Rationale for Ada 2005

John Barnes

6.5 Generic units

There are a number of improvements in the area of generics many of which have already been outlined in earlier chapters.

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 includes null then the actual parameter must also include 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 Section 3.2 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 chapter on the object oriented model (see 2.4), 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 chapter (see 2.4) 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 Ada 95
with procedure Q( ... ) is Some_Proc;     -- OK in Ada 95
with procedure R( ... ) is null;     -- only in Ada 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 Section 2.6 of the chapter 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 Section 2.4 of the chapter 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, video display, historic 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 chapter (see 5.3), 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 Section 5.3 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 Section 4.5. 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 Chapter 8. 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 2.7. 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.