Ada 2005 Abstract Interfaces in GNAT (Extended Abstract)

Gary Dismukes[1] and Javier Miranda[2]

[1] dismukes@adacore.com
AdaCore
104 Fifth Avenue, 15th floor
New York, NY 10011

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

Abstract. One of the most salient object-oriented issues of Ada 2005 are Abstract Interfaces. Although the concept is not new (it is based on Java interfaces), being Ada a language for reliable and real-time applications its implementation must be efficient and have a bounded worst-case execution time. This paper summarizes part of the work done by the GNAT Development Team to have an efficient implementation of this language feature.

Keywords: Ada 2005, Abstract Interfaces, GNAT.

1 Introduction

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 [5, 6] 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 [2]. This model has been found to have much of the power of multiple inheritance, without most of the implementation and semantic difficulties.

During the last year the GNAT Development Team has been working in the implementation of the new Ada 2005 issues [8]. For the implementation of abstract interfaces it has been taken into account that, being Ada a language for reliable and real-time applications, its implementation must be efficient and have a bounded worst-case execution time [10, Section 3.9(1.e)]. In addition, it is also desirable for the implementation to facilitate the interface with other languages.

At compile-time, abstract interfaces are conceptually a special kind of *abstract tagged types* and hence they do not add special complexity to the compiler (most of the current compiler support for abstract tagged types can be reused). However, at run-time additional structures must be added to give support to membership tests and dynamic dispatching through interfaces, and this is the contents of this paper.

This paper is structured as follows: In Section 2 we summarize the main features of Ada 2005 abstract interfaces. In order to understand its implementation, the reader needs to be familiar with the run-time support for tagged types. Hence, in Section 3 we summarize the current GNAT run-time support for tagged types. In Section 4 we describe the implementation of abstract interfaces: Section 4.1 discusses an implementation of the membership test applied to interfaces, and Section 4.2 presents two approaches to give support to dynamic dispatching through interfaces. In Section 5 we give an overview of related work. We close with some conclusions and the bibliography.

2 Abstract Interfaces in Ada 2005

An Ada 2005 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 [2]. 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 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 (a null procedure is introduced by AI-348 [3] 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 [5, 6] 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 operation
```

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 "=").

Before we discuss 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.

3 Tagged Types in GNAT

In the GNAT run-time the *_Tag* is a pointer to an structure that, among other things, has the *Dispatch Table* and the *Ancestors Table* (cf. Figure 1). 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 in constant time the membership test involving class-wide types, that is *"X in T'Class"*. For further information on the other fields read the comments available in the GNAT sources (read files *a-tags.ads* and i-cpp.ads).

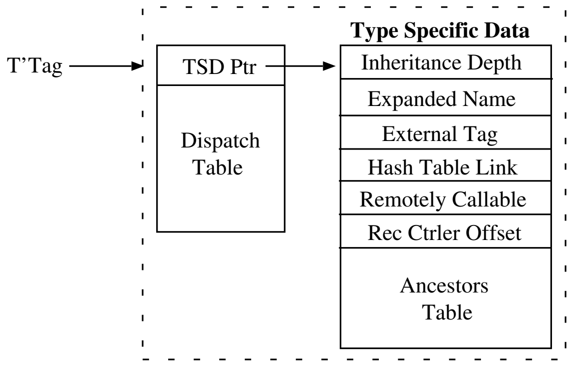

Fig. 1. Run-time structure for tagged types

Let us briefly summarize the elaboration of this structure with the help of Figure 2. In the right side the reader can see a tagged type T with two primitive operations P and Q. In the left side of the same figure we have a simplified version of the structure described above. For clarity reasons, only the dispatch table, the table of ancestor tags, and the inheritance level have been represented. The elaboration of a root tagged type 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.

In case of derived types GNAT does not build the new run-time structure from the scratch, but it copies the contents of the ancestor tables. Figure 3 completes 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) Complete the contents of the new dispatch table with the pointers to the overriding subprograms (as well as the new primitive operations), 3) Initialize the inheritance level to one plus the inheritance level of the ancestor, 4) Copy the contents of the ancestor tags table in a stack manner, that is copy the 0 to i elements of the ancestor tags table in positions 1 to $i + 1$ and save the *self* tag at position 0 of this table.

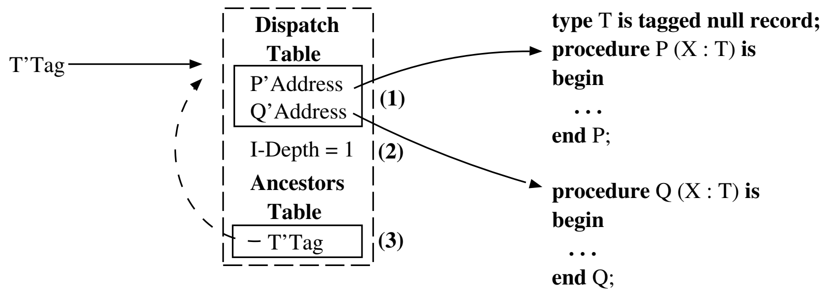

Fig. 2. Elaboration of a root tagged types

Thus the *self* tag is always found at position 0, the tag of the parent is found at position 1, and so on, and knowing the level of inheritance of two types, the membership test *O in T'Class* can be computed in constant time by the formula: $O'Tag.Ancestors\_Table(O'Tag.Idepth - T'Tag.Idepth) = T'Tag$

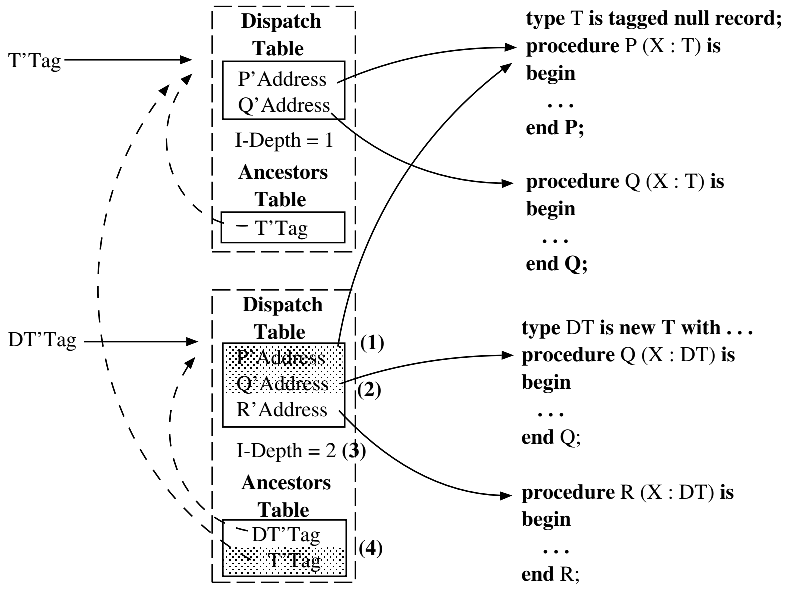

Fig. 3. Elaboration of a derived type

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*, *'Assign*, *'Adjust*, *'Finalize*, and the operator "="). From these primitives, *'Size* is the only operation that is always placed at a fixed position of the dispatch table because it is needed at run-time to compute the size of the parent.

4 Abstract Interfaces in GNAT

As we have seen in Section 2, at run-time the implementation of Interfaces involves two main parts: the implementation of the membership test applied to interfaces, and dispatching calls through interfaces. These topics are analyzed in the following sections.

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

Similar to the Ada 95 membership test applied to class-wide types (described in Section 3), at run-time we can have a compact table containing the tags of all the implemented interfaces (cf. Figure 4). However, because each type can implement many interfaces, the run-time cost of the membership test is the cost of a search for the interface in this table.

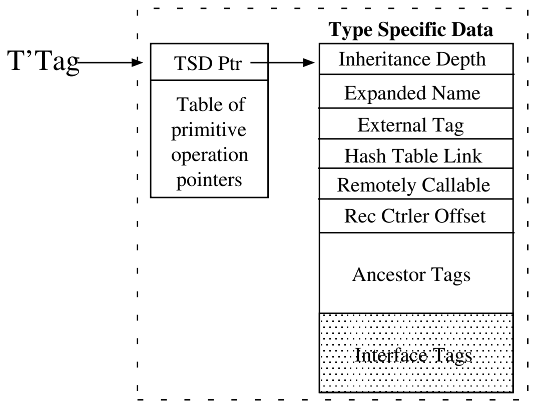

Fig. 4. The Table of Interfaces

This simple approach has the advantage that the elaboration of derived types containing interfaces is simple and efficient: similar to the elaboration of the Ancestors Table (described in Section 3) we can 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 new interfaces.

In order to evaluate if this simple approach is acceptable, we have analyzed the current usage of interfaces in Java. For this purpose we have used the sources available with the Java 2 Platform, Standard Edition (J2SE 5.0) [7]. Figure 5 summarizes the results. From a total of 2746 Java classes, 99.3 per cent implement a maximum of 4 interfaces, and there is a single class (*AWTEventMulticaster*) that implements 17 interfaces.

Number of Implemented Interfaces	0	1	2	3	4	5	6	7	8	17
Number of Java Classes	22	1998	508	160	40	6	7	2	2	1

Fig. 5. Usage of interfaces in J2SE 5.0

As a consequence this simple approach has been considered valid for GNAT. However, if the constant-time requirement is required for the Ada applications, some of the efficient type inclusion tests available in the literature [9, 11] can be implemented (these approaches have not been considered because they introduce additional complexity and data structures to the run-time —-described in Section 5).

4.2 Dispatching calls through Abstract Interfaces

Two approaches have been considered to implement dispatching calls through abstract interfaces: 1) Dispatch tables, and 2) Permutation Maps. The former approach consists in the generation of a dispatch table for each implemented interface. Thus, dispatching a call through an interface has exactly the same cost than any other dispatching call. The latter approach consists in the building of supplementary tables containing indices into the dispatch table; each index establishes the correspondence between the interface subprograms and the tagged type subprograms (permutation maps are discussed in [2]).

The major advantages of a dispatch tables by interface are: 1) Facilitate interfacing with C++, and 2) Efficiency, because at run-time we have direct pointers to the subprograms implementing the interface, and thus the indirection introduced by the permutation map is not needed. By contrast, its major disadvantage is compiler complexity because the compiler must handle the creation and elaboration of all these additional dispatch tables. However, the implementation of the permutation map approach is more simple because the indices in the permutation maps never change (and can be inherited by the derived type).

Currently we have a prototype implementation of the permutation map approach. However, because interfacing with C++ is really important for the Ada

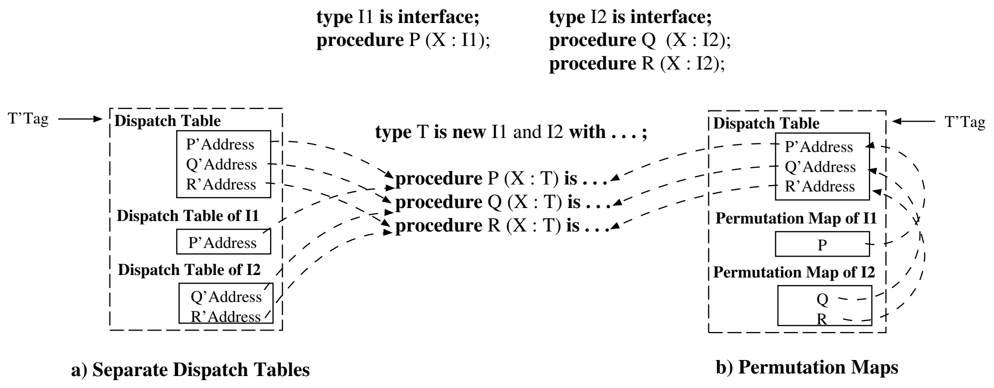

Fig. 6. Separate dispatch tables versus permutation maps

community, we are now evaluating another prototype that implements the separate tables approach to be compatible with g++.

4.3 Dispatching calls to predefined operations

Ada 2005 allows to dispatch calls to predefined operations through abstract interfaces (for example $I'Class'Input(...)$). Conceptually this introduces no additional complexity: similar to any other interface operation, the permutation map associated with each interface can contain contain the index of the right pointer to issue the indirect call.

In order to reduce the size of the tables (either permutation maps or dispatch tables) GNAT will probably fix the position of all the predefined operations in the dispatch table associated with any tagged type. This simple scheme avoids the need to duplicate the entries related with predefined primitive operations in all the interface's dispatch tables (they are always available in the dispatch table of the type).

5 Related Work

Compiler techniques for implementing polymorphic calls can be grouped in two major categories [4]: *Static Techniques*, techniques that precompute all data structures at compile or link time and do not change those data during run-time, and *Dynamic Techniques*, techniques that may precompute some information at compile or link time, but they update these information and the corresponding data structures at run-time. For efficiency reasons, for the GNAT implementation we have considered static techniques.

The static techniques for implementing polymorphic calls are: *Selector Table Indexing*, *Selective Colouring*, *Row Displacement*, *Compact Selector-indexed Dispatch Tables*, and *Virtual Function Tables.* The Selector Table Indexing scheme uses a two-dimensional matrix indexed by class and selector codes. Both classes and selectors are represented by unique, consecutive class or selector codes. Unfortunately, the resulting dispatch table is too large and very sparse, and thus this scheme is generally not valid to be implemented. Selective Colouring, Row Displacement, and Compact Selector-Index Dispatch Tables are variants of STI that reduce the size of the table. Virtual Function Tables (VTBL) are the preferred mechanism for virtual function call resolution in Java and C++. An VTBL contains is a virtual method table for a class, restricted to those methods that match a particular interface. Instead of assigning selector codes globally, VTBL assigns codes only within the scope of a class. Typically, the system stores the VTBL in an array reachable from the class object, and search for the relevant table 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 [1].

Concerning the membership test, [11] and [9] review the previous work in the field of efficient type inclusion tests and discuss several techniques that can be used to implement type inclusion tests in constant time: the *packed encoding*, the *bit-packed encoding* and the *compact encoding.* The former is the most efficient, and the latters are more compact.

6 Conclusions

This paper summarizes part of the work done by the GNAT Development Team to implement Ada 2005 abstract interfaces [2]. Because interfaces are conceptually a special kind of *abstract tagged types*, at compile-time most of the current support for abstract tagged types has been reused. At run-time it is clear that additional structures are required to give support to membership tests as well as dynamic dispatching through interfaces.

At present we already have a prototype implementation of this critical issue that 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 more efficient alternatives to simplify the interfacing with C++ (at least for the g++ compiler).

References

1. 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.

2. ARG. *Abstract interfaces to provide multiple inheritance.* Ada Issue 251, http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00251.TXT.
3. ARG. *Null procedures.* Ada Issue 348, http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00348.TXT.
4. K. Driesen. Software and Hardware Techniques for Efficient Polymorphic Calls. *University of California, Santa Barbara (Phd Dissertation)*, TRCS99-24, June 1999.
5. J. Gosling, B. Joy, G Steele, and G. Bracha. *The Java Language Specification (2nd edition).* Addison-Wesley, 2000.
6. ECMA International. *C# Language Specification —Standard ECMA-334 (2nd edition).* Standardizing Information and Communication Systems, December, 2002.
7. Sun MicroSystems. Java 2 Platform, Standard Edition (J2SE 5.0). *Available at http://java.sun.com/j2se/*, 2004.
8. J. Miranda and E. Schonberg. GNAT: On the Road to Ada 2005. *ACM SigAda'2004*, November 2004.
9. 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.
10. S.Tucker Taft, Robert A. Duff, and Randall L. Brukardt 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.
11. 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.