Abstract Data Types Are Under Full Control with Ada 9X

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*

ABSTRACT. Ada 83 did not provide enough control on the creation, assignment, and destruction of objects of user-defined types. This lack of control restricted type composition and prevented the full exercise of information hiding for abstract data types. Ada 9X brings new mechanisms supporting the full control of abstract data types, powerful type composition, and more extensive information hiding. With better control of abstract data types, Ada code will be easier to write, understand, maintain, and reuse. Thus, applications written in Ada 9X will be safer.

KEYWORDS. Abstract data types, Information hiding, Ada 9X, Object initialization, User-defined assignment, Object finalization, Memory reclamation, Storage management, Reference counting, Type composition, Collections.

1. INTRODUCTION

Ada is known for the quality of its design and the power of its basic mechanisms, especially in support of the construction and composition of abstract data types (ADTs). Today, Ada is emerging from its first revision process [Ada94a], during which it has acquired additional mechanisms serving to further strengthen Ada's support for writing ADTs. Information hiding is an integral part of the language, and serves the development and implementation of representation- and algorithm-independent specifications.

Ada 9X, the revised definition of the Ada programming language, fully supports abstraction-oriented programming [Taf93], building on the already powerful mechanisms separating specification from implementation in Ada 83 [Ada83]. This paper examines some new features of the language and shows how to combine these into valuable programming techniques for ADTs; topics include immediate memory reclamation, reference counting, and the construction of objects with initial values and parameters.

This paper shows how an Ada implementation of an ADT can properly reflect some advanced aspects of the desired semantics, such as the behavior from the point of view of referential integrity.

First, we identify some problems faced by Ada 83 programmers, and introduce some new basic building blocks provided by Ada 9X. We then show exactly how to use these building blocks in order to fully, and cleanly, solve the problems identified earlier.

The paper does not deal with multiple implementations of abstractions through object-oriented programming, it does not present formal specifications (see Anna [Luc84] and [Hen85]), and it does not examine models for and implementations of storage management in Ada (the latter have already been adequately dealt with in the literature—cf. [Men87] and [Kow87]).

1.1. Problems with Abstract Data Types Written in Ada 83

An Ada package specification exporting a private type is not a full ADT specification; it only provides a syntactic description with minimal semantics (such as may for instance be provided by generic parameters). The syntactic specification of an ADT is given by the names, domains, and ranges of operations associated with the private type (i.e. its primitive operations); in addition, but lacking in Ada, an ADT has an axiomatic definition, i.e. a set of equations defining the meaning of the operations by stating their relationships to one another. Informally, the axiomatic definition is intended to capture the distinguishing characteristics of the ADT (e.g. what makes a queue different from a stack) [Gut77].

With Ada 83, programmers often resorted to limited types for ADTs in order to avoid some unwanted effects of the assignment operation (since the effect of the assignment operation could not be user-defined). For instance, if some component of an ADT was of an access type, the avoidance of structural sharing could only be enforced at the price of making the ADT limited, while exporting an Assign subprogram to simulate ":=" for client programmers [Ros87] (to circumvent this limitation, [Bei92] suggested a change in the language to support a functional style of programming with limited ADTs).

Since Ada 83's predefined semantics of the assignment operation was heavily dependent on the particular data representation chosen for an ADT, it restricted freedom of both design and maintenance choices (see e.g. [Rob92]). This potential and frequent dependence on the representation was clearly in direct opposition to information hiding: some representation choices could not be abstracted since they could not be completely isolated from the specification. The "=" operator presented a similar, unfortunately complementary problem: only limited private types allowed its definition by the user, with a complex workaround described by [Goo85]. [Lla84] noted that these problems created a conflict between unlimited ADTs and the reliability of Ada programs, as it hampered both type composition and maintenance.

One very unfortunate consequence of this lack of information hiding was that generic ADTs (such as a generic Stack ADT parameterized with an Item type) usually had to be provided in two variants: one for non-limited Item types, and another one for limited Item types with an Assign subprogram (or else the client programmer had to juggle with access-to-limited types). This unwanted dichotomy made type composition and the creation of reusable components more difficult than they were intended to be (see e.g. [Gen89], [Boo87], and [AQS92]).

Another issue was how to guarantee that all instances of an ADT would always have a valid state, or, formally, how to establish an invariant which could be used in proofs of the ADT's correctness. One prerequisite is that a constructor operation—with appropriate parameters—be used to create instances of the ADT, adequately setting up their state in order to satisfy the invariant (this operation is often called INIT or NEW in the literature). Another desirable mechanism would be pre- and post- conditions on all exported operations, but this was and still is *not* part of Ada.

To ensure the safety of using an ADT, Ada programmers had no "type initializers" or "constructors," so they resorted to always giving some "neutral" initial value to the components of their ADT, while expecting that client programmers would anyway re-initialize the ADTs to suit their own, specific purpose [Boo87]. Even an Ada style guide, [Nis84], advocated systematic default initial values for the components of private types. This approach was inefficient, and it made the selection of such initial invariants awkward for proof purposes as well as more general than necessary ([She82] advocated an efficient but unreliable solution).

Finally, Ada 83 offered no mechanism to support automatic clean-up actions for instances of ADTs. [Sch81] and [She82] described the importance of such finalization procedures for the implementation of ADTs (e.g. recovering storage, closing files, finishing protocols); [Bak91] and [Bak93] showed how to emulate automatic resource management in Ada 83, with limited private types and a clever use of generics.

1.2. The Software Engineer's Context

In this situation, most C programmers would say: drop information hiding, expose the internal structure and pointers, and then let each client programmer code his own Assign subprograms and initialize each component of the ADT, each time he thinks he needs to.

In other words, *"the programmer needs some self-discipline."* (He probably works on miniature, non-critical applications, and likes to test his code with a debugger...) However, we don't want to join this breed of programmers. Rather, we value a language where information hiding is a tool, not an obstacle, i.e. a language which supports the application of sound, well-established software engineering principles such as information hiding, modularization, low coupling, and high cohesion. These principles, and others not enumerated here, set the context of the software engineer.

In this paper we take for given that it is unsatisfactory to rely on the client programmer's self-discipline in order to guarantee the proper use of ADTs. Our view is that proper software engineering practice is to enforce, and to be able to formally check, the disciplined, valid use of ADTs.

1.3. Some Additions of Ada 9X as Solutions to the Implementation of Abstract Data Types

Ada 9X takes advantage of the existing Ada 83 features and extends them with additional well-integrated mechanisms. The revision process did not lead to a hybrid language: a minimal number of building blocks were added. The additions are flexible, solve many problems, and encourage good software engineering principles, as we attempt to show here; these additions remain faithful to the philosophy of Ada 83. The expressive power and support of Ada for ADTs now equals, or even exceeds, that of other object-oriented languages.

For the purpose of solving the problems described above, we consider here two specific new building blocks: controlled types and indefinite types.

1.3.1. Controlled Types

Ada 9X supports user-defined *initialization* and *finalization* of objects and values; this allows precise control over allocation, duplication, and deallocation of data storage. This control over the life cycle both of objects and of their values most notably makes it possible to customize the assignment operation, and it provides immediate control over object disposal (i.e. finalization).

In the following sections, we attempt to categorize the various potential uses of controlled types, in relation to ADTs: reference counting, avoiding memory leaks, breaking structural sharing, instance counting, and automatic "extension" of a type (vs. "intension"). We will show that these possibilities add up to more and better control of ADTs, thus reducing the amount of work necessary to use them.

Some other possible applications are not dealt with in this paper, because they are not directly relevant to the programming of ADTs: debugging (actions on entering/leaving a block), collecting statistics (for final printout), interfacing/implementing persistent objects (e.g. external files, graphical windows, network sockets), automated clean-up measures (in tasks and packages), smart references (à la Macintosh resources), and garbage collection (with user-defined storage pools and, probably, some programming patterns and/or compiler support).

Section 2, "The Nature of Controlled Types," details the mechanisms made available for the control of the birth and death of objects and values. Section 4, "Memory Reclamation," section 5, "Reference Counting," and section 6, "Breaking Structural Sharing," demonstrate the main practical uses of control for ADTs. Finally, section 7, "Types and their Population," shows that some type-related state can be automatically maintained during initialization and finalization of objects belonging to a given controlled type, as well as for each assignment operation to an object of that type.

1.3.2. Indefinite Types

Indefinite types form a new category in the language, subsuming the unconstrained types (i.e. unconstrained array types, and types with "unknown discriminants" or with unconstrained discriminants without defaults). This category is formally recognized and handled with a specific construct in the language, i.e.:

```
type T (<>) is ...
```

Objects belonging to indefinite types need an additional constraint or an explicit initial value, in order to constrain the object. Private types can be announced as being indefinite, which is a step forward in information hiding (e.g. one need not show in public the discriminant of a private type if that type is declared as being indefinite).

Note also that Ada 9X lets a variable receive its constraint either from an explicit initial value or from a constrained subtype indication, as Ada 83 already did for constants; for instance the index constraint of a String variable can be given by its explicit initial value:

```
S : String := "12345";
```

Section 3, "Initializing Objects," shows some practical advantages derived from the declaration of indefinite private types.

2. THE NATURE OF CONTROLLED TYPES

In order to understand how user-defined initialization, finalization, and assignment is managed, it is necessary to grasp that controlled types provide hooks to the *birth* and *death* of objects and values.

The distinction between objects and values is important for programming [Mac82]. In Ada, the term "object" subsumes variables, constants, and heap-allocated storage. For instance, a variable is born where its scope starts and dies where its scope ends. A heap-objects is born when storage is allocated and dies when either it becomes inaccessible or the storage is deallocated. "Value" subsumes the current state of an object. An object may change in value over time, while still keeping its identity [Eck93]. It follows that for each assignment to an object the value of the object dies before a new value is born in the object; the value also dies when the object dies.[1]

In this paper, we speak of initialization and finalization, since these are the terms officially used in [Ada94a]. "Birth" and "death" are more generic. More precisely, type control is about *automatic* birth and death; the explicit, manual, control is and has always been possible, but is error-prone, unsafe, bad for information hiding, and extremely cumbersome.[2]

A word of warning: controlled types do not provide garbage collection. The former allow to manage required logical operations on objects, while the latter is a means to optimize the use of finite/scarce memory. Garbage collection works by doing periodic "heap reclamation" and determining which pieces of allocated memory no longer have any pointers to them, thus letting the programmer forget about when/where objects become useless. In many cases, controlled types take care of situations which are sometimes thought of as requiring garbage collection.

2.1. The Life of Objects and of Their Values

User-defined control is based on object-oriented programming in Ada 9X. It is available for tagged types (extensible record types with run-time type identification for dynamic binding) in two predefined derivation classes (inheritance hierarchies): the *controlled* classes. The abstract root types of these predefined classes (the ultimate ancestors in the inheritance hierarchies) are defined as follows:

```
package Ada.Finalization is
   type Controlled is
      abstract tagged private;

   procedure Initialize (Object : in out Controlled);
   procedure Adjust (Object : in out Controlled);
   procedure Finalize (Object : in out Controlled);
```

1. Software Platonists will probably prefer to speak of the creation and destruction of "references to, or images of, values," not of the mortality of (immutable) "values" themselves.

2. What Ada calls "initialization" and "finalization," C++ calls "default construction" and "destruction." Although the subject matter of this paper is not a comparison of both languages, it should be noted that Ada and C++ do not have identical expressive power in this area.

```
    type Limited_Controlled is
      abstract tagged limited private;

    procedure Initialize (Object : in out Limited_Controlled);
    procedure Finalize (Object : in out Limited_Controlled);
  end Ada.Finalization;
```

Figure 1. *Specification of the abstract root types of the controlled classes (RM9X-7.6;5.0).*

The three user-definable operations (Initialize, Finalize, and, for non-limited types, Adjust) are primitive operations of every *controlled* type—i.e. of every type derived, directly or indirectly, from one of the two predefined abstract *controlled* types.

The three control operations are automatically called according to specific events in the life cycle of the controlled objects and, in the case of non-limited types, of their values. These operations have by default null implementations, i.e. they do nothing unless they are overridden for each user-defined controlled type. We describe below under which conditions these operations are automatically called. They may also be explicitly called, normally in order to implement them on the basis of a parent type's implementation (like a message sent to "super" in Smalltalk).

2.2. Finalize

The operation Finalize is automatically (and implicitly) called before an object's storage is destroyed, and, for non-limited types, before an object's value is overwritten in an assignment operation. Storage destruction happens either on scope exit (for stack allocated objects), or on explicit deallocation of memory by a call to an instantiation of Unchecked_Deallocation (for heap allocated objects)—or on implicit deallocation by the ever-hypothetical, implementation-defined garbage collector.

Programmers must be aware that, for non-limited types, Finalize will be called both before the bitwise-copy part of assignment operations and before storage destruction. The code of that operation cannot discriminate between the two conditions. This is a serious restriction, to which we shall return later.

The expected implementation of finalization is based on per-task lists of to-be-finalized objects. There is no distributed cost. In particular, real-time applications should be able to use controlled types without fear of unpredictable performance or unexpected dynamic activities (such as could occur with most garbage-collecting algorithms), and non-controlled types do not carry any control-related overhead.

2.3. Adjust

The operation Adjust is automatically (and implicitly) called as the last step in an assignment operation targeting a controlled object, i.e. after the bit-wise copy is performed.[3]

The purpose of adjustment in an assignment operation is usually to generate an appropriate value for the target object based on that copied from the source—i.e. to customize the assignment operation, which is how Ada 9X provides user-defined assignment. The assignment operation is used in assignment statements, explicit initialization, by-copy parameter passing, and a few other conditions; adjustment never occurs in parameter passing, since parameters of tagged types are always passed by-reference, and the controlled types are tagged.

Note that when a controlled object is declared with an explicit initial value, Adjust is called as part of the assignment operation involved in setting the initial value.

2.4. Initialize

The operation Initialize is automatically (and implicitly) called after allocating storage for a controlled object and default initializing its components—unless an explicit initial value is provided. In the presence of an explicit initial value, there is no need for automatic initialization; as indicated above, Adjust will be called.

2.5. Allowed Optimizations, and Side-Effects

It is interesting to know what compilers are required or allowed to do with the control operations.

In the case of an assignment statement such as "X := ...;" X is finalized after evaluating the right-hand side expression and before the bitwise copy, since X might be used in the right-hand side expression.

In the case of a "self-assignment", such as "X := X;" the compiler is allowed to either do nothing or use a temporary object. Thus, in this case, if there could be a side-effect to Adjust or Finalize, too bad. If the side-effect is e.g. to compact storage, it should rather be done by an explicit call to a subprogram "Compact (X);". This is clean, readable, and reliable; subsequent maintenance will keep that call unless it is explicitly decided that it should be removed, whereas a self-assignment is prime target for innocuous-looking maintenance changes or new compiler optimizations. Tailored assignment should not be used for other purposes than assignment between distinct objects; self-assignment is held to be a no-op, even if it is only determined at run-time that the source is identical to the target.

An object declared as belonging to a controlled type cannot be optimized away, even if it is not used anywhere. This guarantees that Initialize (or Adjust) and Finalize will be called.

3. *Historical note*: this operation went through several names–Clone, Duplicate, Split, Post_Copy–mainly due to its unclear semantics, depending on the context of its envisioned uses. The name finally chosen, Adjust, reflects the fact that this operation allows users to redefine the assignment operation by adjusting the target after a bitwise copy has been done. Adjustment is thus the means to tailor the effect of the assignment operation.

Passing a controlled type as actual generic parameter matching a generic formal private type has the expected effect: objects of that type, declared and/or manipulated within the generic unit, are controlled with the appropriate automatic calls to the user-defined operations.

3. INITIALIZING OBJECTS

This section deals with explicit initial values for instances of ADTs. Two approaches may be desired: first, the client programmer should be able to specify an initial value, but is not required to; second, the client programmer is required to provide an initial value for each instance of the ADT. In addition, initial values and objects may be parameterized.

We examine four, non-exhaustive variants: i) allowing an explicit initial value with a set of parameters (in addition to a copy of an object of the same type); ii) requiring an explicit initial value, with or without parameters; iii) requiring visible, permanent parameters; and iv) requiring an explicit initial value other than an object of the same type. In the first case, the programmer can also choose to have some elaborate initialization code executed in the absence of an explicit initial value.

3.1. Allowing Parameterized Initial Values

To allow parameterized, explicit initial values, it is sufficient to export a non-limited ADT with appropriate constructors, as could already be done in Ada 83. For instance, the "Cat" ADT shown below takes into account unnamed cats, named cats, and cats of known or unknown kinds:

```
package Cats is
   type Cat_Type is
      private;
   type Kind_Type is ( Siamese, Egyptian, Burmese, ... );
-- constructors
   function Cat (Name : String)
      return Cat_Type;
   function Cat (Name : String; Kind : Kind_Type)
      return Cat_Type;
-- operations
   procedure Feed (C : in out Cat_Type);
   ...
private
   ...
end Cats;

use Cats;
Kira : Cat_Type := Cat ("Kira", Burmese);
A_Cat : Cat_Type; -- unknown name and kind
```

Figure 2. *ADT allowing explicit initial values with parameters.*

3.2. Requiring Explicit Initial Values

Next, we use a private indefinite type (see 1.3.2, "Indefinite Types"), since all instances of an indefinite type require an explicit initial value (the full type declaration could, but need not, have discriminants, or it could be an unconstrained array type). The Create functions declared below correspond exactly to the constructors one finds in C++, and the Ada equivalent of a "default constructor" would be a parameterless Create function. In this example, the compiler enforces the presence of explicit initial values for ID objects—with either one Integer or one String parameter:

```
package Identifiers is
   type ID_Type (<>) is
      private;
-- constructors
   function Create (S : String) return ID_Type;
   function Create (I : Integer) return ID_Type;
   ...
private
   type Disc is ( Str, Int, ... );
   type ID_Type (D : Disc) is
      record
         ...
      end record;
end Identifiers;

use Identifiers;
X : ID_Type := Create ("Francisco");
Y : ID_Type := Create (12);
Z : ID_Type := X;
-- W : ID_Type; -- illegal, missing explicit initial value
```

Figure 3. *ADT requiring explicit initial values.*

3.3. Requiring Visible, Permanent Parameters

In Ada, discriminants are the means to parameterize types and their instances, thus we use discriminants (without default expression) for permanent parameters. A discriminant must be either of a discrete type or, in case the parameter is not discrete, of an access type. The initial value is thus kept during the whole life of the object, and is always visible from the outside. An alternative might be to declare an indefinite type with selectors always returning the value of the initial parameters. In this example, each symbol is parameterized with an access to ID object:

```
with Identifiers;
package Symbols is
   type Sym_Type
      (ID : access Identifiers.ID_Type) is -- access discriminant
      limited private;
   ...
private
   type Sym_Type
      (ID : access Identifiers.ID_Type) is
      limited record -- by-reference limited type
         ... -- components for symbol attributes
      end record;
end Symbols;

use Identifiers, Symbols;
S : Sym_Type (new ID_Type'(Create ("Hélène")));
T : Sym_Type (S.ID'Access); -- different symbol, same ID
```

Figure 4. *ADT requiring permanent parameters.*

3.4. Requiring Explicit Initial Constructors

Finally, in the presence of a limited type—thus without assignment operation—and if discriminant are not satisfactory, an indefinite type can still be used to require initial constructors, but with a trick. The client programmer will have to use **renames** of function calls in order to declare entities of that type!

```
package Houses is
   type House_Type (<>) is
      limited private;
-- constructors
   function House (Owner : String)
      return House_Type;
   function House (Owner : String; Price : Positive)
      return House_Type;
-- operations
   procedure Register
      (H : in House_Type; New_Owner : in String);
   ...
private
   ...
   type House_Type is access ...;
end Houses;

use Houses;
X : House_Type
   renames House (Owner => "Midas Mulligan");
```

Figure 5. *ADT requiring explicit initial constructors.*

Note that since a function call is a name that denotes a constant object, the X declared in figure 5 is a constant. Since renaming has by-reference semantics, it is fine to rename a limited object (no copy is made); the renaming declaration being a constant object sharply reduces its utility, unless the ADT is implemented as an access to the "real thing" with appropriate operations.

The separation between limited and non-limited types shows weaknesses in many attempts at unified uses. In this case, providing initial constructors (not parameters) for a limited type forces one to use access types behind the scene, and to abandon the meaningful use of parameter modes (always **in** for constants). To solve this problem, and others, it would be desirable to sunder and render orthogonal several mechanisms, such as assignment, test for equality, and explicit initial values; for instance, Ada 9X now correctly treats the "=" operator like all other operators, i.e. it is user-definable for all types.

4. MEMORY RECLAMATION

In this section, we demonstrate how to guarantee the absence of memory leaks by encapsulating automatic, immediate per-object memory reclamation in ADTs.

In Ada 83—leaving aside the infamous, hypothetical garbage-collector to be optionally provided by Ada implementations—memory reclamation could, but need not, happen on exit of the scope of each access type and its associated collection [Ada86].

Ada 9X associates storage pools with access types; for each access type, a storage pool may be user-defined, thus implicitly redefining both the operation **new** and the corresponding body of Unchecked_Deallocation. The definition of a storage pool guarantees reclamation on exit of the scope of the corresponding access type, and serves to implement a particular model for storage management [Ada94b].

With both storage collections and storage pools, memory reclamation is defined in terms of access types, not of objects. By contrast, finalization can be used for the immediate disposal of objects on scope exit and deallocation, independently of the model for storage management.

The following example demonstrates that the operation Finalize helps to recover memory on a per-object basis. We show a Stack ADT implemented with a linked list:

```
with Ada.Finalization;
generic
   type Item_Type is private;
package Stacks_G is
   type Stack_Type is
      limited private;
   ... -- various operations
private
   type Cell_Type;
   type Link_Type is access Cell_Type;
   type Cell_Type is
      new Ada.Finalization.Controlled
      with record
         Item : Item_Type;
         Next : Link_Type;
      end record;

   type Stack_Type is
      new Ada.Finalization.Limited_Controlled
      with record
         Items : Link_Type;
      end record;

   procedure Finalize (C : in out Cell_Type);
   procedure Finalize (S : in out Stack_Type);
end Stacks_G;

with Ada.Unchecked_Deallocation;
package body Stacks_G is
   procedure Free is
      new Ada.Unchecked_Deallocation
         (Link_Type, Cell_Type);
   -- storage management: one could choose a different
   -- scheme, e.g. a local free-list

   procedure Finalize (C : in out Cell_Type) is
   begin
      Free (C.Next);
   end Finalize;

   procedure Finalize (S : in out Stack_Type) is
   begin
      Free (S.Items);
   end Finalize;
   ... -- various operations
end Stacks_G;
```

Figure 6. *Per-object memory reclamation on a Stack ADT.*

Automated, immediate per-object memory reclamation could be named "garbage avoidance," or "immediate recycling," by contrast with garbage collection, which is required only under the condition that one first lose track of one's objects (and thus leave garbage around).

Note that, for non-limited controlled types, one cannot know whether Finalize is called for the death of the value of an object or for the death of the object itself, i.e. as part of an assignment operation or on storage destruction. Thus one cannot choose on that basis whether to return memory to the system or to a free-list (a managed pool). However, one may presume that the memory will be useful beyond the life of a few objects, and thus it makes sense to return the memory to the managed pool instead of deallocating it. Also next time the same scope is entered, the managed pool will still be around. Finally, the memory recycled in a managed pool will/can be returned to the system on scope exit *of the managed pool itself*. This leads to a two-tier model of storage management and memory reclamation in some applications: object- and type-based.

It can be seen from this example, that, ideally, there should be two distinct finalization operations for non-limited types: one called on value-death, and the other called on object-death.

5. REFERENCE COUNTING

It is sometimes advantageous to share structures between objects. In such a case, memory reclamation is somewhat more tricky; one needs to maintain reference counts to keep track of the number of "live" references to each (potentially) shared object.

From a data representation point of view, the ADT has at least one component of an access type, and the designated type includes a counter (the reference count, a Positive).

From a design point of view, reference counting should be used when structure sharing has clean semantics of referential integrity for the client of the ADT (e.g. the driver of a car is not a copy of a person, but that person herself), or, as an optimization of last resort, if copying a data structure is an expensive operation. Implementors of reference counting should watch out for cyclic structures: the self- or circular references should not be counted.

[Ros87] described a fairly complex, generic Ada 83 solution, with limited private types but without automatic finalization. The Ada 9X solution is straightforward and completely automatic. Since structure sharing arises with the assignment operation, we need to "Adjust" the latter. Our example shows strings of dynamic length, with structure sharing:

```
with Ada.Finalization;
package Shared_Strings is
   type String_Type is
      private;

   function "+" (S : String) return String_Type; -- conversion
   function "=" (L, R : String_Type) return Boolean;
   ... -- various operations
private
   type Shared_Cell is
      record
         Count : Positive; -- reference count
         ...    -- components for a String data structure
      end record;
   type Handle is access Shared_Cell;

   type String_Type is
      new Ada.Finalization.Controlled
      with record
         H : Handle;
      end record;

   procedure Finalize (S : in out String_Type);
   procedure Adjust (S : in out String_Type);
end Shared_Strings;


with Ada.Unchecked_Deallocation;
package body Shared_Strings is
   function "+" (S : String) return String_Type is
      ...; -- new Shared_Cell'(Count => 1, ... S ...)
   function "=" (L, R : String_Type) return Boolean is
      ...; -- (L = R) or else (L.H.S = R.H.S)
   ... -- various operations

   procedure Free is
      new Ada.Unchecked_Deallocation
         (Handle, Shared_Cell);

   procedure Finalize (S : in out String_Type) is
   begin
      if ( S.H = null ) then
      -- no heap memory is used
         return;
      elsif ( S.H.Count = 1 ) then
      -- no structural sharing, memory reclamation
         Free (S.H);
      else
      -- decrease of structural sharing
         S.H.Count := S.H.Count - 1;
      end if;
   end Finalize;

   procedure Adjust (S : in out String_Type) is
   begin
      if ( S.H /= null ) then
      -- increase of structural sharing
         S.H.Count := S.H.Count + 1;
      end if;
   end Adjust;
end Shared_Strings;
```

Figure 7. *Reference counting on shared dynamic strings.*

With reference counting of an ADT there is no risk of creating and using dangling access values, since there is no need for explicit deallocation.

In a system relying on a virtual memory subsystem, it may be preferable to optimize this scheme in order to avoid page thrashing when the reference count of a number of scattered items reaches 0. The solution lies in keeping the reference count in a memory area separate from that used by the shared cell (e.g. in an array, assuming an initially bounded number of shared cells). Thus we could substitute the following declarations in the private part of figure 7:

```
type Shared_Cell is
   record
      ...   -- components for a String data structure
   end record;
type Handle is access Shared_Cell;

Max_Shared_Cells : constant := 1_000;
type Index is
   range 1 .. Max_Shared_Cells;
Reference_Counts : array (Index) of Positive;

type String_Type is
   new Ada.Finalization.Controlled
   with record
      H : Handle;    -- to shared cell
      R : Index;     -- to reference count of the cell
   end record;
```

Figure 8. *Separating reference counts from shared cells.*

Reference counting may become a very expensive operation if the shared data is not very large. In such a case, it is better to completely avoid structural sharing, assuming compatibility with the semantics of the ADT.

Since the choice of implementation with or without structure sharing is completely hidden in the private part and package body, a client programmer can very easily switch from one version of an ADT to another. Also, the language guarantees that immediate memory reclamation and reference counting will properly be applied when a controlled ADT is used to match a generic formal private type or in a component definition of a composite type.

5.1. Garbage Collection vs. Reference Counting

Garbage collection is a technique used to automatically reclaim storage occupied by objects which have become inaccessible (because no live objects reference them any longer). As [Coh86a] points out, memory reclamation in general, and garbage collection in particular, is needed when a program is confronted by "the finiteness of storage." [Gol83] shows that memory reclamation for a Smalltalk system "can be accomplished either by reference counting or garbage collection."

Garbage collection is a delayed, periodic process; objects not referenced by live pointers occupy storage until they are detected by the collector. But, as we have shown, immediate reclamation can be achieved in Ada 9X, in a direct and natural extension of the programming of ADTs.

Garbage collection may have better average performance than reference counting, but it usually requires more memory (to collect information about live areas), and real-time systems rarely tolerate unpredictable and/or unexpected behavior.[4]

Finally, the interactions of garbage collection algorithms with concurrent programming (tasks and protected objects) and with virtual memory are still research subjects. The support for distributed systems introduced by Ada 9X will probably also lead to some interesting interactions with garbage collection and reference counting.

6. BREAKING STRUCTURAL SHARING

For most dynamic structures, one wants to avoid sharing between objects. Here, memory reclamation is easier, since one need not maintain any reference count.

From a data representation point of view, the issue of breaking structural sharing arises only if a non-limited ADT has at least one component of an access type. Since an assignment operation starts with a bitwise copy of the component access value, a new access value must be generated to designate a copy of the "deep" structure.

Since structure sharing arises with the assignment operation, we need to "Adjust" the latter in order to break the sharing. Our example shows strings of dynamic length, *without* structure sharing; observe that the public part of the specification is exactly the same as for strings *with* structure sharing (shown in the previous section):

```
with Ada.Finalization;
package UnShared_Strings is
   type String_Type is
      private;
   function "+" (S : String) return String_Type; -- conversion
   function "=" (L, R : String_Type) return Boolean;
   ... -- various operations
private
   type Handle is access String;
   type String_Type is
      new Ada.Finalization.Controlled
      with record
         H : Handle;
      end record;

   procedure Initialize (S : in out String_Type);
   procedure Adjust (S : in out String_Type);
   procedure Finalize (S : in out String_Type);
end UnShared_Strings;

with Ada.Unchecked_Deallocation;
package body UnShared_Strings is
   function "+" (S : String) return String_Type is
   begin
      return String_Type' (H => new String'(S));
   end "+";

   function "=" (L, R : String_Type) return Boolean is
   begin
      return (L.H.all = R.H.all);
   end "=";

   ... -- various operations
```

4. As [Ada86] explains, no garbage collection was required in Ada 83, because it was felt to be incompatible with real-time systems. The pragma 'Storage_Size was available, but Ada 83 had no requirement for memory reclamation. In Ada 9X, alternatives are offered with storage pools and controlled type.

```
    procedure Initialize (S : in out String_Type) is
    begin
      S.H := new String'("");
      -- access to null string avoids special-case
      -- handling in other operations
    end Initialize;

    procedure Adjust (S : in out String_Type) is
    begin
      S.H := new String'(S.H.all);
      -- break sharing introduced by bitwise copy of
      -- access value
    end Adjust;

    procedure Free is
      new Ada.Unchecked_Deallocation
        (Handle, String);

    procedure Finalize (S : in out String_Type) is
    begin
      Free (S.H);
    end Finalize;
  end UnShared_Strings;
```

Figure 9. *Breaking the sharing of dynamic strings.*

"Breaking the sharing" is a matter of adjusting assignment to achieve "deep" copy semantics. For instance, a non-limited Stack ADT implemented with a linked list would need an Adjust operation which made a copy of the whole list, so that the source and target of the assignment operation did not end up with a shared structure.[5]

7. TYPES AND THEIR POPULATION

It is sometimes necessary to manage some type-related state, called "class variables" in object-oriented parlance (in the OO-, and particularly Smalltalk-, world, "class variable" is opposed to "instance variable," which is technically known as a "record component" in Ada). This is easily done in Ada, since the package which exports an ADT may also hide various data structures, such as a free-list shared by all instances of a Stack ADT. With controlled types, it is of course possible to automatically maintain such type-related state during the initialization and finalization of objects belonging to the ADT, as well as for each assignment operation to such objects.

The following is an example of use, where a "class variable" is updated: i) when an object of the given class is created; ii) when an object is destroyed; and iii) before and after an object receives a new value. This example keeps a count of the number of existing rats:

5. The deep-copy could be immediate or delayed, i.e. it could happen on a "needs" basis. In the second case, the modifiers of the type would first do a deep-copy if the structure of the instance was marked as currently shared. The cost of delayed structure breaking is to check a flag each time a modifier is called. It is only of interest if multiple copies of an object need be stored and most of these copies are accessed then thrown away without modifying their structure—i.e. for some sort of "short-lived aliasing."

```
with Ada.Finalization;
package Rats is
  type Rat is
    private;
  ... -- various operations
  function Population_Size
    return Natural; -- number of existing rats
private
  type Rat is
    new Ada.Finalization.Controlled
    with record
      ... -- various components
    end record;

  procedure Initialize (R : in out Rat);
  procedure Finalize (R : in out Rat);
  procedure Adjust (R : in out Rat);
end Rats;

package body Rats is
  Number_of_Rats : Natural := 0; -- class variable
  ... -- various operations
  function Population_Size
    return Natural is
  begin
    return Number_of_Rats;
  end Population_Size;

  procedure Initialize (R : in out Rat) is
  begin
    Number_of_Rats := Number_of_Rats + 1;
  end Initialize;

  procedure Finalize (R : in out Rat) is
  begin
    Number_of_Rats := Number_of_Rats - 1;
  end Finalize;

  procedure Adjust (R : in out Rat)
    renames Initialize;
end Rats;
```

Figure 10. *Specification and implementation of privately controlled rats.*

The principle of the code shown in figure 10 is that each time a Rat object is created without initial value, the counter Number_of_Rats is incremented, and each time a Rat object is destroyed, the counter is decremented. For assignment operations, the counter is decremented before the bitwise copy, and subsequently incremented during adjustment.

These steps ensure that the counter accurately reflects the number of existing rat objects with their value, as the commented code of figure 11 demonstrates:

```
with Rats; use Rats;
procedure Population is                                  #rats = 0
  R1 : Rat;     -- Initialize (R1) called:               #rats = 1
  R2 : Rat;     -- Initialize (R2) called:               #rats = 2
begin
  R1 := R2;     -- Finalize (R1) is called:              #rats = 1
                -- followed by a bitwise
                -- copy of R2 into R1:                   #rats = 1
                -- and (last step of assignment operation)
                -- finally Adjust (R1) is called:        #rats = 2
```

```
end Population;
            -- Finalize (R2) is called:        #rats = 1
            -- Finalize (R1) is called:        #rats = 0
```

Figure 11. *How the rats are controlled.*

7.1. Keeping Track of a Population

During analysis, types and collections should be distinguished (and not lumped together with the "class" construct) [Eck94]. For an Ada implementation, an interesting question is how to automatically keep track of all instances of a type, i.e. how to have access to the collection of all objects belonging to a given type (subcollections being a fairly straightforward special case not covered here).

To achieve this we use a limited controlled type together with a class variable which acts as the head of a doubly-linked list of nodes. The principle of the code shown in figure 12 is that each time a Rat object is created, its Cell component is automatically linked into the global list of Cells. In addition, Rat is a (necessarily limited) type of self-referential objects: each Rat's Cell component has an access discriminant which designates the enclosing Rat itself. Note that this solution makes absolutely no use of dynamic allocation:

```
with Ada.Finalization;
package Collected_Rats is
   type Rat is
      limited private;
   generic
      with
         procedure Action (R : in Rat);
   procedure Iterate; -- iterate through all existing rats
   ... -- various operations
private
   type Cell;
   type List is access all Cell;          -- general access type
   type Cell (Me : access Rat) is         -- access discriminant
      new Ada.Finalization.Limited_Controlled
      with record
         Next, Prev : List; -- a node in a doubly-linked list
      end record;

   type Rat is
      limited record
         C : aliased Cell (Me => Rat'Access);
         ... -- various components
      end record;

   procedure Initialize (C : in out Cell);
   procedure Finalize (C : in out Cell);
end Collected_Rats;

package body Collected_Rats is
   Cells : List; -- head of the list of cells in the rats

   procedure Iterate is
      Scout : List := Cells;
   begin
      while ( Scout /= null ) loop
         Action (Scout.Me.all);
         Scout := Scout.Next;
      end loop;
   end Iterate;

   -- standard management of a doubly-linked list
   procedure Initialize (C : in out Cell) is -- Insert
   begin
      C.Next := Cells;
      Cells := C'Access;
      if ( C.Next /= null ) then
         C.Next.Prev := Cells;
      end if;
   end Initialize;

   procedure Finalize (C : in out Cell) is -- Remove
   begin
      if ( C'Access = Cells ) then
         Cells := Cells.Next;
      else
         C.Prev.Next := C.Next;
      end if;
      if ( C.Next /= null ) then
         C.Next.Prev := C.Prev;
      end if;
   end Finalize;
end Collected_Rats;
```

Figure 12. *Collecting rats.*

8. CONCLUSION

We have described controlled abstraction-oriented programming with Ada 9X through the study and application of supporting mechanisms: indefinite and controlled types. Ada 9X takes advantage of the existing Ada 83 features and extends them with well-integrated basic mechanisms; these extensions remain faithful to the philosophy of Ada 83.

User-defined assignment was clamored for by many Ada users. Ada 9X offers user-defined control; it is a semantically clean and simple building-block, and it provides enormous expressive power, including the possibility of customized assignment. In conjunction with indefinite types, controlled types should allow Ada programmers to design and code safe, efficient, and highly reusable ADTs.

Some non-orthogonalities still remain in the language, especially linked to the Midas touch of limitedness.

8.1. Ada 9X, Abstract Data Types, and Software Engineering

"Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. This revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on <u>reliability</u>, <u>maintainability</u>, and <u>efficiency</u>." — Ada 9X Reference Manual, Design Goals [Ada94a] (emphasis added)

For a software engineer, the three concerns stressed in the above quote are proper standards of evaluation for any new language construct. Thus, with regard to the support for the implementation of ADTs we have described:

- *Reliability*: the specification can be made truly representation-independent—without the blind use of limited types—and memory leaks can be systematically guarded against; type composition, by aggregation or inheritance, preserves the properties of ADTs.
- *Maintainability*: parameterized initial values and tailored assignment remove the need for multiple, slightly different, versions of ADTs; they also allow for more extensive and intensive information hiding.
- *Efficiency*: there is no distributed cost, and real-time programmers need not fear the unexpected dynamic activity usually associated with garbage collection.

As this paper shows, the controlled implementation of ADTs with Ada 9X provides components which are: efficient, simple to use, protected against misuse, reusable, and tailorable (five reusability criteria listed in [Coh86b]). Further, controlled ADTs are orthogonal to composition, and portable by virtue of relying exclusively on the standard definition of the language (two more criteria discussed in [Gar86]).

8.2. Some Brief Notes on a Desirable Programming Style for Abstract Data Types and on the Value of Type-Control

The use of access types should be encapsulated with ADTs, in order to achieve automatic memory reclamation. Memory reclamation can guarantee the absence of memory leaks in an ADT, without resorting to limited types or incurring the distributed cost of garbage collection. In addition, storage pools can be used to implement different models of storage management.

In short, client programmers should rely as much as possible on stack allocation and as little as possible on heap allocation, always striving for the encapsulation of dynamic data structures with ADTs. This approach reduces the number of places where memory allocation and deallocation take place; ideally each memory operation should have its own, unique, encapsulated place to facilitate understanding and analysis of dynamically allocated data structures.

8.3. Future Directions of Research

There are many additional directions for future research on the potential uses of controlled types, such as using them for debugging, collecting statistics, interfacing/implementing persistent objects, automated clean-up measures in tasks or packages, smart references, and garbage collection. Also, the formal specification of ADT control and of storage models should prove useful for the reuse of ADTs.

Acknowledgments. The author would like to thank Stéphane Barbey, Didier Buchs, Gabriel Eckert, Robb Nebbe, and the referees for their careful reading of earlier drafts of this paper.

9. BIBLIOGRAPHY

[Ada83] *Reference Manual for the Ada Programming Language.* ANSI/MIL-Std-1815a, 1983.

[Ada86] J.D. Ichbiah, J.G.P. Barnes, R.J. Firth, and M. Woodger. *Rationale for the Design of the Ada Programming Language*. 1986.

[Ada90] *Ada 9X Requirements (Ada 9X Project Report).* Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, Dec. 1990.

[Ada94a] *Programming Language Ada: Language and Standard Libraries (Draft Version 5.0).* ISO/IEC DIS 8652. Ada 9X Mapping/Revision Team, Intermetrics, Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138, MA, USA, June 1994.

[Ada94b] *Rationale for the Programming Language Ada: Language and Standard Libraries (Draft Version 5.0).* ISO/IEC JTC1/SC22 WG9 N 207. Ada 9X Mapping/Revision Team, Intermetrics, Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138, MA, USA, June 1994.

[Ada94c] *Programming Language Ada: Language and Standard Libraries (Annotated Draft Version 5.0).* ISO/IEC JTC1/SC22 WG9 N 193. Ada 9X Mapping/Revision Team, Intermetrics, Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138, MA, USA, June 1994.

[AQS92] *Ada Quality and Style: Guidelines for Professional Programmers (Version 02.01.01).* Software Productivity Consortium, Dec. 1992 (3rd ed.).

[Bak91] H.G. Baker. *Structured Programming with Limited Types in Ada: Nesting is for the Soaring Eagles.* ACM Ada Letters, Vol. XI, No 5, July-Aug. 1991, pp. 79-90.

[Bak93] H.G. Baker. *Safe and Leakproof Resource Management using Ada 83 Limited Types.* ACM Ada Letters, Vol. XIII, No 5, Sep.-Oct. 1993, pp. 32-42.

[Bar93] J. Barnes. *Introduction to Ada 9X*. Ada-Europe 1993 Tutorial, Paris, France.

[Bei92] J. Beidler. *Relaxing the Constraints on Ada's limited private Types Through Functional Expressions.* ACM Ada Letters, Vol. XII, No 2, Mar.-Apr. 1992, pp. 57-61.

[Boo87] G. Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings, 1987.

[Coh86a] N.H. Cohen. *Ada as a Second Language*. McGraw-Hill, 1986.

[Coh86b] S.G. Cohen. *Reusable Software Parts and the Semi-Abstract Data Type.* In *Proceedings of the First International Conference on Ada Programming Language Applications For The NASA Space Station,* ed. by R.L. Bown, Houston, Texas, June 2-5, 1986, pp. G.3.2.1-G.3.2.2.

[Eck93] G. Eckert and M. Kempe. *Modeling with Objects and Values: Issues and Perspectives.* Technical Report No 93/29, Software Engineering Lab., CS Dept., Swiss Federal Institute of Technology in Lausanne, Nov. 1993.

[Eck94] 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.

[Gar86] A. Gargaro and T. Pappas. *Understanding Ada Software Reusability Issues for the Transition of Mission Critical Computer Resource Applications.* In *Proceedings of the Ada-Europe International Conference,* ed. by P.J.L Wallis, Edinburgh, Scotland, May 6-8, 1986, pp. 105-114.

[Gen89] C. Genillard, N. Ebel, and A. Strohmeier. *Rationale for the Design of Reusable Abstract Data Types Implemented in Ada*. ACM Ada Letters, Vol. IX, No 2, Mar.-Apr. 1989, pp. 62-71.

[Gol83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Goo85] J.B. Goodenough. *On defining "=" in Ada.* ACM Ada Letters, Vol. IV, No 4, Jan.-Feb. 1985, pp. 27-31.

[Gut77] J. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM, Vol. 20, No 6, June 1977, pp. 396-404.

[Hen85] F.W. von Henke, D.C. Luckham, B. Krieg-Brueckner, and O. Owe. *Semantic Specification of Ada Packages.* In *Proceedings of the Ada International Conference*, ed. by J.G.P. Barnes and G.A. Fisher Jr., Paris, France, May 14-16, 1985, pp. 185-196.

[Joe93] J. Joergensen. *A Comparison of Object-Oriented Features of Ada and C++*. In *Proceedings of the Ada-Europe International Conference*, Paris, France, June 14-17, 1993.

[Kow87] R. Kownacki and S.T. Taft. *Portable and Efficient Dynamic Storage Management in Ada*. In *Proceedings of the 1987 ACM SIGAda International Conference on the Ada Programming Language*, Boston, Massachusetts, Dec. 9-11, 1987, pp. 190-198.

[Lla84] A. Llamosi, P. Botella, and F. Drejas. *On Unlimited Types and Reliability of Ada Programs.* ACM Ada Letters, Vol. IV, No 1, July-Aug. 1984, pp. 50-60.

[Luc84] D.C. Luckham and F.W. von Henke. *An Overview of Anna, a Specification Language for Ada.* IEEE Software, Vol. 2, No 2, Mar. 1985, pp. 9-22. Also in *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota, Oct. 15-18, 1984, pp. 116-127.

[Mac82] B.J. MacLennan. *Values and Objects in Programming Languages.* SIGPLAN Notices, Vol. 17, No 12, Dec. 1982, pp. 70-79.

[Men87] G.O. Mendal. *Storage Reclamation Models for Ada Programs*. In *Proceedings of the 1987 ACM SIGAda International Conference on the Ada Programming Language*, Boston, Massachusetts, Dec. 9-11, 1987, pp. 180-189.

[Nis84] J. Nissen and P. Wallis. *Portability and Style Guide in Ada*. Cambridge Press, 1984.

[Rob92] S. Roberts. *Difficulties in developing re-usable software components arising from the lack of user redefinition of standard assignment.* ACM Ada Letters, Vol. XII, No 4, July-Aug. 1992, pp. 36-41.

[Ros87] S.M. Rosen. *Controlling Dynamic Objects in Large Ada Systems.* ACM Ada Letters, Vol. VII, No 5, Sep.-Oct. 1987, pp. 79-92.

[Sch81] R.L. Schwartz and P.M. Melliar-Smith. *The Finalization Operation for Abstract Types.* In *Proceedings of the 5th International Conference on Software Engineering,* San Diego, California, Mar. 9-12. 1981, IEEE Computer Society, pp. 273-282.

[She82] M. Sherman, A. Hisgen, and J. Rosenberg. *A Methodology for Programming Abstract Data Types in Ada.* In *Proceedings of the AdaTEC Conference on Ada*, Arlington, Virginia, Oct. 6-8, 1982, pp. 66-75.

[Str92] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1992 (2nd ed.).

[Taf93] S.T. Taft. *Ada 9X: From Abstraction-Oriented to Object-Oriented*. In *Proceedings of OOPSLA'93*, ed. by A. Paepcke, Washington, D.C., Sep. 26-Oct. 1, 1993, ACM SIGPLAN Notices, Vol. 28, No 10, Oct. 1993, pp. 127-136.

[Vli94] S. Van Vlierberghe. *Memory Management in Ada 83 and Ada 9X*. ACM Ada Letters, Vol. XIV, No 4, July-Aug. 1994, pp. 43-57.