Rationale for Ada 2005: 3 Structure and visibility

John Barnes

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

Abstract

This paper describes various improvements in the areas of structure and visibility for Ada 2005.

The most important improvement is perhaps the introduction of limited with clauses which permit types in two packages to refer to each other. A related addition to context clauses is the private with clause which just provides access from a private part.

There are also important improvements to limited types which make them much more useful; these include initialization with aggregates and composition using a new form of return statement.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 2, June 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] identifies the solution of the problem of mutually dependent types as one of the two specific issues that need to be addressed in devising Ada 2005.

Moreover the guidance document also emphasizes

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

OO is largely about structure and visibility and so further improvements and in particular those that remedy shortcomings are desirable.

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

217 Mutually recursive types – limited with

262 Access to private units in the private part

287 Limited aggregates allowed

318 Limited and anonymous access return types

326 Tagged incomplete types

412 Subtypes and renamings of incomplete entities

These changes can be grouped as follows.

First there is the important solution to the problem of mutually dependent types across packages provided by the introduction of limited with clauses (217). Related changes are the introduction of tagged incomplete types (326) and the ability to have subtypes and renamings of incomplete views (412).

Another improvement to the visibility rules is the introduction of private with clauses (262).

There are some changes to aggregates. These were triggered by problems with limited types but apply to aggregates in general (part of 287).

An important area is that of limited types which are somewhat confused in Ada 95. There are two changes which permit limited values to be built *in situ*. One is the use of aggregates for initialization and the other is a more elaborate return statement which enables the construction of limited values when returning from a function (287, 318).

2 Mutually dependent types

For many programmers the solution of the problem of mutually dependent types will be the single most important improvement introduced in Ada 2005.

This topic was discussed in the Introduction using an example of two mutually dependent types, Point and Line. Each type needed to refer to the other in its declaration and of course the solution to this problem is to use incomplete types. In Ada 95 there are three stages. We first declare the incomplete types

```
type Point;                     -- incomplete types
type Line;
```

Suppose for simplicity that we wish to study patterns of points and lines such that each point has exactly three lines through it and that each line has exactly three points on it. (This is not so stupid. The two most fundamental theorems of projective geometry, those of Pappus and Desargues, concern such structures and so does the simplest of finite geometries, the Fano plane.)

Using the incomplete types we can then declare

```
type Point_Ptr is access Point;   -- use incomplete types
type Line_Ptr is access Line;
```

and finally we can complete the type declarations thus

```
type Point is                   -- complete the types
  record
    L, M, N: Line_Ptr;
  end record;

type Line is
  record
    P, Q, R: Point_Ptr;
  end record;
```

Of course, in Ada 2005, as discussed in the previous paper, we can use anonymous access types more freely so that the second stage can be omitted in this example. As a consequence the complete declarations are simply

```
type Point is                   -- complete the types
  record
    L, M, N: access Line;
  end record;

type Line is
  record
```

```
      P, Q, R: access Point;
    end record;
```

This has the important advantage that we do not have to invent irritating identifiers such as Point_Ptr.

But we will stick to Ada 95 for the moment. In Ada 95 there are two rules

- the incomplete type can only be used in the definition of access types;
- the complete type declaration must be in the same declarative region as the incomplete type.

The first rule does actually permit

```
type T;
type A is access procedure (X: in out T);
```

Note that we are here using the incomplete type T for a parameter. This is not normally allowed, but in this case the procedure itself is being used in an access type. The additional level of indirection means that the fact that the parameter mechanism for T is not known yet does not matter.

Apart from this, it is not possible to use an incomplete type for a parameter in a subprogram in Ada 95 except in the case of an access parameter. Thus we cannot have

```
function Is_Point_On_Line(P: Point; L: Line) return Boolean;
```

before the complete type declarations.

It is also worth pointing out that the problem of mutually dependent types (within a single unit) can often be solved by using private types thus

```
  type Point is private;
  type Point_Ptr is access Point;
  type Line is private;
  type Line_Ptr is access Line;
private

  type Point is
    record
      L, M, N: Line_Ptr;
    end record;

  type Line is
    record
      P, Q, R: Point_Ptr;
    end record;
```

But we need to use incomplete types if we want the user to see the full view of a type so the situation is somewhat different.

As an aside, remember that if an incomplete type is declared in a private part then the complete type can be deferred to the body (this is the so-called Taft Amendment in Ada 83). In this case neither the user nor indeed the compiler can see the complete type and this is the main reason why we cannot have parameters of incomplete types whereas we can for private types.

We will now introduce what has become a canonical example for discussing this topic. This concerns employees and the departments of the organization in which they work. The information about employees needs to refer to the departments and the departments need to refer to the employees. We assume that the material regarding employees and departments is quite large so that we naturally wish to declare the two types in distinct packages Employees and Departments. So we would like to say

```
with Departments;  use Departments;
package Employees is
  type Employee is private;
  procedure Assign_Employee(E: in out Employee; D: in out Department);
  type Dept_Ptr is access all Department;
  function Current_Department(E: Employee) return Dept_Ptr;
  ...
end Employees;

with Employees;  use Employees;
package Departments is
  type Department is private;
  procedure Choose_Manager(D: in out Department; M: in out Employee);
  ...
end Departments;
```

We cannot write this because each package has a with clause for the other and they cannot both be declared (or entered into the library) first.

We assume of course that the type Employee includes information about the Department for whom the Employee works and the type Department contains information regarding the manager of the department and presumably a list of the other employees as well – note that the manager is naturally also an Employee.

So in Ada 95 we are forced to put everything into one package thus

```
package Workplace is
  type Employee is private;
  type Department is private;
  procedure Assign_Employee(E: in out Employee; D: in out Department);
  type Dept_Ptr is access all Department;
  function Current_Department(E: Employee) return Dept_Ptr;
  procedure Choose_Manager(D: in out Department; M: in out Employee);
private
  ...
end Workplace;
```

Not only does this give rise to huge cumbersome packages but it also prevents us from using the proper abstractions. Thus the types Employee and Department have to be declared in the same private part and so are not protected from each other's operations.

Ada 2005 solves this by introducing a variation of the with clause – the limited with clause. A limited with clause enables a library unit to have an incomplete view of all the visible types in another package. We can now write

```
limited with Departments;
package Employees is
  type Employee is private;
  procedure Assign_Employee(E: in out Employee; D: access Departments.Department);
  type Dept_Ptr is access all Departments.Department;
  function Current_Department(E: Employee) return Dept_Ptr;
  ...
end Employees;
```

```
limited with Employees;
package Departments is
  type Department is private;
  procedure Choose_Manager(D: in out Department; M: access Employees.Employee);
  ...
end Departments;
```

It is important to understand that a limited with clause does not impose a dependence. Thus if a package A has a limited with clause for B, then A does not depend on B as it would with a normal with clause, and so B does not have to be compiled before A or placed into the library before A.

If we have a cycle of packages we only have to put **limited with** on one package since that is sufficient to break the cycle of dependences. However, for symmetry, in this example we have made them both have a limited view of each other.

Note the terminology: we say that we have a limited view of a package if the view is provided through a limited with clause. So a limited view of a package provides an incomplete view of its visible types. And by an incomplete view we mean as if they were incomplete types.

In the example, because an incomplete view of a type cannot generally be used as a parameter, we have had to change one parameter of each of Assign_Employee and Choose_Manager to be an access parameter.

There are a number of rules necessary to avoid problems. A natural one is that we cannot have both a limited with clause and a normal with clause for the same package in the same context clause (a normal with clause is now officially referred to as a nonlimited with clause). An important and perhaps unexpected rule is that we cannot have a use package clause with a limited view because severe surprises might happen.

To understand how this could be possible it is important to realise that a limited with clause provides a very restricted view of a package. It just makes visible

- the name of the package and packages nested within,
- an incomplete view of the types declared in the visible parts of the packages.

Nothing else is visible at all. Now consider

```
package A is
  X: Integer := 99;
end A;

package B is
  X: Integer := 111;
end B;

limited with A, B;
package P is
  ...                                    -- neither X visible here
end P;
```

Within package P we cannot access A.X or B.X because they are not types but objects. But we could declare a child package with its own with clause thus

```
with A;
package P.C is
  Y: Integer := A.X;
end P.C;
```

The nonlimited with clause on the child "overrides" the limited with clause on the parent so that A.X is visible.

Now suppose we were allowed to add a use package clause to the parent package; since a use clause on a parent applies to a child this means that we could refer to A.X as just X within the child so we would have

```
limited with A, B;
use A, B;                     -- illegal
package P is
  ...                         -- neither X visible here
end P;

with A;
package P.C is
  Y: Integer := X;            -- A.X now visible as just X
end P.C;
```

If we were now to change the with clause on the child to refer to B instead of A, then X would refer to B.X rather than A.X. This would not be at all obvious because the use clause that permits this is on the parent and we are not changing the context clause of the parent at all. This would clearly be unacceptable and so use package clauses are forbidden if we only have a limited view of the package.

Here is a reasonably complete list of the rules designed to prevent misadventure when using limited with clauses

- a use package clause cannot refer to a package with a limited view as illustrated above,

  ```
  limited with P;  use P;     -- illegal
  package Q is ...
  ```

  the rule also prevents

  ```
  limited with P;
  package Q is
    use P;                    -- illegal
  ```

- a limited with clause can only appear on a specification – it cannot appear on a body or a subunit,

  ```
  limited with P;             -- illegal
  package body Q is ...
  ```

- a limited with clause and a nonlimited with clause for the same package may not appear in the same context clause,

  ```
  limited with P; with P;     -- illegal
  ```

- a limited with clause and a use clause for the same package or one of its children may not appear in the same context clause,

  ```
  limited with P; use P.C;    -- illegal
  ```

- a limited with clause may not appear in the context clause applying to itself,

  ```
  limited with P;             -- illegal
  package P is ...
  ```

- a limited with clause may not appear on a child unit if a nonlimited with clause for the same package applies to its parent or grandparent etc,

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

limited with Q;                  -- illegal
package P.C is ...
```

but note that the reverse is allowed as mentioned above

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

with Q;                          -- OK
package P.C is ...
```

- a limited with clause may not appear in the scope of a use clause which names the unit or one of its children,

```
with A;
package P is
  package R renames A;
end P;

with P;
package Q is
  use P.R;                       -- applies to A
end Q;

limited with A;                  -- illegal
package Q.C is ...
```

  without this specific rule, the use clause in Q which actually refers to A would clash with the limited with clause for A.

Finally note that a limited with clause can only refer to a package declaration and not to a subprogram, generic declaration or instantiation, or to a package renaming.

We will now return to the rules for incomplete types. As mentioned above the rules for incomplete types are quite strict in Ada 95 and apart from the curious case of an access to subprogram type it is not possible to use an incomplete type for a parameter other than in an access parameter.

Ada 2005 enables some relaxation of these rules by introducing tagged incomplete types. We can write

```
type T is tagged;
```

and then the complete type must be a tagged type. Of course the reverse does not hold. If we have just

```
type T;
```

then the complete type T might be tagged or not.

A curious feature of Ada 95 was mentioned in the Introduction. In Ada 95 we can write

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

By using the attribute Class, this promises in a rather sly way that the complete type T will be tagged. This is strictly obsolescent in Ada 2005 and moved to Annex J. In Ada 2005 we should write

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

The big advantage of introducing tagged incomplete types is that we know that tagged types are always passed by reference and so we are allowed to use tagged incomplete types for parameters.

This advantage extends to the incomplete view obtained from a limited with clause. If a type in a package is visibly tagged then the incomplete view obtained is tagged incomplete and so the type can then be used for parameters.

Returning to the packages Employees and Departments it probably makes sense to make both types tagged since it is likely that the types Employee and Department form a hierarchy. So we can write

```
limited with Departments;
package Employees is
  type Employee is tagged private;
  procedure Assign_Employee(E: in out Employee; D: in out Departments.Department'Class);
  type Dept_Ptr is access all Departments.Department'Class;
  function Current_Department(E: Employee) return Dept_Ptr;
  ...
end Employees;

limited with Employees;
package Departments is
  type Department is tagged private;
  procedure Choose_Manager(D: in out Department; M: in out Employees.Employee'Class);
  ...
end Departments;
```

The text is a bit cumbersome now with Class sprinkled liberally around but we can introduce some subtypes in order to shorten the names. We can also avoid the introduction of the type Dept_Ptr since we can use an anonymous access type for the function result as mentioned in the previous paper. So we get

```
limited with Departments;
package Employees is
  type Employee is tagged private;
  subtype Dept is Departments.Department;
  procedure Assign_Employee(E: in out Employee; D: in out Dept'Class);
  function Current_Department(E: Employee) return access Dept'Class;
  ...
end Employees;

limited with Employees;
package Departments is
  type Department is tagged private;
  subtype Empl is Employees.Employee;
  procedure Choose_Manager(D: in out Department; M: in out Empl'Class);
  ...
end Departments;
```

Observe that in Ada 2005 we can use a simple subtype as an abbreviation for an incomplete type thus

```
subtype Dept is Departments.Department;
```

but such a subtype cannot have a constraint or a null exclusion. In essence it is just a renaming. Remember that we cannot have a use clause with a limited view. Moreover, many projects forbid use clauses anyway but permit renamings and subtypes for local abbreviations. It would be a pain if such abbreviations were not also available when using a limited with clause.

It's a pity we cannot also write

```
subtype A_Dept is Departments.Department'Class;
```

but then you cannot have everything in life.

A similar situation arises with the names of nested packages. They can be renamed in order to provide an abbreviation.

The mechanism for breaking cycles of dependences by introducing limited with clauses does not mean that the implementation does not check everything thoroughly in a rigorous Ada way. It is just that some checks might have to be deferred. The details depend upon the implementation.

For the human reader it is very helpful that use clauses are not allowed in conjunction with limited with clauses since it eliminates any doubt about the location of types involved. It probably helps the poor compilers as well.

Readers might be interested to know that this topic was one of the most difficult to solve satisfactorily in the design of Ada 2005. Altogether seven different versions of AI-217 were developed. This chosen solution is on reflection by far the best and was in fact number 6.

A number of loopholes in Ada 95 regarding incomplete types are also closed in Ada 2005.

One such loophole is illustrated by the following (this is Ada 95)

```
package P is
  ...
private
  type T;                              -- an incomplete type
  type ATC is access all T'Class;      -- it must be tagged
  X: ATC;
  procedure Op(X: access T);           -- primitive operation
  ...
end P;
```

The incomplete type T is declared in the private part of the package P. The access type ACT is then declared and since it is class wide this implies that the type T must be tagged (the reader will recall from the discussion above that this odd feature is banished to Annex J in Ada 2005). The full type T is then declared in the body. We also declare a primitive operation Op of the type T in the private part.

However, before the body of P is declared, nothing in Ada 95 prevents us from writing a private child thus

```
private package P.C is
  procedure Naughty;
end P.C;

package body P.C is
  procedure Naughty is
  begin
    Op(X);                     -- a dispatching call
  end Naughty;
end P.C;
```

and the procedure Naughty can call the dispatching operation Op. The problem is that we are required to compile this call before the type T is completed and thus before the location of its tag is known.

This problem is prevented in Ada 2005 by a rule that if an incomplete type declared in a private part has primitive operations then the completion cannot be deferred to the body.

Similar problems arise with access to subprogram types. Thus, as mentioned above, Ada 95 permits

```
type T;
type A is access procedure (X: in out T);
```

In Ada 2005, the completion of T cannot be deferred to a body. Nor can we declare such an access to subprogram type if we only have an incomplete view of T arising from a limited with clause.

Another change in Ada 2005 can be illustrated by the Departments and Employees example. We can write

```
limited with Departments;
package Employees is
  type Employee is tagged private;
  procedure Assign_Employee(E: in out Employee; D: in out Departments.Department'Class);
  type Dept_Ptr is access all Departments.Department'Class;
  ...
end Employees;

with Employees;  use Employees;
procedure Recruit(D: Dept_Ptr; E: in out Employee) is
begin
  Assign_Employee(E, D.all);
end Recruit;
```

Ada 95 has a rule that says "thou shalt not dereference an incomplete type". This would prevent the call of Assign_Employee which is clearly harmless. It would be odd to require Recruit to have a nonlimited with clause for Departments to allow the call of Assign_Employee. Accordingly the rule is changed in Ada 2005 so that dereferencing an incomplete view is only forbidden when used as a prefix as, for example, in D'Size.

3 Visibility from private parts

Ada 95 introduced public and private child packages in order to enable subsystems to be decomposed in a structured manner. The general idea is that

- public children enable the decomposition of the view of a subsystem to the user of the subsystem,
- private children enable the decomposition of the implementation of a subsystem.

In turn both public and private children can themselves have children of both kinds. This has proved to work well in most cases but a difficulty has arisen regarding private parts.

Recall that the private part of a package really concerns the implementation of the package rather than specifying the facilities to the external user. Although it does not concern algorithmic aspects of the implementation it does concern the implementation of data abstraction. During the original design of Ada some thought was given to the idea that a package should truly be written and compiled as three distinct parts. Perhaps like this

```
with ...
package P is
```

```
   ...                                     -- visible specification
end;

with ...
package private P is                       -- just dreaming
   ...                                     -- private part
end;

with ...
package body P is
   ...                                     -- body
end;
```

Each part could even have had its own context clause as shown.

However, it was clear that this would be an administrative nightmare in many situations and so the two-part specification and body emerged with the private part lurking at the end of the visible part of the specification (and sharing its context clause).

This was undoubtedly the right decision in general. The division into just two parts supports separate compilation well and although the private part is not part of the logical interface to the user it does provide information about the physical interface and that is needed by the compiler.

The problem that has emerged is that the private part of a public package cannot access the information in private child packages. Private children are of course not visible to the user but there is no reason why they should not be visible to the private part of a public package provided that somehow the information does not leak out. Thus consider a hierarchy

```
package App is
   ...
private
   ...
end App;

package App.Pub is
   ...
private
   ...
end App.Pub;

private package App.Priv is
   ...
private
   ...
end App.Priv;
```

There is no reason why the private parts of App and App.Pub and the visible part of the specification of App.Priv should not share visibility (the private part of App.Priv logically belongs to the next layer of secrecy downwards). But this sharing is not possible in Ada 95.

The public package App.Pub is not permitted to have a with clause for the child package App.Priv since this would mean that the visible part of App.Pub would also have visibility of this information and by mechanisms such as renaming could pass it on to the external user.

The specification of the parent package App is also not permitted to have a with clause for App.Priv since this would break the dependence rules anyway. Any child has a dependence on its parent and so the parent specification has to be compiled or entered into the program library first.

Note that the private part of the public child App.Pub does automatically have visibility of the private part of the parent App. But the reverse cannot be true again because of the dependence rules.

Finally note that the private child App.Priv can have a with clause for its public sibling App.Pub (it creates a dependence of course) but that only gives the private child visibility of the visible part of the public child.

So the only visibility sharing among the three regions in Ada 95 is that the private part of the public child and the visible part of the private child can see the private part of the parent.

The practical consequence of this is that in large systems, information which should really be lower down the hierarchy has to be placed in the private part of the ultimate parent. This tends to mean that the parent package becomes very large thereby making maintenance more difficult and forcing frequent recompilations of the parent and thus the whole hierarchy of packages.

The situation is much alleviated in Ada 2005 by the introduction of private with clauses.

If a package P has a private with clause for a package Q thus

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

then the private part of P has visibility of the visible part of the package Q, whereas the visible part of P does not have visibility of Q and so visibility cannot be transmitted to a user of P. It is rather as if the with clause were attached to just the private part of P thus

```
package P is
  ...
with Q;                          -- we cannot write this
private
  ...
end P;
```

This echoes the three-part decomposition of a package discussed above.

A private with clause can be placed wherever a normal with clause for the units mentioned can be placed and in addition a private with clause which mentions a private unit can be placed on any of its parent's descendants.

So we can put a private with clause for App.Priv on App.Pub thereby permitting visibility of the private child from the private part of its public sibling. Thus

```
private with App.Priv;
package App.Pub is
  ...                            -- App.Priv not visible here
private
  ...                            -- App.Priv visible here
end App.Pub;
```

This works provided we don't run afoul of the dependence rules. The private with clause means that the public child has a dependence on the private child and therefore the private child must be compiled or entered into the program library first.

We might get a situation where there exists a mutual dependence between the public and private sibling in that each has a type that the other wants to access. In such a case we can use a limited private with clause thus

```
limited private with App.Priv;
package App.Pub is
```

```
     ...                              -- App.Priv not visible here
  private
     ...                              -- limited view of App.Priv here
  end App.Pub;
```

The child packages are both dependent on the parent package and so the parent cannot have with clauses for them. But a parent can have a limited with clause for a public child and a limited private with clause for a private child thus

```
  limited with App.Pub;  limited private with App.Priv;
  package App is
     ...                              -- limited view of App.Pub here
  private
     ...                              -- limited view of App.Priv here
  end App;
```

A simple example of the use of private with clauses was given in the Introduction. Here it is somewhat extended

```
  limited with App.User_View;  limited private with App.Secret_Details;
  package App is
     ...                              -- limited view of type Outer visible here
  private
     ...                              -- limited view of type Inner visible here
  end App;

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

  private with App.Secret_Details;
  package App.User_View is

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

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

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

In the previous section we observed that there were problems with interactions between use clauses, nonlimited with clauses, and limited with clauses. Those rules also apply to private with clauses where a private with clause is treated as a nonlimited with clause and a limited private with clause is treated as a limited with clause. In other words private is ignored for the purpose of those rules.

Moreover, we cannot place a package use clause in the same context clause as a private with clause (limited or not). This is because we would then expect it to apply to the visible part as well which would be wrong. However, we can always put a use clause in the private part thus

```
private with Q;
package P is
  ...                              -- Q not visible here
private
  use Q;
  ...                              -- use visibility of Q here
end P;
```

At the risk of confusing the reader it might be worth pointing out that strictly speaking the rules regarding private with are treated as legality rules rather than visibility rules. Here is an example which illustrates this subtlety and the dangers it avoids

```
package P is
  function F return Integer;
end P;

function F return Integer;

with P;
private with F;
package Q is
  use P;
  X: Integer := F;                 -- illegal
  Y: Integer := P.F;               -- legal
private
  Z: Integer := F;                 -- legal, calls the library F
end Q;
```

If we treated the rules regarding private with as pure visibility rules then the call of F in the declaration of X in the visible part would be a call of P.F. So moving the declaration of X to the private part would silently change the F being called – this would be nasty. We can always write the call of F as P.F as shown in the declaration of Y.

So the rules regarding private with are written to make entities visible but unmentionable in the visible part. In practice programmers can just treat them as visibility rules so that the entities are not visible at all which is how we have described them above.

A useful consequence of the unmentionable rather than invisible approach is that we can use the name of a package mentioned in a private with clause in a pragma in the context clause thus

```
private with P;  pragma Elaborate(P);
package Q is ...
```

Private with clauses are in fact allowed on bodies as well, in which case they just behave as a normal with clause. Another minor point is that Ada has always permitted several with clauses for the same unit in one context clause thus

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

To avoid complexity we similarly allow

```
with P;  private with P;
package Q is
```

and then the private with is ignored.

We have introduced private with clauses in this section as the solution to the problem of access to private children from the private part of the parent or public sibling. But they have other important uses. If we have

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

then we are assured that the package Q cannot inadvertently access P in the visible part and, in particular, pass on access to entities in P by renamings and so on. Thus writing **private with** provides additional documentation information which can be useful to both human reviewers and program analysis tools. So if we have a situation where a private with clause is all that is needed then we should use it rather than a normal with clause.

In summary, whereas in Ada 95 there is just one form of with clause, Ada 2005 provides four forms

```
with P;                    -- full view

limited with P;            -- limited view

private with P;            -- full view from private part

limited private with P;    -- limited view from private part
```

Finally, note that if a private with clause is given on a specification then it applies to the body as well as to the private part.

4 Aggregates

There are important changes to aggregates in Ada 2005 which are very useful in a number of contexts. These were triggered by the changes to the rules for limited types which are described in the next section, but it is convenient to first consider aggregates separately.

The main change is that the box notation <> is now permitted as the value in a named aggregate. The meaning is that the component of the aggregate takes the default value if there is one.

So if we have a record type such as

```
type RT is
  record
    A: Integer := 7;
    B: access Integer;
    C: Float;
  end record;
```

then if we write

```
X: RT := (A => <>, B => <>, C => <>);
```

then X.A has the value 7, X.B has the value **null** and X.C is undefined. So the default value is that given in the record type declaration or, in the absence of such an explicit default value, it is the default value for the type. If there is no explicit default value and the type does not have one either then the value is simply undefined as usual.

The above example could be abbreviated to

```
X: RT := (others => <>);
```

The obvious combinations are allowed

```
(A => <>, B => An_Integer'Access, C => 2.5)
(A => 3, others => <>)
(A => 3, B | C => <>)
```

The last two are the same. There is a rule in Ada 95 that if several record components in an aggregate are given the same expression using a **|** then they have to be of the same type. This does not apply in the case of <> because no typed expression is involved.

The <> notation is not permitted with positional notation. So we cannot write

```
(3, <>, 2.5)                    -- illegal
```

But we can mix named and positional notations in a record aggregate as usual provided the named components follow the positional ones, so the following are permitted

```
(3, B => <>, C => 2.5)
(3, others => <>)
```

A minor but important rule is that we cannot use <> for a component of an aggregate that is a discriminant if it does not have a default. Otherwise we could end up with an undefined discriminant.

The <> notation is also allowed with array aggregates. But in this case the situation is much simpler because it is not possible to give a default value for array components. Thus we might have

```
P: array (1.. 1000) of Integer := (1 => 2, others => <>);
```

The array P has its first component set to 2 and the rest undefined. (Maybe P is going to be used to hold the first 1000 prime numbers and we have a simple algorithm to generate them which requires the first prime to be provided.) The aggregate could also be written as

```
(2, others => <>)
```

Remember that **others** is permitted with a positional array aggregate provided it is at the end. But otherwise <> is not allowed with a positional array aggregate.

We can add **others** => <> even when there are no components left. This applies to both arrays and records.

The box notation is also useful with tasks and protected objects used as components. Consider

```
protected type Semaphore is ... ;

type PT is
  record
    Guard: Semaphore;
    Count: Integer;
    Finished: Boolean := False;
  end record;
```

As explained in the next section, we can now use an aggregate to initialize an object of a limited type. Although we cannot give an explicit initial value for a Semaphore we would still like to use an aggregate to get a coverage check. So we can write

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

Note that although we can use <> to stand for the value of a component of a protected type in a record we cannot use it for a protected object standing alone.

```
Sema: Semaphore := <>;          -- illegal
```

The reason is that there is no need since we have no coverage check to concern us and there could be no other reason for doing it anyway.

Similarly we can use <> with a component of a private type as in

```
type Secret is private;

type Visible is
  record
    A: Integer;
    S: Secret;
  end record;

X: Visible := (A => 77; S => <>);
```

but not when standing alone

```
S: Secret := <>;                    -- illegal
```

It would not have any purpose because such a variable will take any default value anyway.

We conclude by mentioning a small point for the language lawyer. Consider

```
function F return Integer;

type T is
  record
    A: Integer := F;
    B: Integer := 3;
  end record;
```

Writing

```
X: T := (A => 5, others => <>);     -- does not call F
```

is not quite the same as

```
X: T;                               -- calls F
...
X.A := 5;  X.B := 3;
```

In the first case the function F is not called whereas in the second case it is called when X is declared in order to default initialize X.A. If it had a nasty side effect then this could matter. But then programmers should not use nasty side effects anyway.

5 Limited types and return statements

The general idea of a limited type is to restrict the operations that a user can do on the type to just those provided by the author of the type and in particular to prevent the user from doing assignment and thus making copies of objects of the type.

However, limited types have always been a problem. In Ada 83 the concept of limitedness was confused with that of private types. Thus in Ada 83 we only had limited private types (although task types were inherently limited).

Ada 95 brought significant improvement by two changes. It allowed limitedness to be separated from privateness. It also allowed the redefinition of equality for all types whereas Ada 83 forbade this for limited types. In Ada 95, the key property of a limited type is that assignment is not predefined and cannot be defined (equality is not predefined either but it can be defined). The general idea of course is that there are some types for which it would be wrong for the user to be able to make copies of objects. This particularly applies to types involved in resource control and types implemented using access types.

However, although Ada 95 greatly improved the situation regarding limited types, nevertheless two major difficulties have remained. One concerns the initialization of objects and the other concerns the results of functions.

The first problem is that Ada 95 treats initialization as a process of assigning the initial value to the object concerned (hence the use of := unlike some Algol based languages which use = for initialization and := for assignment). And since initialization is treated as assignment it is forbidden for limited types. This means that we cannot initialize objects of a limited type nor can we declare constants of a limited type. We cannot declare constants because they have to be initialized and yet initialization is forbidden. This is more annoying in Ada 95 since we can make a type limited but not private.

The following example was discussed in the Introduction

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

Note that this type is explicitly limited (but not private) but its components are not limited. If we declare an object of type T in Ada 95 then we have to initialize the components (by assigning to them) individually thus

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

Not only is this annoying but it is prone to errors as well. If we add a further component D to the type T then we might forget to initialize it. One of the advantages of aggregates is that we have to supply all the components which automatically provides full coverage analysis.

This problem did not arise in Ada 83 because we could not make a type limited without making it also private and so the individual components were not visible anyway.

Ada 2005 overcomes the difficulty by stating that initialization by an aggregate is not actually assignment even though depicted by the same symbol. This permits

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

We should think of the individual components as being initialized individually *in situ* – an actual aggregated value is not created and then assigned.

The reader might recall that the same thing happens when an aggregate is used to initialize a controlled type; this was not as Ada 95 was originally defined but it was corrected in AI-83 and consolidated in the 2001 Corrigendum [2].

We can now declare a constant of a limited type as expected

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

Limited aggregates can be used in a number of other contexts as well

- as the default expression in a component declaration, so if we nest the type T inside some other type (which itself then is always limited – it could be explicitly limited but there is a general rule that a type is implicitly limited if it has a limited component) we might have

```
type Twrapper is
  record
    Tcomp: T := (0, False, 0.0);
  end record;
```

- as an expression in a record aggregate,

  so again using the type Twrapper as in

  ```
  XT: Twrapper := (Tcomp => (1, True, 1.0));
  ```

- as an expression in an array aggregate similarly,

  so we might have

  ```
  type Tarr is array (1 .. 5) of T;

  Xarr: Tarr := (1 .. 5 => (2, True, 2.0));
  ```

- as the expression for the ancestor part of an extension aggregate,

  so if TT were tagged as in

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

  type TTplus is new TT with
    record
      D: Integer;
    end record;
  ...
  XTT: TTplus := ((1, True, 1.0) with 2);
  ```

- as the expression in an initialized allocator,

  so we might have

  ```
  type T_Ptr is access T;
  XT_Ptr: T_Ptr;
  ...
  XT_Ptr := new T'(3, False, 3.0);
  ```

- as the actual parameter for a subprogram parameter of a limited type of mode in

  ```
  procedure P(X: in T);
  ...
  P((4, True, 4.0));
  ```

- similarly as the default expression for a parameter

  ```
  procedure P(X: in T := (4, True, 4.0));
  ```

- as the result in a return statement

```
function F( ... ) return T is
begin
  ...
  return (5, False, 5.0);
end F;
```

this really concerns the other major change to limited types which we shall return to in a moment.

- as the actual parameter for a generic formal limited object parameter of mode in,

```
generic
  FT: in T;
package P is ...
...
package Q is new P(FT => (7, True, 7.0));
```

The last example is interesting. Limited generic parameters were not allowed in Ada 95 at all because there was no way of passing an actual parameter because the generic parameter mechanism for an in parameter is considered to be assignment. But now the actual parameter can be passed as an aggregate. An aggregate can also be used as a default value for the parameter thus

```
generic
  FT: in T := (0, False, 0.0);
package P is ...
```

Remember that there is a difference between subprogram and generic parameters. Subprogram parameters were always allowed to be of limited types since they are mostly implemented by reference and no copying happens anyway. The only exception to this is with limited private types where the full type is an elementary type.

The change in Ada 2005 is that an aggregate can be used as the actual parameter in the case of a subprogram parameter of mode **in** whereas that was not possible in Ada 95.

Sometimes a limited type has components where an initial value cannot be given as in

```
protected type Semaphore is ... ;

type PT is
  record
    Guard: Semaphore;
    Count: Integer;
    Finished: Boolean := False;
  end record;
```

Since a protected type is inherently limited the type PT is also limited because a type with a limited component is itself limited. Although we cannot give an explicit initial value for a Semaphore, we would still like to use an aggregate to get the coverage check. In such cases we can use the box symbol <> as described in the previous section to mean use the default value for the type (if any). So we can write

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

The major rule that must always be obeyed is that values of limited types can never be copied. Consider nested limited types

```
type Inner is limited
  record
    L: Integer;
    M: Float;
  end record;

type Outer is limited
  record
    X: Inner;
    Y: Integer;
end record;
```

If we declare an object of type Inner

```
An_Inner: Inner := (L => 2, M => 2.0);
```

then we could not use An_Inner in an aggregate of type Outer

```
An_Outer: Outer := (X => An_Inner, Y => 3);          -- illegal
```

This is illegal because we would be copying the value. But we can use a nested aggregate as mentioned earlier

```
An_Outer: Outer := (X => (2, 2.0), Y => 3);
```

The other major change to limited types concerns returning values from functions.

We have seen that the ability to initialize an object of a limited type with an aggregate solves the problem of giving an initial value to a limited type provided that the type is not private.

Ada 2005 introduces a new approach to returning the results from functions which can be used to solve this and other problems.

We will first consider the case of a type that is limited such as

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

We can declare a function that returns a value of type T provided that the return does not involve any copying. For example we could have

```
function Init(X: Integer; Y: Boolean; Z: Float) return T is
begin
  return (X, Y, Z);
end Init;
```

This function builds the aggregate in place in the return expression and delivers it to the location specified where the function is called. Such a function can be called from precisely those places listed above where an aggregate can be used to build a limited value in place. For example

```
V: T := Init(2, True, 3.0);
```

So the function itself builds the value in the variable V when constructing the returned value. Hence the address of V is passed to the function as a sort of hidden parameter.

Of course if T is not private then this achieves no more than simply writing

```
V: T := (2, True, 3.0);
```

But the function Init can be used even if the type is private. It is in effect a constructor function for the type. Moreover, the function Init could be used to do some general calculation with the parameters before delivering the final value and this brings considerable flexibility.

We noted that such a function can be called in all the places where an aggregate can be used and this includes in a return expression of a similar function or even itself

```
function Init_True(X: Integer; Z: Float) return T is
begin
  return Init(X, True, Z);
end Init_True;
```

It could also be used within an aggregate. Suppose we have a function to return a value of the limited type Inner thus

```
function Make_Inner(X: Integer; Y: Float) return Inner is
begin
  return (X, Y);
end Make_Inner;
```

then not only could we use it to initialize an object of type Inner but we could use it in a declaration of an object of type Outer thus

```
An_Inner: Inner := Make_Inner(2, 2.0);
An_Outer: Outer := (X => Make_Inner(2, 2.0), Y => 3);
```

In the latter case the address of the component of An_Outer is passed as the hidden parameter to the function Make_Inner.

Being able to use a function in this way provides much flexibility but sometimes even more flexibility is required. New syntax permits the final returned object to be declared and then manipulated in a general way before finally returning from the function.

The basic structure is

```
function Make( ... ) return T is
begin
  ...
  return R: T do                    -- declare R to be returned
    ...                     -- here we can manipulate R in the usual way
    ...                     -- in a sequence of statements
  end return;
end Make;
```

The general idea is that the object R is declared and can then be manipulated in an arbitrary way before being finally returned. Note the use of the reserved word **do** to introduce the statements in much the same way as in an accept statement. The sequence ends with **end return** and at this point the function passes control back to where it was called. Note that if the function had been called in a construction such as the initialization of an object X of a limited type T thus

```
X: T := Make( ... );
```

then the variable R inside the function is actually the variable X being initialized. In other words the address of X is passed as a hidden parameter to the function Make in order to create the space for R. No copying is therefore ever performed.

The sequence of statements could have an exception handler

```
return R: T do
  ...                            -- statements
exception
  ...                            -- handlers
end return;
```

If we need local variables within an extended return statement then we can declare an inner block in the usual way

```
return R: T do
  declare
    ...                          -- local declarations
  begin
    ...                          -- statements
  end;
end return;
```

The declaration of R could have an initial value

```
return R: T := Init( ... ) do
  ...
end return;
```

Also, much as in an accept statement, the **do** ... **end return** part can be omitted, so we simply get

```
return R: T;
```

or

```
return R: T := Init( ... );
```

which is handy if we just want to return the object with its default or explicit initial value.

Observe that extended return statements cannot be nested but could have simple return statements inside

```
return R: T := Init( ... ) do
  if ... then
    ...
    return;                      -- result is R
  end if;
  ...
end return;
```

Note that simple return statements inside an extended return statement do not have an expression since the result returned is the object R declared in the extended return statement itself.

Although extended return statements cannot be nested there could nevertheless be several in a function, perhaps in branches of an if statement or case statement. This would be quite likely in the case of a type with discriminants

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

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

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

This also illustrates the important point that although we introduced these extended return statements in the context of greater flexibility for limited types they can be used with any types at all such as the nonlimited type Person. The mechanism of passing a hidden parameter which is the address for the returned object of course only applies to limited types. In the case of nonlimited types, the result is simply delivered in the usual way.

We can also rename the result of a function call – even if it is limited.

The result type of a function can be constrained or unconstrained as in the case of the type Person but the actual object delivered must be of a definite subtype. For example suppose we have

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

Then the type UA is unconstrained but the subtype CA is constrained. We can use both with extended return statements.

In the constrained case the subtype in the extended return statement has to statically match (typically it will be the same textually but need not) thus

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

In the unconstrained case the result R has to be constrained either by its subtype or by its initial value. Thus

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

or

```
function Make( ... ) return UA is
begin
   ...
   return R: UA := (1 .. N => 0.0) do
      ...
   end return;
end Make;
```

The other important change to the result of functions which was discussed in the previous paper is that the result type can be of an anonymous access type. So we can write a function such as

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

The introduction of explicit access types for the result means that Ada 2005 is able to dispense with the notion of returning by reference.

This does, however, introduce a noticeable incompatibility between Ada 95 and Ada 2005. We might for example have a pool of slave tasks acting as servers. Individual slave tasks might be busy or idle. We might have a manager task which allocates slave tasks to different jobs. The manager might declare the tasks as an array

```
Slaves: array (1 .. 10) of TT;          -- TT is some task type
```

and then have another array of properties of the tasks such as

```
type Task_Data is
  record
    Active: Boolean := False;
    Job_Code: ... ;
  end record;

Slave_Data: array (1 .. 10) of Task_Data;
```

We now need a function to find an available slave. In Ada 95 we write

```
function Get_Slave return TT is
begin
  ...                                   -- find index K of first idle slave
  return Slaves(K);                     -- in Ada 95, not in Ada 2005
end Get_Slave;
```

This is not permitted in Ada 2005. If the result type is limited (as in this case) then the expression in the return statement has to be an aggregate or function call and not an object such as Slaves(K).

In Ada 2005 the function has to be rewritten to honestly return an access value referring to the task type rather than invoking the mysterious concept of returning by reference.

So we have to write

```
function Get_Slave return access TT is
begin
  ...                                   -- find index K of first idle slave
  return Slaves(K)'Access;              -- in Ada 2005
end Get_Slave;
```

and all the calls of Get_Slave have to be changed to correspond as well.

This is perhaps the most serious incompatibility between Ada 95 and Ada 2005. But then, at the end of the day, honesty is the best policy.

References

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

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

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