John Barnes

Ada Rationale 2012

Introduction

Courtesy of AdaCore

Rationale for Ada 2012: Introduction

John Barnes

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

Abstract

This is the first of a number of papers describing the rationale for Ada 2012. In due course it is anticipated that the papers will be combined (after appropriate reformatting and editing) into a single volume for formal publication.

This first paper covers the background to the development of Ada 2012 and gives a brief overview of the main changes from Ada 2005. Later papers will then look at the changes in more detail.

Keywords: rationale, Ada 2012.

1 Revision process

Ada has evolved over a number of years and, especially for those unfamiliar with the background, it is convenient to summarize the processes involved. The first version was Ada 83 and this was developed by a team led by the late Jean Ichbiah and funded by the USDoD. The development of Ada 95 from Ada 83 was an extensive process also funded by the USDoD. Formal requirements were established after comprehensive surveys of user needs and competitive proposals were then submitted resulting in the selection of Intermetrics as the developer under the leadership of Tucker Taft. Then came Ada 2005 and this was developed on a more modest scale. The work was almost entirely done by voluntary effort with support from within the industry itself through bodies such as the Ada Resource Association and Ada-Europe.

After some experience with Ada 2005 it became clear that some further evolution was appropriate. Adding new features as in Ada 2005 always brings some surprises regarding their use and further polishing is almost inevitable. Accordingly, it was decided that a further revision should be made with a goal of completion in 2012.

As in the case of Ada 2005, the development is being performed under the guidance of ISO/IEC JTC1/SC22 WG9 (hereinafter just called WG9). Previously chaired by Jim Moore, it is now under the chairmanship of Joyce Tokar. This committee has included national representatives of many nations including Belgium, Canada, France, Germany, Italy, Japan, Sweden, Switzerland, the UK and the USA. WG9 developed guidelines [1] for a revision to Ada 2005 which were then used by the Ada Rapporteur Group (the ARG) in drafting the revised standard.

The ARG is a team of experts nominated by the national bodies represented on WG9 and the two liaison organizations, ACM SIGAda and Ada-Europe. In the case of Ada 2005, the ARG was originally led by Erhard Plödereder and then by Pascal Leroy. For Ada 2012, it is led by Ed Schonberg. The editor, who at the end of the day actually writes the words of the standard, continues to be the indefatigable Randy Brukardt.

Suggestions for the revised standard have come from a number of sources such as individuals on the ARG, national bodies on WG9, users and implementers via email discussions on Ada-Comment and so on. Also several issues were left over from the development of Ada 2005.

At the time of writing (August 2011), the revision process is approaching completion. The details of all individual changes are now clear and they are being integrated to form a new version of the Annotated Ada Reference Manual. The final approved standard should emerge towards the end of 2012.

2 Scope of revision

The changes from Ada 95 to Ada 2005 were significant (although not so large as the changes from Ada 83 to Ada 95). The main additions were

- in the OO area, multiple inheritance using interfaces and the ability to make calls using prefixed notation,

- more flexible access types with anonymous types, more control over null and constant, and downward closures via access to subprogram types,
- enhanced structure and visibility control by the introduction of limited with and private with clauses and by an extended form of return statement,
- in the real-time area, the Ravenscar profile [2], various new scheduling polices, timers and execution time budget control,
- some minor improvements to exception handling, numerics (especially fixed point) and some further pragmas such as Assert,
- various extensions to the standard library such as the introduction of operations on vectors and matrices, further operations on times and dates, and operations on wide wide characters; and especially:
- a comprehensive library for the manipulation of containers of various kinds.

The changes from Ada 2005 to Ada 2012 were intended to be relatively modest and largely to lead on from the experience of the additions introduced in Ada 2005. But one thing led to another and in fact the changes are of a similar order to those from Ada 95 to Ada 2005.

From the point of view of the ISO standard, Ada 2005 is the Ada 95 standard modified by two documents. First there was a Corrigendum issued in 2001 [3] and then an Amendment issued in 2005 [4]. In principle the poor user thus has to study these three documents in parallel to understand Ada 2005. However, they were informally incorporated into the Ada 2005 Reference Manual [5].

In the case of Ada 2012, this process of developing a further formal amendment would then lead to the need to consult four documents and so the intention is that the new Edition will formally be a single Revision.

The scope of this Revision is guided by a document issued by WG9 to the ARG in October 2008 [1]. The essence is that the ARG is requested to pay particular attention to

A Improvements that will maintain or improve Ada's advantages, especially in those user domains where safety and criticality are prime concerns. Within this area it cites improving the use and functionality of containers, the ability to write and enforce contracts for Ada entities (for instance, via preconditions) and the capabilities of Ada on multicore and multithreaded architectures.

B Improvements that will remedy shortcomings in Ada. It cites in particular the safety, use, and functionality of access types and dynamic storage management.

So the ARG is asked to improve both OO and real-time with a strong emphasis on real-time and high integrity features. Moreover, "design by contract" features should be added whereas for the previous amendment they were rejected on the grounds that they would not be static.

The ARG is also asked to consider the following factors in selecting features for inclusion:

- Implementability. Can the feature be implemented at reasonable cost?
- Need. Do users actually need it?
- Language stability. Would it appear disturbing to current users?
- Competition and popularity. Does it help to improve the perception of Ada and make it more competitive?
- Interoperability. Does it ease problems of interfacing with other languages and systems?
- Language consistency. Is it syntactically and semantically consistent with the language's current structure and design philosophy?

As before, an important further statement is that "In order to produce a technically superior result, it is permitted to compromise backwards compatibility when the impact on users is judged to be acceptable." In other words don't be paranoid about compatibility.

Finally, there is a warning about secondary standards. Its essence is don't use secondary standards if you can get the material into the RM itself.

The guidelines conclude with the target schedule. This includes WG9 approval of the scope of the amendment in June 2010 which was achieved and submission to ISO/IEC JTC1 in late 2011.

3 Overview of changes

It would be tedious to give a section by section review of the changes as seen by the Reference Manual language lawyer. Instead, the changes will be presented by areas as seen by the user. There can be considered to be six areas:

1 Introduction of dynamic contracts. These can be seen to lead on from the introduction of the Assert pragma in Ada 2005. New syntax (using **with** again) introduces aspect specifications which enable certain properties of entities to be stated where they are declared rather than later using representation clauses. This is put to good use in introducing pre- and postconditions for subprograms and similar assertions for types and subtypes.

2 More flexible expressions. The introduction of preconditions and so on increases the need for more powerful forms of expressions. Accordingly, if expressions, case expressions, quantified expressions and expression functions are all added. A related change is that membership tests are generalized.

3 Structure and visibility control. Functions are now permitted to have **out** and **in out** parameters, and rules are introduced to minimize the risk of inadvertent dependence on order of evaluation of parameters and other entities such as aggregates. More flexibility is permitted with incomplete types and another form of use clause is introduced. There are minor enhancements to extended return statements.

4 Tasking and real-time improvements. Almost all of the changes are in the Real-Time Systems annex. New packages are added for the control of tasks and budgeting on multiprocessor systems, and the monitoring of time spent in interrupts. There are also additional facilities for non-preemptive dispatching, task barriers and suspension objects.

5 Improvements to other general areas. More flexibility is allowed in the position of labels, pragmas, and null statements. A number of corrections are made to the accessibility rules, improvements are made to conversions of access types, and further control over storage pools is added. The composability of equality is now the same for both tagged and untagged record types.

6 Extensions to the standard library. Variants on the existing container packages are introduced to handle bounded containers more efficiently. Additional containers are added for a simple holder, multiway trees and queues. Moreover, a number of general features have been added to make containers and other such reusable libraries easier to use. Minor additions cover directories, locale capabilities, string encoding and further operations on wide and wide wide characters.

The reader might feel that the changes are quite extensive but each has an important role to play in making Ada more useful. Indeed the solution of one problem often leads to auxiliary requirements. The desire to introduce stronger description of contracts led to the search for good syntax which led to aspect specifications. And these strengthened the need for more flexible forms of expressions and so on. Other changes were driven by outside considerations such as the multiprocessors and others stem from what now seem to be obvious but minor flaws in Ada 2005.

A number of other changes were rejected as really unnecessary. For example, the author was at one time enthused by a desire for fixed point cyclic types. But it proved foolish without base 60 hardware to match our inheritance of arithmetic in a Babylonian style for angles.

Before looking at the six areas in a little more detail it is perhaps worth saying a few words about compatibility with Ada 2005. The guidelines gave the ARG freedom to be sensible in this area. Of course, the worst incompatibilities are those where a valid program in Ada 2005 continues to be valid in Ada 2012 but does something different. It is believed that incompatibilities of this nature will be most unlikely to arise in practice.

However, incompatibilities whereby a valid Ada 2005 program fails to compile in Ada 2012 are tolerable provided they are infrequent. A few such incompatibilities are possible. The most obvious cause is the introduction of one more reserved word, namely **some**, which is used in quantified expressions to match **all**. Thus if an existing Ada 2005 program uses **some** as an identifier then it will need modification. Once again, the introduction of a new category of unreserved keywords was considered but was eventually rejected as confusing.

3.1 Contracts

One of the important issues highlighted by WG9 for the Amendment was the introduction of material for enforcing contracts such as preconditions and postconditions. As a simple example consider a stack with procedures Push and Pop. An obvious precondition for Pop is that the stack must not be empty. If we have a function Is_Empty for testing the state of the stack then a call of Is_Empty would provide the basis for an appropriate precondition.

The question now is to find a good way to associate the expression **not** Is_Empty with the specification of the procedure Pop. Note that it is the specification that matters since it is the specification that provides the essence of the contract between the caller of the procedure Pop and the writer of its body. The contract provided by a traditional Ada subprogram specification is rather weak – essentially it just provides enough information for the compiler to generate the correct code for the calls but says nothing about the semantic behaviour of the associated body.

The traditional way to add information of this kind in Ada is via a pragma or by giving some kind of aspect clause. However, there were problems with this approach. One is that there is no convenient way to distinguish between several overloaded subprograms and another is that such information is given later on because of interactions with freezing and linear elaboration rules.

Accordingly, it was decided that a radical new approach should be devised and this led to the introduction of aspect specifications which are given with the item to which they relate using the reserved word **with**.

In the case of preconditions and postconditions, Ada 2012 introduces aspects Pre and Post. So to give the precondition for Pop we augment the specification of Pop by writing

```
procedure Pop(S: in out Stack; X: out Item)
  with Pre => not Is_Empty(S);
```

In a similar way we might give a postcondition as well which might be that the stack is not full. So altogether the specification of a generic package for stacks might be

```
generic
  type Item is private;
package Stacks is
  type Stack is private;

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

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

private
    ...
end Stacks;
```

Note how the individual aspects Pre and Post take the form of

```
aspect_mark => expression
```

and that if there are several then they are separated by commas. The final semicolon is of course the semicolon at the end of the subprogram declaration as a whole. Thus the overall syntax is now

```
subprogram_declaration ::=
  [overriding_indicator]
  subprogram_specification
  [aspect_specification] ;
```

and in general

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

Pre- and postconditions are controlled by the same mechanism as assertions using the pragma Assert. It will be recalled that these can be switched on and off by the pragma Assertion_Policy. Thus if we write

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

then assertions are enabled whereas if the parameter of the pragma is Ignore then all assertions are ignored.

In the case of a precondition, whenever a subprogram with a precondition is called, if the policy is Check then the precondition is evaluated and if it is False then Assertion_Error is raised and the subprogram is not entered. Similarly, on return from a subprogram with a postcondition, if the policy is Check then the postcondition is evaluated and if it is False then Assertion_Error is raised.

So if the policy is Check and Pop is called when the stack is empty then Assertion_Error is raised whereas if the policy is Ignore then the predefined exception Constraint_Error would probably be raised (depending upon how the stack had been implemented).

Note that, unlike the pragma Assert, it is not possible to associate a specific message with the raising of Assertion_Error by a pre- or postcondition. The main reason is that it might be confusing with multiple conditions (which can arise with inheritance) and in any event it is expected that the implementation will give adequate information about which condition has been violated.

Note that it is not permitted to give the aspects Pre or Post for a null procedure; this is because all null procedures are meant to be interchangeable.

There are also aspects Pre'Class and Post'Class for use with tagged types (and they can be given with null procedures). The subtle topic of multiple inheritance of pre- and postconditions will be discussed in detail in a later paper.

Two new attributes are useful in postconditions. X'Old denotes the value of X on entry to the subprogram whereas X denotes the value on exit. And in the case of a function F, the value returned by the function can be denoted by F'Result in a postcondition for F.

As a general rule, the new aspect specifications can be used instead of aspect clauses and pragmas for giving information regarding entities such as types and subprograms.

For example rather than

```
type Bit_Vector is array (0 .. 15) of Boolean;
```

followed later by

```
for Bit_Vector'Component_Size use 1;
```

we can more conveniently write

```
type Bit_Vector is array (0 .. 15) of Boolean
  with Component_Size => 1;
```

However, certain aspects such as record representation and enumeration representations cannot be given in this way because of the special syntax involved.

In cases where aspect specifications can now be used, the existing pragmas are mostly considered obsolescent and condemned to Annex J.

It should be noted that pragmas are still preferred for stating properties of program units such as Pure, Preelaborable and so on. However, we now talk about the pure property as being an aspect of a package. It is a general rule that the new aspect specifications are preferred with types and subprograms but pragmas continue to be preferred for program units. Nevertheless, the enthusiast for the new notation could write

```
package Ada_Twin
  with Pure is
end Ada_Twin;
```

which illustrates that in some cases no value is required for the aspect (by default it is True).

A notable curiosity is that Preelaborable_Initialization still has to be specified by a pragma (this is because of problems with different views of the type).

Note incidentally that to avoid confusion with some other uses of the reserved word **with**, in the case of aspect specifications **with** is at the beginning of the line.

There are two other new facilities of a contractual nature concerning types and subtypes. One is known as type invariants and these describe properties of a type that remain true and can be relied upon. The other is known as subtype predicates which extend the idea of constraints. The distinction can be confusing at first sight and the following extract from one of the Ada Issues might be helpful.

"Note that type invariants are not the same thing as constraints, as invariants apply to all values of a type, while constraints are generally used to identify a subset of the values of a type. Invariants are only meaningful on private types, where there is a clear boundary (the enclosing package) that separates where the invariant applies (outside) and where it need not be satisfied (inside). In some ways, an invariant is more like the range of values specified when declaring a new integer type, as opposed to the constraint specified when defining an integer subtype. The specified range of an integer type can be violated (to some degree) in the middle of an arithmetic computation, but must be satisfied by the time the value is stored back into an object of the type."

Type invariants are useful if we want to ensure that some relationship between the components of a private type always holds. Thus suppose we have a stack and wish to ensure that no value is placed on the stack equal to an existing value on the stack. We can modify the earlier example to

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

   function Is_Empty(S: Stack) return Boolean;
   function Is_Full(S: Stack) return Boolean;
   function Is_Unduplicated(S: Stack) return Boolean;

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

   -- and so on
```

The function Is_Unduplicated then has to be written (in the package body as usual) to check that all values of the stack are different.

Note that we have mentioned Is_Unduplicated in the type invariant before its specification. This violates the usual "linear order of elaboration". However, there is a general rule that all aspect specifications are only elaborated when the entity they refer to is frozen. Recall that one of the reasons for the introduction of aspect specifications was to overcome this problem with the existing mechanisms which caused information to become separated from the entities to which it relates.

The invariant on a private type T is checked when the value can be changed from the point of view of the outside user. That is primarily

- after default initialization of an object of type T,
- after a conversion to type T,
- after a call that returns a result of a type T or has an **out** or **in out** or access parameter of type T.

The checks also apply to subprograms with parameters or results whose components are of the type T.

In the case of the stack, the invariant Is_Unduplicated will be checked when we declare a new object of type Stack and each time we call Push and Pop.

Note that any subprograms internal to the package and not visible to the user can do what they like. It is only when a value of the type Stack emerges into the outside world that the invariant is checked.

The type invariant could be given on the full type in the private part rather than on the visible declaration of the private type (but not on both). Thus the user need not know that an invariant applies to the type.

Type invariants, like pre- and postconditions, are controlled by the pragma Assertion_Policy and only checked if the policy is Check. If an invariant fails to be true then Assertion_Error is raised at the appropriate point.

There is also an aspect Type_Invariant'Class for use with tagged types.

The subtype feature of Ada is very valuable and enables the early detection of errors that linger in many programs in other languages and cause disaster later. However, although valuable, the subtype mechanism is somewhat limited. We can only specify a contiguous range of values in the case of integer and enumeration types.

Accordingly, Ada 2012 introduces subtype predicates as an aspect that can be applied to type and subtype declarations. The requirements proved awkward to satisfy with a single feature so in fact there are two aspects: Static_Predicate and Dynamic_Predicate. They both take a Boolean expression and the key difference is that the static predicate is restricted to certain types of expressions so that it can be used in more contexts.

Suppose we are concerned with seasons and that we have a type Month thus

```
type Month is (Jan, Feb, Mar, Apr, May, ..., Nov, Dec);
```

Now suppose we wish to declare subtypes for the seasons. For most people winter is December, January, February. (From the point of view of solstices and equinoxes, winter is from December 21 until March 21 or thereabouts, but March seems to me generally more like spring rather than winter and December feels more like winter than autumn.) So we would like to declare a subtype embracing Dec, Jan and Feb. We cannot do this with a constraint but we can use a static predicate by writing

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

and then we are assured that objects of subtype Winter can only be Dec, Jan or Feb (provided once more that the Assertion_Policy pragma has set the Policy to Check). Note the use of the subtype name (Winter) in the expression where it stands for the current instance of the subtype.

The aspect is checked whenever an object is default initialized, on assignments, on conversions, on parameter passing and so on. If a check fails then Assertion_Error is raised.

The observant reader will note also that the membership test takes a more flexible form in Ada 2012 as explained in the next section.

If we want the expression to be dynamic then we have to use Dynamic_Predicate thus

```
type T is ... ;
function Is_Good(X: T) return Boolean;
subtype Good_T is T
  with Dynamic_Predicate => Is_Good(Good_T);
```

Note that a subtype with predicates cannot be used in some contexts such as index constraints. This is to avoid having arrays with holes and similar nasty things. However, static predicates are allowed in a for loop meaning to try every value. So we could write

```
for M in Winter loop...
```

Beware that the loop uses values for M in the order, Jan, Feb, Dec and not Dec, Jan, Feb as the user might have wanted.

As another example, suppose we wish to specify that an integer is even. We might expect to be able to write

```
subtype Even is Integer
  with Static_Predicate => Even mod 2 = 0;          -- illegal
```

Sadly, this is illegal because the expression in a static predicate is restricted and cannot use some operations such as **mod**. We have to use a dynamic predicate thus

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

and then we cannot write

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

but have to spell it out in detail such as

```
for X in Integer loop
  if X mod 2 = 0 then                      -- or if X in Even then
    ... -- body of loop
  end if;
end loop;
```

The assurance given by type invariants and subtype predicates can depend upon the object having a sensible initial value. There is a school of thought that giving default initial values (such as zero) is bad since it can obscure flow errors. However, it is strange that Ada does allow default initial values to be given for components of records but not for scalar types or array types. This is rectified in Ada 2012 by aspects Default_Value and Default_Component_Value. We can write

```
type Signal is (Red, Amber, Green)
  with Default_Value => Red;

type Text is new String
  with Default_Component_Value => Ada.Characters.Latin_1.Space;

type Day is range 1 .. 31
  with Default_Value => 1;
```

Note that, unlike default initial values for record components, these have to be static.

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 attributes thus

```
X'Has_Same_Storage(Y)
X'Overlaps_Storage(Y)
```

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.

3.2 Expressions

Those whose first language was Algol 60 or Algol 68 or who have had the misfortune to dabble in horrid languages such as C will have been surprised that a language of the richness of Ada does not have conditional expressions. Well, the good news is that Ada 2012 has at last introduced conditional expressions which take two forms, if expressions and case expressions.

The reason that Ada did not originally have conditional expressions is probably that there was a strong desire to avoid any confusion between statements and expressions. We know that many errors in C arise because assignments can be used as expressions. But the real problem with C is that it also treats Booleans as integers, and confuses equality and assignment. It is this combination of fluid styles that causes problems. But just introducing conditional expressions does not of itself introduce difficulties if the syntax is clear and unambiguous.

If expressions in Ada 2012 take the form as shown by the following statements:

```
S := (if N > 0 then +1 else 0);

Put(if N = 0 then "none" elsif N = 1 then "one" else "lots");
```

Note that there is no need for **end if** and indeed it is not permitted. Remember that **end if** is vital for good structuring of if statements because there can be more than one statement in each branch. This does not arise with if expressions so **end if** is unnecessary and moreover would be heavy as a closing bracket. However, there is a rule that an if expression must always be enclosed in parentheses. Thus we cannot write

```
X := if L > 0 then M else N + 1;            -- illegal
```

because there would be confusion between

```
X := (if L > 0 then M else N) + 1;          -- and

X := (if L > 0 then M else (N + 1));
```

The parentheses around N+1 are not necessary in the last line above but added to clarify the point.

However, if the context already provides parentheses then additional ones are unnecessary. Thus an if expression as a single parameter does not need double parentheses.

It is clear that if expressions will have many uses. However, the impetus for providing them in Ada 2012 was stimulated by the introduction of aspects of the form

```
Pre => expression
```

There will be many occasions when preconditions have a conditional form and without if expressions these would have to be wrapped in a function which would be both heavy and obscure. For example suppose a procedure P has two parameters P1 and P2 and that the precondition is that if P1 is positive then P2 must also be positive but if P1 is not positive then there is no restriction on P2. We could express this by writing a function such as

```
function Checkparas(P1, P2: Integer) return Boolean is
begin
  if P1 > 0 then
    return P2 > 0;
  else                    -- P1 is not positive
    return True;          -- so don't care about P2
  end if;
end Checkparas;
```

and then we can write

```
procedure P(P1, P2: Integer)
  with Pre => Checkparas(P1, P2);
```

This is truly gruesome. Apart from the effort of having to declare the wretched function Checkparas, the consequence is that the meaning of the precondition can only be determined by looking at the body of Checkparas and that could be miles away, typically in the body of the package containing the declaration of P. This would be a terrible violation of information hiding in reverse; we would be forced to hide something that should be visible.

However, using if expressions we can simply write

```
Pre => (if P1 > 0 then P2 > 0 else True);
```

and this can be abbreviated to

```
Pre => (if P1 > 0 then P2 > 0);
```

because there is a convenient rule that a trailing **else** True can be omitted when the type is a Boolean type. Many will find it much easier to read without **else** True anyway since it is similar to saying P1 > 0 implies P2 > 0. Adding an operation such as implies was considered but rejected as unnecessary.

The precondition could be extended to say that if P1 equals zero then P2 also has to be zero but if P1 is negative then we continue not to care about P2. This would be written thus

```
Pre => (if P1 > 0 then P2 > 0 elsif P1 = 0 then P2 = 0);
```

There are various sensible rules about the types of the various branches in an if expression as expected. Basically, they must all be of the same type or convertible to the same expected type. Thus consider a procedure Do_It taking a parameter of type Float and the call

```
Do_It (if B then X else 3.14);
```

where X is a variable of type Float. Clearly we wish to permit this but the two branches of the if statement are of different types, X is of type Float whereas 3.14 is of type *universal_real*. But a value of type *universal_real* can be implicitly converted to Float which is the type expected by Do_It and so all is well.

There are also rules about accessibility in the case where the various branches are of access types; the details need not concern us in this overview!

The other new form of conditional expression is the case expression and this follows similar rules to the if expression just discussed. Here is an amusing example based on one in the AI which introduces case expressions.

Suppose we are making a fruit salad and add various fruits to a bowl. We need to check that the fruit is in an appropriate state before being added to the bowl. Suppose we have just three fruits given by

```
type Fruit_Kind is (Apple, Banana, Pineapple);
```

then we might have a procedure Add_To_Salad thus

```
procedure Add_To_Salad(Fruit: in Fruit_Type);
```

where Fruit_Type is perhaps a discriminated type thus

```
type Fruit_Type (Kind: Fruit_Kind) is private;
```

In addition there might be functions such as Is_Peeled that interrogate the state of a fruit.

We could then have a precondition that checks that the fruit is in an edible state thus

```
Pre => (if Fruit.Kind = Apple then Is_Crisp(Fruit)
           elsif Fruit.Kind = Banana then Is_Peeled(Fruit)
           elsif Fruit.Kind = Pineapple then Is_Cored(Fruit));
```

(This example is all very well but it has allowed the apple to go in uncored and the pineapple still has its prickly skin.)

Now suppose we decide to add Orange to type Fruit_Kind. The precondition will still work in the sense that the implicit **else** True will allow the orange to pass the precondition unchecked and will go into the fruit salad possibly unpeeled, unripe or mouldy. The trouble is that we have lost the full coverage check which is such a valuable feature of case statements and aggregates in Ada.

We overcome this by using a case expression and writing

```
Pre => (case Fruit.Kind is
                    when Apple => Is_Crisp(Fruit),
                    when Banana => Is_Peeled(Fruit),
                    when Pineapple => Is_Cored(Fruit),
                    when Orange => Is_Peeled(Fruit));
```

and of course without the addition of the choice for Orange it would fail to compile.

Note that there is no **end case** just as there is no **end if** in an if expression. Moreover, like the if expression, the case expression must be in parentheses. Similar rules apply regarding the types of the various branches and so on.

Of course, the usual rules of case statements apply and so we might decide not to bother about checking the crispness of the apple but to check alongside the pineapple (another kind of apple!) that it has been cored by writing

```
Pre => (case Fruit.Kind is
                    when Apple | Pineapple => Is_Cored(Fruit),
                    when Banana | Orange => Is_Peeled(Fruit));
```

We can use **others** as the last choice as expected but this would lose the value of coverage checking. There is no default **when others** => True corresponding to **else** True for if expressions because that would defeat coverage checking completely.

A further new form of expression is the so-called quantified expression. Quantified expressions allow the checking of a boolean expression for a given range of values and will again be found useful in pre- and postconditions. There are two forms using **for all** and **for some**. Note carefully that **some** is a new reserved word.

Suppose we have an integer array type

```
type Atype is array (Integer range <>) of Integer;
```

then we might have a procedure that sets each element of an array of integers equal to its index. Its specification might include a postcondition thus

```
procedure Set_Array(A: out Atype)
  with Post => (for all M in A'Range => A(M) = M);
```

This is saying that for all values of M in A'Range we want the expression A(M) = M to be true. Note how the two parts are separated by =>.

We could devise a function to check that some component of the array has a given value by

```
function Value_Present(A: Atype; X: Integer) return Boolean
  with Post => Value_Present'Result = (for some M in A'Range => A(M) = X);
```

Note the use of Value_Present'Result to denote the result returned by the function Value_Present.

As with conditional expressions, quantified expressions are always enclosed in parentheses.

The evaluation of quantified expressions is as expected. Each value of M is taken in turn (as in a for statement and indeed we could insert **reverse**) and the expression to the right of => then evaluated. In the case of universal quantification (a posh term meaning **for all**) as soon as one value is found to be False then the whole quantified expression is False and no further values are checked; if all values turn out to be True then the quantified expression is True. A similar process applies to existential quantification (that is **for some**) where the roles of True and False are reversed.

Those with a mathematical background will be familiar with the symbols ∀ and ∃ which correspond to **for all** and **for some** respectively. Readers are invited to discuss whether the A is upside down and the E backwards or whether they are both simply rotated.

As a somewhat more elaborate example suppose we have a function that finds the index of the first value of M such that A(M) equals a given value X. This needs a precondition to assert that such a value exists.

```
function Find(A: Atype; X: Integer) return Integer
  with
    Pre => (for some M in A'Range => A(M) = X),
    Post => A(Find'Result) = X and
        (for all M in A'First .. Find'Result–1 => A(M) /= X);
```

Note again the use of Find'Result to denote the result returned by the function Find.

Quantified expressions can be used in any context requiring an expression and are not just for pre- and postconditions. Thus we might test whether an integer N is prime by

```
RN := Integer(Sqrt(Float(N)));
if (for some K in 2 .. RN => N mod K = 0) then
  ...          -- N not prime
```

or we might reverse the test by

```
if (for all K in 2 .. RN => N mod K / = 0) then
  ...          -- N is prime
```

Beware that this is not a recommended technique if N is at all large!

We noted above that a major reason for introducing if expressions and case expressions was to avoid the need to introduce lots of auxiliary functions for contexts such as preconditions. Nevertheless the need still arises from time to time. A feature of existing functions is that the code is in the body and this is not visible in the region of the precondition – information hiding is usually a good thing but here it is a problem. What we need is a localized and visible shorthand for a little function. After much debate, Ada 2012 introduces expression functions which are essentially functions whose visible body comprises a single expression. Thus suppose we have a record type such as

```
type Point is tagged
  record
    X, Y: Float := 0.0;
  end record;
```

and the precondition we want for several subprograms is that a point is not at the origin. Then we could write

```
function Is_At_Origin(P: Point) return Boolean is
  (P.X = 0.0 and P.Y = 0.0);
```

and then

```
procedure Whatever(P: Point; ... )
  with Pre => not P.Is_At_Origin;
```

and so on.

Such a function is known as an expression function; naturally it does not have a distinct body. The expression could be any expression and could include calls of other functions (and not just expression functions). The parameters could be of any mode (see next section).

Expression functions can also be used as a completion. This arises typically if the type is private. In that case we cannot access the components P.X and P.Y in the visible part. However, we don't want to have to put the code in the package body. So we declare a function specification in the visible part in the normal way thus

```
function Is_At_Origin(P: Point) return Boolean;
```

and then an expression function in the private part thus

```
private
  type Point is ...

  function Is_At_Origin(P: Point) return Boolean is
    (P.X = 0.0 and P.Y = 0.0);
```

and the expression function then completes the declaration of Is_At_Origin and no function body is required in the package body.

Observe that we could also use an expression function for a completion in a package body so that rather than writing the body as

```
function Is_At_Origin(P: Point) return Boolean is
begin
  return P.X = 0.0 and P.Y = 0.0;
end Is_At_Origin;
```

we could write an expression function as a sort of shorthand.

Incidentally, in Ada 2012, we can abbreviate a null procedure body in a similar way by writing

```
procedure Nothing(...) is null;
```

as a shorthand for

```
procedure Nothing(...) is
begin
  null;
end Nothing;
```

and this will complete the procedure specification

```
procedure Nothing(...);
```

Another change in this area is that membership tests are now generalized. In previous versions of Ada, membership tests allowed one to see whether a value was in a range or in a subtype, thus we could write either of

```
if D in 1 .. 30 then

if D in Days_In_Month then
```

but we could not write something like

```
if D in 1 | 3 | 5 | 6 ..10 then
```

This is now rectified and following **in** we can now have one or more of a value, a range, or a subtype or any combination separated by vertical bars. Moreover, they do not have to be static.

A final minor change is that the form qualified expression is now treated as a name rather than as a primary. Remember that a function call is treated as a name and this allows a function call to be used as a prefix. For example suppose F returns an array (or more likely an access to an array) then we can write

```
F(...)(N)
```

and this returns the value of the component with index N. However, suppose the function is overloaded so that this is ambiguous. The normal technique to overcome ambiguity is to use a qualified expression and write T'(F(...)). But in Ada 2005 this is not a name and so cannot be used as a prefix. This means that we typically have to copy the array (or access) and then do the indexing or (really ugly) introduce a dummy type conversion and write T(T'(F(...)))(N). Either way, this is a nuisance and hence the change in Ada 2012.

3.3 Structure and visibility

What will seem to many to be one of the most dramatic changes in Ada 2012 concerns functions. In previous versions of Ada, functions could only have parameters of mode **in**. Ada 2012 permits functions to have parameters of all modes.

There are various purposes of functions. The purest is simply as a means of looking at some state. Examples are the function Is_Empty applying to an object of type Stack. It doesn't change the state of the stack but just reports on some aspect of it. Other pure functions are mathematical ones such as Sqrt. For a given parameter, Sqrt always returns the same value. These functions never have any side effects. At the opposite extreme we could have a function that has no restrictions at all; any mode of parameters permitted, any side effects permitted, just like a general procedure in fact but also with the ability to return some result that can be immediately used in an expression.

An early version of Ada had such features, there were pure functions on the one hand and so-called value-returning procedures on the other. However, there was a desire for simplification and so we ended up with Ada 83 functions.

In a sense this was the worst of all possible worlds. A function can perform any side effects at all, provided they are not made visible to the user by appearing as parameters of mode **in out**! As a consequence, various tricks have been resorted to such as using access types (either directly or indirectly). A good example is the function Random in the Numerics annex. It has a parameter Generator of mode **in** but this does in fact get updated indirectly whenever Random is called. So parameters can change even if they are of mode **in**. Moreover, the situation has encouraged programmers to use access parameters unnecessarily with increased runtime cost and mental obscurity.

Ada 2012 has bitten the bullet and now allows parameters of functions to be of any mode. But note that operators are still restricted to only **in** parameters for obvious reasons.

However, there are risks with functions with side effects whether they are visible or not. This is because Ada does not specify the order in which parameters are evaluated nor the order in which parts of an expression are evaluated. So if we write

```
X := Random(G) + Random(G);
```

we have no idea which call of Random occurs first – not that it matters in this case. Allowing parameters of all modes provides further opportunities for programmers to inadvertently introduce order dependence into their programs.

So, in order to mitigate the problems of order dependence, Ada 2012 has a number of rules to catch the more obvious cases. These rules are all static and are mostly about aliasing. For example, it is illegal to pass the same actual parameter to two formal **in out** parameters – the rules apply to both functions and procedures. Consider

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

```
   Triple := Triple * 3;
end Do_It;
```

Now if we write

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

then Var might become 4 or 6 in Ada 2005 according to the order in which the parameters are copied back.

These rules also apply to any context in which the order is not specified and which involves function calls with **out** or **in out** parameters. Thus an aggregate such as

```
(Var, F(Var))
```

where F has an **in out** parameter is illegal since the order of evaluation of the expressions in an aggregate is undefined and so the value of the first component of the aggregate will depend upon whether it is evaluated before or after F is called.

Full details of the rules need not concern the normal programmer – the compiler will tell you off!

Another change concerning parameters is that it is possible in Ada 2012 to explicitly state that a parameter is to be aliased. Thus we can write

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

An aliased parameter is always passed by reference and the accessibility rules are modified accordingly. This facility is used in a revision to the containers which avoids the need for expensive and unnecessary copying of complete elements when they are updated. The details will be given in a later paper.

A major advance in Ada 2005 was the introduction of limited with clauses giving more flexibility to incomplete types. However, experience has revealed a few minor shortcomings.

One problem is that an incomplete type in Ada 2005 cannot be completed by a private type. This prevents the following mutually recursive structure of two types having each other as an access discriminant

```
type T1;
type T2 (X: access T1) is private;
type T1 (X: access T2) is private;          -- OK in Ada 2012
```

The rules in Ada 2012 are changed so that an incomplete type can be completed by any type, including a private type (but not another incomplete type obviously).

Another change concerns the use of incomplete types as parameters. Generally, we do not know whether a parameter of a private type is passed by copy or by reference. The one exception is that if it is tagged then we know it will be passed by reference. As a consequence there is a rule in Ada 2005 that an incomplete type cannot be used as a parameter unless it is tagged incomplete. This has forced the unnecessary use of access parameters.

In Ada 2012, this problem is remedied by permitting incomplete types to be used as parameters (and as function results) provided that they are fully defined at the point of call and where the body is declared.

A final change to incomplete types is that a new category of formal generic parameter is added that allows a generic unit to be instantiated with an incomplete type. Thus rather than having to write a signature package as

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

which must be instantiated with complete types, we can now write

```
generic
  type Element;
  type Set;
  with function Empty return Set;
  ...
package Set_Signature is end;
```

where the formal parameters Element and Set are categorized as incomplete. Instantiation can now be performed using any type, including incomplete or private types as actual parameters. This permits the cascading of generic packages which was elusive in Ada 2005 and will be explained in detail in a later paper. Note that we can also write **type** Set **is tagged**; which requires the actual parameter to be tagged but still permits it to be incomplete.

There is a change regarding discriminants. In Ada 2005, a discriminant can only have a default value if the type is not tagged. Remember that giving a default value makes a type mutable. But not permitting a default value has proved to be an irritating restriction in the case of limited tagged types. Being limited they cannot be changed anyway and so a default value is not a problem and is permitted in Ada 2012. This feature is used in the declaration of the protected types for synchronized queues in Section 3.6.

Another small but useful improvement is in the area of use clauses. In Ada 83, use clauses only apply to packages and everything in the package specification is made visible. Programming guidelines often prohibit use clauses on the grounds that programs are hard to understand since the origin of entities is obscured. This was a nuisance with operators because it prevented the use of infixed notation and forced the writing of things such as

```
P."+"(X, Y)
```

Accordingly, Ada 95 introduced the use type clause which just makes the operators for a specific type in a package directly visible. Thus we write

```
use type P.T;
```

However, although this makes the primitive operators of T visible it does not make everything relating to T visible. Thus it does not make enumeration literals visible or other primitive operations of the type such as subprograms. This is a big nuisance.

To overcome this, Ada 2012 introduces a further variation on the use type clause. If we write

```
use all type P.T;
```

then *all* primitive operations of T are made visible (and not just primitive operators) and this includes enumeration literals in the case of an enumeration type and class wide operations of tagged types.

Finally, there are a couple of small changes to extended return statements which are really corrections to amend oversights in Ada 2005.

The first is that a return object can be declared as **constant**. For example

```
function F(...) return LT is
...
  return Result: constant LT := ... do
    ....
  end return;
end F;
```

We allow everything else to be declared as **constant** so we should here as well especially if LT is a limited type. This was really an oversight in the syntax.

The other change concerns class wide types. If the returned type is class wide then the object declared in the extended return statement need not be the same in Ada 2012 provided it can be converted to the class wide type.

Thus

```
function F(...) return T'Class is
...
  return X: TT do
  ...
  end return;
end F;
```

is legal in Ada 2012 provided that TT is descended from T and thus covered by T'Class. In Ada 2005 it is required that the result type be identical to the return type and this is a nuisance with a class wide type because it then has to be initialized with something and so on. Note the analogy with constraints. The return type might be unconstrained such as String whereas the result (sub)type can be constrained such as String(1 .. 5).

3.4 Tasking and real-time facilities

There are a number of improvements regarding scheduling and dispatching in the Real-Time Systems annex.

A small addition concerns non-preemptive dispatching. In Ada 2005, a task wishing to indicate that it is willing to be preempted has to execute

```
delay 0.0;
```

(or **delay until** Ada.Real_Time.Time_First in Ravenscar). This is ugly and so a procedure Yield is added to the package Ada.Dispatching.

A further addition is the ability to indicate that a task is willing to be preempted by a task of higher priority (but not the same priority). This is done by calling Yield_To_Higher which is declared in a new child package with specification

```
package Ada.Dispatching.Non_Preemptive is
  pragma Preelaborate(Non_Preemptive);
  procedure Yield_To_Higher;
  procedure Yield_To_Same_Or_Higher renames Yield;
end Ada.Dispatching.Non_Preemptive;
```

Another low-level scheduling capability concerns suspension objects; these were introduced in Ada 95. Recall that we can declare an object of type Suspension_Object and call procedures to set it True or False. By calling Suspend_Until_True a task can suspend itself until the state of the object is true.

Ada 2005 introduced Earliest Deadline First (EDF) scheduling. The key feature here is that tasks are scheduled according to deadlines and not by priorities. A new facility introduced in Ada 2012 is the ability to suspend until a suspension object is true and then set its deadline sometime in the future. This is done by calling the aptly named procedure Suspend_Until_True_And_Set_Deadline in a new child package Ada.Synchronous_Task_Control.EDF.

A new scheduling feature is the introduction of synchronous barriers in a new child package Ada.Synchronous_Barriers. The main features are a type Synchronous_Barrier with a discriminant giving the number of tasks to be waited for.

```
type Synchronous_Barrier(Release_Threshold: Barrier_Limit) is limited private;
```

There is also a procedure

```
procedure Wait_For_Release(The_Barrier: in out Synchronous_Barrier;
                                                Notified: out Boolean);
```

When a task calls Wait_For_Release it gets suspended until the number waiting equals the discriminant. All the tasks are then released and just one of them is told about it by the parameter Notified being True. The general idea is that this one task then does something on behalf of all the others. The count of tasks waiting is then reset to zero so that the synchronous barrier can be used again.

A number of other changes in this area are about the use of multiprocessors and again concern the Real-Time Systems annex.

A new package System.Multiprocessors is introduced as follows

```
package System.Multiprocessors is
  type CPU_Range is range 0..implementation-defined;
  Not_A_Specific_CPU: constant CPU_Range := 0:
  subtype CPU is CPU_Range range 1 .. CPU_Range'Last;
  function Number_Of_CPUs return CPU;
end System.Multiprocessors;
```

A value of subtype CPU denotes a specific processor. Zero indicates don't know or perhaps don't care. The total number of CPUs is determined by calling the function Number_Of_CPUs. This is a function rather than a constant because there could be several partitions with a different number of CPUs on each partition.

Tasks can be allocated to processors by an aspect specification. If we write

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

then My_Task will be executed by processor number 10. In the case of a task type then all tasks of that type will be executed by the given processor. The expression giving the processor for a task can be dynamic. The aspect can also be set by a corresponding pragma CPU. (This is an example of a pragma born obsolescent.) The aspect CPU can also be given to the main subprogram in which case the expression must be static.

Further facilities are provided by the child package System.Multiprocessors.Dispatching_Domains. The idea is that processors are grouped together into dispatching domains. A task may then be allocated to a domain and it will be executed on one of the processors of that domain.

Domains are of a type Dispatching_Domain. They are created by a function Create

```
function Create(First, Last: CPU) return Dispatching_Domain;
```

that takes the first and last numbered CPU of the domain and then returns the domain. All CPUs are initially in the System_Dispatching_Domain. If we attempt to do something silly such as create overlapping domains, then Dispatching_Domain_Error is raised.

Tasks can be assigned to a domain in two ways. One way is to use an aspect

```
task My_Task
  with Dispatching_Domain => My_Domain;
```

The other way is by calling the procedure Assign_Task whose specification is

```
procedure Assign_Task(Domain: in out Dispatching_Domain;
                                            CPU: in CPU_Range := Not_A_Specific_CPU;
                                            T: in Task_Id := Current_Task);
```

There are a number of other subprograms for manipulating domains and CPUs. An interesting one is Delay_Until_And_Set_CPU which delays the calling task until a given real time and then sets the processor.

The Ravenscar profile is now defined to be permissible with multiprocessors. However, there is a restriction that tasks may not change CPU. Accordingly the definition of the profile now includes the following restriction

```
No_Dependence => System.Multiprocessors.Dispatching_Domains
```

In order to clarify the use of multiprocessors with group budgets the package Ada.Execution_Time.Group_Budgets introduced in Ada 2005 is slightly modified. The type Group_Budget (which is currently just **tagged limited private**) has a discriminant in Ada 2012 giving the CPU thus

```
type Group_Budget(CPU: System.Multiprocessors.CPU :=
                                             System.Multiprocessors.CPU'First) is tagged limited
private;
```

This means that a group budget only applies to a single processor. If a task in a group is executed on another processor then the budget is not consumed. Note that the default value for CPU is CPU'First which is always 1.

Another improvement relating to times and budgets concerns interrupts. Two Boolean constants are added to the package Ada.Execution_Time

```
Interrupt_Clocks_Supported: constant Boolean := implementation-defined;
Separate_Interrupt_Clocks_Supported: constant Boolean := implementation-defined;
```

The constant Interrupt_Clocks_Supported indicates whether the time spent in interrupts is accounted for separately from the tasks and then Separate_Interrupt_Clocks_Supported indicates whether it is accounted for each interrupt individually. There is also a function

```
function Clocks_For_Interrupts return CPU_Time;
```

This function gives the time used over all interrupts. Calling it if Interrupt_Clocks_Supported is false raises Program_Error.

A new child package accounts for the interrupts separately if Separate_Interrupt_Clocks_Supported is true.

```
package Ada.Execution_Time.Interrupts is
  function Clock(Interrupt: Ada.Interrupts.Interrupt_Id) return CPU_Time;
  function Supported(Interrupt: Ada.Interrupts.Interrupt_Id) return Boolean;
end Ada.Execution_Time.Interrupts;
```

The function Supported indicates whether the time for a particular interrupt is being monitored. If it is then Clock returns the accumulated time spent in that interrupt handler (otherwise it returns zero). However, if the overall constant Separate_Interrupt_Clocks_Supported is false then calling Clock for a particular interrupt raises Program_Error.

Multiprocessors have an impact on shared variables. The existing pragma Volatile (now the aspect Volatile) requires access to be in memory but this is strictly unnecessary. All we need is to ensure that reads and writes occur in the right order. A new aspect Coherent was considered but was rejected in favour of simply changing the definition of Volatile.

The final improvement in this section is in the core language and concerns synchronized interfaces and requeue. The procedures of a synchronized interface may be implemented by a procedure or entry or by a protected procedure. However, in Ada 2005 it is not possible to requeue on a procedure of a synchronized interface even if it is implemented by an entry. This is a nuisance and prevents certain high level abstractions.

Accordingly, Ada 2012 has an aspect Synchronization that takes one of By_Entry, By_Protected_Procedure, and Optional. So we might write

```
type Server is synchronized interface;
procedure Q(S: in out Server; X: in Item);
  with Synchronization => By_Entry;
```

and then we are assured that we are permitted to perform a requeue on any implementation of Q.

As expected there are a number of consistency rules. The aspect can also be applied to a task interface or to a protected interface. But for a task interface it obviously cannot be By_Protected_Procedure.

In the case of inheritance, any Synchronization property is inherited. Naturally, multiple aspect specifications must be consistent. Thus Optional can be overridden by By_Entry or by By_Protected_Procedure but other combinations conflict and so are forbidden.

A related change is that if an entry is renamed as a procedure then we can do a requeue using the procedure name. This was not allowed in Ada 95 or Ada 2005.

3.5 General improvements

As well as the major features discussed above there are also a number of improvements in various other areas.

We start with some gentle stuff. Ada 95 introduced the package Ada thus

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

However, a close reading of the RM revealed that poor Ada is illegal since the pragma Pure is not in one of the allowed places for a pragma. Pragmas are allowed in the places where certain categories are allowed but not *in place of them*. In the case of a package specification the constructs are basic declarative items, but "items" were not one of the allowed things. This has been changed to keep Ada legal.

A related change concerns a sequence of statements. In a construction such as

```
if B then
  This;
else
  That;
end if;
```

there must be at least one statement in each branch so if we don't want any statements then we have to put a null statement. If we want a branch that is just a pragma Assert then we have to put a null statement as well thus

```
if B then
  pragma Assert(...); null;
end if;
```

This is really irritating and so the rules have been changed to permit a pragma in place of a statement in a sequence of statements. This and the problem with the package Ada are treated as Binding Interpretations which means that they apply to Ada 2005 as well.

A similar change concerns the position of labels. It is said that gotos are bad for you. However, gotos are useful for quitting an execution of a loop and going to the end in order to try the next iteration. Thus

```
for I in ... loop
  ...
     if this-one-no-good then goto End_Of_Loop; end if;
  ...
<<End_Of_Loop>> null;                    -- try another iteration
end loop;
```

Ada provides no convenient way of doing this other than by using a goto statement. Remember that **exit** transfers control out of the loop. The possibility of a continue statement as in some other languages was discussed but it was concluded that this would obscure the transfer of control. The great thing about **goto** is that the label sticks out like a sore thumb. Indeed, a survey of the code in a well known compiler revealed an orgy of uses of this handy construction.

However, it was decided that having to put **null** was an ugly nuisance and so the syntax of Ada 2012 has been changed to permit the label to come right at the end.

There is a significant extension to the syntax of loops used for iteration. This arose out of a requirement to make iteration over containers easier (as outlined in the next section) but the general ideas can be illustrated with an array. Consider

```
for K in Table'Range loop
  Table(K) := Table(K) + 1;
end loop;
```

This can now be written as

```
for T of Table loop
  T := T + 1;
end loop;
```

The entity T is a sort of generalized reference and hides the indexing. This mechanism can also be used with multidimensional arrays in which case just one loop replaces a nested set of loops.

A minor problem has arisen with the use of tags and Generic_Dispatching_Constructor. There is no way of discovering whether a tag represents an abstract type other than by attempting to create an object of the type which then raises Tag_Error; this is disgusting. Accordingly, a new function

```
function Is_Abstract(T: Tag) return Boolean;
```

is added to the package Ada.Tags.

There were many changes to access types in Ada 2005 including the wide introduction of anonymous access types. Inevitably some problems have arisen.

The first problem is with the accessibility of stand-alone objects of anonymous access types such as

```
A: access T;
```

Without going into details, it turns out that such objects are not very useful unless they carry the accessibility level of their value in much the same way that access parameters carry the accessibility level of the actual parameter. They are therefore modified to do this.

Programmers have always moaned about the need for many explicit conversions in Ada. Accordingly, implicit conversions from anonymous access types to named access types are now permitted provided the explicit conversion is legal. The idea is that the need for an explicit conversion with access types should only arise if the conversion could fail. A curious consequence of this change is that a preference rule is needed for the equality of anonymous access types.

An issue regarding allocators concerns their alignment. It will be recalled that when implementing a storage pool, the attribute Max_Size_In_Storage_Units is useful since it indicates the maximum size that could be requested by a call of Allocate. Similarly, the new attribute Max_Alignment_For_Allocation indicates the maximum alignment that could be requested.

Another problem is that allocators for anonymous access types cause difficulties in some areas. Rather than forbidding them completely a new restriction identifier is added so that we can write

```
pragma Restrictions(No_Anonymous_Allocators);
```

Another new restriction is

```
pragma Restrictions(No_Standard_Allocators_After_Elaboration);
```

This can be used to ensure that once the main subprogram has started no further allocation from standard storage pools is permitted. This prevents a long lived program suffering from rampant heap growth.

However, this does not prevent allocation from user-defined storage pools. To enable users to monitor such allocation, additional functions are provided in Ada.Task_Identification, namely Environment_Task (returns the Task_Id of the environment task) and Activation_Is_Complete (returns a Boolean result indicating whether a particular task has finished activation).

A new facility is the ability to define subpools using a new package System.Storage_Pools.Subpools. A subpool is a separately reclaimable part of a storage pool and is identified by a subpool handle name. On allocation, a handle name can be given.

Further control over the use of storage pools is provided by the ability to define our own default storage pool. Thus we can write

```
pragma Default_Storage_Pool(My_Pool);
```

and then all allocation within the scope of the pragma will be from My_Pool unless a different specific pool is given for a type. This could be done using the aspect Storage_Pool as expected

```
type Cell_Ptr is access Cell
  with Storage_Pool => Cell_Ptr_Pool;
```

A pragma Default_Storage_Pool can be overridden by another one so that for example all allocation in a package (and its children) is from another pool. The default pool can be specified as **null** thus

```
pragma Default_Storage_Pool(null);
```

and this prevents any allocation from standard pools.

Note that coextensions and allocators as access parameters may nevertheless be allocated on the stack. This can be prevented (somewhat brutally) by the following restrictions

```
pragma Restrictions(No_Coextensions);

pragma Restrictions(No_Access_Parameter_Allocators);
```

A number of other restrictions have also been added. The introduction of aspects logically requires some new restrictions to control their use. Thus by analogy with No_Implementation_Pragmas, we can write

```
pragma Restrictions(No_Implementation_Aspect_Specifications);
```

and this prevents the use of any implementation-defined aspect specifications. The use of individual aspects such as Default_Value can be prevented by

```
pragma Restrictions(No_Specification_of_Aspect => Default_Value);
```

Implementations and indeed users are permitted to add descendant units of Ada, System and Interfaces such as another child of Ada.Containers. This can be confusing for maintainers since they may be not aware that they are using non-standard packages. The new restriction

```
pragma Restrictions(No_Implementation_Units);
```

prevents the use of such units.

In a similar vein, there is also

```
pragma Restrictions(No_Implementation_Identifiers);
```

and this prevents the use of additional identifiers declared in packages such as System.

A blanket restriction can be imposed by writing

```
pragma Profile(No_Implementation_Extensions);
```

and this is equivalent to the following five restrictions

```
No_Implementation_Aspect_Specifications,
No_Implementation_Attributes,
No_Implementation_Identifiers,
No_Implementation_Pragmas,
No_Implementation_Units.
```

Finally, the issue of composability of equality has been revisited. In Ada 2005, tagged record types compose but untagged record types do not. If we define a new type (a record type, array type or a derived type) then equality is defined in terms of equality for its various components. However, the behaviour of components which are records is different in Ada 2005 according to whether they are tagged or not. If a component is tagged then the primitive operation is used (which might have been redefined), whereas for an untagged type, predefined equality is used even though it might have been overridden. This is a bit surprising and so has been changed in Ada 2012 so that all record types behave the same way and use the primitive operation. This is often called composability of equality so that we can say that in Ada 2012, record types always compose for equality. Remember that this only applies to records; components which are of array types and elementary types always use predefined equality.

3.6 Standard library

The main improvements in the standard library concern containers. But there are a number of other changes which will be described first.

In Ada 2005, additional versions of Index and Index_Non_Blank were added to the package Ada.Strings.Fixed with an additional parameter From indicating the start of the search. The same should have been done for Find_Token. So Ada 2012 adds

```
procedure Find_Token(Source: in String;
                     Set: in Maps.Character_Set;
                     From: in Positive;
                     Test: in Membership;
                     First: out Positive;
                     Last: out Natural);
```

Similar versions are added for bounded and unbounded strings to the corresponding packages.

New child packages of Ada.Strings are added to provide conversions between strings, wide strings, or wide wide strings and UTF8 or UTF16 encodings. They are

Ada.Strings.UTF_Encoding – declares a function Encoding to convert a String into types UTF_8, UTF_16BE, or UTF_16LE where BE and LE denote Big Endian and Little Endian respectively.

Ada.Strings.UTF_Encoding.Conversions – declares five functions Convert between the UTF schemes.

Ada.Strings.UTF_Encoding.Strings – declares functions Encode and Decode between the type String and the UTF schemes.

Ada.Strings.UTF_Encoding.Wide_Strings – declares six similar functions for the type Wide_String.

Ada.Strings.UTF_Encoding.Wide_Wide_Strings – declares six similar functions for the type Wide_Wide_String.

Further new packages are Ada.Wide_Characters.Handling and Ada.Wide_Wide_Characters.
Handling. These provide classification functions such as Is_Letter and Is_Lower and conversion functions such as To_Lower for the types Wide_Character and Wide_Wide_Character in a similar way to the existing package Ada.Characters.Handling for the type Character.

Experience with the package Ada.Directories added in Ada 2005 has revealed a few shortcomings.

One problem concerns case sensitivity. Unfortunately, common operating systems differ in their approach. To remedy this the following are added to Ada.Directories

```
type Name_Case_Kind is (Unknown, Case_Sensitive, Case_Insensitive, Case_Preserving);

function Name_Case_Equivalence(Name: in String) return Name_Case_Kind;
```

Calling Name_Case_Equivalence enables one to discover the situation for the operating system concerned.

Another problem is that the basic approach in Ada.Directories is a bit simplistic and assumes that file names can always be subdivided into a directory name and a simple name. Thus the existing function Compose is

```
function Compose(Containing_Directory: String := "";
                 Name: String; Extension: String := "") return String;
```

and this requires that the Name is a simple name such as "My_File" with possibly an extension if one is not provided.

Accordingly, an optional child package is introduced, Ada.Directories.Hierarchical_File_Names, and this adds the concept of relative names and a new version of Compose whose second parameter is a relative name and various functions such as Is_Simple_Name and Is_Relative_Name.

Programs often need information about where they are being used. This is commonly called the Locale. As an example, in some regions of the world, a sum such as a million dollars is written as \$1,000,000.00 whereas in others it appears as \$1.000.000,00 with point and comma interchanged.

An early attempt at providing facilities for doing the right thing was fraught with complexity. So Ada 2012 has adopted the simple solution of enabling a program to determine the country code (two characters) and the language code (three characters) and then do its own thing. The codes are given by ISO standards. Canada is interesting in that it has one country code ("CA") but uses two language codes ("eng" and "fra").

The information is provided by a new package Ada.Locales which declares the codes and the two functions Language and Country to return the current active locale (that is, the locale associated with the current task).

And finally, we consider the container library. Containers were a major and very valuable addition to Ada 2005 but again, experience with use has indicated that some enhancements are necessary.

We start with a brief summary of what is in Ada 2005. The parent package Ada.Containers has six main children namely Vectors, Doubly_Linked_Lists, Hashed_Maps, Ordered_Maps, Hashed_Sets, and Ordered_Sets. These manipulate definite types.

In addition there are another six for manipulating indefinite types with names such as Indefinite_Vectors and so on.

There are also two packages for sorting generic arrays, one for unconstrained types and one for constrained types.

There are four new kinds of containers in Ada 2012

- bounded forms of the existing containers,
- a container for a single indefinite object,
- various containers for multiway trees, and
- various containers for queues.

In addition there are a number of auxiliary new facilities whose purpose is to simplify the use of containers.

We will start by briefly looking at each of the new kinds of containers in turn.

The existing containers are unbounded in the sense that there is no limit to the number of items that can be added to a list for example. The implementation is expected to use storage pools as necessary. However, many applications in high integrity and real-time areas forbid the use of access types and require a much more conservative approach. Accordingly, a range of containers is introduced with bounded capacity so that there is no need to acquire extra storage dynamically.

Thus there are additional packages with names such as Containers.Bounded_Doubly_Linked_Lists. A key thing is that the types List, Vector and so on all have a discriminant giving their capacity thus

```
type List(Capacity: Count_Type) is tagged private;
```

so that when a container is declared its capacity is fixed. A number of consequential changes are made as well. For example, the bounded form has to have a procedure Assign

```
procedure Assign(Target: in out List; Source: in List);
```

because using built-in assignment would raise Constraint_Error if the capacities were different. Using a procedure Assign means that the assignment will work provided the length of the source is not greater than the capacity of the target. If it is, the new exception Capacity_Error is raised.

Moreover, a similar procedure Assign is added to all existing unbounded containers so that converting from a bounded to an unbounded container or vice versa is (reasonably) straightforward.

Conversion between bounded and unbounded containers is also guaranteed with respect to streaming.

There are no bounded indefinite containers; this is because if the components are indefinite then dynamic space allocation is required for the components anyway and making the overall container bounded would be pointless.

In Ada, it is not possible to declare an object of an indefinite type that can hold any value of the type. Thus if we declare an object of type String then it becomes constrained by the mandatory initial value.

```
S: String := "Crocodile";
```

We can assign other strings to S but they must also have nine characters. We could assign "Alligator" but not "Elephant". (An elephant is clearly too small!)

This rigidity is rather a nuisance and so a new form of container is defined which enables the cunning declaration of an object of a definite type that can hold a single value of an indefinite type. In other words it is a wrapper. The new package is Ada.Containers.Indefinite_Holders and it takes a generic parameter of the indefinite type and declares a definite type Holder which is tagged private thus

```
generic
  type Element_Type (<>) is private;
  with function "="(Left, Right: Element_Type) return Boolean is <>;
package Ada.Containers.Indefinite_Holders is
  type Holder is tagged private;
  ...                    -- various operations
end Ada.Containers. Indefinite_Holders;
```

The various operations include a procedure Replace_Element which puts a value into the holder and a function Element which returns the current value in the holder.

Three new containers are added for multiway trees (unbounded, bounded, and indefinite). It might have been thought that it would be easy to use the existing containers such as the list container. But it is difficult for various reasons concerning memory management. And so it was concluded that new containers for multiway trees should be added to Ada 2012.

The package Ada.Containers.Multiway_Trees is the unbounded form similar to the existing containers for other structures. It has all the operations required to operate on a tree structure where each node can have multiple child nodes to any depth. Thus there are operations on subtrees, the ability to find siblings, to insert and remove children and so on. The other new containers are Ada.Containers.Bounded_Multiway_Trees and Ada.Containers.Indefinite_Multiway_Trees which provide bounded and indefinite forms respectively.

Finally, there is a group of containers for queues. This topic is particularly interesting because it has its origins in the desire to provide container operations that are task safe. However, it turned out that it was not easy to make the existing containers task safe in a general way which would satisfy all users because there are so many possibilities.

However, there was no existing container for queues and in the case of queues it is easy to see how to make them task safe.

There are in fact four queue containers and all apply to queues where the element type is definite; these come in both bounded and unbounded forms and for synchronized and priority queues. We get (writing AC as an abbreviation for Ada.Containers)

- AC.Unbounded_Synchronized_Queues,

- AC.Bounded_Synchronized_Queues,
- AC.Unbounded_Priority_Queues,
- AC.Bounded_Priority_Queues.

These in turn are all derived from a single synchronized interface. This is a good illustration of the use of synchronized interfaces and especially the aspect Synchronization discussed earlier (see Section 3.4). First there is the following generic package which declares the type Queue as a synchronized interface (writing AC as an abbreviation for Ada.Containers and ET for Element_ Type)

```
generic
  type ET is private;  -- element type for definite queues
package AC.Synchronized_Queue_Interfaces is
  pragma Pure(...);
  type Queue is synchronized interface;

  procedure Enqueue(Container: in out Queue; New_Item: in ET) is abstract
    with Synchronization => By_Entry;

  procedure Dequeue(Container: in out Queue; Element: out ET) is abstract
    with Synchronization => By_Entry;

  function Current_Use(Container: Queue) return Count_Type is abstract;
  function Peak_Use(Container: Queue) return Count_Type is abstract;
end AC.Synchronized_Queue_Interfaces;
```

Then there are generic packages which enable us to declare actual queues. Thus the essence of the unbounded synchronized version is as follows (still with abbreviations AC for Ada.Containers, ET for Element_Type)

```
with System; use System;
with AC.Synchronized_Queue_Interfaces;
generic
  with package Queue_Interfaces is new AC.Synchronized_Queue_Interfaces(<>);
  Default_Ceiling: Any_Priority := Priority'Last;
package AC.Unbounded_Synchronized_Queues is
  pragma Preelaborate(...);

  package Implementation is
     -- not specified by the language
  end Implementation;

  protected type Queue(Ceiling: Any_Priority := Default_Ceiling)
                  with Priority => Ceiling
                    is new Queue_Interfaces.Queue with

    overriding
    entry Enqueue(New_Item: in Queue_Interfaces.ET)
    overriding
    entry Dequeue(Element: out Queue_Interfaces.ET);

    overriding
    function Current_Use return Count_Type;
```

```
      overriding
      function Peak_Use return Count_Type;

    private
      ...
    end Queue;

  private
    ...
  end AC.Unbounded_Synchronized_Queues;
```

The discriminant gives the ceiling priority and for convenience has a default value. Remember that a protected type is limited and when used to implement an interface (as here) is considered to be tagged. In Ada 2012, defaults are allowed for discriminants of tagged types provided they are limited as mentioned in Section 3.3.

Note that the Priority is given by an aspect specification. Programmers who are allergic to the multiple uses of **with** could of course use the old pragma Priority in their own code.

(The need for the package Implementation will be briefly explained in a later paper and can be completetly ignored by the user.)

Now to declare our own queue of integers say we first write

```
package My_Interface is new AC.Synchronized_Queue_Interfaces(ET => Integer);
```

This creates an interface for dealing with integers. Then to obtain an unbounded queue package for integers we write

```
package My_Q_Package is new AC.Unbounded_Synchronized_Queues(My_Interface);
```

This creates a package which declares a protected type Queue. Now at last we can declare an object of this type and perform operations on it.

```
The_Queue: My_Q_Package.Queue;
...
The_Queue.Enqueue(37);
```

The various calls of Enqueue and Dequeue are likely to be in different tasks and the protected object ensures that all is well.

The other generic queue packages follow a similar style. Note that unlike the other containers, there are no queue packages for indefinite types. Indefinite types can be catered for by using the holder container as a wrapper or by using an access type.

In Ada 2005 there are two generic procedures for sorting arrays; one is for constrained arrays and one is for unconstrained arrays. In Ada 2012, a third generic procedure is added which can be used to sort any indexable structure. Its specification is

```
generic
  type Index_Type is (<>);
  with function Before(Left, Right: Index_Type) return Boolean;
  with procedure Swap(Left, Right: Index_Type);
procedure Ada.Containers.Generic_Sort(First, Last: Index_Type'Base);
pragma Pure(Ada.Containers.Generic_Sort);
```

Note that there is no parameter indicating the structure to be sorted; this is all done indirectly by the subprograms Before and Swap working over the range of values given by First and Last. It's almost magic!

A frequent requirement when dealing with containers is the need to visit every node and perform some action, in other words to iterate over the container. And there are probably many different iterations to be performed. In Ada 2005, this has to be done by the user defining a subprogram for each iteration or writing out detailed loops involving calling Next and checking for the last element of the container and so on. And we have to write out this mechanism for each such iteration.

In Ada 2012, after some preparatory work involving the new package Ada.Iterator.Interfaces it is possible to simplify such iterations hugely. For example, suppose we have a list container each of whose elements is a record containing two components of type Integer (P and Q say) and we want to add some global X to Q for all elements where P is a prime. In Ada 2005 we have to write the laborious

```
C := The_List.First;          -- C declared as of type Cursor
loop
   exit when C = No_Element;
   E := Element(C);
   if Is_Prime(E.P) then
      Replace_Element(C, (E.P, E.Q + X));
   end if;
   C := Next(C);
end loop;
```

Not only is this tedious but there is lots of scope for errors. However, in Ada 2012 we can simply write

```
for E of The_List loop
   if Is_Prime(E.P) then E.Q := E.Q + X; end if;
end loop;
```

The mechanism is thus similar to that introduced in the previous section for arrays.

There are also a number of minor new facilities designed to simplify the use of containers. These include the introduction of case insensitive operations for comparing strings and for writing hash functions.

4 Conclusions

This overview of Ada 2012 should have given the reader an appreciation of the important new features in Ada 2012. Some quite promising features failed to be included partly because the need for them was not clear and also because a conclusive design proved elusive.

Further papers will expand on the six major topics of this overview in more detail.

It is worth briefly reviewing the guidelines (see Section 2 above) to see whether Ada 2012 meets them.

The group A items were about extending the advantages of Ada and specifically mentioned containers, contracts and real-time. There are many new features for containers, pre- and postconditions have been added and so have facilities for multiprocessors.

The group B items were about eliminating shortcomings, increasing safety and particularly mentioned improvements to access types and storage management. This has been achieved with corrections to accessibility checks, the introduction of subpools and so on.

It seems clear from this brief check that indeed Ada 2012 does meet the objectives set for it.

Finally, I need to thank all those who have helped in the preparation of this paper and especially Randy Brukardt, Ed Schonberg and Tucker Taft.

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] ISO/IEC TR 24718:2004 *Guide for the use of the Ada Ravenscar profile in high integrity systems.*

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

[4] ISO/IEC 8652:1995/AMD 1:2007 *Ada Reference Manual – Amendment 1*.

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