Towards Ada 2012: An Interim Report

Edmond Schonberg

AdaCore Inc.
schonberg@gnat.com

Abstract. The Ada Rapporteur Group (ARG), following the instructions of ISO/IEC JTC1/SC22/WG9 is preparing an update to the Ada 2005 standard. This paper presents a snapshot of the more important language enhancements under discussion. Even though these enhancements are not yet in their final form, and will not become part of the proposed new standard until approved by ISO, the description that follows is an accurate reflection of the main directions in which the language is evolving. However, the names of packages, subprograms, and formal parameters, as well as some details of the syntax might change from what is presented here.

1 Introduction

The WG9 committee, after discussions with the ARG and with members of the Ada community, has instructed the ARG to complete the Amendment to Ada 2005 [1] so that ISO standardization of the new version can be completed by 2012. This is a relatively short horizon. but it matches the interval between previous releases, demonstrates that the language continues to evolve, and at the same time places a bound on the changes to the language, and ensures that they do not present an undue implementation burden on existing compilers.

This paper is an informal survey of the more important enhancements that the ARG is discussing. These enhancement are grouped as follows:

- Section 2 discusses enhancements directly related to correctness, namely the introduction of more powerful assertion mechanisms in the language: pre- and postconditions, global assertions, type invariants, are other mechanisms that encourage the programmer to better specify the meaning of the code they write, and allow the run-time to verify that this meaning is in fact obeyed.
- Section 3 discusses enhancements to the Containers library.
- Section 4 presents language enhancements that contribute to expressiveness and readability: conditional expressions, case expressions, more powerful membership tests, and corresponding iterator forms. Most of these are syntactic enhancements whose semantics is intuitive and fit well in Ada. One addition in this category has a larger import because it reverses an early design decision that had been controversial ever since Ada 83: functions will now have in and in out formal parameters.
- Section 5 discusses visibility mechanisms: more powerful use clauses, and integrated packages that provide better access to declarations in nested packages.

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 238–250, 2010.

- Section 6 presents concurrency and real-time enhancements that address the multicore revolution.
- Section 7 mentions other minor syntactic enhancements that simplify the programming task and polish some awkward corners of the language.

Each one of the enhancements we describe corresponds to one or more Ada Issues (AIs). We must emphasize that our descriptions are informal, and reflect the state of affairs as of this writing (March 2010). Please refer to the database at the Ada Information Clearinghouse (see *http://www.ada-auth.org/AI-SUMMARY.HTML*) where the interested reader will find up-to-date descriptions and a full list of Amendment AIs.

2 Program Correctness

The enhancements in this area address the familiar issue of "programming by contract" (see [2] for a modern discussion). They provide the programmer with tools to specify formally the intent of a construct. This formal description can then be verified/enforced at execution time, or even confirmed statically by analysis tools. Such contract specifications allow compilers and other tools to catch errors in usage or implementation earlier in the development cycle. They also provide valuable documentation of the intended semantics of an abstraction.

2.1 Aspect Specifications (AI05-0183)

Assertions about the behavior of subprograms, types, and objects are *aspects* of the corresponding entities, and a uniform syntax will be available to specify these, as well as more familiar operational and representational attributes of various kinds of entities. Thus the notion of aspect generalizes the familiar Ada concept of attribute. The properties of these attributes can be specified with representation clauses (for example size representation clauses) or with pragmas (for example pragma Pack). These specifications are unified with the new notion of *aspect*:

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

An aspect specification can appear in object definitions, type declarations, all manner of subprogram declarations, component declarations and entry declarations.

2.2 Pre- and Postconditions for Subprograms (AI05-0145)

These aspects are predicates, i.e. boolean expressions that must be True on entry to (resp. on exit from) the subprogram to which they apply. The declaration of the function Pop for the canonical Stack data type might be as follows:

```
function Pop(S : in out Stack) return Elem
   with
     Pre  => not Is_Empty(S),
     Post => not Is_Full(S);
```

A postcondition often needs to refer to the value of an in out parameter or of a global object before the subprogram is executed. The attribute 'Old, applied to any nonlimited entity, denotes its value on entry to the subprogram. Similarly, the attribute 'Result denotes the value returned by a function.

2.3 Type Invariants (AI05-0146)

From a contractual point of view, the visible behavior of a private type is described indirectly through the pre- and postconditions that apply to the primitive operations of the type. Other contractual details of an abstraction are better described directly as properties of the type itself.

The new aspect notation allows us to write:

```
type T (...) is private
   with Invariant => Is_Valid(T);

type T2 (...) is abstract tagged private
   with Invariant'Class => Is_Valid(T2);

function Is_Valid (X : T) return Boolean;
```

Note that Is_Valid is referenced before its declaration. This may seem like a break from the Ada canonical linear order of elaboration, but in fact it corresponds to the rule that aspects are elaborated at the point the entity to which they apply is frozen (this is the point at which all the characteristics of the type must be known). In most cases this means the end of the enclosing library package declaration.

An invariant can be type-specific, or class-wide, in which case it applies to all extensions of the type. An invariant is checked on exit from any visible subprogram that is a primitive operation of the type.

2.4 Global In/Out Annotations (AI05-00186)

In order to more fully specify the semantics of a program, it is necessary to indicate what effect subprograms may have on the environment, that is to say enclosing scopes and library-level entities. Global in/out annotations indicate what objects global to a given subprogram S are read or modified by an execution of S. The aspects in, out and in out can specify the following:

<object_name> -- the named object, and all of its subcomponents
null -- the empty set (the subprogram is pure)
others -- the universal set (all globals may be affected)
others in pkg -- all objects declared or whose designated type is declared in pkg

There are additional annotations to indicate that all objects designated by a particular access type, or by access types defined in the current package, may be accessed or modified. Annotations for generic formal parameters are also available. This rather heavy machinery is indispensable if we want to specify to the compiler the possible effect of a subprogram call, without having to rely on global program analysis tools.

3 Containers

Container libraries have become ubiquitous in modern programming environments. The enhancements in this area provide abstractions with better storage properties, task-safety, and useful search properties. There is one common advantage of standardized containers that is worth emphasizing: memory management of collections of values is handled by the container operations, not directly by the user. Storage allocation and reclamation are behind the scene, thus freeing the programmer from some of the more delicate and error-prone aspects of low-level programming. This is particularly important when indefinite types, such as class-wide types, are involved.

3.1 Bounded Containers (AI05-0001)

In their more general form containers place objects and auxiliary data structures in the heap. Even though most heap management is hidden from the programmer, thanks to the use of controlled types, such heap usage is forbidden in high-integrity environments, which renders 2005 containers virtually useless in this realm of application. Ada 2012 introduces bounded variants of containers (vectors, lists, maps) that have a fixed capacity and so can be stack-allocated. The new container types are all discriminated types, constrained by capacity. The bounded containers are not themselves controlled types, which allows for a lighter implementation. (Of course, if the element type of a bounded container is controlled, the container itself will have to be finalized.)

This AI also adds to the Ada library several general purpose packages for case-insensitive operations on strings for sorting and for hashing.

3.2 Holder Containers (AI05-0069)

In Ada 2005 it is not possible to declare a variable of an indefinite type without giving it an initial value that fixes its constraints once and for all. The holder container is a wrapper that can hold a single value of some (possibly indefinite) type. This value can be queried and modified, thus providing the equivalent of a variable of an indefinite type.

3.3 Synchronized Queues (AI05-0159)

Queues were omitted from the 2005 Container library, because they were considered trivial to write, and too elementary to be included in a language standard. However, queues that are task-safe are somewhat more complex, and it is worthwhile to standardize an efficient version of such shared data-structures. Ada 2012 introduces a synchronized interface Queue, declared in package Ada.Container.Synchronized_-Queues, and several generic packages that implement that interface:

Bounded_Synchronized_Queues
Unbounded_Synchronized_Queues
Bounded_Priority_Queues
Unbounded_Priority_Queues!

The flexibility of Synchronized interfaces (which can be implemented by tasks or by protected types) is put to good use here: each of these packages is parameterized by an instantiation of Synchronized_Queues. For example:

```
with Ada.Containers.Synchronized_Queues;
generic
  with package Queues is new
    Ada.Containers.Synchronized_Queues (<>);
package Ada.Containers.Unbounded_Synchronized_Queues is
   pragma Preelaborate;

   type Queue is synchronized new Queues.Queue with private;
private
       -- not specified by the language
end Ada.Containers.Unbounded_Synchronized_Queues;
```

3.4 Multiway Trees (AI05-0136)

Trees are the quintessential dynamic data structures, and ones for which hiding storage management activities in the implementation is particularly worthwhile. The Container library will now include a very general tree structure, a multiway tree, where each internal node has a vector of descendant nodes, so that there is easy navigation from a node to its siblings and to its ancestors. Search and insertion operations on this structure must have a complexity of O (Log (N)).

4 Functions, Expressions, Control Structures

The enhancements in this group aim to simplify programming in the small: more expressive function declarations, new expression forms, better notation for existing constructs. Most of these can be considered syntactic sugar, that is to say shortcuts to common program fragments that can be written in today's Ada. The first enhancement in this group, however, has a deeper semantic impact.

4.1 In Out Parameters for Functions (AI05-0143)

Ever since Ada 83, functions have had only in parameters, with the justification that they were intended to be the equivalent of mathematical (pure) functions with no side effects. However functions can modify global variables, and thus have arbitrary side effects for which there is no syntactic indication. In Ada 2012, functions will have both out and in out parameters, to indicate more explicitly the way in which a function call may affect the state of the program.

4.2 Dangerous Order Dependences (AI05-0144)

This in turn highlights a weakness in the way Ada specifies (or fails to specify) the order of evaluation of expressions and parameters in calls. If functions have in out parameters, there is a greater danger that side effects make the evaluation of an expression non-deterministic. To alleviate the problem, Ada 2012 mandates static

checks that make many common order-dependences illegal. For example, if F is a function with an in out parameter, the expression:

F (Obj) + G (Obj)

has an illegal order-dependence because the result may be different depending on the order in which the operands of this expression are evaluated. Similarly, the new rules force a compiler to reject aliasing between two actual parameters of an elementary type, when one of the formals is not an in parameter. The checks mandated by AI05-0144 can be made linear in the size of the expression (call or assignment). These checks depend on a static definition of when two names denote the same object, or when one name denotes a prefix of another name. Unlike more rigorous verification systems such as Spark [3], the checks proposed by this AI cannot be complete, given that arbitrary side effects may be present through global variables; they do nevertheless eliminate the most egregious examples of order-dependences. Ada will remain free of idioms that rely on a particular order of evaluation, such as the celebrated C idiom for copying strings:

```
(while *p++ = *q++);
```

4.3 Conditional and Case Expressions (AI05-0147 and AI05-0188)

The chief purpose of these syntactic shortcuts (familiar from other programming languages, such as C++ and various functional languages) is to simplify writing pre- and postconditions, as well as type invariants. These are often complex predicates which would have to be written as off-line functions, thus making them more opaque. Conditional and case expressions allow these predicates to be directly attached to the declaration of the entity to which they apply:

```
procedure Append (V : Vector; To : Vector)
  with Pre =>
    (if Size (V) > 0 then
        Capacity (To) > Size (V) else True);
```

It is frequently the case that predicates impose a check in one case but not in the other, so the trailing else True can be omitted in that case:

```
procedure Append (V : Vector; To : Vector)
  with Pre =>
    (if Size (V) > 0 then Capacity (To) > Size (V));
```

Conditional expressions can also be useful to simplify existing code involving if statements, though here tastes may differ. For example, there might be a definite advantage in rewriting an if statement if both of its branches control two subprogram calls that differ only in one actual parameter, e.g.:

```
Eval (X + Y, F (if Cond then 1 else 0));
```

instead of

```
if Cond then
   Eval (X + Y, F (1));
else
   Eval (X + Y, F (0));
end if;
```

The semantics of conditional expressions is identical to that of short-circuit expressions. Conditional expressions are static if the condition and both dependent expressions are static.

Case expressions stand in the same relation to case statements as conditional expressions to if statements. The well-understood advantage of case expressions is that the compiler can verify that all cases are covered. Thus a case expression is safer than a conditional expression with a series of tests.

4.4 Iterators (AI05-0139)

Traversing a collection is an extremely common programming activity. If the collection is described by one of the library containers, iteration over it can be described by means of the primitive operations First and Next. These operations typically use cursors to provide access to elements in the collection. However, it is often clearer to refer directly to the elements of the collection, without the indirection implied by the presence of the cursor. This AI will make it possible to write, for example:

```
for  Cursor in Iterate (My_Container) loop
   My_Container (Cursor) := My_Container (Cursor) + 1;
end loop;
```

as well as:

```
for Element of My_Container loop
   Element := Element + 1;
end loop;
```

This syntactic extension is obtained by means of a predefined interface Basic_Iterator, a function with special syntax that provides the equivalent of indexing a container with a cursor, and an implicit "dereference" operation that retrieves an element of the container when presented with a reference to such an element:

```
generic
   type Cursor is private;
   No_Element : in Cursor;
package Ada.Iterator_Interfaces is

  type Basic_Iterator is limited interface;
  function First (Object : Basic_Iterator) return Cursor;
  function Next (Object: Basic_Iterator;
                 Position: Cursor) return Cursor;
  type Reversible_Iterator is
     limited interface and Basic_Iterator;
  function Last (Object : Reversible_Iterator)
```

```
     return Cursor;
  function Previous (Object  : Reversible_Iterator;
                     Position: Cursor) return Cursor;
end Ada.Iterator_Interfaces;
```

An instantiation of this package is present in every predefined container, but the user can instantiate such a generic and provide special-purpose First and Next functions to perform partial iterations and iterations in whatever order is convenient. The familiar keyword **reverse** can be used to determine the direction of iteration.

4.5 Extended Membership Operations (AI05-0158)

The current machinery to define subtypes has no provision for declaring a subset of the values of a scalar type that is not contiguous. Membership in such a subset must be expressed as a series of tests:

```
type Color is (Red, Green, Blue, Cyan,
               Magenta, Yellow, Black);
Hue : Color;
   ...
if Hue = Red or else Hue = Blue or else Hue = Yellow then ...
```

The proposed membership notation allows sequences of values to appear as the right operand:

```
if Hue in (Red |  Blue | Yellow ) then ...
```

Once this notation is introduced, it can be extended to any non-limited type:

```
if Name in ("Entry" | "Exit" | Dict("Urgence") | then ...
```

4.6 Quantified Expressions (AI05-0176)

Invariants declared over containers are often expressed as predicates over all the elements of the container. The familiar notation of Set Theory provides the model for introducing quantified expressions into Ada:

Quantified_Expression ::= Quantifier Iterator "|" Predicate

Quantifier ::= **for all** | **for some**
Iterator ::= defining_identifier **in** expression
Predicate ::= Boolean_expression

For example, a postcondition on a sorting routine might be written as:

```
for all J in A'First .. Index'Pred (A'Last) |
             A (J) <= A (Index'Succ (J));
```

We have departed from the standard notation and rejected the use of *exists* as a new keyword, because it is in common use in existing software. Instead, the non-reserved

word **some**, appearing after keyword **for**, specifies that the expression is existentially quantified. For example, the predicate Is_Composite applied to some positive integer might be (inefficiently) described thus:

```
(for some J in 2 .. N /2 | N mod J = 0)
```

The iterator forms proposed in AI05-0139 will also be usable in quantified expressions.

5 Visibility

The AIs in this category try to simplify the programming task by providing simpler ways for names to denote specific entities, and by allowing wider uses for entities of certain kinds.

5.1 Use All Type (AI05-0150)

The Ada community has been divided over the use of use clauses ever since Ada 83. Certain style guides forbid use clauses altogether, which forces programmers to qualify all names imported from another unit. To lighten this rather heavy burden, which among other things forces the use of the awkward notation P."+", the use type clause introduced in Ada 95 provides use-visibility to the operators of a type defined in another unit, so that infix notation (X + Y) is legal even when the type of X and Y is not use-visible. AI05-0150 extends this visibility to all primitive operations of a type (including the literals of an enumeration type). If the type is tagged, this is extended as well to subprograms that operate on T'Class.

5.2 Issues of Nested Instantiations (AI05-0135 and Others)

It is common for a library package P to contain an instantiation of some other package Inner, in order to export a type T declared within Inner. This is often done by means of a derived type DT, which inherits the operations of T and makes them available to a client of P. However, this derivation is a programming artifact (in the vernacular, a kludge) and it is desirable to find a more direct way of re-exporting the entities declared in an inner package. A related issue is that of private instantiations: a package declares a private type PT and needs to declare a container of objects of this type. The instantiation cannot appear in the same package before the full declaration for PT, which leads to a contorted architecture. The ARG is examining several proposals to simplify these programming patterns, including integrated packages (whose contents are immediately visible in the enclosing package) and formal incomplete types for generic units.

5.3 Incomplete Types Completed by Partial Views (AI05-0162)

In many situations we declare an incomplete type in order to provide an access type used in some other construct. The completion of the incomplete type must occur within the same declarative part. For purposes of information hiding, we may want to complete the incomplete declaration with a private type, but this is currently

forbidden by the language. This AI proposes that the completion of an incomplete type declaration may be any type declaration (except for another incomplete one). Type declarations can thus be given in three parts: an incomplete type declaration, a private type declaration, and finally a full type declaration.

5.4 Incomplete Parameter and Result Types (AI05-0151)

Limited with_clauses make it possible to describe mutually recursive types declared in separate packages, by providing incomplete views of types. If such an incomplete view is tagged, then it can be used as the formal in a subprogram declaration and even in a call, because it is known to be passed by reference. This AI extends the use of untagged incomplete types obtained through limited views, so they can be used as parameter types and result types, as long as the full view of the type is available at a point of call.

6 Concurrency and Real Time

Most programming languages are proposing new constructs to make proper use of the multicore chips that will dominate the hardware landscape of the next decade. The International Real-Time Ada Workshop has proposed a number of language extensions to simplify the programming of such architectures [4].

6.1 Affinities (AI05-0167)

The first requirement is a mechanism to describe the set of available processing cores in a chip, and to specify a mapping (partitioning) between tasks and cores. In existing operating systems, this mapping is often described as the "affinity" of the task, and the control of task affinities in multiprocessor systems is as important as the control of priorities. This is achieved by means of the following child packages of System (only the outlines are provided):

```
package System.MultiProcessors is
   Number_of_CPUs : constant Positive := <implementation-defined>;
   type CPU is range 1 .. Number_of_CPUs;
   Default_CPU : constant CPU := <implementation-defined>;
   type CPU_Set is array (CPU) of Boolean;
end System.MultiProcessors;

with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Real_Time;           use Ada.Real_Time;
package System.Multiprocessors.Dispatching_Domains is
   type Dispatching_Domain is limited private;
   System_Dispatching_Domain: constant Dispatching_Domain;
   function Create(PS: CPU_Set) return Dispatching_Domain;
   function Get_CPU_Set(AD : Dispatching_Domain) return CPU_Set;
   procedure Assign_Task (AD : in out Dispatching_Domain;
                          T  : Task_Id :=  Current_Task);
   procedure Delay_Until_And_Set_CPU
          (Delay_Until_Time : Ada.Real_Time.Time; P : CPU);
```

```
private
   type Dispatching_Domain is new CPU_Set;
   System_Dispatching_Domain : constant Dispatching_Domain :=
                                  (others => True);
end System.Multiprocessors.Dispatching_Domains;
```

Needless to say, the implementation of these operations depends on the availability of lower-level constructs in the underlying operating system. The paper by Sáez and Crespo [5] indicates that at least on GNU-Linux operating systems their implementation is relatively straightforward today.

6.2 Extending the Ravenscar Profile to Multiprocessor Systems (AI05-0171)

The Ravenscar profile [6] has been extremely successful for real-time applications, and is in wide use today. This AI proposes its extension to multi-processor systems, to facilitate the construction of deterministic and analyzable tasking programs that can be supported with a run-time system of reduced size and complexity. The proposed extensions to the Ravenscar profile depend on the partitioning facilities described above, but forbid dynamic task migration and require that a task be on the ready queue of a single processor.

6.3 Barriers (AI05-0174)

Barriers are basic synchronization primitives that were originally motivated by loop parallelism. Operating systems such as POSIX already provide barriers, where a set of tasks is made to wait until a specified number of them are ready to proceed, at which time all of them are released. The effect of a barrier can be simulated with a protected type, but only with substantial overhead and potential serialization, so a new mechanism is needed. This mechanism is provided by means of a new package:

```
package Ada.Synchronous_Barriers is
   type Barrier(Number_Waiting : Positive) is limited private;
   procedure Wait_For_Release (The_Barrier : in out Barrier;
                               Last_Released : out Boolean);
end Ada.Synchronous_Barriers;
```

When a variable of type Barrier is created with Number_Waiting = N, there are no waiting tasks and the barrier is set to block tasks. When the count reaches N, all tasks are simultaneously released and the “Last_Released” out parameter is set in an arbitrary one of the callers, which then performs cleanup actions for the whole set. Note that this is different from the Ada 2005 proxy model for a protected operation: there the task that modifies the barrier executes sequentially, in some unspecified order, the pending actions of all tasks queued on the barrier.

6.4 Requeue on Synchronized Interfaces (AI05-0030)

The introduction of synchronized interfaces is one of the most attractive innovations of Ada 2005: a concurrent construct may be implemented by means of an active entity (a task) or a passive one (a protected object), both of which may include queues to

enforce mutual exclusion. However, only functions and procedures are allowed as primitive operations of interfaces. It is desirable to support the construction of concurrent algorithms that involve requeue statements, where the construct on which the requeue is to take place may be either a task or a protected object. This AI proposes a pragma to indicate that a given interface operation may allow requeuing, as the following example demonstrates:

```
type Server is synchronized interface;
procedure Q(S : in out Server; X : Item);
pragma Implemented (Q, By_Entry);
```

The pragma can also take the parameters By_Protected_Procedure and By_Any.

7 Syntactic Frills

The AIs in this category address small programming irritants in the syntax of the language, and simplify the programming of common idioms.

7.1 Labels Count as Statements (AI05-0179)

One of the most common uses of gotos in Ada is to provide the equivalent of a "continue" statement in a loop, namely to skip part of the body of the loop and start the next iteration. The following pattern will be familiar:

```
loop
   ...
   if Cond then
      goto Continue;
   end if;
   ...
   <<Continue>> null;
end loop;
```

The null statement is only noise, forced by the current rule that a label is part of a statement. The rule proposed in this AI is that a label by itself is a valid constituent of a statement sequence. This simple rule was chosen, instead of the more contentious introduction of a new reserved word **continue** to be used as a new statement form.

7.2 Pragmas Instead of Null (AI05-0163)

Programmers have found the current rules for pragma placement confusing and error-prone. The syntactic rules concerning them have been simplified, so that they can appear in an otherwise empty sequence of statements, without requiring the presence of an explicit null statement to make the sequence legal.

8 Conclusions

The Ada 2012 amendment strikes a balance between conflicting requirements:

- On the one hand, the evolution of software engineering suggests new language features to facilitate the construction of ever-more-complex systems.
- On the other hand, the large established software base mandates that all new constructs be upward compatible, easy to describe and relatively easy to implement.
- Finally, the Ada community expects the language to evolve, reflecting the development of software methodologies and the evolution of other languages in the same domain.

Time will tell how well the proposed amendment navigates between these constraints. Some partial implementations of the new features are appearing in existing compilers, which will allow language enthusiasts to experiment with them early on. We trust that the Ada community will welcome the new face of the language.

Acknowledgements

This paper summarizes the collective work of the ARG. Particular thanks are due to John Barnes and Tullio Vardanega, for many helpful suggestions for improvements. Remaining errors and omissions are the author's sole responsibility.

References

[1] Tucker Taft, S., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P.: Ada Reference Manual. LNCS, vol. 4348. Springer, Heidelberg (2006)
[2] Fowler, M.: UML distilled, 3rd edn. Pearson Education, Boston (2004)
[3] Barnes, J.: High Integrity Software. In: The SPARK approach to Safety and Security, Pearson Education, Boston (2003)
[4] Special issue of Ada Letters on the proceedings of IRTAW-14, Portovenere, Italy (2009) (in press)
[5] Sáez, S., Crespo, A.: Preliminary Support of Ada2012 in GNU/Linux systems: Ada-Europe 2010. LNCS (2010) (these proceedings)
[6] Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in High Integrity Systems. Ada Letters XXIV(2), 1–74 (2004)