John Barnes

Ada Rationale 2012

Contracts and Aspects

Courtesy of

AdaCore

Rationale for Ada 2012: 1 Contracts and aspects

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 the mechanisms for including contracts in Ada 2012.

The main feature is that preconditions and postconditions can be given for subprograms. In addition, invariants can be given for types and predicates can be given for subtypes.

In attempting to find a satisfactory way of adding these features it was found expedient to introduce the concept of an aspect specification for describing properties of entities in general. It is thus convenient to describe aspect specifications in this paper.

Keywords: rationale, Ada 2012.

1 Overview of changes

The WG9 guidance document [1] identifies very large complex systems as a major application area for Ada. It further identifies four areas for improvements, one of which is

> Improving the ability to write and enforce contracts for Ada entities (for instance, via preconditions).

The idea of contracts has been a cornerstone of programming for many years. The very idea of specifying parameters for subroutines is a simple form of contract going back to languages such as Fortran over half a century ago. More recently the idea of contracts has been brought to the fore by languages such as SPARK and Eiffel.

SPARK is, as many readers will be aware, a subset of Ada with annotations providing assertions regarding state embedded as Ada comments. The subset excludes features such as access types and dynamic dispatching but it does include Ravenscar tasking and generics. The subset was chosen to enable the contracts to be proved prior to execution. Thus SPARK is a very appropriate vehicle for real programs that just have to be correct because of concerns of safety and security.

Eiffel, on the other hand, is a language with a range of dynamic facilities much as in Ada and has found favour as a vehicle for education. Eiffel includes mechanisms describing contracts which are monitored on a dynamic basis at program execution.

The goal of this amendment to Ada is to incorporate matters such as pre- and postconditions but with the recognition that they are, like those in Eiffel, essentially for checking at runtime.

Adding pre- and postconditions and similar features has had quite a wide ranging impact on Ada and has required much more flexibility in many areas such as the form of expressions which will be addressed in later papers.

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

145	Pre- and postconditions
146	Type invariants
153	Subtype predicates
183	Aspect specifications
191	Aliasing predicates
228	Default initial values for types
229	Specifiable aspects
230	Inheritance of null procedures with precondition
243	Clarification of categorization
247	Preconditions, postconditions, multiple inheritance and dispatching calls

250 Thoughts on type invariants

254 Do we really have contracts right?

267 Improvements for aspect specifications

287 Some questions on subtype predicates

289 Invariants and in mode parameters

297 First_Valid and Last_Valid attributes

These changes can be grouped as follows.

First we lay the syntactic foundations necessary to introduce features such as preconditions by discussing aspect specifications which essentially replace or provide an alternative to pragmas for specifying many features (183, 229, 243, 267).

Then we discuss the introduction of pre- and postconditions on subprograms including the problems introduced by multiple inheritance (145, 230, 247, 254).

Two other related topics are type invariants and subtype predicates which provide additional means of imposing restrictions on types (146, 153, 250, 287, 289, 297).

Finally, two auxiliary features are the ability to provide default values for scalar types and array types (228) and means of checking that aliasing does not occur between two objects (191).

2 Aspect specifications

Although in a sense the introduction of aspect specifications is incidental to the main themes of Ada 2012 which are contracts, real-time, and containers, the clarity (and some might say upheaval) brought by aspect specifications merits their description first.

An early proposal to introduce preconditions was by the use of pragmas. Thus to give a precondition **not** Is_Full to the usual Push procedure acting on a stack S and a corresponding postcondition **not** Is_Empty, it was proposed that this should be written as

```
pragma Precondition(Push, not Is_Full(S));
pragma Postcondition(Push, not Is_Empty(S));
```

But this looks ugly and is verbose since it mentions Push in both pragmas. Moreover, potential problems with overloading means that it has to be clarified to which procedure Push they apply if there happen to be several. As a consequence it was decreed that the pragmas had to apply to the immediately preceding subprogram. Which of course is not the case with pragma Inline which with overloading applies to all subprograms with the given name. Other curiosities include the need to refer to the formal parameters of Push (such as S) so that the expression has to be resolved taking heed of these even though it is detached from the actual specification of Push.

Other pragmas proposed were Inherited_Precondition and Inherited_Postcondition for use with dispatching subprograms.

So it was a mess and an alternative was sought. The solution which evolved was to get away from wretched pragmas in such circumstances. Indeed, the Ada 83 Rationale [2] says "In addition, a program text can include elements that have no influence on the meaning of the program but are included as information and guidance for the human reader or for the compiler. These are: Comments; Pragmas..."

So pragmas were meant to have no effect on the meaning of the program. Typical pragmas in Ada 83 were List, Inline, Optimize and Suppress. But in later versions of Ada, pragmas are used for all sorts of things. The days when pragmas had no effect are long gone!

The basic need was to tie the pre- and postconditions syntactically to the specification of Push so that there could be no doubt as to which subprogram they applied; this would also remove the need to mention the name of the subprogram again. And so, as described in the introductory paper (in the previous issue of this esteemed journal) we now have

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S),
    Post => not Is_Empty(S);
```

The syntax for aspect specification is

```
aspect_specification ::=
  with aspect_mark [ => expression] { ,
       aspect_mark [ => expression] }
```

and this can be used with a variety of structures, subprogram declaration being the example here.

Note especially the use of the reserved word **with**. Serious attempts were made to think of another word so as to avoid using **with** again but nothing better was suggested. It might be thought that it would be confusing to use **with** which is firmly associated with context clauses. However, recall that **with** has also been used to introduce generic formal subprogram parameters without causing confusion since 1983. Thus

```
generic
  with function This ...
procedure That ...
```

Moreover, Ada 95 introduced the use of **with** for type extension as in

```
type Circle is new Object with
  record
    Radius: Float;
  end record;
```

So in Ada 95 there were already many distinct uses of **with** and another one will surely do no harm. It's a versatile little word.

Any risk of confusion is easily avoided by using a sensible layout. Thus a **with** clause should start on a new line at the left and aligned with the following unit to which it applies. A formal generic parameter starting with **with** should be aligned with other formal parameters and indented after the word generic. In the case of type extension, **with** should be at the end of the line. Finally, in the case of aspect specifications, **with** should be at the beginning of a line and indented after the entity to which it applies.

Having introduced aspect specifications which are generally so much nicer than pragmas, it was decided to allow aspect specifications for all those situations where pragmas are used and an aspect specification makes sense (typically where it applies to an entity rather than a region of text). And then to make most of the pragmas obsolete.

Before looking at the old pragmas concerned in detail, two general points are worth noting.

The usual linear elaboration rules do not apply to the expression in an aspect specification. It is essentially sorted out at the freezing point of the entity to which the aspect applies. The reason for this was illustrated by an example in the Introduction which was

```
type Stack is private
  with
    Type_Invariant => Is_Unduplicated(Stack);
```

The problem here is that the function Is_Unduplicated cannot be declared before that of the type Stack and yet it is needed in the aspect specification of the declaration of Stack. So there is a circularity which is broken by saying that the elaboration of aspect specifications is deferred.

The other general point is that some aspects essentially take a Boolean value. For example the pragma Inline is replaced by the aspect Inline so that rather than writing

```
procedure Do_It( ... );
pragma Inline(Do_It);
```

we now write

```
procedure Do_It( ... )
  with Inline;
```

The aspect Inline has type Boolean and so we could write

```
procedure Do_It( ... )
  with Inline => True;
```

To have insisted on this would have been both pedantic and tedious and so in the case of a Boolean aspect there is a rule that says that => True can be omitted and True is then taken by default. But this does not apply to Default_Value and Default_Component_Value as explained later in the section on default initial values.

Note however that omitting the whole aspect by just writing

```
procedure Do_It( ... );
```

results of course in the Inline aspect of Do_It being False.

A mad programmer could even use defaults for preconditions and postconditions. Thus writing

```
procedure Curious( ... )
  with Pre;
```

in which by default the precondition is taken to be True, results in the Curious procedure always being callable.

We will now consider the fate of the various pragmas in Ada 2005. Some are replaced by aspect specifications and the pragmas made obsolete (of course, they can still be used, but should be discouraged in new programs). Some are paralleled by aspect specifications and the user left with the choice. Some are unchanged since for various reasons aspect specifications were inappropriate. Some pragmas are new to Ada 2012 and born obsolete.

The following are the obsolete pragmas with some examples of corresponding aspect specifications.

The pragmas Inline, No_Return, and Pack are examples having Boolean aspects. We can now write

```
procedure Do_It( ... )
  with Inline;

procedure Fail( ... )
  with No_Return;

type T is ...
  with Pack;
```

Some thought was given as to whether the name of the Pack aspect should be Packing rather than Pack because this gave better resonance in English. But the possible confusion in having a different name to that of the pragma overrode the thought of niceties of (human) language.

Curiously enough the old pragmas Inline and No_Return could take several subprograms as argument but naturally the aspect specification is explicitly given to each one.

If several aspects are given to a procedure then we simply put them together thus

```
procedure Kill
  with Inline, No_Return;
```

rather than having to supply several pragmas (which careless program maintenance might have scattered around).

In the case of a procedure without a distinct specification, the aspect specification goes in the procedure body before **is** thus

```
procedure Do_It( ... )
  with Inline is
  ...
begin
  ...
end Do_It;
```

This arrangement is because the aspect specification is very much part of the specification of the subprogram. This will be familiar to users of SPARK where we might have

```
procedure Do_It( ... )
--# global in out Stuff;
is ...
```

If a subprogram has a distinct specification then we cannot give a language-defined aspect specification on the body; this avoids problems of conformance. If there is a stub but
no specification then any aspect specification goes on the stub but not the body. Thus aspect specifications go on the first of specification, stub, and body but are never repeated. Note also that we can give aspect specifications on other forms of stubs and bodies such as package bodies, task bodies and entry bodies but none are defined by the language.

In the case of a stub, abstract subprogram, and null subprogram which never have bodies, the aspect specification goes after **is separate**, **is abstract** or **is null** thus

```
procedure Action(D: in Data) is separate
  with Convention => C;

procedure Enqueue( ... ) is abstract
  with Synchronization => By_Entry;

procedure Nothing is null
  with Something;
```

The above example of the use of Synchronization is from the package Synchronized_Queue_
Interfaces, a new child of Ada.Containers as mentioned in the Introduction.

The same style is followed by the newly introduced expression functions thus

```
function Inc (A: Integer) return Integer is (A + 1)
   with Inline;
```

Other examples of Boolean aspects are Atomic, Volatile, and Independent. We now write for example

```
Converged: Boolean := False
  with Atomic;
```

The aspects Atomic_Components, Volatile_Components and Independent_Components are similar.

The three pragmas Convention, Import and Export are replaced by five aspects, namely Import, Export, Convention, External_Name and Link_Name.

For example, rather than, (see [3] page 702)

```
type Response is access procedure (D: in Data);
pragma Convention(C, Response);

procedure Set_Click(P: in Response);
pragma Import(C, Set_Click);

procedure Action(D: in Data) is separate;
pragma Convention(C, Action);
```

we now more neatly write

```
type Response is access procedure (D: in Data)
  with Convention => C;

procedure Set_Click(P: in Response)
  with Import, Convention => C;

procedure Action(D: in Data) is separate
  with Convention => C;
```

Note that the aspects can be given in any order whereas in the case of pragmas, the parameters had to be in a particular order. We could have written **with** Import => True but that would have been pedantic. As another example (see the RM 7.4), instead of

```
CPU_Identifier: constant String(1 .. 8);
pragma Import(Assembler, CPU_Identifier, Link_Name => "CPU_ID");
```

we now have

```
CPU_Identifier: constant String(1 .. 8)
  with Import, Convention => Assembler, Link_Name => "CPU_ID";
```

Observe that we always have to give the aspect name such as Convention whereas with pragmas Import and Export, the parameter name Convention was optional. Clearly it is better to have to give the name.

The pragma Controlled which it may be recalled told the system to keep its filthy garbage collector off my nice access type is plain obsolete and essentially abandoned. It is doubted whether it was ever used. The subclause of the RM (13.11.3) relating to this pragma is now used by a new pragma Default_Storage_Pools which will be discussed in a later paper.

The pragma Unchecked_Union is another example of a pragma replaced by a Boolean aspect. So we now write

```
type Number(Kind: Precision) is
  record
    ...
  end record
  with Unchecked_Union;
```

Many obsolete pragmas apply to tasks. The aspect Storage_Size takes an expression of any integer type. Thus in the case of a task type without a task definition part (and thus without **is** and matching **end**) we write

```
task type T
  with Storage_Size => 1000;
```

In the case of a task type with entries we write

```
task type T
  with Storage_Size => 1000 is
  entry E ...
  ...
end T;
```

The interrupt pragmas Attach_Handler and Interrupt_Handler now become

```
procedure P( ... )
  with Interrupt_Handler;
```

which specifies that the protected procedure P can be a handler and

```
procedure P( ... )
  with Attach_Handler => Some_Id;
```

which actually attaches P to the interrupt Some_Id.

The pragmas Priority and Interrupt_Priority are replaced by corresponding aspect specifications for example

```
task T
  with Interrupt_Priority => 31;

protected Object
  with Priority => 20 is              -- ceiling priority
```

Note that a protected type or singleton protected object always has **is** and the aspect specification goes before it.

Similarly, instead of using the pragma Relative_Deadline we can write

```
task T
  with Relative_Deadline => RD;
```

The final existing pragma that is now obsolete is the pragma Asynchronous used in the Distributed Systems Annex and which can be applied to a remote procedure or remote access type. It is replaced by the Boolean aspect Asynchronous.

That covers all the existing Ada 2005 pragmas that are now obsolete.

Two new pragmas in Ada 2012 are CPU and Dispatching_Domain but these are born obsolete. Thus we can write either of

```
task My Task is
  pragma CPU(10);
```

or

```
task My_Task
  with CPU => 10 is
```

and similarly

```
task Your_Task is
  pragma Dispatching_Domain(Your_Domain);
```

or

```
task Your_Task
  with Dispatching_Domain => Your_Domain is
```

The reason for introducing these pragmas is so that existing tasking programs with copious use of pragmas such as Priority can use the new facilities in a similar style. It was considered inelegant to write

```
task My_Task
  with CPU => 10 is
  pragma Priority(5);
```

and a burden to have to change programs to

```
task My_Task
  with CPU => 10, Priority => 5 is
```

So existing programs, can be updated to

```
task My_Task is
  pragma CPU(10);
  pragma Priority(5);
```

(One other pragma that was never born at all was Implemented which turned into the aspect Synchronization often used to ensure that an abstract procedure is actually implemented by an entry as illustrated earlier.)

A number of existing pragmas are paralleled by aspect specifications but the pragmas are not made obsolete. Examples are the pragmas relating to packages such as Pure, Preelaborate, Elaborate_Body and so on.

Thus we can write either of

```
package P is
  pragma Pure(P);
end P;
```

or

```
package P
  with Pure is
end P;
```

The author prefers the former but some avant garde programmers might like to use the latter.

Note that Preelaborable_Initialization is unusual in that it cannot be written as an aspect specification for reasons that need not bother us. The inquisitive reader can refer to AI-229 for the details.

Finally, there are many pragmas that do not relate to any particular entity and so for which an aspect specification would be impossible. These include Assert and Assertion_Policy, Suppress and Unsuppress, Page and List, Optimize and Restrictions.

As well as replacing pragmas, aspect specifications can be used instead of attribute definition clauses. For example rather than

```
type Byte is range 0 .. 255;
```

followed (perhaps much later) by

```
for Byte'Size use 8;
```

we can now write

```
type Byte is range 0 .. 255
  with Size => 8;
```

Similarly

```
type My_Float is digits 20
  with Alignment => 16;

Loose_Bits: array (1 .. 10) of Boolean
  with Component_Size => 4;

type Cell_Ptr is access Cell
  with Storage_Size => 500 * Cell'Size / Storage_Unit, Storage_Pool => Cell_Ptr_Pool;

S: Status
  with Address => 8#100#;

type T is delta 0.1 range –1.0 .. +1.0
  with Small => 0.1;
```

But we cannot use this technique to replace an enumeration representation clause or record representation clause. Thus although we can write

```
type RR is
  record
    Code: Opcode;
    R1: Register;
    R2: Register;
  end record
    with Alignment => 2, Bit_Order => High_Order_First;
```

the layout information has to be done by writing

```
for RR use
  record
    Code at 0 range 0 .. 7;
    R1 at 1 range 0 .. 3;
    R2 at 1 range 4 .. 7;
  end record;
```

It is interesting to note that attribute definition clauses were not made redundant in the way that many pragmas were made redundant. This is because there are things that one can do with attribute definition clauses that cannot be done with aspect specifications. For example a visible type can be declared in a visible part and then details of its representation can be given in a private part. Thus we might have

```
package P is
  type T is ...
private
  Secret_Size: constant := 16;
  for T'Size use Secret_Size;
end P;
```

It's not that convincing because the user can use the attribute T'Size to find the Secret_Size anyway. But some existing programs are structured like that and hence the facility could hardly be made redundant.

The examples above have shown aspect specifications with the following constructions: subprogram declaration, subprogram body, stub, abstract subprogram declaration, null procedure declaration, full type declaration, private type declaration, object declaration, package declaration, task type declaration, single task declaration, and single protected declaration. In addition they can be used with subtype declaration, component declaration, private extension declaration, renaming declaration, protected type declaration, entry declaration, exception declaration, generic declaration, generic instantiation, and generic formal parameter declaration.

The appropriate layout should be obvious. In the case of a large structure such as a package specification and any body, the aspect specification goes before **is**. But when something is small and all in one piece such as a procedure specification, stub, null procedure, object declaration or generic instantiation any aspect specification goes at the end of the declaration; it is then more visible and less likely to interfere with the layout of the rest of the structure.

In some cases such as exception declarations there are no language defined aspects that apply but implementations might define their own aspects.

3 Preconditions and postconditions

We will look first at the simple case when inheritance is not involved and then look at more general cases. Specific preconditions and postconditions are applied using the aspects Pre and Post respectively whereas class wide conditions are applied using the aspects Pre'Class and Post'Class.

To apply a specific precondition Before and/or a specific postcondition After to a procedure P we write

```
procedure P(P1: in T1; P2: in out T2; P3: out T3)
  with Pre => Before,
       Post => After;
```

where Before and After are expressions of a Boolean type (that is of type Boolean or a type derived from it).

The precondition Before and the postcondition After can involve the parameters P1 and P2 and P3 and any visible entities such as other variables, constants and functions. Note that Before can involve an **out** parameter such as P3 (if necessary it will be copied in to enable this).

The attribute X'Old will be found useful in postconditions; it denotes the value of X on entry to P. Old is typically applied to parameters of mode **in out** such as P2 but it can be applied to any visible entity such as a global variable. This can be useful for monitoring global variables which are updated by the call of P. But note that 'Old can only be used in postconditions and not in arbitrary text and it cannot be applied to objects of a limited type.

Perhaps surprisingly 'Old can also be applied to parameters of mode **out**. For example, in the case of a parameter of a record type that is updated as a whole, nevertheless we might want to check that a particular component has not changed. Thus in updating some personal details, such as address and occupation, we might want to ensure that the person's date of birth and sex are not tampered with by writing

```
Post => P.Sex = P.Sex'Old and P.Dob = P.Dob'Old
```

In the case of an array, we can write A(I)'Old which means the original value of A(I). But A(I'Old) is different since it is the component of the final value of A but indexed by the old value of I.

Remember that the result of a function is an object and so 'Old can be applied to it. Note carefully the difference between F(X)'Old and F(X'Old). The former applies F to X on entry to the subprogram and saves it. The latter saves X and applies F to it when the postcondition is evaluated. These could be different because the function F might also involve global variables which have changed.

Generally 'Old can be applied to anything but there are restrictions on its use in certain conditional structures in which it can only be applied to statically determined objects. This is illustrated by the following (based on an example in the AARM)

```
Table: array (1 .. 10) of Integer := ... ;
procedure P(I: in out Natural)
  with Post => I > 0 and then Table(I)'Old = 1;    -- illegal
```

The programmer's intent is that the postcondition uses a short circuit form to avoid evaluating Table(I) if I is not positive on exit from the procedure. But, 'Old is evaluated and stored on entry and this could raise Constraint_Error because I might for example be zero. This is a conundrum since the compiler cannot know whether the value of Table(I) will be needed and also I can change so it cannot know which I anyway. So such structures are forbidden.

(The collector of Ada curiosities might be amused to note that we can write

```
subtype dlo is Character;
```

and then in a postcondition we could have

```
dlo'('I')'old
```

which is palindromic. If the subtype were blo rather than dlo then the expression would be mirror reflective!

I am grateful to Jean-Pierre Rosen for this example.)

In the case of a postcondition applying to a function F, the result of the function is denoted by the attribute F'Result. Again this attribute can only be used in postconditions.

Some trivial examples of declarations of a procedure Pinc and function Finc to perform an increment are

```
procedure Pinc(X: in out Integer)
   with Post => X = X'Old+1;

function Finc(X: Integer) return Integer
  with Post => Finc'Result = X'Old+1;
```

Preconditions and postconditions are controlled by the pragma Assertion_Policy. They are enabled by

```
pragma Assertion_Policy(Check);
```

and disabled by using parameter Ignore. It is the value in effect at the point of the subprogram declaration that matters. So we cannot have a situation where the policy changes during the call so that preconditions are switched on but postconditions are off or vice versa.

And so the overall effect of calling P with checks enabled is roughly that, after evaluating any parameters at the point of call, it as if the body were

```
if not Before then                  -- check precondition
  raise Assertion_Error;
end if;

evaluate and store any 'Old stuff;

call actual body of P;

if not After then                   -- check postcondition
  raise Assertion_Error;
end if;
```

```
copy back any by-copy parameters;

return to point of call;
```

Occurrences of Assertion_Error are propagated and so raised at the point of call; they cannot be handled inside P. Of course, if the evaluation of Before or After themselves raise some exception then that will similarly be propagated to the point of call.

Note that conditions Pre and Post can also be applied to entries.

Before progressing to the problems of inheritance it is worth reconsidering the purpose of pre- and postconditions.

> A precondition Before is an obligation on the caller to ensure that it is true before the subprogram is called and it is a guarantee to the implementer of the body that it can be relied upon on entry to the body.
>
> A postcondition After is an obligation on the implementer of the body to ensure that it is true on return from the subprogram and it is a guarantee to the caller that it can be relied upon on return.

The symmetry is neatly illustrated by the diagram below

	Pre	Post
Call writer	obligation	guarantee
Body writer	guarantee	obligation

The simplest form of inheritance occurs with derived types that are not tagged. Suppose we declare the procedure Pinc as above with the postcondition shown and supply a body

```
procedure Pinc(X: in out Integer) is
begin
  X := X+1;
end Pinc;
```

and then declare a type

```
type Apples is new Integer;
```

then the procedure Pinc is inherited by the type Apples. So if we then write

```
No_Of_Apples: Apples;
...
Pinc(No_Of_Apples);
```

what actually happens is that the code of the procedure Pinc originally written for Integer is called and so the postcondition is inherited automatically.

If the user now wants to add a precondition to Pinc that the number of apples is not negative then a completely new subprogram has to be declared which overrides the old one thus

```
procedure Pinc(X: in out Apples)
  with Pre => X >= 0,
       Post => X = X'Old+1;
```

and a new body has to be supplied (which will of course in this curious case be essentially the same as the old one). So we cannot inherit an operation and change its conditions at the same time.

We now turn to tagged types and first continue to consider the specific conditions Pre and Post. As a perhaps familiar example, consider the hierarchy consisting of a type Object and then direct descendants Circle, Square and Triangle.

Suppose the type Object is

```
type Object is tagged
  record
    X_Coord, Y_Coord: Float;
  end record;
```

and we declare a function Area thus

```
function Area(O: Object) return Float
  with Pre => O.X_Coord > 0.0,
       Post => Area'Result = 0.0;
```

This imposes a requirement on the caller that the function is called only with objects with positive *x*-coordinate (for some obscure reason), and a requirement on the implementer of the body that the area is zero (raw objects are just points and have no area).

If we now declare a type Circle as

```
type Circle is new Object with
  record
    Radius: Float;
  end record;
```

and override the inherited function Area then the Pre and Post conditions on Area for Object are not inherited and we have to supply new ones, perhaps

```
function Area(C: Circle)
  with Pre => C.X_Coord – C.Radius > 0.0,
       Post => Area'Result > 3.1 * C.Radius**2 and
               Area'Result < 3.2 * C.Radius**2;
```

The conditions ensure that all of the circle is in the right half-plane and that the area is about right!

So the rules so far are exactly as for the untagged case. If an operation is not overridden then it inherits the conditions from its ancestor but if it is overridden then those conditions are lost and new ones have to be supplied. And if no new ones are supplied then they are by default taken to be True.

In conclusion, the conditions Pre and Post are very much part of the actual body. One consequence of this is that an abstract subprogram cannot have Pre and Post conditions because an abstract subprogram has no body.

We now turn to the class wide conditions Pre'Class and Post'Class which are subtly different. The first point is that the class wide ones apply to all descendants as well even if the operations are overridden. In the case of Post'Class if an overridden operation has no condition given then it is taken to be True (as in the case of Post). But in the case of Pre'Class, if an overridden operation has no condition given then it is only taken to be True if no other Pre'Class applies (no other is inherited). We will now look at the consequences of these rules.

It might be that we want certain conditions to hold throughout the hierarchy, perhaps that all objects concerned have a positive *x*-coordinate and nonnegative area. In that case we can use class wide conditions.

```
function Area(O: Object) return Float
  with Pre'Class => O.X_Coord > 0.0,
       Post'Class => Area'Result >= 0.0;
```

Now when we declare Area for Circle, Pre'Class and Post'Class from Object will be inherited by the function Area for Circle. Note that within a class wide condition a formal parameter of type T is interpreted as of T'Class. Thus O is of type Object'Class and thus applies to Circle. The inherited postcondition is simply that the area is not negative and uses the attribute 'Result.

If we do not supply conditions for the overriding Area for Circle and simply write

```
overriding
function Area(C: Circle) return Float;
```

then the precondition inherited from Object still applies. In the case of the postcondition not only is the postcondition from Object inherited but there is also an implicit postcondition of True. So the applicable conditions for Area for Circle are

```
Pre'Class for Object

Post'Class for Object
True
```

Suppose on the other hand that we give explicit Pre'Class and Post'Class for Area for Circle thus

```
overriding
function Area(C: Circle) return Float
  with Pre'Class => ... ,
       Post'Class => ... ;
```

We then find that the applicable conditions for Area for Circle are

```
Pre'Class for Object
Pre'Class for Circle

Post'Class for Object
Post'Class for Circle
```

Incidentally, it makes a lot of sense to declare the type Object as abstract so that we cannot declare pointless objects. In that case Area might as well be abstract as well. Although we cannot give conditions Pre and Post for an abstract operation we can still give the class wide conditions Pre'Class and Post'Class.

If the hierarchy extends further, perhaps Equilateral_Triangle is derived from Triangle which itself is derived from Object, then we could add class wide conditions to Area for Triangle and these would also apply to Area for Equilateral_Triangle. And we might add specific conditions for Equilateral_Triangle as well. So we would then find that the following apply to Area for Equilateral_Triangle

```
Pre'Class for Object
Pre'Class for Triangle
Pre for Equilateral Triangle

Post'Class for Object
Post'Class for Triangle
Post for Equilateral_Triangle
```

The postconditions are quite straightforward, all apply and all must be true on return from the function Area. The compiler can see all these postconditions when the code for Area is compiled and so they are all checked in the body. Note that any default True makes no difference because B **and** True is the same as B.

However, the rules regarding preconditions are perhaps surprising. The specific precondition Pre for Equilateral_Triangle must be true (checked in the body) but so long as just one of the class wide preconditions Pre'Class for Object and Triangle is true then all is well. Note that class wide preconditions are checked at the point of call. Do not get confused over the use of the word apply. They all apply but only the ones seen at the point of call are actually checked.

The reason for this state of affairs concerns dispatching and especially redispatching. Consider the case of Ada airlines which has Basic, Nice and Posh passengers. Basic passengers just get a seat. Nice passengers also get a meal and Posh passengers also get a limo. The types Reservation, Nice_Reservation and Posh_Reservation form a hierarchy with Nice_Reservation being extended from Reservation and so on. The facilities are assigned when a reservation is made by calling an appropriate procedure Make thus

```
procedure Make(R: in out Reservation) is
begin
  Select_Seat(R);
end Make;

procedure Make(NR: in out Nice_Reservation) is
begin
  Make(Reservation(NR));
  Order_Meal(NR);
end Make;

procedure Make(PR: in out Posh_Reservation) is
  Make(Nice_Reservation(PR));
  Arrange_Limo(PR);
end Make;
```

Each Make calls its ancestor in order to avoid duplication of code and to ease maintenance.

A variation involving redispatching introduces two different procedures Order_Meal, one for Nice passengers and one for Posh passengers. We then need to ensure that Posh passengers get a posh meal rather than a nice meal. We write

```
procedure Make(NR: in out Nice_Reservation) is
begin
  Make(Reservation(NR));
                     -- now redispatch to appropriate Order_Meal
  Order_Meal(Nice_Reservation'Class(NR));
end Make;
```

Now suppose we have a precondition Pre'Class on Order_Meal for Nice passengers and one on Order_Meal for Posh passengers. The call of Order_Meal sees that it is for Nice_Reservation'Class and so the code includes a test of Pre'Class on Nice_Reservation. It does not necessarily know of the existence of the type Posh_Reservation and cannot check Pre'Class on that Order_Meal. At a later date we might add Supersonic passengers (RIP Concorde) and this can be done without recompiling the rest of the system so it certainly cannot do anything about checking Pre'Class on Order_Meal for Supersonic_Reservation which does not exist when the call is compiled. So when we eventually get to the body of one of the procedures Order_Meal all we know is that some Pre'Class on Order_Meal has been checked somewhere. And that is all that the writer of the code of Order_Meal can rely upon. Note that nowhere does the compiled code actually "or" a lot of preconditions together.

In summary, class wide preconditions are checked at the point of call. Class wide postconditions and both specific pre- and postconditions are checked in the actual body.

A small point to remember is that a class wide operation such as

```
procedure Do_It(X: in out T'Class);
```

is not a primitive operation of T and so although we can specify Pre and Post for Do_It we cannot specify Pre'Class and Post'Class for Do_It.

We noted above that the aspects Pre and Post cannot be specified for an abstract subprogram because it doesn't have a body. They cannot be given for a null procedure either, since we want all null procedures to be identical and do nothing and that includes no conditions.

We now turn to the question of multiple inheritance and progenitors.

In the case of multiple inheritance we have to consider the so-called Liskov Substitution Principle (LSP). The usual consequence of LSP is that in the case of preconditions they are combined with "or" (thus weakening) and the rule for postconditions is that they are combined with "and" (thus strengthening). But the important thing is that a relevant concrete operation can be substituted for the corresponding operations of all its relevant ancestors.

In Ada, a type T can have one parent and several progenitors. Thus we might have

```
type T is new P and G1 and G2 with ...
```

where P is the parent and G1 and G2 are progenitors. Remember that a progenitor cannot have components and cannot have concrete operations (apart possibly for null procedures). So the operations of the progenitors have to be abstract or null and cannot have Pre and Post conditions. However, they can have Pre'Class and Post'Class conditions. It is possible that the same operation Op is primitive for more than one of these. Thus the progenitors G1 and G2 might both have an operation Op thus

```
procedure Op(X: G1) is abstract;
procedure Op(X: G2) is abstract;
```

If they are conforming (as they are in this case) then the one concrete operation Op of the type T derived from both G1 and G2 will implement both of these. (If they don't conform then they are simply overloadings and two operations of T are required). Hence the one Op for T can be substituted for the Op of both G1 and G2 and LSP is satisfied.

Now suppose both abstract operations have pre- and postconditions. Take postconditions first, we might have

```
procedure Op(X: G1) is abstract
  with Post'Class => After1;

procedure Op(X: G2) is abstract
  with Post'Class => After2;
```

Users of the Op of G1 will expect the postcondition After1 to be satisfied by any implementation of that Op. So if using the Op of T which implements the abstract Op of G1, it follows that Op of T must satisfy the postcondition After1. By a similar argument regarding G2, it must also satisfy the postcondition After2.

It thus follows that the effective postcondition on the concrete Op of T is as if we had written

```
procedure Op(X: T)
  with Post'Class => After1 and After2;
```

But of course we don't actually have to write that since we simply write

```
overriding
procedure OP(X: T);
```

and it automatically inherits both postconditions and the compiler inserts the appropriate code in the body. Remember that if we don't give a condition then it is True by default but anding in True makes no difference.

If we do provide another postcondition thus

```
overriding
procedure OP(X: T)
  with Post'Class => After_T;
```

then the overall class wide postcondition to be checked before returning will be After1 **and** After2 **and** After_T.

Now consider preconditions. Suppose the declarations of the two versions of Op are

```
procedure Op(X: G1) is abstract
  with Pre'Class => Before1;

procedure Op(X: G2) is abstract
  with Pre'Class => Before2;
```

Assuming that there is no corresponding Op for P, we must provide a concrete operation for T thus

```
overriding
procedure Op(X: T)
  with Pre'Class => Before_T;
```

This means that at a point of call of Op the precondition to be checked is Before_T **or** Before1 **or** Before2. As long as this is satisfied it does not matter that Before1 and Before2 might have been different.

If we do not provide an explicit Pre'Class then the condition to be checked at the point of call is Before1 **or** Before2.

An interesting case arises if a progenitor (say G1) and the parent have a conforming operation. Thus suppose P itself has the operation

```
procedure Op(X: P);
```

and moreover that the operation is not abstract. Then (ignoring preconditions for the moment) this Op for P is inherited by T and thus provides a satisfactory implementation of Op for G1 and all is well.

Now suppose that Op for P has a precondition thus

```
procedure OP(X: P)
  with Pre'Class => Before_P;
```

and that Before_P and Before1 are not the same. If we do not provide an explicit overriding for Op, it would be possible to call the body of Op for P when the precondition it knows about, Before_P, is False (since Before1 being True would be sufficient to allow the call to proceed). This would effectively mean that no class wide preconditions could be trusted within the subprogram body and that would be totally unacceptable. So in this case there is a rule that an explicit overriding is required for Op for T.

If Op for P is abstract then a concrete Op for T must be provided and the situation is just as in the case for the Op for G1 and G2.

If T itself is declared as abstract (and P is not abstract and Op for P is concrete) then the inherited Op for T is abstract.

(These rules are similar to those for functions returning a tagged type when the type is extended; it has to be overridden unless the type is abstract in which case the inherited operation is abstract.)

We finish this somewhat mechanical discussion of the rules by pointing out that if silly inappropriate preconditions are given then we will get a silly program.

At the end of the day, the real point is that programmers should not write preconditions that are not sensible and sensibly related to each other. Because of the generality, the compiler cannot tell so stupid things are hard to prohibit. There is no defence against stupid programmers.

A concrete example using simple numbers might help. Suppose we have a tagged type T1 and an operation Solve which takes a parameter of type T1 and perhaps finds the solution to an equation defined by the components of T1. Solve delivers the answer in a parameter A with a parameter D giving the number of significant digits required in the answer. Also we impose a precondition on the number of digits D thus

```
type T1 is tagged record ...

procedure Solve(X: in T1; A: out Float; D: in Integer)
  with Pre'Class => D < 5;
```

The intent here is that the version of Solve for the type T1 always works if the number of significant digits asked for is less than 5.

Now suppose we declare a type T2 derived from T1 and that we override the inherited Solve with a new version that works if the number of significant digits asked for is less than 10

```
type T2 is new T1 with ...

overriding
procedure Solve(X: in T2; A: out Float; D: in Integer)
  with Pre'Class => D < 10;
```

And so on with a type T3

```
type T3 is new T2 with ...

overriding
procedure Solve(X: in T3; A: out Float; D: in Integer)
  with Pre'Class => D < 15;
```

Thus we have a hierarchy of algorithms Solve with increasing capability.

Now suppose we have a dispatching call

```
An_X: T1'Class := ... ;
Solve(An_X, Answer, Digs);
```

this will dispatch to one of the Solve procedures but we do not know which one. The only precondition that applies is that on the Solve for T1 which is D < 5. That is fine because D < 5 implies D < 10 and D < 15 and so on. Thus the preconditions work because the hierarchy weakens them.

Similarly, if we have

```
An_X: T2'Class := ... ;
Solve(An_X, Answer, Digs);
```

then it will dispatch to a Solve for one of T2, T3, ..., but not to the Solve for T1. The applicable preconditions are D < 5 and D < 10 and these are notionally ored together which means D < 10 is actually required. To see this suppose we supply D = Digs = 7. Then D < 5 is False but D < 10 is True so by oring False and True we get True, so the call works.

On the other hand if we write

```
An_X: T2 := ... ;
Solve(An_X, Answer, Digs);
```

then no dispatching is involved and the Solve for T2 is called. But both class wide preconditions D < 5 and D < 10 apply and so again the resulting ored precondition that is required is D < 10.

Now it should be clear that if the preconditions do not form a weakening hierarchy then we will be in trouble. Thus if the preconditions were D < 15 for T1, D < 10 for T2, and D < 5 for T3, then dispatching from the root will only check D < 15. However, we could end up calling the Solve for T2 which expects the precondition D < 10 and this might not be satisfied.

Care is thus needed with preconditions that they are sensibly related.

4 Type invariants

Type invariants are designed for use with private types where we want some relationship to always hold between components of the type. Like pre- and postconditions there are both specific invariants that can be applied to any type and class wide invariants that can only be applied to tagged types.

One example mentioned above and discussed in the Introduction was a type Stack with specific invariant Is_Unduplicated. Thus we write

```
type Stack is private
  with Type_Invariant  => Is_Unduplicated(Stack);
```

After calls of Push and Pop and any other operations that manipulate the stack, the function Is_Unduplicated is called to ensure that there are no duplicates on the stack.

The monitoring is controlled by the pragma Assertion_Policy in the same way as pre- and postconditions. If an invariant fails (that is, has value False) then Assertion_Error is raised.

The invariant Is_Unduplicated is a curious example because it cannot be violated by Pop anyway since if there were no duplicates then removing the top item cannot make one appear.

Moreover, Push needs to ensure that the item to be added is not a duplicate of one on the stack already and so essentially much of the checking is repeated. Indeed, when writing Push we should be able to assume that no items are already duplicated and hence all we need to do is check that the new item to be added is not equal to one of the existing items (so n comparisons). However, a general function Is_Unduplicated will need to compare all pairs and thus require a double loop (so $n(n+1)/2$ comparisons).

The reader is invited to meditate over this conundrum. One's first reaction might be that this is a bad example. However, one way to ensure reliability is to introduce redundancy. Thus if the encoding of Is_Unduplicated and Push are done independently then there is an increased probability that any error will be detected.

The aspect Type_Invariant requires an expression of a Boolean type. The mad programmer could therefore also write

```
type Stack is private
  with Type_Invariant;
```

which would thus be True by default and so useless! Actually it might not be entirely useless since it might act as a placeholder for an invariant to be defined later and meanwhile the program will compile and execute.

Type invariants are useful whenever a type is more than just the sum of its components. Note carefully that the invariant may not hold when an object is being manipulated by a subprogram having access to the full type. In the case of Push and Pop and the invariant Is_Unduplicated this will not happen but consider the following simple example.

Suppose we have a type Point which describes the position of an object in a plane. It might simply be

```
type Point is
  record
    X, Y: Float;
  end record;
```

Now suppose we want to ensure that all points are within a unit circle. We could ensure that a point lies within a square by means of range constraints by writing

```
type Point is
  record
    X, Y: Float range –1.0 .. +1.0;
  end record;
```

but we need to ensure that X**2 + Y**2 is not greater than 1.0, and that cannot be done by individual constraints. So we might declare a type Disc_Pt with an invariant as follows

```
package Places is

  type Disc_Pt is private
    with Type_Invariant => Check_In(Disc_Pt);

  function Check_In(D: Disc_Pt) return Boolean
    with Inline;
  ...                        -- various operations on disc points
private

  type Disc_Pt is
    record
      X, Y: Float range –1.0 .. +1.0;
    end record;

  function Check_In(D: Disc_Pt) return Boolean is
    (D.X**2 + D.Y**2 <= 1.0);

end Places;
```

Note that we have used an expression function for Check_In. Expression functions were outlined in the Introduction and will be discussed in detail in the next paper. They are very useful for small functions in situations like this and typically will be given the aspect Inline on the specification as shown.

Now suppose that we wish to make available to the user a procedure Flip that reflects a Disc_Pt in the line *x* = *y*, or in other words interchanges its X and Y components. The body might be

```
procedure Flip(D: in out Disc_Pt) is
  T: Float;                  -- temporary
begin
  T := D.X;  D.X := D.Y;  D.Y := T;
end Flip;
```

This works just fine but note that just before the assignment to D.Y, it is quite likely that the invariant does not hold. If the original value of D was (0.1, 0.8) then at the intermediate stage it will be (0.8, 0.8) and so well outside the unit circle.

So there is a general principle that an intermediate value not visible externally need not satisfy the invariant. There is an analogy with numeric types. The intermediate value of an expression can fall outside the range of the type but will be within range when the final value is assigned to the object. For example, suppose type Integer is 16 bits (a small machine) but the registers perform arithmetic in 32 bits, then a statement such as

```
J := K * L / M;
```

could easily produce an intermediate result K * L outside the range of Integer but the final value could be in range.

In many cases it will not be necessary for the user to know that a type invariant applies to the type; it is after all merely a detail of the implementation. So perhaps the above should be rewritten as

```
package Places is

  type Disc_Pt is private;
  ...                          -- various operations on disc points
private

  type Disc_Pt is
    record
      X, Y: Float range –1.0 .. +1.0;
    end record
    with Type_Invariant => Disc_Pt.X**2 + Disc_Pt.Y**2 <= 1.0;

end Places;
```

In this case we do not need to declare a function Check_In at all. Note the use of the type name Disc_Pt in the invariant expression. This is another example of the use of a type name to denote a current instance (this is familiar from way back in Ada 83 with task type names).

We now turn to consider the places where a type invariant on a private type T is checked. These are basically when it can be changed from the point of view of the outside user. They are

- after default initialization of an object of type T,
- after a conversion to type T,
- after assigning to a view conversion involving descendants and ancestors of type T,
- after a call of T'Read or T'Input,
- after a call of a subprogram declared in the immediate scope of T and visible outside that has a parameter (of any mode including an access parameter) with a part of type T or returns a result with a part of type T.

Note that by saying a part of type T, the checks not only apply to subprograms with parameters and results of type T but they also apply to parameters and results whose components are of the type T or are view conversions involving the type T. Observe that parameters of mode **in** are also checked because, as is well known, there are accepted techniques for changing such parameters.

Beware, however, that the checks do not extend to deeply nested situations, such as components with components that are access values to objects that themselves involve type T or worse. Thus there are holes in the protection offered by type invariants. However, if the types are straightforward and the writer does not do foolish things like surreptitiously exporting access types referring to T then all will be well. It is another example of there being no defence against foolish programmers.

The checks on type invariants regarding parameters and results can be conveniently implemented in the body of the subprogram in much the same way as for postconditions. This saves duplicating the code of the tests at each point of call.

If a subprogram such as Flip which is visible outside is called from inside then the checks still apply. This is not strictly necessary of course, but fits the simple model of the checks being in the body and so simplifies the implementation.

If an untagged type is derived then any existing specific invariant is inherited for inherited operations. However, a further invariant can be given as well and both will apply to the inherited operations. This fits in with the model of view conversions used to describe how an inherited subprogram works on derivation. The parameters of the derived type are view converted to the parent type before the body is called and back again afterwards. As mentioned above, view conversions are one of the places where invariants are checked.

However, if we add new operations then the old invariant does not apply to them. In truth, the specific invariant is not really inherited at all; it just comes along for free with the inherited operations that are not overridden. So if we do add new operations then we need to state the total invariant required.

Note that this is not quite the same model as specific postconditions. We cannot add postconditions to an inherited operation but have to override it and then any specific postconditions on the parent are lost. In any event, in both cases, if we want to use inheritance then we should really use tagged types and class wide aspects.

So there is also an aspect Type_Invariant'Class for use with private tagged types. The distinction between Type_Invariant and Type_Invariant 'Class has similarities to that between Post and Post'Class.

The specific aspect Type_Invariant can be applied to any type but Type_Invariant'Class can only be applied to tagged types. A tagged type can have both an aspect Type_Invariant and Type_Invariant'Class.

Type_Invariant cannot be applied to an abstract type.

Type_Invariant'Class is inherited by all derived types; it can also be applied to an abstract type.

Note the subtle difference between Type_Invariant and Type_Invariant'Class. Type_Invariant'Class is inherited for all operations of the type but as noted above Type_Invariant is only incidentally inherited by the operations that are inherited.

An interesting rule is that Type_Invariant'Class cannot be applied to a full type declaration which completes a private type such as Disc_Pt in the example above. This is because the writer of an extension will need to see the applicable invariants and this would not be possible if they were in the private part.

So if we have a type T with a class wide invariant thus

```
type T is tagged private
  with Type_Invariant'Class => F(T);
procedure Op1(X: in out T);
procedure Op2(X: in out T);
```

and then write

```
type NT is new T with private
  with Type_Invariant'Class => FN(NT);
overriding
procedure Op2(X: in out NT);
```

```
not overriding
procedure Op3(X: in out NT);
```

then both invariants F and FN will apply to NT.

Note that the procedure Op1 is inherited unchanged by NT, procedure Op2 is overridden for NT and procedure Op3 is added.

Now consider various calls. The calls of Op1 will involve view conversions as mentioned earlier and these will apply the checks for FN and the inherited body will apply the checks for F. The body of Op2 will directly include checks for F and FN as will the body of Op3. So the invariant F is properly inherited and all is well.

Remember that if the invariants were specific and not class wide then although Op1 will have checks for F and FN, Op2 and Op3 will only check FN.

In the case of the type Disc_Pt we might decide to derive a type which requires that all values are not only inside the unit circle but outside an inner circle – in other words in an annulus or ring. We use the class wide invariants so that the parent package is

```
package Places is

  type Disc_Pt is tagged private
    with Type_Invariant'Class => Check_In(Disc_Pt);

  function Check_In(D: Disc_Pt) return Boolean
    with Inline;
  ...                          -- various operations on disc points
private

  type Disc_Pt is tagged
    record
      X, Y: Float range –1.0 .. +1.0;
    end record;

  function Check_In(D: Disc_Pt) return Boolean is
    (D.X**2 + D.Y**2 <= 1.0);

end Places;
```

And then we might write

```
package Places.Inner is

  type Ring_Pt is new Disc_Pt with null record
    with Type_Invariant'Class => Check_Out(Ring_Pt);

  function Check_Out(R: Ring_Pt) return Boolean
    with Inline;

private

  function Check_Out(R: Ring_Pt) return Boolean is
    (R.X**2 + R.Y**2 >= 0.25);

end Places.Inner;
```

And now the type Ring_Pt has both its own type invariant but also that inherited from Disc_Pt thereby ensuring that points are within the ring or annulus. It is unfortunate that we could not make the size of the inner circle a discriminant but a discriminant cannot be of a real type. Ah well, perhaps in Ada 2020??

Finally, it is worth emphasizing that it is good advice not to use inheritance with specific invariants but they are invaluable for checking internal and private properties of types.

5 Subtype predicates

The final major facility to be discussed here is subtype predicates. These are not really contractual in the sense that preconditions, postconditions and invariants are contractual but are more akin to constraints.

Subtype predicates are of two kinds, Static_Predicate and Dynamic_Predicate. They can be applied to subtype declarations and to type declarations using aspect specifications. For example, in the Introduction we met

```
subtype Even is Integer
  with Dynamic_Predicate => Even mod 2 = 0;

subtype Winter is Month
  with Static_Predicate => Winter in Dec | Jan | Feb;
```

The predicates take an expression of a Boolean type and again we note the use of the subtype name to denote the current instance. In the case of Dynamic_Predicate, the expression can be any Boolean expression.

However, in the case of Static_Predicate, the expression is restricted and can only be

- a static membership test where the choice is selected by the current instance,
- a case expression whose dependent expressions are static and selected by the current instance,
- a call of the predefined operations =, /=, <, <=, >, >= where one operand is the current instance,
- an ordinary static expression,

and, in addition, a call of a Boolean logical operator **and**, **or**, **xor**, **not** whose operands are such static predicate expressions, and, a static predicate expression in parentheses.

So we see that the predicate in the subtype Even cannot be a static predicate because the operator **mod** is not permitted with the current instance. But **mod** could be used in an inner static expression.

However, the predicate in the subtype Winter can be a static predicate because it takes the from of a membership test where the choice is selected by the current instance and whose individual items are static. Note that membership tests are considerably enhanced in Ada 2012; further details will be given in a later paper. Another useful example of this kind is

```
subtype Letter is Character
  with Static_Predicate => Letter in 'A' .. 'Z' | 'a' .. 'z';
```

Static case expressions are valuable because they provide the comfort of covering all values of the current instance. Suppose we have a type Animal

```
type Animal is (Bear, Cat, Dog, Horse, Wolf);
```

We could then declare a subtype of friendly animals

```
subtype Pet is Animal
  with Static_Predicate => Pet in Cat | Dog | Horse;
```

and perhaps

```
subtype Predator is Animal
  with Static_Predicate => not (Predator in Pet);
```

or equivalently

```
subtype Predator is Animal
  with Static_Predicate => Predator not in Pet;
```

Now suppose we add Rabbit to the type Animal. Assuming that we consider that rabbits are pets and not food, we should change Pet to correspond but we might forget with awkward results. Maybe we have a procedure Hunt which aims to eliminate predators

```
procedure Hunt(P: in out Predator);
```

and we will find that our poor rabbit is hunted rather than petted!

What we should have done is use a case expression controlled by the current instance thus

```
subtype Pet is Animal
  with Static_Predicate =>
    (case Pet is
          when Cat | Dog | Horse => True,
          when Bear | Wolf => False);
```

and now if we add Rabbit to Animal and forget to update Pet to correspond then the program will fail to compile.

Note that a similar form of if expression where the current instance has to be of a Boolean type would not be useful and so is excluded.

Static subtypes with static predicates can also be used in case statements. Thus elsewhere in the program we might have

```
case Animal is
  when Pet =>          ...       -- feed it
  when Predator =>     ...       -- feed on it
end case;
```

Observe that we do not have to list all the individual animals and naturally there is no others clause. If other animals are added to Pet or Predator then this case statement will not need changing. Thus not only do we get the benefit of full coverage checking, but the code is also maintenance free. Of course if we add an animal that is neither a Pet nor Predator (Sloth perhaps?) then the case statement will need updating.

Subtype predicates, like pre- and postconditions and type invariants are similarly monitored by the pragma Assertion_Policy. If a predicate fails (that is, has value False) then Assertion_Error is raised.

Subtype predicates are checked in much the same sort of places as type invariants. Thus

- on a subtype conversion,
- on parameter passing (which covers expressions in general),
- on default initialization of an object.

Note an important difference from type invariants. If a type invariant is violated then the damage has been done. But subtype predicates are checked before any damage is done. This difference essentially arises because type invariants apply to private types and can become temporarily false inside the defining package as we saw with the procedure Flip applying to the type Disc_Pt.

If an object is declared without initialization and no default applies then any subtype predicate might be false in the same way that a subtype constraint might be violated.

Beware that subtype predicates like type invariants are not foolproof. Thus in the case of a record type they apply to the record as a whole but they are not checked if an individual component is modified.

Subtype predicates can be given for all types in principle. Thus we might have

```
type Date is
  record
    D: Integer range 1 .. 31;
    M: Month;
    Y: Integer;
  end record;
```

and then

```
subtype Winter_Date is Date
  with Dynamic_Predicate => Winter_Date.M in Winter;
```

Note how this uses the subtype Winter which was itself defined by a subtype predicate. However, Winter_Date has to have a Dynamic_Predicate because the selector is not simply the current instance but a component of it.

We can now declare and manipulate a Winter_Date

```
WD: Winter_Date := (25, Dec, 2011);
...
Do_Date(WD);
```

and the subtype predicate will be checked on the call of Do_Date. However, beware that if we write

```
WD.Month := Jun;                    -- dodgy
```

then the subtype predicate is not checked because we are modifying an individual component and not the record as a whole.

Subtype predicates can be given with type declarations as well as with subtype declarations. Consider for example declaring a type whose only allowed values are the possible scores for an individual throw when playing darts. These are 1 to 20 and doubles and trebles plus 50 and 25 for an inner and outer bull's eye. We could write these all out explicitly

```
type Score is new Integer
  with Static_Predicate =>
     Score in 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21
              | 22 | 24 | 25 | 26 | 27 | 28 | 30 | 32 | 33 | 34 | 36 | 38 | 39 | 40 | 42 | 45 | 48 | 50
              | 51 | 54 | 57 | 60;
```

But that is rather boring and obscures the nature of the predicate. We can split it down by first defining individual subtypes for singles, doubles and trebles as follows

```
subtype Single is Integer range 1 .. 20;

subtype Double is Integer
  with Static_Predicate =>
     Double in 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20;

subtype Treble is Integer
  with Static_Predicate =>
    Treble in 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30;
```

```
subtype Score is Integer
  with Static_Predicate =>
    Score in Single or Score in Double or Score in Treble or Score in 25 | 50;
```

Note that it would be neater to write

```
subtype Score is Integer
  with Static_Predicate =>
    Score in Single | Double | Treble | 25 | 50;
```

Observe that it does not matter that the individual predicates overlap. That is a score such as 12 is a Single, a Double and a Treble.

If we do not mind the predicates being dynamic then we can write

```
subtype Double is Integer
  with Dynamic_Predicate =>
    Double mod 2 = 0 and Double / 2 in Single;
```

and so on. Or we could even use a quantified expression

```
subtype Double is Integer
  with Dynamic_Predicate =>
    (for some K in Single => Double = 2*K);
```

or go all the way in one lump

```
type Dyn_Score is new Integer
  with Dynamic_Predicate =>
    (for some K in 1 .. 20 => Score = K or Score = 2*K or Score = 3*K) or Score in 25 | 50;
```

There are some restrictions on the use of subtypes with predicates.

If a subtype has a static or dynamic predicate then it cannot be used as an array index subtype. This is to avoid arrays with holes. So we cannot write

```
type Winter_Hours is array (Winter) of Hours;            -- illegal

type Hits is array (Score range <>) of Integer;          -- illegal
```

Similarly, we cannot use a subtype with a predicate to declare the range of an array object or to select a slice. So if we have

```
type Month_Days is array (Month range <>) of Integer;
The_Days: Month_Days := (31, 28, 31, 30, ... );
```

then we cannot write

```
Winter_Days: Month_Days(Winter);                         -- illegal array

The_Days(Winter) := (Jan | Dec => 31, Feb => 29);        -- really nasty illegal slice
```

However, a subtype with a static predicate can be used in a for loop thus

```
for W in Winter loop ...
```

and in a named aggregate such as

```
(Winter => 10.0, others => 14.0);                 -- OK
```

but a subtype with a dynamic predicate cannot be used in these ways. Actually the restriction is slightly more complicated. If the original subtype is not static such as

```
subtype To_N is Integer range 1 .. N;
```

then even if To_N has a static predicate it still cannot be used in a for loop or named aggregate.

These rules can also be illustrated by considering the dartboard. We might like to accumulate a count of the number of times each particular score has been achieved. So we might like to declare

```
type Hit_Count is array (Score) of Integer;          -- illegal
```

but sadly this would result in an array with holes and so is forbidden. However, we could declare an array from 1 to 60 and then initialize it with 0 for those components used for hits and –1 for the unused components. Of course, we ought not to repeat literals such as 1 and 60 because of potential maintenance problems. But, we can use new attributes First_Valid and Last_Valid thus

```
type Hit_Count is array (Score'First_Valid .. Score'Last_Valid) of Integer :=
                                                            (Score => 0, others => –
1);
```

which uses Score to indicate the used components. The attributes First_Valid and Last_Valid can be applied to any static subtype but are particularly useful with static predicates.

In detail, First_Valid returns the smallest valid value of the subtype. It takes any range and/or predicate into account whereas First only takes the range into account. Similarly Last_Valid returns the largest value. Incidentally, they are illegal on null subtypes (because null subtypes have no valid values at all).

The Hit_Count array can then be updated by the value of each hit as expected

```
A_Hit: Score := ... ;                    -- next dart

Hit_Count(A_Hit) := Hit_Count(A_Hit) + 1;
```

If we attempt to assign a value of type Integer which is not in the subtype Score to A_Hit then Assertion_Error is raised.

After the game, we can now loop through the subtype Score and print out the number of times each hit has been achieved and perhaps accumulate the total at the same time thus

```
for K in Score loop
  New_Line;  Put(Hit);  Put(Hit_Count(K));
  Total := Total + K * Hit_Count(K);
end loop;
```

The reason for the distinction between static and dynamic predicates is that the static form can be implemented as small sets with static operations on the small sets. Hence the loop

```
for K in Score loop ...
```

can be implemented simply as a sequence of 43 iterations. However, a loop such as

```
for X in Even loop ...
```

which might look innocuous requires iterating over the whole set of integers. Thus we insist on having to write

```
for X in Integer loop
  if X in Even then ...
```

which makes the situation quite clear.

Another restriction on the use of subtypes with predicates is that the attributes First, Last and Range cannot be applied. But Pred and Succ are permitted because they apply to the underlying type. As a consequence, if a generic body uses First, Last or Range on a formal type and the actual type has a subtype predicate then Program_Error is raised.

Subtype predicates can be applied to abstract types but not to incomplete types.

Subtype predicates are inherited as expected on derivation. Thus if we have

```
type T is ...
  with Static_Predicate => SP(T);
```

and then

```
type NT is new T
  with Dynamic_Predicate => DP(NT);
```

the result is that both predicates apply to NT rather as if we had written the predicate as SP(NT) **and** DP(NT). So if several apply they are anded together. If any one is dynamic then restrictions on the use of subtypes with a dynamic predicate apply.

There is no need for special predicates for class wide types in the way that we have both Type_Invariant and Type_Invariant'Class. So in the general case where a tagged type is derived from a parent and several progenitors

```
type T is new P and G1 and G2 with ...
```

where P is the parent and G1 and G2 are progenitors, the subtype predicate applicable to T is simply those for P, G1 and G2 all anded together.

6 Default initial values

It is often important that we can rely upon an object having a value within its subtype even before it is assigned to and this especially applies in the face of type invariants and subtype predicates. Consider a type Location whose type invariant In_Place requires the point to be within some place.

```
package Places is
  type Location is private
    with Type_Invariant => In_Place(Location);

  function In_Place(L: Location) return Boolean;
  procedure Do_It(X: in out Location; ... );

private

  type Location is
    record
      X, Y: Float range –1.0 .. +1.0;
    end record;

  ...

end Places;
```

If we just declare an object of type Location thus

```
Somewhere: Location;
```

then there is no guarantee that Somewhere is anywhere in particular. If the type invariant In_Place applies and a subprogram with an **in out** parameter such as Do_It is called

```
Do_It(Somewhere);
```

then it might be that some paths through Do_It do not assign a new value to X. Nevertheless, on return from Do_It, the type invariant In_Place will be checked on the parameter. If Somewhere by chance had an accidental initial value outside the space implied by In_Place then the call will fail. Now it might be that other parameters of the procedure indicate to the caller that Somewhere has not been updated in this case but unfortunately this information is unlikely to be available to the invariant.

One solution to this is to ensure that objects always have an initial value satisfying the requisite constraints, predicates or invariants. One might do this by assigning a safe initial value thus

```
Somewhere: Location := (0.0, 0.0);                    -- illegal
```

but this is illegal because the type is private. We could of course export from the package Places a safe initial value so that we could write

```
Somewhere: Location := Places.Haven;
```

But this is often frowned upon because giving an explicit initial value can hide flow errors. It is thus best to ensure that the object automatically has a safe default value by writing perhaps

```
  type Location is
    record
      X, Y: Float range –1.0 .. +1.0 := 0.0;
    end record;
```

It is curious that Ada allows default initial values for components of records and provides them automatically for access types (**null**) but not for scalar types or for array types. This is remedied in Ada 2012 by the introduction of aspects Default_Value and Default_Component_Value for scalar types and arrays of scalar types respectively. The format is as expected

```
type My_Float is digits 20
  with Default_Value => 0.5;

type OK is new Boolean
  with Default_Value => True;
```

The usual rule regarding the omission of => True does not apply in the case of Default_Value for Boolean types for obvious reasons.

If possible, a special value indicating the status of the default should be supplied. This particularly applies to enumeration types. For example

```
type Switch is (On, Off, Unknown)
  with Default_Value => Unknown;
```

In the case of an array type this can be constrained or unconstrained and the default value will apply to all components.

```
type Vector is array (Integer range <>) of Integer
  with Default_Component_Value => 0;
```

Default initial values cannot be given to the predefined types but they can be given to types derived from them such as the Boolean type OK above.

In the case of a private type, any default has to be given on the full type declaration.

It is important to note that default initial values can only be given for types and not for subtypes. If a default initial value lies outside the range of a subtype then declaring an object of a subtype without its own specific initial value will raise Constraint_Error. So writing

```
subtype Known_Switch is Switch range On .. Off;
A_Switch: Known_Switch;
```

raises Constraint_Error because the default initial value Unknown is outside the range of the subtype Known_Switch.

If a record type is declared and some components are given initial values but others are not then explicitly given initial values take precedence over default values given by these aspects. Thus if we have

```
type Location is
  record
    X: My_Float range –1.0 .. +1.0 := 0.0;
    Y: My_Float range  –1.0 .. +1.0;
  end record;
```

then the component X has default value 0.0 but component Y has default value 0.5, (since My_Float declared above has default value 0.5).

A final important point is that default initial values supplied by these aspects have to be static unlike default initial values for record components.

7 Storage occupancy checks

Finally, two new attributes are introduced to aid in the writing of preconditions. Sometimes it is necessary to check that two objects do not occupy the same storage in whole or in part. This can be done with two functional attributes X'Has_Same_Storage and X'Overlaps_Storage which apply to an object X of any type.

Their specifications are

```
function X'Has_Same_Storage(Arg: any_type) return Boolean;

function X'Overlaps_Storage(Arg: any_type) return Boolean;
```

As an example we might have a procedure Exchange and wish to ensure that the parameters do not overlap in any way. We can write

```
procedure Exchange(X, Y: in out T)
  with Pre => not X'Overlaps_Storage(Y);
```

Attributes are used rather than predefined functions since this enables the semantics to be written in a manner that permits X and Y to be of any type and moreover does not imply that X or Y are read.

The object X and the parameter Y could be components such as A(5) or indeed A(J) or even a slice A(1 .. N). Thus the actual addresses to be checked may not be statically determined but have to be determined at the point of call.

AI-191 shows the following curious example

```
procedure Count(A: in out Arrtype; B: in Arrtype)
  with Pre => not A'Overlaps_Storage(B)
is
  -- intended to count in A the number of value
  -- occurrences in B as part of a distribution sort
begin
  for I in B'Range loop
    A(B(I)) := A(B(I)) + 1;
  end loop;
end Count;
```

The author seems to have assumed that the array A has appropriate components and that they are initialized to zero. This also illustrates the use of an aspect specification in a subprogram body.

At the machine level Overlaps_Storage means that at least one bit is in common and Has_Same_Storage means that all bits are in common. Hence X'Has_Same_Storage(Y) implies X'Overlaps_Storage(Y).

In some applications involving the possibility of aliasing (messing with tree structures comes to mind) we do really want to check that two entities are not in the same place rather than just overlapping in which case it is more logical to use Has_Same_Storage.

References

[1] ISO/IEC JTC1/SC22/WG9 N498 (2009) *Instructions to the Ada Rapporteur Group from SC22/ WG9 for Preparation of Amendment 2 to ISO/IEC 8652*.

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

[3] J. G. P. Barnes (2006) *Programming in Ada 2005*, Addison-Wesley.