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.