Heterogeneous Data Structures and Cross-Classification of Objects with Ada 95

Magnus Kempe

Swiss Federal Institute of Technology in Lausanne
Software Engineering Laboratory
EPFL–DI–LGL
CH–1015 Lausanne, Switzerland
e-mail: *Magnus.Kempe @ di.epfl.ch*

This paper was first published in the Ada-Europe '95 Conference Proceedings (Springer-Verlag).

ABSTRACT. The implementation of ADTs for homogeneous data structures has become a classic example of ADT in Ada 83. With some effort, it was also possible to implement a restricted form of heterogeneous data structures, based on variant records. We show that various approaches in implementing flexible heterogeneous data structures with Ada 95 are now possible. One of these approaches is generalized to create heterogeneous catalogues of cross-referenced objects, thus implementing one kind of multiple classification.

KEYWORDS. Abstract Data Types, Ada 95, Heterogeneous Collections, Classification, Catalogues.

1. INTRODUCTION

It has long been known how to implement various homogeneous data structures with Ada 83 (e.g. stacks of integers, or lists of characters). However, it is sometimes necessary to collect polymorphic items in one data structure, i.e. in a heterogeneous container ADT. Since each object in Ada has one specific type, the question is then how to build a collection of objects which do not all have the same specific type, while remaining within the frame of Ada's strong typing.

In other words, is it possible to create dynamic data structures with elements of various kinds in Ada? This paper demonstrates how, using Ada 83 mechanisms only [1], this could be done with some difficulty and much inflexibility, and then goes on to show several alternative and flexible approaches using new Ada 95 constructs [2].

First, we show how to use variant records, i.e. record types with discriminants.

Second, we explain how tagged types, class-wide access types, and genericity can be used.

Third, we describe an alternative design using tagged types and class-wide access types, but no genericity, to create a type-extensible framework.

Finally we develop an extremely versatile technique, based on self-referential types (with access discriminants). This technique has the additional benefit of offering a straightforward approach for cross-referencing catalogues or, in more general terms, for multiple classification.

This tour of techniques for implementing heterogeneous data structures in Ada 95 shows that the revised definition of the language provides a set of extremely powerful constructs, and that this set is more than adequate to solve many advanced design and coding problems.

2. VARIANT RECORDS

The programmer can apply a technique to implement heterogeneous data structures simply with the constructs offered by Ada 83, namely with variant records. This technique requires both the declaration of an enumeration type which lists the different kinds of items to be dealt with, and the declaration of a record type with a discriminant of the enumerated type such that each variant in the record type corresponds to the representation of one kind of item.

For instance, considering a very simple case in which one would like to implement a list mixing integers with characters, one could write the following declarations (for presentation purpose, we choose to present a simple array and ignore issues of information hiding):

```
type Kind_Type is
-- two kinds of items: integers and characters
   (K_Integer, K_Character, K_Null);
```

```
type Item_Type (Kind : Kind_Type := K_Null) is
   record          -- default: no value, no kind
      case Kind is
         when K_Integer
            => I : Integer;
         when K_Character
            => C : Character;
         when K_Null -- default variant: undefined value
            => null;
      end case;
   end record;

type Bounded_Array_Type is
   array (Positive range <>) of
      Item_Type;
```

Figure 1. *Heterogeneous array, with variant records.*

With such declarations, one could also instantiate e.g. a generic list ADT:

```
generic
   type Item_Type is private;

package Lists_G is
   type List_Type is limited private;
   procedure Insert_Front
      (X : in Item_Type;
       L : in out List_Type);
   ...
end Lists_G;

package List is
   new Lists_G (Item_Type);
```

Figure 2. *Heterogeneous list, with variant records as item type.*

Note that the generic list ADT is not, and need not be, specially designed for use as a heterogeneous data structure. It is simply the same as what used to be seen as a homogeneous list in Ada 83.

Unfortunately, this technique has several drawbacks.

First, each time one wants to add a new kind of item, one has to update the type Kind_Type and then propagate the change to the variant record type Item_Type as well as to the case statements likely to appear in the code to distinguish between kinds of items. Such modifications are not always possible or desirable. For instance, it would be better if we could avoid changing existing code, since there would be no consequent need to recompile it and everything that depends on it.

Second, the use of variant records is not necessarily space-efficient, since the compiler is likely to systematically reserve as much space as needed for the largest variant. Depending on the differences in size between variants, this may turn out to be a heavy penalty to pay in terms of memory usage.

If an access type is declared, it is possible to directly use this access type to instantiate the required (Ada83-style) ADT, with better average use of memory as a consequence (but with additional headaches due to memory management):

```
type Reference_Type is
   access Item_Type;

package List is
   new Lists_G (Reference_Type);
```

Figure 3. *Heterogeneous list, with access to variant records as item type.*

3. TAGGED TYPES AND CLASS-WIDE ACCESS TYPES

With the advent of tagged types in Ada 95, a new kind of access types has also been introduced: *class-wide access types*. A class-wide access type is associated to the root of a hierarchy of tagged types (i.e. the root of an inheritance tree), and its values may designate objects of any specific type within that hierarchy.

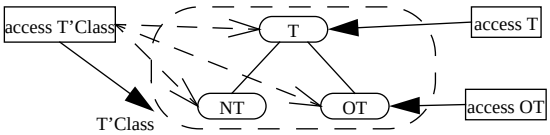

Figure 4. *"T'Class" as covering a hierarchy of specific types; contrast between "access T" and "access T'class".*

In figure 4, we show with dashed lines that, given an inheritance hierarchy with two types (NT and OT) derived from a root type T, the class-wide type T'Class covers values of all three specific types and a class-wide access type ("access T'Class") would cover values like those of specific access types (e.g. "access T").

This leads to a simple solution in order to implement heterogeneous data structures: create a hierarchy of tagged types, each one corresponding to a separate kind of items to be handled, and work with a related class-wide access type. This creates a *constrained* heterogeneous data structure, because *only* items belonging to a given class are allowed in the structure. The tag of each item corresponds exactly to the discriminant of the first example.

A combination of the examples shown in figures 1 and 2 and a translation to use tagged types and class-wide access gives:

```
type Item_Type is -- parent of all kinds of items
   abstract tagged null record;

type Integer_Type is -- first kind: integers
   new Item_Type
   with record
      I : Integer;
   end record;
```

```
type Character_Type is -- second kind: characters
   new Item_Type
   with record
      C : Character;
   end record;

type Reference_Type is -- access any kind
   access Item_Type'Class;

type Bounded_Array_Type is
   array (Positive range <>) of
      Reference_Type;
```

Figure 5. *Heterogeneous array, with class-wide access as item type.*

Again, the access type can be passed directly to the required generic list ADT. The values stored in the ADT are access values to objects within the Item_Type'Class hierarchy of types:

```
package List is
   new Lists_G (Reference_Type);
```

Figure 6. *Heterogeneous list, with class-wide access as item type.*

An obvious advantage of this technique is that it is possible to declare new types within the hierarchy without any changes to the declarations already shown. A noteworthy difference and potential advantage is that the kinds of items are not linearly enumerated but can be organized in a hierarchy of kinds, if a taxonomy is needed.

In addition, the space requirements are strictly related to the actual type of each item stored in the container. This is a direct consequence of the level of indirection introduced by the class-wide access type which is used.

Most object-oriented programming languages do in fact automatically create a level of indirection to represent objects, which explains why many also offer heterogeneous data structures by default. Ada's advantage in this context is that the programmer is *free* to decide whether he wants a homogeneous or a heterogeneous data structure, and then the language enforces that decision.[1]

It is possible to design a new breed of container ADTs, one which is specifically tuned to working with heterogeneous collections implemented in the form of hierarchies of tagged types.

We present for instance the specification of a generic heterogeneous stack ADT, which, given some tagged type T used as an actual parameter to instantiate the ADT, will store and return objects of any type in the class rooted at T (i.e. in T'Class):

1. To some, this may be seen as a drawback, since the programmer is required to decide which to use...

```
generic
   type Item_Type is
      tagged private;

package Heterogeneous_Stacks_G is
   type Stack_Type is
      private;

   procedure Push
      (X : in Item_Type'Class;
       S : in out Stack_Type);

   function Top (S : Stack_Type)
      return Item_Type'Class;

   procedure Pop
      (S : in out Stack_Type);

private
   type Reference_Type is
      access Item_Type'Class;
   ...
end Heterogeneous_Stacks_G;
```

Figure 7. *Explicitly heterogeneous stack ADT.*

4. TYPE EXTENSION FRAMEWORK

A different technique relying on tagged types is to create a framework, with a non-generic ADT exporting a tagged type that the programmer will extend to create the various kinds of item types he needs.

Here is an example specification of a singly-linked list, with just one operation profile shown, "insert in front of the list":

```
package Heterogeneous_Lists is
   type Item_Type is -- parent of all kinds of items
      abstract tagged private;

   type List_Type is
      limited private;

   procedure Insert_Front
      (X : in Item_Type'Class;
       L : in out List_Type);
   ...
private
   type List_Type is
      access Item_Type'Class;

   type Item_Type is
      abstract tagged
      record
         Next : List_Type;
      end record;
end Heterogeneous_Lists;
```

Figure 8. *Explicitly heterogeneous singly-linked list ADT.*

This technique is quite unusual for someone who is accustomed to working with generic ADTs written in Ada 83, but it is not an unusual OO programming technique. Here, the client programmer is able to define the kind of items to store in the list by

externally deriving a new type from Item_Type (and defining appropriate additional components and/or operations):

```
type Integer_Item_Type is
   new Heterogeneous_Lists.Item_Type
   with record
      I : Integer;
   end record;
```

Figure 9. *A new kind of items for the heterogeneous list.*

As in section 3, since new types may be created by extending Item_Type to create a hierarchy of types rooted at Item_Type, the list data structure implemented above is heterogeneous in a *constrained* but *open-ended* fashion. It is not "arbitrarily" heterogeneous in the Smalltalk way [5].

There is no apparent advantage or disadvantage in using this version, except for a potential obstacle: an existing inheritance hierarchy would have to be modified at the root to derive from the ADT's Item_Type; the problem comes from trying to use inheritance to achieve multiple, distinct goals.

The raw interface to Item_Type is very limited, but it is possible to get to the properties of the derived types either by declaring a root user type derived from Item_Type, with dispatching to subprograms common to all kinds of items to be defined, or by using the membership test operator "**in**" and type conversions with run-time checks.

One difference with the technique presented in section 3 is that genericity is not used here, but exclusively type extension and a class-wide access type. This may be taken as yet another illustration of the extreme flexibility inherent in both the mechanisms of genericity and inheritance.

5. SELF-REFERENTIAL TYPES

There is yet another technique for the implementation of heterogeneous data structures, based on self-referential types (with access discriminants).

The construction is similar in spirit to one used by [6] in order to automatically collect all instances of a given type. An object which should be contained in some ADT has a component of type Hook_to_*ADT*, and that component has an access discriminant designating the enclosing object—thus allowing both ADT traversal and access to the object enclosing each node acting as a hook.

The example we will use is a representation for heterogeneous binary trees.

```
type Node;
type Node_Ref is access Node'Class;
type Node is
-- similar role to that of Item_Type in figure 8
-- (heterogeneous list ADT)
   abstract tagged
   limited record
      Parent,
      Left_Child,
      Right_Child : Node_Ref;
   end record;
```

Figure 10. *Heterogeneous binary tree data structure.*

The next figure shows how to add a new kind of items given the definition of heterogeneous tree presented above:

```
type T;
type Hook_to_Tree (Outer : access T) is
   new Node
   with null record;
type T is -- T may appear in a tree
   limited record
      ...
      Tree_Hook : Hook_to_Tree (T'access); -- self-reference
   end record;
```

Figure 11. *Item type T in heterogeneous binary tree.*

If T was defined as a tagged type, we could derive a new type NT with a second hook, thus defining a kind of item that may appear twice in a tree or simultaneously in two trees:

```
type T is -- T may appear in a tree
   tagged limited record
   -- assume we use T'Class in access discriminant of
   -- the hook
      ...
      Tree_Hook : Hook_to_Tree (T'access); -- self-reference
   end record;

type NT is
-- may appear twice in a tree, or in two separate trees
   new T
   with record
      Second_Hook : Hook_to_Tree (NT'access);
      -- second self-reference
   end record;
```

Figure 12. *Item type NT potentially twice in a tree.*

Any type having a component of a hook type (derived from Node) can thus be part of the heterogeneous tree structure. This provides *unconstrained* and *open-ended* heterogeneous data structures, without any restriction to types within an inheritance hierarchy (such as the ones based directly on class-wide access types have, as presented in sections 3 and 4). Also, note that type T need not be tagged.

This kind of classification is different from the usual "class" concept: objects can in their lifetime belong to various catalogues [4]. A *catalogue* is a collection of references to objects; the objects are

described in the catalogue but exist independently and remain external to it, unlike objects which are included in and parts of a collection.

This technique is rather complex to design and implement, but it is easy to understand and apply. To turn it into standard practice, it is best to abstract the structure with a generic unit; thus we will create a "framework" that captures this "pattern."

We now demonstrate multiple classification on the basis of mixins [3, 7] with the following specification:

```
package Heterogenous_Trees is
   -- same as before: pointer structure of a binary tree
   type Node;
   type Node_Ref is access Node'Class;
   type Node is
      abstract tagged
      limited record
         Parent,
         Left_Child,
         Right_Child : Node_Ref;
      end record;

   -- generic "add a kind of items"
   generic
      type T is -- to be derived from and extended with a hook
         tagged limited private;

   package In_Tree_G is
      -- a hooking node with access discriminant to
      -- T'Class objects
      type Hook_to_Tree (Outer : access T'Class) is
         new Node
         with null record;

      -- a new kind of T, self-referential through a hooking
      -- node component
      type Tree_T is
         new T
         with record
            Tree_Hook : Hook_to_Tree (Tree_T'Access);
         end record;
   end In_Tree_G;

end Heterogenous_Trees;
```

Figure 13. *Heterogeneous tree ADT with mixin hook.*

On the first hand, this technique has the disadvantage that it is applicable exclusively to limited types, since a component with an access discriminant is limited, the container type is limited too. However, self-referential objects must be of a specific nature, viz. they must have complete by-reference semantics. To simultaneously belong to several collections, an object has to be shared by reference, i.e. copy semantics would violate the referential integrity of the object; thus "**limited**" is necessary to enforce the desired integrity.

On the other hand, this approach has the definite advantage that it allows us to get out of the "constrained heterogeneous" and tagged type frame, since any type which contains an appropriate hook component can be added to our heterogeneous container without interference with already existing inheritance hierarchies. An additional advantage is that this technique renders multiple classification fairly easy. For instance, some items could be defined as potentially belonging to either a list or a tree, or to both at the same time and independently.

It is interesting to note that, where available, multiple inheritance (MI) is often used for the purpose of such multiple classification, i.e. for cases of multiple, cross-referencing catalogues. But MI is not necessarily the best tool, since for instance a famous area of dispute is what "repeated inheritance" should mean. By contrast, with the technique presented in this section, an item may have as many hook components as necessary, and the meaning of each component is immediately clear. E.g. an item containing two hooks may appear twice in a catalogue or in two otherwise unrelated catalogues.

6. CONCLUSION

We have described solutions and options in the implementation of heterogeneous data structures with Ada 95. The initial design and implementation may seem complex, but understanding and applying the techniques developed here is straightforward, especially if these techniques (or "patterns") are captured by frameworks available in the form of generic packages.

There are several alternative approaches that one may choose from, depending on specific requirements; these approaches all remain safely within the context of Ada's strong typing.

A generalization of the self-referential technique presented in [6] leads to a clean concept and implementation of multiple classification. This is achieved through the disciplined composition and application of self-referential types, not with multiple inheritance.

7. ACKNOWLEDGMENTS

The author would like to thank Stéphane Barbey, Gabriel Eckert, and Robb Nebbe for very helpful comments and discussions which helped improve this paper.

8. BIBLIOGRAPHY

1. *Reference Manual for the Ada Programming Language.* ANSI/ MIL-Std-1815a, 1983
2. *Programming Language Ada: Language and Standard Libraries*. ISO/IEC 8652:1995. Ada 9X Mapping/Revision Team, Intermetrics, Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138, MA, USA, January 1995. Also available at URL http://lglwww.epfl.ch/Ada/LRM/9X/rm9x/rm9x-toc.html
3. G. Bracha and W. Cook. *Mixin-Based Inheritance*. In *Proceedings of the OOPSLA/ECOOP'90 Conference*, ed. by N. Meyrowitz, Ottawa, Canada, 21-25 October, 1990, ACM SIGPLAN 25(10):303-312
4. G. Eckert. *Types, Classes and Collections in Object-Oriented Analysis.* In *Proceedings of First International Conference on Requirements Engineering*, Colorado Springs, Colorado, Apr. 18-22, 1994, IEEE Computer Society, pp. 32-39
5. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley 1983
6. M. Kempe. *Abstract Data Types Are Under Full Control with Ada 9X*. In *Proceedings of the TRI-Ada'94 Conference*, ed. by C. Engle Jr., Baltimore, Maryland, November 6-11, 1994, pp. 141-152. Also available at URL http://lglwww.epfl.ch/Ada/Resources/Papers/OO/ ADT_Control-revised.ps
7. M. Kempe. *The Composition of Abstractions: Evolution of Software Component Design with Ada 95.* In *Proceedings of the TRI-Ada'95 Conference*, ed. by C. Engle Jr., Anaheim, California, November 5-10, 1995. Also available at URL http://lglwww.epfl.ch/Ada/Resources/Papers/OO/ Components-revised.ps