THE GNAT IMPLEMENTATION OF CONTROLLED TYPES

Cyrille Comar
comar @ gnat.com
Ada Core Technologies
73 Fifth Avenue
New York, NY 10003

Gary Dismukes
dismukes @ gnat.com
Ada Core Technologies
10397 Avenida Magnifica
San Diego, CA 92131

Franco Gasperoni
gasperon @ enst.fr
Telecom Paris-ENST
Dep. INF, 46 rue Barrault,
75634 Paris, Cedex 13, France

1. Abstract.

This paper discusses the implementation model for supporting Ada 95 *controlled types* in the GNAT compiler [1]. After reviewing the semantics of controlled types, we outline the associated implementation problems and describe their solution in GNAT. The design addresses the management of controlled operations on various entities, including dynamically allocated objects, transient objects (function results and aggregates), and composite objects containing controlled components. The interaction of the controlled type features with exceptions is also covered. Finally, we discuss alternative implementation approaches and possible enhancements to the current model.

2. Controlled Types

Ada 95 [2] provides direct support for user-defined initialization, assignment, and finalization. These capabilities are important for the support of abstract data types because they permit full control over the creation, update, and clean-up of objects and their associated resources. In Ada 83 such control could only be partially achieved through the mechanisms of private types, default initialization, and by the use of limited types. In any event, the clean-up of resources upon scope exit was not possible.

In Ada 95, these operations are made available by means of "controlled types". Ada 95 provides two predefined tagged types, Controlled and Limited_Controlled, whose primitive operations are shown in figure 1. The dispatching operations Initialize, Finalize, and Adjust are used to provide default initialization, clean-up, and copying respectively.

```
type Controlled is abstract tagged private;
procedure Initialize (Object : in out Controlled);
procedure Adjust     (Object : in out Controlled);
procedure Finalize   (Object : in out Controlled);
```



```
type Limited_Controlled is abstract tagged limited private;
procedure Initialize (Object : in out Limited_Controlled);
procedure Finalize   (Object : in out Limited_Controlled);
```

Figure 1: Primitive operations for types Controlled and Limited_Controlled.

By default, the procedures Initialize, Adjust, and Finalize have no effect, but this behavior can be overridden by the user. Specifically, every controlled type (that is, types derived directly or indirectly from Controlled or Limited_Controlled) can be provided with implicit initialization, adjustment, and clean-up that apply to objects of the type:

- Initialization: Initialize is automatically invoked upon object creation, if there is no explicit initialization.
- Clean-up: Finalize is automatically invoked when the scope of the object is completed, to perform whatever clean-up is desired (for example, deallocation of indirect structures, release of locks, etc.).
- Adjustment:
  During an assignment OBJ1:= OBJ2 (only for nonlimited controlled types), OBJ1 is first finalized, the new value OBJ2 is copied into OBJ1, and then Adjust is automatically invoked on OBJ1.

Adjust can be used to specify whatever actions are needed to complete the construction of the new value for OBJ2. For example, it can effect the copying of any indirect data structures associated with OBJ2. This enables the designer to provide what is sometimes called a deep copy of the object, where a new copy of the indirect data is created for the target. This is in contrast to the shallow copy semantics that happen on normal assignment where attached structures are shared after the copy of pointers within the object.

Since Limited_Controlled is just a special case of Controlled (with no assignment or aggregates), in the remainder of this paper we will only talk in terms of the type Controlled.

When a scope contains several objects of controlled types, each object is initialized in the order of its declaration within the scope. Upon scope exit the objects are finalized in the reverse order. The reverse order is important since later ob-

jects may contain references to earlier objects. If an exception occurs during initialization, then only those controlled objects that have been initialized thus far will be finalized.

For instance, consider the following package that implements a set of integers:

```
package Int_Set is
  type Set is private;
  function  Empty return Set;
  procedure Insert (Elmt : Int; Into : in out Set);
  function  Exists (Elmt : Int; Into : Set)
                                    return Boolean;
private
  type Set is new Controlled with ...;
  procedure Initialize (S : in out Set);
  procedure Adjust     (S : in out Set);
  procedure Finalize   (S : in out Set);
end Int_Set;

with Int_Set; use Int_Set;
procedure Try is
  S1 : Set;              -- implicit call: Initialize (S1)
  S2 : Set;              -- implicit call: Initialize (S2)
begin
   Insert (1, S1);
   S2 := S1;             -- Finalize (S2) before the copy
                         -- Adjust  (S2) after the copy
end Try;                 -- Finalize (S2); Finalize (S1);
```

Figure 2: The Set Abstraction Example

Note that in this example, the finalization mechanism is hidden from the client, which only sees a regular private type. Alternatively, it could have been made apparent by defining Set as a private extension of Controlled.

The special operations of controlled types apply not only to stand-alone declared objects, but also to dynamically allocated objects and controlled components of composite objects. A dynamically allocated object is finalized either when the scope of its associated access type is exited, or when the programmer explicitly deallocates it.

An object that contains components of a controlled type is finalized in a manner similar to other objects: the components are finalized in the reverse order of their initialization within the containing object. The Adjust operation occurs for components when they are assigned individually, or upon assignment to their enclosing object.

If a controlled object includes controlled components, Initialize or Adjust is first invoked on the components and then on the enclosing object. Finalization is done in the reverse order. Finalization actions also occur for anonymous objects such as aggregates and function results. For these special objects, the finalization will occur upon completion of execution of the smallest enclosing construct, once the value of the aggregate or function result is no longer needed.

3. Implementation Problems

Not surprisingly, there are some tricky interactions between finalization and other languages features. In this section we discuss the most interesting of these issues.

- Exceptional Block Exit

In the event of an abnormal block termination, such as when an exception is raised or when the task containing the block is aborted, there may exist objects which have not yet been created and received proper initialization. For these objects, Finalize should not be called. For instance in the following code:

```
declare
   S1 : Set;
   X  : Pos := Random (0,1); -- Constraint_Error is
                             -- randomly raised
   S2 : Set;
begin
   null;
end;
```

Figure 3: Simple Use of Sets

S1 is initialized, but S2 might not be. Consequently finalization should always occur for S1 whereas S2 should be finalized only if it has been initialized. Thus an implementation consisting of a simple-minded insertion of explicit calls to Finalize at the end of the block is inadequate. Note that exceptions may be raised during initialization of composite object containing controlled components, in which case only the initialized part of the object needs finalization.

- Finalization of Anonymous Objects

Finalization actions for anonymous objects have to occur upon completion of execution of the smallest enclosing construct, that is, as soon as their value is no longer needed. This mechanism has to work even if an exception is raised in the middle of such a construct. In the following code, Empty is a function returning a controlled object that is kept in an anonymous object during the execution of the enclosing call to Exists:

```
declare
   X : Boolean := Exists (1, Into => Empty);
   -- The result of the call to Empty is kept in an anonymous
   -- object during the execution of Exists, and Finalize should
   -- be invoked no later than the semicolon.
begin
   if Exists (2, Into => Empty) then ...
   else ...
   end if;
   -- Here the anonymous object has to be finalized before
   -- the execution of either branche of the if statement
end;
```

Figure 4: Controlled Anonymous Objects Case

- Finalization of Dynamically Allocated Objects

In contrast to other similar languages, Ada 95 requires dynamically allocated objects to be finalized even if they are not deallocated explicitly. This default finalization occurs when the scope containing the access type is left. That is to say, in the following code, all objects allocated for type Set_Ptr, and not yet deallocated, must be finalized exactly between the finalization of Obj2 and Obj1:

```
declare
   Obj1 : Set;
   type Set_Ptr is access Set;
   Obj2 : Set;
begin ... end;
```

Figure 5: Access to Controlled Objects Case

- Problems Related to Mutable Objects

A variable of a discriminated type with defaulted discriminants may contain differing numbers of controlled components at different times. This possibility introduces an asymmetry between elaboration and finalization. In the following example no controlled components are present at the beginning of the execution, but after the assignment, X will contain three components:

```
declare
   type Sets is array (Natural range <>) of Set;
   subtype Index is Natural range 0 .. 10;
   type Rec (N : Index := 0) is record
       T : Sets (1 .. N);
   end record;
   X : Rec; -- 0 controlled components
begin
  X := (3, (1..3 => Empty)); -- 3 controlled components
end;
```

Figure 6: Mutable Objects Case

It is not easy to find a way to manage controlled operations for an object which may have a varying number of components during its lifetime.

- Controlled Class-Wide Objects

Type extensions can introduce additional controlled components. In general it's not possible to know, at compile time, whether a class-wide object will contain controlled components. If such an object contains controlled components, its initialization requires these components to be adjusted as shown in the next figure. A worst-case approach seems unavoidable.

```
package Test is
   type T is tagged null record;
   function F return T'Class is separate;
end Test;

with Test; use Test;
procedure Try is
   V : T'Class := F; -- does F yield a value containing
                     -- controlled components ?
begin
   ...;
end Try;
```

Figure 7: Class-wide Objects Case

4. Management of Controlled Types in GNAT

4.1. Basic Scheme

For each block that contains controlled objects, GNAT defines a Finalization Chain. When a controlled object is elaborated, it is first Initialized or Adjusted (depending on whether an initial value was present or not), then attached at the beginning of this chain. The following example gives an idea of the code generated. The following declarations:

```
X : Ensemble;
Y : Ensemble := X;
```

are transformed into:

```
X : Ensemble;
Initialize (X);
Attach_To_Final_List (F, Finalizable (X));
Y : Ensemble := X;
Adjust (Y);
Attach_To_Final_List (F, Finalizable (Y));
```

Finalizable is the name of the class representing all controlled objects, limited or not, and is defined in the GNAT library as follows:

```
subtype Finalizable is Root_Controlled'Class;
```

Upon scope exit, the scope's finalization chain is traversed and Finalize is called on each element. Note that, since objects are inserted at the beginning of the list, the ordering of the chain is just right for the required sequence of finalization. The fact that the chain is dynamically built ensures that only successfully elaborated objects are dealt with in case of exceptional exit.

To ensure that finalization happens regardless of how the a scope is left, we have introduced an "At_End" construct in the compiler. This mechanism consists of a call to a parameterless subprogram executed unconditionally upon scope exit. This routine performs all clean-up actions required by the semantics of the language, such as waiting for subtask completion. The next figure shows the simple clean-up procedure that is generated for scope finalization and its point of call:

```
declare
    Final_Chain : Finalizable_Ptr;
    procedure _Clean is
    begin
       Finalize_List (Final_Chain);
```

```
    end _Clean;
begin
   <user code using controlled objects>
at end
   _Clean; -- executed before leaving the scope
end;
```

Figure 8: The AT END Mechanism

Finalize_List is a library routine that finalizes all objects on the list referenced by its parameter, regardless of any exception that could be raised during the process. The list is heterogeneous because Finalize_Ptr is an access-to-class-wide type, and any object whose type is derived from Controlled can be attached to this list. This is handled by a call of the form "Finalize (Ptr.all)" in the Finalize_List procedure, which will dispatch to the appropriate user-defined finalization procedure.

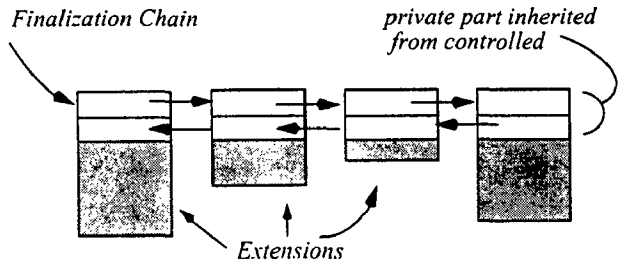

Figure 9: A Finalization Chain

Finalization chains are bidirectional to ease the removal of a single object from the middle of the list. The predefined types Controlled and Limited_Controlled both include two hidden pointers. Removing an element from the middle of a finalization list occurs during the deallocation of dynamically allocated objects. However, user driven deallocation can happen in an order that is not related to allocation.

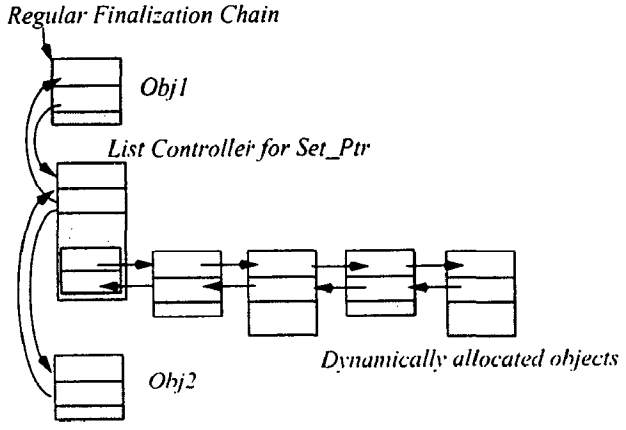

Figure 10: Finalization Lists for Dynamically Allocated Objects

To ensure that objects that are not explicitly deallocated are finalized at the right time, the finalization chain on which they are attached is implicitly defined at the point of the access type definition. This finalization chain is itself enclosed in a controlled object (a List_Controller) whose finalization consists of calling Finalize_List on its associated internal list. The previous figure shows the finalization structures associated with the block defined in figure 5.

4.2. Assignment of Controlled Objects

A simple-minded implementation of the assignment operation Set1:= Set2; discussed in section 2, might be:

```
Finalize (Set1);
Set1 := Set2;
Adjust (Set1);
```

There are various problems that make such an implementation unworkable: first, Set1 may refer to objects present in Set2 and thus cannot be finalized before Set2 is evaluated; second, the assignment itself must be specialized since copying the hidden pointers that hook objects to a finalization list doesn't make any sense; third, the self assignment (X := X;), although not a particularly useful construct, has to be addressed specially, either by introducing a temporary object or by avoiding the production of any finalization actions. Here is a model that works in the general case and can be optimized in many cases:

```
Anon1 : Ctrl_Typ renames Set1;
Anon2 : Ctrl_Typ_Access := Eval (Set2);
if Anon1'address /= Access_To_Address (Anon2) then
   Finalize (Anon1);
   Copy_Explicit_Part (Anon2.all, To => Anon1);
   Adjust (Anon1);
end if;
```

Figure 11: Code for controlled assignment

Note that the target object, even though it has been finalized, remains in the finalization list because it still need to be finalized upon scope exit.

4.3. Management of Anonymous Objects

Some constructs such as aggregates and functions generate anonymous objects that are used to store an intermediate result. When such objects are controlled, they must be finalized as soon as they are no longer needed, that is to say before the beginning of the next statement. GNAT defines "transient blocks" to handle this. Such a block is expanded around the construct that uses the intermediate objects. For instance, in the following example, the function Empty yields a controlled value that is only used during the execution of Exists:

```
declare
   X : Boolean;
begin
   ...
   X := Exists (1, inside => Empty);
   ...
end;
```

```
declare
   Anon : Set := Empty;
begin
   X := Exists (1, inside => Anon);
end;
```

Figure 12: Expansion of a Transient Block around a Statement

An intermediate block can be introduced without changing the semantics of the program in order to make the anonymous object explicit. This new block contains a controlled object and thus will be expanded using the scheme presented in section 4.1. The same kind of mechanism can be extended to deal with anonymous objects that appear in the Boolean expressions of control structures such as if and while statements. For instance:

```
while Exists (Elmt, Empty) loop
   ...
end loop;
```

```
loop
   declare
      Anon : Set := Empty;
   begin
      Res := Exists (Elmt, Anon);
   end;
   exit when not Res;
   ...
end loop;
```

Figure 13: Expansion of a Transient Block around an Expression

The problem is a bit more complex when controlled anonymous objects appear in a declaration since transient blocks are not allowed in such a context: if such blocks were allowed in declarative parts, they would make the declaration they are enclosing local to their scope, which is obviously improper. To handle this case, the anonymous object is created in the original enclosing scope but it is attached to an intermediate finalization list, represented by the same List_Controller that was used for the dynamic allocation case, and which is finalized right after the declaration. For example:

```
B    : Boolean := Exists (1, Empty);
```

is transformed into:

```
L    : List_Controller;
Anon : Set := Empty;
Adjust (Anon);
Attach_To_Final_List (L, Anon);
B    : Boolean := Exists (1, Anon);
Finalize (L);
```

List_Controller is itself a controlled type. Thus, an object of that type is attached to the enclosing scope's finalization chain, ensuring that the anonymous object will be finalized even if an exception is raised between its definition and the finalize call. In the normal case, the List_Controller is finalized twice, once right after the declaration, and once upon scope exit, So the Finalize routine makes sure that the second finalization has no effect.

4.4. Management of Objects with Controlled Components

Composite type such as records (tagged or not) and arrays can contain controlled components. Objects of these types require specific actions that take care of calling the proper Initialize, Adjust, and Finalize routines on their components. These actions are carried out in implicit procedures called _Deep_Initialize, _Deep_Adjust and _Deep_Finalize that are used in a manner similar to their counterparts for regular controlled types. Here is the body of _Deep_Adjust for a type T that is a one-dimensional array of controlled objects:

```
procedure _Deep_Adjust (V : in out T;
                        C : Final_List;
                        B : Boolean) is
begin
   for J in V'range loop
      Adjust (V (J));
      if B then
         Attach_To_Final_List (C, V (J));
      end if;
   end loop;
end;
```

Figure 14: Adjustment of Array's Controlled Components

Note that the deep procedures have a conditional mechanism to attach objects to the finalization chain so that the same procedure can be used in a context where attachment is required, such as explicit initialization, as well as when it is not needed, such as in the assignment case. Note also the recursive nature of the above definition: Deep_Adjust on an array is defined in term of Deep_Adjust of its components. Ultimately, if the component type is a simple controlled type with no controlled components Deep_Adjust ends up just being a call to the user-defined Adjust subprogram.

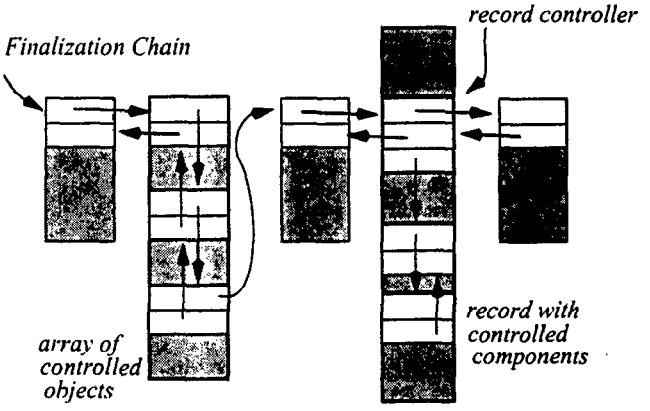

Figure 15: Finalization Chain of Objects with Controlled Components

A similar method could have been used for records. In that case, deep procedures would have been implicitly defined to perform the finalization actions on the controlled components in the right order, depending of the structure of the type itself. The controlled components would have been stored on the finalization list of the enclosing scope. Unfortunately such a model makes the assignment of mutable objects quite difficult: the number of objects on the finalization list may be different before and after the assignment, so all the controlled components of the target would need to be removed from it before the assignment and afterwards put back at the same place on the list. To avoid such a problem as well as to simplify the definition of deep procedures for records a different approach has been used. Records are considered as scopes and they have their own internal finalization chain on which all controlled components are chained. This is achieved by inserting a hidden component, the Record_Controller within the record itself. The next figure shows the compiler's transformation of the mutable record Rec presented in figure 6:

```
type Rec (N : Index := 0) is record
   _Controller : Record_Controller;
   T : Sets (1 .. N);
end record;
```

Record_Controller plays a role equivalent to List_Controller: it introduces an indirection in the enclosing finalization list. The finalization list of controlled components is local to the object. So, upon assignment the number of controlled components may vary without affecting the enclosing finalization list. This provides a simple solution to the mutable record problem. In GNAT, finalization pointers are absolute and when they are part of a local component list, they have to be adjusted after a copy. This action is carried out by the Adjust subprogram associated with the Record _Controller type which uses for this purpose its own component my_address that is initialized with the original address and is used to compute the displacement:

```
type Record_Controller is new Root_Controlled with
   record
      Component_List : Finalizable_Ptr;
      My_Address     : System.Address;
   end record;
```

Class-wide objects present an interesting challenge since the compiler doesn't know how many, if any, controlled components are present in such objects. To address this problem, class-wide types are considered "potentially" controlled and calls to the *deep* procedures are always generated for initializations and assignments. Dispatching is used to ensure that the appropriate *deep* procedure is called. Thus, such deep procedures must be defined as hidden primitive operations for all tagged types.

5. An Alternative Implementation: The Mapping Approach

The solution presented in the previous section is one of several possible approaches. Its storage costs are fairly high since every controlled object contains two additional pointers and the size of a record with N controlled components is increased by 2*N+4 pointers.

There exists another approach to implementing finalization, sometimes called PC mapping. This method is often used in C++ compilers.

The method derives its name from the way in which abnormal block exits are handled. More specifically, if, during an abnormal block exit, it were possible to determine the program counter (PC) where the abnormal exit occurred, it would be possible to figure out the precise list of objects that need to be finalized.

In what follows we explain a simplified version of the PC map approach which actually does not use any PC information. Albeit less efficient, this simpler method is hardware independent and hence more portable than the full-blown PC map.

The idea is to build a map for every scope (be it for a block, procedure, or record) containing controlled objects. Every entry in this map represents the state of a controlled object created in the block. If the entry for a given object is true, then the object has been initialized and hence must be finalized upon block exit.

This map can be implemented as a packed Boolean array. In the absence of anonymous objects, the map of a given scope is sequentially updated to true during elaboration.

The presence of anonymous objects may create holes in a map (see example below). This is why we need a full-fledged map rather than a simple counter.

The following example shows how the map is initialized

and how finalization is carried out. Here is a simple user program that uses a few of the controlled types features described in this paper:

```
declare
   S1  : Set;
   V   : Integer;
   B   : Boolean := Exists (2, Into => Empty);
   Obj : Rec_With_Ctrl;
begin
   null;
end;
```

Here is a possible transformation of this program fragment that illustrates the use of the maps for finalization:

```
declare
   Map: array (1..3) of Boolean := (others=>False);
   S1 : Set;
   Initialize (S1);
   Map (1) := True;
   V : Integer;
   Anon : Set := Empty;
   Map (2) := True;
   B : Boolean := Exists (2, Into => Anon);
   Finalize (Anon);
   Map (2) := False;
   Obj : Rec_With_Ctrl;
   _Deep_Initialize (Obj);
   Map (3) := True;

   procedure Clean is
   begin
      for I in Map'range loop
         if Map (I) then
            case I is
               when 1 => Finalize (S1);
               when 2 => Finalize (Anon);
               when 3 => _Deep_Finalize (Obj);
            end case;
         end if;
      end loop;
   end;
begin
   null;
AT_END: Clean;
end;
```

Records containing controlled components have their own (internal) initialization map as an additional field. For each array of controlled objects we need to add a specific map with as many entries as there are elements in the array.

The mapping model looks fairly attractive from the storage point of view since it only requires an additional bit per controlled object. On the down side, this method requires more complex deep finalization and clean-up procedures and the space gained in the objects is paid by an increase in object code size.

The major drawback of the mapping approach is its inability to cope with dynamically allocated controlled objects. For that case, the linked list implementation seems to be the only possible choice.

6. Conclusion

This paper explains the implementation of Ada 95 controlled types in the GNAT compiler. Even though the implementation method described in this paper entails some space overhead at run time (compared to the map-based implementation described in section 4), the method is very portable (in contrast to PC maps) and the machinery that is embedded in the user code by GNAT is modest. Indeed a good part of this machinery is implemented once and for all in the GNAT library and can therefore be shared by all controlled types.

One possible improvement in our implementation would be to omit the hidden backward pointer for statically allocated controlled objects. The only purpose of this pointer is to support user deallocation. This would necessitate specialized allocation of dynamic objects requiring finalization.

7. Bibliography

[1] E. Schonberg, B. Banner, "The GNAT Project: A GNU-Ada9X Compiler", in Conference Proceedings of TRI-Ada 94, 1994.

[2] Ada 9X Mapping/Revision Team, "Programming Language Ada--Language and Standard Libraries", Version 6.0, Intermetrics, January 1995.

[3] A. Appel, "Garbage Collection", in "Topics in Advanced Language Implementation", MIT Press, 1991, pp. 89-100.

[4] N. Sankaran, "A Bibliography on Garbage collection and Related Topics", SIGPLAN Notices, vol. 29, no. 9, September 1994, pp. 149-158.

[5] P.R. Wilson, "Uniprocessor Garbage Collection Techniques", in "International Workshop on Memory Management", Lecture Notes in Computer Science no. 637, Springer-Verlag, 1992, pp. 1-42.