

GNAT: On the Road to Ada 2005

Javier Miranda
Applied Microelectronics Research Institute
University of Las Palmas de Gran Canaria
Spain
jmiranda@iuma.ulpgc.es

Edmond Schonberg
Courant Institute of Mathematical Sciences
New York University
U.S.A.
and AdaCore Technologies
schonberg@cs.nyu.edu

ABSTRACT

The GNAT Development Team is directly involved with the Ada 2005 effort, both participating in the Ada Rapporteur Group (ARG), and implementing and testing the new features proposed for the language revision. In this paper we summarize the Ada 2005 issues already implemented in the development version of GNAT, and give a brief overview of the implementation of the more complex ones. We find that the proposed language enhancements fit well into the existing compiler structure, and present no major implementation hurdles.

Categories and Subject Descriptors: D.3.0 [Programming Languages]: General—Ada, D.3.3 [Language Constructs and Features]: abstract data types, classes and objects, packages, data types and structures.

General Terms: Languages, Standardization, Reliability.

Keywords: Ada 2005, Compiler, Front-end, GNAT.

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 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* [18], 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. As of this writing, the GNAT development team has implemented the following approved Corrigendum 200Y amendments:

- **Visibility issues:** Limited with clauses (AI-217), subprograms within private units (AI-220), and private with clauses (AI-262).
- **Access-type issues:** Generalized use of anonymous access types (AI-230), access to constant parameters and null-excluding access subtypes (AI-231), resolution of *'Access* (AI-235), and anonymous access to subprogram types (AI-254).
- **Aggregates issues:** Aggregates for limited types (AI-287).
- **Abstract-subprogram issues:** Abstract non-dispatching operations (AI-310).
- **Real-time and high-integrity issues:** New pragma and additional restriction identifiers for real-time systems (AI-305), and Ravenscar profile for high-integrity systems (AI-249).
- **Object Oriented Programming issues:** abstract interfaces to provide multiple inheritance (AI-251), and the *object.operation* notation (AI-252).
- **Interfacing with other languages:** unchecked unions, that is to say variant records with no run-time discriminant (AI-216).

This paper is structured in two parts. In the first part (Section 2) we summarize the Ada 2005 issues already implemented in GNAT. In the second part (Section 3) we give

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGAda’04, November 14–18, 2004, Atlanta, Georgia, USA.
Copyright 2004 ACM 1-58113-906-3/04/0011 ...\$5.00.

an overview of the most interesting details of the GNAT implementation of these issues. We close with remarks on the amendment process and the remaining path to Ada 2005.

2. PART I: SUMMARY OF ADA 2005 ISSUES

In this tutorial section we review briefly the semantics of the important AI's that have been implemented in GNAT. A more complete discussion of the full set of AI's can be found in [17].

2.1 Visibility Issues

2.1.1 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 [7]. For example, the following Ada 2005 packages provide two mutually recursive types that have components that are references to each other:

```

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

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

2.1.2 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
begin
  ...
end A.X;

```

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 [18], Section 10.1.2(8)). In Ada 2005 this is clearly considered illegal because a public declaration must never depend on a private unit [13].

2.1.3 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* [10]. 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).

2.2 Access-Type Issues

2.2.1 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 [18], 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 [6] 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.

2.2.2 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 formal 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 [3]. 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.

2.2.3 Resolution of 'Access (AI-235)

In Ada 95 the expected type of 'Access and 'Unchecked_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. [18], 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 [12] and allows the use of the prefix of the 'Access and 'Unchecked_Access attributes to resolve the access reference.

2.2.4 Anonymous access to subprogram types (AI-254)

Ada 2005 introduces anonymous access-to-subprogram types [5] 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);
```

2.3 Aggregates Issues

2.3.1 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 [4]. 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.

2.4 Abstract-subprogram Issues

2.4.1 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;
  function Image (L : U) return String;

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

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

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–). Although 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 [2].

2.5 Real-Time and High-Integrity Issues

2.5.1 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 [8]: **No_Calendar**, forbids any semantic dependence on package *Ada.Calendar*; **No_Dynamic_Attachment**, 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_Statements**, no select statements are allowed; **No_Task_Attributes_Package**, forbids any semantic dependence on package *Ada.Task_Attributes*; **No_Local_Protected_Objects**, 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.

2.5.2 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 [11]:

```
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);
```

2.6 Object Oriented Programming Issues

2.6.1 Object.Operation notation (AI-252)

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 [9] 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.

2.6.2 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 [15, 16] 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 [1]. This model has been found to have 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 [15, 16] 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 "=").

2.7 Interfacing with other languages

2.7.1 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 [14]. 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 *inferable* 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.

3. PART II: GNAT IMPLEMENTATION

In this section we give an overview of most interesting details of the implementation of the above described Ada 2005 issues in GNAT. As usual, full details can be found in the sources of GNAT itself.

3.1 Visibility Issues

3.1.1 Limited with clause (AI-217)

The Ada Reference Manual (Section 10.1.4(1)) defines the notion of an “*environment declarative-part*” that at compile-time contains all the library-items of interest. Under this

model, the order of the library-items is such that there are no forward semantic dependences: with-clauses introduce semantic dependences that control the order, and the new limited with-clauses adds no semantic dependence but forces the implicit declaration of a “*limited view*” of a package that only includes names of packages (and nested packages) and incomplete types.

To implement this model, GNAT builds both views: the non-limited view and the limited view that includes only the information described above. Visibility analysis uses one of these views, depending on the with-clause in effect. For code-generation purposes, entities in the limited-view reference their counterparts in the non-limited view. The following example shows the limited and non-limited view of a package specification:

```
Limited View
-----
package Q is
  type T_1; -- Incomplete type declaration
  package Local is
    -- Incomplete tagged type declaration
    type T_2 is tagged;
  end Local;
end Q;

Non-limited View:
-----
package Q is
  type T_1 (D : Integer) is record
    ...
  end record;
  package Local is
    type T_2 is tagged record
      ...
    end record;
  end Local;
end Q;
```

The common usage is to use limited_with clauses in package specifications to declare mutually recursive structures, and to have normal with-clauses in the corresponding package bodies. For example:

```
limited with Q; -- 1
package P is
  type Acc_T2 is access Q.T2;
  ...
end P;

with Q; -- 2
package body P is
  Obj : Acc_T2 := new Q.T2; -- 3
  ...
end P;
```

The compilation of the package specification for P requires the limited view of Q (see -1-). However, the compilation of the package body installs the non-limited view of Q (see -2-) which is then used to generate the code that creates the object at -3-.

This implementation model is straightforward in source-based compilers such as GNAT (it may be problematic for library-based compilers). For example, the compilation of the package specification builds and installs the limited view of Q, but does not generate any object code; the compilation of the package body installs the entities of its withed packages (the context clauses found at -2-) and then loads and compiles the package specification (just before compiling the body of the package), and thus the context clauses of

its specification are re-analyzed. As a consequence, at –1– the semantic analyzer finds Q fully analyzed and its full-view installed. Because the full-view supersedes the limited-view the semantics just discards the installation of the limited view, thus giving the right interpretation to the limited with-clause.

This implementation is based on the earlier “with type” feature available in GNAT from version 3.12a onwards.

3.1.2 Subprograms withing private compilation units (AI-220)

GNAT is one of the compilers that implemented the rule related with this issue as it was originally intended in Ada 95 (not as it was written in the Reference Manual), and thus no further modification was required.

3.1.3 Private with clause (AI-262)

In case of private with-clauses found in the specification of a library-level package GNAT installs the context clauses in two stages: 1) Non-private with-clauses are installed before compiling the public part of the package, and 2) Private with-clauses are installed just before compiling the private part of the package. In case of library subprograms the private with clauses are installed after the specification of the subprogram has been analyzed (as documented in AI-262.).

Private with-clauses combine well with limited with-clauses. In case of limited-private-with clauses, GNAT builds the incomplete view of the named compilation units as described in Section 3.1.1 and installs it as described in the previous paragraph.

3.2 Access-Type Issues

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

The required modifications to the GNAT compiler to implement this AI were simple: the strictness of the GNAT semantic analyzer was relaxed to allow the use of anonymous access types in component definitions (thus covering array types and record components), discriminants of non-limited types, and object renaming declarations.

The accessibility level of the anonymous access type is determined at the time the access-definition is elaborated. For an access object that cannot be altered during its lifetime (parameter of mode IN or discriminant of a limited type), its level is determined by the accessibility level of its initial value. For a component definition or a discriminant of a non-limited type, the level is the same as that of the enclosing composite type. For renamings the level is the same as the level of the type of the renamed object. These language rules are necessary to simplify implementation and to avoid dangling references when an access object is updated while being “viewed” at a deeper level than it truly is.

Finally, we also had to modify the compiler to incorporate the following additional equality operators for the universal-access type in package *Standard*:

```
function "=" (Left, Right : Universal_Access)
  return Boolean;
function "/=" (Left, Right : Universal_Access)
  return Boolean;
```

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

The implementation of access to constant parameters was immediate: the compiler just has to remember that the designated object is not allowed to be modified, and thus cannot be used in the left-side of an expression, or as an actual for an in-out parameter.

On the other hand, the GNAT implementation of null-excluding access subtypes consists of four main parts: a) Propagation of the null-excluding attribute to subtypes, objects and components depending on a null-excluding access-type declaration, b) Addition of new checks to the front-end to statically detect bad usages of null-excluding types (for example, the assignment of the *null* value to a null-excluding object), c) Generation of the null-exclusion run-time check when required, and finally d) Relax the front-end to permit the *null* value in anonymous access types (the default semantics of Ada 95 never allows the null value in anonymous access types).

The ability to specify an access subtype that excludes null for both named and anonymous access types not only provides useful documentation but also higher efficiency. This is especially true for actuals, by allowing the null check to be moved to the point of the call, where it can be more likely removed.

3.2.3 Resolution of 'Access (AI-235)

Few modifications were done in the GNAT front-end to allow the use of the prefix of the 'Access (and 'Unchecked_Access) attributes to resolve the access type.

3.2.4 Anonymous access to subprogram types (AI-254)

In a compiler that uses static links to handle variables global to a subprogram (like GNAT), an access to subprogram value is generally represented by a pair of addresses —the address of the subprogram’s code, and the static link. However, given the Ada 2005 semantic rules, an anonymous access to a subprogram can be represented solely by its code address. This is desirable not only because it is efficient, but also because it allows the representation to easily match C’s typical representation of function pointers.

In addition, Ada 2005 rules ensure that accessibility checks are never required for anonymous access to subprograms and thus they do not need to carry an accessibility level with them (in contrast to access-to-object parameters).

3.3 Limited Aggregates Issues

In implementation terms, the initialization of limited components of aggregates required by AI-287 must be carried out in their final destination —no copying can take place. For this purpose, GNAT converts the aggregate into a set of individual assignments. In case of limited components, GNAT generates a call to the default initialization subprogram associated with the limited type.

The implementation of this AI adds no special complexity to the compiler: initializing in place is already required for controlled objects by the ARM (see [18], Section 7.6(17.1/1)), and the semantics of a box used in a component association is essentially the same as for a subtype mark used as the ancestor part of an extension aggregate. It should be mentioned that semantic analysis and expansion of aggregate is an extremely complex portion of the front-end semantics,

and the initialization of limited components adds infinitesimally to this complexity.

3.4 Abstract Non-Dispatching Issue

The implementation of abstract non-dispatching operations (AI-310) required a minor modification to the name resolution rules so that abstract non-dispatching subprograms are discarded, when analyzing an overloaded call.

3.5 Real-Time and High-Integrity Issues

The addition of restrictions and profiles (AI-305 and AI-249) add no special complexity to the compiler. If the new restrictions are specified in the source, the front-end increases its strictness and reduces the set of Ada features allowed in the application.

3.6 Object Oriented Programming Issues

3.6.1 *Object.Operation* notation (AI-252)

The implementation of the basic functionality required by this issue is simple: when the analysis of a selected component *Object.Operation (...)* fails, instead of immediately generating an error message, the front end rewrites it using the standard functional Ada notation—that is, *Operation (Object, ...)*—and repeats the analysis. The name of the operation is rewritten as an expanded name, using the scope of the object itself as a prefix. If analysis of the rewritten notation generates no error, it is correct. The use of this simple transformation also enforces the Ada 2005 semantic rule that the operations made visible by means of the new notation are hidden by components with the same identifier as the operation.

```
package P is
  type TP is tagged record
    Data : Integer := 999;
  end record;

  function Data (X : TP) return Integer;
  -- Returns some value different from 999
end P;

with P; use P;
procedure Test1_Ada2005 is
  Obj : TP;
  Value : Integer := Obj.Data;
  -- Initialized to 999: analysis
  -- succeeds without rewriting
begin
  null;
end;
```

The implementation of class-wide calls requires some more work, because the scope of the type of the object does not necessarily designate the scope of the operation: it may be declared in the scope of any of the ancestors. Consider the following example:

```
package P is
  type T is tagged ..
  procedure Proc (X : T'Class); -- 1
end P;

with P;
package Q is
  type DT is new P.T with ... -- 2
  ..
end Q;
```

```
with P, Q;
procedure Test2_Ada2005 is
  ..
  Obj : Q.DT; -- 3
begin
  Obj.Proc; -- 4
end;
```

At point –2– we define a descendant of tagged type *P.T*. At point –1– we have defined a class wide subprogram applicable to all descendants of *P.T*. At point –3– we define an object of type *Q.DT*. The call at –4– (by means of the new notation) is valid because the object is covered by –1–. The implementation of this feature requires that the front-end traverse the chain of derivations from the object type to the root of the class, to locate the unique class-wide subprogram that covers the call.

3.6.2 *Abstract Interfaces to provide multiple inheritance* (AI-251)

At present we are working on a prototype implementation of this critical issue. The model uses a combination of dispatch table for the primitive operations of the type, and permutation maps that establish how a given interface is satisfied by existing primitive operations. Although this model supports the Ada 2005 semantics, we are currently evaluating alternatives that may be more efficient at run-time, such as using an array of tags, one that points to the type dispatch table, and one that points to a separate dispatch table for each implemented interface. The trade-offs between compiler complexity, run-time efficiency, and upwards compatibility are being analyzed. Ideally, we would also like the chosen model to simplify interfacing to C++ classes (at least for the g++ compiler) but it is not clear whether this last goal is achievable at reasonable cost.

3.7 Interfacing with other languages

A simple implementation of `Unchecked_Union` has been available in GNAT for several years. However, the notion of inferrable discriminant, which is central to the semantics of this AI, complicates the full implementation considerably. Initialization, assignment, and equality are all impacted by the possible presence of such discriminants: temporaries must be created for them, and they must be used selectively in the expansion of the above operations, instead of the discriminant references that would otherwise be generated. Instead of a simple mechanism to interface to common C unions, this AI makes `Unchecked_Union` types into full-blown variant records with off-line discriminants: it is unclear whether this level of complication is justified by the gain in functionality. The implementation effort, though manageable, was roughly four times larger than expected. This episode is a reminder that grafting small semantic changes into a large compiler may have surprisingly complex consequences. We hope to report a year from now that there were very few other instances of such surprises on the road to Ada 2005.

4. ACATS FOR ADA 2005

The Ada Conformity Assessment Test Suite (ACATS) is the test suite used for Ada processor conformity testing. In addition to the implementation of the new Ada 2005 issues,

the GNAT development team has submitted 43 new tests to the ARG that help to verify Ada 2005 compilers.

5. 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-excluding 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 availability 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.

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.

6. REFERENCES

- [1] ARG. *Abstract interfaces to provide multiple inheritance*. Ada Issue 251, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00251.TXT>.
- [2] ARG. *Abstract non-dispatching operations*. Ada Issue 310, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00310.TXT>.
- [3] ARG. *Access to constant parameters and null-excluding access subtypes*. Ada Issue 231, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00231.TXT>.
- [4] ARG. *Aggregates for limited types*. Ada Issue 287, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00287.TXT>.
- [5] ARG. *Anonymous access to subprogram types*. Ada Issue 254, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00254.TXT>.
- [6] ARG. *Generalized use of anonymous access types*. Ada Issue 230, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00230.TXT>.
- [7] ARG. *Limited with clause*. Ada Issue 217, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-50217.TXT>.
- [8] ARG. *New pragma and additional restriction identifiers for real-time*. Ada Issue 305, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00305.TXT>.
- [9] ARG. *Object.Operation notation*. Ada Issue 252, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00252.TXT>.
- [10] ARG. *Private with clause*. Ada Issue 262, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00262.TXT>.
- [11] ARG. *Ravenscar profile for high-integrity systems*. Ada Issue 249, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00249.TXT>.
- [12] ARG. *Resolution of 'access*. Ada Issue 235, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00235.TXT>.
- [13] ARG. *Subprograms within private compilation units*. Ada Issue 220, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00220.TXT>.
- [14] ARG. *Unchecked Unions: Variant records with no run-time discriminant*. Ada Issue 216, <http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00216.TXT>.
- [15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (2nd edition)*. Addison-Wesley, 2000.
- [16] E. International. *C# Language Specification — Standard ECMA-334 (2nd edition)*. Standardizing Information and Communication Systems, December, 2002.
- [17] P. Leroy. An Invitation to Ada 2005. *8th Ada-Europe International Conference on Reliable Software Technologies*, LNCS 2265:1–23, June 2003.
- [18] S. Taft, R. A. Duff, and R. L. B. and Erhard Ploedereder (Eds). *Consolidated Ada Reference Manual with Technical Corrigendum 1. Language Standard and Libraries. ISO/IEC 8652:1995(E)*. Springer Verlag. ISBN: 3-540-43038-5, 2000.