Rationale for Ada 2005: 2 Access types

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 concerning access types for Ada 2005.

Ada 2005 permits all access types to be access to constant types and to indicate that null is not an allowed value in all contexts. Anonymous access types are permitted in more contexts than just as access parameters and discriminants; they can also be used for variables and all components of composite types. This further use of access types is of considerable value in object oriented programming by reducing the need for (unnecessary) explicit type conversions.

A further major improvement concerns access to subprogram types which are now allowed to be anonymous in line with access to object types. This permits so-called "downward closures" and allows the flexible use of procedures as parameters of subprograms and thereby avoids excessive use of generic units.

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] does not specifically mention access types as an area needing attention. Access types are, of course, more of a tactical detail than a strategic issue and so this is not surprising.

However, the guidance document strongly emphasizes improvements to object oriented programming and the use of access types figures highly in that area. Indeed one of the motivations for changes was to reduce the number of explicit access type conversions required for OOP.

The guidance document also asks for "improvements that will remedy shortcomings in Ada". The introduction of anonymous access-to-subprogram types comes into that category in the minds of many users.

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

230 Generalized use of anonymous access types

231 Access to constant parameters, null-excluding types

254 Anonymous access to subprogram types

318 Limited and anonymous access return types

363 Eliminating access subtype problems

382 Current instance rule and anonymous access types

384 Discriminated type conversion rules

385 Stand-alone objects of anonymous access types

392 Prohibit unsafe array conversions

402 Access discriminants of nonlimited types

404 Not null and all in access parameters and types

406 Aliased permitted with anonymous access types

409 Conformance with access to subprogram types

416 Access results, accessibility and return statements

420 Resolution of universal operations in Standard

423 Renaming, null exclusion and formal objects

These changes can be grouped as follows.

First, there is a general orthogonalization of the rules regarding whether the designated type is constant and whether the access subtype includes null (231, part of 404, part of 423).

A major change is the ability to use anonymous access types more widely (230, part of 318, 385, 392, part of 404, 406, part of 416, part of 420). This was found to require some redefinition of the rules regarding the use of a type name within its own definition (382). Access discriminants are now also permitted with nonlimited types (402).

The introduction of anonymous access-to-subprogram types enables local subprograms to be passed as parameters to other subprograms (254, 409). This has been a feature of many other programming languages for over 40 years and its omission from Ada has always been both surprising and irritating and forced the excessive use of generics.

Finally there are some corrections to the rules regarding changing discriminants which prevent attempting to access components of variants that do not exist (363). There is also a change to the rules concerning type conversions and discriminants to make them symmetric (384).

2 Null exclusion and constant

In Ada 95, anonymous access types and named access types have unnecessarily different properties. Furthermore anonymous access types only occur as access parameters and access discriminants.

Anonymous access types in Ada 95 never have null as a value whereas named access types always have null as a value. Suppose we have the following declarations

```
type T is
  record
    Component: Integer;
  end record;

type Ref_T is access T;
T_Ptr: Ref_T;
```

Note that T_Ptr by default will have the value **null**. Now suppose we have a procedure with an access parameter thus

```
procedure P(A: access T) is
  X: Integer;
begin
```

```
   X := A.Component;              -- read a component of A
                                  -- no check for null in 95
   ...
end P;
```

In Ada 95 an access parameter such as A can never have the value null and so there is no need to check for null when doing a dereference such as reading the component A.Component. This is assured by always performing a check when P is called. So calling P with an actual parameter whose value is null such as P(T_Ptr) causes Constraint_Error to be raised at the point of call. The idea was that within P we would have more efficient code for dereferencing and dispatching at the cost of just one check when the procedure is called. Such an access parameter we now refer to as being of a subtype that excludes null.

Ada 2005 extends this idea of access types that exclude null to named access types as well. Thus we can write

```
type Ref_NNT is not null access T;
```

In this case an object of the type Ref_NNT cannot have the value null. An immediate consequence is that all such objects should be explicitly initialized – they will otherwise be initialized to null by default and this will raise Constraint_Error.

Since the property of excluding null can now be given explicitly for named types, it was decided that for uniformity, anonymous access types should follow the same rule whenever possible. So, if we want an access parameter such as A to exclude null in Ada 2005 then we have to indicate this in the same way

```
procedure PNN(A: not null access T) is
   X: Integer;
begin
   X := A.Component;              -- read a component of A
                                  -- no check for null in 2005
   ...
end PNN;
```

This means of course that the original procedure

```
procedure P(A: access T) is
   X: Integer;
begin
   X := A.Component;              -- read a component of A
                                  -- check for null in 2005
   ...
end P;
```

behaves slightly differently in Ada 2005 since A is no longer of a type that excludes null. There now has to be a check when accessing the component of the record because null is now an allowed value of A. So in Ada 2005, calling P with a null parameter results in Constraint_Error being raised within P only when we attempt to do the dereference, whereas in Ada 95 it is always raised at the point of call.

This is of course technically an incompatibility of an unfortunate kind. Here we have a program that is legal in both Ada 95 and Ada 2005 but it behaves differently at execution time in that Constraint_Error is raised at a different place. But of course, in practice if such a program does raise Constraint_Error in this way then it clearly has a bug and so the difference does not really matter.

Various alternative approaches were considered in order to eliminate this incompatibility but they all seemed to be ugly and it was felt that it was best to do the proper thing rather than have a permanent wart.

However the situation regarding controlling access parameters is somewhat different. Remember that a controlling parameter is a parameter of a tagged type where the operation is primitive – that is declared alongside the tagged type in a package specification (or inherited of course). Thus consider

```
package PTT is
  type TT is tagged
    record
      Component: Integer;
    end record;

  procedure Op(X: access TT);              -- primitive operation
  ...
end PTT;
```

The type TT is tagged and the procedure Op is a primitive operation and so the access parameter X is a controlling parameter.

In this case the anonymous access (sub)type still excludes null as in Ada 95 and so null is not permitted as a parameter. The reason is that controlling parameters provide the tag for dispatching and null has no tag value. Remember that all controlling parameters have to have the same tag. We can add **not null** to the parameter specification if we wish but to require it explicitly for all controlling parameters was considered to be too much of an incompatibility. But in newly written programs, we should be encouraged to write **not null** explicitly in order to avoid confusion during maintenance.

Another rule regarding null exclusion is that a type derived from a type that excludes null also excludes null. Thus given

```
type Ref_NNT is not null access T;
type Another_Ref_NNT is new Ref_NNT;
```

then Another_Ref_NNT also excludes null. On the other hand if we start with an access type that does not exclude null then a derived type can exclude null or not thus

```
type Ref_T is access T;
type Another_Ref_T is new Ref_T;
type ANN_Ref_T is new not null Ref_T;
```

then Another_Ref_T does not exclude null but ANN_Ref_T does exclude null.

A technical point is that all access types including anonymous access types in Ada 2005 have null as a value whereas in Ada 95 the anonymous access types did not. It is only subtypes in Ada 2005 that do not always have null as a value. Remember that Ref_NNT is actually a first-named subtype.

An important advantage of all access types having null as a value is that it makes interfacing to C much easier. If a parameter in C has type *t then the corresponding parameter in Ada can have type **access** T and if the C routine needs null passed sometimes then all is well – this was a real pain in Ada 95.

An explicit null exclusion can also be used in object declarations much like a constraint. Thus we can have

```
type Ref_Int is access all Integer;
X: not null Ref_Int := Some_Integer'Access;
```

Note that we must initialize X otherwise the default initialization with **null** will raise Constraint_Error.

In some ways null exclusions have much in common with constraints. We should compare the above with

```
Y: Integer range 1 .. 10;
...
Y := 0;
```

Again Constraint_Error is raised because the value is not permitted for the subtype of Y. A difference however is that in the case of X the check is Access_Check whereas in the case of Y it is Range_Check.

The fact that a null exclusion is not actually classified as a constraint is seen by the syntax for subtype_indication which in Ada 2005 is

```
subtype_indication ::= [null_exclusion] subtype_mark [constraint]
```

An explicit null exclusion can also be used in subprogram declarations thus

```
function F(X: not null Ref_Int) return not null Ref_Int;
procedure P(X: in not null Ref_Int);
procedure Q(X: in out not null Ref_Int);
```

But a difference between null exclusions and constraints is that although we can use a null exclusion in a parameter specification we cannot use a constraint in a parameter specification. Thus

```
procedure P(X: in not null Ref_Int);          -- legal
procedure Q(X: in Integer range 1 .. N);      -- illegal
```

But null exclusions are like constraints in that they are both used in defining subtype conformance and static matching.

We can also use a null exclusion with access-to-subprogram types including protected subprograms.

```
type F is access function (X: Float) return Float;
Fn: not null F := Sqrt'Access;
```

and so on.

A null exclusion can also be used in object and subprogram renamings. We will consider subprogram renamings here and object renamings in the next section when we discuss anonymous access types. This is an area where there is a significant difference between null exclusions and constraints.

Remember that if an entity is renamed then any constraints are unchanged. We might have

```
procedure P(X: Positive);
...
procedure Q(Y: Natural) renames P;
...
Q(0);                                   -- raises Constraint_Error
```

The call of Q raises Constraint_Error because zero is not an allowed value of Positive. The constraint Natural on the renaming is completely ignored (Ada has been like that since time immemorial).

We would have preferred that this sort of peculiar behaviour did not extend to null exclusions. However, we already have the problem that a controlling parameter always excludes null even if it does not say so. So the rule adopted generally with null exclusions is that "null exclusions never lie". In other words, if we give a null exclusion then the entity must exclude null; however, if no null

exclusion is given then the entity might nevertheless exclude null for other reasons (as in the case of a controlling parameter).

So consider

```
procedure P(X: not null access T);
...
procedure Q(Y: access T) renames P;                    -- OK
...
Q(null);          -- raises Constraint_Error
```

The call of Q raises Constraint_Error because the parameter excludes null even though there is no explicit null exclusion in the renaming. On the other hand (we assume that X is not a controlling parameter)

```
procedure P(X: access T);
...
procedure Q(Y: not null access T) renames P;           -- NO
```

is illegal because the null exclusion in the renaming is a lie.

However, if P had been a primitive operation of T so that X was a controlling parameter then the renaming with the null exclusion would be permitted.

Care needs to be taken when a renaming itself is used as a primitive operation. Consider

```
package P is
  type T is tagged ...
  procedure One(X: access T);                  -- excludes null

  package Inner is
    procedure Deux(X: access T);               -- includes null
    procedure Trois(X: not null access T);     -- excludes null
  end Inner;

  use Inner;

  procedure Two(X: access T) renames Deux;           -- NO
  procedure Three(X: access T) renames Trois;        -- OK
  ...
```

The procedure One is a primitive operation of T and its parameter X is therefore a controlling parameter and so excludes null even though this is not explicitly stated. However, the declaration of Two is illegal. It is trying to be a dispatching operation of T and therefore its controlling parameter X has to exclude null. But Two is a renaming of Deux whose corresponding parameter does not exclude null and so the renaming is illegal. On the other hand the declaration of Three is permitted because the parameter of Trois does exclude null.

The other area that needed unification concerned **constant**. In Ada 95 a named access type can be an access to constant type rather than an access to variable type thus

```
type Ref_CT is access constant T;
```

Remember that this means that we cannot change the value of an object of type T via the access type.

Remember also that Ada 95 introduced more general access types whereas in Ada 83 all access types were pool specific and could only access values created by an allocator. An access type in Ada 95 can also refer to any object marked **aliased** provided that the access type is declared with **all** thus

```
type Ref_VT is access all T;
X: aliased T;
R: Ref_VT := X'Access;
```

So in summary, Ada 95 has three kinds of named access types

```
access T;                -- pool specific only, read & write
access all T             -- general, read & write
access constant T        -- general, read only
```

But in Ada 95, the distinction between variable and constant access parameters is not permitted. Ada 2005 rectifies this by permitting **constant** with access parameters. So we can write

```
procedure P(X: access constant T);       -- legal 2005
procedure P(X: access T);
```

Observe however, that **all** is not permitted with access parameters. Ordinary objects can be constant or variable thus

```
C: constant Integer := 99;
V: Integer;
```

and access parameters follow this pattern. It is named access types that are anomalous because of the need to distinguish pool specific types for compatibility with Ada 83 and the subsequent need to introduce **all**.

In summary, Ada 2005 access parameters can take the following four forms

```
procedure P1(X: access T);
procedure P2(X: access constant T);
procedure P3(X: not null access T);
procedure P4(X: not null access constant T);
```

Moreover, as mentioned above, controlling parameters always exclude null even if this is not stated and so in that case P1 and P3 are equivalent. Controlling parameters can also be constant in which case P2 and P4 are equivalent.

Similar rules apply to access discriminants; thus they can exclude null and/or be access to constant.

3 Anonymous access types

As just mentioned, Ada 95 permits anonymous access types only as access parameters and access discriminants. And in the latter case only for limited types. Ada 2005 sweeps away these restrictions and permits anonymous access types quite freely.

The main motivation for this change concerns type conversion. It often happens that we have a type T somewhere in a program and later discover that we need an access type referring to T in some other part of the program. So we introduce

```
type Ref_T is access all T;
```

And then we find that we also need a similar access type somewhere else and so declare another access type

```
type T_Ptr is access all T;
```

If the uses of these two access types overlap then we will find that we have explicit type conversions all over the place despite the fact that they are really the same type. Of course one might argue that planning ahead would help a lot but, as we know, programs often evolve in an unplanned way.

A more important example of the curse of explicit type conversion concerns object oriented programming. Access types feature quite widely in many styles of OO programming. We might have a hierarchy of geometrical object types starting with a root abstract type Object thus

```
type Object is abstract;
type Circle is new Object with ...

type Polygon is new Object with ...
type Pentagon is new Polygon with ...

type Triangle is new Polygon with ...
type Equilateral_Triangle is new Triangle with ...
```

then we might well find ourselves declaring named access types such as

```
type Ref_Object is access all Object'Class;
type Ref_Circle is access all Circle;
type Ref_Triangle is access all Triangle'Class;
type Ref_Equ_Triangle is access all Equilateral_Triangle;
```

Conversion between these clearly ought to be permitted in many cases. In some cases it can never go wrong and in others a run time check is required. Thus a conversion between a Ref_Circle and a Ref_Object is always possible because every value of Ref_Circle is also a value of Ref_Object but the reverse is not the case. So we might have

```
RC: Ref_Circle := A_Circle'Access;
RO: Ref_Object;
...
RO := Ref_Object(RC);          -- explicit conversion, no check
...
RC := Ref_Circle(RO);          -- needs a check
```

However, it is a rule of Ada 95 that type conversions between these named access types have to be explicit and give the type name. This is considered to be a nuisance by many programmers because such conversions are allowed without naming the type in other OO languages. It would not be quite so bad if the explicit conversion were only required in those cases where a run time check was necessary.

Moreover, these are trivial (view) conversions since they are all just pointers and no actual change of value takes place anyway; all that has to be done is to check that the value is a legal reference for the target type and in many cases this is clear at compilation. So requiring the type name is very annoying.

In fact the only conversions between named tagged types (and named access types) that are allowed implicitly in Ada are conversions to a class wide type when it is initialized or when it is a parameter (which is really the same thing).

It would have been nice to have been able to relax the rules in Ada 2005 perhaps by saying that a named conversion is only required when a run time check is required. However, such a change would have caused lots of existing programs to become ambiguous.

So, rather than meddle with the conversion rules, it was instead decided to permit the use of anonymous access types in more contexts in Ada 2005. Anonymous access types have the interesting property that they are anonymous and so necessarily do not have a name that could be used in a conversion. Thus we can have

```
RC: access Circle := A_Circle'Access;
RO: access Object'Class;          -- default null
...
RO := RC;                         -- implicit conversion, no check
```

On the other hand we cannot write

```
RC := RO;                         -- implicit conversion, needs a check
```

because the general rule is that if a check is required then the conversion must be explicit. So typically we will still need to introduce named access types for some conversions.

We can of course also use null exclusions with anonymous access types thus

```
RC: not null access Circle := A_Circle'Access;
RO: not null access Object'Class;          -- careful
```

The declaration of RO is unfortunate because no initial value is given and the default of null is not permitted and so it will raise Constraint_Error; a worthy compiler will detect this during compilation and give us a friendly warning.

Note that we never never write **all** with anonymous access types.

We can of course also use **constant** with anonymous access types. Note carefully the difference between the following

```
ACT: access constant T := T1'Access;
CAT: constant access T := T1'Access;
```

In the first case ACT is a variable and can be used to access different objects T1 and T2 of type T. But it cannot be used to change the value of those objects. In the second case CAT is a constant and can only refer to the object given in its initialization. But we can change the value of the object that CAT refers to. So we have

```
ACT := T2'Access;                 -- legal, can assign
ACT.all := T2;                    -- illegal, constant view
CAT := T2'Access;                 -- illegal, cannot assign
CAT.all := T2;                    -- legal, variable view
```

At first sight this may seem confusing and consideration was given to disallowing the use of constants such as CAT (but permitting ACT which is probably more useful since it protects the accessed value). But the lack of orthogonality was considered very undesirable. Moreover Ada is a left to right language and we are familiar with equivalent constructions such as

```
type CT is access constant T;
ACT: CT;
```

and

```
type AT is access T;
CAT: constant AT;
```

(although the alert reader will note that the latter is illegal because I have foolishly used the reserved word **at** as an identifier).

We can of course also write

```
CACT: constant access constant T := T1'Access;
```

The object CACT is then a constant and provides read-only access to the object T1 it refers to. It cannot be changed to refer to another object such as T2 nor can the value of T1 be changed via CACT.

An object of an anonymous access type, like other objects, can also be declared as aliased thus

```
X: aliased access T;
```

although such constructions are likely to be used rarely.

Anonymous access types can also be used as the components of arrays and records. In the Introduction we saw that rather than having to write

```
type Cell;
type Cell_Ptr is access Cell;

type Cell is
  record
    Next: Cell_Ptr;
    Value: Integer;
  end record;
```

we can simply write

```
type Cell is
  record
    Next: access Cell;
    Value: Integer;
  end record;
```

and this not only avoids have to declare the named access type Cell_Ptr but it also avoids the need for the incomplete type declaration of Cell.

Permitting this required some changes to a rule regarding the use of a type name within its own declaration – the so-called current instance rule.

The original current instance rule was that within a type declaration the type name did not refer to the type itself but to the current object of the type. The following task type declaration illustrates both a legal and illegal use of the task type name within its own declaration. It is essentially an extract from a program in Section 18.10 of [2] which finds prime numbers by a multitasking implementation of the Sieve of Eratosthenes. Each task of the type is associated with a prime number and is responsible for removing multiples of that number and for creating the next task when a new prime number is discovered. It is thus quite natural that the task should need to make a clone of itself.

```
task type TT (P: Integer) is
  ...
end;

type ATT is access TT;

task body TT is
  function Make_Clone(N: Integer) return ATT is
  begin
    return new TT(N);                -- illegal
  end Make_Clone;

  Ref_Clone: ATT;
  ...
```

```
begin
   ...
   Ref_Clone := Make_Clone(N);
   ...
   abort TT;                          -- legal
   ...
end TT;
```

The attempt to make a slave clone of the task in the function Make_Clone is illegal because within the task type its name refers to the current instance and not to the type. However, the abort statement is permitted and will abort the current instance of the task. In this example the solution is simply to move the function Make_Clone outside the task body.

However, this rule would have prevented the use of the type name Cell to declare the component Next within the type Cell and this would have been infuriating since the linked list paradigm is very common.

In order to permit this the current instance rule has been changed in Ada 2005 to allow the type name to denote the type itself within an anonymous access type declaration (but not a named access type declaration). So the type Cell is permitted.

Note however that in Ada 2005, the task TT still cannot contain the declaration of the function Make_Clone. Although we no longer need to declare the named type ATT since we can now declare Ref_Clone as

```
Ref_Clone: access TT;
```

and we can declare the function as

```
function Make_Clone(N: Integer) return access TT is
begin
   return new TT(N);
end Make_Clone;
```

where we have an anonymous result type, nevertheless the allocator **new** TT inside Make_Clone remains illegal if Make_Clone is declared within the task body TT. But such a use is unusual and declaring a distinct external function is hardly a burden.

To be honest we can simply declare a subtype of a different name outside the task

```
subtype XTT is TT;
```

and then we can write **new** XTT(N); in the function and keep the function hidden inside the task. Indeed we don't need the function anyway because we can just write

```
Ref_Clone := new XTT(N);
```

in the task body.

The introduction of the wider use of anonymous access types requires some revision to the rules concerning type comparisons and conversions. This is achieved by the introduction of a type *universal_access* by analogy with the types *universal_integer* and *universal_real*. Two new equality operators are defined in the package Standard thus

```
function "=" (Left, Right: universal_access) return Boolean;

function "/=" (Left, Right: universal_access) return Boolean;
```

The literal **null** is now deemed to be of type *universal_access* and appropriate conversions are defined as well. These new operations are only applied when at least one of the arguments is of an anonymous access types (not counting **null**).

Interesting problems arise if we define our own equality operation. For example, suppose we wish to do a deep comparison on two lists defined by the type Cell. We might decide to write a recursive function with specification

```
function "=" (L, R: access Cell) return Boolean;
```

Note that it is easier to use access parameters rather than parameters of type Cell itself because it then caters naturally for cases where null is used to represent an empty list. We might attempt to write the body as

```
function "=" (L, R: access Cell) return Boolean is
begin
  if L = null or R = null then              -- wrong =
    return L = R;                           -- wrong =
  elsif L.Value = R.Value then
    return L.Next = R.Next;                 -- recurses OK
  else
    return False;
  end if;
end "=" ;
```

But this doesn't work because the calls of "=" in the first two lines recursively call the function being declared whereas we want to call the predefined "=" in these cases.

The difficulty is overcome by writing Standard."=" thus

```
  if Standard."=" (L, null) or Standard."=" (R, null) then
    return Standard."=" (L, R);
```

The full rules regarding the use of the predefined equality are that it cannot be used if there is a user-defined primitive equality operation for either operand type unless we use the prefix Standard. A similar rule applies to fixed point types as we shall see in a later paper.

Another example of the use of the type Cell occurred in the previous paper when we were discussing type extension at nested levels. That example also illustrated that access types have to be named in some circumstances such as when they provide the full type for a private type. We had

```
package Lists is
  type List is limited private;             -- private type
  ...
private
  type Cell is
    record
      Next: access Cell;                    -- anonymous type
      C: Colour;
    end record;

  type List is access Cell;                 -- full type
end;

package body Lists is
  procedure Iterate(IC: in Iterator'Class; L: in List) is
    This: access Cell := L;                 -- anonymous type
  begin
```

```
      while This /= null loop
        IC.Action(This.C);                    -- dispatches
        This := This.Next;
      end loop;
    end Iterate;
  end Lists;
```

In this case we have to name the type List because it is a private type. Nevertheless it is convenient to use an anonymous access type to avoid an incomplete declaration of Cell.

In the procedure Iterate the local variable This is also of an anonymous type. It is interesting to observe that if This had been declared to be of the named type List then we would have needed an explicit conversion in

```
        This := List(This.Next);              -- explicit conversion
```

Remember that we *always* need an explicit conversion when converting to a named access type. There is clearly an art in using anonymous types to best advantage.

The Introduction showed a number of other uses of anonymous access types in arrays and records and as function results when discussing Noah's Ark and other animal situations. We will now turn to more weighty matters.

An important matter in the case of access types is accessibility. The accessibility rules are designed to prevent dangling references. The basic rule is that we cannot create an access value if the object referred to has a lesser lifetime than the access type.

However there are circumstances where the rule is unnecessarily severe and that was one reason for the introduction of access parameters. Perhaps some recapitulation of the problems would be helpful. Consider

```
type T is ...
Global: T;
type Ref_T is access all T;
Dodgy: Ref_T;

procedure P(Ptr: access T) is
begin
  ...
  Dodgy := Ref_T(Ptr);                        -- dynamic check
end P;

procedure Q(Ptr: Ref_T) is
begin
  ...
  Dodgy := Ptr;                               -- legal
end Q;
...
declare
  X: aliased T;
begin
  P(X'Access);                                -- legal
  Q(X'Access);                                -- illegal
end;
```

Here we have an object X with a short lifetime and we must not squirrel away an access referring to X in an object with a longer lifetime such as Dodgy. Nevertheless we want to manipulate X indirectly using a procedure such as P.

If the parameter were of a named type such as Ref_T as in the case of the procedure Q then the call would be illegal since within Q we could then assign to a variable such as Dodgy which would then retain the "address" of X after X had ceased to exist.

However, the procedure P which uses an access parameter permits the call. The reason is that access parameters carry dynamic accessibility information regarding the actual parameter. This extra information enables checks to be performed only if we attempt to do something foolish within the procedure such as make an assignment to Dodgy. The conversion to the type Ref_T in this assignment fails dynamically and disaster is avoided.

But note that if we had called P with

```
P(Global'Access);
```

where Global is declared at the same level as Ref_T then the assignment to Dodgy would be permitted.

The accessibility rules for the new uses of anonymous access types are very simple. The accessibility level is simply the level of the enclosing declaration and no dynamic information is involved. (The possibility of preserving dynamic information was considered but this would have led to inefficiencies at the points of use.)

In the case of a stand-alone variable such as

```
V: access Integer;
```

then this is essentially equivalent to

```
type anon is access all Integer;
V: anon;
```

A similar situation applies in the case of a component of a record or array type. Thus if we have

```
type R is
  record
    C: access Integer;
    ...
  end record;
```

then this is essentially equivalent to

```
type anon is access all Integer;
type R is
  record
    C: anon;
    ...
  end record;
```

Further if we now declare a derived type then there is no new physical access definition, and the accessibility level is that of the original declaration. Thus consider

```
procedure Proc is
  Local: aliased Integer;
  type D is new R;
  X: D := D'(C => Local'Access, ... );          -- illegal
begin
```

```
  ...
end Proc;
```

In this example the accessibility level of the component C of the derived type is the same as that of the parent type R and so the aggregate is illegal. This somewhat surprising rule is necessary to prevent some very strange problems which we will not explore in this paper.

One consequence of which users should be aware is that if we assign the value in an access parameter to a local variable of an anonymous access type then the dynamic accessibility of the actual parameter will not be held in the local variable. Thus consider again the example of the procedure P containing the assignment to Dodgy

```
procedure P(Ptr: access T) is
begin
  ...
  Dodgy := Ref_T(Ptr);                    -- dynamic check
end P;
```

and this variation in which we have introduced a local variable of an anonymous access type

```
procedure P1(Ptr: access T) is
  Local_Ptr: access T;
begin
  ...
  Local_Ptr := Ptr;                       -- implicit conversion
  Dodgy := Ref_T(Local_Ptr);              -- static check, illegal
end P1;
```

Here we have copied the value in the parameter to a local variable before attempting the assignment to Dodgy. (Actually it won't compile but let us analyze it in detail anyway.)

The conversion in P using the access parameter Ptr is dynamic and will only fail if the actual parameter has an accessibility level greater than that of the type Ref_T. So it will fail if the actual parameter is X and so raise Program_Error but will pass if it has the same level as the type Ref_T such as the variable Global.

In the case of P1, the assignment from Ptr to Local_Ptr involves an implicit conversion and static check which always passes. (Remember that implicit conversions are never allowed if they involve a dynamic check.) However, the conversion in the assignment to Dodgy in P1 is also static and will always fail no matter whether X or Global is passed as actual parameter.

So the effective behaviours of P and P1 are the same if the actual parameter is X (they both fail, although one dynamically and the other statically) but will be different if the actual parameter has the same level as the type Ref_T such as the variable Global. The assignment to Dodgy in P will work in the case of Global but the assignment to Dodgy in P1 never works.

This is perhaps surprising, an apparently innocuous intermediate assignment has a significant effect because of the implicit conversion and the consequent loss of the accessibility information. In practice this is very unlikely to be a problem. In any event programmers are aware that access parameters are special and carry dynamic information.

In this particular example the loss of the accessibility information through the use of the intermediate stand-alone variable is detected at compile time. More elaborate examples can be constructed whereby the problem only shows up at execution time. Thus suppose we introduce a third procedure Agent and modify P and P1 so that we have

```
procedure Agent(A: access T) is
begin
  Dodgy := Ref_T(A);                    -- dynamic check
end Agent;

procedure P(Ptr: access T) is
begin
  Agent(Ptr);                           -- may be OK
end P;

procedure P1(Ptr: access T) is
  Local_Ptr: access T;
begin
  Local_Ptr := Ptr;                     -- implicit conversion
  Agent(Local_Ptr);                     -- never OK
end P1;
```

Now we find that P works much as before. The accessibility level passed into P is passed to Agent which then carries out the assignment to Dodgy. If the parameter passed to P is the local X then Program_Error is raised in Agent and propagated to P. If the parameter passed is Global then all is well.

The procedure P1 now compiles whereas it did not before. However, because the accessibility of the original parameter is lost by the assignment to Local_Ptr, it is the accessibility level of Local_Ptr that is passed to Agent and this means that the assignment to Dodgy always fails and raises Program_Error irrespective of whether P1 was called with X or Global.

If we just want to use another name for some reason then we can avoid the loss of the accessibility level by using renaming. Thus we could have

```
procedure P2(Ptr: access T) is
  Local_Ptr: access T renames Ptr;
begin
  ...
  Dodgy := Ref_T(Local_Ptr);            -- dynamic check
end P2;
```

and this will behave exactly as the original procedure P.

As usual a renaming just provides another view of the same entity and thus preserves the accessibility information.

A renaming can also include **not null** thus

```
Local_Ptr: not null access T renames Ptr;
```

Remember that not null must never lie so this is only legal if Ptr is indeed of a type that excludes null (which it will be if Ptr is a controlling access parameter of the procedure P2).

A renaming might be useful when the accessed type T has components that we wish to refer to many times in the procedure. For example the accessed type might be the type Cell declared earlier in which case we might usefully have

```
Next: access Cell renames Ptr.Next;
```

and this will preserve the accessibility information.

Anonymous access types can also be used as the result of a function. In the Introduction we had

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

The accessibility level of the result in this case is the same as that of the declaration of the function itself.

We can also dispatch on the result of a function if the result is an access to a tagged type. Consider

```
function Unit return access T;
```

We can suppose that T is a tagged type representing some category of objects such as our geometrical objects and that Unit is a function returning a unit object such as a circle of unit radius or a triangle with unit side.

We might also have a function

```
function Is_Bigger(X, Y: access T) return Boolean;
```

and then

```
Thing: access T'Class := ... ;
...
Test: Boolean := Is_Bigger(Thing, Unit);
```

This will dispatch to the function Unit according to the tag of Thing and then of course dispatch to the appropriate function Is_Bigger.

The function Unit could also be used as a default value for a parameter thus

```
function Is_Bigger(X: access T; Y: access T := Unit) return Boolean;
```

Remember that a default used in such a construction has to be tag indeterminate.

Permitting anonymous access types as result types eliminates the need to define the concept of a "return by reference" type. This was a strange concept in Ada 95 and primarily concerned limited types (including task and protected types) which of course could not be copied. Enabling us to write **access** explicitly and thereby tell the truth removes much confusion. Limited types will be discussed in detail in a later paper.

Access return types can be a convenient way of getting a constant view of an object such as a table. We might have an array in a package body (or private part) and a function in the specification thus

```
package P is
  type Vector is array (Integer range <>) of Float;

  function Read_Vec return access constant Vector;
  ...
private

end;

package body P is

  The_Vector: aliased Vector :=   ;

  function Read_Vec return access constant Vector is
  begin
    return The_Vector'Access;
  end;
  ...
end P;
```

We can now write

```
X := Read_Vec(7);                    -- read element of array
```

This is strictly short for

```
X := Read_Vec.all(7);
```

Note that we cannot write

```
Read_Vec(7) := Y;                    -- illegal
```

although we could do so if we removed **constant** from the return type (in which case we should use a different name for the function).

The last new use of anonymous access types concerns discriminants. Remember that a discriminant can be of a named access type or an anonymous access type (as well as oher things). Discriminants of an anonymous access type are known as access discriminants. In Ada 95, access discriminants are only allowed with limited types. Discriminants of a named access type are just additional components with no special properties. But access discriminants of limited types are special. Since the type is limited, the object cannot be changed by a whole record assignment and so the discriminant cannot be changed even if it has defaults. Thus

```
type Minor is ...

type Major(M: access Minor) is limited
  record
    ...
  end record;

Small: aliased Minor;
Large: Major(Small'Access);
```

The objects Small and Large are now bound permanently together.

In Ada 2005, access discriminants are also allowed for nonlimited types. However, defaults are not permitted so that the discriminant cannot be changed so again the objects are bound permanently together. An interesting case arises when the discriminant is provided by an allocator thus

```
Larger: Major(new Minor( ... ));
```

In this case we say that the allocated object is a coextension of Larger. Coextensions have the same lifetime as the major object and so are finalized when it is finalized. There are various accessibility and other rules concerning objects which have coextensions which prevent difficulty when returning such objects from functions.

4 Downward closures

This section is really about access to subprogram types in general but the title downward closures has come to epitomize the topic.

The requirements for Ada 83, (Strawman .. Steelman) were strangely silent about whether parameters of subprograms could themselves be subprograms as was the case in Algol 60 and Pascal. Remember that Pascal was one of the languages on which the designs for the DoD language were to be based.

The predictability aspects of the requirements were interpreted as implying that all subprogram calls should be identified at compilation time on the grounds that if you didn't know what was being called than you couldn't know what the program was going to do. This was a particularly stupid attitude to take. The question of predictability (presumably in some safety or security context) really concerns the behaviour of particular programs rather than the universe of all programs that can be constructed in a language.

In any event the totality of subprograms that might be called in a program is finite and closed. It simply consists of the subprograms in the program. Languages such as Ada are not able to construct totally new subprograms out of lesser components in the way that they can create say floating point values.

So the world had to use generics for many applications that were natural for subprograms as parameters of other subprograms. Thankfully many implementers avoided the explosion that might occur with generics by clever code sharing which in a sense hid the parameterization behind the scenes.

The types of applications for which subprograms are natural as parameters are any where one subroutine is parameterized by another. They include many mathematical applications such as integration and maximization and more logical applications such as sorting and searching and iterating.

As outlined in the Introduction, the matter was partly improved in Ada 95 by the introduction of named access-to-subprogram types. This was essentially done to allow program call back to be implemented.

Program call back is when one program passes the "address" of a subprogram within it to another program so that this other program can later respond by calling back to the first program using the subprogram address supplied. This is often used for communication between an Ada application program and some other software such as an operating system which might even be written in another language such as C.

Named access to subprogram types certainly work for call back (especially with languages such as C that do not have nested subprograms) but the accessibility rules which followed those for general access to object types were restrictive. For example, suppose we have a general library level function for integration using a named access to subprogram type to pass the function to be integrated thus

```
type Integrand is access function(X: Float) return Float;

function Integrate(Fn: Integrand; Lo, Hi: Float) return Float;
```

then we cannot even do the simplest integration of our own function in a natural way. For example, suppose we wish to integrate a function such as Exp(X**2). We can try

```
with Integrate;
procedure Main is
  function F(X: Float) return Float is
  begin
    return Exp(X**2);
  end F;

  Result, L, H: Float;
begin
  ...                    -- set bounds in L and H say
  Result := Integrate(F'Access, L, H);        -- illegal in 95
  ...
end Main
```

But this is illegal because of the accessibility check necessary to prevent us from writing something like

```
Evil: Integrand;
X: Float;
...
declare
  Y: Float;
  function F(X: Float) return Float is
    ...
    Y := X;                          --assign to Y in local block
    ...
  end F;
begin
  Evil := F'Access: -- illegal
end;
  X := Evil(X);                      -- call function out of context
```

Here we have attempted to assign an access to the local function F in the global variable Evil. If this assignment had been permitted then the call of Evil would indirectly have called the function F when the context in which F was declared no longer existed; F would then have attempted to assign to the variable Y which no longer existed and whose storage space might now be used for something else. We can summarise this perhaps by saying that we are attempting to call F when it no longer exists.

Ada 2005 overcomes the problem by introducing anonymous access to subprogram types. This was actually considered during the design of Ada 95 but it was not done at the time for two main reasons. Firstly, the implementation problems for those who were using display vectors rather than static links were considered a hurdle. And secondly, a crafty technique was available using the newly introduced tagged types. And of course one could continue to use generics. But further thought showed that the implementation burden was not so great after all and nobody understood the tagged type technique which was really incredibly contorted. Moreover, the continued use of generics when other languages forty years ago had included a more natural mechanism was tiresome. So at long last Ada 2005 includes anonymous access to subprogram types.

We rewrite the integration function much as follows

```
function Integrate(Fn: access function(X: Float) return Float;
                       Lo, Hi: Float) return Float is
  Total: Float;
  N: constant Integer := ... ;          -- no of subdivisions
  Step: Float := (Hi – Lo) / Float(N);
  X: Float := Lo;                       -- current point
begin
  Total := 0.5 * Fn(Lo);                -- value at low bound
  for I in 1 .. N–1 loop
    X := X + Step;                      -- add values at
    Total := Total + Fn(X);             -- intermediate points
  end loop;
  Total := Total + 0.5 * Fn(Hi);        -- add final value
  return Total * Step;                  -- normalize
end Integrate;
```

The important thing to notice is the profile of Integrate in which the parameter Fn is of an anonymous access to subprogram type. We have also shown a simple body which uses the trapezium/trapezoid method and so calls the actual function corresponding to Fn at the two end points of the range and at a number of equally spaced intermediate points.

(NB It is time for a linguistic interlude. Roughly speaking English English trapezium equals US English trapezoid. They both originate from the Greek τραπεζα meaning a table (literally with four feet). Both originally meant a quadrilateral with no pairs of sides parallel. In the late 17th century, trapezium came to mean having one pair of sides parallel. In the 18th century trapezoid came to mean the same as trapezium but promptly faded out of use in England whereas in the US it continues in use. Meanwhile in the US, trapezium reverted to its original meaning of totally irregular. Trapezoid is rarely used in the UK but if used has reverted to its original meaning of totally irregular. A standard language would be useful. Anyway, the integration is using quadrilateral strips with one pair of sides parallel.)

With this new declaration of Integrate, the accessibility problems are overcome and we are allowed to write Integrate(F'Access, ...) just as we could write P(X'Access) in the example in the previous section where we discussed anonymous access to object types.

We still have to consider how a type conversion which would permit an assignment to a global variable is prevented. The following text illustrates both access to object and access to subprogram parameters.

```
type AOT is access all Integer;
type APT is access procedure (X: in out Float);

Evil_Obj: AOT;
Evil_Proc: APT;

procedure P(Objptr: access Integer;
            Procptr: access procedure (X: in out Float)) is
begin
  Evil_Obj := AOT(Objptr);              -- fails at run time
  Evil_Proc := APT(Procptr);            -- fails at compile time
end P;

declare
  An_Obj: aliased Integer;
  procedure A_Proc(X: in out Float) is
  begin ... end A_Proc;
begin
  P(An_Obj'Access, A_Proc'Access);      -- legal
end;

Evil_Obj.all := 0;                      -- assign to nowhere
Evil_Proc.all( ... );                   -- call nowhere
```

This repeats some of the structure of the previous section. The procedure P has an access to object parameter Objptr and an access to subprogram parameter Procptr; they are both of anonymous type. The call of P in the local block passes the addresses of a local object An_Obj and a local procedure A_Proc to P. This is permitted. We now attempt to assign the parameter values from within P to global objects Evil_Obj and Evil_Proc with the intent of assigning indirectly via Evil_Obj and calling indirectly via Evil_Proc after the object and procedure referred to no longer exist.

Both of these wicked deeds are prevented by the accessibility rules.

In the case of the object parameter Objptr it knows the accessibility level of the actual An_Obj and this is seen to be greater than that of the type AOT and so the conversion is prevented at run time and in fact Program_Error is raised. But if An_Obj had been declared at the same level as AOT and not within an inner block then the conversion would have been permitted.

However, somewhat different rules apply to anonymous access to subprogram parameters. They do not carry an indication of the accessibility level of the actual parameter but simply treat it as if it were infinite (strictly – deeper than anything else). This of course prevents the conversion to the type APT and all is well; this is detected at compile time. But note that if the procedure A_Proc had been declared at the same level as APT then the conversion would still have failed because the accessibility level is treated as infinite.

There are a number of reasons for the different treatment of anonymous access to subprogram types. A big problem is that named access to subprogram types are implemented in the same way as C *func in almost all compilers. Permitting the conversion from anonymous access to subprogram types to named ones would thus have caused problems because that model does not work especially for display based implementations. Carrying the accessibility level around would not have prevented these conversions. The key goal was simply to provide a facility corresponding to that in Pascal and not to encourage too much fooling about with access to subprogram types. Recall that the attribute Unchecked_Access is permitted for access to object types but was considered far too dangerous for access to subprogram types for similar reasons.

The reader may be feeling both tired and that there are other ways around the problems of accessibility anyway. Thus the double integration presented in the Introduction can easily be circumvented in many cases. We computed

$$\int_0^1 \int_0^1 xy \, dy \, dx$$

using the following program

```
with Integrate;
procedure Main is
  function G(X: Float) return Float is
    function F(Y: Float) return Float is
    begin
      return X*Y;
    end F;
  begin
    return Integrate(F'Access, 0.0, 1.0);
  end G;

  Result: Float;
begin
  Result:= Integrate(G'Access, 0.0, 1.0);
  ...
end Main;
```

The essence of the problem was that F had to be declared inside G because it needed access to the parameter X of G. But the astute reader will note that this example is not very convincing because the integrals can be separated and the functions both declared at library level thus

```
function F(Y: Float) return Float is
begin
  return Y;
end F;
```

```
function G(X: Float) return Float is
begin
  return X;
end G;

Result:= Integrate(F'Access, 0.0, 1.0) * Integrate(G'Access, 0.0, 1.0);
```

and so it all works using the Ada 95 version of Integrate anyway.

However, if the two integrals had been more convoluted or perhaps the region had not been square but triangular so that the bound of the inner integral depended on the outer variable as in

$$\int_0^1 \int_0^x xy \, dy \, dx$$

then nested functions would be vital.

We will now consider a more elegant example which illustrates how we might integrate an arbitrary function of two variables $F(x, y)$ over a rectangular region.

Assume that we have the function Integrate for one dimension as before

```
function Integrate(Fn: access function(X: Float) return Float;
                   Lo, Hi: Float) return Float;
```

Now consider

```
function Integrate(Fn: access function(X, Y: Float) return Float;
                   LoX, HiX: Float;
                   LoY, HiY: Float) return Float is
  function FnX(X: Float) return Float is
    function FnY(Y: Float) return Float is
    begin
      return Fn(X, Y);
    end FnY;
  begin
    return Integrate(FnY'Access, LoY, HiY);
  end FnX;
begin
  return Integrate(FnX'Access, LoX, HiX);
end integrate;
```

The new function Integrate for two dimensions overloads and uses the function Integrate for one dimension (a good example of overloading). With this generality it is again impossible to arrange the structure in a manner which is legal in Ada 95.

We might use the two-dimensional integration routine to solve the original trivial problem as follows

```
function F(X, Y: Float) return Float is
begin
  return X*Y;
end F;
...

Result := Integrate(F'Access, 0.0, 1.0, 0.0, 1.0);
```

As an exercise the reader might like to rewrite the two dimensional function to work on a non-rectangular domain. The trick is to pass the bounds of the inner integral also as functions. The profile then becomes

```
function Integrate(Fn: access function(X, Y: Float) return Float;
                   LoX, HiX: Float
                   LoY, HiY: access function(X: Float) return Float)
                                                        return Float;
```

In case the reader should think that this topic is all too mathematical it should be pointed out that anonymous access to subprogram parameters are widely used in the new container library thereby saving the unnecessary use of generics.

For example the package Ada.Containers.Vectors declares procedures such as

```
procedure Update_Element
  (Container: in Vector; Index: in Index_Type;
   Process: not null access procedure (Element: in out Element_Type));
```

This updates the element of the vector Container whose index is Index by calling the procedure Process with that element as parameter. Thus if we have a vector of integers V and we need to double the value of those with index in the range 5 to 10, then we would first declare a procedure such as

```
procedure Double(E: in out Integer) is
begin
  E := 2 * E;
end Double ;
```

and then write

```
for I in 5 .. 10 loop
  Update_Element(V, I, Double'Access);
end loop;
```

Further details of the use of access to subprogram types with containers will be found in a later paper.

Finally it should be noted that anonymous access to subprogram types can also be used in all those places where anonymous access to object types are allowed. That is as stand-alone objects, as components of arrays and records, as function results, in renamings, and in access discriminants.

The reader who likes long sequences of reserved words should realise by now that there is no limit in Ada 2005. This is because a function without parameters can return an access to function as its result and this in turn could be of a similar kind. So we would have

```
type FF is access function return access function return access function ...
```

Attempts to compile such an access to function type will inevitably lead to madness.

5 Access types and discriminants

This final topic concerns two matters. The first is about accessing components of discriminated types that might vanish or change mysteriously and the second is about type conversions.

Recall that we can have a mutable variant record such as

```
type Gender is (Male, Female, Neuter);
```

```
type Mutant(Sex: Gender := Neuter) is
  record
    Birth: Date;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Children: Integer;
      when Neuter =>
        null;
    end case;
  end record;
```

This represents a world in which there are three sexes, males which can have beards, females which can bear children, and neuters which are fairly useless. Note the default value for the discriminant. This means that if we declare an unconstrained object thus

```
The_Thing: Mutant;
```

then The_Thing is neuter by default but could have its sex changed by a whole record assignment thus

```
The_Thing := (Male, The_Thing.Birth, True);
```

It now is Male and has a beard but the date of birth retains its previous value.

The problem with this sort of object is that components can disappear. If it were changed to be Female then the beard would vanish and be replaced by children. Because of this ghostly behaviour certain operations on mutable objects are forbidden.

One obvious rule is that it is not permissible to rename components which might vanish. So

```
Hairy: Boolean renames The_Thing.Bearded;        -- illegal
```

is not permitted. This was an Ada 83 rule. It was probably the case that the rules were watertight in Ada 83. However, Ada 95 introduced many more possibilities. Objects and components could be marked as **aliased** and the Access attribute could be applied. Additional rules were then added to prevent creating references to things that could vanish.

However, it was then discovered that the rules in Ada 95 regarding access types were not watertight. Accordingly various attempts were made to fix them in a somewhat piecemeal fashion. The problems are subtle and do not seem worth describing in their entirety in this general presentation. We will content ourselves with just a couple of examples.

In Ada 95 we can declare types such as

```
type Mutant_Name is access all Mutant;
type Things_Name is access all Mutant(Neuter);
```

Naturally enough an object of type Things_Name can only be permitted to reference a Mutant whose Sex is Neuter.

```
Some_Thing: aliased Mutant;
Thing_Ptr: Things_Name := Some_Thing'Access;
```

Things would now go wrong if we allowed Some_Thing to have a sex change. Accordingly there is a rule in Ada 95 that says that an aliased object such as Some_Thing is considered to be constrained. So that is quite safe.

However, matters get more difficult when a type such as Mutant is used for a component of another type such as

```
type Monster is
  record
    Head: Mutant(Female);
    Tail: aliased Mutant;
  end record;
```

Here we are attempting to declare a nightmare monster whose head is a female but whose tail is deceivingly mutable. Those with a decent education might find that this reminds them of the Sirens who tempted Odysseus by their beautiful voices on his trip past the monster Scylla and the whirlpool Charybdis. Those with an indecent education can compare it to a pantomime theatre horse (or mare, maybe indeed a nightmare). We could then write

```
M: Monster;
Thing_Ptr := Monster.Tail'Access;
```

However, there is an Ada 95 rule that says that the Tail has to be constrained since it is aliased so the type Monster is not allowed. So far so good.

But now consider the following very nasty example

```
generic
  type T is private;
  Before, After: T;
  type Name is access all T;
  A_Name: in out Name;
procedure Sex_Change;

procedure Sex_Change is
  type Single is array (1..1) of aliased T;
  X: Single := (1 => Before);
begin
  A_Name := X(1)'Access;
  X := (1 => After);
end Sex_Change;
```

and then

```
A_Neuter: Mutant_Name(Neuter);          -- fixed neuter

procedure Surgery is new Sex_Change(
                    T => Mutant,
                    Before => (Sex => Neuter),
                    After => (Sex => Male, Bearded, True),
                    Name => Mutant_Name,
                    A_Name => A_Neuter);

Surgery;                                -- call of Surgery makes A_Neuter hairy
```

The problem here is that there are loopholes in the checks in the procedure Sex_Change. The object A_Name is assigned an access to the single component of the array X whose value is Before. When this is done there is a check that the component of the array has the correct subtype. However the subsequent assignment to the whole array changes the value of the component to After and this can change the subtype of X(1) surreptitiously and there is no check concerning A_Name. The key point is that the generic doesn't know that the type T is mutable; this information is not part of the generic contract.

So when we call Surgery, the object A_Neuter suddenly finds that it has grown a beard!

A similar difficulty occurs when private types are involved because the partial view and full view might disagree about whether the type is constrained or not. Consider

```
package Beings is
  type Mutant is private;
  type Mutant_Name is access Mutant;
  F, M: constant Mutant;
private
  type Mutant(Sex: Gender := Neuter) is
    record
      ...                        -- as above
    end record;

  F: constant Mutant := (Female, ... );
  M: constant Mutant := (Male, ... );
end Beings;
```

Now suppose some innocent user (who has not peeked at the private part) writes

```
Chris: Mutant_Name := new Mutant'(F);      -- OK
...
Chris.all := M;                            -- raises Constraint_Error
```

This is very surprising. The user cannot see that the type Mutant is mutable and in particular cannot see that M and F are different in some way. From the outside they just look like constants of the same type. The big trouble is that there is a rule in Ada 95 that says that an object created by an allocator is constrained. So the new object referred to by Chris is permanently Female and therefore the attempt to assign the value of M with its Bearded component to her is doomed.

Attempting to fix these and related problems with a number of minimal rules seemed fated not to succeed. In the end the approach has been taken of getting to the root of the matter in Ada 2005 and disallowing access subtypes for general access types that have defaults for their discriminants. So both the explicit Things_Name and also Mutant_Name(Neuter) are forbidden in Ada 2005.

Moreover we cannot even have an access type such as Mutant_Name when the access type completes a private view that has no discriminants.

By removing these nasty access subtypes it is now possible to say that heap objects are no longer considered constrained in this situation.

The other change in this area concerns type conversions. A variation on the gender theme is illustrated by the following

```
type Gender is (Male, Female);

type Person(Sex: Gender) is
  record
    Birth: Date;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Children: Integer;
    end case;
  end record;
```

Note that this type is not mutable so all persons are stuck with their sex from birth.

We might now declare some access types

```
type Person_Name is access all Person;
type Mans_Name is access all Person(Male);
type Womans_Name is access all Person(Female);
```

so that we can manipulate various names of people. We would naturally use Person_Name if we did not know the sex of the person and otherwise use Mans_Name or Womans_Name as appropriate. We might have

```
It: Person_Name := Chris'Access;
Him: Mans_Name := Jack'Access;
Her: Womans_Name := Jill'Access;
```

If we later discover that Chris is actually Christine then we might like to assign the value in It to a more appropriate variable such as Her. So we would like to write

```
Her := Womans_Name(It);
```

But curiously enough this is not permitted in Ada 95 although the reverse conversion

```
It := Person_Name(Her);
```

is permitted. The Ada 95 rule is that any constraints have to statically match or the conversion has to be to an unconstrained type. Presumably the reason was to avoid checks at run time. But this lack of symmetry is unpleasant and the rule has been changed in Ada 2005 to allow conversion in both directions with a run time check as necessary.

The above example is actually Exercise 19.8(1) in the textbook [2]. The poor student was invited to solve an impossible problem. But they will be successful in Ada 2005.

References

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

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