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.