John Barnes

Ada Rationale 2012

Structure and Visibility

Courtesy of AdaCore

Rationale for Ada 2012: 3 Structure and visibility

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

Abstract

This paper describes various improvements in the areas of structure and visibility for Ada 2012.

Perhaps the most amazing change is that functions may now have parameters of all modes. In earlier versions of Ada, functions could only have parameters of mode ***in*** *and so could not change variables explicitly passed as parameters; however, they could silently manipulate global variables in any way whatsoever. In order to ameliorate any risks of foolishness with this new freedom, there are new rules regarding order dependence.*

There are also important improvements to incomplete types which make them much more useful; these include completion by a private type, their use as parameters and a new form of generic parameter.

Other improvements include a new form of use clause and changes to extended return statements.

Keywords: rationale, Ada 2012.

1 Overview of changes

The WG9 guidance document [1] does not specifically identify problems in this area other than through a general exhortation to remedy shortcomings.

The following Ada Issues cover the relevant changes and are described in detail in this paper:

15 Constant return objects

19 Primitive subprograms are frozen with a tagged type

32 Extended return statements for class-wide functions

53 Aliased views of unaliased objects

142 Explicitly aliased parameters

143 In out parameters for functions

144 Detecting dangerous order dependencies

150 Use all type clause

151 Incomplete types as parameters and result

162 Incomplete types completed by partial views

213 Formal incomplete types

214 Default discriminants for limited tagged types

235 Accessibility of explicitly aliased parameters

277 Aliased views of extended return objects

296 Freezing of subprograms with incomplete parameters

These changes can be grouped as follows.

First there is the exciting business of allowing parameters of all modes for functions (143) and the associated rules to prevent certain order dependences (144). Another change concerning parameters is permitting explicitly aliased parameters (142, 235).

There are then a number of improvements in the area of incomplete types (151, 162) including the ability to permit them as formal generic parameters (213, 296). There are also related changes to the freezing rules (19).

There is also a minor change regarding discriminants (214).

The existing two forms of use clause (use package clause and use type clause) are augmented by a third form: the use all type clause (150).

Finally, there are a number of changes (corrections really) to extended return statements which were introduced in Ada 2005 (15, 32, 277). An associated change is the introduction of the idea of an immutably limited type (53).

2 Subprogram parameters

The main topic here is the fact that functions (but not operators) in Ada 2012 can have parameters of any mode.

This is a topic left over from Ada 2005. The epilogue to the Rationale for Ada 2005 [2] discusses a number of topics that were abandoned and in the case of function modes says:

"Clearly, Ada functions are indeed curious. But strangely this AI (that is AI95-323) 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.)" It was not possible to agree on a way forward and so effort was devoted to other topics.

But mists clear with time. The big concern was that allowing parameters of all modes might open the door to dangerous programming practices but a solution to that was found in the introduction of stricter rules preventing many order dependences.

It is instructive to quickly go through the various historical documents.

A probably little known document is one written in 1976 by David Fisher of the Institute for Defense Analyses [3] which provided the foundation for the requirements for the development of a new language. It doesn't seem to distinguish between procedures and functions; it does mention the need for parameters which are constant and those which effectively rename a variable. Moreover, it does say (item C1 on page 81): *Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.* This does not actually require left-to-right evaluation but the behaviour must be as if it were. I have always thought it tragic that this was not observed.

This document was followed by a series known as Strawman, Woodenman, Tinman, Ironman [4] and finishing with Steelman [5].

The requirement on left-to-right evaluation remained in Tinman and was even stronger in Ironman but was somewhat weakened in Steelman to allow instrumentation and ends with a warning about being erroneous.

Further requirements are introduced in Ironman which requires both functions and procedures as we know them. Moreover, Ironman has a requirement about assignment to variables non-local to a function; they must be encapsulated in a region that has no calls on the function; this same requirement notes that it implies that functions can only have input parameters. This requirement does not seem to have carried forward to Steelman.

However, Ironman also introduces a requirement on restrictions to prevent aliasing. One is that the same actual parameter of a procedure cannot correspond to more than one input-output parameter. This requirement does survive into Steelman. But, it only seems to apply to procedures and not to functions and Steelman appears not to have noticed that the implied requirement that functions can only have input parameters has vanished.

It interesting to then see what was proposed in the sequence of languages leading to Ada 83, namely, Preliminary Green [6], Green [7], Preliminary Ada [8], and Ada [9]. Note that Preliminary Green was based on Ironman whereas Green was based on Steelman.

In Preliminary Green we find procedures and functions. Procedures can have parameters of three modes, **in**, **out** and **access** (don't get excited, **access** meant **in out**). Functions can only have parameters of mode **in**. Moreover,

> side effects to variables accessible at the function call are not allowed. In particular, variables that are global to the function body may not be updated in the function body. The rationale for Preliminary Green makes it quite clear that functions can have no side effects whatsoever.

In Green we find the three modes **in**, **out**, and **in out**. But the big difference is that as well as procedures and functions as in preliminary Green, there are now value returning procedures such as

> **procedure** Random **return** Real **range** –1.0 .. 1.0;

The intent is that functions are still free of all side effects whereas value returning procedures have more flexibility. However, value returning procedures can only have parameters of mode **in** and

> assignments to global variables are permitted within value returning procedures. Calls of such procedures are only valid at points of the program where the corresponding variables are not within the scope of their declaration. The order of evaluation of these calls is strictly that given in the text of the program. Calls to value returning procedures are only allowed in assignment statements, initializations and procedure calls.

The rationale for Green notes that if you want to instrument a function then use a pragma. It also notes that functions

> with arbitrary side effects would undermine the advantage of the functional approach to software. In addition it would complicate the semantics of all language structures where expressions involving such calls may occur. Hence this form of function is not provided.

And now we come to Ada herself. There are manuals dated July 1979 (preliminary Ada), July 1980 (draft mil-std), July 1982 (proposed ANSI standard), and January 1983 (the ANSI standard usually known as Ada 83).

In Preliminary Ada, we have procedures, functions and value returning procedures exactly as in Green. Indeed, it seems that the only difference between Green and Preliminary Ada is that the name Green has been converted to Ada.

But the 1980 Ada manual omits value returning procedures and any mention of any restrictions on what you can do in a function. And by 1982 we find that we are warned that parameters can be evaluated in any order and so on.

The Rationale for Ada 83 [10] didn't finally emerge until 1986 and discusses briefly the reason for the change which is basically that benevolent side effects are important. It concludes by quoting from a paper regarding Algol 60 [11]

> The plain fact of the matter is (1) that side-effects are sometimes necessary, and (2) programmers who are irresponsible enough to introduce side-effects unnecessarily will soon lose the confidence of their colleagues and rightly so.

However, an interesting remark in the Rationale for Ada 83 in the light of the change in Ada 2012 is

> The only limitation imposed in Ada on functions is that the mode of all parameters must be **in**: it would not be logical to allow **in out** and **out** parameters for functions in a language that excludes nested assignments within an expression.

Hmm. That doesn't really seem to follow. Allowing assignments in expressions as in C is obnoxious and one of the sources of errors in C programs. It is not so much that permitting side-effects in expressions via functions is unwise but more that treating the result of an assignment as a value

nested within an expression is confusing. Such nested constructions are naturally still excluded from Ada 2012 and so it is very unlikely that the change will be regretted.

Now we must turn to the question of order dependences. Primarily, to enable optimization, Ada does not define the order of evaluation of a number of constructions. These include

- the parameters in a subprogram or entry call,
- the operands of a binary operator,
- the destination and value in an assignment,
- the components in an aggregate,
- the index expressions in a multidimensional name,
- the expressions in a range,
- the barriers in a protected object,
- the guards in a select statement,
- the elaboration of library units.

The expressions involved in the above constructions can include function calls. Indeed, as AI-144 states "Arguably, Ada has selected the worst possible solution to evaluation order dependences (by not specifying an order of evaluation), it does not detect them in any way, and then says that if you depend upon one (even if by accident), your code will fail at some point in the future when your compiler changes. Something should be done about this."

It is far too late to do anything about specifying the order of evaluation so the approach taken is to prevent as much aliasing as possible since aliasing is an important cause of order of evaluation problems. Ada 2012 introduces rules for determining when two names are "known to denote the same object".

Thus they denote the same object if

- both names statically denote the same stand-alone object or parameter; or
- both names are selected components, their prefixes are known to denote the same object, and their selector names denote the same component.

and so on with similar rules for dereferences, indexed components and slices. There is also a rule about renaming so that if we have

```
C: Character renames S(5);
```

then C and S(5) are known to denote the same object. The index naturally has to be static.

A further step is to define when two names "are known to refer to the same object". This covers some cases of overlapping. Thus given a record R of type T with a component C, we say that R and R.C are known to refer to the same object. Similarly with an array A we say that A and A(K) are known to refer to the same object (K does not need to be static in this example).

Given these definitions we can now state the two basic restrictions.

The first concerns parameters of elementary types:

- For each name N that is passed as a parameter of mode **in out** or **out** to a call of a subprogram S, there is no other name among the other parameters of mode **in out** or **out** to that call of S that is known to denote the same object.

Roughly speaking this comes down to saying two or more parameters of mode **out** or **in out** of an elementary type cannot denote the same object. This applies to both functions and procedures.

This excludes the example given in the Introduction which was

```
procedure Do_It(Double, Triple: in out Integer) is
begin
  Double := Double * 2;
  Triple := Triple * 3;
end Do_It;
```

with

```
Var: Integer := 2;
...
Do_It(Var, Var);                    -- illegal in Ada 2012
```

The key problem is that parameters of elementary types are always passed by copy and the order in which the parameters are copied back is not specified. Thus Var might end up with either the value of Double or the value of Triple.

The other restriction concerns constructions which have several constituents that can be evaluated in any order and can contain function calls. Basically it says:

- If a name N is passed as a parameter with mode **out** or **in out** to a function call that occurs in one of the constituents, then no other constituent can involve a name that is known to refer to the same object.

Constructions cover many situations such as aggregates, assignments, ranges and so on as listed earlier.

This rule excludes the other example in the Introduction, namely, the aggregate

```
(Var, F(Var))                       -- illegal in Ada 2012
```

where F has an **in out** parameter.

The rule also excludes the assignment

```
Var := F(Var);                      -- illegal
```

if the parameter of F has mode **in out**. Remember that the destination of an assignment can be evaluated before or after the expression. So if Var were an array element such as A(I) then the behaviour could vary according to the order. To encourage good practice, it is also forbidden even when Var is a stand-alone object.

Similarly, the procedure call

```
Proc(Var, F(Var));                  -- illegal
```

is illegal if the parameter of F has mode **in out**. Examples of overlapping are also forbidden such as

```
ProcA(A, F(A(K));                   -- illegal

ProcR(R, F(R.C));                   -- illegal
```

assuming still that F has an **in out** parameter and that ProcA and ProcR have appropriate profiles because, as explained above, A and A(K) are known to refer to the same object as are R and R.C.

On the other hand

```
Proc(A(J), F(A(K));                 -- OK
```

is permitted provided that J and K are different objects because this is only a problem if J and K happen to have the same value.

For more details the reader is referred to the AI. The intent is to detect situations that are clearly troublesome. Other situations that might be troublesome (such as if J and K happen to have the same value) are allowed, since to prevent them would make many programs illegal that are not actually dubious. This would cause incompatibilities and upset many users whose programs are perfectly correct.

The other change in Ada 2012 concerning parameters is that they may be explicitly marked **aliased** thus

```
procedure P(X: aliased in out T; ... );
```

As a consequence within P we can write X'Access. Recall that tagged types were always considered implicitly aliased anyway and always passed by reference. If the type T is a by-copy type such as Integer, then adding **aliased** causes it to be passed by reference. (So by-copy types are not always passed by copy!)

The possibility of permitting explicitly aliased function results such as

```
function F( ... ) return aliased T;          -- illegal Ada 2012
```

was considered but this led to difficulties and so was not pursued.

The syntax for parameter specification is modified thus

```
parameter_specification ::=
   defining_identifier_list: [aliased] mode [null exclusion] subtype_mark
                                                          [:= default_expression]
 | defining_identifier_list: access_definition [:= default_expression]
```

showing that **aliased** comes first as it does in all contexts where it is permitted.

The rules for mode conformance are modified as expected. Two profiles are only mode conformant if both or neither are explicitly marked as aliased. Although adding **aliased** for a tagged type parameter makes little difference since tagged types are implicitly aliased, if this is done for a subprogram declaration then it must be done for the corresponding body as well.

There are (of course) rules regarding accessibility; these are much as expected although a special case arises in function return statements. As usual, if the foolish programmer does something silly, the compiler will draw attention to the error.

Explicitly aliased parameters were largely introduced to overcome problems in the container library. Examples will be given in the paper addressing containers.

3 Incomplete types

Incomplete types in Ada 83 were very incomplete. They were mostly used for the traditional linked list such as

```
type Cell;                          -- incomplete
type Cell_Ptr is access Cell;

type Cell is                        -- the completion
  record
    Next: Cell_Ptr;
    Element: Pointer;
  end record;
```

The incomplete type could only be used in the declaration of an access type. Moreover, the incomplete declaration and its completion had to be in the same list of declarations. However, if the

incomplete declaration is in a private part then the completion can be deferred to the body; this is the so-called Taft Amendment added to Ada 83 at the last minute.

Ada 95 introduced tagged types and generalized access types and so made the language much more flexible but made no changes to incomplete types as such. However, it soon became clear that the restrictive nature of incomplete types was a burden regarding mutually dependent types and was a key issue in the requirements for Ada 2005.

The big step forward in Ada 2005 was the introduction of the limited with clause. This enables a package to have an incomplete view of a type in another package and solves many problems of mutually recursive types.

However, the overall rule remained that an incomplete type could only be completed by a full type declaration and, moreover, a parameter could not (generally) be of an incomplete type. This latter restriction encouraged the use of access parameters.

As mentioned in the Introduction, the first rule prevented the following

```
type T1;
type T2 (X: access T1) is private;
type T1 (X: access T2) is private;          -- illegal in Ada 2005
```

since the completion of T1 could not be by a private type.

This is changed in Ada 2012 so that an incomplete type can be completed by any type (other than another incomplete type). Note especially that an incomplete type can be completed by a private extension as well as by a private type.

The other major problem in Ada 2005 was that with mutually dependent types in different packages we could not use incomplete types as parameters because it was not known whether they were by-copy or by-reference. Of course, if they were tagged then we did know they were by reference but that was a severe restriction.

The need to know whether parameters are by reference or by copy was really a red herring. The model used for parameter passing in versions of Ada up to and including Ada 2005 was basically that at the point of the declaration of a subprogram we need to have all the information required to call the subprogram. Thus we needed to know how to pass parameters and so whether they were by reference or by copy.

But this is quite unnecessary; we don't need to know the mechanisms involved until a point where the subprogram is actually called or the body itself is encountered since it is only at those points that the parameter mechanism is really required. It is only at those points that the compiler has to grind out the code for the call or for the body.

So the rules in Ada 2012 are changed to use this "when we need to know" model. This is discussed in AI-19 which is actually a binding interpretation and thus retrospectively applies to Ada 2005 as well. This is formally expressed by the difference between freezing a subprogram and freezing its profile. This was motivated by a problem with tagged types whose details need not concern us.

As a highly benevolent consequence, we are allowed to use incomplete types as both parameters and function results provided that they are fully defined at the point of call and at the point where the body is defined.

But another consequence of this approach is that we cannot defer the completion of an incomplete type declared in a private part to the corresponding body. In other words, parameters of an incomplete type are allowed provided the Taft Amendment is not used for completing the type.

The other exciting change regarding incomplete types is that in Ada 2012 they are allowed as generic parameters. In Ada 2005 the syntax is

```
formal_type_declaration ::=
        type defining_identifier [discriminant_part] is formal_type_definition ;
```

whereas in Ada 2012 we have

```
formal_type_declaration ::=
        formal_complete_type_declaration
      | formal_incomplete_type_declaration

formal_complete_type_declaration ::=
        type defining_identifier [discriminant_part] is formal_type_definition ;

formal_incomplete_type_declaration ::=
        type defining_identifier [discriminant_part] [is tagged] ;
```

So the new kind of formal generic parameter has exactly the same form as the declaration of an incomplete type. It can be simply **type** T; or can require that the actual be tagged by writing **type** T **is tagged**; – and in both cases a discriminant can be given.

A formal incomplete type can then be matched by any appropriate incomplete type. If the formal specifies **tagged**, then so must the actual. If the formal does not specify **tagged** then the actual might or might not be tagged. Of course, a formal incomplete type can also be matched by an appropriate complete type. And also, in all cases, any discriminants must match as well.

An example of the use of a formal incomplete type occurs in the package Ada.Iterator_Interfaces whose generic formal part is

```
generic
  type Cursor;
  with function Has_Element(Position: Cursor) return Boolean;
package Ada.Iterator_Interfaces is ...
```

The formal type Cursor is incomplete and can be matched by an actual incomplete type. The details of this package will be described in a later paper.

Another example is provided by a signature package as mentioned in the Introduction. We can write

```
generic
  type Element;
  type Set;
  with function Empty return Set is <>;
  with function Unit(E: Element) return Set is <>;
  with function Union(S, T: Set) return Set is <>;
  with function Intersection(S, T: Set) return Set is <>;
  ...
package Set_Signature is end;
```

Such a signature generic can be instantiated with an actual set type and then the instance can be passed into other generics that have a formal package such as

```
generic
  type VN is private;
  type VN_Set is private;
  with package Sets is
    new Set_Signature(Element => VN, Set => VN_Set, others => <>);
  ...
package Analyse is ...
```

This allows the construction of a generic that needs a Set abstraction such as a flow analysis package. Remember that the purpose of a signature is to group several entities together and to check that various relationships hold between the entities. In this case the relationships are that the types Set and Element do have the various operations Empty, Unit and so on.

The set generic could be included in a set container package thus

```
generic
  type Element is private;
package My_Sets is
  type Set is tagged private;

  function Empty return Set;
  function Unit(E: Element) return Set;
  function Union(S, T: Set) return Set;
  function Intersection(S, T: Set) return Set;
  ...
  package My_Set is new Set_Signature(Element, Set);
private
  ...
end My_Sets;
```

The key point is that normally an instantiation freezes a type passed as a generic parameter. But in the case of a formal incomplete untagged type, this does not happen. Hence the actual in the instantiation of Set_Signature in the generic package My_Sets can be a private type such as Set.

This echoes back to the earlier discussion of changing the freezing rules. We cannot call a subprogram with untagged incomplete parameters (whether formal or not) because we do not know whether they are to be passed by copy or by reference. But we can call a subprogram with tagged incomplete parameters because we do know that they are passed by reference (and this has to remain true for compatibility with Ada 2005). So just in case the actual subprogram in the tagged case is called within the generic, the instantiation freezes the profile. But in the untagged case, we know that the subprogram cannot be called and so there is no need to freeze the profile.

This means that the type Set should not be given as tagged incomplete in the package Set_Signature since we could not then use the signature in the package My_Sets.

If a subprogram has both tagged and untagged formal incomplete parameters then the untagged incomplete parameters win and the subprogram cannot be called.

(If this is all too confusing, do not worry, the compiler will moan at you if you make a mistake.)

Another rule regarding incomplete formal types is that the controlling type of a formal abstract subprogram cannot be incomplete.

4 Discriminants

There is one minor change in this area which was mentioned in the Introduction.

In Ada 2005, a discriminant can only have a default if it is not tagged. But in Ada 2012, a default is also permitted in the case of a limited tagged type.

Ada typically uses defaults as a convenience so that in many cases standard information can be omitted. Thus it is convenient that the procedure New_Line has a default of 1 since it would be boring to have to write New_Line(1); all the time.

In the case of discriminants however, a default as in

```
type Polynomial(N: Index := 0) is
  record
    A: Integer_Vector(0 .. N);
  end record;
```

also indicates that the type is mutable. This means that the value of the discriminant of an object of the type can be changed by a whole record assignment. However, tagged types in Ada 2005 never have defaults because we do not want tagged types to be mutable. On the other hand, if a tagged type is limited then it is immutable anyway. And so it was concluded that there is no harm in permitting a limited tagged type to have a default discriminant.

This may seem rather academic but the problem arose in designing containers for queues. It was felt desirable that the protected type Queue should have a discriminant giving its ceiling priority and that this should have a default for convenience. As illustrated in the Introduction this resulted in a structure as follows

```
generic
  with package Queue_Interfaces is new ...
  Default_Ceiling: Any_Priority := Priority'Last;
package AC.Unbounded_Synchronized_Queues is
  ...
  protected type Queue(Ceiling: Any_Priority := Default_Ceiling)
           with Priority => Ceiling
         is new Queue_Interfaces.Queue with ...
```

Now the problem is that a protected type such as Queue which is derived from an interface is considered to be tagged because interfaces are tagged. On the other hand a protected type is always limited and its discriminant provides a convenient way of providing the ceiling priority. So there was a genuine need for a change to the rule.

Note incidentally that the default is itself provided with the default value of Priority'Last since it is a generic parameter with its own default.

5 Use clauses

Ada 2012 introduces a further form of use clause. In order to understand the benefit it is perhaps worth just recalling the background to this topic.

The original use clause in Ada 83 made everything in a package directly visible. Consider the following package

```
package P is
  I, J, K: Integer;

  type Colour is (Red, Orange, Yellow, Green, Blue, ... );
  function Mix(This, That: Colour) return Colour;

  type Complex is
    record
      Rl, Im: Float;
    end record;
  function "+"(Left, Right: Complex) return Complex;
  ...
end P;
```

Now suppose we have a package Q which manipulates entities declared in P. We need a with clause for P, thus

```
with P;
package Q is ...
```

With just a with clause for P we have to refer to entities in P using the prefix P. So we get statements and declarations in Q such as

```
P.I := P.J + P.K;

Mucky: P.Colour := P.Mix(P.Red, P.Green);

W: P.Complex := (1.0, 2.0);
Z: P.Complex := (4.0, 5.0);
D: P.Complex := P."+"(W, Z);
```

This is generally considered tedious especially if the package name is not P but A_Very_Long_Name. However, adding a package use clause to Q thus

```
with P; use P;
package Q ...
```

enables the P prefix to be omitted and in particular allows infix notation for operators so we can now simply write

```
D: Complex := W + Z;
```

But as is well known, the universal use of such use clauses introduces ambiguity (if the same identifier is in two different packages and we have a use clause for both), obscurity (you can't find the wretched declaration of Red) and possibly a maintenance headache (another package is added which duplicates some identifiers). So there is a school of thought that use clauses are bad for you.

However, although the prefix denoting the package is generally beneficial it is a pain to be forced to always use the prefix notation for operators. So in Ada 95, the use type clause was added enabling us to write

```
with P;  use type P.Complex;
package Q is ...
```

This has the effect that only the primitive *operators* of the type Complex are directly visible. So we can now write

```
D: P.Complex := W + Z;
```

Note that the type name Complex is not itself directly visible so we still have to write P.Complex in the declaration of D.

However, some users still grumbled. Why should only those primitive operations that happen to be denoted by operators be visible? Why indeed? Why cannot Mucky be declared similarly without using the prefix P for Mix, Red and Green?

It might be worth briefly recalling exactly which operations of a type T are primitive operations of T. They are basically

- predefined operations such as "=" and "+",
- subprograms declared in the same package as T and which operate on T,
- enumeration literals of T,
- for a derived type, inherited or overridden subprograms.

The irritation is solved in Ada 2012 by the **use all type** clause which makes all primitive operations visible. (Note another use for the reserved word **all**.)

So we can write

```
with P;  use all type P.Colour;
package Q is ...
```

and now within Q we can write

```
Mucky: P.Colour := Mix(Red, Green);
```

Thus the enumeration literals such as Red are made directly visible as well as obvious primitive subprograms such as Mix.

Another impact concerns tagged types and in particular operations on class wide types.

Remember that subprograms with a parameter (or result) of type T'Class are not primitive operations unless they also have a parameter (or result of type T) as well.

Actually it is usually very convenient that operations on a class wide type are not primitive operations because it means that they are not inherited and so cannot be overridden. Thus we are assured that they do apply to all types of the class.

So, suppose we have

```
package P is
  type T is tagged private;
  procedure Op1(X: in out T);
  procedure Op2(Y: in T; Z: out T);
  function Fop(W: T) return Integer;
  procedure List(TC: in T'Class);
private
  ...
end P;
```

Then although List is not a primitive operation of T it will certainly look to many users that it belongs to T in some broad sense. Accordingly, writing **use all type** P.T; makes not only the primitive operations such as Op1, Op2 and Fop, visible but it also makes List visible as well.

Note that this is the same as the rule regarding the prefixed form of subprogram calls which can also be used for both primitive operations and class wide operations. Thus given an object A of type T, as well as statements A.Op1; and A.Op2(B); and a function call A.Fop we can equally write

```
A.List;                    -- prefixed call of class wide procedure
```

Moreover, suppose we declare a type NT in a package NP thus

```
package NP is
  type NT is new T with ...
  ...
end NP;
```

If we have an object AN of type NT then not only can we use prefixed calls for inherited and overridden operations but we can also use prefixed calls for class wide operations in ancestor packages such as P. So we can write

```
AN.List;                   -- prefixed call of List in ancestor package
```

Similarly, writing **use all type** NP.NT; on Q makes the inherited (or overridden) operations Op1, Op2 and Fop visible and also makes the class wide operation List declared in P visible. We do not also have to write **use all type** P.T; on Q as well.

We conclude by remarking that the maintenance problem of name clashes really only applies to use package clauses. In the case of use type and use all type clauses, the entities made visible are overloadable and a clash only occurs if two have the same profile which is very unlikely and almost inevitably indicates a bug.

6 Extended return statements

The final topic in this paper is the extended return statement. This was introduced in Ada 2005 largely to solve problems with limited types. However, some glitches have come to light and these are corrected in Ada 2012.

A description of the reasons for and general properties of the extended return statement will be found in [2].

The syntax for extended return statement in Ada 2005 as found in [12] is

```
extended_return_statement ::=
        return defining_identifier: [aliased] return_subtype_indication [:= expression] [do
          handled_sequence_of_statements
        end return] ;
```

Before going further, it should be mentioned that there was some confusion regarding limited types and so the term immutably limited was introduced in the course of the maintenance of Ada 2005. There were various problems. Basically, limitedness is a property of a view of a type. Thus even in Ada 83 a private type might be limited but the full view found in the private part would not be limited. Ada 95 introduced explicitly limited types. Ada 2005 introduced coextensions and these could even include such obviously limited things as task types thus adding a limited part to what was otherwise a seemingly nonlimited type. It became clear that it was necessary to introduce a term which meant that a type was really and truly limited and could not subsequently become nonlimited for example in a private part or in a child unit. So a type is immutably limited if

- it is an explicitly limited record type,
- it is a task type, protected type or synchronized interface,
- it is a non-formal limited private type that is tagged or has an access discriminant with a default expression,
- it is derived from an immutably limited type.

It was then realised that there were problems with extended return statements containing an explicit **aliased**. Consequently, it was decided that there was really no need for **aliased** if there was a rule that immutably limited return objects were implicitly aliased. So **aliased** was removed from the syntax. However, some users had already written **aliased** and this would have introduced an irritating incompatibility. So finally it was decided that **aliased** could be written but only if the type were immutably limited.

Another small problem concerned constants. Thus we might write

```
return X: T do
  ...                                -- compute X
end return;
```

However, especially in the case of a limited type LT, we might also give the return object an initial value, thus

```
return X: LT := (A, B, C) do
  ...                                -- other stuff
end return;
```

Now it might be that although the type as a whole is limited one or more of its components might not be and so could be manipulated in the sequence of statements. But if we want to ensure that this does not happen, it would be appropriate to indicate that X were constant. But, almost surely by an oversight, we cannot do that since it is not permitted by the syntax. So the syntax needed changing to permit the addition of constant.

To aid the description the syntax in Ada 2012 is actually written as two productions as follows

```
extended_return_object_declaration ::=
        defining_identifier: [aliased] [constant] return_subtype_indication [:= expression]

extended_return_statement ::=
        return extended_return_object_declaration [do
          handled_sequence_of_statements
        end return] ;
```

The other small change to the extended return statement concerns the subtype give in the profile of the function and that in the extended return statement itself. The result type of the function can be constrained or unconstrained but that given in the extended return statement must be constrained.

This can be illustrated by considering array types. (These examples are from [2].) Suppose we have

```
type UA is array (Integer range <>) of Float;
subtype CA is UA(1 .. 10);
```

then we can write

```
function Make( ... ) return CA is
begin
  ...
  return R: UA(1 .. 10) do          -- statically matches
    ...
  end return;
end Make;
```

This is allowed because the subtypes statically match.

If the subtype in the function profile is unconstrained then the result must be constrained either by its subtype or by its initial value. For example

```
function Make( ... ) return UA is
begin
  ...
  return R: UA(1 .. N) do
    ...
  end return;
end Make;
```

and here the result R is constrained by its subtype. A similar situation can arise with records with discriminants. Thus we can have

```
type Person(Sex: Gender) is ... ;

function F( ... ) return Person is
begin
  if ... then
    return R: Person(Sex => Male) do
      ...
    end return;
```

```
  else
    return R: Person(Sex => Female) do
      ...
    end return;
  end if;
end F;
```

which shows that we have the possibility of returning a person of either gender.

However, what is missing from Ada 2005 is that we can have analogous situations with tagged types in that a function might wish to return a value of some type in a class.

So we would like to write things such as

```
function F( ... ) return Object'Class is
begin
  if ... then
    return C: Circle do
      ...
    end return;
  elsif ... then
    return S: Square do
      ...
    end return;
   end if;
end F;
```

This is not permitted in Ada 2005 which required the types to be the same. This can be overcome by writing

```
return C: Object'Class := Circle_Func do
  ...
end return;
```

where Circle_Func is some local function that returns the required object of type Circle.

This is all rather irksome so the wording is changed in Ada 2012 to say that in this situation the subtype in the extended return statement need not be the same as that in the profile but simply must be covered by it. There are also related slight changes to the accessibility rules.

References

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

[2] John Barnes (2008) *Ada 2005 Rationale*, LNCS 5020, Springer-Verlag.

[3] David Fisher (1976) *A Common Programming Language for the Department of Defense – Background and Technical Requirements*, Institute for Defense Analyses, Arlington, Virginia.

[4] Defense Advanced Research Projects Agency (1977) *Department of Defense Requirements for High Order Computer Programming Languages – Revised IRONMAN*, USDoD.

[5] Defense Advanced Research Projects Agency (1978) *Department of Defense Requirements for High Order Computer Programming Languages – STEELMAN*, USDoD.

[6] Jean Ichbiah et al (1978) *Preliminary Reference Manual for the Green Programming Language*, Honeywell Inc.

[7] Jean Ichbiah et al (1979) *Reference Manual for the Green Programming Language*, Honeywell Inc.

[8] ACM (1979) *Preliminary Ada Reference Manual*, Sigplan Notices, Vol 14, No 6.

[9] ANSI / Mil–Std 1815A (1983) *Ada Reference Manual.*

[10] Jean Ichbiah, John Barnes, Robert Firth, Mike Woodger (1986) *Rationale for the Design of the Ada Programming Language*, Honeywell & Alsys.

[11] B Higman (1963) *What everybody should know about Algol*, Computer Journal, vol 6, no 1, pp 50-56.

[12] S. T. Taft et al (eds) (2007) *Ada 2005 Reference Manual*, LNCS 4348, Springer-Verlag.