not symmetric then Argument_Error is raised.

Other errors such as the mismatch of array bounds raise Constraint_Error by analogy with built-in array operations.

The reader will observe that the facilities provided here are rather humble and presented in a simple black-box style. It is important to appreciate that we do not see the Ada predefined numerics library as being in any way in competition with or as a substitute for professional libraries such as the renowned BLAS (Basic Linear Algebra Subprograms, see www.netlib.org/blas). Indeed our overall goal is twofold

- to provide commonly required simple facilities for the user who is not a numerical professional,
- to provide a baseline of types and operations that forms a firm foundation for binding to more general facilities such as the BLAS.

We do not expect users to apply the operations in our Ada packages to the huge matrices that arise in areas such as partial differential equations. Such matrices are often of a special nature such as

banded and need the facilities of a comprehensive numerical library. We have instead striven to provide easy to use facilities for the programmer who has a small number of equations to solve such as might arise in navigational applications.

Simplicity is evident in that functions such as Solve do not reveal the almost inevitable underlying LU decomposition or provide parameters controlling for example whether additional iterations should be applied. However, implementations are advised to apply an additional iteration and should document whether they do or not.

Considerations of simplicity also led to the decision not to provide automatic scaling for the determinant or to provide functions for just the largest eigenvalue and so on.

Similarly we only provide for the eigensystems of symmetric real matrices. These are the ones that commonly arise and are well behaved. General nonsymmetric matrices can be troublesome.

Appropriate accuracy requirements are specified for the inner product and L2-norm operations. Accuracy requirements for Solve, Inverse, Determinant, Eigenvalues and Eigenvectors are implementation defined which means that the implementation must document them.

The complex package is very similar and will not be described in detail. However, the generic formal parameters are interesting. They are

```
with Ada.Numerics.Generic_Real_Arrays, Ada.Numerics.Generic_Complex_Types;
generic
  with package Real_Arrays is new Ada.Numerics.Generic_Real_Arrays(<>);
  use Real_Arrays;
  with package Complex_Types is new Ada.Numerics.Generic_Complex_Types(Real);
  use Complex_Types;
package Ada.Numerics.Generic_Complex_Arrays is
  ...
```

Thus we see that it has two formal packages which are the corresponding real array package and the existing Ada 95 complex types and operations package. The formal parameter of the first is <> and that of the second is Real which is exported from the first package and ensures that both are instantiated with the same floating point type.

As well as the obvious array and matrix operations, the complex package also has operations for composing complex arrays from cartesian and polar real arrays, and computing the conjugate array by analogy with scalar operations in the complex types package. There are also mixed real and complex array operations but not mixed imaginary, real and complex array operations. Altogether the complex array package declares some 80 subprograms (there are around 30 in the real array package) and adding imaginary array operations would have made the package unwieldy (and the reference manual too heavy).

By analogy with real symmetric matrices, the complex package has subprograms for determining the eigensystems of Hermitian matrices. A Hermitian matrix is one whose complex conjugate equals its transpose; such matrices have real eigenvalues and are well behaved.

We conclude this discussion of the Numerics annex by mentioning one minute change regarding complex input–output. Ada 2005 includes preinstantiated forms of Ada.Text_IO.Complex_IO such as Ada.Complex_Text_IO (for when the underlying real type is the type Float), Ada.Long_Complex_Text_IO (for type Long_Float) and so on. These are by analogy with Float_Text_IO, Long_Float_Text_IO and their omission from Ada 95 was probably an oversight.

7 Categorization of library units

It will be recalled that library units in Ada 95 are categorized into a hierarchy by a number of pragmas thus

```
pragma Pure( ... );
pragma Shared_Passive( ... );
pragma Remote_Types( ... );
pragma Remote_Call_Interface( ... );
```

Each category imposes restrictions on what the unit can contain. An important rule is that a unit can only depend on units in the same or higher categories (the bodies of the last two are not restricted).

The pragmas Shared_Passive, Remote_Types, and Remote_Call_Interface concern distributed systems and thus are rather specialized. A minor change made in the 2001 Corrigendum was that the pragma Remote_Types was added to the package Ada.Finalization in order to support the interchange of controlled types between partitions in a distributed system.

Note that the pragma Preelaborate does not fit into this hierarchy. In fact there is another hierarchy thus

```
pragma Pure( ... );
pragma Preelaborate( ... );
```

and again we have the same rule that a unit can only depend upon units in the same or higher category. Thus a pure unit can only depend upon other pure units and a preelaborable unit can only depend upon other preelaborable or pure units.

A consequence of this dual hierarchy is that a shared passive unit cannot depend upon a preelaborable unit – the units upon which it depends have to be pure or shared passive and so on for the others. However, there is a separate rule that a unit which is shared passive, remote types or RCI must itself be preelaborable and so has to also have the pragma Preelaborate.

The categorization of individual predefined units is intended to make them as useful as possible. The stricter the category the more useful the unit because it can be used in more circumstances.

The categorization was unnecessarily weak in Ada 95 in some cases and some changes are made in Ada 2005.

The following packages which had no categorization in Ada 95 have pragma Preelaborate in Ada 2005

```
Ada.Asynchronous_Task_Control
Ada.Dynamic_Priorities
Ada.Exceptions
Ada.Synchronous_Task_Control
Ada.Tags
Ada.Task_Identification
```

The following which had pragma Preelaborate in Ada 1995 have been promoted to pragma Pure in Ada 2005

```
Ada.Characters.Handling
Ada.Strings.Maps
Ada.Strings.Maps.Constants
System
System.Storage_Elements
```

These changes mean that certain facilities such as the ability to analyse exceptions are now available to preelaborable units. Note however, that Wide_Maps and Wide_Maps.Wide_Constants stay as preelaborable because they may be implemented using access types.

Just for the record the following packages (and functions, Hash is a function) which are new to Ada 2005 have the pragma Pure

Ada.Assertions
Ada.Characters.Conversions
Ada.Containers
Ada.Containers.Generic_Array_Sort
Ada.Containers.Generic_Constrained_Array_Sort
Ada.Dispatching
Ada.Numerics.Generic_Real_Arrays
Ada.Numerics.Generic_Complex_Arrays
Ada.Strings.Hash

And the following new packages and functions have the pragma Preelaborate

Ada.Containers.Doubly_Linked_Lists
Ada.Containers.Hashed_Maps
Ada.Containers.Hashed_Sets
Ada.Containers.Ordered_Maps
Ada.Containers.Ordered_Sets
Ada.Containers.Vectors
Ada.Environment_Variables
Ada.Strings.Unbounded_Hash
Ada.Strings.Wide_Wide_Maps
Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants
Ada.Tags.Generic_Dispatching_Constructor
Ada.Task_Termination

plus the indefinite containers as well.

A problem with preelaborable units in Ada 95 is that there are restrictions on declaring default initialized objects in a unit with the pragma Preelaborate. For example, we cannot declare objects of a private type at the library level in such a unit. This is foolish for consider

```
package P is
  pragma Preelaborate(P);
  X: Integer := 7;
  B: Boolean := True;
end;
```

Clearly these declarations can be preelaborated and so the package P can have the pragma Preelaborate. However, now consider

```
package Q is
  pragma Preelaborate(Q);                    -- legal
  type T is private;
private
  type T is
    record
      X: Integer := 7;
      B: Boolean := True;
```

```
    end record;
end Q;

with Q;
package P is
  pragma Preelaborate(P);                    -- illegal
  Obj: Q.T;
end P;
```

The package Q is preelaborable because it does not declare any objects. However, the package P is not preelaborable because it declares an object of the private type T – the theory being of course that since the type is private we do not know that its default initial value is static.

This is overcome in Ada 2005 by the introduction of the pragma Preelaborable_Initialization. Its syntax is

```
pragma Preelaborable_Initialization(direct_name);
```

We can now write

```
package Q is
  pragma Preelaborate(Q);
  type T is private;
  pragma Preelaborable_Initialization(T);
private
  type T is
    record
      X: Integer := 7;
      B: Boolean := True;
    end record;
end Q;
```

The pragma promises that the full type will have preelaborable initialization and the declaration of the package P above is now legal.

The following predefined private types which existed in Ada 95 have the pragma Preelaborable_Initialization in Ada 2005

```
Ada.Exceptions.Exception_Id
Ada.Exceptions.Exception_Occurrence
Ada.Finalization.Controlled
Ada.Finalization.Limited_Controlled
Ada.Numerics.Generic_Complex_Types.Imaginary
Ada.Streams.Root_Stream_Type
Ada.Strings.Maps.Character_Mapping
Ada.Strings.Maps.Character_Set
Ada.Strings.Unbounded.Unbounded_String
Ada.Tags.Tag
Ada.Task_Identification.Task_Id
Interfaces.C.Strings.chars_ptr
System.Address
System.Storage_Pool.Root_Storage_Pool
```

Wide and wide-wide versions also have the pragma as appropriate. Note that it was not possible to apply the pragma to Ada.Strings.Bounded.Generic_Bounded_Length.Bounded_String because it would have made it impossible to instantiate Generic_Bounded_Length with a non-static expression for the parameter Max.

The following private types which are new in Ada 2005 also have the pragma Preeleborable_Initialization

```
Ada.Containers.Vectors.Vector
Ada.Containers.Vectors.Cursor
Ada.Containers.Doubly_Linked_Lists.List
Ada.Containers.Doubly_Linked_Lists.Cursor
Ada.Containers.Hashed_Maps.Map
Ada.Containers.Hashed_Maps.Cursor
Ada.Containers.Ordered_Maps.Map
Ada.Containers.Ordered_Maps.Cursor
Ada.Containers.Hashed_Sets.Set
Ada.Containers.Hashed_Sets.Cursor
Ada.Containers.Ordered_Sets.Set
Ada.Containers.Ordered_Sets.Cursor
```

and similarly for the indefinite containers.

A related change concerns the definition of pure units. In Ada 2005, pure units can now use access to subprogram and access to object types provided that no storage pool is created.

Finally, we mention a small but important change regarding the partition communication subsystem System.RPC. Implementations conforming to the Distributed Systems annex are not required to support this predefined interface if another interface would be more appropriate – to interact with CORBA for example.

8 Streams

Important improvements to the control of streams were described in the paper on the object oriented model where we discussed the new package Ada.Tags.Generic_Dispatching_Constructor and various changes to the parent package Ada.Tags itself. In this section we mention two other changes.

There is a problem with the existing stream attributes such as (see RM 13.13.2)

```
procedure S'Write(Stream: access Root_Stream_Type'Class; Item: in T);
```

where S is a subtype of T. Note that for the parameter Item, its type *T* is in italic and so has to be interpreted according to the kind of type. In the case of integer and enumeration types it means that the parameter Item has type T'Base.

Given a declaration such as

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

different implementations might use different representations for Index'Base – some might use 8 bits others might use 32 bits and so on.

Now stream elements themselves are typically 8 bits and so with an 8-bit base, there will be one value of Index per stream element whereas with a 32-bit base each value of Index will take 4 stream elements. Clearly a stream written by the 8-bit implementation cannot be read by the 32-bit one.

This problem is overcome in Ada 2005 by the introduction of a new attribute Stream_Size. The universal integer value S'Stream_Size gives the number of bits used in the stream for values of the subtype S. We are guaranteed that it is a multiple of Stream_Element'Size. So the number of stream elements required will be

```
S'Stream_Size / Stream_Element'Size
```

We can set the attribute in the usual way provided that the value given is a static multiple of Stream_Element'Size. So in the case above we can write

```
for Index'Stream_Size use 8;
```

and portability is then assured. That is provided that Stream_Element_Size is 8 anyway and that the implementation accepts the attribute definition clause (which it should).

A minor change is that the parameter Stream of the various atttributes now has a null exclusion so that S'Write is in fact

```
procedure S'Write(Stream: not null access Root_Stream_Type'Class; Item: in T);
```

Perhaps surprisingly this does not introduce any incompatibilities since in Ada 95 passing null raises Constraint_Error anyway and so this change just clarifies the situation.

On this dullish but important topic here endeth the Rationale for Ada 2005 apart from various exciting appendices and an extensive subpaper on containers.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] ISO/IEC 13813:1997 (1997) Generic packages of real and complex type declarations and basic operations for Ada (including vector and matrix types).

[3] *Ada 95 Rationale* (1995) LNCS 1247, Springer-Verlag.

[4] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.

Rationale for Ada 2005: 6a Containers

John Barnes
John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes the predefined container library in Ada 2005.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 4, December 2005. Other papers in this series will be found in other issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Organization of containers

A major enhancement to the predefined library in Ada 2005 is the addition of a container library. This is quite extensive and merits this separate paper on its own. Other aspects of the predefined library and the overall rationale for extending the library were described in the previous paper.

The main packages in the container library can be grouped in various ways. One set of packages concerns the manipulation of objects of definite types and another, essentially identical, set concerns indefinite types. (Remember that an indefinite (sub)type is one for which we cannot declare an object without giving a constraint.) The reason for the duplication concerns efficiency. It is much easier to manipulate definite types and although the packages for indefinite types can be used for definite types, this would be rather inefficient.

We will generally only consider the definite packages. These in turn comprise two groups.

Sequence containers – these hold sequences of elements. There are packages for manipulating vectors and for manipulating linked lists. These two packages have much in common. But they have different behaviours in terms of efficiency according to the pattern of use. In general (with some planning) it should be possible to change from one to the other with little effort.

Associative containers – these associate a key with each element and then store the elements in order of the keys. There are packages for manipulating hashed maps, ordered maps, hashed sets and ordered sets. These four packages also have much in common and changing between hashed and ordered versions is usually feasible.

There are also quite separate generic procedures for sorting arrays which we will consider later.

The root package is

```
package Ada.Containers is
   pragma Pure(Containers);

   type Hash_Type is mod implementation-defined;
   type Count_Type is range 0 .. implementation-defined;

end Ada.Containers;
```

The type Hash_Type is used by the associative containers and Count_Type is used by both kinds of containers typically for the number of elements in a container. Note that we talk about elements in a

container rather than the components in a container – components is the Ada term for the items of an array or record as an Ada type and it is convenient to use a different term since in the case of containers the actual data structure is hidden.

Worst-case and average-case time complexity bounds are given using the familiar *O*(...) notation. This encourages implementations to use techniques that scale reasonably well and avoid junk algorithms such as bubble sort.

Perhaps a remark about using containers from a multitasking program would be helpful. The general rule is given in paragraph 3 of Annex A which says "The implementation shall ensure that each language defined subprogram is reentrant in the sense that concurrent calls on the same subprogram perform as specified, so long as all parameters that could be passed by reference denote nonoverlapping objects." So in other words we have to protect ourselves by using the normal techniques such as protected objects when container operations are invoked concurrently on the same object from multiple tasks even if the operations are only reading from the container.

2 Lists and vectors

We will first consider the list container since in some ways it is the simplest. Here is its specification interspersed with some explanation

```
generic
   type Element_Type is private;
   with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
   pragma Preelaborate(Doubly_Linked_Lists);

   type List is tagged private;
   pragma Preelaborable_Initialization(List);
   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);
   Empty_List: constant List;
   No_Element: constant Cursor;
```

The two generic parameters are the type of the elements in the list and the definition of equality for comparing elements. This equality relation must be such that x = y and y = x always have the same value.

A list container is an object of the type List. It is tagged since it will inevitably be implemented as a controlled type. The fact that it is visibly tagged means that all the advantages of object oriented programming are available. For one thing it enables the use of the prefixed notation so that we can write operations such as

```
My_List.Append(Some_Value);
```

rather than

```
Append(My_List, Some_Value);
```

The type Cursor is an important concept. It provides the means of access to individual elements in the container. Not only does it contain a reference to an element but it also identifies the container as well. This enables various checks to be made to ensure that we don't accidentally meddle with an element in the wrong container.

The constants Empty_List and No_Element are as expected and also provide default values for objects of types List and Cursor respectively.

```
function "=" (Left, Right: List) return Boolean;
function Length(Container: List) return Count_Type;
function Is_Empty(Container: List) return Boolean;
procedure Clear(Container: in out List);
```

The function "=" compares two lists. It only returns true if both lists have the same number of elements and corresponding elements have the same value as determined by the generic parameter "=" for comparing elements. The subprograms Length, Is_Empty and Clear are as expected.

Note that A_List = Empty_List, Is_Empty(A_List) and Length(A_List) = 0 all have the same value.

```
function Element(Position: Cursor) return Element_Type;

procedure Replace_Element(Container: in out List; Position: in Cursor;
                          New_Item: in Element_Type);
```

These are the first operations we have met that use a cursor. The function Element takes a cursor and returns the value of the corresponding element (remember that a cursor identifies the list as well as the element itself). The procedure Replace_Element replaces the value of the element identified by the cursor by the value given; it makes a copy of course.

Note carefully that Replace_Element has both the list and cursor as parameters. There are two reasons for this concerning correctness. One is to enable a check that the cursor does indeed identify an element in the given list. The other is to ensure that we do have write access to the container (the parameter has mode **in out**). Otherwise it would be possible to modify a container even though we only had a constant view of it. So as a general principle any operation that modifies a container must have the container as a parameter whereas an operation that only reads it such as the function Element does not.

```
procedure Query_Element(Position: in Cursor;
                        Process: not null access procedure (Element: in Element_Type));

procedure Update_Element(Container: in out List; Position: in Cursor;
                         Process: not null access procedure (Element: in out Element_Type));
```

These procedures provide *in situ* access to an element. One parameter is the cursor identifying the element and another is an access to a procedure to be called with that element as parameter. In the case of Query_Element, we can only read the element whereas in the case of Update_Element we can change it as well since the parameter mode of the access procedure is **in out**. Note that Update_Element also has the container as a parameter for reasons just mentioned when discussing Replace_Element.

The reader might wonder whether there is any difference between calling the function Element to obtain the current value of an element and using the seemingly elaborate mechanism of Query_Element. The answer is that the function Element makes a copy of the value whereas Query_Element gives access to the value without making a copy. (And similarly for Replace_Element and Update_Element.) This wouldn't matter for a simple list of integers but it would matter if the elements were large or of a controlled type (maybe even lists themselves).

```
procedure Move(Target, Source: in out List);
```

This moves the list from the source to the target after first clearing the target. It does not make copies of the elements so that after the operation the source is empty and Length(Source) is zero.

```
procedure Insert(Container: in out List;
                 Before: in Cursor;
                 New_Item: in Element_Type;
                 Count: in Count_Type := 1);
```

```
procedure Insert(Container: in out List;
                 Before: in Cursor;
                 New_Item: in Element_Type;
                 Position: out Cursor;
                 Count: in Count_Type := 1);

procedure Insert(Container: in out List;
                 Before: in Cursor;
                 Position: out Cursor;
                 Count: in Count_Type := 1);
```

These three procedures enable one or more identical elements to be added anywhere in a list. The place is indicated by the parameter Before – if this is No_Element, then the new elements are added at the end. The second procedure is similar to the first but also returns a cursor to the first of the added elements. The third is like the second but the new elements take their default values. Note the default value of one for the number of elements.

```
procedure Prepend(Container: in out List;
                  New_Item: in Element_Type;
                  Count: in Count_Type := 1);

procedure Append(Container: in out List;
                 New_Item: in Element_Type;
                 Count: in Count_Type := 1);
```

These add one or more new elements at the beginning or end of a list respectively. Clearly these operations can be done using Insert but they are sufficiently commonly needed that it is convenient to provide them specially.

```
procedure Delete(Container: in out List;
                 Position: in out Cursor;
                 Count: in Count_Type := 1);

procedure Delete_First(Container: in out List; Count: in Count_Type := 1);

procedure Delete_Last(Container: in out List; Count: in Count_Type := 1);
```

These delete one or more elements at the appropriate position. In the case of Delete, the parameter Position is set to No_Element upon return. If there are not as many as Count elements to be deleted at the appropriate place then it just deletes as many as possible (this clearly results in the container becoming empty in the case of Delete_First and Delete_Last).

```
procedure Reverse_Elements(Container: in out List);
```

This does the obvious thing. It would have been nice to call this procedure Reverse but sadly that is a reserved word.

```
procedure Swap(Container: in out List; I, J: in Cursor);
```

This handy procedure swaps the values in the two elements denoted by the two cursors. The elements must be in the given container otherwise Program_Error is raised. Note that the cursors do not change.

```
procedure Swap_Links(Container: in out List; I, J: in Cursor);
```

This performs the low level operation of swapping the links rather than the values which can be much faster if the elements are large. There is no analogy in the vectors package.

```
procedure Splice(Target: in out List;
                 Before: in Cursor;
                 Source in out List);

procedure Splice(Target: in out List;
                 Before: in Cursor;
                 Source: in out List;
                 Position: in out Cursor);

procedure Splice(Container: in out List;
                 Before: in Cursor;
                 Position: in out Cursor);
```

These three procedures enable elements to be moved (without copying). The place is indicated by the parameter Before – if this is No_Element, then the elements are added at the end. The first moves all the elements of Source into Target at the position given by Before; as a consequence, like the procedure Move, after the operation the source is empty and Length(Source) is zero. The second moves a single element at Position from the list Source to Target and so the length of target is incremented whereas that of source is decremented; Position is updated to its new location in Target. The third moves a single element within a list and so the length remains the same (note the formal parameter is Container rather than Target in this case). There are no corresponding operations in the vectors package because, like Swap_Links, we are just moving the links and not copying the elements.

```
function First(Container: List) return Cursor;
function First_Element(Container: List) return Element_Type;
function Last(Container: List) return Cursor;
function Last_Element(Container: List) return Element_Type;
function Next(Position: Cursor) return Cursor;
function Previous(Position: Cursor) return Cursor;
procedure Next(Position: in out Cursor);
procedure Previous(Position: in out Cursor);

function Find(Container: List;
              Item: Element_Type;
              Position: Cursor:= No_Element) return Cursor;

function Reverse_Find(Container: List;
              Item: Element_Type;
              Position: Cursor:= No_Element) return Cursor;

function Contains(Container: List; Item: Element_Type) return Boolean;
```

Hopefully the purpose of these is almost self-evident. The function Find searches for an element with the given value starting at the given cursor position (or at the beginning if the position is No_Element); if no element is found then it returns No_Element. Reverse_Find does the same but backwards. Note that equality used for the comparison in Find and Reverse_Find is that defined by the generic parameter "=".

```
function Has_Element(Position: Cursor) return Boolean;
```

This returns False if the cursor does not identify an element; for example if it is No_Element.

```
procedure Iterate(Container: in List;
                  Process: not null access procedure (Position: in Cursor));

procedure Reverse_Iterate(Container: in List;
                  Process: not null access procedure (Position: in Cursor));
```

These apply the procedure designated by the parameter Process to each element of the container in turn in the appropriate order.

```
generic
  with function "<" (Left, Right: Element_Type) return Boolean is <>;
package Generic_Sorting is
  function Is_Sorted(Container: List) return Boolean;
  procedure Sort(Container: in out List);
  procedure Merge(Target, Source: in out List);
end Generic_Sorting;
```

This generic package performs sort and merge operations using the order specified by the generic formal parameter. Note that we use generics rather than access to subprogram parameters when the formal process is given by an operator. This is because the predefined operations have convention Intrinsic and one cannot pass an intrinsic operation as an access to subprogram parameter. The function Is_Sorted returns True if the container is already sorted. The procedure Sort arranges the elements into order as necessary – note that no copying is involved since it is only the links that are moved. The procedure Merge takes the elements from Source and adds them to Target. After the merge Length(Source) is zero. If both lists were sorted before the merge then the result is also sorted.

And finally we have

```
private
  ... -- not specified by the language
end Ada.Containers.Doubly_Linked_Lists;
```

If the reader has got this far they have probably understood how to use this package so extensive examples are unnecessary. However, as a taste, here is a simple stack of floating point numbers

```
package Stack is
  procedure Push(X: in Float);
  function Pop return Float;
  function Size return Integer;
  exception Stack_Empty;
end;

with Ada.Containers.Doubly_Linked_Lists;
use Ada.Containers;
package body Stack is

  package Float_Container is new Doubly_Linked_Lists(Float);
  use Float_Container;
  The_Stack: List;

  procedure Push(X: in Float) is
  begin
    Append(The_Stack, X);                    -- or The_Stack.Append(X);
  end Push;

  function Pop return Float is
    Result: Float;
  begin
    if Is_Empty(The_Stack) then
      raise Stack_Empty;
    end if;
    Result := Last_Element(The_Stack);
```

```
      Delete_Last(The_Stack);
      return Result;
    end Pop;

    function Size return Integer is
    begin
      return Integer(Length(The_Stack));
    end Size;

  end Stack;
```

This barely needs any explanation. The lists package is instantiated in the package Stack and the object The_Stack is of course the list container. The rest is really straightforward. We could of course use the prefixed notation throughout as indicated in Push.

An important point should be mentioned concerning lists (and containers in general). This is that attempts to do foolish things typically result in Constraint_Error or Program_Error being raised. This especially applies to the procedures Process in Query_Element, Update_Element, Iterate and Reverse_Iterate. The concepts of tampering with cursors and elements are introduced in order to dignify a general motto of "Thou shalt not violate thy container".

Tampering with cursors occurs when elements are added to or deleted from a container (by calling Insert and so on) whereas tampering with elements means replacing an element (by calling Replace_Element for example). Tampering with elements is a greater sin and includes tampering with cursors. The procedure Process in Query_Element and Update_Element must not tamper with elements and the procedure Process in the other cases must not tamper with cursors. The reader might think it rather odd that Update_Element should not be allowed to tamper with elements since the whole purpose is to update the element; this comes back to the point mentioned earlier that update element gives access to the existing element *in situ* via the parameter of Process and that is allowed – calling Replace_Element within Process would be tampering. Tampering causes Program_Error to be raised.

We will now consider the vectors package. Its specification starts

```
generic
  type Index_Type is range <>;
  type Element_Type is private;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Vectors is
  pragma Preelaborate(Vectors);
```

This is similar to the lists package except for the additional generic parameter Index_Type (note that this is an integer type and not a discrete type). This additional parameter reflects the idea that a vector is essentially an array and we can index directly into an array.

In fact the vectors package enables us to access elements either by using an index or by using a cursor. Thus many operations are duplicated such as

```
function Element(Container: Vector; Index: Index_Type) return Element_Type;
function Element(Position: Cursor) return Element_Type;

procedure Replace_Element(Container: in out Vector;
                          Index: in Index_Type;
                          New_Item: in Element_Type);

procedure Replace_Element(Container: in out Vector;
                          Position: in Cursor;
                          New_Item: in Element_Type);
```

If we use an index then there is always a distinct parameter identifying the vector as well. If we use a cursor then the vector parameter is omitted if the vector is unchanged as is the case with the function Element. Remember that we stated earlier that a cursor identifies both an element and the container but if the container is being changed as in the case of Replace_Element then the container has to be passed as well to ensure write access and to enable a check that the cursor does identify an element in the correct container.

There are also functions First_Index and Last_Index thus

```
function First_Index(Container: Vector) return Index_Type;

function Last_Index(Container: Vector) return Extended_Index;
```

These return the values of the index of the first and last elements respectively. The function First_Index always returns Index_Type'First whereas Last_Index will return No_Index if the vector is empty. The function Length returns Last_Index–First_Index+1 which is zero if the vector is empty. Note that the irritating subtype Extended_Index has to be introduced in order to cope with end values. The constant No_Index has the value Extended_Index'First which is equal to Index_Type'First–1.

There are operations to convert between an index and a cursor thus

```
function To_Cursor(Container: Vector; Index: Extended_Index) return Cursor;

function To_Index(Position: Cursor) return Extended_Index;
```

It is perhaps slightly messier to use the index and vector parameters because of questions concerning the range of values of the index but probably slightly faster and maybe more familiar. And sometimes of course using an index is the whole essence of the problem. In the paper on access types we showed a use of the procedure Update_Element to double the values of those elements of a vector whose index was in the range 5 to 10. This would be tedious with cursors.

But an advantage of using cursors is that (provided certain operations are avoided) it is easy to replace the use of vectors by lists.

For example here is the stack package rewritten to use vectors

```
with Ada.Containers.Vectors;                          -- changed
use Ada.Containers;
package body Stack is

  package Float_Container is new Vectors(Natural, Float); -- changed
  use Float_Container;
  The_Stack: Vector;                                  -- changed

  procedure Push(X: in Float) is
  begin
    Append(The_Stack, X);
  end Push;

  -- etc exactly as before

end Stack;
```

So the changes are very few indeed and can be quickly done with a simple edit.

Note that the index parameter has been given as Natural rather than Integer. Using Integer will not work since attempting to elaborate the subtype Extended_Index would raise Constraint_Error when evaluating Integer'First–1. But in any event it is more natural for the index range of the container to start at 0 (or 1) rather than a large negative value such as Integer'First.

There are other important properties of vectors that should be mentioned. One is that there is a concept of capacity. Vectors are adjustable and will extend if necessary when new items are added. However, this might lead to lots of extensions and copying and so we can set the capacity of a container by calling

```
procedure Reserve_Capacity(Container: in out Vector; Capacity: in Count_Type);
```

There is also

```
function Capacity(Container: Vector) return Count_Type;
```

which naturally returns the current capacity. Note that Length(V) cannot exceed Capacity(V) but might be much less.

If we add items to a vector whose length and capacity are the same then no harm is done. The capacity will be expanded automatically by effectively calling Reserve_Capacity internally. So the user does not need to set the capacity although not doing so might result in poorer performance.

There is also the concept of "empty elements". These are elements whose values have not been set. There is no corresponding concept with lists. It is a bounded error to read an empty element. Empty elements arise if we declare a vector by calling

```
function To_Vector(Length: Count_Type) return Vector;
```

as in

```
My_Vector: Vector := To_Vector(100);
```

There is also the much safer

```
function To_Vector(New_Item: Element_Type; Length: Count_Type) return Vector;
```

which sets all the elements to the value New_Item.

There is also a procedure

```
procedure Set_Length(Container: in out Vector; Length: in Count_Type);
```

This changes the length of a vector. This may require elements to be deleted (from the end) or to be added (in which case the new ones are empty).

The final way to get an empty element is by calling one of

```
procedure Insert_Space(Container: in out Vector;
                       Before: in Extended_Index;
                       Count: in Count_Type := 1);

procedure Insert_Space(Container: in out Vector;
                       Before: in Cursor;
                       Position: out Cursor;
                       Count: in Count_Type := 1);
```

These insert the number of empty elements given by Count at the place indicated. Existing elements are slid along as necessary. These should not be confused with the versions of Insert which do not provide an explicit value for the elements – in those cases the new elements take their default values.

Care needs to be taken if we use empty elements. For example we should not compare two vectors using "=" if they have empty elements because this implies reading them. But the big advantage of empty elements is that they provide a quick way to make a large lump of space in a vector which can then be filled in with appropriate values. One big slide is a lot faster than lots of little ones.

For completeness, we briefly mention the remaining few subprograms that are unique to the vectors package.

There are further versions of Insert thus

```
procedure Insert(Container: in out Vector;
                 Before: in Extended_Index; New_Item: in Vector);

procedure Insert(Container: in out Vector;
                 Before: in Cursor; New_Item: in Vector);

procedure Insert(Container: in out Vector;
                 Before: in Cursor; New_Item: in Vector; Position: out Cursor);
```

These insert copies of a vector into another vector (rather than just single elements).

There are also corresponding versions of Prepend and Append thus

```
procedure Prepend(Container: in out Vector; New_Item: in Vector);

procedure Append(Container: in out Vector; New_Item: in Vector);
```

Finally, there are four functions "&" which concatenate vectors and elements by analogy with those for the type String. Their specifications are

```
function "&" (Left, Right: Vector) return Vector;
function "&" (Left: Vector; Right: Element_Type) return Vector;
function "&" (Left: Element_Type; Right: Vector) return Vector;
function "&" (Left, Right: Element_Type) return Vector;
```

Note the similarity between

```
Append(V1, V2);
V1 := V1 & V2;
```

The result is the same but using "&" is less efficient because of the extra copying involved. But "&" is a familiar operation and so is provided for convenience.

3 Maps

We will now turn to the maps and sets packages. We will start by considering maps which are more exciting than sets and begin with ordered maps which are a little simpler and then consider hashed maps.

Remember that a map is just a means of getting from a value of one type (the key) to another type (the element). This is not a one-one relationship. Given a key there is a unique element (if any), but several keys may correspond to the same element. A simple example is an array. This is a map from the index type to the component type. Thus if we have

```
S: String := "animal";
```

then this provides a map from integers in the range 1 to 6 to some values of the type Character. Given an integer such as 3 there is a unique character 'i' but given a character such as 'a' there might be several corresponding integers (in this case both 1 and 5).

More interesting examples are where the set of used key values is quite sparse. For example we might have a store where various spare parts are held. The parts have a five-digit part number and there are perhaps twenty racks where they are held identified by a letter. However, only a handful of the five digit numbers are in use so it would be very wasteful to use an array with the part number as index. What we want instead is a container which holds just the pairs that matter such as (34618,

'F'), (27134, 'C') and so on. We can do this using a map. We usually refer to the pairs of values as nodes of the map.

There are two maps packages with much in common. One keeps the keys in order and the other uses a hash function. Here is the specification of the ordered maps package generally showing just those facilities common to both.

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function "<" (Left, Right: Key_Type) return Boolean is <>;
   with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Maps is
   pragma Preelaborate(Ordered_Maps);

   function Equivalent_Keys(Left: Right: Key_Type) return Boolean;
```

The generic parameters include the ordering relationship "<" on the keys and equality for the elements.

It is assumed that the ordering relationship is well behaved in the sense that if x < y is true then y < x is false. We say that two keys x and y are equivalent if both x < y and y < x are false. In other words this defines an equivalence class on keys. The relationship must also be transitive, that is, if x < y and y < z are both true then x < z must also be true.

This concept of an equivalence relationship occurs throughout the various maps and sets. Sometimes, as here, it is defined in terms of an order but in other cases, as we shall see, it is defined by an equivalence function.

It is absolutely vital that the equivalence relations are defined properly and meet the above requirements. It is not possible for the container packages to check this and if the operations are wrong then peculiar behaviour is almost inevitable.

For the convenience of the user the function Equivalent_Keys is declared explicitly. It is equivalent to

```
function Equivalent_Keys(Left, Right: Key_Type) return Boolean is
begin
   return not (Left < Right) and not (Right < Left);
end Equivalent_Keys;
```

The equality operation on elements is not so demanding. It must be symmetric so that x = y and y = x are the same but transitivity is not required (although cases where it would not automatically be transitive are likely to be rare). The operation is only used for the function "=" on the containers as a whole.

Note that Find and similar operations for maps and sets work in terms of the equivalence relationship rather than equality as was the case with lists and vectors.

```
   type Map is tagged private;
   pragma Preelaborable_Initialization(Map);
   type Cursor is private;
   pragma Preelaborable_Initialization(Cursor);
   Empty_Map: constant Map;
   No_Element: constant Cursor;
```

The types Map and Cursor and constants Empty_Map and No_Element are similar to the corresponding entities in the lists and vectors containers.

```
function "=" (Left, Right: Map) return Boolean;
function Length(Container: Map) return Count_Type;
function Is_Empty(Container: Map) return Boolean;
procedure Clear(Container: in out Map);
```

These are again similar to the corresponding entities for lists. Note that two maps are said to be equal if they have the same number of nodes with equivalent keys (as defined by "<") whose corresponding elements are equal (as defined by "=").

```
function Key(Position: Cursor) return Key_Type;
function Element(Position: Cursor) return Element_Type;

procedure Replace_Element(Container: in out Map;
                          Position: in Cursor;
                          New_Item: in Element_Type);

procedure Query_Element(Position: in Cursor;
   Process: not null access procedure (Key: in Key_Type; Element: in Element_Type));

procedure Update_Element(Container: in out Map; Position: in Cursor;
   Process: not null access procedure (Key: in Key_Type; Element: in out Element_Type));
```

In this case there is a function Key as well as a function Element. But there is no procedure Replace_Key since it would not make sense to change a key without changing the element as well and this really comes down to deleting the whole node and then inserting a new one.

The procedures Query_Element and Update_Element are slightly different in that the procedure Process also takes the key as parameter as well as the element to be read or updated. Note again that the key cannot be changed. Nevertheless the value of the key is given since it might be useful in deciding how the update should be performed. Remember that we cannot get uniquely from an element to a key but only from a key to an element.

```
procedure Move(Target, Source: in out Map);
```

This moves the map from the source to the target after first clearing the target. It does not make copies of the nodes so that after the operation the source is empty and Length(Source) is zero.

```
procedure Insert(Container: in out Map;
                 Key: in Key_Type;
                 New_Item: in Element_Type;
                 Position: out Cursor;
                 Inserted: out Boolean);

procedure Insert(Container: in out Map;
                 Key: in Key_Type;
                 Position: out Cursor;
                 Inserted: out Boolean);

procedure Insert(Container: in out Map;
                 Key: in Key_Type;
                 New_Item: in Element_Type);
```

These insert a new node into the map unless a node with an equivalent key already exists. If it does exist then the first two return with Inserted set to False and Position indicating the node whereas the third raises Constraint_Error (the element value is not changed). If a node with equivalent key is not found then a new node is created with the given key, the element value is set to New_Item when that is given and otherwise it takes its default value (if any), and Position is set when given.

Unlike vectors and lists, we do not have to say where the new node is to be inserted because of course this is an ordered map and it just goes in the correct place according to the order given by the generic parameter "<".

```
procedure Include(Container: in out Map;
                  Key: in Key_Type;
                  New_Item: in Element_Type);
```

This is somewhat like the last Insert except that if an existing node with an equivalent key is found then it is replaced (rather than raising Constraint_Error). Note that both the key and the element are updated. This is because equivalent keys might not be totally equal.

For example the key part might be a record with part number and year of introduction, thus

```
type Part_Key is
  record
    Part_Number: Integer;
    Year: Integer;
  end record;
```

and we might define the ordering relationship to be used as the generic parameter simply in terms of the part number

```
function "<" (Left, Right: Part_Key) return Boolean is
begin
  return Left.Part_Number < Right.Part_Number;
end "<";
```

In this situation, the keys could match without the year component being the same and so it would need to be updated. In other words with this definition of the ordering relation, two keys are equivalent provided just the part numbers are the same.

```
procedure Replace(Container: in out Map;
                  Key: in Key_Type;
                  New_Item: in Element_Type);
```

In this case, Constraint_Error is raised if the node does not already exist. On replacement both the key and the element are updated as for Include.

Perhaps a better example of equivalent keys not being totally equal is if the key were a string. We might decide that the case of letter did not need to match in the test for equivalence but nevertheless we would probably want to update with the string as used in the parameter of Replace.

```
procedure Exclude(Container: in out Map; Key: in Key_Type);
```

If there is a node with an equivalent key then it is deleted. If there is not then nothing happens.

```
procedure Delete(Container: in out Map; Key: in Key_Type);

procedure Delete(Container: in out Map; Position: in out Cursor);
```

These delete a node. In the first case if there is no such equivalent key then Constraint_Error is raised (by contrast to Exclude which remains silent in this case). In the second case if the cursor is No_Element then again Constraint_Error is raised – there is also a check to ensure that the cursor otherwise does designate a node in the correct map (remember that cursors designate both an entity and the container); if this check fails then Program_Error is raised.

Perhaps it is worth observing that Insert, Include, Replace, Exclude and Delete form a sort of progression from an operation that will insert something, through operations that might insert, will

neither insert nor delete, might delete, to the final operation that will delete something. Note also that Include, Replace and Exclude do not apply to lists and vectors.

```
function First(Container: Map) return Cursor;
function Last(Container: Map) return Cursor;
function Next(Position: Cursor) return Cursor;
procedure Next(Position: in out Cursor);
function Find(Container: Map; Key: Key_Type) return Cursor;
function Element(Container: Map; Key: Key_Type) return Element;
function Contains(Container: Map; Key: Key_Type) return Boolean;
```

These should be self-evident. Unlike the operations on vectors and lists, Find logically searches the whole map and not just starting at some point (and since it searches the whole map there is no point in having Reverse_Find). (In implementation terms it won't actually search the whole map because it will be structured in a way that makes this unnecessary – as a balanced tree perhaps.) Moreover, Find uses the equivalence relation based on the "<" parameter so in the example it only has to match the part number and not the year. The function call Element(My_Map, My_Key) is equivalent to Element(Find(My_Map, My_Key)).

```
function Has_Element(Position: Cursor) return Boolean;

procedure Iterate(Container: in Map;
                  Process: not null access procedure (Position: in Cursor));
```

These are also as for other containers.

And at last we have

```
private
   ... -- not specified by the language
end Ada.Containers.Ordered_Maps;
```

We have omitted to mention quite a few operations that have no equivalent in hashed maps – we will come back to these in a moment.

As an example we can make a container to hold the information concerning spare parts. We can use the type Part_Key and the function "<" as above. We can suppose that the element type is

```
type Stock_Info is
   record
      Shelf: Character range 'A' .. 'T';
      Stock: Integer;
   end record;
```

This gives both the shelf letter and the number in stock.

We can then declare the container thus

```
package Store_Maps is
    new Ordered_Maps(Key_Type => Part_Key,
                     Element_Type => Stock_Info,
                     "<" => "<");

The_Store: Store_Maps.Map;
```

The last parameter could be omitted because the formal has a <> default.

We can now add items to our store by calling

```
The_Store.Insert((34618, 1998), ('F', 25));
The_Store.Insert((27134, 2004), ('C', 45));
...
```

We might now have a procedure which, given a part number, checks to see if it exists and that the stock is not zero, and if so returns the shelf letter and year number and decrements the stock count.

```
procedure Request(Part: in Integer; OK: out Boolean;
                  Year: out Integer; Shelf: out Character) is
  C: Cursor;
  K: Part_Key;
  E: Stock_Info;
begin
  C := The_Store.Find((Part, 0));
  if C = No_Element then
    OK := False; return;                    -- no such key
  end if;
  E := Element(C);  K := Key(C);
  Year := K.Year;  Shelf := E.Shelf;
  if E.Stock = 0 then
    OK := False; return;                    -- out of stock
  end if;
  Replace_Element(C, (Shelf, E.Stock–1));
  OK := True;
end Request;
```

Note that we had to put a dummy year number in the call of Find. We could of course use the new <> notation for this

```
C := The_Store.Find((Part, others => <>));
```

The reader can improve this example at leisure – by using Update_Element for example.

As another example suppose we wish to check all through the stock looking for parts whose stock is low, perhaps less than some given parameter. We can use Iterate for this as follows

```
procedure Check_Stock(Low: in Integer) is

  procedure Check_It(C: in Cursor) is
  begin
    if Element(C).Stock < Low then
      -- print a message perhaps
      Put("Low stock of part ");
      Put_Line(Key(C).Part_Number);
    end if;
  end Check_It;

begin
  The_Store.Iterate(Check_It'Access);
end Check_Stock;
```

Note that this uses a so-called downward closure. The procedure Check_It has to be declared locally to Check_Stock in order to access the parameter Low. (Well you could declare it outside and copy the parameter Low to a global variable but that is just the sort of wicked thing one has to do in lesser languages (such as even Ada 95). It is not task safe for one thing.)

Another approach is to use First and Next and so on thus

```
procedure Check_Stock(Low: in Integer) is
  C: Cursor := The_Store.First;
begin
  loop
    exit when C = No_Element;
    if Element(C).Stock < Low then
      -- print a message perhaps
      Put("Low stock of part ");
      Put_Line(Key(C).Part_Number);
    end if;
    C := The_Store.Next(C);
  end loop;
end Check_Stock;
```

We will now consider hashed maps. The trouble with ordered maps in general is that searching can be slow when the map has many entries. Techniques such as a binary tree can be used but even so the search time will increase at least as the logarithm of the number of entries. A better approach is to use a hash function. This will be familiar to many readers (especially those who have written compilers). The general idea is as follows.

We define a function which takes a key and returns some value in a given range. In the case of the Ada containers it has to return a value of the modular type Hash_Type which is declared in the root package Ada.Containers. We could then convert this value onto a range representing an index into an array whose size corresponds to the capacity of the map. This index value is the preferred place to store the entry. If there already is an entry at this place (because some other key has hashed to the same value) then a number of approaches are possible. One way is to create a list of entries with the same index value (often called buckets); another way is simply to put it in the next available slot. The details don't matter. But the overall effect is that provided the map is not too full and the hash function is good then we can find an entry almost immediately more or less irrespective of the size of the map.

So as users all we have to do is to define a suitable hash function. It should give a good spread of values across the range of Hash_Type for the population of keys, it should avoid clustering and above all for a given key it must *always* return the same hash value. A good discussion on hash functions by Knuth will be found in [1].

Defining good hash functions needs care. In the case of the part numbers we might multiply the part number by some obscure prime number and then truncate the result down to the modular type Hash_Type. The author hesitates to give an example but perhaps

```
function Part_Hash(P: Part_Key) return Hash_Type is
  M31: constant := 2**31–1;                -- a nice Mersenne prime
begin
  return Hash_Type(P.Part_Number) * M31;
end Part_Hash;
```

On reflection that's probably a very bad prime to use because it is so close to half of 2**32 a typical value of Hash_Type'Last+1. Of course it doesn't have to be prime but simply relatively prime to it such as 5**13. Knuth suggests dividing the range by the golden number $\tau = (\sqrt{5}+1)/2 = 1.618...$ and then taking the nearest number relatively prime which is in fact simply the nearest odd number (in this case it is 2654435769).

Here is a historic interlude. Marin Mersenne (1588-1648) was a Franciscan monk who lived in Paris. He studied numbers of the form $M_p = 2^p - 1$ where p is prime. A lot of these are themselves prime. Mersenne gave a list of those upto 257 which he said were prime (namely 2, 3, 5, 7, 13, 17, 19, 31,

67, 127, 257). It was not until 1947 that it was finally settled that he got some of them wrong (61, 89, and 107 are also prime but 67 and 257 are not). At the time of writing there are 42 known Mersenne primes and the largest which is also the largest known prime number is $M_{25964951}$ − see www.mersenne.org.

The specification of the hashed maps package is very similar to that for ordered maps. It starts

```
generic
   type Key_Type is private;
   type Element_Type is private;
   with function Hash(Key: Key_Type) return Hash_Type;
   with function Equivalent_Keys(Left, Right: Key_Type) return Boolean;
   with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Hashed_Maps is
   pragma Preelaborate(Hashed_Maps);
```

The differences from the ordered maps package are that there is an extra generic parameter Hash giving the hash function and the ordering parameter "<" has been replaced by the function Equivalent_Keys. It is this function that defines the equivalence relationship for hashed maps; it is important that Equivalent_Keys(X, Y) is always the same as Equivalent_Keys(Y, X). Moreover if X and Y are equivalent and Y and Z are equivalent then X and Z must also be equivalent.

Note that the function Equivalent_Keys in the ordered maps package discussed above corresponds to the formal generic parameter of the same name in this hashed maps package. This should make it easier to convert between the two forms of packages.

Returning to our example, if we now write

```
function Equivalent_Parts(Left, Right: Part_Key) return Boolean is
begin
   return Left.Part_Number = Right.Part_Number;
end Equivalent_Parts;
```

then we can instantiate the hashed maps package as follows

```
package Store_Maps is
    new Hashed_Maps(Key_Type => Part_Key,
                    Element_Type => Stock_Info,
                    Hash => Part_Hash,
                    Equivalent_Keys => Equivalent_Parts);

The_Store: Store_Maps.Map;
```

and then the rest of our example will be exactly as before. It is thus easy to convert from an ordered map to a hashed map and vice versa provided of course that we only use the facilities common to both.

We will finish this discussion of maps by briefly considering the additional facilities in the two packages.

The ordered maps package has the following additional subprograms

```
procedure Delete_First(Container: in out Map);
procedure Delete_Last(Container: in out Map);

function First_Element(Container: Map) return Element_Type;
function First_Key(Container: Map) return Key_Type;
function Last_Element(Container: Map) return Element_Type;
function Last_Key(Container: Map) return Key_Type;
```

```
function Previous(Position: Cursor) return Cursor;
procedure Previous(Position: in out Cursor);

function Floor(Container: Map; Key: Key_Type) return Cursor;
function Ceiling(Container: Map; Key: Key_Type) return Cursor;

function "<" (Left, Right: Cursor) return Boolean;
function ">" (Left, Right: Cursor) return Boolean;
function "<" (Left: Cursor; Right: Key_Type) return Boolean;
function ">" (Left: Cursor; Right: Key_Type) return Boolean;
function "<" (Left: Key_Type; Right: Cursor) return Boolean;
function ">" (Left: Key_Type; Right: Cursor) return Boolean;

procedure Reverse_Iterate(Container: in Map;
          Process: not null access procedure (Position: in Cursor));
```

These are again largely self-evident. The functions Floor and Ceiling are interesting. Floor searches for the last node whose key is not greater than Key and similarly Ceiling searches for the first node whose key is not less than Key – they return No_Element if there is no such element. The subprograms Previous are of course the opposite of Next and Reverse_Iterate is like Iterate only backwards.

The functions "<" and ">" are mostly for convenience. Thus the first is equivalent to

```
function "<" (Left, Right: Cursor) return Boolean is
begin
  return Key(Left) < Key(Right);
end "<";
```

Clearly these additional operations must be avoided if we wish to retain the option of converting to a hashed map later.

Hashed maps have a very important facility not in ordered maps which is the ability to specify a capacity as for the vectors package. (Underneath their skin the hashed maps are a bit like vectors whereas the ordered maps are a bit like lists.) Thus we have

```
procedure Reserve_Capacity(Container: in out Map; Capacity: in Count_Type);

function Capacity(Container: Map) return Count_Type;
```

The behaviour is much as for vectors. We don't have to set the capacity ourselves since it will be automatically extended as necessary but it might significantly improve performance to do so. In the case of maps, increasing the capacity requires the hashing to be redone which could be quite time consuming, so if we know that our map is going to be a big one, it is a good idea to set an appropriate capacity right from the beginning. Note again that Length(M) cannot exceed Capacity(M) but might be much less.

The other additional subprograms for hashed maps are

```
function Equivalent_Keys(Left, Right: Cursor) return Boolean;
function Equivalent_Keys(Left: Cursor; Right: Key_Type) return Boolean;
function Equivalent_Keys(Left: Key_Type; Right: Cursor) return Boolean;
```

These (like the additional "<" and ">" for ordered maps) are again mostly for convenience. The first is equivalent to

```
function Equivalent_Keys(Left, Right: Cursor) return Boolean is
begin
  return Equivalent_Keys(Key(Left), Key(Right));
end Equivalent_Keys;
```

Before moving on to sets it should be noticed that there are also some useful functions in the string packages. The main one is

```
with Ada.Containers;
function Ada.Strings.Hash(Key: String) return Containers.Hash_Type;
pragma Pure(Ada.Strings.Hash);
```

There is a similar function Ada.Strings.Unbounded.Hash where the parameter Key has type Unbounded_String. It simply converts the parameter to the type String and then calls Ada.Strings.Hash. There is also a generic function for bounded strings which again calls the basic function Ada.Strings.Hash. For completeness the function Ada.Strings.Fixed.Hash is a renaming of Ada.Strings.Hash.

These are provided because it is often the case that the key is a string and they save the user from devising good hash functions for strings which might cause a nasty headache.

We could for example save ourselves the worry of defining a good hash function in the above example by making the part number into a 5-character string. So we might write

```
function Part_Hash(P: Part_Key) return Hash_Type is
begin
  return Ada.Strings.Hash(P.Part_Number);
end Part_Hash;
```

and if this doesn't work well then we can blame the vendor.

4 Sets

Sets, like maps, come in two forms: hashed and ordered. Sets are of course just collections of values and there is no question of a key (we can perhaps think of the value as being its own key). Thus in the case of an ordered set the values are stored in order whereas in the case of a map, it is the keys that are stored in order. As well as the usual operations of inserting elements into a set and searching and so on, there are also many operations on sets as a whole that do not apply to the other containers – these are the familiar set operations such as union and intersection.

Here is the specification of the ordered sets package giving just those facilities that are common to both kinds of sets.

```
generic
  type Element_Type is private;
  with function "<" (Left, Right: Element_Type) return Boolean is <>;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets is
  pragma Preelaborate(Ordered_Sets);

  function Equivalent_Elements(Left, Right: Element_Type) return Boolean;

  type Set is tagged private;
  pragma Preelaborable_Initialization(Set);
  type Cursor is private;
  pragma Preelaborable_Initialization(Cursor);
  Empty_Set: constant Set;
  No_Element: constant Cursor;
```

The only differences from the maps package (apart from the identifiers) are that there is no key type and both "<" and "=" apply to the element type (whereas in the case of maps, the operation "<" applies to the key type). Thus the ordering relationship "<" defined on elements defines equivalence between the elements whereas "=" defines equality.

It is possible for two elements to be equivalent but not equal. For example if they were strings then we might decide that the ordering (and thus equivalence) ignored the case of letters but that equality should take the case into account. (They could also be equal but not equivalent but that is perhaps less likely.)

And as in the case of the maps package, the equality operation on elements is only used by the function "=" for comparing two sets.

Again we have the usual rules as explained for maps. Thus if x < y is true then y < x must be false; x < y and y < z must imply x < z; and x = y and y = x must be the same.

For the convenience of the user the function Equivalent_Elements is declared explicitly. It is equivalent to

```
function Equivalent_Elements(Left, Right: Element_Type) return Boolean is
begin
  return not (Left < Right) and not (Right < Left);
end Equivalent_Elements;
```

This function Equivalent_Elements corresponds to the formal generic parameter of the same name in the hashed sets package discussed below. This should make it easier to convert between the two forms of packages.

```
function "=" (Left, Right: Set) return Boolean;
function Equivalent_Sets(Left, Right: Set) return Boolean;
function To_Set(New_Item: Element_Type) return Set;
function Length(Container: Set) return Count_Type;
function Is_Empty(Container: Set) return Boolean;
procedure Clear(Container: in out Set);
```

Note the addition of Equivalent_Sets and To_Set. Two sets are equivalent if they have the same number of elements and the pairs of elements are equivalent. This contrasts with the function "=" where the pairs of elements have to be equal rather than equivalent. Remember that elements might be equivalent but not equal (as in the example of a string mentioned above). The function To_Set takes a single element and creates a set. It is particularly convenient when used in conjunction with operations such as Union described below. The other subprograms are as in the other containers.

```
function Element(Position: Cursor) return Element_Type;

procedure Replace_Element(Container: in out Set;
                          Position: in Cursor;
                          New_Item: in Element_Type);

procedure Query_Element(Position: in Cursor;
                        Process: not null access procedure (Element: in Element_Type));
```

Again these are much as expected except that there is no procedure Update_Element. This is because the elements are arranged in terms of their own value (either by order or through the hash function) and if we just change an element *in situ* then it might become out of place (this problem does not arise with the other containers). This also means that Replace_Element has to ensure that the value New_Item is not equivalent to an element in a different position; if it is then Program_Error is raised. We will return to the problem of the missing Update_Element later.

```
procedure Move(Target, Source: in out Set);
```

This is just as for the other containers.

```
procedure Insert(Container: in out Set;
                 New_Item: in Element_Type;
                 Position: out Cursor;
                 Inserted: out Boolean);

procedure Insert(Container: in out Set;
                 New_Item: in Element_Type);
```

These insert a new element into the set unless an equivalent element already exists. If it does exist then the first one returns with Inserted set to False and Position indicating the element whereas the second raises Constraint_Error (the element value is not changed). If an equivalent element is not in the set then it is added and Position is set accordingly.

```
procedure Include(Container: in out Set; New_Item: in Element_Type);
```

This is somewhat like the last Insert except that if an equivalent element is already in the set then it is replaced (rather than raising Constraint_Error).

```
procedure Replace(Container: in out Set; New_Item: in Element_Type);
```

In this case, Constraint_Error is raised if an equivalent element does not already exist.

```
procedure Exclude(Container: in out Set; Item: in Element_Type);
```

If an element equivalent to Item is already in the set, then it is deleted.

```
procedure Delete(Container: in out Set; Item: in Element_Type);

procedure Delete(Container: in out Set; Position: in out Cursor);
```

These delete an element. In the first case if there is no such equivalent element then Constraint_Error is raised. In the second case if the cursor is No_Element then again Constraint_Error is also raised – there is also a check to ensure that the cursor otherwise does designate an element in the correct set (remember that cursors designate both an entity and the container); if this check fails then Program_Error is raised.

And now some new stuff, the usual set operations.

```
procedure Union(Target: in out Set; Source: in Set);
function Union(Left, Right: Set) return Set;
function "or" (Left, Right: Set) return Set renames Union;

procedure Intersection(Target: in out Set; Source: in Set);
function Intersection(Left, Right: Set) return Set;
function "and" (Left, Right: Set) return Set renames Intersection;

procedure Difference(Target: in out Set; Source: in Set);
function Difference(Left, Right: Set) return Set;
function "–" (Left, Right: Set) return Set renames Difference;

procedure Symmetric_Difference(Target: in out Set; Source: in Set);
function Symmetric_Difference (Left, Right: Set) return Set;
function "xor" (Left, Right: Set) return Set renames Symmetric_Difference;
```

These all do exactly what one would expect using the equivalence relation on the elements.

```
function Overlap(Left, Right: Set) return Boolean;
function Is_Subset(Subset: Set; Of_Set: Set) return Boolean;
```

These are self-evident as well.

```
function First(Container: Set) return Cursor;
function Last(Container: Set) return Cursor;
function Next(Position: Cursor) return Cursor;
procedure Next(Position: in out Cursor);
function Find(Container: Set; Item: Element_Type) return Cursor;
function Contains(Container: Set; Item: Element_Type) return Boolean;
```

These should be self-evident and are very similar to the corresponding operations on maps. Again unlike the operations on vectors and lists, Find logically searches the whole set and not just starting at some point (there is also no Reverse_Find). Moreover, Find uses the equivalence relation based on the "<" parameter.

```
function Has_Element(Position: Cursor) return Boolean;

procedure Iterate(Container: in Set;
                  Process: not null access procedure (Position: in Cursor));
```

These are also as for other containers.

The sets packages conclude with an internal generic package called Generic_Keys. This package enables some set operations to be performed in terms of keys where the key is a function of the element. Note carefully that in the case of a map, the element is defined in terms of the key whereas here the situation is reversed. An equivalence relationship is defined for these keys as well; this is defined by a generic parameter "<" for ordered sets and Equivalent_Keys for hashed sets.

In the case of ordered sets the formal parameters are

```
generic
  type Key_Type(<>) is private;
  with function Key(Element: Element_Type) return Key_Type;
  with function "<" (Left, Right: Key_Type) return Boolean is <>;
package Generic_Keys is
```

The following are then common to the package Generic_Keys for both hashed and ordered sets.

```
function Key(Position: Cursor) return Key_Type;
function Element(Container: Set; Key: Key_Type) return Element_Type;

procedure Replace(Container: in out Set;
                  Key: in Key_Type; New_Item: in Element_Type);

procedure Exclude(Container: in out Set; Key: in Key_Type);
procedure Delete(Container: in out Set; Key: in Key_Type);

function Find(Container: Set; Key: Key_Type) return Cursor;
function Contains(Container: Set; Key: Key_Type) return Boolean;

procedure Update_Element_Preserving_Key(
                  Container: in out Set; Position: in Cursor;
                  Process: not null access procedure (Element: in out Element_Type));
```

and then finally

```
end Generic_Keys;

private
  ... -- not specified by the language
end Ada.Containers.Ordered_Sets;
```

It is expected that most user of sets will use them in a straightforward manner and that the operations specific to sets such as Union and Intersection will be dominant.

However, sets can be used as sort of economy class maps by using the inner package Generic_Keys. Although this is certainly not for the novice we will illustrate how this might be done by reconsidering the stock problem using sets rather than maps. We declare

```
type Part_Type is
  record
    Part_Number: Integer;
    Year: Integer;
    Shelf: Character range 'A' .. 'T';
    Stock: Integer;
  end record;
```

Here we have put all the information in the one type.

We then declare "<" much as before

```
function "<" (Left, Right: Part_Type) return Boolean is
begin
  return Left.Part_Number < Right.Part_Number;
end "<";
```

and then instantiate the package thus

```
package Store_Sets is new Ordered_Sets(Element_Type => Part_Type);

The_Store: Store_Sets.Set;
```

We have used the default generic parameter mechanism for "<" this time by way of illustration.

In this case we add items to the store by calling

```
The_Store.Insert((34618, 1998, 'F', 25));
The_Store.Insert((27134, 2004, 'C', 45));
...
```

The procedure for checking the stock could now become

```
procedure Request(Part: in Integer: OK: out Boolean;
                  Year: out Integer; Shelf: out Character) is
  C: Cursor;
  E: Part_Type;
begin
  C := The_Store.Find((Part, others => <>));
  if C = No_Element then
    OK := False; return;                    -- no such item
  end if;
  E := Element(C);
  Year := E.Year;
  Shelf := E.Shelf;
  if E.Stock = 0 then
    OK := False; return;                    -- out of stock
  end if;
  Replace_Element(C, (E.Part_Number, Year; Shelf, E.Stock–1));
  OK := True;
end Request;
```

This works but is somewhat unsatisfactory. For one thing we have had to make up dummy components in the call of Find (using <>) and moreover we have had to replace the whole of the element although we only wanted to update the Stock component. Moreover, we cannot use Update_Element because it is not defined for sets at all. Remember that this is because it might make things out of order; that wouldn't be a problem in this case because we don't want to change the part number and our ordering is just by the part number.

A better approach is to use the part number as a key. We define

```
type Part_Key is new Integer;

function Part_No(P: Part_Type) return Part_Key is
begin
  return Part_Key(P.Part_Number);
end Part_No;
```

and then

```
package Party is new Generic_Keys(Key_Type => Part_Key, Key => Part_No);
use Party;
```

Note that we do not have to define "<" on the type Part_Key at all because it already exists since Part_Key is an integer type. And the instantiation uses it by default.

And now we can rewrite the Request procedure as follows

```
procedure Request(Part: in Part_Key; OK: out Boolean;
                  Year: out Integer; Shelf: out Character) is
  C: Cursor;
  E: Part_Type;
begin
  C := Find(The_Store, Part);
  if C = No_Element then
    OK := False; return;                    -- no such item
  end if;
  E := Element(C);
  Year := E.Year;  Shelf := E.Shelf;
  if E.Stock = 0 then
    OK := False; return;                    -- out of stock
  end if;

  -- we are now going to update the stock level
  declare
    procedure Do_It(E: in out Part_Type) is
    begin
      E.Stock := E.Stock – 1;
    end Do_It;
  begin
    Update_Element_Preserving_Key(The_Store, C, Do_It'Access);
  end;
  OK := True;
end Request;
```

This seems hard work but has a number of advantages. The first is that the call of Find is more natural and only involves the part number (the key) – note that this is a call of the function Find in the instantiation of Generic_Keys and takes just the part number. And the other is that the update only involves the component being changed. We mentioned earlier that there was no

Update_Element for sets because of the danger of creating a value that was in the wrong place. In the case of the richly named Update_Element_Preserving_Key it also checks to ensure that the element is indeed still in the correct place (by checking that the key is still the same); if it isn't it removes the element and raises Program_Error.

But the user is warned to take care when using the package Generic_Keys. It is absolutely vital that the relational operation and the function (Part_No) used to instantiate Generic_Keys are compatible with the ordering used to instantiate the parent package Containers.Ordered_Sets itself. If this is not the case then the sky might fall in.

Incidentally, the procedure for checking the stock which previously used the maps package now becomes

```
procedure Check_Stock(Low: in Integer) is

  procedure Check_It(C: in Cursor) is
  begin
    if Element(C).Stock < Low then
      -- print a message perhaps
      Put("Low stock of part ");
      Put_Line(Element(C).Part_Number);          -- changed
    end if;
  end Check_It;

begin
  The_Store.Iterate(Check_It'Access);
end Check_Stock;
```

The only change is that the call of Key in

```
Put_Line(Key(C).Part_Number);
```

when using the maps package has been replaced by Element. A minor point is that we could avoid calling Element twice by declaring a constant E in Check_It thus

```
E: constant Part_Type := Element(C);
```

and then writing E.Stock < Low and calling Put_Line with E.Part_Number.

A more important point is that if we have instantiated the Generic_Keys inner package as illustrated above then we can leave Check_It unchanged to call Key. But it is important to realise that we are then calling the function Key internal to the instantiation of Generic_Keys (flippantly called Party) and not that from the instantiation of the parent ordered sets package (Store_Sets) because that has no such function. This illustrates the close affinity between the sets and maps packages.

And finally there is a hashed sets package which has strong similarities to both the ordered sets package and the hashed maps package. We can introduce this much as for hashed maps by giving the differences between the two sets packages, the extra facilities in each and the impact on the part number example.

The specification of the hashed sets package starts

```
generic
  type Element_Type is private;
  with function Hash(Element: Element_Type) return Hash_Type;
  with function Equivalent_Elements(Left, Right: Element_Type) return Boolean;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Hashed_Sets is
  pragma Preelaborate(Hashed_Sets);
```

The differences from the ordered sets package are that there is an extra generic parameter Hash and the ordering parameter "<" has been replaced by the function Equivalent_Elements.

So if we have

```
function Equivalent_Parts(Left, Right: Part_Type) return Boolean is
begin
  return Left.Part_Number = Right.Part_Number;
end Equivalent_Parts;

function Part_Hash(P: Part_Type) return Hash_Type is
  M31: constant := 2**31–1;                -- a nice Mersenne prime
begin
  return Hash_Type(P.Part_Number) * M31;
end Part_Hash;
```

(which are very similar to the hashed map example – the only changes are to the parameter type name) then we can instantiate the hashed sets package as follows

```
package Store_Sets is
   new Hashed_Sets(Element_Type => Part_Type,
                   Hash => Part_Hash,
                   Equivalent_Elements => Equivalent_Parts);

The_Store: Store_Sets.Set;
```

and then the rest of our example will be exactly as before. It is thus easy to convert from an ordered set to a hashed set and vice versa provided of course that we only use the facilities common to both.

It should also be mentioned that the inner package Generic_Keys for hashed sets has the following formal parameters

```
generic
  type Key_Type(<>) is private;
  with function Key(Element: Element_Type) return Key_Type
  with function Hash(Key: Key_Type) return Hash_Type;
  with function Equivalent_Keys(Left, Right: Key_Type) return Boolean;
package Generic_Keys is
```

The differences from that for ordered sets are the addition of the function Hash and the replacement of the comparison operator "<" by Equivalent_Keys.

(Incidentally the package Generic_Keys for ordered sets also exports a function Equivalent_Keys for uniformity with the hashed sets package.)

Although our example itself is unchanged we do have to change the instantiation of Generic_Keys thus

```
type Part_Key is new Integer;

function Part_No(P: Part_Type) return Part_Key is
begin
  return Part_Key(P.Part_Number);
end Part_No;

function Part_Hash(P: Part_Key) return Hash_Type is
  M31: constant := 2**31–1;                -- a nice Mersenne prime
begin
```

```
  return Hash_Type(P) * M31;
end Part_Hash;

function Equivalent_Parts(Left: Right: Part_Key) return Boolean is
begin
  return Left = Right;
end Equivalent_Parts;
```

and then

```
package Party is
   new Generic_Key(Key_Type => Part_Key,
                   Key => Part_No;
                   Hash => Part_Hash
                   Equivalent_Keys => Equivalent_Parts);
use Party;
```

The hash function is similar to that used with hashed maps. The type Part_Key and function Part_No are the same as for ordered sets. We don't really need to declare the function Equivalent_Parts since we could use "=" as the actual parameter for Equivalent_Keys.

We will finish this discussion of sets by briefly considering the additional facilities in the two sets packages (and their inner generic keys packages) just as we did for the two maps packages (the discussion is almost identical).

The ordered sets package has the following additional subprograms

```
procedure Delete_First(Container: in out Set);
procedure Delete_Last(Container: in out Set);

function First_Element(Container: Set) return Element_Type;
function Last_Element(Container: Set) return Element_Type;
function Previous(Position: Cursor) return Cursor;
procedure Previous(Position: in out Cursor);

function Floor(Container: Set; Item: Element_Type) return Cursor;
function Ceiling(Container: Set; Item: Element_Type) return Cursor;

function "<" (Left, Right: Cursor) return Boolean;
function ">" (Left, Right: Cursor) return Boolean;
function "<" (Left: Cursor; Right: Element_Type) return Boolean;
function ">" (Left: Cursor; Right: Element_Type) return Boolean;
function "<" (Left: Element_Type; Right: Cursor) return Boolean;
function ">" (Left: Element_Type; Right: Cursor) return Boolean;

procedure Reverse_Iterate(Container: in Set;
          Process: not null access procedure (Position: in Cursor));
```

These are again largely self-evident. The functions Floor and Ceiling are similar to those for ordered maps – Floor searches for the last element which is not greater than Item and Ceiling searches for the first element which is not less than Item – they return No_Element if there is not one.

The functions "<" and ">" are very important for ordered sets. The first is equivalent to

```
function "<" (Left, Right: Cursor) return Boolean is
begin
  return Element(Left) < Element(Right);
end "<";
```

There is a general philosophy that the container packages should work efficiently even if the elements themselves are very large – perhaps even other containers. We should therefore avoid copying elements. (Passing them as parameters is of course no problem since they will be passed by reference if they are large structures.) So in this case the built-in comparison is valuable because it can avoid the copying which would occur if we wrote the function ourselves with the explicit internal calls of the function Element.

On the other hand, there is a general expectation that keys will be small and so there is no corresponding problem with copying keys. Thus such built-in functions are less important for maps than sets but they are provided for maps for uniformity.

The following are additional in the package Generic_Keys for ordered sets

```
function Equivalent_Keys(Left, Right: Key_Type) return Boolean;
```

This corresponds to the formal generic parameter of the same name in the package Generic_Keys for hashed sets as mentioned earlier.

```
function Floor(Container: Set; Key: Key_Type) return Cursor;
function Ceiling(Container: Set; Key: Key_Type) return Cursor;
```

These are much as the corresponding functions in the parent package except that they use the formal parameter "<" of Generic_Keys for the search.

Hashed sets, like hashed maps also have the facility to specify a capacity as for the vectors package. Thus we have

```
procedure Reserve_Capacity(Container: in out Set; Capacity: in Count_Type);

function Capacity(Container: Set) return Count_Type;
```

The behaviour is much as for vectors and hashed maps. We don't have to set the capacity ourselves since it will be automatically extended as necessary but it might significantly improve performance to do so. Note again that Length(S) cannot exceed Capacity(S) but might be much less.

The other additional subprograms for hashed sets are

```
function Equivalent_Elements(Left, Right: Cursor) return Boolean;
function Equivalent_Elements(Left: Cursor; Right: Element_Type) return Boolean;
function Equivalent_Elements(Left: Element_Type; Right: Cursor) return Boolean;
```

Again, these are very important for sets. The first is equivalent to

```
function Equivalent_Elements(Left, Right: Cursor) return Boolean is
begin
  return Equivalent_Elements(Element(Left), Element(Right));
end Equivalent_Elements;
```

and once more we see that the built-in functions can avoid the copying of the type Element that would occur if we wrote the functions ourselves.

5 Indefinite containers

There are versions of the six container packages we have just been discussing for indefinite types.

As mentioned in Section 1, an indefinite (sub)type is one for which we cannot declare an object without giving a constraint (either explicitly or though an initial value). Moreover we cannot have an array of an indefinite subtype. The type String is a good example. Thus we cannot declare an array of the type String because the components might not all be the same size and indexing would be a pain. Class wide types are also indefinite.

The specification of the indefinite container for lists starts

```
generic
  type Element_Type(<>) is private;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Indefinite_Doubly_Linked_Lists is
  pragma Preelaborate(Indefinite_Doubly_Linked_Lists);
```

where we see that the formal type Element_Type has unknown discriminants and so permits the actual type to be any indefinite type (and indeed a definite type as well). So if we want to manipulate lists of strings where the individual strings can be of any length then we declare

```
package String_Lists is new  Ada.Containers.Indefinite_Doubly_Linked_Lists(String);
```

In the case of ordered maps we have

```
generic
  type Key_Type(<>) is private;
  type Element_Type(<>)  is private;
  with function "<" (Left, Right: Key_Type) return Boolean is <>;
  with function "=" (Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Indefinite_Ordered_Maps is
  pragma Preelaborate(Indefinite_Ordered_Maps);
```

showing that both Element_Type and Key_Type can be indefinite.

There are two other differences from the definite versions which should be noted.

One is that the Insert procedures for Vectors, Lists and Maps which insert an element with its default value are omitted (because there is no way to create a default initialized object of an indefinite type anyway).

The other is that the parameter Element of the access procedure Process of Update_Element (or the garrulous Update_Element_Preserving_Key in the case of sets) can be constrained even if the type Element_Type is unconstrained.

As an example of the use of an indefinite container consider the problem of creating an index. For each word in a text file we need a list of its occurrences. The individual words can be represented as just objects of the type String. It is perhaps convenient to consider strings to be the same irrespective of the case of characters and so we define

```
function Same_Strings(S, T: String) return Boolean is
begin
  return To_Lower(S) = To_Lower(T);
end Same_Strings;
```

where the function To_Lower is from the package Ada.Characters.Handling.

We can suppose that the positions of the words are described by a type Place thus

```
type Place is
  record
    Page: Text_IO.Positive_Count;
    Line: Text_IO.Positive_Count;
    Col: Text_IO.Positive_Count;
  end record;
```

The index is essentially a map from the type String to a list of values of type Place. We first create a definite list container for handling the lists thus

```
package Places is new Doubly_Linked_Lists(Place);
```

We then create an indefinite map container from the type String to the type List thus

```
package Indexes is new Indefinite_Hashed_Maps(
            Key_Type => String;
            Element_Type => Places.List;
            Hash => Ada.Strings.Hash;
            Equivalent_Keys => Same_Strings;
            "=" => Places."=");
```

The index is then declared by writing

```
The_Index: Indexes.Map;
```

Note that this example illustrates the use of nested containers since the elements in the map are themselves containers (lists).

It might be helpful for the index to contain information saying which file it refers to. We can extend the type Map thus (remember that container types are tagged)

```
type Text_Map is new Indexes.Map with
  record
    File_Ref: Text_IO.File_Access;
  end record;
```

and now we can more usefully declare

```
My_Index: Text_Map := (Indexes.Empty_Map with My_File'Access);
```

We can now declare various subprograms to manipulate our map. For example to add a new item we have first to see whether the word is already in the index – if it is not then we add the new word to the map and set its list to a single element whereas if it is already in the index then we add the new place entry to the corresponding list. Thus

```
procedure Add_Entry(Index: in out Text_Map; Word: String; P: Place) is
  M_Cursor: Indexes.Cursor;
  A_List: Places.List;                    -- empty list of places
begin
  M_Cursor := Index.Find(Word);
  if M_Cursor = Indexes.No_Element then
    -- it's a new word
    A_List.Append(P);
    Index.Insert(Word, A_List);
  else
    -- it's an old word
    A_List := Element(M_Cursor);          -- get old list
    A_List.Append(P);                     -- add to it
    Index.Replace_Element(M_Cursor, A_List);
  end if;
end Add_Entry;
```

A number of points should be observed. The type Text_Map being derived from Indexes.Map inherits all the map operations and so we can write Index.Find(Word) which uses the prefixed notation (or we can write Indexes.Find(Index, Word)). On the other hand auxiliary entities such as the type Cursor and the constant No_Element are of course in the package Indexes and have to be referred to as Indexes.Cursor and so on.

A big problem with the procedure as written however is that it uses Element and Replace_Element rather than Update_Element. This means that it copies the whole of the existing list, adds the new item to it, and then copies it back. Here is an alternative version

```
procedure Add_Entry(Index: in out Text_Map; Word: String; P: Place) is
  M_Cursor: Indexes.Cursor;
  A_LIst: Places.List;                    -- empty list of places
begin
  M_Cursor := Index.Find(Word);
  if M_Cursor = Indexes.No_Element then
    -- it's a new word
    A_LIst.Append(P);
    Index.Insert(Word, A_List);
  else
    -- it's an old word
    declare
      -- this procedure adds to the list in situ
      procedure Add_It(The_Key: in String; The_List: in out Places.List) is
      begin
        The_List.Append(P);
      end Add_It;
    begin
      -- and here we call it via Update_Element
      Index.Update_Element(M_Cursor, Add_It'Access);
    end;
  end if;
end Add_Entry;
```

This is still somewhat untidy. In the case of a new word we might as well make the new map entry with an empty list and then update it thereby sharing the calls of Append. We get

```
procedure Add_Entry(Index: in out Text_Map; Word: String; P: Place) is
  M_Cursor: Indexes.Cursor := Index.Find(Word);
  OK: Boolean;
begin
  if M_Cursor = Indexes.No_Element then
    -- it's a new word
    Index.Insert(Word, Places.Empty_List, M_Cursor, OK);
    -- M_Cursor now refers to new position
    -- and OK will be True
  end if;
  declare
   -- this procedure adds to the list in situ
    procedure Add_It(The_Key: in String; The_List: in out Places.List) is
    begin
      The_List.Append(P);
    end Add_It;
  begin
    -- and here we call it via Update_Element
    Index.Update_Element(M_Cursor, Add_It'Access);
  end;
end Add_Entry;
```

It will be recalled that there are various versions of Insert. We have used that which has two out parameters being the position where the node was inserted and a Boolean parameter indicating whether a new node was inserted or not. In this case we know that it will be inserted and so the final parameter is a nuisance (but sadly we cannot default out parameters). Note also that we need not give the parameter Places.Empty_List because another version of Insert will do that automatically since that is the default value of a list anyway.

Yet another approach is not to use Find but just call Insert. We can even use the defaulted version – if the word is present then the node is not changed and the position parameter indicates where it is, if the word is not present then a new node is made with an empty list and again the position parameter indicates where it is.

```
procedure Add_Entry(Index: in out Text_Map; Word: String; P: Place) is
  M_Cursor: Indexes.Cursor;
  Inserted: Boolean;
begin
  Index.Insert(Word, M_Cursor, Inserted);
  -- M_Cursor now refers to position of node
  -- and Inserted indicates whether it was added
  declare
   -- this procedure adds to the list in situ
    procedure Add_It(The_Key: in String; The_List: in out Places.List) is
    begin
      The_List.Append(P);
    end Add_It;
  begin
    -- and here we call it via Update_Element
    Index.Update_Element(M_Cursor, Add_It'Access);
  end;
end Add_Entry;
```

Curiously enough we do not need to use the value of Inserted. We leave the reader to decide which of the various approaches is best.

We can now do some queries on the index. For example we might want to know how many different four-lettered words there are in the text. We can either use Iterate or do it ourselves with Next as follows

```
function Four_Letters(Index: Text_Map) return Integer is
  Count: Integer := 0;
  C: Indexes.Cursor := Index.First;
begin
  loop
    if Key(C)'Length = 4 then
      Count := Count + 1;
    end if;
    Indexes.Next(C);
    exit when C = Indexes.No_Element;
  end loop;
  return Count;
end Four_Letters;
```

We might finally wish to know how many four-lettered words there are on a particular page. (This is just an exercise – it would clearly be simplest to search the original text!) We use Iterate this time both to scan the map for the words and then to scan each list for the page number

```
function Four_Letters_On_Page(Index: Text_Map;
                              Page: Text_IO.Positive_Count) return Integer is
  Count: Integer := 0;

  procedure Do_It_Map(C: Indexes.Cursor) is

    procedure Do_It_List(C: Places.Cursor) is
    begin
      if Element(C).Page = Page then
        Count := Count + 1;
      end if;
    end Do_It_LIst;

    procedure Action(K: String; E: Places.List) is
    begin
      if K'Length = 4 then
        -- now scan list for instances of Page
        E.Iterate(Do_It_List'Access);
      end if;
    end Action;

  begin
    Indexes.Query_Element(C, Action'Access);
  end Do_It_Map;

begin
  Index.Iterate(Do_It_Map'Access);
  return Count;
end Four_Letters_On_Page;
```

We could of course have used First and Next to search the list. But in any event the important point is that by using Query_Element we do not have to copy the list in order to scan it.

6 Sorting

The final facilities in the container library are generic procedures for array sorting. There are two versions, one for unconstrained arrays and one for constrained arrays. Their specifications are

```
generic
  type Index_Type is (<>);
  type Element_Type is private;
  type Array_Type is array (Index_Type range <>) of Element_Type;
  with function "<" (Left, Right: Element_Type) return Boolean is <>;
procedure Ada.Containers.Generic_Array_Sort(Container: in out Array_Type);
pragma Pure(Ada.Containers.Generic_Array_Sort);
```

and

```
generic
  type Index_Type is (<>);
  type Element_Type is private;
  type Array_Type is array (Index_Type) of Element_Type;
  with function "<" (Left, Right: Element_Type) return Boolean is <>;
```

```
procedure Ada.Containers.Generic_Constrained_Array_Sort(Container: in out Array_Type);
pragma Pure(Ada.Containers.Generic_Constrained_Array_Sort);
```

These do the obvious thing. They sort the array Container into order as defined by the generic parameter "<". The emphasis is on speed.

7 Summary table

This paper concludes with an appendix showing at a glance the various facilities in the six main containers.

References

[1] D. E. Knuth (1973). The Art of Computer Programming, vol 3 – Searching and Sorting, Addison-Wesley.

==========

Appendix Container summary

In order to save space the following abbreviations are used in the table:

T	container type eg Map	H_T	Hash_Type
C: T	Container: container type	I_T	Index_Type
P: C	Position: Cursor	K_T	Key_Type
L, R	Left, Right	Ex_Index	Extended_Index
C_T	Count_Type	B	Boolean
E_T	Element_Type		

also Index – means that another subprogram exists with similar parameters except that the first parameters are of type Vector and Index_Type (or Extended_Index) rather than those involving cursors.

also Key and also Element similarly apply to maps and sets respectively.

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets
generic	Y	Y	Y	Y	Y	Y
type Index_Type is range <>;	Y					
type Key_Type is private;			Y	Y		
type Element_Type is private;	Y	Y	Y	Y	Y	Y
with function Hash(...) return Hash_Type;			on Key		on Element	
with function Equivalent_...(L, R: ...) return Boolean;			on Key		on Element	
with function "<" (L, R: ...) return Boolean is <>;				on Key		on Element
with function "=" (L, R: E_T) return B is <>;	Y	Y	Y	Y	Y	Y
package Ada.Containers.... is	Vectors	Doubly_ Linked_ Lists	Hashed_ Maps	Ordered_ Maps	Hashed_ Sets	Ordered_ Sets
pragma Preelaborate(...);	Y	Y	Y	Y	Y	Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets
function Equivalent_...(L, R: ...) return Boolean;				on Key		on Element
subtype Extended_Index ... No_Index: constant Ex_Ind := Ex_Ind'First;	Y					
type T is tagged private; pragma Preelaborable_Initialization(T);	Vector	List	Map	Map	Set	Set
type Cursor is private; pragma Preelaborable_Initialization(Cursor);	Y	Y	Y	Y	Y	Y
Empty_T: constant T;	Vector	List	Map	Map	Set	Set
No_Element: constant Cursor;	Y	Y	Y	Y	Y	Y
function "=" (Left, Right: T) return Boolean;	Y	Y	Y	Y	Y	Y
function Equivalent_Sets(L, R: Set) return Boolean; function To_Set(New_Item: E_T) return Set;					Y	Y

function To_Vector(Length: C_T) return Vector;
function To_Vector(New_Item: E_T;
Length: C_T) return Vector;
	Y					

function "&" (L, R: Vector) return Vector; function "&" (L: Vector; R: E_T) return Vector; function "&" (L: E_T; R: Vector) return Vector; function "&" (L, R: E_T) return Vector;	Y					
function Capacity(C: T) return C_T; procedure Reserve_Capacity(C: T; Capacity: C_T);	Y		Y		Y	
function Length(C: T) return Count_Type;	Y	Y	Y	Y	Y	Y
procedure Set_Length(C: in out T; Length: in C_T);	Y					
function Is_Empty(C: T) return B; procedure Clear(C: in out T);	Y	Y	Y	Y	Y	Y
function To_Cursor(C: Vector; Index: Ex_Ind) return Cursor; function To_Index(P: C) return Ex_Ind;	Y					
function Key(P: C) return K_T;			Y	Y		
function Element(P: C) return E_T;	Y also Index	Y	Y	Y	Y	Y
procedure Replace_Element(C: in out T; P: C; New_Item: E_T);	Y also Index	Y	Y	Y	Y	Y
procedure Query_Element(P: C; Process: not null acc proc(...));	in Element also Index	in Element	in Key, in Element	in Key, in Element	in Element	in Element
procedure Update_Element(C: in out T; P: C; Process: not null acc proc(...));	in out Elem also Index	in out Elem	in Key, in out Elem	in Key, in out Elem		
procedure Move(Target, Source: in out T);	Y	Y	Y	Y	Y	Y

procedure Insert(C: in out Vector; Before: Ex_Ind;
New_Item: Vector);
procedure Insert(C: in out Vector; Before: Cursor;
New_Item: Vector);
procedure Insert(C: in out Vector; Before: Cursor;
New_Item: Vector; Position: out Cursor);
	Y					

procedure Insert(C: in out T; Before: C; New_Item: E_T; Count: C_T := 1);	Y also Index	Y				

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets
procedure Insert(C: in out T; Before: C; New_Item: E_T; Position: out Cursor; Count: C_T := 1);	Y	Y				
procedure Insert(C: in out T; Before: C; Position: out Cursor; Count: C_T := 1); element has default value	Y also Index	Y				
procedure Insert(C: in out T; Key: K_T; New_Item: E_T; Position: out Cursor; Inserted: out B);			Y	Y	Y (no key)	Y (no key)
procedure Insert(C: in out T; Key: K_T; Position: out Cursor; Inserted: out B); element has default value			Y	Y		
procedure Insert(C: in out T; Key: K_T; New_Item: E_T);			Y	Y	Y (no key)	Y (no key)
procedure Prepend(C: in out Vector; New_Item: Vector);	Y					
procedure Prepend(C: in out T; New_Item: E_T; Count: C_T := 1);	Y	Y				
procedure Append(C: in out Vector; New_Item: Vector);	Y					
procedure Append(C: in out T; New_Item: E_T; Count: C_T := 1);	Y	Y				
procedure Insert_Space(C: in out V; Before: Cursor; Position: out Cursor; Count: C_T := 1);	Y also Index					
procedure Include(C: in out T; Key: Key_Type; New_Item: E_T);			Y	Y	Y (no key)	Y (no key)
procedure Replace(C: in out T; Key: Key_Type; New_Item: E_T);			Y	Y	Y (no key)	Y (no key)
procedure Exclude(C: in out T; Key: Key_Type);			Y	Y	Y (Item not key)	Y (item not key)
procedure Delete(C: in out T; P: in out C; Count: C_T := 1);	Y also Index	Y	Y (no count) also Key	Y (no count) also Key	Y (no count) also Element	Y (no count) also Element
procedure Delete_First(C: in out T; Count: C_T := 1); procedure Delete_Last(C: in out T; Count: C_T := 1);	Y	Y		Y (no count)		Y (no count)
procedure Reverse_Elements(C: in out T);	Y	Y				
procedure Swap(C: in out T; I, J: Cursor);	Y also Index	Y				
procedure Swap_Links(C: in out List; I, J: Cursor);		Y				

procedure Splice(Target: in out List; Before: Cursor;
Source: in out List);
procedure Splice(Target: in out List; Before: Cursor;
Source: in out List; Position: in out Cursor);
procedure Splice(Container: in out List; Before: Cursor;
Position: in out Cursor);
		Y				

procedure Union(Target: in out Set; Source: Set);
function Union(L, R: Set) return Set;
function "or" (L, R: Set) return Set renames Union;
					Y	Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets

procedure Intersection(Target: in out Set; Source: Set);
function Intersection(L, R: Set) return Set;
function "and" (L, R: Set) return Set renames Intersection;
					Y	Y

procedure Difference(Target: in out Set; Source: Set);
function Difference(L, R: Set) return Set;
function "–" (L, R: Set) return Set renames Difference;
					Y	Y

procedure Symmetric_Difference(Target: in out Set; Source: Set); function Symmetric_Difference (L, R: Set) return Set; function "xor" (L, R: Set) return Set renames Symmetric_Difference;					Y	Y
function Overlap(L, R: Set) return Boolean; function Is_Subset(Subset: Set; Of_Set: Set) return B;					Y	Y
function First_Index(C: T) return Index_Type;	Y					
function First(C: T) return Cursor;	Y	Y	Y	Y	Y	Y
function First_Element(C: T) return Element_Type;	Y	Y		Y		Y
function First_Key(C: T) return Key_Type;				Y		
function Last_Index(C: T) return Ex_Ind;	Y					
function Last(C: T) return Cursor;	Y	Y		Y		Y
function Last_Element(C: T) return Element_Type;	Y	Y		Y		Y
function Last_Key(C: T) return Key_Type;				Y		
function Next(P: C) return Cursor; procedure Next(P: in out C);	Y	Y	Y	Y	Y	Y
function Previous(P: C) return Cursor; procedure Previous(P: in out C);	Y	Y		Y		Y
function Find_Index(C: T; Item: E_T; Index: I_T := I_T'First) return Ex_Ind;	Y					
function Find(C: T; ... ; P: C := No_Element) return Cursor;	Element	Element	Key (no position)	Key (no position)	Element (no position)	Element (no position)
function Element(C: T; Key: K_T) return E_T;			Y	Y		
function Reverse_Find_Index(C: T; Item: E_T; Index: I_T := I_T'First) return Ex_Ind;	Y					
function Reverse_Find(C: T; ... ; P: C := No_Element) return Cursor;	Element	Element				
function Floor(C: T; ...) return Cursor; function Ceiling(C: T; ...) return Cursor;				Key: K_T		Item: E_T
function Contains(C: T; ...) return Boolean;	Element	Element	Key	Key	Element	Element
function Has_Element(P: C) return Boolean;	Y	Y	Y	Y	Y	Y
function Equivalent_... (L, R: Cursor) return Boolean; function Equivalent_... (L: Cursor; R:...) return Boolean; function Equivalent_... (L:...; R: Cursor) return Boolean;			Keys		Elements	

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets
function "<" (L, R: Cursor) return Boolean; function ">" (L, R: Cursor) return Boolean; function "<" (L, Cursor; R: ...) return Boolean; function ">" (L, Cursor; R: ...) return Boolean; function "<" (L:...; R: Cursor) return Boolean; function ">" (L:...; R: Cursor) return Boolean;				Key		Element
procedure Iterate(C: in T; Process: not null acc proc (P: C));	Y	Y	Y	Y	Y	Y
procedure Reverse_Iterate(C: in T; Process: not null acc proc (P: C));	Y	Y		Y		Y

generic
with function "<" (Left, Right: E_T) return B is <>;
package Generic_Sorting is
function Is_Sorted(C: T) return Boolean;
procedure Sort(C: in out T);
procedure Merge(Target, Source: in out T);
end Generic_Sorting;
	Y	Y				

generic type Key_Type (<>) is private;					Y	Y
with function Key(Element: E_T) return Key_Type;					Y	Y
with function Hash(Key: K_T) return Hash_Type;					Y	
with function Equivalent_Keys (L, R: Key_Type) return Boolean;					Y	
with function "<" (L, R: Key_Type) return B is <>;						Y
package Generic_Keys is					Y	Y
function Equivalent_Keys(L, R: Key_Type) return B;						Y
function Key(P: C) return Key_Type;					Y	Y
function Element(C: T; Key: K_T) return Element_T;					Y	Y

procedure Replace(C: in out T; Key: Key_Type;
New_Item: E_T);
procedure Exclude(C: in out T; Key: Key_Type);
procedure Delete(C: in out T; Key: Key_Type);
					Y	Y

function Find(C: T; Key: K_T) return Cursor;					Y	Y
function Floor(C: T; Key: K_T) return Cursor; function Ceiling(C: T; Key: K_T) return Cursor;						Y
function Contains(C: T; Key: K_T) return Boolean;					Y	Y
procedure Update_Element_Preserving_Key (C: in out T; P: C; Process: not null acc proc (Element: in out E_T));					Y	Y
end Generic_Keys;					Y	Y
private ... -- *not specified by the language* **end** Ada.Containers....;	Y	Y	Y	Y	Y	Y

Rationale for Ada 2005: Epilogue

John Barnes
John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This is the last of a number of papers describing the rationale for Ada 2005. In due course it is anticipated that the papers will be combined (after appropriate reformatting and editing) into a single volume for formal publication.

This last paper summarizes a small number of general issues of importance to the user such as compatibility between Ada 2005 and Ada 95. It also briefly considers a few potential changes that were considered for Ada 2005 but rejected for various reasons.

Keywords: rationale, Ada 2005.

1 Compatibility

There are two main sorts of problems regarding compatibility. These are termed Incompatibilities and Inconsistencies.

An incompatibility is a situation where a legal Ada 95 program is illegal in Ada 2005. These can be annoying but not a disaster since the compiler automatically detects such situations.

An inconsistency is where a legal Ada 95 program is also a legal Ada 2005 program but might have a different effect at execution time. These can in principle be really nasty but typically the program is actually wrong anyway (in the sense that it does not do what the programmer intended) or its behaviour depends upon the raising of a predefined exception (which is generally considered poor style) or the situation is extremely unlikely to occur.

As mentioned below in Section 2, during the development of Ada 2005 a number of corrections were made to Ada 95 and these resulted in some incompatibilities and inconsistencies with the original Ada 95 standard. These are not considered to be incompatibilities or inconsistencies between Ada 95 and Ada 2005 and so are not covered in this section.

1.1 Incompatibilities with Ada 95

Each incompatibility listed below gives the AI concerned and the paragraph in the AARM which in some cases will give more information. Where relevant, the section in this rationale where the topic is discussed is also given. Where appropriate the incompatibilities are grouped together.

1 – The words **interface**, **overriding** and **synchronized** are now reserved. Programs using them as identifiers will need to be changed. (AI-284, 2.9(3.c))

This is perhaps the most important incompatibility in terms of visibility to the average programmer. It is discussed in paper 1 section 2.

2 – If a predefined package has additional entities then incompatibilities can arise. Thus suppose the predefined package Ada.Stuff has an additional entity More added to it. Then if an Ada 95 program has a package P containing an entity More then a program with a use clause for both Ada.Stuff and P will become illegal in Ada 2005 because the reference to More will become ambiguous. This also applies if further overloadings of an existing entity are added.

Because of this there has been reluctance to extend existing packages but a preference to add child packages. Nevertheless in some cases extending a package seemed more appropriate especially if the identifiers concerned are unlikely to have been used by programmers.

The following packages have been extended with additional entities as listed.

Ada.Exceptions – Wide_Exception_Name, Wide_Wide_Exception_Name. (AI-400, 11.4.1(19.bb))

Ada.Real_Time – Seconds, Minutes. (AI-386, D.8(51.a))

Ada.Strings – Wide_Wide_Space. (AI-285, A.4.1(6.a))

Ada.Strings.Fixed – Index, Index_Non_Blank. (AI-301, A.4.3(109.a))

Ada.Strings.Bounded – Set_Bounded_String, Bounded_Slice, Index, Index_Non_Blank. (AI-301, A.4.4(106.f))

Ada.Strings.Unbounded – Set_Unbounded_String, Unbounded_Slice, Index, Index_Non_Blank. (AI-301, A.4.5(88.c))

There are similar additions to Ada.Strings.Wide_Fixed, Ada.Strings.Wide_Bounded and Ada.Strings.Wide_Unbounded. (AI-301, A.4.7(48.a))

Ada.Tags – No_Tag, Parent_Tag, Interface_Ancestor_Tags, Descendant_Tag, Is_Descendant_At_Same_Level, Wide_Expanded_Name, Wide_Wide_Expanded_Name. (AI-260, 344, 400, 405, 3.9(33.d))

Ada.Text_IO – Get_Line. (AI-301, A.10.7(26.a))

Interfaces.C – char16_t, char32_t and related types and operations. (AI-285, B.3(84.a))

It seems unlikely that existing programs will be affected by these potential incompatibilities.

3 – If a subprogram has an access parameter (without a null exclusion) and is not a dispatching operation then it cannot be renamed as a dispatching operation in Ada 2005 although it can be so renamed in Ada 95. See paper 2, section 2 for an example. (AI-404, 3.9.2(24.b))

4 – As discussed in paper 2 section 5, there are many awkward situations in Ada 95 regarding access types, discriminants and constraints. One problem is that some components can change shape or disappear. The rules in Ada generally aim to prevent such components from being accessed or renamed. However, in Ada 95, some entities don't look constrained but actually are constrained. The consequence is that it is difficult to prevent some constrained objects from having their constraints changed and this can cause components to change or disappear even though they might be accessed or renamed.

A key rule in Ada 95 was that aliased variables were always constrained with the intent that that would solve the problems. But loopholes remained and so the rules have been changed considerably. Aliased variables are not necessarily constrained in Ada 2005 and other rules now disallow certain constructions that were permitted in Ada 95 and this gives rise to a number of minor incompatibilities.

If a general access subtype refers to a type with default discriminants then that access subtype cannot have constraints in Ada 2005. Consider

```
type T(Disc: Boolean := False) is
  record
    ...
  end record;
```

The discriminated type T has a default and so unconstrained objects of type T are mutable. Suppose we now have

```
type T_Ptr is access all T;
subtype Sub_True_T_Ptr is T_Ptr(Disc => True);     -- subtype illegal in Ada 2005
```

The type T_Ptr is legal in both Ada 95 and Ada 2005 of course, but the subtype Sub_True_T_Ptr is only legal in Ada 95 and not in Ada 2005. The reason why the subtype cannot be permitted is illustrated by the following

```
Some_T: aliased T := (Disc => True, ...);
A_True_T: Sub_True_T_Ptr := Some_T'Access;
...
Some_T := (Disc => False, ...);
```

When Some_T'Access is evaluated there is a check that the discriminant has the correct value so that A_True_T is assigned a valid value. But the second assignment to Some_T means that the discriminant changes and so A_True_T would no longer have a valid value.

In Ada 95, all aliased variables were considered constrained and so the second assignment would not have been permitted anyway. But, as mentioned above, aliased variables are not considered to be constrained in Ada 2005 just because they are aliased.

Note that there is no similar restriction on types; thus we can still write

```
type True_T_Ptr is access all T(Disc => True);
```

because any conversion which might cause difficulties is forbidden as explained in one of the examples below.

The restriction on subtypes does not apply if the discriminants do not have defaults, nor to pool-specific types. (AI-363, 3.7.1(15.c))

Since aliased variables are not necessarily constrained in Ada 2005 there are situations where components might change shape or disappear in Ada 2005 that could not happen in Ada 95. Applying the Access attribute to such components is thus illegal in Ada 2005. Suppose the example above has components as follows

```
type T(Disc: Boolean := False) is
  record
    case Disc is
      when False =>
        Comp: aliased Integer;
      when True =>
        null;
    end case;
  end record;
```

Since objects of type T might be mutable, the component Comp might disappear.

```
type Int_Ptr is access all Integer;
Obj: aliased T;                          -- mutable object
Dodgy: Int_Ptr := Obj.Comp'Access;       -- take care
...
Obj:= (Disc => True);                    -- Comp gone
```

In Ada 95, the assignment to Dodgy is permitted but then the assignment to Obj raises Constraint_Error because there might be dodgy pointers.

In Ada 2005, the assignment statement to Dodgy is illegal since we cannot write Obj.Comp'Access. The assignment to Obj is itself permitted because we now know that there cannot be any dodgy pointers.

See (AI-363, 3.10.2(41.b)). Similarly, renaming an aliased component such as Comp is also illegal. (AI-363, 8.5.1(8.b))

There are related situations regarding discriminated private types where type conversions and the Access attribute are forbidden. Suppose we have a private type and an access type and that the full type is in fact the discriminated type above thus

```
package P is
  type T is private;
  type T_Ptr is access all T;
  function Evil return T_Ptr;
  function Flip(Obj: T) return T;
private
  type T(Disc: Boolean := False) is
    record
      ...
    end record;
  ...
end P;

package body P is

  type True_T_Ptr is access all T(Disc => True);
  subtype Sub_True_T_Ptr is T_Ptr(Disc => True);  -- legal in Ada 95, illegal in Ada 2005

  True_Obj: aliased T(Disc => True);
  TTP: True_T_Ptr := True_Obj'Access;
  STTP: Sub_True_T_Ptr := True_Obj'Access;

  function Evil return T_Ptr is
  begin
    if ... then
      return T_Ptr(TTP);                    -- OK in 95, not in 2005
    elsif ... then
      return True_Obj'Access;               -- OK in 95, not in 2005
    else
     return STTP;
    end if;
  end Evil;

  function Flip(Obj: T) return T is
  begin
    case Obj.Disc is
      when True => return (Disc => False, ...);
      when False => return (Disc => True, ...);
    end case;
  end Flip;

end P;
```

The function Evil has three branches illustrating various possible ways of returning a value of the type T. The function Flip just returns a value of the type T with opposite discriminants to the parameter. Now consider

```
with P;  use P;
procedure Do_It is
  A: T;
```

```
   B: T_Ptr := new T;
   C: T_Ptr := Evil;
begin
   A := Flip(A);
   B.all := Flip(B.all);
   C.all := Flip(C.all);
end Do_It;
```

This declares an object A of type T and then two objects B and C of the access type T_Ptr and initializes them in different ways. Finally it attempts to change the discriminant of the three objects by calling the function Flip.

In Ada 95 all objects on the heap are constrained. This means that clients cannot change the discriminants even if they do not know that they exist. So the assignment to B.**all** raises Constraint_Error since B.**all** is on the heap and thus constrained whereas the assignment to A is fine since A is not constrained. However, from the client's point of view they both really do the same thing and so the behaviour is very curious. Remember that the client doesn't know about the discriminants and so both operations look the same in the abstract. This is unfortunate and breaks privacy which is sinful. There is a similar example in paper 2, section 5 where we try to change Chris but do not know that the new value has a beard and this fails because Chris is female.

To prevent such privacy breaking the rules are changed in Ada 2005 so that objects on the heap are unconstrained in this one case. So the assignments to B.**all** and C.**all** do not have checks on the discriminant. As a consequence Evil must not return an object which is constrained otherwise the assignment to C would result in True_Obj having its discriminant turned to False.

All three possible branches in Evil are prevented in Ada 2005. The conversion in the first branch is forbidden and the Access attribute in the second branch is forbidden. In the case of the third branch the return itself is acceptable in principle because STTP is of the correct type. However, this is prevented by the rule mentioned above since the subtype Sub_True_T_Ptr is itself forbidden and so the object STTP could not be declared in the first place.

See (AI-363, 3.10.2(41.e) and 4.6(71.k)).

5 – Aggregates of limited types are permitted in Ada 2005 as discussed in paper 3, section 5. This means that in obscure situations an aggregate might be ambiguous in Ada 2005 and thus illegal. Consider

```
type Lim is limited
  record
    Comp: Integer;
  end record;

type Not_Lim is
  record
    Comp: Integer;
  end record;

procedure P(X: LIm);
procedure P(X: Not_Lim);

P((Comp => 123));                    -- illegal in Ada 2005
```

In Ada 95, the aggregate cannot be of a limited type and so the type Lim is not considered for resolution. But Ada 2005 permits aggregates of limited types and so the aggregate is ambiguous. (AI-287, 4.3(6.e))

Another similar situation with limited types and nonlimited types concerns assignment. Again this relates to the fact that limitedness is no longer considered for name resolution. Consider

```
type Acc_Not_Lim is access Not_Lim;
function F(X: Integer) return Acc_Not_Lim;
type Acc_Lim is access Lim;
function F(X: Integer) return Acc_Lim;
F(1).all := F(2).all;                    -- illegal in Ada 2005
```

In Ada 95, only the first F is considered for name resolution and the program is valid. In Ada 2005, there is an ambiguity because both functions are considered. Note of course that the assignment for the limited function is still illegal anyway but the compiler meets the ambiguity first. Clearly this is an obscure situation. (AI-287. 5.2(28.d))

6 – Because of the changes to the fixed-fixed multiplication and division rules there are situations where a legal program in Ada 95 becomes illegal in Ada 2005. Consider

```
package P is
  type My_Fixed is delta ... ;
  function "*" (L, R: My_Fixed) return My_Fixed;
end P;

use P;
A, B: My_Fixed;
D: Duration := A * B;                    -- illegal in Ada 2005
```

Although this is legal in Ada 95, the new rule in Ada 2005 says that if there is a user-defined operation involving the type concerned then the predefined operation cannot be used unless there is a type conversion or we write Standard."*"(...).

So in Ada 2005 a conversion can be used thus

```
D: Duration := Duration(A * B);
```

See paper 5, section 3. (AI-364, 4.5.5(35.d))

7 – The concept of return by reference types has gone. Instead the user has to explicitly declare a function with an anonymous access type as the return type. This only affects functions that return an existing limited object such as choosing a task from among a pool of tasks. See paper 3 section 5 for an example. (AI-318, 6.5(27.g))

8 – There is a very curious situation regarding exporting multiple homographs from an instantiation that is now illegal. This is a side effect of adding interfaces to the language. (AI-251, 8.3(29.s))

9 – The introduction of more forms of access types has changed the rules regarding name resolution. Consider the following contrived example

```
type Cacc is access constant Integer;
procedure Proc(Acc: access Integer);
procedure Proc(Acc: Cacc);
List: Cacc := ... ;
...
Proc(List);                              -- illegal in Ada 2005
```

In Ada 95 the call of Proc is resolved because the parameters Acc are anonymous access to variable in one case and access to constant in the other. In Ada 2005, the name resolution rules do not take this into account so it becomes ambiguous and thus illegal which is a good thing because it is likely that the Ada 95 programmer made a mistake anyway. (AI-409, 8.6(34.n))

10 – In Ada 2005, a procedure call that might be an entry is permitted in timed and conditional entry calls. See paper 4, section 3. In Ada 95, a procedure could not be so used and this fact is used in name resolution in Ada 95 but does not apply in Ada 2005. Hence if a procedure and an entry have the same profile then an ambiguity can exist in Ada 2005. (AI-345, 9.7.2(7.b))

11 – It is now illegal to have an allocator for an access type with Storage_Size equal to zero whereas in Ada 95 it raised Storage_Error on execution. It is always better to detect errors at compile time wherever possible. The reason for the change is to allow Pure units to use access types provided they do not use allocators. If the storage size is zero then this is now known at compile time. (AI-366, 4.8(20.g))

12 – The requirement that a partial view with available stream attributes be externally streamable can cause an incompatibility in extremely rare cases. This also relates to pragma Pure. (AI-366, 10.2.1(28.e))

13 – It is now illegal to use an incomplete view as a parameter or result of an access to subprogram type or as an access parameter of a primitive operation if the completion is deferred to the package body. See paper 3, section 2 for examples. (AI-326, 3.10.1(23.h, i))

14 – The specification of System.RPC can now be tailored for an implementation by adding further operations or by changing the profile of existing operations. If it is tailored in this way then an existing program might not compile in Ada 2005. See paper 6, section 7. (AI-273, E.5(30.a))

1.2 Inconsistencies with Ada 95

1 – The awkward situations regarding access types, discriminants and constraints discussed in paper 2 section 5, can also give rise to obscure inconsistencies.

Unconstrained aliased objects of types with discriminants with defaults are no longer constrained by their initial values. This means that a program that raised Constraint_Error in Ada 95 because of attempting to change the discriminants will no longer do so.

Thus consider item 4 in the previous section. We had

```
type Int_Ptr is access all Integer;
Obj: aliased T;                          -- mutable object
Dodgy: Int_Ptr := Obj.Comp'Access;       -- take care
...
Obj:= (Disc => True);                    -- Comp gone
```

We noted that in Ada 2005, the assignment statement to Dodgy is illegal because we cannot write Obj.Comp'Access. The assignment to Obj is itself permitted because we now know that there cannot be any dodgy pointers. Suppose that the assignment to Dodgy is removed. Then in Ada 95, the assignment to Obj will raise Constraint_Error but it will not in Ada 2005. It is extremely unlikely that any correct program relied upon this behaviour. (AI-363, 3.3.1(33.f) and 3.10(26.d))

A related situation applies with allocators where the allocated type is a private type with hidden discriminants. This is also illustrated by an earlier example where we had

```
with P;  use P;
procedure Do_It is
  A: T;
  B: T_Ptr := new T;
  C: T_Ptr := Evil;
begin
  A := Flip(A);
  B.all := Flip(B.all);                  -- Constraint_Error in Ada 95, not in 2005
```

```
   C.all := Flip(C.all);
end Do_It;
```

The assignment to B.**all** raises Constraint_Error in Ada 95 but not in Ada 2005 as explained above. Again it is extremely unlikely that any correct program relied upon this behaviour. (AI-363, 4.8(20.f))

2 – In Ada 2005 the categorization of certain wide characters is changed. As a consequence Wide_Character'Wide_Value and Wide_Character'Wide_Image will change in some rare situations. A further consequence is that for some subtypes S of Wide_Character the value of S'Wide_Width is different. But the value of Wide_Character'Wide_Width itself is not changed. (AI-285, 3.5.2(9.h) and AI-395, 3.5.2(9.i, j))

3 – There is an interesting analogy to incompatibility number 2 which concerns adding further entities to existing predefined packages. If we add further entries to Standard itself then an inconsistency is possible. Thus if an additional entity More is added to the package Standard and an existing program has a package P with an existing entity More and a use clause for P then, in Ada 2005, references to More will now be to that in Standard and not that in P. In the most unlikely event that the program remains legal, it will behave differently. The only such identifiers added to Standard are Wide_Wide_Character and Wide_Wide_String so this is extremely unlikely. (AI-285, 3.5.2(9.k) and 3.6.3(8.g))

4 – Access discriminants and non-controlling access parameters no longer exclude null in Ada 2005. A program that passed null to these will behave differently.

The usual situation is that Constraint_Error will be raised within the subprogram when an attempt to dereference is made rather than at the point of call. If the subprogram has no handler for Constraint_Error then the final effect will be much the same.

But clearly it is possible for the behaviour to be quite different. For example, the access value might not be dereferenced or the subprogram might have a handler for Constraint_Error which does something unusual. And there might even be a pragma Suppress for the check in which case the program will become erroneous.

See paper 2, section 2 for an example. (AI-231, 3.10(26.c))

5 – The lower bound of strings returned by functions Expanded_Name and External_Name (and wide versions) in Ada.Tags are defined to be 1 in Ada 2005. Ada 95 did not actually define the value and so if an implementation has chosen to return some other lower bound such as 77 then the program might behave differently. (AI-417, 3.9(33.c)) See also 2.2 item 4 below.

6 – The upper bound of the range of Year_Number in Ada 2005 is 2399 whereas it was 2099 in Ada 95. See paper 6, section 3. (AI-351, 9.6(40.e))

2 Retrospective changes to Ada 95

In the course of the development of Ada 2005, a number of small changes were deemed to apply also to Ada 95 and thus were classified as binding interpretations rather than amendments. Accordingly they are not (generally) covered by the changes discussed in the previous papers. Note however, that AI-241 on exceptions was discussed in paper 5 even though it was eventually classified as a binding interpretation. Moreover, AI-329 on exceptions was split and the part stating that Raise_Exception never returns (also applying to Ada 95) was formed into AI-446.

AI-438 adds subprograms Read_Exception_Occurrence and Write_Exception_Occurence plus corresponding attribute definition clauses for streams to the package Ada.Exceptions thus

```
procedure Read_Exception_Occurrence
   (Stream: not null access Root_Stream_Type'Class; Item: out Exception_Occurrence);
```

```
procedure Write_Exception_Occurrence
   (Stream: not null access Root_Stream_Type'Class; Item: in Exception_Occurrence);

for Exception_Occurrence'Read use Read_Exception_Occurrence;

for Exception_Occurrence'Write use Write_Exception_Occurrence;
```

These attributes enable the type Exception_Occurrence to be streamed. Note that this is a limited type and so streaming is only possible if predefined. A survey of other existing and new predefined limited types showed that no others needed to be treated in this way.

No other retrospective AIs actually affect the specification of any units but typically add or correct a number of rules. Of these some are of special interest because they introduce minor incompatibilities or inconsistencies. They are

108 Inheritance of stream attributes for type extensions

(108 was actually in the 2001 Corrigendum)

133 Controlling bit ordering

195 Streams (this covers many issues regarding streams)

220 Subprograms withing private compilation units

225 Aliased current instance for limited types

229 Accessibility rules and generics

238 Lower bound of Ada.Strings.Bounded_Slice

240 Stream attributes for limited types in Annex E

242 Surprise behavior of Update

246 Conversions between arrays of a by-reference type

253 Pragmas Attach_Handler and Interrupt_Handler

268 Rounding of real static expressions

279 Tag read by T'Class'Input

283 Truncation of stream files by Close and Reset

306 Class-wide extension aggregate expressions

341 Primitive subprograms are frozen with a tagged type

360 Types that need finalization

377 Naming of generic child packages

378 The bounds of Ada.Exceptions.Exception_Name

403 Preelaboration checks and formal objects

435 Storage pools for access-to-subprogram types

446 Raise_Exception for Null_Id

These are briefly discussed in the following subsections.

2.1 Incompatibilities with original Ada 95

There are a small number of incompatibilities between the original Ada 95 and that resulting from various corrections.

1 – A limited type can become nonlimited. Applying the Access or Unchecked_Access attribute to the current instance of such a type is now illegal. (AI-225, 3.10(26.e))

This is fairly obscure. Remember that the current instance rule is about referring to a type within its own declaration such as

```
type Strange is limited
  record
    Me: access Strange := Strange'Unchecked_Access;
    ...
  end record;
```

This is fine. It only makes sense to permit the attribute if the type is limited. But a type can be limited by virtue of having a limited component. for example

```
type Limp is limited private;

type Strange is
  record
    Me: access Strange := Strange'Unchecked_Access;
    C: Limp;
  end record;
```

If the component is limited private and it turns out that the full type of the component is not limited after all then the enclosing type becomes nonlimited. In such a case the attribute is now not allowed. The cure is to make the enclosing type explicitly limited.

2 – Conversions between unrelated array types that are limited or (for view conversions) might be by-reference types are now illegal. This is because they might not have the same representation and they cannot be copied in order to change the representation. (AI-246, 4.6(71.j))

3 – The meaning of a record representation clause and the storage place attributes for the non-default bit order is now clarified. One consequence is that the equivalence of bit 1 in word 1 to bit 9 in word 0 for a machine with Storage_Unit = 8 no longer applies for the non-default order. (AI-133, 13.5.1 (31.d) and 13.5.2(5.c))

4 – Various new freezing rules were added in order to fix a number of holes in the original rules for Ada 95. (AI-341, 13.14(20.p))

5 – The type Unbounded_String is defined to need finalization. If the partition has No_Nested_Finalization and moreover the implementation of Unbounded_String does not have a controlled part then it will not be allowed in local objects now although it was in original Ada 95. Clearly this is extremely unlikely. (AI-360, A.4.5(88.b)). The same applies to the type Generator in Numerics.Float_Random and Discrete_Random (AI-360, A.5.2(61.a)) and to File_Type in Sequential_IO (AI-360, A.8.1(17.b)), Direct_IO (AI-360, A.8.4(20.a)), Text_IO (AI-360, A.10.1(86.c)) and Stream_IO (AI-360, A.12.1(36.b)). See also D.7(22.a).

This problem is unlikely with types such as Unbounded_String which were introduced into Ada 95 at the same time as controlled types and thus are almost inevitably implemented in terms of controlled types. It is more likely with the file types that existed in Ada 83 since some implementations might not have changed them to use controlled types.

6 – It is now illegal to apply the Access attribute to a subprogram declared in the specification of a generic unit in the body of that unit. The usual workaround applies which is to move the use of the attribute to the private part. (AI-229, 3.10.2(41.f))

7 – It is now illegal for the ancestor expression in an extended aggregate to be of a class wide type or to be dispatching call (probably most readers would never dream of doing that anyway). Thus if we have tagged type T and a type NT extended from it and we declare

```
X: T'Class := ... ;
```

then the aggregate

```
NT'(X with ... )                    -- illegal
```

is illegal. We have to use a type conversion and write

```
NT'(T(X) with ... )                 -- legal
```

Similarly the ancestor part cannot be a dispatching call such as F(X) where the function F is

```
function F(Y: T) return T is
begin
  return Y;
end F;
...
NT'(F(X) with ... )                 -- illegal since X class wide
```

Again it can be fixed by a suitable conversion to a specific type. (AI-306, 4.3.2((13.b))

8 – If a generic library unit and an instance of it both have child units with the same name then they now hide each other. Thus

```
generic package G is ... ;          -- a generic G

generic package G.C is ... ;        -- a child C

with G;
package I is new G;                 -- the instance

package I.C is ... ;                -- child of instance

with G.C;  with I.C;                -- illegal, both hidden
package P ...
```

Originally it seems that this was allowed but it was not specified which package C would refer to. This was fairly foolish and confusing. (AI-377, 8.3(29.z))

9 – A subprogram body acting as a declaration (that is without a distinct specification) cannot with a private child. This was allowed by mistake originally and permitted the export of types declared in private child packages. (AI-220, 10.1.2(31.f)

10 – For the purposes of deciding whether a unit can be preelaborable a generic formal object is nonstatic. (AI-403, 10.2.1(28.f))

11 – Storage pools (and the attribute Storage_Size) are not permitted for access to subprogram types. Originally it looked as if they were allowed provided they were never used (or the size was zero). (AI-435, 13.11(43.d))

12 – The rules for the two pragmas Interrupt _Handler and Attach_Handler are the same with respect to where they are permitted. Originally it appeared that Interrupt_Handler could be declared in a place remote from the subprogram it was referring to. (AI-253, C.3.1(25.a))

13 – There are some changes regarding attributes in remote type and RCI units. These changes primarily concern streams for limited types. (AI-240, E.2.2(18.a), E.2.3(20.b))

2.2 Inconsistencies with original Ada 95

There are a small number of inconsistencies between the original Ada 95 and that resulting from various corrections.

1 – The function Exception_Identity applied to the value Null_Occurrence now returns Null_Id whereas it originally raised Constraint_Error in Ada 95. See paper 5, section 2. (AI-241, 11.4.1(19.y))

2 – The procedure Raise_Exception applied to the value Null_Id now raises Constraint_Error whereas it originally did nothing (and thus returned). See paper 5, section 4. (AI-466, 11.4.1(19.aa))

3 – Rounding of static real expressions is now implementation-defined whereas it was originally defined as away from zero. The reason for the change is to match the behaviour of the hardware; this also means that static and non-static expressions are more likely to get the same answer which is comforting. (AI-268, 4.9(44.s))

4 – The lower bounds of strings returned by functions Exception_Name, Exception_Message, and Exception_
Information (and wide versions) are now defined to be 1. (AI-378, 417, 11.4.1(19.z))

Similarly the bounds of the various functions Slice are now defined. (AI-238, A.4.4(106.e))

5 – There are some changes regarding stream attributes. (AI-108, 13.13.2(60.g) and AI-195, 13.13.2(60.h))

6 – There are changes regarding truncation of stream files. (AI-283, A.12.1(36.a))

7 – There is a potential inconsistency regarding the use of Internal_Tag outside of streaming. However, there was an implementation permission to do as is now required and so programs were not portable anyway. (AI-279, 3.9(33.b))

8 – The procedure Update in Interfaces.C.Strings no longer adds a nul character. (AI-242, B.3.1(60.a))

3 Unfinished topics

A number of topics which seemed to be good ideas initially were abandoned for various reasons. Usually the reason was simply that a good solution could not be produced in the time available and the trouble with a bad solution is that it is hard to put it right later. In other cases it is now felt that the topic deserved further consideration in the light of better understanding; sometimes there was fairly general agreement that the current situation was not ideal and ought to be improved, nevertheless there was no agreement on what should be done. And in some cases the good idea seemed a bad idea after further discussion.

So it might be that when Ada is next revised these further features might be reconsidered and so perhaps this section might be called forthcoming attractions. But on the other hand maybe other matters will need to be dealt with in the light of user experience with Ada 2005.

The following subsections briefly outline the main topics – for a fuller discussion, consult the text of the Ada Issue concerned.

3.1 Aggregates for private types (AI- 389)

The <> notation was introduced for aggregates to mean the default value if any. See paper 3 section 4. A curiosity is that we can write

```
type Secret is private;

type Visible is
  record
```

```
    A: Integer;
    S: Secret;
  end record;

X: Visible := (A => 77; S => <>);
```

but we cannot write

```
S: Secret := <>;                          -- illegal
```

The argument is that this would be of little use since the components take their default values anyway.

For uniformity AI-389 proposed allowing

```
S: Secret := (others => <>);
```

for private types and also for task and protected types. One advantage would be that we could then write

```
S: constant Secret := (others => <>);
```

whereas at the moment it is not possible to declare a constant of a private type because we are unable to give an initial value.

However, discussion of this issue lead into a quagmire concerning the related AI-413 and in the end both were abandoned.

3.2 Partial generic instantiation (AI-359)

Certain attempts to use signature packages lead to circularities. The AI outlines the following example

```
generic
  type Element is private;
  type Set is private;
  with function Union(L, R: Set) return Set  is <>;
  with function Intersection(L, R: Set) return Set is <>;
  ... -- and so on
package Set_Signature is end;
```

Remember that a signature is a generic package consisting only of a specification. When we instantiate it, the effect is to assert that the actual parameters are consistent and the instantiation provides a name to refer to them as a group.

If we now attempt to write

```
generic
  type Elem is private;
  with function Hash(E: Elem) return Integer;
package Hashed_Sets is
  type Set is private;
  function Union(L, R: Set) return Set;
  function Intersection(L, R: Set) return Set;
  ...
  package Signature is new Set_Signature(Elem, Set);
private
  type Set is
    record
      ...
```

```
        end record;
    end Hashed_Sets;
```

then we are in trouble. The problem is that the instantiation of Set_Signature tries to freeze the type Set prematurely.

Other similar examples concern the use of access types with private types. The essence of the problem is that we want to instantiate a package with a private type before the full declaration of that type.

The solution proposed was to split an instantiation into two parts, a partial instantiation and a full (that is, normal) instantiation. The partial instantiation might take the form

```
package P is new G(Private_Type) with private;
```

and this can be done with the partial view of the type. The full instantiation can then be given after the full declaration of the type.

This fell by the wayside at the last minute largely because of fears that awkward situations might be introduced inadvertently.

3.3 Support for IEEE 559: 1989 (AI-315)

The proposal was to provide full support for all aspects of IEEE 559 arithmetic such as Nans (a Nan is Not A Number). This would have necessitated adding attributes such as S'Infinity, S'Is_Nan, S'Finite and so on plus a package Ada.Numerics.IEC_559.

The proposal was abandoned because it would have had a big impact on implementers and it was not clear that there was sufficient demand.

3.4 Defaults for generic parameters (AI-299)

Generic subprogram parameters and object parameters of mode in can have defaults. But other parameters such as packages and types cannot. This was considered irksome and untidy and efforts were made to define a suitable notation for all possible generic parameters.

However, it was abandoned partly because an appropriate syntax seemed hard to find and more importantly, it was not felt to be that important.

3.5 Pre/post-conditions for subprograms (AI-288)

This proposal was to add pragmas such as Pre_Assert and Post_Assert. Thus in the case of a subprogram Push on a type Stack we might write

```
procedure Push(S: in out Stack; X: in Item);
pragma Pre_Assert(Push, not Is_Full(S));
pragma Post_Assert(Push, not Is_Empty(S));
```

These pragmas would be controlled by the pragma Assertion_Policy which controls the pragma Assert (which was of course incorporated into Ada 2005). Optional message parameters were allowed as well.

The general idea was that when the procedure Push was called, the expression Is_Full(S) would be evaluated and if this were false then action would be taken as for an Assert pragma. Note that the key difference from assert is that the pragmas go on the subprogram specification whereas to use Assert it would have to be placed in the body.

There were other pragmas for dispatching subprograms and so this was not quite so simple as at first appeared.

The proposal was abandoned for a number of reasons. There were more important matters to deal with and we were running out of time. Moreover, it seemed just the sort of topic where user

experience on a trial implementation would be helpful in deciding what was required. And there was some feeling that since this was all dynamic it was not helpful to the high integrity community where the emphasis was on static analysis and proof.

3.6 Type and package invariants (AI-375)

This defined further pragmas similar to those in the previous proposal (AI-288) but concerned with packages and types. Thus the pragma Package_Invariant identified a function returning a Boolean result. This function would be implicitly called after the call of each subprogram in the package and if the result were false the behaviour would be as for an Assert pragma that failed.

This proposal was abandoned for the same reasons as AI-288.

3.7 Exceptions as types (AI-264)

This AI originally arose out of a workshop organized by Ada-Europe. The proposal was quite complex and considered far too radical a change and probably expensive to implement. As a consequence it was slimmed down considerably. But having been slimmed down it seemed pointless and was then abandoned. The only part to survive was the idea of raise with message which became a separate AI and was incorporated into Ada 2005.

3.8 Sockets operations (AI-292)

This seemed a very good idea at the time but no detailed proposal was forthcoming and so it died.

3.9 In out parameters for functions (AI-323)

This is a really interesting topic. Ada functions are curious. On the one hand they look as if they are going to be well behaved since they only allow in parameters and thus it appears as if they cannot have side effects. But of course they can have any side effects they like by using global variables! And parameters can be access types and nothing prevents the accessed values from being changed. Indeed access parameters are a sort of sly way of getting in out parameters anyway.

The proposal was to allow functions to have parameters of all modes. The rationale for the proposal is well summarized in the problem part of the AI thus "Ada functions can have arbitrary side effects, but are not allowed to announce that in their specifications".

Clearly, Ada functions are indeed curious. But strangely, this AI was abandoned quite early in the revision process on the grounds that it was "too late". (Perhaps too late in this context meant 25 years too late.) In any event there was no agreement on a way forward since there are strong arguments both ways. But there was agreement that time would be better spent discussing and agreeing other matters.

One suggestion is that two kinds of functions should be supported. Absolutely pure side-effect free functions that merely deliver the value of some state. Functions in SPARK [1] are like this. And the other sort of function could be one that is just like a procedure and can do anything and have all modes of parameters but for convenience returns a result which can then be used in an expression.

It is interesting to note that Preliminary Ada [2] had value returning procedures as well as functions. The functions were pure but value returning procedures were much as current functions and could have side effects. But value returning procedures could not have out and in out parameters. The difference between the two was thus not enough and so pure functions were dropped and value returning procedures became functions.

This topic may deserve to be revisited at some time.

3.10 Application defined scheduling (AI-358)

The International Real-Time Ada Workshops have been a source of suggestions for improvements to Ada. The Workshop at Oporto suggested a number of further scheduling algorithms [3]. Most of

these such as Round Robin and EDF have been included in Ada 2005. But that for application defined scheduling was not.

The reason is perhaps that it was felt desirable to see how those that had been included worked out before adding yet more burden for implementers.

4 Acknowledgements

This is the last of the papers in this series and so this seems a good moment to once more thank all those who have helped by reviewing various drafts and pointing out where I had gone astray. I am especially grateful to Randy Brukardt, Pascal Leroy and Tucker Taft for their diligence and patience.

I must also thank Ada-Europe and the Ada Resource Association and also the British Standards Institute for financial support for attending various meetings.

Writing this rationale has been a learning experience for me and I trust that readers will also have found the material useful in learning about Ada 2005. An integrated description of Ada 2005 as a whole including some further examples will be found in a forthcoming version of the textbook [4].

References

[1] J. G. P. Barnes (2003) *High Integrity Software – The SPARK Approach to Safety and Security*, Addison-Wesley.

[2] ACM (1979) *Preliminary Ada Reference Manual*, Sigplan Notices, Vol. 14, No. 6.

[3] ACM (2003) *Proceedings of the 12th International Real-Time Ada Workshop,* Ada Letters, Vol 32, No 4.

[4] J. G. P. Barnes (2006) *Programming in Ada 2005*, Addison-Wesley.