The Implementation of Ada 2005 Interface Types in the GNAT Compiler

Javier Miranda[1], Edmond Schonberg[2] and Gary Dismukes[2]

[1] jmiranda@iuma.ulpgc.es
Applied Microelectronics Research Institute
University of Las Palmas de Gran Canaria
Spain

[2] {schonberg|dismukes}@adacore.com
AdaCore
104 Fifth Avenue, 15th floor
New York, NY 10011

Abstract. One of the most important object-oriented features of the new revision of the Ada Programming Language is the introduction of Abstract Interfaces to provide a form of multiple inheritance. Ada 2005 Abstract Interface Types are based on Java interfaces, and as such support inheritance of operation specifications, rather than the general complexity of inheritance of implementations as in full multiple inheritance. Real-time uses of Ada demand efficient and bounded worst-case execution time for interface calls. In addition, modern systems require mixed-language programming. This paper summarizes part of the work done by the GNAT Development Team to provide an efficient implementation of this language feature and simplifies interfacing with C++.

Keywords: Ada 2005, Abstract Interface Types, Tagged Types, GNAT.

1 Introduction

During the design of Ada 95 there was much debate about 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 [6, 8] 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. 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 full-blown multiple inheritance, without most of the implementation and semantic difficulties.

During the last year the GNAT Development Team has been working on the implementation of Ada 2005 features [10]. For the implementation of abstract interfaces, we have adopted the design policy that the implementation must be efficient and have a bounded worst-case execution time [7, Section 3.9(1.e)]. In addition, we desire an implementation that simplifies mixed-language programming, in particular when interfacing Ada with the g++ implementation of C++.

At compile time, an interface type is conceptually a special kind of *abstract tagged type* and hence do not add special complexity to the compiler (in fact, most of the current compiler support for abstract tagged types can be reused). However, at run time, additional structures must be created to support dynamic dispatching through interfaces as well as membership tests. This paper concentrates on these issues.

The paper has the following structure: In Section 2 we summarize the main features of Ada 2005 abstract interfaces. In Section 3 we give an overview of the state of the art for implementing polymorphic calls and we sketch the GNAT implementation approach. In order to understand the proposed implementation, the reader needs to be familiar with the existing run-time support for tagged types. Hence, in Section 4 we summarize the GNAT run-time support for Ada 95 tagged types. In Section 5 we describe the implementation of abstract interfaces: Section 5.1 presents two approaches to support dynamic dispatching through interfaces, Section 5.2 presents the new layout adopted by GNAT that is compatible with the C++ Application Binary Interface in order to simplify the interfacing of Ada 2005 with C++, and Section 5.3 describes the run-time support for the membership test applied to interfaces. We close with some conclusions and the bibliography.

2 Abstract Interfaces in Ada 2005

An Ada 2005 interface type consists solely of a type declaration together with a set of operation specifications: the interface type has no data components and no implementation of operations. The specifications may be either abstract or null by default. A type may implement multiple interfaces, but can inherit operation implementations from only one parent type [1]. For example:

```
package Pkg is
  type I1 is interface;                                  -- 1
  procedure P (A : I1) is abstract;
  procedure Q (X : I1) is null;

  type I2 is interface and 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
   ...
end Pkg;
```

The interface *I1* defined at –1– has two subprograms: the abstract subprogram *P* and the null subprogram *Q* (null procedures are introduced by AI-348 [2]; they behave as if their body consists solely of a *null_statement*). The interface *I2* defined at –2– has the same operations as 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– type *DT2* extends DT1, inheriting all the primitive operations and interfaces of its ancestor.

The power of multiple inheritance is realized by the ability to dispatch calls through interface subprograms, using a controlling argument of a class-wide interface type. In addition, languages providing interfaces [6, 8] also have a mechanism to determine at run time whether a given object implements a particular interface. For this purpose Ada 2005 extends the membership operation to interfaces so that the programmer can write *O in I'Class*. Let us look at an example that uses both features:

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

   I1'Write (Stream, Obj)           -- 4: Dispatching call to
                                    --    predefined operation
end Dispatch_Call;
```

The type of the formal *Obj* covers all the types that implement the interface *I*1, 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 might also implement I2. Therefore at –1– we use the membership test to check at run-time whether the object also implements I2, and then call subprogram *R* instead of *P* (applying a conversion to the descendant interface type I2). 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*, *Adjust*, *Finalize*, and the equality operator.

In the next section we briefly present the state of the art in the implementation of multiple inheritance and interfaces, and we sketch the approach followed in GNAT.

3 Implementation Strategies for Interfaces

Compiler techniques for implementing polymorphic calls can be grouped into two major categories [5]: *Static Techniques*, which involve precomputing all data structures at compile or link time and do not change those data during run time, and *Dynamic Techniques*, where some information may be precomputed at compile or link time, but which may involve updating the information and the corresponding data structures at run time. For efficiency reasons, the GNAT implementation uses only static techniques.

The static techniques for implementing polymorphic calls are: *Selector Table Indexing*, *Selective Coloring*, *Row Displacement*, *Compact Selector-Index Dispatch Tables*, and *Virtual Function Tables*. The Selector Table Indexing scheme (STI) uses a two-dimensional matrix indexed by class and selector codes (where a selector code denotes a concrete primitive operation). Both classes and selectors are represented by unique, consecutive integer encodings. Unfortunately, the resulting dispatch table is too large and very sparse, and thus this scheme is generally not implemented as described. Selective Coloring, Row Displacement, and Compact Selector-Index Dispatch Tables are variants of STI that reduce the size of the table.

The approach of Virtual Function Tables (VTBL) is the preferred mechanism for virtual function call resolution in Java and C++. The VTBL is a table containing pointers to the primitive operations of a class. Instead of assigning selector codes globally, VTBL assigns codes only within the scope of a class. In Java the implementation typically stores the VTBL in an array reachable from the class object, and searchs by name and profile for the relevant table entry at run time. Most Java compilers augment the basic search approach of the VTBL with some form of cache or move-to-front algorithm to exploit temporal locality in the table usage to reduce expected search times [3].

[3] also proposes a new interface-dispatch mechanism called the Interface Method Table. IMT is supported by Jalapeño, a virtual machine for Java servers written in Java at the IBM Research Group. The authors remark that their method is efficient in both time and space. The key idea is to convert the "Selector Index Tables" method into a hash table that assigns a fixed-sized Interface Method Table to each class that implements an interface. This approach handles collisions by means of custom-generated conflict resolution stubs (that is, subprograms with a "case" statement to determine which of the several signatures that share this slot is the desired target). These stubs are built incrementally as the program runs, and hence this technique is not considered appropriate for GNAT.

Two variants of the Virtual Function Tables (VTBL) approach have been considered for implementing dispatching calls for abstract interfaces in GNAT: 1) Permutation Maps, and 2) Multiple Dispatch Tables. In the former approach, each tagged type has one dispatch table plus one supplementary table per interface containing indices into the dispatch table; each index establishes the correspondence between the interface subprograms and the tagged type sub-

programs (permutation maps are discussed in [1]). Multiple Dispatch Tables, which are standard for C++ implementations, involves the generation of a dispatch table for each implemented interface. A dispatching call with an interface controlling argument locates the dispatch table corresponding to the interface (using an interface tag within the controlling argument), and then performs the usual indirect call through the appropriate entry in that table. Thus, dispatching a call through an interface has the same cost as any other dispatching call. We have written prototype implementations of both approaches. Although the second approach uses significantly more space for the tagged type than is required by permutation maps and adds complexity to the compiler, it has two major benefits: 1) Constant-time dispatching through interfaces, and 2) Simplified interfacing with C++ abstract classes and pure virtual functions. Because these two benefits are important for the Ada community, this latter approach has been selected for GNAT (further details are given in Section 5.1).

Concerning interface membership tests, [11] and [12] discuss several techniques that can be used to implement type-inclusion tests in constant time, independently of the number of interfaces implemented by a given type: *packed encoding*, *bit-packed encoding* and *compact encoding*. The former is the most efficient, and the latter two are more compact. Because these techniques introduce additional complexity and data structures to the run-time, we decided to evaluate their appropriateness for GNAT. For this purpose we examined the current usage of interfaces in Java, and we selected the sources available with the Java 2 Platform, Standard Edition (J2SE 5.0) [9]. Figure 1 summarizes the results: from a total of 2746 Java classes, 99.3 percent implement a maximum of four interfaces, and there is a single class (*AWTEventMulticaster*) that implements 17 interfaces. Because of these results, and also because constant-time is not really required for the interface membership test —worst-case time-cost is enough—, we have decided to implement the interface membership test in GNAT using an additional data structure: a compact table containing the tags of all the implemented interfaces (the structure of this table will be discussed in Section 5.3). Hence, the cost of the interface membership test is the cost of a search for the interface in this table, and is proportional to the number of implemented interfaces.

Number of Implemented Interfaces

0	1	2	3	4	5	6	7	8	17
22	1998	508	160	40	6	7	2	2	1

Number of Java Classes

Fig. 1. Usage of interfaces in J2SE 5.0

Before we discuss the details of the implementation of abstract interfaces in GNAT, the reader needs to be familiar with the GNAT run-time support for tagged types. This is summarized in the next section.

4 Tagged Types in GNAT

In the GNAT run-time, the *_Tag* component of an object is a pointer to a structure that, among other things, holds the *Dispatch Table* and the *Ancestors Table* (cf. Figure 2). The Dispatch Table contains the pointers to the primitive operations of the type. The Ancestors Table contains the tags of all the ancestor types; it is used to compute class-wide membership tests in constant time. For further information on the other fields, see the comments in the GNAT sources (files *a-tags.ads* and *a-tags.adb*).

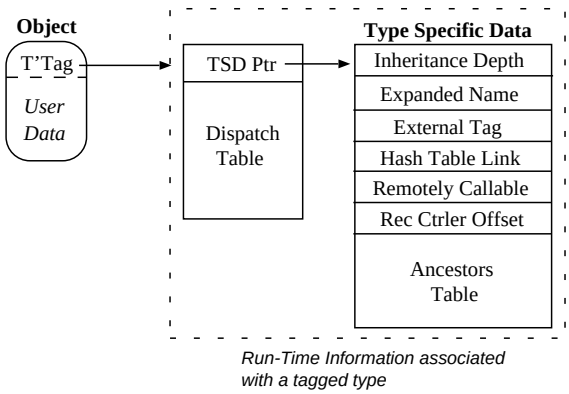

Fig. 2. Run-time data structure for tagged types

Let us briefly summarize the elaboration of this structure with the help of Figure 3. On the right side, the reader can see a tagged type T with two primitive operations P and Q. On the left side of the same figure we have a simplified version of the structure described above. For clarity, only the dispatch table, the table of ancestor tags, and the inheritance level are shown. The elaboration of a root tagged type declaration carries out the following actions: 1) Initialize the Dispatch Table with the pointers to the primitive operations, 2) Set the inheritance level *I-Depth* to one, and 3) Initialize the table of ancestor tags with the *self* tag.

For derived types, GNAT does not build the new run-time structure from scratch, but starts by copying the contents of the ancestor tables. Figure 4 extends our previous example with a derived type DT. The elaboration of the tables corresponding to DT involves the following actions: 1) Copy the contents of the dispatch table of the ancestor, 2) Fill in the contents of the new dispatch

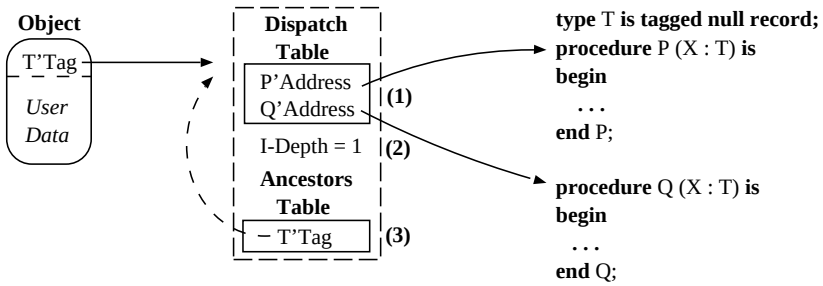

Fig. 3. Elaboration of a root tagged type

table with the pointers to the overriding subprograms (as well as any new primitive operations), 3) Increment the inheritance level to one plus the inheritance level of the ancestor, 4) Copy the contents of the ancestor tags table in a stack-like manner (that is, copy the 0 to i elements of the ancestor tags table into positions 1 to $i+1$ and save the *self* tag at position 0 of this table. Thus the *self* tag is always found at position 0, the tag of the parent is found at position 1, and so on. Knowing the level of inheritance of two types, the membership test *O in T'Class* can be computed in constant time by means of the formula: $O'Tag.Ancestors\_Table(O'Tag.Idepth - T'Tag.Idepth) = T'Tag$

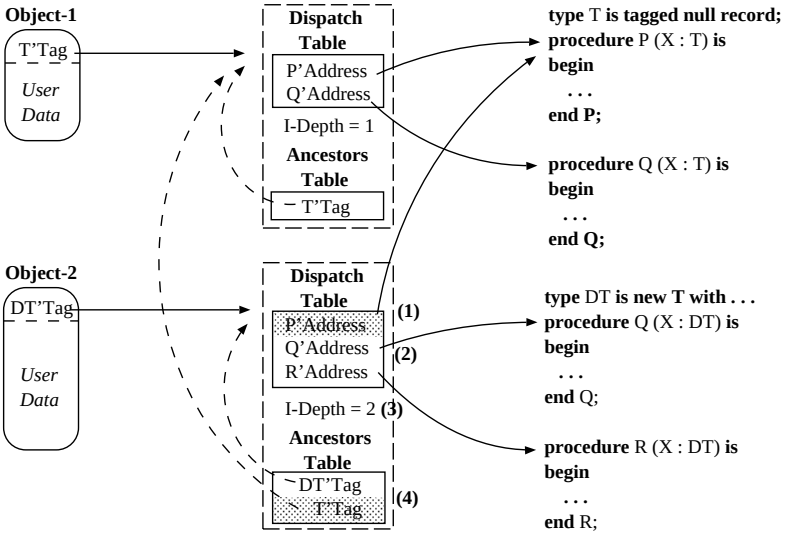

Fig. 4. Elaboration of a derived type

In addition to the user-defined primitive operations, the dispatch table contains the pointers to all the predefined operations of the tagged type (that is, *'Size*, *'Alignment*, *'Read*, *'Write*, *'Input*, *'Output*, *Adjust*, *Finalize*, and the equality operator).

5 Abstract Interfaces in GNAT

As we explained in Section 2, at run time the implementation of abstract interfaces involves support for two main features: 1) Dispatching calls through interfaces, and 2) Membership Tests applied to interfaces. In the following sections we describe the GNAT implementation of these features.

5.1 Dispatching calls through Abstract Interfaces

Two variants of the Virtual Function Tables (VTBL) approach presented in Section 3 for implementing dispatching calls through abstract interfaces were evaluated in GNAT (cf. Figure 5): 1) Permutation Maps, and 2) Multiple Dispatch Tables. In the former approach, each tagged type has one dispatch table plus, for each implemented interface, one supplementary table containing indices into the dispatch table; each index establishes the correspondence between an interface subprogram and the tagged type's implementation of that subprogram (permutation maps are discussed in [1]). The latter approach involves the generation of a dispatch table for each implemented interface. Thus, dispatching a call through an interface has the same cost as any other dispatching call.

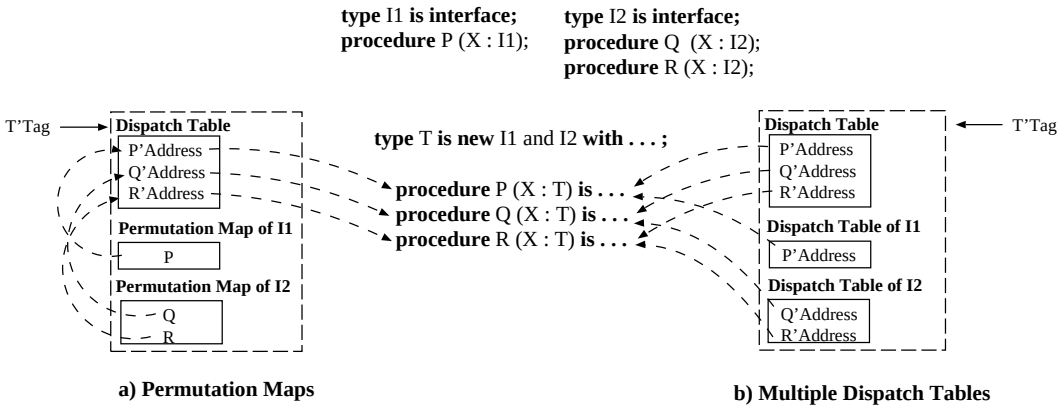

Fig. 5. Permutation Maps versus Multiple Dispatch Tables

The implementation of the permutation map approach is simpler than the implementation of multiple dispatch tables because the indices in the permuta-

tion maps never change, so they can simply be inherited directly by any descendant types. By contrast, although multiple dispatch tables require significantly more space and are more complex to implement (because the compiler must take care of generating additional code to create and elaborate these additional dispatch tables), it has two major benefits: 1) Constant-time dispatching through interfaces, and 2) Easier interfacing with C++. The first benefit is crucial for a real-time language like Ada, and the second benefit is important for the Ada community in general because it allows interfacing with C++ abstract classes and pure virtual functions. Hence, GNAT implements interfaces by means of multiple dispatch tables.

5.2 C++ ABI Layout Compatibility

In order to have true compatibility with C++ we have modified the layout of tagged objects as well as the run-time data structure associated with tagged types to follow the C++ Application Binary Interface (ABI) described in [4]. Figure 6 presents an example with the new layout: at the top of this figure we have the layout of an object of a tagged type. Compared with the previous GNAT layout, the main difference is found in the run-time structure: the dispatch table has a header containing the offset to the top and the Run-Time Type Information Pointer (RTTI). For a primary dispatch table, the first field is always set to 0 and the RTTI pointer points to the GNAT Type Specific Data structure described in Section 4. In addition, the tag of the object points to the table of pointers to primitive operations that is available after the header.

At the bottom of Figure 6 we have the layout of a derived type that implements the interfaces *I1* and *I2*. When a type implements several interfaces, its run-time data structure contains one primary dispatch table and one secondary dispatch table per interface. Regarding the layout of the object (left side of the figure), the derived object contains all the components of its immediate ancestor followed by 1) the tag of all the implemented interfaces, and 2) its additional user-defined components. Regarding the contents of the dispatch tables, the primary dispatch table is an extension of the primary dispatch table of its immediate ancestor, and thus contains direct pointers to all the primitive subprograms of the derived type. The *offset_to_top* component of the secondary tables holds the displacement to the top of the object from the object component containing the interface tag. (This offset provides a way to find the top of the object from any derived object that contains secondary virtual tables and is necessary in C++ for *dynamic_cast*.)

In the example shown in Figure 6, the offset of the tag corresponding to the interfaces I1 and I2 are m and n respectively. In addition, rather than containing direct pointers to the primitive operations associated with the interfaces, the secondary dispatch tables contain pointers to small fragments of code called *thunks*. These thunks are used to adjust the pointer to the base of the object. To better understand its behavior, we consider an example of the use of the above

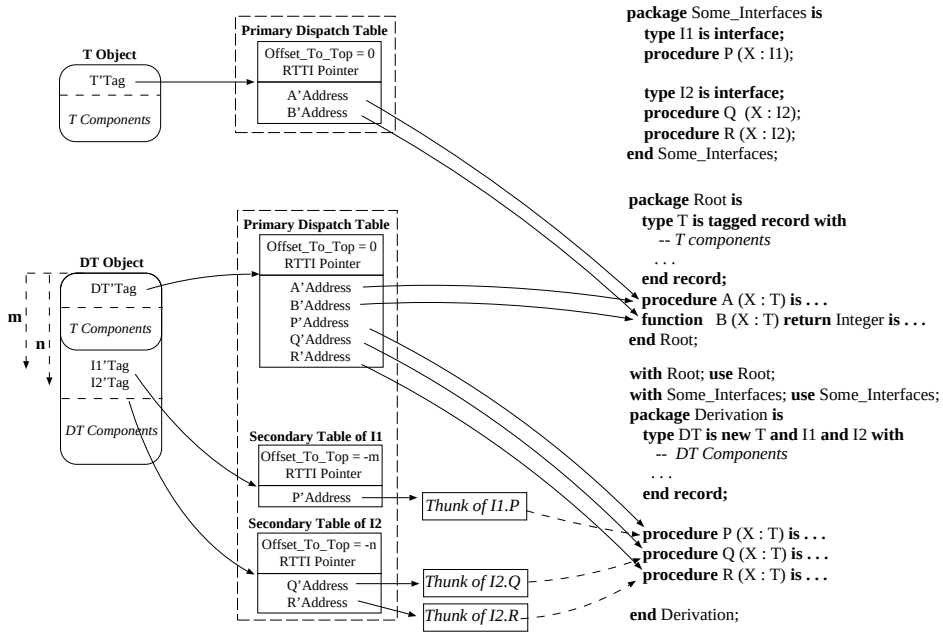

Fig. 6. Layout compatibility with C++

run-time data structure and analyze the full execution sequence of the following code that issues a dispatching call to the subprogram R of the interface I2.

```
with Derivation;        use Derivation;
with Some_Interfaces; use Some_Interfaces;
procedure Test is
   procedure Class_Wide_Call (Obj : I2'Class) is
      --  3: The pointer to the object received in the
      --     actual parameter is in fact a displaced
      --     pointer that points to the I2'Tag
      --     component or the object (see Figure 6)
   begin
      --  4: Dispatch call to the thunk through the
      --     secondary dispatch table associated with
      --     the interface I2
      R (Obj);
   end Class_Wide_Call;

   O1 : DT;                  --  1: Object declaration
begin
   Class_Wide_Call (O1); --  2: Displace the pointer to
                         --     the base of the object
                         --     by n bytes
end Test;
```

At –1– we declare an object that has the layout described in Figure 6. At –2– we have a call to a subprogram with a class-wide interface formal, and the compiler generates code that displaces the pointer to the base of the object by n bytes to point to the object component containing the I2'Tag (cf. Figure 6). This adjusted address is passed as the pointer to the actual object in the call to *Class_Wide_Call*. Inside this subprogram (at –3–), all dispatching calls through interfaces are handled as if they were normal dispatching calls. For example, because R is the second primitive operation of the interface I2, at –4– the compiler generates code that issues a call to the subprogram identified by the second entry of the primary dispatch table associated with the actual parameter. Because the actual parameter is a displaced pointer that points to the I2'Tag component of the object, we are really issuing a call through the secondary table of the object associated with the interface I2. Rather than a direct pointer to the R subprogram, the compiler has generated code that fills this entry of the interface dispatch table with the address of the thunk that 1) subtracts the m byte displacement corresponding to I2 in order to adjust the address so that it refers to the real base of the object, and 2) jumps to the primitive subprogram R.

5.3 Interface Membership Test: *O in I'Class*

In analogy with the Ada 95 membership test applied to class-wide types (described in Section 4), at run time we have a compact table containing the tags of all the implemented interfaces (cf. Figure 7). The reasons behind the selection of this simple structure were previously discussed in Section 3. The run-time cost of the membership test applied to interfaces is the cost of a search for the interface in this table.

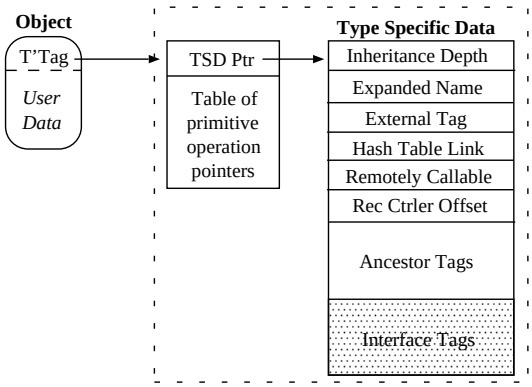

Fig. 7. The Table of Interfaces

This simple approach has the advantage that the elaboration of derived types implementing interfaces is simple and efficient. In analogy with the elaboration of the Ancestors Table (described in Section 4), we elaborate the new table of interfaces as follows: 1) Copy the contents of the table of interfaces of the immediate ancestor (because the derived type inherits all the interfaces implemented by its immediate ancestor), and 2) Add the tags of any new interfaces.

6 Conclusions

This paper summarizes part of the work done by the GNAT Development Team to implement Ada 2005 abstract interface types. Because interfaces are conceptually a special kind of *abstract tagged type*, at compile time most of the current support for abstract tagged types has been reused. At run time, additional structures were required to give support to membership tests as well as dynamic dispatching through interfaces.

We developed two prototype implementations of abstract interfaces. The first implementation uses a combination of a dispatch table for the primitive operations of a tagged type, and permutation maps that establish how a given interface is mapped onto that type's primitive operations. Although the implementation of this model was rather simple and correctly supports the Ada 2005 semantics, in order to have constant time in dispatching calls through interfaces, and also simplify the interfacing of Ada 2005 with C++ (at least for the g++ compiler), we developed an alternative prototype that is more complex and uses separate dispatch tables for all the implemented interfaces. Because of these important benefits for the Ada community, this second approach has been selected as the final version supported by GNAT.

Acknowledgments

We wish to thank Cyrille Comar and Matt Heaney for the discussions that helped us to clarify the main concepts described in this paper. We also wish to thank Arnaud Charlet, Geert Bosch, Robert Dewar, Paul Hilfinger, and Richard Kenner for helping us to clarify details of the underlying technology. Finally, we also wish to thank the dedicated and enthusiastic members of AdaCore, and the myriad supportive users of GNAT whose suggestions keep improving the system.

References

1. Ada Rapporteur Group (ARG). *Abstract interfaces to provide multiple inheritance.* Ada Issue 251, http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00251.TXT.
2. Ada Rapporteur Group (ARG). *Null procedures.* Ada Issue 348, http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00348.TXT.

3. B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, ACM Press. http://www.research.ibm.com/jalapeno/publication.html, October 2001.
4. CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI. Technical Report Revision 1.75, www.codesourcery.com.prev/cxx-abi, 2004.
5. K. Driesen. Software and Hardware Techniques for Efficient Polymorphic Calls. *University of California, Santa Barbara (PhD Dissertation)*, TRCS99-24, June 1999.
6. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (2nd edition)*. Addison-Wesley, 2000.
7. Intermetrics Inc and the MITRE Corporation. *Annotated Ada Reference Manual with Technical Corrigendum 1. Language Standard and Libraries. ISO/IEC 8652:1995(E)*. http://www.ada-auth.org/arm-files/AARM.PDF, 2000.
8. ECMA International. *C# Language Specification —Standard ECMA-334 (2nd edition)*. Standardizing Information and Communication Systems, December, 2002.
9. Sun MicroSystems. Java 2 Platform, Standard Edition (J2SE 5.0). *Available at http://java.sun.com/j2se/*, 2004.
10. J. Miranda and E. Schonberg. GNAT: On the Road to Ada 2005. *ACM SigAda 2004*, November 2004.
11. K. Palacz and J. Vitek. Java Subtype Tests in Real-Time. *Proceedings of the European Conference on Object-Oriented Programming*, http://citeseer.ist.psu.edu/660723.html, 2003.
12. J. Vitek, R.N Horspoo, and A. Krall. Efficient Type Inclusion Tests. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 97)*, ACM Press. http://citeseer.ist.psu.edu/vitek97efficient.html, 1997.