ACM *Ada Letters* **XIII**,5 (Sept./Oct. 1993), 32-42.

Safe and Leakproof Resource Management using Ada83 Limited Types

Henry G. Baker

Nimble Computer Corp., 16231 Meadow Ridge Way, Encino, CA 91436 (818) 986-1436 (818) 986-1360 (FAX)

Safe, *leakproof* and *automatic* resource managers are essential to the implementation of every embedded system, yet the standard examples of Ada83 resource managers are either *unsafe*: they risk allocating the same resource for two different purposes, *leaky*: they risk permanently losing track of some resources, or *non-automatic*: they require explicit deallocation, which allows for a host of opportunities for single-point system failures. Nevertheless, it is possible to construct a safe, leakproof and automatic resource manager—at least for systems with only a *single task*—by a careful combination of certain features of Ada83, and a careful avoidance of other features. We illustrate our scheme with a safe, leakproof and fully automatic dynamic-string package. The same techniques also work for an arbitrary precision arithmetic package [Fisher83] and for managing the "roots" of a real-time garbage-collected heap [Baker78] [Baker91SP] [Baker92Tread].

Suggestions are offered to improve the use of limited private types in Ada9X for automatic, safe and leakproof resource management.

1. Introduction

Complex embedded systems require dynamic resource management to achieve flexibility. However, unless these resources are carefully controlled, the resource management process itself may become a significant source of system failures. If, for example, the same resource is accidently allocated for two different purposes at the same time, the inadvertent simultaneous use of the resource by the two different parties can result in total chaos, leading to immediate system failure. If, on the other hand, some portion of the resource becomes misplaced, then the system can fail due to resource "starvation". Although the first case is clearly worse than the second, resource starvation will quickly cause system failure in very dynamic systems where the turnover of a resource exceeds the amount available.

It is not enough, however, that the individual resource management primitives are properly constructed; their use by the system must be capable of reclaiming resources *automatically* when they are no longer in use.[1] This is because as the size and complexity of a system grows, the likelihood also increases for the situation where a program which depends upon explicit deallocation either forgets to deallocate the resource, or worse, deallocates it twice, which may later lead to double allocation. Of course, there exist schemes for detecting some of these errors [Mendal87], but most of these schemes work only at run-time, when it may be too late. Furthermore, any scheme capable of recognizing *every* erroneous situation of this type has the same overhead as would a fully-automatic deallocation scheme. Finally, the overheads for fully-automatic resource management are not nearly as expensive as many assume. Thus, in a large and complex system, it becomes prudent and efficient to utilize fully-automatic resource deallocation, so that a much greater level of safety which can be achieved, coupled with a relatively small increase in overhead.

2. Fully Automatic Resource Management

One of the crucial issues to be faced in the building of a fully-automatic manager of dynamically-allocated resources in Ada83 is how to design the interface to the system in such a way that it is efficient, yet *safe* and *leakproof*. Such a manager is *safe* if it does not give access to the same resource to two different customers, and it is *leakproof* if it manages to eventually recover all resources when they can no longer be accessed by their current owner. *Safety* is clearly a primary concern, since the attempted use of a resource by two different customers for different purposes at the same time will almost certainly lead to disaster. That a resource manager is *leakproof* is also important, since the amount of the resource to be allocated is necessarily limited; otherwise, there would be no need for management. If resources are allowed to leak away, then the system will eventually fail from resource starvation.

In [Baker91SP], we described one safe and leakproof scheme for allocating a certain scarce resource—a *root* of a garbage-collected heap. Because the safety and leakproofness of the entire garbage-collected heap depends critically upon the allocation and automatic deallocation of the roots of the heap, we used extraordinary efforts to design a safe and leakproof manager for these roots. Unfortunately, the scheme described in [Baker91SP] is still not totally safe, because it depends upon the program avoiding certain programming practises *which, while deemed by Ada to be "erroneous", are not checked by either the compiler or the run-time system.*

For example, an aliasing of **`IN OUT`** parameters can lead to an unsafe condition in the scheme of [Baker91SP] (see [Baker93Steal] for details). While this aliasing is considered "erroneous" by the Ada83 standard, it is nearly impossible to check for, and therefore most Ada83 systems don't bother to check for it. Since one of the primary

[1][Mendal87] states "there is no universal method by which an Ada program can itself automatically detect that its data will no longer be referenced". This paper and [Baker91SP] show that Mendal was needlessly pessimistic.

purposes of *automatic* resource management is in the context of very large and/or complex systems where manual resource management becomes too risky, we feel that the dependence of our previous scheme on proper programming practices is not reasonable. In other words, not only do we not want to rely on error-free programmers, we want a resource manager which cannot be compromised even by a malicious programmer.

There were two other problems with our previous scheme besides its lack of complete safety. It required a stilted programming style in which nested functional expressions were not allowed, but were replaced by sequences of procedure calls. It also required the universal use of **`IN OUT`** parameters for objects of the managed type, and the efficiency of this parameter-passing mechanism left much to be desired in the Ada83 implementations we tested. We were thus led to the consideration of a new style of programming with Ada83 limited private types which could be efficient, safe and leakproof, in addition to being completely automatic.

3. Ada83 Limited Private Types

Ada83 *limited* types are types for which assignment ("`:=`") and aggregates are not allowed; equality ("=") is also not predefined for such types, but it can be defined by the programmer. Since initialization is considered by Ada83 to require assignment, a user cannot himself initialize either a new variable or a new constant object of the type, but must rely on whatever default initialization is already provided by the designer of the limited type.

An Ada83 limited *private* type is a private type with the additional restriction of being a limited type, as well. Such a limited private type is limited either because it incorporates limited components into its representation type, or because the designer of the private type wants additional control over the proliferation of the values, objects and variables of the type.

The designer of an Ada83 *private* type already has complete control over the bit-patterns that are used to represent the type; every bit pattern of the type is produced and interpreted by operations of the package which defines the private type. Unlimited private types are most useful for datatypes whose objects are timeless *values*—e.g., complex numbers or quaternions. Since such objects do not carry any internal state, there is no problem with implementation issues such as parameter passing "by reference" versus parameter passing "by copy". The lack of any services such as *finalization* is also not a problem for such datatypes.

The designer of an Ada83 datatype whose instances have internal state, and for whom the consistency of this internal state is paramount, must use a *limited* private type to gain more control. For example, the designer of the datatype may wish to control all new *variables* of the type to be those under the complete control of the package; this control can be achieved through the use of a default initialization that is *guaranteed* to produce an error. Since variables defined outside the package cannot use explicit initialization, they are initialized with the default initialization expression, which then causes a run-time error when executed. Thus, variables of the type cannot be defined outside the defining package. Of course, the defining package itself can always initialize its own variables with *explicit* initialization, and therefore avoid the error-causing *default* initialization.

The designer of a limited private datatype can deny to the user any *variables* of the type, if it exports no variables of the type and does not otherwise make such variables available—e.g., through generic units [Baker91SP]. If all operations of the type take only **`IN`** parameters, then the user is provided with a system of private *values* only, with no means to produce new values, and no means to "squirrel" away values.

4. A Safe and Leakproof Resource Manager

We now consider an example of resource management that is easy to understand—the management of dynamic strings. Ada83 built-in strings are difficult to use, and are inefficient for long strings, due to the copying which must be performed. Standard Ada83 strings are instances of an "unconstrained" type `STRING` which has severe restrictions on its use—e.g., no *variables* of the unconstrained type `STRING` can be defined, although variables of a specific fixed string length can be defined. Thus, although unconstrained parameters of type `STRING` can be used by procedures and functions, and function results of type `STRING` can be produced, these `STRING` values must eventually be stored into variables of some more constrained type. Most Ada83 textbooks show how to implement a more dynamic string mechanism using an access type to a discriminated record with a vector of characters as a component.

Unfortunately, these textbooks do not show how the *memory* for these dynamic strings can be managed in an automatic, safe and leakproof manner. In particular, the too-early deallocation of a string can lead to an unsafe situation, and it is trivial to leak storage if a string is not explicitly deallocated. Thus, the improper use of these dynamic strings has the potential to crash a mission-critical system, and is therefore not recommended.

We now describe below a *completely automatic* manager of dynamic strings which cannot be compromised by a user who attempts to deallocate a string too early, or forgets to deallocate it at all, because the user is not given any deallocation operation at all!


```
package dstrings is
 type dstring is limited private;     -- The type of dynamic string values.
 type dstring_var is limited private; -- The type of dynamic string variables.
 epsilon: constant dstring;           -- An empty dynamic string.
 function "="(x,y: dstring) return boolean;       -- Test for string equality.
 function length(x: dstring) return natural;      -- Get length of string.
 function image(x: dstring) return string;        -- Coerce string value.
 function make_dstring(s: string) return dstring; -- Make a new dyn. string.
 function "&"(x,y: dstring) return dstring;       -- dstring concatenation.
 function c(v: dstring_var) return dstring;       -- Contents of dstring var.
 procedure assign(v: dstring_var; x: dstring);    -- dstring var. assignment.
 pragma inline("=",length,image,c,assign);        -- Make these efficient.
 package g is
  generic with procedure f(v: dstring_var); -- Define a dstring variable.
  package prog is end;
  generic with function f(v: dstring_var) return dstring; -- Define dstring
  function let return dstring;              -- variable in valued block.
  pragma inline(let);                       -- Almost always called one time.
 end g;
private
 type idx is range 0..32767;
 type dstring_rec(l: natural) is            -- Don't want/need default here.
  record
   g: idx;               -- reGistration of lowest non-var binding, else ∞.
   r: idx;               -- count of dstring_var References.
   s: string(1..l);      -- the actual String of characters.
  end record;
 type ads is access dstring_rec;
 function die return ads;
 type dstring is record i: ads := die; end record;
 epsilon: constant dstring := (i=>new dstring_rec'(0, 0, 0, ""));
 function die return idx;
 type dstring_var is record i: idx := die; end record;
 end dstrings;
with unchecked_deallocation;
package body dstrings is
 type vector_ads is array(idx) of ads;
 vars: vector_ads := (epsilon.i, others => null);
 vidx: idx := 0;                            -- The index to the vars.
 dont_make_dstring_variables_this_way: exception;
 function die return ads is
  begin raise dont_make_dstring_variables_this_way; return null; end;
 function die return idx is
  begin raise dont_make_dstring_variables_this_way; return 0; end;
 procedure release(i: idx) is
  procedure kill is new unchecked_deallocation(dstring_rec,ads);
  x: ads renames vars(i);
  begin
   if x/=null and then x.r=0 and then x.g=idx'last then kill(x); end if;
  end release;
 function register(x: ads) return ads is
  begin
   if x/=null and then x.g>vidx then
    vidx:=vidx+1; vars(vidx):=x; x.g:=vidx; end if;
   return x;
  end register;
 pragma inline(release,register);
```

```
function "="(x,y: dstring) return boolean is
 begin
  return x.i=y.i or else (x.i/=null and then y.i/=null and then x.i.s=y.i.s);
 end "=";
function length(x: dstring) return natural is
 begin
  if x.i=null then return 0; else return x.i.l; end if;
 end length;
function image(x: dstring) return string is
 begin
  if x.i=null then return ""; else return x.i.s; end if;
 end image;
function make_dstring(s: string) return dstring is
 r: constant ads := new dstring_rec'(s'length,idx'last,0,s);
 begin return (i=>register(r)); end;
function "&"(x,y: dstring) return dstring is
 r: constant ads := new dstring_rec'(x.i.l+y.i.l,idx'last,0,x.i.s&y.i.s);
 begin return (i=>register(r)); end;
function c(v: dstring_var) return dstring is
 begin return (i=>register(vars(v.i))); end;
procedure decr(i: idx) is
 x: ads renames vars(i);
 begin if x/=null then x.r:=x.r-1; release(i); end if; end;
pragma inline(decr);
procedure assign(v: dstring_var; x: dstring) is
 begin
  if x.i/=null then x.i.r:=x.i.r+1; end if; decr(v.i); vars(v.i):=x.i;
 end assign;
procedure unwind(oidx: idx) is
 begin
  while vidx>oidx loop
   declare x: ads renames vars(vidx);
   begin
    if x/=null and then x.g>oidx then x.g:=idx'last; release(vidx); end if;
   end;
   vidx:=vidx-1; end loop;
 end unwind;
pragma inline(unwind);
package body g is
 package body prog is
  oidx: constant idx:=vidx; nidx: constant idx:=oidx+1;
  begin
   vidx:=nidx; vars(nidx):=null; f((i=>nidx));
   unwind(nidx); decr(nidx); vidx:=oidx;
  exception when others => unwind(nidx); decr(nidx); vidx:=oidx; raise;
  end prog;
 function let return dstring is
  oidx:= constant idx:=vidx; nidx: constant idx:=oidx+1; v: ads;
  begin
   vidx:=nidx; vars(nidx):=null; v:=f((i=>nidx)).i;
   if v/=null and then v.g>nidx then
    v.g:=nidx; unwind(nidx); decr(nidx); vidx:=nidx; vars(nidx):=v;
   else unwind(nidx); decr(nidx); vidx:=oidx; end if;
   return (i=>v);
  exception when others=>unwind(nidx); decr(nidx); vidx:=oidx; raise;
  end let;
 end g;
end dstrings;
```



```
with dstrings,text_io; use dstrings,text_io;
procedure main is
 procedure main_body(v1: dstring_var) is
  function q(v2: dstring_var) return dstring is
   begin
    assign(v1,make_dstring("q1") & make_dstring("q2"));
    assign(v2,c(v1));
    return make_dstring("q3") & make_dstring("q4");
   end q;
  pragma inline(q);
  function let_q is new g.let(q);
  begin
   assign(v1,let_q);
   put("c(v1)="); put(image(c(v1))); new_line;
   assign(v1, make_dstring("hi") & make_dstring("there"));
   put("c(v1)="); put(image(c(v1))); new_line;
  end main_body;
 pragma inline(main_body);
 package p is new g.prog(main_body);
 begin
  put_line("main: finished");
 end main;
```

There are many points in the above code that deserve discussion. First, we note that our `DSTRINGS` package exports *two* limited private types—one type for the dynamic strings themselves, and one type for dynamic string *variables*. Most books and papers recommend exporting only a single type, and then using mode **`IN OUT`** for variables of the private value [Mendal87] [Kownacki87]. We cannot recommend this approach, however, because Ada83 steadfastly refuses to require the passing of mode **`IN OUT`** parameters of a limited type "by reference". Since a "copy-in, copy-out" implementation of mode **`IN OUT`** can be used outside the defining package to destroy the integrity of the resource protected by a limited private type [Baker93Steal], and since we aspire to the greatest level of safety, we are forced to emulate "by reference" parameter passing ourselves for our dynamic string variables.[2]

Once we have a separate type for dynamic string *variables*, we will need a separate coercion to extract values from these variables; the function `C` (for "contents of") is provided for this purpose. We also need an assignment function to assign string values to string variables; the procedure `ASSIGN` is provided for this purpose.

Before discussing the two generic units `PROG` and `LET`, we first discuss the string representation. Our dynamic strings consist of a record discriminated by the length of the string, and have three additional components—a *registration number*, a *reference count*, and the vector of characters for the string. This discriminated record is not itself the limited private type, but an access type to this record "should be" the limited private type. We say "should be", but access types have a default initialization of `null`, and we would like to designate a non-`null` default initialization expression. We can achieve this in Ada83 only by wrapping a record around this access type, so that the record component can have an explicit default initialization. Our default initialization calls upon the package-defined function `DIE`, which is guaranteed to raise an exception. Therefore, we achieve our goal of disallowing the user of our package to declare any variables of type `DSTRING` himself, as these are certain to fail when they are elaborated; we only wish that there were some way of causing a *compile*-time error in this situation.

The dynamic string *variables* provided by our package will all be allocated from a vector `VARS` of dynamic string pointers, which is indexed by non-negative integers. Since these indices uniquely identify the dynamic string variables, we would like to use this index as the representation of the variable type itself. We could do this by making the index a *derived type* of the positive integers, but since this representation would not allow us to define our own default initialization, we again wrap a record around the index to construct the dynamic string *variable* type. As in the case of the `DSTRING` *values*, we guarantee that the user cannot himself create his own variables of our dynamic string variable type by providing a default initialization that is guaranteed to fail.

Although they are not strictly necessary for our dynamic string type, we have shown how we can define *constants* of the type in the defining package by showing how to define a zero-length string constant `EPSILON`.

---

[2]We note that Algol-68 incorporated the equivalent of Ada83 *types* and *modes* into a unified notion. The appearance in Ada9X of "transparent" access types ameliorates a basic flaw of Ada83, but also indicates that Ada83 should have followed Algol-68 [van Wijngaarden77] more closely in this regard.

Since our new type is a *limited* private type, we can define our own equality predicate ("=") for it. Of course, we must remember to check for `null` access values. The definition of concatenation ("&") requires some comment. Besides allocating and constructing the value to be returned, we must also *register* the value in our vector of pointers to dynamic strings. This registration will allow us to enumerate all of the strings which are referenceable by the user's Ada program. We use the `VARS` stack level at the time of allocation as a convenient registration number. If a `DSTRING` is returned to a "lower" level, this registration number is updated to a lower value. If a `DSTRING` is currently accessible through a `DSTRING_VAR`, but *is otherwise inaccessible by the Ada program*, this registration number is set to +∞ (infinity), meaning that it can be reclaimed if its `DSTRING_VAR` reference count goes to zero.

We now come to the meat of our `DSTRINGS` package—its generic units. These generic units are defined within an inner package `G`, rather than being defined at the same level as the other exported procedures, because Ada83 allows for the `G` package to be renamed, but does not allow for generic units themselves to be renamed. Since many customers of the package may not want to "use" the package, but to locally rename some of its "operations", this inner package `G` makes it easier for "generic renaming" to be performed.[3]

We provide two different generic units which allocate and make string *variables* available to the user's program. The first—`PROG`—is a generic *package*[4] which takes a procedure as a formal parameter. This procedure parameter is provided by the user, and it encompasses that portion of the user's program that manipulates the new string variable which is passed to it as a parameter. The use of `PROG` can be seen in the `MAIN` example, where the procedure `MAIN_BODY` is defined which accepts a new string variable as an argument, and then gives it the name `V1`. The reason for this roundabout way of providing new string variables is that not only can the `DSTRINGS` package *initialize* any new string variables in this way, but it can also *finalize* any new string variables; furthermore, the package can allocate these variables from a resource totally under its control. Finalization is possible, because when the user is finished with the variable, it returns from the procedure `MAIN_BODY`, either via a normal return, or via an exception, but in either case, the new variable will be finalized before its scope is exited. *So far as we know, this is the only programming technique which allows for the finalization of variables in portable Ada83*.

`PROG` operates by allocating the next slot in the `VARS` vector, and initializing it to `null`. `PROG` then calls the user-supplied procedure with the index of the newly allocated variable as an argument. If the user-supplied procedure exits normally, then `PROG` calls `UNWIND` to deregister any values at the top of the `VARS` stack and then `DECR` to finalize the variable it created before exiting normally. If the user-supplied procedure raises an exception, the exception is caught by `PROG`'s universal exception handler, which also calls `UNWIND` to deregister the top of the `VARS` stack and then `DECR` to finalize the variable, but then re-raises the same exception again. Thus, we can rely on the top items of the `VARS` stack being deregistered and finalized, so long as either the normal or the exceptional code is executed. If a catastrophic error occurs which terminates the exception-handling code, then we assume that the system has already failed, so the inability to recover storage at this point is probably the least of the system's problems.

We note that `PROG` deregisters and finalizes more than the one *variable* that it allocated on the top of the `VARS` stack. Any values given to the `PROG` body via `MAKE_DSTRING`, "&" or `C` are also registered on the `VARS` stack, if they have not already been registered at a lower level. Any values registered at this level were either created at this level, or made available to the user's program for the first time at this level. Since `PROG` cannot return a value, it is possible for a value registered at this level to escape to a higher level only through `ASSIGN`, which is also defined by our package. Since the user cannot perform an assignment "behind our back"—both because he cannot use `:=` assignment on a limited type, and because he cannot define variables outside of the package in which to hide these values—we know that any values registered at the current level of the `VARS` stack which also have no references from `DSTRING_VAR`'s are now completely inaccessible outside the `DSTRINGS` package, so `PROG` can finalize them.

Since we would like to share `DSTRING` values—i.e., an assignment is a simple assignment of access values, not a string copy—we utilize a *reference count* on each string that is allocated by the `DSTRINGS` package. This reference count will at all times indicate the number of references to the string from `DSTRING_VAR`'s. Thus, in order to implement the `ASSIGN` assignment operation, we must first increment the reference count of the value being stored, then decrement the reference count of the value being replaced, and then perform the assignment. The increment must be done before the decrement to allow for the situation where `ASSIGN`(x,C(x)) is called—i.e., the value being stored is the same as the value being replaced. If the only reference to the value happened to be the variable itself, and if the

---

[3]Some might advocate putting each separate generic unit into its own individual package to allow each to be separately renamed. A naming convention of this kind would put the "real" name on the package, with the perfunctory name "`G`" on the generic itself.

[4]It would be more symmetrical to make `PROG` a generic procedure, but this requires the additional overhead of actually calling the procedure, while the package executes during its elaboration.

decrement were performed first, then the decrement routine could deallocate the value before it realized that a reference to it was being saved back again [Kownacki87]. Incrementing before decrementing, on the other hand, always works.

The other generic unit defined by `DSTRINGS` is the generic `LET` *function*. The generic `LET` function is similar to the generic `PROG` package in that it takes a user subprogram formal parameter and passes a new string variable to it. However, the subprogram formal parameter to `LET` is a *function* which returns a value of type `DSTRING`. Furthermore, `LET` is itself a generic *function*, whose returned value is also of type `DSTRING`. `LET` captures the value returned by the user's function formal parameter and returns this value as its own value, but only after calling `UNWIND` and `DECR` to clean up the `VARS` stack. The rationale for `LET` is for use within a user function which returns a value of type `DSTRING`, but which also performs a number of `DSTRING` allocations within the function. Since the only values that can escape from such a function are values that are transferred by means of a `ASSIGN` assignment or by means of the returned value, we would like to deregister any other intermediate values that may have been created or used inside the function. By using `LET` within such a function, all values which are allocated within the user's function can be deregistered, except for the one being returned, and thus we can achieve more prompt deregistration for intermediate values. Below is an example of `LET`'s use.

```
function "**"(x: dstring; n: natural) return dstring is
 -- "Exponentiate" a string to the power n.
 function expt_body(v: dstring_var) return dstring is
  begin return x&(x**(n-1)); end expt_body;
 pragma inline(expt_body);
 function let_expt_body is new g.let(expt_body);
 begin
  if n=0 then return epsilon;
  elsif n=1 then return x;
  else return let_expt_body; end if;
 end "**";
```

The reason why the last line of this function is not simply "`return x&(x**(n-1))`" is that the recursive calls to "`**`" produce intermediate results which become garbage after the larger string has been produced. By using `LET` in this way, we allow the `DSTRINGS` package to deallocate these intermediate results more quickly, since we have no assurances that the outer caller of "`**`" will himself call `PROG` or `LET` to allow these intermediate results to be deallocated. We note that in this instance, we did not actually use the extra variable `V` which `LET` provided us.

The particular generic units `PROG` and `LET` are shown only as prototypes, since a "library-quality" string package would provide a wider variety of `PROG`- and `LET`-type constructs which are more closely tuned to a user's needs. For example, we might provide a whole series of `PROG` generic units—`PROGV0`, `PROGV1`, `PROGV2`, ...—which define zero, one, two, ... variables for the subprogram parameter to use. Similarly, we might also provide a whole series of `LET` generic functions—`LETV0`, `LETV1`, `LETV2`, ...—which define zero, one, two, ... variables for the function parameter to use. While this multiple-variable functionality can be achieved by a composition of the `PROG` and `LET` already provided here, the allocation of multiple variables can be more efficient at run-time. In an Ada system where generic units are implemented by means of macro-expansion, the host of these specific generics has no run-time cost, because generics which are not used do not appear in the executable image of the program.

5. Correctness

We do not provide a formal proof of correctness for the `DSTRINGS` package above, but merely mention some issues that are involved in such a proof. *The comments in this section refer to a serial (single-task) system only.*

A `DSTRING` is accessible if it is accessible by some Ada reference or if it is referenced by a `DSTRING_VAR`'iable. Therefore, a `DSTRING` may not be reclaimed until it has `DSTRING_VAR` reference count of zero *and* it is not referenced by any Ada name. References from `DSTRING_VAR`'s are controlled by reference counting, while references from Ada names are controlled by registration. Any reference given to Ada is registered, and that registration is good until the current scope is exited. If a reference is returned out of a scope, it is given a new, more permanent registration for the wider scope. Note that the management schemes for the two kinds of references are completely independent, although we implement them with the same stack for convenience.

`DSTRING` values are allocated in only three places—constant initialization, `MAKE_DSTRING`, and "`&`"—and in each case the allocated record is initialized with a reference count of zero, and the `DSTRING` is registered on the `VARS` stack. Thus, the reference count is initialized correctly to count the number of `DSTRING_VAR` references to the record. A `DSTRING_VAR` reference from the `VARS` stack can be modified in only two ways—`ASSIGN` and `DECR`. As discussed above, `ASSIGN` preserves the number of references, because it increments one count and decrements another. The `VARS` stack is only contracted when slots hold no values, so references are conserved by contraction.

Both PROG and LET remember the initial size of the VARS stack with a *local constant*, which is then used to determine how far to unwind at the end of the user's subprogram. Thus, even if the user's subprogram makes a mess of the environment, the value of this local constant will still be correct, so the stack can be unwound properly.

6. Pragmatic Issues

For a representative Ada83 implementation, DSTRING'SIZE is 32 bits, indicating that the record "wrapper" on the access type does not require any additional space. Furthermore, DSTRING_VAR'SIZE is 16 bits, as most applications are well-served with a string reference stack having 16 bit indices.

In our testing, the small overheads of managing the reference counts and the VARS stack are swamped by the costs of allocating and deallocating the string records themselves.[5] Thus, unless allocation, and especially deallocation, can be performed *en masse*, then our scheme does not require significant overhead over and above the normal costs of performing dynamic allocation and deallocation.

The costs of the "wrapper records"[6] necessary to ensure "birth control" for our limited private types are quite significant, however. If we convert both DSTRING and DSTRING_VAR to use derived types instead of a wrapped record type, we almost double the performance on a simple loop which tests the C and ASSIGN functions. Since most of this improvement comes from the representation change to the DSTRING limited type, and since the existence of user-created DSTRING variables is more of an irritation than a real problem, we suggest "unwrapping" the DSTRING type for higher performance. We do not recommend, however, that the DSTRING_VAR type be unwrapped, as user-defined variables of this type undermine the safety of the entire scheme.

7. Rough Spots

Although we have achieved our goal of developing a safe and leakproof programming style in Ada83 which also allowed nested functional composition, this development has come at the cost of requiring that all variables of the managed type be allocated by means of a generic unit. Furthermore, the user must be wary of loops with a large number of iterations, because storage can only be recovered at PROG and LET boundaries. Therefore, it may make sense to convert an iteration into a recursion solely for the purpose of using an intermediate PROG or LET.

Although our programming style allows the use of nested functional notation, the efficient evaluation of *common subexpressions* presents a problem. Consider the problem of evaluating the expression `h(f(x),g(f(x)));` `f(x)` is evaluated twice. We could avoid one evaluation using the following sequence: `assign(fx,f(x)); return h(c(fx),g(c(fx)))`. However, this use of assignments solely for common subexpressions is both inefficient and annoying. If DSTRING were not a limited type, we could have utilized the following code:

```
declare fx: constant dstring := f(x); begin return h(fx,g(fx)); end;
```

However, due to Ada83's confusion over the difference between a variable—a container for a value—and a value itself [MacLennan82] [Baker93ER], Ada83 mistakenly thinks that the ":=" in a "constant initialization" is a form of assignment, when it is just another form of *renaming*, only for values rather than for variables. We are therefore forced to use the following subterfuge, *which works in Ada83 even for limited types*:

```
declare
 function helper(fx: dstring) return dstring is begin return h(fx,g(fx)); end;
 pragma inline(helper);
 begin return helper(f(x)); end;
```

Another problem for a user of our dynamic strings is that he cannot put these strings into his own arrays and records. The reason for this is obvious—we cannot afford to lose track of any of these dynamic strings, and if the user were allowed to incorporate them into his own data structures, and then lose track of those data structures, then some of our dynamic strings would be lost.

If the user is intent on incorporating safe dynamic strings into complex data structures, then he will have to modify the DSTRINGS package to provide automatic management of those data structures, as well. Of course, the user is now faced with the problems of building a complete garbage collector, so that even long chains of data structure references can be traced to determine accessibility. In short, the automatic storage management of dynamic strings requires the automatic storage management of any data structures which can incorporate or point to these objects.

A final rough spot is the bunching of deallocations at the end of a PROG or a LET, which could negatively impact the response time in a real-time system. This problem occurs because an arbitrarily large number of allocations could

---

[5]Our implementation suppressed many subtype range checks for accessing the VARS stack and manipulating the reference counts.

[6]These are not the same as "rapper records"—e.g., by "Ice-T".

take place after the beginning of the most current `PROG` or `LET`, but before its end.[7] When the finalization code for the `PROG` or `LET` is executed, it must examine all of these allocations to see which ones require deallocation. During this time, no additional user code will be run. Of course, it would be possible to handle interrupts during this time, and perhaps even perform additional allocations, so long as these interrupts were perfectly nested in a static priority manner, so that the LIFO stack semantics of our `VARS` stack would be preserved. However, since it is unreasonable to rely on this kind of behavior in an Ada83 implementation, we do not recommend this approach.

8. Parallelism

The `DSTRINGS` package described in this paper is only for use in a serial, single-task system. The reason is that several important resources—e.g., the stack and stack pointer—are not protected from multithreaded access. But more important is the fact that the basic premise of the package—that one could tell when no further references to a record remained—is violated in an Ada83 program with multiple tasks, because the portions of the `VARS` stack for several different tasks can get intertwined, in which case the `VARS` stack cannot be correctly managed as a LIFO stack. What is required in this case is a "tree-shaped" ("cactus") stack, where the correct branch of the stack is addressed by the current task identifier. If such task identifiers are readily available; if a tree-shaped `VARS` "stack" is constructed; and if the accesses of this "stack" are protected by concurrency controls, then a `DSTRINGS` package useful for a multi-task environment can be constructed.

9. Other Applications

The dynamic string scheme described above is actually quite similar to the mechanisms already used by some existing Ada83 compilers for managing "unconstrained" objects, although the pointers for those objects are "below the surface" and are not seen by the user's program.

In addition to the management of strings, our scheme can be used for the automatic management of other dynamic objects for which functional notation is attractive—e.g., unlimited precision integers ("bignums"), unlimited precision rational numbers (e.g., for use within an Ada83 compiler), large precision software floating point numbers, etc. Many algebraic constructs—e.g., polynomials, rational functions, matrices and vectors, can be automatically managed using a similar scheme. In these cases, traditional nested functional notation can be used without the user having to worry about the complexities of storage management for the objects which can vary dramatically in size.

The most sophisticated use of these ideas is for the management of the *roots* of a real-time garbage-collected heap. The *roots* of a heap are the traditional interface between the programming language variables and the anonymous objects in the heap. Every object in the heap is accessible if and only if it is accessible from some root, either directly, or by some finite chain of pointers. In particular, no program variable can point directly into the heap without going through a root, and the roots themselves can be enumerated by the heap manager.

Our scheme relaxes this constraint by allowing certain program constant names to point directly into the heap, but only if the heap object is also *directly* pointed at by a *constant* root. If we can guarantee that the set of objects referenced by program names is always a subset of the set of objects pointed at directly by the roots, then our garbage-collected heap will be safe and leakproof. It will be safe, because no program name or variable can "squirrel away" a reference to an object, which may then be deallocated, after which access through that name or variable could cause a system failure. The heap will be leak-proof if inaccessible objects can be eventually reclaimed, and our garbage-collected heap is allowed to reclaim objects which are not accessible from some root.

The root management scheme described above cannot be used for a heap manager which uses a copying garbage collection implementation, because such a manager would need to update *all* references into the heap, not just those referenced by the roots. Unfortunately, Ada83 makes it impossible to build such a relocating heap manager.

10. Suggestions for Improving Limited Private Types in Ada9X

In the process of developing this technique for automatic resource management in Ada83, we have encountered a number of small and/or irritating misfeatures of Ada83 that could easily be fixed in Ada9X without negatively impacting existing programs. The following comments are grouped in no particular order.

• The protection from unwanted copying and assignment of Ada83 limited types is severely compromised by Ada83's allowance of "copy-in, copy-out" semantics for `IN OUT` parameters of these types (see [Baker93Steal] for an in-depth study of this problem). As we have discussed above, the insecurities of potential copying due to `IN OUT` parameters mean that *mode `IN OUT` parameters are completely useless* in an Ada83 secure limited private type. The semantics of the built-in task and file types already use "by reference" parameter passing to avoid chaos, so that same

---

[7] We note that our previous scheme presented in [Baker91SP] could be more prompt in deallocating, since its stack expanded and contracted in strict synchrony with the main task's stack. The ability to return values of the managed type makes such synchrony impossible in the current scheme.

security should be offered to limited types defined by the user. In particular, Ada83's *requirement* that limited private types whose representation types are "scalars"—e.g., integers and access types—use copy-in, copy-out semantics is completely non-sensical.

• The proposed *finalization* feature of Ada9X might seem to make our scheme of using generic instantiation of new variables obsolete. This should be especially true in conjunction with the ability to offer access variables to any Ada object, rather than just to objects dynamically allocated from a storage pool. Nevertheless, our scheme offers a level of control not possible with simple finalization—e.g., the ability to distinguish the handling of finalization under normal scope exit from the handling under exceptional scope exit. Furthermore, our scheme also allows more control over where the storage for a new variable is allocated, while the finalization of Ada9X presumes that storage is still locally allocated in the current stack frame. On the other hand, Ada9X will presumably offer the ability to finalize not only user-visible program variables, but also compiler-generated temporaries—e.g., the intermediate values of a nested functional expression. In addition, the Ada9X ability to provide access types to local variables can be used to solve the problem of automatic storage management in the context of multiple tasks.

• As we have shown with the `helper` function above, Ada83's inability to initialize a constant object of a limited type does not prevent this functionality, because it can be emulated by means of parameter passing to a local function. However, this irritating restriction does greatly compromise the readability of a program. In the same vein, Ada83's irritating restriction on not allowing `IN` formal objects of a limited type is also non-sensical, since the same effect can easily be achieved through `IN` parameters to a subprogram.

• The inability of Ada83 to rename a generic unit is just plain silly. While it is always possible to wrap a renamable package around every generic, this severely clutters the code for no redeeming value. Ada9X should allow a generic unit to be renamed (not just the *instances* of the unit).

• As we have seen above, the ability to deny the user of a private type the ability to declare variables of the type can be quite useful. Unfortunately, the mechanism we have chosen works only at run-time, not at compile-time. One can usually get a compile-time error by providing an initialization value which does not belong to the subtype, but this warning comes during the compilation of the package defining the private type, *not* during the compilation of the unit declaring the variable. We have suggested elsewhere that Ada9X provide a "`:=<>`" default initialization, which will cause a *compile*-time error if a variable is declared without an explicit initialization. Such a facility would be useful even for non-private types.

• The inability in Ada83 to export from the defining package of a private type a generic unit which takes a formal object of the private type is a very irritating restriction which appears to be due to a nasty pun in the Ada83 standard which makes formal objects appear to be real objects, at least in their syntax. Since real objects of the private type cannot be exported from the defining package due to a combination of other silly and irritating rules of Ada83, formal objects are also verboten. At the same time that Ada9X removes the restrictions on generic formal IN parameters, Ada9X should also eliminate this restriction, so that more powerful generic units can be exported by the package defining a private type.

• The extension of our dynamic resource manager to handle the situation with many tasks will require that the task manager efficiently obtain the Task ID of the executing task. This Task ID should be a discrete type, so that the Task ID can be used to index into different pools of resources for different tasks. In this way, each task-specific resource pool can be initialized with resources obtained "at wholesale", so that during any task-specific allocation, *the appropriate task-specific pool can be utilized without the need for any locking*. If, on the other hand, the allocation of every resource—however small—required accessing a global lock, most of the benefits of parallelism would be lost.[8]

11. Conclusions

We have shown how safe, automatic storage management can be performed in Ada83. Our scheme can be reasonably efficient on an implementation which supports the "inlining" of functions passed to generics as well as of generic subprograms themselves, and which implements the passing of scalars wrapped in records as efficiently as it passes normal scalars. Our handling of references which escape to Ada has certain similarities to the scheme proposed in [Baker92Cons] as well as to certain "generational" garbage collection schemes [Lieberman83] [Ungar84].

We have extended the scheme used here into a management scheme for the roots of a garbage-collected Lisp system written in Ada83 [Baker89]. Although generational garbage collectors are often more efficient than reference-counting collectors, we have found that the additional cost of performing both reference counting *and* tracing garbage collection in Ada83 is only marginally greater than the cost of performing simple tracing garbage collection. Furthermore, reference counting recovers significantly more temporary garbage than the generational scheme we tested.

[8] The task-specific pools of resources are reminiscent of the "private bottles" kept by the bartender for each customer in "dry" states in the Southern United States.

12. References

[Ada83LRM] *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Office, Wash., DC, 1983.

[Baker78] Baker, H.G. "List Processing in Real Time on a Serial Computer". *CACM 21*,4 (April 1978), 280-294.

[Baker89] Baker, H. "The NIMBLE Project—Real-Time Common Lisp for Embedded Exprt System Applications". *Proc. 1989 AIAA Computers in Aerospace Conf.*, Monterey, CA, 1989.

[Baker91SP] Baker, H.G. "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles". ACM *Ada Letters XI*,5 (July/Aug. 1991), 79-90.

[Baker92Tread] Baker, H.G. "The Treadmill: Real-Time Garbage Collection without Motion Sickness". ACM *Sigplan Not. 27*,3 (March 1992), 66-70.

[Baker92CONS] Baker, H.G. "CONS Should not CONS its Arguments, or, a Lazy Alloc is a Smart Alloc". ACM *Sigplan Not. 27*,3 (March 1992), 24-34.

[Baker93ER] Baker, H.G. "Equal Rights for Functional Object, or, The More Things Change, The More They Are the Same". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

[Baker92LLL] Baker, H.G. "Lively Linear Lisp—'Look Ma, No Garbage!'". ACM *Sigplan Notices* 27,8 (Aug. 1992), 89-98.

[Baker93Steal] Baker, H.G. "How to Steal from a Limited Private Account—Why Mode `IN OUT` Parameters for Limited Types *Must* be Passed by Reference". ACM *Ada Letters XIII*,3 (May/June 1993), 91-95.

[Beidler92] Beidler, John. "Relaxing the Constraints on Ada's `limited private` Types through Functional Expressions". *Ada Letters XII*, 2 (Mar/Apr 1992),57-61.

[Fisher83] Fisher, Gerry. "Universal Arithmetic Packages". ACM *Ada Letters III*,6 (1983),30-47.

[Harms91] Harms, D.E., and Weide, B.W. "Copying and Swapping: Influences on the Design of Reusable Software Components". *IEEE Trans. SW Engrg. 17*,5 (May 1991), 424-435.

[Kieburtz76] Kieburtz, R.B. "Programming without pointer variables". *Proc. Conf. on Data: Abstraction, Definition and Structure, Sigplan Not. 11* (spec. issue 1976), 95-107.

[Kownacki87] Kownacki, R., and Taft, S.T. "Portable and Efficient Dynamic Storage Management in Ada". *Proc. ACM SigAda Int'l. Conf. "Using Ada"*, Dec. 1987, 190-198.

[Lieberman83] Lieberman, H., and Hewitt, C. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *CACM 26*,6 (June 1983), 419-429.

[MacLennan82] MacLennan, B.J. "Values and Objects in Programming Languages". *Sigplan Not. 17*,12 (Dec. 1982), 70-79.

[Mendal87] Mendal, G.O. "Storage Reclamation Models for Ada Programs". *Proc. ACM SigAda Int'l. Conf., Ada Letters*, Dec. 1987, 180-189.

[Rosen87] Rosen, Steven M. "Controlling Dynamic Objects in Large Ada Systems". ACM *Ada Letters VII*,5 (1987), 79-92.

[STEELMAN78] Dept. of Defense STEELMAN requirements for high order computer programming languages. June, 1978.

[Taft91] Taft, S. Tucker, *et al*. *Ada-9X Draft Mapping Document*. Wright Lab., AFSC, Eglin AFB,FL, Feb. 1991.

[Ungar84] Unger, D. "Generation Scavenging: A non-disruptive, high performance storage reclamation algorithm". *ACM Soft. Eng. Symp. on Prac. SW Dev. Envs., Sigplan Not. 19*,6 (June 1984), 157-167.

[vanWijngaarden77] van Wijngaarden, A., *et al*. "Revised Report on the Algorithmic Language Algol 68". ACM *Sigplan Not. 12*,5 (May 1977), 1-70.

[Yen90] Yen, Mike. "Using a Dynamic Memory Management Package to Facilitate Building Lisp-like Data Structures in Ada". *Proc. AIDA-90*, Nov. 1990, 85-93.

