GNAT and Ada 2005

An introduction to Ada 2005

Javier Miranda and Edmond Schonberg

January 2005

Table of Contents

1 Introduction **1**

2 Visibility Issues **3**

2.1 Limited with clause (AI-217) 3
2.2 Subprograms within private compilation units (AI-220) 3
2.3 Private with clause (AI-262) 4

3 Access-Type Issues **5**

3.1 Generalized use of anonymous access types (AI-230) 5
3.2 Access to constant parameters and null-excluding access subtypes (AI-231) 6
3.3 Resolution of 'Access (AI-235) 6
3.4 Anonymous access to subprogram types (AI-254) 7

4 Aggregates Issues **9**

4.1 Aggregates for limited types (AI-287) 9

5 Abstract Subprogram Issues **11**

5.1 Abstract non-dispatching operations (AI-310) 11

6 Real-Time and High-Integrity Issues **13**

6.1 New pragma and additional restriction identifiers for real-time systems (AI-305) 13
6.2 Ravenscar profile for high-integrity systems (AI-249) 13

7 Object Oriented Programming Issues **15**

7.1 Object.Operation notation 15
7.2 Abstract Interfaces to provide multiple inheritance (AI-251) 16

8 Interfacing with other languages **19**

8.1 Unchecked unions. Variant records with no run-time discriminant (AI-216) 19

9 Summary **21**

9.1 Acknowledgments 22

Appendix A GNU Free Documentation License **23**

Index **29**

1 Introduction

As part of the ongoing standardization activities for Ada, the language is reviewed periodically to see if corrections and/or new features are warranted. Such a review is currently in progress. A revision in the form of official amendments to the Ada 95 standard is scheduled for release for the current year (2005), and has thus come to be known informally as *"Ada 2005."* This process is carried out under the auspices of the *International Organization for Standardization* (ISO), more specifically by the *Ada Rapporteur Group* (ARG), a technical committee from ISO's Working Group for Ada (WG9) that includes Ada compiler implementors, users, and other language experts.

Over the years since the previous standard, the ARG and WG9 have been working on two major documents: *Corrigendum 2000* that completes the definition of Ada 95, and it is currently implemented by most Ada compilers, and *Corrigendum 200Y*, a working document that contains the new Ada 2005 issues. New issues are brought to the ARG by users, implementors, and language designers. Each Ada Issue (AI) is given a unique code (i.e. AI-231) and classified as: a) *Confirmation*, the ARM is correct and clear, b) *Ramification*, the ARM is correct but obscure, c) *No action*, irrelevant or too far afield, d) *Binding interpretation*, the text of the ARM is incorrect and must be modified, and finally e) *Amendment*, which means "new language feature". This is the set of AI's that is of greatest interest to users and implementors, as it defines the "new look" and greater expressiveness of the next version of the language.

The GNAT Development Team participates actively in the activities of the ARG, and has been prototyping the implementation of the most substantial AI's, to make sure that they do not present major implementation hurdles or upheavals in the architecture of the compiler.

The implementation of these new language features is currenty available to users of GNAT PRO, under a switch (-gnat05) that controls the acceptability of language extensions (note that these extensions are not part of the current definition of the language, and cannot be used by programs that intend to be strictly Ada95-conformant). These features are also available in the GNAT compiler that is distributed under the *GNAT Academic Program* (GAP), an AdaCore initiative that has three major objectives:

1. Encourage and prolong the use of Ada in Academia by providing quality-assured software packages, amongst other material, that facilitate Ada programming for students
2. Create a collaborative platform for the Ada academic community where they will be able to find help and support in various areas (technology, advocacy, teaching materials, etc.) and contribute their own ideas.
3. Create stronger links between academia and the professional Ada community. We hope that the early availablity of the Ada 2005 features to the academic community will stimulate experimentation with the new language, and spread the use of Ada as a teaching and research vehicle.

2 Visibility Issues

2.1 Limited with clause (AI-217)

Ada 95 allows mutually recursive types to be declared only if they are all declared within the same library unit, which is clearly undesirable for software engineering purposes. This is probably the most glaring source of frustration for programmers who are used to the free use of incomplete references in Java and C++. Ada 2005 allows mutual recursion among types declared in separate packages by means of a new kind of with-clause: the **limited with** clause For example, the following Ada 2005 packages provide two mutually recursive types that have components that are references to each other:

```
limited with Q;                  limited with P;
package P is                     package Q is
   type Acc_T2 is access Q.T2;     type Acc_T1 is access P.T1;
   type T1 is record                type T2 is record
     Ref : Acc_T2;                      Ref : Acc_T1;
     ...                                ...
   end record;                        end record;
end P;                             end Q;
```

There are two key features in this new construct: 1) It introduces no semantic dependence on the named packages (and hence no elaboration dependence, thus leaving the compiler to choose the order of elaboration), and 2) It provides visibility of **only** the names of the packages and the packages nested within them, and an incomplete view of all types in the packages and nested packages (such incomplete views imply the usual restrictions on incomplete types). Thus, cyclic chains of with-clauses are allowed, so long as the chain is broken by at least one limited-with-clause, and no elaboration circularities are created.

2.2 Subprograms within private compilation units (AI-220)

This issue, classified as Binding Interpretation, fixes a gap found in Ada 95 related to a *with_clause* that denotes a descendant of a private package. Consider the following scenario:

```
package A is
   ...
end A;

private package A.B is
   ...
end A.B;

package A.B.C is
   ...
end A.B.C;

with A.B.C;        -- Not legal in Ada 2005
procedure A.X is ...
```

Because the library subprogram *A.X* has no separate declaration, in Ada 95 it is unclear whether the with_clause of descendant *C* of its private sibling *A.B* is allowed or not (see the current wording of the ARM, Section 10.1.2(8)). In Ada 2005 this is clearly considered illegal because a public declaration must never depend on a private unit.

2.3 Private with clause (AI-262)

Ada 95 provides private packages to organize the implementation of a subsystem, but unfortunately these packages cannot be referenced in the private part of a public package. This forces the programmer to move declarations to the private part of some common ancestor, which complicates the organization of the subsystem. For example, in the following code, because at position –4– the programmer needs to make use of *Internal_Type* (declared at –2–) Ada 95 forces the programmer to move this declaration to the private part of their common parent (that is, at position –1–).

```
package Parent is
   ...
private
   ...                                    -- 1
end Parent;

private package Parent.Private_Child is
   type Internal_Type is ...             -- 2
   ...
end Parent.Private_Child;

private                                  -- Ada2005
        with Parent.Private_Child;        -- 3
package Parent.Public_Child is
   ...
private
   -- Need to use Internal_Type
   ...                                    -- 4
end Parent.Public_Child;
```

Ada 2005 extends the with-clause with the optional qualifier *"private"*. A library unit mentioned in a private with-clause cannot be referenced in the public part of a compilation unit; it and its contents are only available in the private part of the compilation unit. Following with our example, in Ada 2005 we can leave the declaration of the type in the private package and add a private with-clause (see –3–).

Private with-clauses can be combined with limited with-clauses to give full support to mutually recursive types (**limited private with** clause).

3 Access-Type Issues

3.1 Generalized use of anonymous access types (AI-230)

The strict and safe type model of Ada forces the programmer to add numerous explicit conversions when manipulating related access types. However, modern object oriented languages make good use of the fact that types that are references to a derived tagged type T can safely be implicitly converted to types that are references to some ancestor of T. Consider the following Ada 95 example, which declares a root tagged type, two derived types, and an access to any descendant of the root tagged type. Note the number of explicit conversions that are forced on the programmer because of Ada 95 rules.

```
--  Ada 95 example

type Root   is tagged record . . .
type D1     is new Root with . . .
type D2     is new Root with . . .

type Root_Ref is access all Root'Class;

Table : array (1 .. 2) of Root_Ref
   := (Root_Ref (new D1),  -- Explicit conversion
       Root_Ref (new D2)); -- Explicit conversion

type My_Rec is record
  Component : Root_Ref := Root_Ref (new D1);
                                -- Explicit conversion
end record;
```

Ada 2005 extends the usage of the *"access_definition"* syntactic category (see ARM, Section 3.10(6)). In addition to formal parameters and discriminants of limited types, this category is now permitted in 1) component definitions, 2) discriminants of non-limited types, and 3) object renaming declarations. The type associated is an anonymous access type, which permits implicit conversions from other access types with appropriately compatible designated subtypes (as defined by the ARM, Section 4.6(13-17)). As an example, the previous code can be written as follows in Ada 2005:

```
-- Ada 2005 example

type Root   is tagged record . . .
type D1     is new Root with . . .
type D2     is new Root with . . .

Table : array (1 .. 2) of access Root'Class
  := (new D1, new D2); -- Implicit conversions

type My_Rec is record
   Component : access Root'Class := new D1;
                            -- Implicit conversion
end record;
```

In addition, this new Ada 2005 issue also helps minimize the "named access type proliferation." This occurs when, for one reason or another, an access type is not defined close to the point of declaration of the designated type, and thus users of the designated type end up declaring their own access types, creating yet more need for unnecessary conversions.

3.2 Access to constant parameters and null-excluding access subtypes (AI-231)

Ada 95 does not give support to anonymous access types that reference constant objects (neither in subprogram formals nor in discriminants). However, there are several circumstances where they are appropriate. For example: 1) As a controlling parameter of an operation that does not modify the designated object; 2) As a way to force pass-by-reference when interfacing with a foreign language, when the external operation does not update the designated object, and 3) As a way to provide read-only access via a discriminant.

For this purpose, Ada 2005 permits the programmer to specify anonymous access-to-constant objects. In addition, Ada 2005 also allows the programmer to specify whether the **null** value is allowed in anonymous access types. Ada 95 disallows *"null"* for access parameters and access discriminants, and that behavior is not desirable in all cases, specially when interfacing with a foreign language.

```
function LowerCase
  (Name : not null access constant String)
  return String;
```

This new construct can be combined with AI-230, and thus can also be used in the declaration of array and record components.

3.3 Resolution of 'Access (AI-235)

In Ada 95 the expected type of 'Access and 'Uncheck\-ed_Access must be "a single access type". This means that the type has to be determinable from context using only the fact that it is an access type (cf. ARM, Section 3.10.2(2a)). Therefore, if the prefix of 'Access is overloaded, the programmer is forced to add extra code to help the compiler to resolve the call. For example:

```
package P is
  procedure Proc (X : access Integer);
  procedure Proc (X : access Float);
end P;

with P;
procedure AI_235 is
  Value  : aliased Integer := 10;

  type Int_Ptr is access all Integer; -- Ada 95
begin
  P.Proc (Int_Ptr'(Value'Access));    -- Ada 95
  P.Proc (Value'Access);              -- Ada 2005
end;
```

In Ada 95 the last call is surprisingly illegal: it is ambiguous because the prefix of the access attribute is not used to resolve the call, and the context does not impose a single access type. To work around the problem in Ada 95, a named access type *Int_Ptr* must be introduced. However, this is not quite equivalent to the use of an anonymous access, because accessibility rules types are different for anonymous and named access types. (The use of the qualified expression changes the accessibility check for the Access attribute from the anonymous type to the named type.) Thus, the user will have to declare an access type in the scope of each call, or will have to change to using 'Unchecked_Access.

Code like this is common in interfacing with other languages where in-out parameters are not allowed (for example, C). Ada 2005 fixes the problem and allows the use of the prefix of the 'Access and 'Unchecked_Access attributes to resolve the access reference.

3.4 Anonymous access to subprogram types (AI-254)

Ada 2005 introduces anonymous access-to-subprogram types to simplify the use of access to subprograms, and in particular to allow an access to a subprogram at any accessibility level to be passed as an actual parameter to another subprogram. Consider the following classical example of an integration function:

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

The actual subprogram corresponding to an anonymous access to subprogram parameter must be an access value designating a subprogram which is subtype conformant to the formal profile. We want to be able to call *Integrate* from within any scope, using subprograms declared at any other level. Thus we might have:

```
function Double (X: Float) return Float is ...

-- Use a local function
... := Integrate (Double'Access, 2.0, 3.0);

-- Use one of the functions available in the
-- package Ada.Numerics.Elementary_Functions
... := Integrate (Sqrt'Access, 3.0, 6.0);
```

Ada 95 named access types would prevent such uses because of accessibility considerations.

The only operations permitted on a formal access to subprogram parameter are a) to call it, b) to pass it as an actual parameter to another subprogram call, and c) to rename it. This simple semantics avoids many implementation problems. Default parameters are permitted for access to subprogram parameters with the usual semantics for parameters of mode *in*. Protected subprograms are also supported, but in this case the access definition must include the word **protected**. Anonymous access to subprograms can also be combined with AI-230 and AI-231. For example we can write:

```
type T_Table is array (1 .. 2) of not null
        access function (X : Float) return Float;
```

```
Table : T_Table := (Double'Access, Sqrt'Access);
```

4 Aggregates Issues

4.1 Aggregates for limited types (AI-287)

One important benefit of Ada over other programming languages is that it allows the programmer to initialize *all* the components of a composite object in a single statement. This feature is especially important in case of type extensions, where the initialization of some components is inherited from an ancestor. Limited types constitute an orthogonal feature that allow programmers to express the idea that *"copying values of this type does not make sense"*. Both language features are highly desirable for reliable code, but cannot be combined in Ada 95 because one cannot write aggregates for limited types. It is clear that initialization does not imply copying, and therefore that it should be possible to use an aggregate construct to describe the full initialization of an object of a limited type, while prohibiting assignments that do imply copying.

Ada 2005 combines both issues in an orthogonal way, allowing programmers to get the benefits of both. The box notation ("<>") is now used to denote the default initialization for a component of an aggregate, that is to say an invocation of the initialization procedure for the component type. If this corresponds to a limited component, this specifies a value that could not otherwise be written. Consider the following example that represents the implementation of some abstract data type.

```
package ADT is
  type Data is limited private;
  type T_Data_Ptr is access Data;

  function New_Data (Value : ... )
     return T_Data_Ptr;
private
  type Data is limited record
     Info      : ... ;
     Lock      : ... ; -- 1 (a protected type)
     More_Info : ... ; -- 2
  end Data;
end ADT;
```

In this package, the concurrent access to the abstract data type is protected by means of a lock implemented by means of some protected type (that is, a limited type). In Ada 95 it is not possible to write an aggregate for Data, so a value of the type must be constructed by providing individual values to the non-limited components by means of a set of individual assignments. For example:

```
-- Ada 95 version
function New_Data (Value : ... )
  return T_Data_Ptr
is
  Aux : T_Data_Ptr := new Data;
begin
  Aux.Info := ...;
  -- Lock is silently default-initialized
  return Aux;
end New_Data;
```

This code is error prone because the compiler can not detect whether some components were left uninitialized (i.e. components at –2–). In Ada 2005 the constructor can be written as follows:

```
-- Ada 2005 version
function New_Data (Value : ... )
  return T_Data_Ptr is
begin
  return new Data'
              (Info   => Value,
               Lock   => <>,  -- 3
               others => <>); -- 4
end New_Data;
```

The box at –3– specifies the default initialization of the limited component, and the box at –4– request the default initialization of all the other components. Note that the *"others $=> <>$"* notation is allowed even when the associated components are not of the same type. Its meaning is as follows: if a component has a default expression in the record type, the expression is used; otherwise, the normal default initialization for its type is used.

5 Abstract Subprogram Issues

5.1 Abstract non-dispatching operations (AI-310)

In Ada 95, declaring an abstract override for an inherited operation has the effect of "undefining" this operation. However, such an operation is still visible, and participates in overload resolution, leading to unexpected anomalies. For example, consider the following code:

```
package P is
   type U is new Float;
   function "*" (L, R : U) return U
      is abstract;                          -- 1
   function Image  (L : U) return String;

   type D_U is new U;
   function "*" (L, R : U) return D_U;       -- 2
   --  Implicit Image function declared here.
end P;

use P;
X : U := 1.0;
S : String := Image (X * X);                 -- 3
```

As explained above, the intent of –1– is to make the operation unavailable. This forces descendants of the type to declare their own non-abstract version of the operation. (see –2–). Althougth the call at –3– seems to be unambiguous, in Ada 95 the declaration at –1– is still considered a possible interpretation for overload resolution, making the call to the overloaded *Image* function ambiguous. In Ada 2005 a non-dispatching abstract operation is not a candidate interpretation in an overloaded call, so that the call at –3– is unambiguous.

6 Real-Time and High-Integrity Issues

6.1 New pragma and additional restriction identifiers for real-time systems (AI-305)

Ada 2005 introduces new restriction identifiers to define runtime behaviors for highly efficient tasking runtime systems: **No_Calendar**, forbids any semantic dependence on package *Ada.Calendar*; **No_Dy\-na\-mic_Attach\-ment**, does not allow calls to any of the operations defined in package *Ada.Interrupts*; **No_Protected_Type_Allocators**, forbids allocators for protected types or types containing protected type components; **No_Relative_Delay**, does not allow delay-relative statements; **No_Requeue_Statements**, forbids the use of requeue statements; **No_Select_State\-ments**, no select statements are allowed; **No_\-Task_Attrib\-utes_Package**, forbids any semantic dependence on package *Ada.Task_Attributes*; **No_Lo\-cal_Pro\-tected_Obj\-ects**, protected objects shall be declared only at library level; **No_Task_Termination**, all tasks are non-terminating; and finally **Simple_Barriers**, the entry barrier shall be either a static Boolean expression or a Boolean component of the enclosing protected object.

In addition, a new dynamic restriction-parameter-identifier is defined to specify the maximum number of calls that can be queued on an entry, and a new pragma to force the compiler to detect potentially blocking operations within a protected operation (pragma **Detect_Blocking**). The presence of all the above restrictions allows the compiler to generate simpler code and a smaller footprint runtime.

6.2 Ravenscar profile for high-integrity systems (AI-249)

Ada 2005 defines a pragma-based mechanism to allow applications to request use of the *Ravenscar Profile* semantics. Ravenscar defines a set of execution-time restrictions suitable for use in High-Integrity and Safety-Critical applications. The Ada 2005 Ravenscar Profile is equivalent to the following set of pragmas:

```
pragma Task_Dispatching_Policy
  (FIFO_Within_Priorities);

pragma Locking_Policy
  (Ceiling_Locking);

pragma Detect_Blocking;

pragma Restrictions
  (Max_Entry_Queue_Length => 1,
   Max_Protected_Entries  => 1,
   Max_Task_Entries       => 0,
   No_Abort_Statements,
   No_Asynchronous_Control,
   No_Calendar,
   No_Dynamic_Attachment,
   No_Dynamic_Priorities,
   No_Implicit_Heap_Allocations,
   No_Local_Protected_Objects,
   No_Protected_Type_Allocators,
   No_Relative_Delay,
   No_Requeue_Statements,
   No_Select_Statements,
   No_Task_Allocators,
   No_Task_Attributes_Package,
   No_Task_Hierarchy,
   No_Task_Termination,
   Simple_Barriers);
```

7 Object Oriented Programming Issues

7.1 Object.Operation notation

It is well known that the Ada 95 syntax for object oriented programming differs from the syntax provided by other programming languages: the code must identify (explicitly or implicitly) the package in which the operation is defined, in addition to the primary "controlling" object to which the operation is to be applied. Identifying both the package and the object is to some extent redundant, makes object-oriented programming in Ada 95 wordier than necessary, and encourages heavy use of potentially confusing use-clauses. For example, let us consider the following code:

```
package P is
  type T is tagged record
    Component : Integer;
  end record;
  function F    (X : T)       return Integer;
  function Self (X : T'Class) return T'Class;
end P;

with P;
--  use P; can simplify the notation below.
procedure Test_Ada95 is
  type Ptr_Obj is access all P.T'Class;
  Obj : P.T;
  Ptr : Ptr_Obj := new P.T;

  O_1 : P.TP'Class := P.Self (Obj);
  O_2 : Integer    := P.F (P.Self (Obj));
  O_3 : Integer    := P.Self (Obj).Component;
  O_4 : Integer    := P.F (P.Self (Ptr.All));
begin
  null;
end Test_Ada95;
```

Of course, the expanded names for F and *Self* can be replaced with direct names if the context includes a use_clause for P. However, if the operation is inherited, the use_clause must be on the package that declares the type of the object, not the original operation. It is plain that once the type of the dispatching argument is known, the location of the operation is known as well, and the burden of specifying its package of origin can be removed.

Ada 2005 incorporates the Object.Operation notation as an alternative syntax to allow an object-oriented programming model that is based on applying operations to objects, rather than selecting operations from a package and then applying them to parameters. If we rewrite the above procedure with the new notation we have the following code:

```
with P;  -- No need for a use clause!
procedure Test_Ada2005 is
  type Ptr_Obj is access all P.T'Class;
  Obj   : P.T;
```

```
     Ptr   : Ptr_Obj := new P.T;

     O_1   : P.TP'Class := Obj.Self;
     O_2   : Integer    := Obj.Self.F;
     O_3   : Integer    := Obj.Self.Component;
     O_4   : Integer    := Ptr.Self.F;
                        -- Implicit dereference!
   begin
     null;
   end Test_Ada2005;
```

This notation also provides some additional functionality. For example, consider:

```
   package P is
     type T is tagged record ...
     procedure Init (X : access T);
   end P;

   with P;
   package Q is
     type T_Ptr is access all P.T;   -- 1
   end Q;

   with Q;
   procedure Test_Ada2005 is
     Ptr : Q.T_Ptr;
   begin
     Ptr.Init;    -- 2: package P is not in the
                  --    context
   end Test_Ada2005;
```

Package P declares the tagged type T with its primitive operation *Init*; package Q declares an access to T. Although the test subprogram has only a with clause on this latter package, by means of the new Object.Operation notation it can call the primitive operation defined in P, while the package itself is not directly visible.

7.2 Abstract Interfaces to provide multiple inheritance (AI-251)

During the design of Ada 95 there was much debate on whether the language should incorporate multiple inheritance. The outcome of the debate was to support single-inheritance only. In recent years, a number of language designs have adopted a compromise between full multiple inheritance and strict single inheritance, which is to allow multiple inheritance of *specifications* but only single inheritance of *implementations.* Typically this is obtained by means of "interface" types. An interface consists solely of a set of operation specifications: the interface type has no data components and no operation implementations. The specifications may be either abstract or null by default. A type may implement multiple interfaces, but can inherit code from only one parent type. much of the power of multiple inheritance, without most of the implementation and semantic difficulties. For example:

```
  type I1 is interface;                         -- 1
```

```
procedure P (A : I1) is abstract;
procedure Q (X : I1) is null;

type I2 is interface I1;                      -- 2
procedure R (X : I2) is abstract;

type Root is tagged record ...               -- 3
type DT1 is new Root and I2 with ...          -- 4
-- DT1 must provide implementations for P and R
...

type DT2 is new DT1 with ...                  -- 5
-- Inherits all the primitives and interfaces
-- of the ancestor
```

The interface $I1$ defined at –1– has two subprograms: the abstract subprogram P and the null subprogram Q (a null procedure is introduced by AI-348 and behaves as if it has a body consisting solely of a null statement.) The interface I2 defined at –2– has the same operations of $I1$ plus operation R. At –3– we define the root of a derivation class. At –4– $DT1$ extends the root type, with the added commitment of implementing all the subprograms of interface I2. Finally, at –5– we extend DT1, thus inheriting all the primitive operations and interfaces of the ancestor.

The power of multiple inheritance consists in the ability to dispatch calls through interface subprograms, when the controlling argument is of a classwide interface type. In addition, languages providing interfaces also have a mechanism to determine at run-time whether a given object implements a particular interface. Ada 2005 extends the membership operation to interfaces, so that one can write *O in I'Class*. Let us see an example that uses both features:

```
procedure Dispatch_Call (O : I1'Class) is
begin
  if O in I2'Class then -- 1: Membership test
     R (O);             -- 2: Dispatching call
  else
     P (O);             -- 3: Dispatching call
  end if;
end Dispatch_Call;

I1'Class'Write (...)    -- 4: Dispatching call
                        --    to predefined op
```

The formal O covers all the objects that implement the interface $I1$, and hence at –3– the subprogram can safely dispatch the call to P. However, because I2 is an extension of I1, an object implementing I1 may also implement I2. Hence, at –1– we use the membership test to check at run-time if the object also implements I2 and then call subprogram R instead of P. Finally, at –4– we see that, in addition to user defined primitives, we can also dispatch calls to predefined operations (that is, *'Size*, *'Alignment*, *'Read*, *'Write*, *'Input*, *'Output*, *'Assign*, *'Adjust*, *'Finalize*, and the operator "=").

8 Interfacing with other languages

8.1 Unchecked unions. Variant records with no run-time discriminant (AI-216)

In Ada discriminated types carry explicit discriminant components, and the values of these components can be queried at runtime to verify the legality of an operation. By contrast, C unions are free unions, and carry no runtime indication of the intended type of their current contents. In order to interface to C programs, it is important to provide some means of mapping C unions into Ada. Ada 2005 introduce the pragma Unchecked_Union for this purpose. If this pragma applies to a discriminated record type, the runtime representation of this type does not carry the value of the discriminant. This makes it possible to map the C declaration:

```
struct T_Data {
   char *name;
   union {
     float f1;
     int   f2;
   };
};
```

. . . into the following Ada 2005 type:

```
type T_Data (Discr : Boolean) is
   Name : Interfaces.C.Strings.Char_Ptr;
   case Discr is
     when False =>
       F1 : Float;
     when True  =>
       F2 : Integer;
   end case;
end record;
pragma Unchecked_Union (T_Data);
```

It is clear that the use of such types can lead to erroneous execution, because discriminant checks cannot in general be applied. However, in order to preserve as much type safety as possible, AI-216 introduces the notion of *inferrable* discriminant: the discriminant of an object may be inferred from its declaration, or from some default initialization, even if not present in the run-time representation of the object, and the inferred value can be used to verify the legality of some operations on such types, thus providing some additional type safety that is completely absent from the C model. This provides safer semantics than the C union, at considerable implementation expense, as we describe below.

9 Summary

Over the last year the GNAT development team has been working on the implementation of the most important Ada 2005 issues. This paper summarizes the current status of this effort. The following list summarizes the AI's already implemented in GNAT, classified by their ARG/WG9 priority:

- **High Priority**
  - 217: Limited with clause
  - 220: Subprograms within private compilation units
  - 235: Resolution of 'Access
  - 249: Ravenscar profile for high-integrity systems
  - 251: Abstract interfaces to provide multiple inheritance
  - 252: Object.Operation notation
  - 254: Anonymous access to subprogram types
  - 305: New pragma and additional restriction identifiers for RT-Systems
  - 310: Abstract non-dispatching operations
- **Medium Priority**
  - 216: Unchecked unions: variant records with no run-time discriminant
  - 230: Generalized use of anonymous access types
  - 231: Access to constant parameters and null-ex\-cluding access subtypes
  - 262: Access to private units in the private part
  - 287: Limited aggregates

At the this point the architecture of GNAT has proven to be flexible enough not to give support to the new Ada 2005 issues while maintaining full conformance for Ada 95.

The implementation of these new language features is available to users of GNAT PRO, under a switch that controls the acceptability of language extensions (note that these extensions are not part of the current definition of the language, and cannot be used by programs that intend to be strictly Ada95-conformant). These features are also available in the GNAT compiler that is distributed under the *GNAT Academic Program* (GAP), an AdaCore initiative that has three major objectives: 1) Encourage and prolong the use of Ada in Academia by providing quality-assured software packages, amongst other material, that facilitate Ada programming for students; 2) Create a collaborative platform for the Ada academic community where they will be able to find help and support in various areas (technology, advocacy, teaching materials, etc.) and contribute their own ideas, and 3) Create stronger links between academia and the professional Ada community. We hope that the early availablity of the Ada2005 features to the academic community will stimulate experimentation with the new language, and spread the use of Ada as a teaching and research vehicle.

9.1 Acknowledgments

The GNAT compiler is the product of several hundred person-years of work, starting with the NYU team that created the first validated Ada 83 translator more than 20 years ago, and continuing today with the dedicated and enthusiastic members of AdaCore, and the myriad supportive users of GNAT whose suggestions keep improving the system. It is impractical to acknowledge all of the above by name, but we must express our very special thanks and admiration for Robert Dewar, chief architect, team leader, creator of some of the most interesting algorithms in GNAT, tireless enforcer of good programming practices, and an unsurpassable example of how to write impeccable software.

Appendix A GNU Free Documentation License

the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Heading 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Index

A

Abstract Interfaces to provide multiple inheritance 16
Abstract non-dispatching operations 11
Abstract Subprogram Issues 11
Access to constant parameters and null-excluding access subtypes 6
Access-Type Issues 5
Aggregates for limited types 9
Aggregates Issues 9
AI-216 19
AI-217 3
AI-220 3
AI-230 5
AI-231 6
AI-235 6
AI-249 13
AI-251 16
AI-252 15
AI-254 7
AI-262 4
AI-287 9
AI-305 13
AI-310 11
Anonymous access to subprogram types 7

F

Free Documentation License, GNU 23

G

Generalized use of anonymous access types 5
GNU Free Documentation License 23

I

Interfacing with other languages 19

L

License, GNU Free Documentation 23
Limited with clause 3

N

New pragma and additional restriction identifiers for real-time systems 13

O

Object Oriented Programming Issues 15

P

Private with clause 4

R

Ravenscar profile for high-integrity systems 13
Real-Time and High-Integrity Issues 13
Resolution of 'Access 6

S

Subprograms within private compilation units . . . 3

U

Unchecked unions. Variant records with no run-time discriminant 19

V

Visibility Issues 3