Rationale Update
for
Ada 2012

John Barnes

John Barnes Informatics,
11 Albert Road, Caversham, Reading RG4 7AN, UK

Tel: +44 118 947 4125
email: jgpb@jbinfo.demon.co.uk

Table of Contents

Chapter 1: Introduction

The first version of Ada (Ada 83) was developed by a team led by the late Jean Ichbiah and funded by the USDoD. The development of Ada 95 was done under the leadership of Tucker Taft and also funded by the USDoD. Then came Ada 2005 and Ada 2012 [1, 2] which were developed on a more modest scale and largely done by voluntary effort with support from within the industry itself by bodies such as the Ada Resource Association and Ada-Europe.

The Ada Rapporteur Group (ARG) is a team of experts nominated by the national bodies represented on WG9 (ISO/IEC JTC 1/SC22/WG9, the working group for Ada) 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 was led by Ed Schonberg. Since the Ada 2012 standard was issued the ARG has been led by Jeff Cousins. The editor, who at the end of the day actually writes the words of the standard, continues to be the indefatigable Randy Brukardt and the convenor of WG9 is Joyce Tokar.

The changes made to Ada 2012 by the update are presented in the same areas as in the Ada 2012 Rationale [3] to ease any comparison. In each area the relevant Ada Issues (AIs) are listed and this is then followed by a brief discussion. It will be observed that many issues concern corner cases which will not be of much interest to the average user. Moreover, the changes are usually corrections to the descriptive text. But just occasionally we get the excitement of a new aspect, a new type or subprogram and even a new bit of syntax!

Note that some Ada Issues were mentioned in the Postscript section of the Ada 2012 Rationale published after the standard was approved (and in Programming in Ada 2012 [4] by the author). They are given here for completeness and are marked with *. But note that they might have changed again since the Rationale was written.

The updated version of Ada 2012 was approved by WG 9 in June 2015, by SC 22 in December 2015, and published by ISO in February 2016. It is formally ISO/IEC 8652:2012/Cor 1:2016, entitled Technical Corrigendum 1 for Ada 2012, or TC1 for short [5].

Chapter 2: Contracts and aspects

This is perhaps the most exciting area of Ada 2012 and covers matters such as the new notation for aspects which replaces many uses of pragmas and also the introduction of contracts. Contracts include pre- and postconditions, type invariants, and subtype predicates.

The following Ada issues cover this area:

32	Questions on 'Old
41	Type_Invariant'Class for interface types
42	Type invariant checking rules
44*	Calling visible functions from type invariant expressions
45*	Pre- and Postconditions are allowed for generic subprograms
49	Invariants need to be checked on the initialization of deferred constants
54*	Aspect Predicate_Failure
68	Predicates and the current instance of a subtype
71*	Order of evaluation when multiple predicates apply
77	Has_Same_Storage on objects of size zero
96	The exception raised when a subtype conversion fails a predicate check
99	Wording problems with predicates
104	Overriding an aspect is undefined
105	Pre and Post are not allowed on any subprogram completion
113	Class-wide preconditions and statically bound calls
116	Private types and predicates
131	Inherited Pre'Class when unspecified on initial subprogram
133	Type invariants and default initialized objects
149	Type invariants are checked for functions returning access-to-type
150	Class-wide type invariants and statically bound calls
154	Aspects of library units

These changes can be grouped as follows.

There are a number of clarifications on pre- and postconditions in general (45, 105), some additional rules on the use of the aspect Old (32), and a trivial omission on the attribute Has_Same_Storage (77).

There are also clarifications regarding types and the use of class wide conditions and invariants (113, 150).

A number of additional rules are required describing the places where type invariants are checked (41, 42, 44, 49, 133, 149).

A new aspect Predicate_Failure and rules concerning checking order are added (54, 71, 96). A number of clarifications to the rules for predicates are added (68, 99, 116).

Finally, there are minor clarifications regarding aspects in general (104, 154).

AI-45 notes that pre-and postconditions (both specific and classwide) are allowed on generic programs but they are not allowed on instances of generic subprograms. This avoids awkward maintenance problems that might arise if such conditions were allowed on both a generic and an instance. On a similar matter, it was the intent that pre- and postconditions should always be visible to the user and so go on specifications and not bodies which are completions. However, the wording forgot to mention the new expression functions which as well as standing alone can also be completions (see the example of Non_Zero in AI-49 below). Such expression functions and null procedures acting as completions cannot have pre- and postconditions either (AI-105).

The attribute 'Old which is used in a postcondition to indicate the initial value of a parameter (or maybe a global) is a bit tricky. AI-32 adds some more (fairly obvious) detail such as that X and X'Old have the same type even if it is anonymous.

A trivial point about the attribute Has_Same_Storage which is useful in preconditions is that the expression X'Has_Same_Storage(Y) returns False if X or Y or both occupy zero bits. (AI-77).

AI-131 concerns Pre'Class. If the initial definition of a subprogram does not specify Pre'Class then the corresponding subprograms of derived types just inherit True as one would expect. It also notes that one cannot specify Pre'Class for an overriding subprogram of a type unless Pre'Class is specified for some ancestor. This is because it would be ored with True and thus have no effect.

AI-113 and AI-150 address a problem with Pre'Class, Post'Class and Type_Invariant'Class and parameters. Typically we might have something like

```
procedure P(S: in out T; ...)
  with Pre'Class => expression involving S
```

where T is a tagged type. The procedure P can be called with a statically bound parameter or indeed with a dynamically bound classwide parameter. The wording is clarified that the actual types of the actual parameters are always used. The reader is referred to the text of the Ada Issues which have a large example which should be helpful.

It is well-known that type invariants are not intended to be foolproof but to be helpful in catching many flaws (unlike pre- and postconditions which are meant to be perfect). As time has passed more little quirks have been found regarding when type invariants should or should not be checked. Remember that invariants apply to private types. When we have a full view of the type (as in a subprogram in the body of the package declaring the type) then we can change an object of the type temporarily to a state where the invariant does not apply (this is often necessary for intermediate stages in manipulation). However, when we leave the full view by for example returning from a subprogram in the package body then checks are applied on parameters and so on.

AI-44 states that type invariants are not checked on **in** parameters of functions but are checked on **in** parameters of procedures. This is necessary to avoid infinite recursion which would arise if an invariant itself calls a function with a parameter of the type. Moreover, a classwide invariant could not be used at all without this modification.

AI-49 adds that invariants need to be checked on the initialization of deferred constants (other initializations were already covered). An example is as follows

```
package R is
  type T is private
    with Type_Invariant => Non_Zero(T);
  function Non_Zero(X: T) return Boolean;
  Zero: constant T;
private
  type T is new Integer;
  function Non_Zero(X: T) return Boolean is
    (X /= 0);
  Zero: constant T := 0;
end R;
```

Currently, this is not caught but the declaration of Zero should raise Assertion_Error. Note the neat use of an expression function for the completion of Non_Zero.

AI-133 concerns default initializations and type invariants. Remember that initialization by default occurs in various circumstances. If a record type has components with an initial value as in

```
type R is
  record
    C: Integer := 99;
  end record;
```

then when we declare an object of type R without any initialization thus

```
X: R;
```

then we are assured that X.C will have the value 99. Moreover, in Ada 2012 we can give default values for other types by the new aspects Default_Value and Default_Component_Value (these are discussed in 3). But we cannot give defaults for existing predefined types such as Integer since we can only give the default when the type is declared.

The question arises as to whether or not the rule about default initializations being checked for the type invariant applies to an object declared inside the package body. Normally, of course, we can do what we like inside the body so we might expect such initializations not to be checked. However, since the default initialization mechanism is the same for clients outside the body as it is for the writer of the body, it was felt that it would be better if they were all checked for consistency and simplicity. An error might thus be detected earlier.

But there is an exception. If the partial view of the type has unknown discriminants then the user cannot declare objects of the type anyway and so no check could be performed externally. For uniformity no check is performed internally either.

AI-149 adds checks for functions whose return type is an access type with a designated type having a part of the type concerned (parameters were already covered). Further similar checks are added by AI-42 concerning type extensions. It is all a bit subtle and maybe other problems are lurking. One is tempted to quote a wee bonny poem of Sir Walter Scott

O what a tangled web we weave, When first we practise to deceive!

A more exciting change (AI-41) is to allow Type_Invariant'Class on interface types. This can be used to ensure that any type derived from the interface satisfies certain properties. The following example is given

```
type Window is interface
  with Type_Invariant'Class =>
    Window.Width * Window.Height = 100;
```

```
function Width(W: Window) return Integer is abstract;
function Height(W: Window) return Integer is abstract;
```

This ensures that any type derived from the interface Window will provide functions giving the height and width of the windows such that their product is exactly 100. This is a display window of 100 pixels (poor quality I fear).

If we derive a type from several such interfaces then the Type_Invariant'Class is of course the conjunction of the individual invariants.

A new aspect Predicate_Failure is introduced by AI-54 (with a wording correction in AI-96). A related issue concerning the order of evaluation of predicates is discussed in AI-71. These were discussed in detail in the Ada 2012 Rationale and in Section 16.5 of Programming in Ada 2012. A short extract will suffice here.

The expected type of the expression defined by the aspect Predicate_Failure is String and gives the message to be associated with a failure. So we can write

```
subtype Open_File_Type is File_Type
  with
    Dynamic_Predicate => Is_Open(Open_File_Type),
    Predicate_Failure => "File not open";
```

If the predicate fails then Assertion_Error is raised with the message "File not open".

We can also use a raise expression (see AI-22 in the next section) and thereby ensure that a more appropriate exception is raised. If we write

```
Predicate_Failure =>
  raise Status_Error with "File not open";
```

then Status_Error is raised rather than Assertion_Error with the given message. We could of course explicitly mention Assertion_Error thus by writing

```
Predicate_Failure =>
  raise Assertion_Error with "A message";
```

Finally, we could omit any message and just write

```
Predicate_Failure => raise Status_Error;
```

in which case the message is null.

A related issue is discussed in AI-71. If several predicates apply to a subtype which has been declared by a refined sequence then the predicates are evaluated in the order in which they occur. This is especially important if different exceptions are specified by the use of Predicate_Failure since without this rule the wrong exception might be raised. The same applies to a combination of predicates, null exclusions and old-fashioned subtypes.

There are a number of minor wording omissions and corrections with the description of predicates. AI-116 notes that one cannot give a specification of an aspect such as Dynamic_Predicate to both the full view and private view of a type (there is such a rule for most aspects but it was forgotten for predicates). A subtle point covered by AI-68 is that within a predicate the current instance (such as T in the expression Non_Zero(T) in the example of AI-49) acts like a value rather than an object. (This prevents certain undesirable uses of T such as applying a number of object attributes.) AI-99 confirms that **not** is an allowed operation in a Static_Predicate and also corrects some wording in relation to tasks and protected objects.

An interesting topic is addressed by AI-154. It may be recalled that rather than writing

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

we can instead write

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

or more pedantically even

```
package P
  with Pure => True is
...
```

or really foolishly

```
package P
  with Pure => False is
...
```

Using a pragma rather than an aspect specification is the preferred style for library unit pragmas. But there might be some benefit in using an aspect specification since one could change the state of a whole group of packages in one blow by a structure such as

```
package State_Control is
  Purity: constant Boolean := True;
  ...
end State_Control;

...

with State_Control;
package P
  with Pure  => State_Control.Purity is
  ...
end P;

...        --  and so on for several packages
```

We could then make Purity false for adding some debugging material during development and then set it to true for final production.

Observe that AI-154 illustrates that one could be silly enough to try to write

```
package P with Pure => Purity is
  ...
  Purity: constant Boolean := True;    -- illegal
end P;
```

which raises the question of when is the wretched static expression Purity evaluated. The answer is immediately of course just like the pragma and so the example is illegal because Purity has not yet been elaborated. Note that aspects such as type invariants in

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

require the elaboration to be deferred. But this cannot be done with Pure because it controls the application of some Legality Rules.

The final AI-104 regarding aspects simply replaces a confusing sentence by a (confusing?) user note. Aspects such as Constant_Indexing can be inherited; the aspects themselves cannot be redefined but the functions they denote can be modified by overriding or by overloading.

Chapter 3: Expressions

The introduction of contracts triggered the need for more flexible forms of expressions in Ada 2012. These are conditional expressions (if and case), quantified expressions, and expression functions. In addition membership tests were made much more flexible.

The following Ada issues cover this area:

22*	Raise expressions
39*	Ambiguity in syntax for membership expression removed
40	Resolving the selected_expression of a case_expression
50	Conformance of quantified expressions
62	Raise expression with failing string function
84	Box expressions in array aggregates
100	A qualified expression makes a predicate check
103	Expression functions that are completions in package specifications
141	Add raise expression to Introduction
147	Expression functions and null procedures can be declared in a protected_body
152	Eliminate ambiguities in raise expression and derived type syntax
157	Missing rules for expression functions
158	Definition of quantified expressions

These changes can be grouped as follows.

The most important change in this update is perhaps the introduction of raise expressions (22, 62, 141, 152); it is convenient to discuss a change to the syntax for membership test at the same time (39).

A number of changes relate to expression functions (103, 147, 157). There are also some clarifications regarding quantified expressions (50, 158).

There are miscellaneous changes to qualified expressions (100), case expressions (40), and array aggregates (84).

The introduction of raise expressions by AI-22 was deemed important enough to be mentioned in the Introduction to the revised RM. It was discussed in some detail in the Postscriptchapter of the Ada 2012 Rationale which was written after the Ada 2012 standard was published. However, the discussion there needs updating since the syntax rules have been modified as a consequence of ambiguities mentioned in AI-39 and AI-152. So here is a more integrated description.

The raise expression, is added by analogy with if statements and the raise statement. Thus as well as

```
if X < Y then
  Z := +1;
elsif X > Y then
  Z := –1;
else
  raise Error;
end if;
```

we can also write

```
Z := (if X<Y then 1 elsif X>Y then –1 else raise Error);
```

The syntax for raise expression is now as follows

```
raise_expression ::=
     raise exception_name [with string_simple_expression]
```

Note that unlike in a raise statement, the string expression has to be a simple_expression rather than an expression in order to avoid ambiguities involving logical operations.

A raise expression is a new form of relation (as will be seen in the syntax in a moment) and has the same precedence and so will need to be in parentheses in some contexts. But as illustrated above it does not need parentheses when used in a conditional expression which itself will have parentheses.

Raise expressions will be found useful with pre- and postconditions. Thus if we have

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

and the precondition is false then Assertion_Error is raised. But we can now alternatively write

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S) or else raise Stack_Error;
```

and of course we can also add a message thus

```
    Pre => not Is_Full(S) or else
           raise Stack_Error with "wretched stack is full";
```

Another issue concerns what happens if the string expression in a raise expression (or indeed in a raise statement) itself raises an exception; it could be a function call which returns the string. The answer is that the one caused by the string expression is propagated instead of the one given in the raise expression or statement (AI-62).

On a closely related topic the syntax for membership tests has been found to cause ambiguities (AI-39).

Thus

```
A in B and C
```

could be interpreted as either of the following

```
(A in B) and C        -- or
A in (B and C)
```

This is cured by changing the syntax for relation to

```
relation ::=
   simple_expression [relational_operator simple_expression]
  | tested_simple_expression [not] in membership_choice_list
  | raise_expression
```

and changing membership choice to use simple_expression as well

```
membership_choice ::=
   choice_simple_expression | range | subtype_mark
```

Thus a membership_choice no longer uses a choice_expression. However, the form choice_expression is still used in discrete_choice.

When first written, AI-22 showed the syntax for a raise expression using *string*_expression just as in raise statement. However, this caused ambiguities as mentioned earlier so it was changed to *string*_simple_expression by AI-152. Curiously enough it was also necessary to change the syntax of digits constraint and delta constraint to use simple expression as well. The AI has the following bizarre example

```
Atomic: String := "Gotcha";

type Fun is new My_Decimal_Type digits
  raise TBD_Error with Atomic;
```

This could be parsed as either

```
type Fun is new My_Decimal_Type digits
  (raise TBD_Error with Atomic);
```

or

```
type Fun is new My_Decimal_Type digits
  (raise TBD_Error) with Atomic;
```

So we now have

```
digits_constraint ::=
  digits static_simple_expression [range_constraint]
delta_constraint ::=
  delta static_simple_expression [range_constraint]
```

It seems cruel to have to change delta constraint which one might have thought was peacefully buried in Annex J for obsolescent features.

These potential ambiguities are unlikely to impact the normal user. If the compiler complains then the judicious insertion of some parentheses will undoubtedly cure the problem.

Expression functions were added in Ada 2012. Remember that an expression function takes the form

```
function F ( ... ) return T is
  (expression of subtype T);
```

A good example was given earlier thus

```
  function Non_Zero(X: T) return Boolean is
    (X /= 0);
```

Remember that such functions can act as a complete function or as a completion of a traditional function specification.

A number of points were overlooked in the definition of Ada 2012. One has already been mentioned namely that function expressions acting as completions cannot have pre- and postconditions (see AI-105 in Chapter 2).

Another point is that expression functions and indeed null procedures can be used in the body of a protected type as a completion of a protected subprogram (AI-147). This requires a modification to the syntax which becomes

```
protected_operation_item ::=
   subprogram_declaration
  | subprogram_body
  | null_procedure_declaration
  | expression_function_declaration
  | entry_body
  | aspect_clause
```

An interesting situation arises if the result expression of an expression function can be written as an aggregate as for example

```
function Conjugate (C: Complex) return Complex is
   ((C.Rl, –C.Im));          -- double parens
```

Remember that the conjugate of a complex number *C* has the same real part but the imaginary part changes sign so that the conjugate point in the Argand plane is the reflection of *C* in the real axis.

The original rules say that the expression of an expression function is simply an expression in parentheses. However, this is ugly if the expression already has parentheses as occurs with an aggregate as above. Now Ada dislikes double parentheses so we have rules for if expressions that they have to be in parentheses unless the context already supplies parentheses as in the case of a subprogram call with a single parameter. Consequently, AI-157 concludes that the second lot of parentheses are unnecessary so we can just write

```
function Conjugate (C: Complex) return Complex is
   (C.Rl, –C.Im);            -- single parens
```

As a result the syntax is revised to

```
expression_function_declaration ::=
   [overriding _indicator]
   function_specification is
      (expression)
      [aspect_specification] ;
 | [overriding _indicator]
   function_specification is
      aggregate
      [aspect_specification] ;
```

The above example shows a record aggregate. The same applies to array aggregates in parentheses. But of course if the result is given as a string then the parentheses are necessary. So we can have either of

```
function Piggy return String is ('P', 'I', 'G');

function Piggy return String is ("PIG");
```

There are also changes to the freezing rules which will probably leave the reader cold. These are in AI-157 and AI-103. A simple one is that expression functions acting as completions only freeze the expression and nothing else and null procedures never freeze anything.

There are a couple of minor points regarding quantified expressions.

AI-158 clarifies the result of a quantified expression where the array concerned has zero elements. In the case of the existential qualifier **for some**, the result is False whereas in the case of the universal qualifier **for all**, the result is True. So consider

```
B := (for all K in A'Range => A(K) = 0);
```

which assigns true to B if every element of A is zero. If A doesn't have any elements then the result is still true. Similarly

```
B := (for some K in A'Range => A(K) = 0);
```

assigns true to B provided at least one element of A is zero. If A doesn't have any elements then the answer is false. This all seems pretty obvious but the wording was deemed to require clarification for those not having a Fields Medal in mathematics.

Another quirk is discussed by AI-50 which is concerned with conformance. Remember that the parameters in the specification and body of a subprogram have to conform. The introduction of quantified expressions means that such an expression could occur as the default value in a subprogram specification; thus using the example above we might have

```
procedure P(B : Boolean :=
  (for all K in A'Range => A(K) = 0));
```

The corresponding text in the procedure body has to conform and additional rules are required to ensure this. The new thing is that quantified expressions introduce declarations such as that of K in the parameter list and we have to say that these two (technically different) declarations are the same in specification and body and specifically that the two defining identifiers are the same and are used in the same way.

A minor omission concerns qualified expressions (AI-100) Remember the difference between a qualified expression and a conversion. Qualification (which takes a quote) just states that an expression has the given (sub)type and is often used for resolving ambiguities. Conversion (which does not have a quote) actually changes the type (if necessary). Qualification also checks any relevant subtype properties. But on the addition of subtype predicates although it was added that they were checked on type conversions, it was forgotten to add that any subtype predicates should also be checked on qualification.

A very minor omission is covered by AI-40 which says that case expressions and case statements resolve in exactly the same way — that is have the same rules for type matching.

Finally, AI-84 concerns the use of the box notation <> in array aggregates. This was added in Ada 2005 and indicates that a component takes its default value which is the same as the default value for a stand-alone object.

However, Ada 2012 added the aspects Default_Value and Default_Component_Value. So we might write

```
type My_Integer is new Integer
  with Default_Value => 999;

type My_Array is array (Integer range <>) of My_Integer
  with Default_Component_Value => 666;
```

If we declare

```
X: My_Array(1 .. 10);
```

then the value of X(1) will be 666 of course using the aspect Default_Component_Value.

But if we write

```
X: My_Array(1..10) := (others => <>);
```

then very surprisingly X(1) was 999 rather than 666. The rules for <> were not updated to note that if the Default_Component_Value has been given then that applies rather than the stand-alone value. This is put right by AI-84 so that the value is now 666 in both cases.

Chapter 4: Structure and visibility

One of the most dramatic changes in Ada 2012 concerns subprogram parameters and is that functions can have parameters of all modes. Other areas covered here include incomplete types and discriminants.

The following Ada Issues cover this area:

65	Descendants of incomplete views
74	View conversions and out parameters passed by copy
94	An access definition should be a declarative region
95	Generic formal types and constrained partial views
97	Tag of the return object of a simple return expression
101	Incompatibility of hidden untagged record equality
109	Representation of untagged derived types
132	Freezing of renames-as-body
137	Incomplete views and access to class wide types

These changes can be grouped as follows.

A number of issues concern views. There are clarifications of incomplete views (65, 137) and omissions concerning view conversions (74) and constrained views (95).

An amusing issue concerns the definition of a declarative region (94).

Miscellaneous issues concern renaming (132), untagged record equality (101), untagged derived types (109), and the tag of return objects (97).

A curious situation is discussed by AI-65. A type T3 can be a descendant of T1 but nevertheless inherits no characteristics of T1 because of an intervening type T2. Consider

```
package P is
  type T1 is private;              -- partial view
  C: constant: T1;
private
  type T1 is new Integer;          -- complete view
  C: constant := 37;               -- my favourite number
end P;

with P;
package Q is
  type T2 is new P.T1;
end Q;

with Q;
package P.Child is
  type T3 is new Q.T2;
private
                                   -- what can we do with T3 here?
end P.Chlld;
```

In this example T3 is derived from T2 and T2 is derived from T1. The fact that T2 is derived from Integer is not visible to the declaration of T3. Nevertheless the conversion rules allow a value of type T1 to be converted to T3 in the private part of the child package. But the fact that T3 is an integer type is not visible.

We say that T3 is effectively a descendant of an incomplete view of T1. (Note "effectively"; it's not technically an incomplete view but behaves in some ways as if it were.) So we can convert C but not 73 to type T3 in the private part of C.Child.

```
X: T3 := T3(P.C)            -- OK
Y: T3 := T3(73);            -- No, T3 is not visibly numeric!
```

It was meant to be like this in Ada 95; Ada 2005 meddled with it and Ada 2012 made a confusing "improvement". Hopefully the clarifications made now will be the end of the story.

It is helpful to remember the distinction between a partial view and an incomplete view.

- A partial view is the view given by a private type declaration in contrast to the full view given by the full declaration in the private part. As in type T1 above.
- An incomplete view is the view given by an incomplete declaration such as occurs with access types. Thus

```
type Cell;                  -- incomplete view
type Link is access Cell:

type Cell is                -- completion
  record
    Next: Link;
    ...
  end record;
```

The use of the concept of incomplete views was much extended in Ada 2005 by the introduction of the limited with clause. It was extended again in Ada 2012 by allowing incomplete types to be completed by types other than access types and allowing incomplete views as parameters.

There are many rules concerning access types that designate incomplete views. AI-137 clarifies that they also apply to access to class wide types.

AI-95 concerns an omission/confusion with regard to generic untagged formal types and partial views. Briefly, within a generic body we assume the worst as to whether or not a formal subtype has a constrained partial view. In particular we assume that untagged formal private and derived types do indeed have a constrained partial view.

As Ada has grown there have been further lexical amusements such as functions returning access to functions. Thus we can now have

```
type T is
  access function( ... ) return
    access function( ... ) return
      access function( ... ) return ...
```

ad infinitum. To be more specific the rules seem to prohibit

```
type T is
  access function(A: Integer) return
    access function(A: Float) return Boolean;
```

because here we have two instances of A in the declarative region for T. There is no real reason why this should not be permitted so the definition of declarative region is extended to include an access definition (AI-94).

A somewhat different topic is addressed by AI-74 and concerns parameters of mode **out** which have always been the source of troubles. The basic problem is that such parameters can become undefined. Consider this simple procedure to find the two roots of a quadratic equation

```
procedure Quadratic(A, B, C: in Real;
      Root_1, Root_2: out Real; OK: out Boolean) is
  D: constant Real := B**2 – 4.0*A*C;
begin
  if D < 0.0 or A = 0.0 then
    OK := False; return;
  end if;
  Root_1 := (–B + Sqrt(D)) / (2.0*A);
  Root_2 := (–B – Sqrt(D)) / (2.0*A);
  OK := True;
end Quadratic;
```

If the equation has complex roots then no values are assigned to Root_1 and Root_2 so they are likely to contain rubbish. So if we call Quadratic thus

```
Quadratic(AA, BB, CC, R1, R2, State);
```

then because of the copy in and out rules for parameters of elementary types, the variables R1 and R2 which might have had respectable values will now contain rubbish.

Of course if we had made the parameters Root_1 and Root_2 of mode **in out** then the original values of R1 and R2 would have been retained if no assignments were made to Root_1 and Root_2.

However, if we had been wise and used the Default_Value aspect introduced in Ada 2012 thus

```
type Real is new Float
  with Default_Value := 0.0;
```

then the behaviour is different. In this case Root_1 and Root_2 will behave essentially as if they were of mode **in out** and will remain unchanged. Note carefully that they will not take the default values of 0.0 and so the existing values of R1 and R2 will not be disturbed. Of course if we had declared a local variable R_Temp of type Real then it would take the initial value of 0.0.

This technique of initially copying in parameters of mode **out** has existed in Ada for access types since Ada 83. Remember that access types always have a default initial value of **null** and so this copying in behaviour is identical. Incidentally, this copying in is done "in the raw" without making any subtype checks such as range constraints; again this follows the behaviour of access types. Note also that the whole purpose of an **out** parameter is to give it some value without concern for the original value of the actual parameter and so gratuitously checking the original value of the actual could be irritating if it raised an exception. Another point is that the default value applies to the type and not to the subtype.

However, do remember that we cannot give a default value to the predefined types such as Float so this is a good reason for declaring our own types.

Other problems arise when an actual parameter is a view conversion and this is the real topic of AI-74. Consider the following simple example

```
procedure Inc(X: in out Integer) is
begin
  X := X + 1;
end Inc;
...
F: Float;
...
F := 3.14;
Inc(Integer(F));
```

Remember that the behaviour is that the value of F is converted to type Integer (and thus becomes 3) and this is the initial value of the parameter X which is then incremented to 4 and finally converted to 4.0 and copied back into F. This is as in Ada 83.

But problems arise if the parameter is an **out** parameter and not an **in out** parameter. Consider

```
procedure P(X: out My_Integer) is ...
...
Y: Long_Float := 1.0E20;
...
P(My_Integer(Y));
```

Now suppose we have given Default_Value for My_Integer. An important goal of Default_Value is to ensure that junk values do not arise. This is done by treating **out** parameters essentially as **in out** parameters as illustrated by Quadratic. But now we are in trouble because we are unlikely to be able to convert the giant floating value Y to the type My_Integer.

This problem is overcome by saying that if the aspect Default_Value is given for the type of the formal parameter then there must be an ancestor of both the target type and the operand type of the view conversion and the operand type itself must also have the aspect Default_Value. If the conversion meets these requirements, it is bound to work. Otherwise, the view conversion (such as the example above) is made illegal.

AI-132 concerns expression functions and freezing again (see the brief mention of AI-103 in the previous Chapter). If we have an expression function such as

```
function F(...) return T is
  (expression of subtype T);
```

then it can occur in a renaming as body thus

```
function G( ... ) return T renames F;
```

This AI points out that this renaming freezes the expression of the expression function F.

The redefining of equality has always been a bother. Originally there were different rules for composition of tagged and untagged types. The difference was removed in Ada 2012 in order to make composition more uniform. However, a quirk in the rules meant that a hidden definition of equality for an untagged record type as in

```
package P is
  type PT is private;
private
  type PT is record ... end record;   -- untagged
  function "=" (L, R: PT) return Boolean;
end P;
```

was not permitted. This was a mistake and accordingly this restriction is removed by AI-132.

There are omissions regarding aspect specifications and derived types. One of the advantages of the introduction of aspect specifications is that they occur with the entity to which they apply. This means that the traditional linear elaboration does not always apply because the aspect might refer to things that have not yet been declared. AI-109 clarifies the situation with regard to the freezing of the representation of untagged types.

Finally, AI-97 addresses a minor error in the description of the tag of an object in a return statement. The introduction of the extended return statement where we have

```
return R: T do
   ...
end return;
```

needed clarification because T might not be identical to the return type given in the function specification (it might be a subtype; perhaps the function has an indefinite type and the return is definite, perhaps classwide and specific and so on.) So the rules were rewritten to cover the extended return. Unfortunately the rules were written in a way that was incorrect for an old-fashioned return statement. This is now put right.

Chapter 5: Tasking and real-time facilities

The major topic in this area is providing features for multiprocessors and increasing control for scheduling.

The following Ada Issues cover this area:

1	Independence and representation clauses for atomic objects
33*	Sets of CPUs when defining dispatching domains
48	Default behavior of tasks on a multiprocessor with a specified dispatching policy
51	The Priority aspect can be specified when Attach_ Handler is specified
52	Implicit objects are considered overlapping
55	All properties of a usage profile are defined by pragmas
73	Synchronous Barriers are not allowed with Ravenscar
81	Real-time aspects need to specify when they are evaluated
82	Definition of "dispatching domain"
90	Pre- and postconditions and requeues
98	Problematic examples for ATC
107	A prefixed view of a By_Protected_Procedure interface has convention protected
114	Overlapping objects designated by access parameters are not thread-safe
117	Restriction No_Tasks_Unassigned_To_CPU
129	Make protected objects more protecting

These changes can be grouped as follows.

First there are some clarifications and additions to dispatching domains which were added in 2012 (33, 48, 82). There are also some changes to the definition of the Ravenscar profile (55, 73, 117) and clarifications to some real-time aspects (51, 81). These are all in the Real-Time Systems annex (D).

The examples of the use of ATC (asynchronous transfer of control) need further explanation (98).

The question of being thread-safe in the face of overlapping objects has always been tricky and the text in the opening part of Annex A is modified (52, 114).

There are some improvements to the ability to control concurrent access in the core language (107, 129).

The required support for aspects such as Pack and their interactions with atomicity is rationalized (1). Note that this AI is a hangover from Ada 2005.

Finally, there are changes to the core language regarding pre- and postconditions and requeue (90).

As defined in Ada 2012 a dispatching domain consists of a set of processors whose CPU values are contiguous. However, this is unrealistic since CPUs are often grouped together in other ways. Accordingly, AI-33 adds a type CPU_Set and two functions to the package System.Multiprocessors.Dispatching_Domains thus

```
type CPU_Set is array (CPU range <>) of Boolean;

function Create(Set: CPU_Set) return Dispatching_Domain;
function Get_CPU_Set(Domain: Dispatching_Domain) return CPU_Set;
```

Moreover, the original functions Create and Get_Last_CPU are modified to be

```
function Create(First: CPU; Last: CPU_Range) return Dispatching_Domain;
function Get_Last_CPU(Domain: Dispatching_Domain) return CPU_Range;
```

The changes enable Last to be zero thereby allowing for null domains. Remember that the type CPU_Range has lower bound of zero whereas the subtype CPU has lower bound of one. If a domain is empty then Get_Last_CPU returns zero and Get_First_CPU returns one.

A minor editorial change is that many instances of Dispatching_Domain (which is a type name) are changed to dispatching domain (the concept) by AI-82. An important clarification concerns the behaviour in the absence of any use of CPU and dispatching domains. The summary of AI-48 says that in the absence of any setting of the CPU of a task and the creation of any dispatching domains, a partition that specifies a language-defined dispatching policy will allow all tasks to run on all processors.

With regard to Ravenscar, the whole essence of its intention is to enable the use of a very simple runtime system. Accordingly, the newly added synchronous barriers should not be allowed and so the additional restriction

```
No_Dependence => Ada.Synchronous_Barriers
```

is added to the definition of the Ravenscar profile by AI-73. Furthermore, in the case of multiprocessors, in order to permit analysis we need to ensure that all tasks (including the environment task) are assigned to a specific CPU and especially that no task is assigned zero which indicates Not_A_Specific_CPU. So a new restriction is introduced, namely

```
No_Tasks_Unassigned_To_CPU
```

and this is also added to the definition of the Ravenscar profile (AI-117). Moreover, Ravenscar requires that the CPUs are denoted statically so another restriction is introduced

```
No_Dynamic_CPU_Assignment
```

and this is also added to Ravenscar (AI-55).

There are some clarifications concerning aspects in the Real-Time annex. One is simply to say that the real-time aspects of a type are evaluated when an object of the task type is actually declared (AI-81). This applies to the aspects Priority and Interrupt_Priority and also to CPU. Remember that the aspects do not necessarily have to be static, in particular they could be discriminants of the type and different for different objects. Thus

```
task type Slave(N: CPU_Range)
  with CPU => N;

Tom: Slave(1);   Dick: Slave(2);   Harry: Slave(3);
```

It is easy to be confused regarding priorities and interrupt priorities. Typically, a protected procedure used as an interrupt handler would have aspects giving the priority and interrupt to be handled thus

```
procedure Handler
  with Attach_Handler => Some_Interrupt,
       Interrupt_Priority => Hot_Priority;
```

However, if the procedure Handler is in a protected type Monitor then the priority could be given on Monitor itself using the aspects Priority or Interrupt_Priority. If no interrupt priority or priority aspect is specified, the priority is implementation-defined but in the range of interrupt priority. (AI-51).

The mechanism for Asynchronous Transfer of Control (ATC) uses a form of select statement thus

```
select
  delay ... ;                    -- triggering alternative
  ...
then abort
  Do_Something;                  -- abortable part
end select;
```

This depends upon there being places in the abortable part that are abort completion points. The examples given in the RM in 9.7.4 rely upon this and some extra explanation is added (AI-98).

There has always been "boilerplate" in A(3) about reentrancy. The obvious example is that

```
task T1 is
begin
  Put(A_File, "Text");
end T1;

task T2 is
begin
  Put(A_File, "More Text");
end T2;
```

is not required to work (being unsafe use of a shared variable, namely A_File). But if we change T2 to

```
task T2 is
begin
  Put(B_File, "More Text");
end T2;
```

then it is required to work provided A_File and B_File are indeed truly different files. The wording in A(3) is improved to be more explicit with regard to parameter passing (AI-52, AI-114). It now becomes

> "The implementation shall ensure that each language-defined subprogram is reentrant in the sense that concurrent calls on any language-defined subprogram perform as specified, so long as all objects that are denoted by parameters that could be passed by reference or designated by parameters of an access type are nonoverlapping."

An important difference between protected functions and protected procedures (and entries) is that protected functions can be accessed concurrently. The principle is that such functions should be used for interrogating state and not to change it. However, in the case of calling functions inside a container, they do often change state. Accordingly, to enable containers to be used by parallel tasks and to impose control of access by protected objects, it is necessary to be able to make protected functions behave like protected procedures and so prevent multiple access. This can be done by a new aspect Exclusive_Functions which can be given for a protected type or a single protected object. Thus we write

```
protected type PT
  with Exclusive_Functions;
  ...
```

and then all protected functions declared within PT have exclusive access. Note carefully that the aspect is not permitted on individual protected functions but on the protected type (or object) as a whole. (AI-129).

The usual convention for a prefixed view of a subprogram is Intrinsic which means that 'Access cannot be applied. However, an awkward situation has been discovered in the case of a subprogram with aspect Synchronization being By_Protected_Procedure. (Remember that the possible values for the aspect Synchronization are By_Entry, By_Protected_Procedure, and Optional.) In that case the convention is deemed to be protected so that Access can be applied. (AI-107).

AI-1 is a hangover from Ada 2005. The essence of the problem is that the language is inconsistent regarding the pragma (now aspect) Pack. On the one hand the text of the RM regarding packing says that entities have to be squeezed up really tightly. On the other hand alignment properties, atomicity and so on ought to be respected. The revised text clarifies that Pack should not do anything that violates other requirements.

Finally, a bother with requeue is addressed by AI-90 (requeue has been the source of many bothers in the past so another one is not unexpected). This time the problem concerns pre- and postconditions. Suppose we have entries E1 and E2 with pre- and postconditions. And suppose that E1 does a requeue onto E2. The current text is unclear as to what exactly is checked and when. Do we avoid checking any postcondition on E1 and do we bypass any precondition on E2? Certainly not is the brief answer. Basically we require that the postcondition on E2 implies that on E1 by saying that they must fully conform. And moreover any precondition on E2 is indeed checked when the call is requeued. Remember that parameters are passed on unchanged and that the requeue statement does not have any explicit parameters itself.

Chapter 6: Iterators, pools, etc.

This area covers the new iterators introduced in Ada 2012 plus access types and storage pools but also various miscellaneous features.

The following Ada Issues cover this area:

3	Specifying the standard storage pool
27	Access values should never designate unaliased components
36	The actual for an untagged formal derived type cannot be tagged
38	Shared_Passive package restrictions
43	Details of the storage pool used when Storage_Size is specified
46	Enforcing legality for anonymous access components in record aggregates
47	Generalized iterators and discriminant-dependent components
67	Accessibility of explicitly aliased parameters of procedures and entries
70	9.3(2) does not work for anonymous access types
72	Missing rules for Discard_Names aspect
76	Variable state in pure packages
85	Missing aspect cases for Remote_Types
89	Accessibility rules need to take into account that a generic function is not a function
93	Iterator with indefinite cursor
120	Legality and exceptions of generalized loop iteration
124	Add Object'Image
136	Language-defined packages and aspect Default_Storage_Pool
138	Iterators of formal derived types
142	Bad subpool implementations
145	Pool_of_Subpool returns null when called too early
148	Dangling references
151	Meaning of subtype_indication in array component iterators

These changes can be grouped as follows.

A number of issues concern default and standard storage pools in general (3, 43, 136) and some issues concern the newly introduced subpools (142, 145, 148).

Several issues concern clarifications and omissions regarding generalized iterators (47, 93, 120, 138, 151).

As ever there are issues regarding accessibility rules, anonymous access types and related topics (27, 46, 67, 70, 89).

There are some clarifications and omissions about package state such as Pure and Shared_Passive (38, 76, 85).

Finally, there are miscellaneous issues on derived types (36), Discard_Names (72), and Object'Image (124).

Remember that when we declare an access type we can specify which storage pool it is to use. If we do not specify one then the default is used. Originally this default was just the "standard pool". The pragma Default_Storage_Pool was introduced in Ada 2012. It enables the user to specify which pool is to be used by default if none is specified for the access type. Thus we might write

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

Moreover, the parameter can be null thus

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

which ensures that we must always specify the pool to be used and prevents any allocation by default. AI-3 enables us to go back to the standard pool by writing

```
pragma Default_Storage_Pool(Standard);
```

This additional argument means that there is a minor syntax change thus

```
storage_pool_indicator ::=
  storage_pool_name | null | Standard
```

Note that the indicator Standard has nothing to do with the package Standard as such.

AI-43 makes subtle changes to the behaviour of the aspect Storage_Size as applied to storage pools. Briefly, the pool used by an access type that has Storage_Size given must not allocate additional storage when the original amount is exhausted and no other type can use the same pool unless requested. So we might have

```
type T is access ...
for T'Storage_Size use 1000;

type S is access ...
for S'Storage_Pool use T'Storage_Pool;  -- share pools
```

Note that if we do give the aspect Storage_Size for a type then that implies the (implementation-defined) storage pool for the type and so we cannot also give the aspect Storage_Pool for that type. Contrariwise if we do give the pool explicitly by for example

```
for T'Storage_Pool use My_Pool;
```

then the storage size is determined by the behaviour of My_Pool and the aspect Storage_Size cannot be given explicitly (the writer of the pool will have had to declare a function Storage_Size as part of the implementation of the pool and that will act as the attribute.)

AI-136 concerns the use of default storage pools with language defined generic units. After some discussion it is concluded that the effect of specifying the aspect Default_Storage_Pool on an instance of a language-defined generic unit is implementation-defined. One consequence of this is that one cannot rely upon using the aspect Default_Storage_Pool to change the storage pool used by a container such as a linked list if the container is an instance of the language-defined container Doubly_Linked_List.

Three AIs concern subpools which were introduced in Ada 2012 and clarify a number of omissions. AI-142 simply says that Allocate_From_Subpool could be erroneous if not implemented in accordance with the given rules. AI-145 says that the function Pool_Of_Subpool returns **null** if called before calling the procedure Set_Pool_Of_Subpool (pretty obvious). AI-148 tidies up the loose wording regarding what happens when we deallocate subpools (all objects that were in them cease to exist of course so beware dangling references as usual).

There are some omissions regarding iterators which were added in Ada 2012. AI-138 concerns the inheritance of aspects such as Constant_Indexing and Iterator_Element. Remember that a type such as List in Doubly_Linked_Lists has aspects thus

```
type List is tagged private
  with Constant_Indexing => Constant_Reference,
         Variable_Indexing => Reference,
         Default_Iterator => Iterate,
         Iterator_Element => Element_Type;
```

If we derive a type from List then we cannot change Iterator_Element into something other than Element_Type. We say that these aspects are non-overridable (they could be confirmed).

The other AIs in this group (47, 93, 120, 151) concern the new generalized iterators and address a number of curious omissions.

It might be recalled that if we have an array of type T thus

```
type ATT is array (1 .. N) of T;
The_Array: ATT;
```

then rather than express iteration as

```
for I in The_Array'Range loop
  The_Array(I) := 99;    -- do something to The_Array(I)
end loop;
```

we can more briefly use **of** rather than **in** and write

```
for E of The_Array loop
  E := 99;               -- do something to component E
end loop;
```

Optionally we can give the subtype of E thus

```
for E: T of The_Array loop ...
```

AI-151 says that any subtype given must statically match that of the component of the array (obvious really). Adding T is essentially a comment to aid the reader but the kindly compiler checks that it is correct.

The other AIs of this group essentially come down to the same thing. Generalized iteration enables us to write something in a shorthand way. When the shorthand is expanded, what is done using the resulting long form must not be illegal. For example AI-AI-47 shows how we might appear to make the array object vanish, AI-93 says that exceptions might be raised and 120 covers problems with limitedness and constantness.

Access types and particularly the anonymous access types introduced in Ada 2005 are often a source of problems. AI-27 clarifies the behaviour of value conversions of composite objects. AI-67 clarifies the accessibility of explicitly aliased parameters. AI-46 addresses the issue of the legality of record components of anonymous access types. 70 covers issues of the master of tasks created by anonymous access types.

AI-89 is more interesting. It suggests that Ada programmers should carefully remember the golden rules "a generic function is not a function", "a generic procedure is not a procedure", and "a generic package is not a package". The details of the AI are a bit elusive but revolve around the above rules.

Another group of issues concern matters such as the state of packages. AI-76 shows how an apparently pure package could seem to have its state changed via tricks such as using a self-referential type. Such trickery is deemed erroneous.

AI-85 notes that we cannot permit giving the aspects Storage_Size or Storage_Pool for remote access to class wide types which are given in a package with the aspect Remote_Types.

AI-38 concerns packages that are Shared_Passive. Various rules concerning the misuse of access types are strengthened.

A curious error in the matching rules for generic parameters has long been overlooked and is corrected by AI-36. If a formal parameter of a generic unit is derived untagged, then a corresponding actual parameter must also be untagged. Thus if we have

```
generic
  type T is private;
  type TT is new T;
package P ...
```

then we cannot instantiate P with a tagged type for TT. This has been wrong ever since Ada 95.

The pragma Discard_Names was introduced in Ada 95. It tells the compiler to throw away tedious tables of names at runtime associated with things such as Image and Value. AI-72 points out that Ada 2012 forgot to say that Discard_Names is now an aspect and can be given as such. So if we have an enumeration type with lots of long identifiers such as

```
type Greek is (alpha, beta, gamma, ... , omega);
```

then rather than separately giving

```
pragma Discard_Names(Greek);
```

we can add the aspect when the type is declared thus

```
type Greek is (alpha, beta, gamma, ... , omega)
  with Discard_Names;
```

Finally, AI-124 proudly announces the extension of the attribute Image to apply to objects as well as to types.

At the moment if a slovenly programmer wants to avoid the majesty of the full might of Integer_Text_IO to print out the value of N of some integer type such as My_Nice_Integer_Type (perhaps for diagnostic purposes) then they write

```
Put(My_Nice_Integer_Type'Image(N));
```

And now thanks to AI-124, this becomes

```
Put(N'Image);
```

Note that GNAT users have been writing N'Img for a long time.

Chapter 7: Predefined library

The main improvements in the standard library in Ada 2012 concern containers and these are addressed in the next section. There were also other additions such as UTF encoding packages, extensions to directories and the package Locales.

The following Ada Issues cover this area:

28*	Import of variadic C functions
30	Formal derived types and stream attribute availability
31	All_Calls_Remote and indirect calls
34	Remote stream attribute calls
37*	New types in Ada.Locales cannot be converted to/from strings
88	UTF_Encoding.Conversions and overlong characters on input
102	Stream_IO.File_Type has Preelaborable_Initialization
106	Write'Class attribute
121	Stream-oriented aspects
130	All I/O packages should have Flush
135	Enumeration types should be eligible for convention C
146	Should say stream-oriented attribute

These changes can be grouped as follows.

A number of issues concern streams, their aspects and attributes (30, 34, 102, 106, 121, 146).

Two issues concern interfacing to the C language (28, 135).

Minor issues concern I/O packages and Flush (130), the aspect All_Calls_Remote (31), and UTF_Encoding (88).

Finally, there is a major revision to the new package Ada.Locales (37).

AI-121 points out that the stream-oriented attributes such as 'Read and 'Write can be given using an aspect specification as well as by an attribute definition clause. Thus for a type Date we can declare a procedure Date_Write and associate them using

```
for Date'Write use Date_Write;
```

Alternatively, we can declare Date as

```
type Date is record ... end record
  with Write => Date_Write, ... ;
```

and then declare the procedure Date_Write.

AI-146 just corrects a bit of wording; strictly we always talk about stream-oriented attributes and not stream attributes. AI-30 clarifies the use of stream-oriented attributes with untagged formal derived types but ironically refers to them as stream attributes. AI-34 concerns the use of streams and Remote_Types packages; in summary, dereferencing a remote access-to-classwide type to make a dispatching call to a stream-oriented attribute such as 'Write is not allowed.

AI-106 clarifies the way in which a class wide stream-oriented aspect is given. For example

```
type My_Type is abstract tagged null record
  with Write'Class => My_Write;
```

AI-102 adds the pragma Preelaborable_Initialization to the type File_Type in the package Ada.Streams.Stream_IO. It points out that the package was made preelaborable in Ada 2012 so that it was more useful but the corresponding change to the private type File_Type was forgotten.

Two issues concern interfacing to C. AI-28 discusses the import of variadic C functions (that is functions with a variable number of parameters). In Ada 95, it was expected that such functions would use the same calling conventions as normal C functions; however, that is not true for some targets today. Accordingly, this AI adds additional conventions to describe variadic C functions so that the Ada compiler can compile the correct calling sequence. The other issue (135) concerns enumeration types and makes them eligible for convention C provided certain range conditions are satisfied.

With regard to input-output in general AI-130 adds the procedure Flush to the packages Sequential_IO and Direct_IO so that any internal buffers can be flushed in the same way as in Text_IO and Stream_IO.

A rather specialized change is made by AI-31 concerning the aspect All_Calls_Remote. It states that the aspect applies to all indirect or dispatching remote subprogram calls to the RCI (remote call interface unit) as well as to direct calls from outside. All indirect or dispatching calls should go through the PCS (partition communication subsystem).

A number of packages for UTF encoding were added in Ada 2012. AI-88 addresses a minor issue regarding overlong characters on input, that is as a parameter to a function Encode or Convert. It confirms that such overlong encodings do not raise Encoding_Error.

Finally, AI-37 discusses a curious difficulty found in attempting to use the seemingly innocuous new package Ada.Locales. It was mentioned in the Ada 2012 Rationale which we repeat.

The types Language_Code and Country_Code were originally declared as

```
type Language_Code is array (1 .. 3) of Character
          range 'a' .. 'z';

type Country_Code is array (1 .. 2) of Character
          range 'A' .. 'Z';
```

The problem is that a value of these types is not a string and cannot easily be converted into a string because of the range constraints and so cannot be a simple parameter of a subprogram such as Put. If LC is of type Language_Code then we have to write something tedious such as

```
Put(LC(1));  Put(LC(2));  Put(LC(3));
```

Accordingly, these types are changed so that they are derived from the type String and the constraints on the letters are then imposed by dynamic predicates. So we have

```
type Language_Code is new String(1 .. 3)
  with Dynamic_Predicate =>
    (for all E of Language_Code => E in 'a' .. 'z';
```

with a similar construction for Country_Code.

Readers might like to continue to contemplate whether this is an excellent illustration of some of the new features of Ada 2012 or simply an illustration of static strong or maybe string typing going astray.

Chapter 8: Containers

The container library was considerably enhanced in Ada 2012. A few issues have arisen since.

The following Ada Issues cover this area:

35	Accessibility checks for indefinite elements of containers
69	Inconsistency in Tree container definition
78	Definition of node for tree container is confusing
110	Tampering checks are performed first

These changes can be grouped as follows.

AI-69 and AI-78 both address the same issue regarding the fact that the root node of a tree has no element.

AI-35 concerns problems with accessibility checks necessary to prevent dangling references when using the indefinite containers.

AI-110 addresses the question of when tampering checks are performed.

It is fundamental to the organization of trees that each node of the tree has an associated element containing a value except the root node which has no such associated element. Both AI-69 and AI-78 make various corrections to the wording such as to point out that an iterator never visits the root node.

AI-35 addresses the question of accessibility checks when manipulating indefinite containers (these containers were introduced in Ada 2005). Certain operations of instances of the indefinite container packages require accessibility checks to prevent dangling references. The term "perform indefinite insertion" is defined and then this is used in the description of the various operations. Thus in the case of Indefinite_Doubly_Linked_Lists we are told that Append, Insert, Prepend, and Replace_Element that have a parameter of the Element_Type perform indefinite insertion.

AI-110 concerns the order of making various checks. The conclusion is that tampering checks are always performed before any other checks such as that for capacity.

Chapter 9: Conclusions

A number of presentation AIs (56, 80, 134, 159) which cover mostly trivial typos have not been discussed. One amusing example will suffice. Paragraph 10 of clause A.18.25 on bounded multiway trees says that the function Copy is declared as

```
function Copy(Source: Tree; Capacity: Count_Type := 0)
  return List;
```

Clearly List is an alternative spelling for Tree!

Finally, I need to thank all those who have helped in the preparation of this document and especially Randy Brukardt, Jeff Cousins and Joyce Tokar.

References

[1] ISO/IEC 8652:2012(E), *Information technology — Programming languages — Ada*. *(Online versions can be found at http://www.adaic.org/ada-resources/standards/ada12/.)*

[2] S. T. Taft et al (eds) (2012) *Ada 2012 Reference Manual*, LNCS 8339, Springer-Verlag.

[3] John Barnes (2013) *Ada 2012 Rationale*, LNCS 8338, Springer-Verlag. *(Online versions can be found at http://www.ada-auth.org/standards/rationale12.html.)*

[4] John Barnes (2014) *Programming in Ada 2012*, Cambridge University Press.

[5] ISO/IEC 8652:2012/Cor 1:2016(E), *Information technology — Programming languages — Ada — Technical Corrigendum 1*. *(An unofficial version can be found at http://www.ada-auth.org/corrigendum1-12.html. A version that consolidates TC1 with the Ada Reference Manual can be found at http://www.ada-auth.org/standards/ada12_w_tc1.html.)*

Index

Entries in this index reference chapter numbers and not page numbers.