ACM *Ada Letters XI,5 (July/August 1991),79-90.*

Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles[1]

Henry G. Baker

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

Many traditional styles of programming cannot cope with the limitations of Ada limited private types. We present a style of programming that enables structured programming in Ada with limited private types. The problem of "finalization" is safely and cleanly handled through the use of a generic package which guarantees that a variable of the limited private type will be finalized even in the presence of exceptions.

Finalization is often desired in order to perform *automatic garbage collection*, whether by reference counting or by marking. We have proved our structured technique by developing a garbage-collected Lisp system in Ada in which the Lisp values are instances of a limited private type. The Ada code was so highly structured, that it was generated automatically using a simple compiler. The garbage collector in this system was easily and safely implemented, and different garbage collection techniques were tested without affecting user code.

A. INTRODUCTION

Private and limited private types have been advertised as Ada's versions of abstract data types. Many books and papers have extolled the virtues of these types, and given many examples of their construction. Ordinary private types retain all of the value transmission operations of Ada, and so they do not often constrain programming style; indeed, they tend to make normal structured programming styles even cleaner and more structured.

Limited private types (hereinafter called LP types), however, can severely cramp a programmer's style, since assignment is prohibited.[2] While the programmer of a package defining an LP type can also define an equivalent "assign" procedure, we claim below that the existence of an unrestricted assignment procedure destroys one of the major benefits of an LP type, and one may be better off in such a case using an ordinary private type.[3]

One of the major benefit of LP types is the possibility of providing *birth control* for the objects of these types. In other words, one can provide "object identity" [Baker93] for variables of these types in such a way that the defining package of the LP type controls the allocation and initialization of all useful objects of this type. (Due to the vagaries of the Ada language, one can declare variables of the LP type without informing the package defining the LP type, but since the package can guarantee that these variables are initialized to impotent values, these declarations become useless.)

Important primitive uses of LP types in Ada are *tasks* and *files*. Both tasks and files require significant operating system interaction, where the *identity* of the task or file object is more important than any values associated with the object. Through the use of the concept of an LP type, and the careful definition of the operations on the object, it becomes impossible to create a copy of the object which might result in ambiguity about the object's actual identity. If an Ada

---

[1]An obscure reference to [Clarke80]. As we shall see, *higher*-order (generic) constructs allow us to soar.

[2]The lack of assignment does not in and of itself cramp programming style; *functional* programming languages [Backus78]—including pure Lisp—provide quite elegant programming styles. Ada83, however, does not provide the necessary support for a completely functional style of programming.

[3]Some authors use LP types with such an "assign" procedure to allow the redefinition of the "=" operator; [Goodenough84] shows that this is not necessary, as "=" can be redefined for any type, albeit in a roundabout fashion.

implementation universally uses "by-reference" parameter-passing, then the object's identity becomes isomorphic to its address, because the limitations on LP types ensure that no copies can be made.

B. INITIALIZATION AND FINALIZATION

Since one of the major uses of LP types is to manage the proliferation of objects of the type, we address the problem of initializing and finalizing objects of an LP type. In order to ensure that an object is finalized before it becomes inaccessible, we must have a mechanism to ensure that "cleanup" code is executed whenever control leaves a certain scope—even if control is transferred as the result of an exception. *Common Lisp* [Steele90] provides a construct called `unwind-protect`, and a generalization called `wind-unwind` has been proposed for *Scheme* [Hanson84] [Haynes87]. Below we show an analogue of Scheme's `wind-unwind` in Ada.[4]

```
generic with procedure initialize;        -- initialization before main action
        with procedure action;            -- main action.
        with procedure cleanup;           -- cleanup after main action
package5 wind_unwind is end wind_unwind;  -- Look ma; no specifications!
package body wind_unwind is
begin
  begin
    initialize; action;                   -- initialize & then do main action
  exception when others => cleanup; raise; -- if exception, then cleanup
  end;                                    -- and then re-raise same exception
  cleanup;                                -- cleanup on normal exit.6
end wind_unwind;
```

We will now utilize our Ada `wind_unwind` generic package to emulate another useful construct which we call `init_fini`. This package declares a new variable of a given LP type (which must either be constrained or have defaults), and performs an initialization before, and a finalization after, a main body of code which uses this variable.

```
with wind_unwind;
generic type lp is limited private;
        with procedure init(x: in out lp);
        with procedure action(x: in out lp);
        with procedure fini(x: in out lp);
package init_fini is end init_fini;
package body init_fini is
    x: lp;                           -- Declare a variable of type lp.
    procedure initialize is          -- Initialize the lp variable.
    begin init(x); end;
    procedure main is                -- Perform the action on the lp variable.
    begin action(x); end;
    procedure cleanup is             -- Cleanup the lp variable.
    begin fini(x); end;
    package wu is new wind_unwind(initialize,main,cleanup); -- Execute all 3
end init_fini;
```

[4]Without first-class continuations, Ada can't emulate the more interesting features of `wind-unwind` [Haynes87].

[5]We here use generic *packages* instead of generic parameterless *procedures*, because they tend to be slightly more efficient in many implementations that do not support inlining of procedures. There are slight differences in the reporting of errors between the two implementations, however, since the sequence of statements of the package bodies are executed during elaboration.

[6]`cleanup` should also be capable of cleaning up after an incomplete `initialize`. Normal exit `cleanup` cannot be moved inside the exception block, because exceptions within `cleanup` should not go back to `cleanup`; exception loops could otherwise occur. In any case, these are the correct Common Lisp semantics.

We will now use our Ada `init_fini` generic unit to emulate another useful construct stolen from Common Lisp, `with-open-file`. The Lisp macro `with-open-file` provides an environment in which a variable is bound to an open file which can be used for I/O operations. This construct also provides automatically for the initialization and the finalization of the variable which is bound to the file, so that the user need not worry about closing the file. The `init_fini` generic unit assures that any file opened by `with-open-file` will be closed, even if exceptions are raised in the user's code.

```
with text_io; use text_io;
generic with procedure action(f: in out file_type); -- Code using the file.
        name: string := "";                         -- Name of file.
        mode: file_mode := in_file;                 -- Mode of file.
        form: string := "";                         -- Add'l options.
package with_open_file is end;

with init_fini;
package body with_open_file is
  procedure my_open(f: in out file_type) is         -- Need own open proc.
  begin
    case mode is
      when in_file  => open(f,mode,name,form);
      when out_file => create(f,mode,name,form);
    end case;
  end my_open;
  package oac is new init_fini(file_type,my_open,action,close);
end with_open_file;
```

This `with_open_file` example shows the generic use of our `init_fini` generic package, which can be similarly used for a whole host of operations which must be paired. For example, a frequent operation in a multiprogramming system is that of locking and unlocking a resource. While Ada provides a high-level tasking construct with a rendezvous capability, synchronization using this capability may not be efficient enough for the low-level locking operations of a shared-memory multiprocessor system. A "spin lock", like that below, may be more efficient.[7]

```
package spin_lock is
  type lock is limited private;
  generic with procedure action(l: in out lock);     -- Export only generics.
  package with_new_lock is end;
  generic with procedure action;
  procedure with_lock_locked(l: in out lock);
private ...
end spin_lock;

with init_fini,wind_unwind;
package body spin_lock is
  procedure new_lock(l: in out lock) is begin ... end;
  procedure kill_lock(l: in out lock) is begin ... end;
  package body with_new_lock is
    package nak is new init_fini(lock,new_lock,action,kill_lock);
  end with_new_lock;
  procedure lock_lock(l: in out lock) is begin ... end;
  procedure unlock_lock(l: in out lock) is begin ... end;
  procedure with_lock_locked(l: in out lock) is
    procedure initialize is begin lock_lock(l); end;
    procedure cleanup is begin unlock_lock(l); end;
    package wu is new wind_unwind(initialize,action,cleanup);
  begin null; end with_lock_locked;
end spin_lock;
```

---

[7]The *protected records* of Ada-9X [Taft91] may provide a more structured and portable means to the same end.

This lock example shows a package which defines an LP type and only two generic units, both of which are nesting constructs. By programming using these constructs, it is impossible to forget to unlock a lock, or to forget to kill a lock when it is done. In particular, since we have not exported any of the unmatched operations (e.g., `lock_lock` or `unlock_lock`), we are capable of only "properly parenthesized" orders of execution.

We must utilize a generic *procedure* instead of a generic *package* for `with_lock_locked`, because this operation requires a lock *object* parameter. Due to an unfortunate Ada83 pun, a "generic formal object" has the same syntax as a variable declaration, and since public variables of a private type cannot be defined within the private type's package, neither can generics which have "formal objects" as parameters.[8] This restriction puts generic units at a disadvantage relative to subprograms, which have no such restriction on their formal parameters, thus indicating that the restriction was a mistake.

C. NESTED EXPRESSIONS OF LIMITED PRIVATE OBJECTS

The ability to program with nested expressions is one of the advantages that higher-level languages have over assembly languages. Indeed, the word "FORTRAN" is an acronym for "FORmula TRANslator", because it translates nested mathematical formulae into sequences of machine commands. The ability to program with nested expressions depends on the compiler's ability to generate temporary variables ("temporaries") which hold the intermediate results of subexpressions of a formula [Aho86]. A nested expression in Ada could involve a subexpression whose type was that of an LP type, however, and a compiler might in this case generate a temporary variable to hold the intermediate value of this subexpression. Such an automatically-generated temporary would be out of the control of the package defining the LP type, and this variable would not be finalized. Thus, although there is no restriction in Ada on the use of functions which take arguments and return values of an LP type, we can not use these capabilities, because the package defining the LP type would then lose control over any intermediate results of the LP type.

Thus, the style of programming that uses nested expressions and "functions" such as `CONS`, which style is so dear to the hearts of Lisp programmers, is not safe in Ada, unless the Ada system incorporates a true garbage collector. There is, however, a closely related style of programming—widely used in Lisp and Scheme compilers—called "continuation-passing" [Steele78]. Continuation-passing style expands an expression into another expression in which all temporaries are explicit. Converting to continuation-passing style requires the ability to pass functional objects (functions and procedures) as arguments, however, and Ada83 does not support this capability.[9] However, many situations requiring functional arguments in other languages can be emulated in Ada83 through the use of generic units, which can take functions and procedures as arguments. We will therefore utilize Ada generic units in this manner to provide a structured programming methodology for LP types which totally controls the management of temporaries.

We first show the conversion of a simple arithmetic expression to Ada continuation-passing form to illustrate the style.

```
declare c: constant integer := a*a+b*b; begin <statements using c> end;
-- a,b defined outside this context.
-- this fragment converts into the following "continuation-passing" fragment
```

[8] Converting formal objects into generic procedure parameters is analogous to "currying" in the lambda calculus.

[9] This lack will be corrected in Ada-9X [Taft91].

```
package my_integers is
  type my_integer is limited private;
  function "="(x,y: my_integer) return boolean;
  generic with procedure cont(z: in out my_integer);
  procedure plus(x,y: my_integer);
  generic with procedure cont(z: in out my_integer);
  procedure times(x,y: my_integer);
  pragma inline(plus,times);
private
  type my_integer is new integer;
end my_integers;

package body my_integers is
  function "="(x,y: my_integer) return boolean is
  begin return integer(x)=integer(y); end;
  procedure plus(x,y: my_integer) is
    z: my_integer := my_integer(integer(x)+integer(y));
  begin cont(z); end plus;
  procedure times(x,y: my_integer) is
    z: my_integer := my_integer(integer(x)*integer(y));
  begin cont(z); end times;
end my_integers;

with my_integers; use my_integers;
declare          -- a,b defined lexically outside this fragment.
  procedure cont_a2(a2: in out my_integer) is
    procedure cont_b2(b2: in out my_integer) is
      procedure cont_c(c: in out my_integer) is
      begin <statements using c> end cont_c;    -- c used here.
      pragma inline(cont_c);
      procedure plus_c is new plus(cont_c);
    begin plus_c(a2,b2); end cont_b2;           -- c ":=" a2+b2 computed here
    pragma inline(cont_b2);
    procedure times_b2 is new times(cont_b2);
  begin times_b2(b,b); end cont_a2;             -- b2 ":=" b*b computed here
  pragma inline(cont_a2);
  procedure times_a2 is new times(cont_a2);
begin times_a2(a,a); end;                       -- a2 ":=" a*a computed here
```

While the above translation appears inscrutable, it can be seen to be assembly language, except that it must be read backward. We first instantiate `times` with a continuation `cont_a2` which will accept the product; we then call this instantiation to compute `a2=a*a`. We then instantiate `times` with another continuation `cont_b2` which accepts the product, and then call this instantiation to compute `b2=b*b`. Then we instantiate `plus` with a continuation `cont_c` which accepts the sum, and call this instantiation to compute `c=a2+b2`. Within `cont_c`, we perform a calculation with the argument `c`, and this calculation makes its result known by means of side-effects—output operations or assignments to more global variables. It might also appear that this translation is inefficient, but if an Ada compiler utilizes macro-expansion for generic units (virtually all do), and follows `pragma inline` advice, then the translated code should be just as efficient as the original!

The major reason for going to all this trouble is to allow "functions" such as a Lisp `CONS` to allocate new data structures without the possibility of losing track of them. If `CONS` is not a function at all, but a *generic unit* which takes as arguments a `CAR`, a `CDR`, and a continuation procedure, then `CONS` can control the fate of the cells it allocates (as well as the references to them) and passes as arguments to the continuation procedures.

Our translation so far has finessed the problem of the dynamic allocation of LP objects by continually calling another (generic) "procedure"—i.e., extending the stack. While this method can be used to simulate any computation through the use of an enormous stack [Fischer72], such a simulation is not a reasonable use of storage. Furthermore, we can only pass and call continuations using Ada generic units in certain restricted situations; we cannot avoid the problem of passing true continuations for recursive procedures, for example. Therefore, we must eventually face the issue of *returning* an LP value.

One approach to solving the value return problem is to convert every function into a procedure with one additional argument which is a variable into which the result should be placed. This approach has been used by FORTRAN compilers since the early 1960's, and is currently used by many C compilers. Once this conversion has been made, then the problem of returning a value becomes identical to the problem of *assigning* a value to a variable in an outer context. If we provided for a *safe* assignment procedure for the LP type, then our problem would be solved.

Providing a safe assignment procedure for a limited private type requires that the assignment procedure be capable of distinguishing variables which are under the control of the defining package from "fake" variables which have been defined by the programmer. This distinction is required because Ada provides no mechanism for restricting the user from defining variables of an LP type. The LP type can prevent any meaningful use of such illegitimate ("bastard") variables, however, by checking at run-time which variables were initialized using the default initialization (e.g., `null` for access types), and which were initialized through package-provided procedures. Assuming that such a test is incorporated into the defined assignment procedure,[10] then the assignment procedure can only be used to transfer values from one controlled variable to another controlled variable, and a mechanism such as reference counting can be used to keep track of which objects are no longer referenced by any of the controlled variables. Since all package-controlled variables will be properly finalized, references cannot be lost, and storage cannot *leak*.[11]

Another option open to the designer of the LP type is the provision of "write-once", or "logical" variables of the type. When a new package-controlled variable is created for the purpose of receiving a "function" result, for example, it is initialized to the "empty" state, and when this variable receives a value through assignment, it becomes "full". The assignment procedure can then be restricted to only assign to "empty" variables; it becomes erroneous to re-assign a variable. A properly constructed system of write-once variables has many similarities to certain functional languages (e.g., *I-structures* [Arvind89]) and to certain "concurrent logic" languages.

D. PROOF OF SAFETY

We now wish to prove that our structured LP types cannot be compromised, even by an unscrupulous user of the LP type, so long as he does not use `unchecked_conversion`. The proof is by computational induction, starting from the initialization of a variable of the LP type.

Basis. Variables of the LP type can only be created by the defining package, or through a user declaration. Because variables of the LP type cannot be explicitly initialized by the user, each must receive the default initial value provided by the package. So long as the defining package initializes its controlled variables to values distinguishable from the default initial value, then non-controlled variables can be detected.

Induction. We assume that all package-defined variables currently have legitimate values, while all user-defined variables have the default initial value. All parameters of the LP type of package-defined functions and procedures must use **`in out`** mode.[12] As a result, only explicit, named *variables* of the LP type are acceptable as arguments to these procedures; nested expressions returning values of the LP type are not acceptable as arguments, and hence compiler-generated temporaries are not acceptable as arguments. Furthermore, all package-defined procedures check all

---

[10]The programmer can often arrange for the compiler to automatically generate this test as a subtype constraint check.

[11]Assuming that garbage has no directed cycles of pointers.

[12]Ada83 unfortunately restricts parameters of *functions* to have only mode **`in`**, so only *procedures* can be so used.

of their arguments of the LP type to make sure that they do not have the default initial value. Package-defined procedures also do not assign any of their parameters to have the default initial value, hence no package-defined operation can result in a package-defined variable acquiring the default initial value. Since the user cannot assign a new value to any of his declared variables of the LP type, nor can any of his functions or procedures, his declared variables of the type can never change, and always retain their default initial value. QED

We note that a user can happily define as many variables of the LP type as he wishes—including components of user-defined records, arrays and access types. He can also define functions which take the LP type as arguments and return it as a value. Using these functions, he can create an unlimited number of copies of values stolen from legitimate controlled variables. These copied *values* are useless, however, as there is no way to get them into a *variable*. As a result, user-defined *functions* which return values of the type are completely useless, and user-defined *procedures* are simply complex compositions of package-defined operations, all of which thoroughly check their arguments for legitimacy.

We note that if our structured technique is used for the purposes of garbage collection, then one must be careful about what one does with copies of legitimate values. Suppose that the LP type defines Lisp-like variables which are capable of holding symbols, integers or lists. Suppose further that the defining package provides an operation to extract the "print-name" Ada string from a Lisp symbol. We now suppose that the programmer saves such a value, and then a garbage collection operation decides that the symbol is unreferenced, and therefore deallocates the string. The user now has a "dangling reference"!

We note further that even if an implementation uses "copy-in, copy-out" for `in out` parameters, programs which involve aliasing cannot defeat the safety of the LP type. Consider, for example, the operation `procedure p(x,y,z:in out lp) is begin x:=y; end`. If one inadvertently (and erroneously) calls `p(a,b,a)`, one may achieve either `a=b`, or a fancy no-op, depending upon the sequence of `out` updates. If `p` checks *all* of its LP arguments for legitimacy, however, then all of the `out`-assigned values will be legitimate ones. Thus, while the program may malfunction due to the aliasing, the safety of the LP type will not be compromised.[13]

E. AUTOMATIC GARBAGE COLLECTION

In order to prove our concept of structured programming with LP types, we built a simple portable Lisp interpreter with an automatic garbage collector[14] in Ada using the nested style discussed above. As we hoped, many implementation decisions, such as whether the storage was managed using reference counts or mark-sweep garbage collection were completely hidden within the defining package.[15] Interestingly, our interpreter does not use either `unchecked_conversion` or `unchecked_deallocation` (reclaimed storage is put back onto a package-managed "free storage list").

[13]The Ada83 indeterminacy regarding "by-reference" or "copy-in, copy-out" of `in out` parameters is inconsistent with Ada's stated goals of perspicuous code and trusted systems, since copy-out can cause havoc, and yet Ada83 is unwilling to force the detection of parameter aliasing, either through its type system or through run-time checks.

[14]The garbage collector is a non-copying variant of the "real-time" GC described in [Baker78].

[15]Unfortunately, a copying garbage collector requires that every reference be updated, which means that copy-in, copy-out semantics for **`in out`** parameters become unacceptable.

Since programming with explicit temporaries is such drudgery, we implemented a simple compiler to compile nested expressions with implicit temporaries into our structured Ada code with explicit temporaries. An example of actual generated Ada code is shown below. `cons`, `car`, `cdr`, `nullp` and `setq` are defined by the type manager package for the LP type `t_type`; names in upper case are found in the Lisp source code.

```
procedure REVERSE(result,L: in out t_type) is
 procedure REVERSE_BODY(t0: in out t_type) is begin
  declare -- labels
   procedure REVERSE1(result,L,R: in out t_type);16
   procedure REVERSE1(result,L,R: in out t_type) is
    procedure REVERSE1_BODY(t1,t2,t3: in out t_type) is begin
     if not nullp(L) then
      CDR(t1,L);                                -- t1:=cdr(l);
      CAR(t3,L);                                -- t3:=car(l);
      CONS(t2,t3,R);                            -- t2:=cons(t3,r);
      REVERSE1(result,t1,t2);17                 -- result:=reverse1(t1,t2);
     else
      setq(result,R);                           -- result:=r;
     end if;
    end REVERSE1_BODY;
    package p is new progv3(REVERSE1_BODY);     -- need 3 temporaries.
    begin null; end REVERSE1;
  begin -- labels body of REVERSE_BODY
   REVERSE1(result,L,t0); -- t0 already initialized to nil.
  end; -- labels body of REVERSE_BODY
 end REVERSE_BODY;
 package p is new progv1(REVERSE_BODY);         -- need 1 temporary.
 begin null; end REVERSE;
```

```
;;; The above Ada code was automatically generated (except for some comments)
;;; from the following Lisp code:

(defun reverse (l)
  (labels ((reverse1 (l r)
             (if l (reverse1 (cdr l) (cons (car l) r))
                 r)))
    (reverse1 l nil)))
```

The expansion ratio of 23:5 is about the same (~4X) as that reported by previous researchers. Since the nested style used by the translator is required by any Ada83 program which *safely* manages dynamic storage, and since our compiler generates this style in a structured way, it is not clear what advantage might be gained through "hand translating" or "re-engineering" the Lisp code into Ada. Lisp (with perhaps an ML-style type checker [Wand84]) should be reconsidered as a powerful "specification language" which compiles into an Ada "implementation".

We do not yet have good timing data on our Lisp/Ada system, because its efficiency depends critically on a few simple Ada compiler optimizations, which not all Ada compilers have. If the Ada compiler uses these optimizations, then our scheme should be very efficient, while if the Ada compiler does not use these optimizations, then it could be several times slower.

[16] Our compiler generates the specifications before generating the procedure bodies defined in `labels` expressions so that mutually recursive procedures will work correctly; for a single procedure, however, the specification is redundant.

[17] Our simple compiler still lacks a "tail-recursion" optimization [Steele78] which would transform `REVERSE1` into an iterative loop.

F. PREVIOUS WORK

This paper discusses work which is a continuation of [Baker90]. Our birth control scheme for objects of LP type bears a strong resemblance to Barnes'es scheme for controlling access to a resource [Barnes89,9.3]. Barnes does not export a nesting generic unit to handle deallocation, however.

[Cohen86] describes a technique which exports a generic unit for updating elements of an abstract collection in place. He does not use LP types, however, nor is he concerned about controlling the proliferation of references.

[Rosen87] uses LP types to define a generic storage management package. His package exports a "destroy" operation instead of a nesting generic unit, and his package must be instantiated with a non-limited type. Thus, although his goals are similar to ours in wanting a completely safe storage manager, his package is not completely safe. The user can forget to "destroy" objects, and the use of reference counts can allow cyclicly referenced garbage to "leak away".

Our simple Lisp interpreter uses representations quite similar to those of [Smith88]; his system uses a "descoping" procedure rather than our safer nesting generic unit to control proliferation of references. Our interpreter uses a generic `mapc` unit, which emulates Common Lisp `mapc`; the use of this generic unit is quite similar to [Hosch90] and [Musser89], although these authors are less concerned with safe storage management. Yen [Yen89] [Yen90] has also developed implementations of Lisp capabilities in Ada.

Mendal [Mendal87] and Kownacki [Kownacki87] both provide excellent discussions of storage management in Ada. Nevertheless, their managers require some help from the user, and utilize reference counts which can leak storage.

G. PROBLEMS WITH Ada83

There are several problems with the current Ada83 language which diminish the attractiveness of our structured style of programming with limited private types:

• Ada generic units cannot be *renamed*; therefore a generic unit defined within another package must always be referred to using qualified ("dot") notation, or made unqualifiably accessible through **`use`**. Since many experts decry the widespread use of **`use`** [Barnes89,p.150], this makes generic units "second class citizens" when one is attempting to "re-export" the operations (including generic operations) of an LP type. The current workaround is to put every generic unit into its own "wrapper" package, and use renaming on that wrapper, but this solution is hard on the eyes.

• Ada generic units are not *derived* along with operations of derived types. This limitation[18] closes off the possibility of using derivation to avoid the massive renaming of operations when re-exporting [Bardin88] (and renaming cannot be done using generic units, in any case).

• Ada generic units cannot be overloaded. This lack makes the *shadowing* of generic operations for simulating a single-inheritance hierarchy impossible [Perez88].

• Ada visible generic units in the defining package of a private type cannot have formal parameters which are objects of the private type. As a result of this restriction, a private type cannot export an operation which is implemented as a generic unit, and which requires objects, rather than types or subprograms, as parameters. This restriction is due not to any conscious desire on the part of the Ada83 designers for safety and security, but due to a nasty Ada83 syntax pun. The defining package of a private type cannot declare any "variables" of the private type until the full declaration of the type, which doesn't happen until the private part at the end of the specification. Unfortunately, generic formal objects have the syntactic appearance of a variable declaration, so

---

[18] Other non-operations—e.g., constants—are also not derived; workarounds exist for some of these, however. For example, constants can be defined as parameterless functions with pragma `inline`; they are derived along with other subprograms, and inlining achieves the efficiency of a normal constant.

they fall prey to this particular "gotcha" of Ada83. This restriction rules out the use of generic formal **in out** objects instead of procedure **in out** parameters to overcome the cache'ing behavior and the aliasing restrictions of the latter, at least for LP types, where the reference semantics of generic formal objects are sorely needed Ada-9X should allow the programmer to toggle between public and private during the specification of a private type, to allow the full type to become known earlier, so that type derivations, variables and generics may be declared in the public part of a private type specification.

• The current Ada83 coyness about specifying whether **in out** parameters are passed by reference or by "copy-in, copy-out" [Schwartz80] is pessimal for performance, in the sense that the portable programmer is forced into another level of indirection with access types (or vector indices) in order to force "by reference" behavior. For example, in order to implement a completely portable and understandable version of our garbage collection scheme using LP types, we must avoid all side-effects to the **in out** parameters, in which case no copy-out need ever be done. Procedures must return values, however, and this requires side-effects, which we implement by assigning to the *target* of the appropriate (read-only) access parameter. We can reliably only use **in out** mode for protecting object identity, but not for transferring data out of the procedure! Thus, Ada83 saddles the vast majority of non-distributed Ada code with significant inefficiencies in order to provide for the faint hope of efficient distribution. Yet "copy-in, copy-out" semantics for *variables* are not appropriate even for distributed systems (see [Baker93]).[19] Clean language semantics are more important to overall *system-level* efficiency than are efficiency "hacks" like "copy-in, copy-out". We therefore strongly suggest that Ada-9X **in out** parameters be defined in terms of **renames**—even for "elementary" types like integers and booleans (including packed booleans)—or failing that, Ada-9X should define a **ref** parameter mode.

• Although aliasing of **in out** parameters can cause havoc in Ada83 procedures using copy-in, copy-out semantics, Ada provides no mechanism for detecting such aliasing at run-time. The alternative of determining aliasing at compile-time is impossible without seeing all the callers of all procedures, so run-time detection is required. While one can compare the *values* of two in out parameters using "=", this does not decide whether they are really the *same* variable (i.e., aliased).[20] Ada either needs to provide an object identity predicate [Baker93] (perhaps called `EQ`) on formal parameters—both procedure and generic parameters—or eliminate the complexity of **in out** and **out** parameters altogether in favor of C++ style *references*, which can then be manipulated like any other access type.

• The Ada83 restriction on functions taking only parameters of mode **in** is the only remaining trace of an early failed attempt to eliminate side-effects from Ada functions. Ada83 functions *can* have side-effects, however, so this restriction is silly. There are serious issues regarding the order of evaluation of portions of expressions, but the inclusion of side-effects within functions can already be used to discover evaluation order. The lack of **in out** parameters to functions (which return non-controversial values like booleans or integers) complicates programming for no gain, whatsoever.

• *Anonymous functions and procedures* would enhance the readability and understandability of our nested style of programming. The concept of a nameless function or procedure is of limited usefulness in Ada83, where such an anonymous construct could only be used as an argument to a generic instantiation. In Ada-9X, however, anonymous functions and procedures will become considerably more useful with the ability to pass functions and procedures as arguments to other functions and procedures.

• *Anonymous packages* sound like a contradiction in terms, but as we have seen above, they can be quite useful as instantiated generics called only for effect.

---

[19]Ada83's "cure" (copy-in, copy-out) is therefore worse than the "disease" (aliasing).

[20]Ada83 provides the attribute `X'ADDRESS`, which can compare parameters—e.g., `X'ADDRESS=Y'ADDRESS`. However, with copy-in, copy-out semantics, parameters with different addresses could still be aliased.

• Ada-9X should allow non-corporeal `declare`'s. It should be possible to have a "`declare` <declarations> `end`" sequence for declarations with no bodies; i.e., without requiring `begin null`.

H. CONCLUSIONS

We have shown several Ada generic constructs which we have used to construct an structured style for programming with limited private (LP) types. These constructs provide a mostly nested programming style which solves the problem of getting these objects initialized and finalized. Finally, we have shown that this style can allow for the safe allocation and deallocation of dynamic storage.

Our method for using LP types to manage dynamically-allocated objects is more structured and less error-prone than previous methods [Mendal87] [Kownacki87], because our method cannot be compromised by the user of the LP type. While the code may appear involuted, it is quite structured, and can be easily read with practice. Due to its highly structured nature, there is less opportunity for the user to err by forgetting to finalize objects himself, and the resulting code is cleaner as a result of this finalization code having been removed.

Our method allows such tight control of objects of the LP type that one can preclude the use of these objects as components of composite objects. Of course, within the defining package one can provide for such composite objects either directly, or by exporting an appropriate generic unit. Only with this degree of control can advanced storage managers such as garbage collectors be safely programmed.

Some may criticize the appearance of the Ada code in our structured style. Given the structure and constraints of the Ada83 language, we believe that this appearance cannot be improved (other than with better comments and/or better formatting). One is then left with the quandry—code which is pleasing in appearance is unsafe, while code in our style is safe. In the final analysis, programming style should conform to the Bauhaus criterion "form follows function"; i.e., the elegance of a programming technique is correlated to its usefulness, not its outward appearance.

I. ACKNOWLEDGEMENTS

We appreciate the improvements in this paper suggested by Geoff Mendal of Systems Engineering Research Corporation and Carl Friedlander of the ISX Corporation.

J. REFERENCES

Ada83. *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Office, Wash., DC, 1983.

Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

Arvind, Nikhil, Rishiyur S., and Pingali, Keshav K. "I-Structures: Data Structures for Parallel Computing". *ACM TOPLAS 11*,4 (Oct. 1989),598-632.

Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". *CACM 21*,8 (Aug. 1978),613-641.

Baker, Henry. "List Processing in Real Time on a Serial Computer". *Comm. of the ACM 21*,4 (April 1978),280-294.

Baker, Henry. "The Automatic Translation of Lisp Applications into Ada". *Proc. 8'th Conf. on Ada Tech.*, Atlanta, GA (March 1990),633-639.

Baker, Henry. "CONS Should not CONS its Arguments, or A Lazy Alloc is a Smart Alloc". ACM *Sigplan Not. 27*,3 (Mar. 1992), 24-34.

Baker, Henry. "Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Bardin, Bryce M., and Thompson, Christopher J. "Using the Re-Export Paradigm to Build Composable Ada Software Components". *ACM Ada Letters 8*,2 (1988),39-54.

Barnes, J.G.P. *Programming in Ada: Third Edition*. Addison-Wesley, Reading, MA, 1989,494p.

Clarke, et al. "Nesting in Ada Programs is for the Birds". *Proc. ACM Symp. on Ada, Sigplan Not. 15*,11 (1980).

Cohen, Ellis S. "Updating Elements of a Collection in Place". *ACM Ada Letters 6*,1 (1986),55-62.

Collard, P. "Object-Oriented Programming Techniques with Ada: An Example". *ACM Ada Letters 9*,6 (Sept./Oct. 1989),119-126.

Fischer, M.J. "Lambda Calculus Schemata". *Proc. ACM Conf. on Proving Assertions about Programs, Sigplan Not. 7*,1 (Jan. 1972).

Goodenough, John B. "On defining "=" in Ada". *ACM Ada Letters 4*,4 (1984),27-31.

Hanson, Christopher, and Lamping, John. "Dynamic Binding in Scheme". Unpublished manuscript, 1984.

Harper, R., MacQueen, D., and Milner, R. "Standard ML". ECS-LFCS-86-2, Comp. Sci. Dept., U. of Edinburgh, March 1986,70p.

Harper, R., Milner, R., Tofte, Mads. "The Definition of Standard ML, Version 2". ECS-LFCS-88-62, Comp. Sci. Dept., U. of Edinburgh, Aug. 1988,97p.

Haynes, Christopher T., and Friedman, Daniel P. "Embedding Continuations in Procedural Objects". *ACM TOPLAS 9*,4 (Oct. 1987),582-598.

Hosch, Frederick A. "Generic Instantiations as Closures". *ACM Ada Letters 10*,1 (1990),122-130.

Kernighan, Brian W., and Ritchie, Dennis. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

Kownacki, Ron, and Taft, S. Tucker. "Portable and Efficient Dynamic Storage Management in Ada". *Proc. ACM SigAda Int'l Conf., Ada Letters*, Dec. 1987,190-198.

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

Perez, E.P. "Simulating Inheritance with Ada". *ACM Ada Letters 8*,5 (1988),37-46.

Rosen, Steven M. "Controlling Dynamic Objects in Large Ada Systems". *ACM Ada Letters 7*,5 (1987),79-92.

Schwartz, Richard L., and Melliar-Smith, Peter M. "The Suitability of Ada for Artificial Intelligence Applications". Final Report, Contract #AAG29-79-C--0216, SRI Int'l., Menlo Park, CA, May 1980,48p.

Smith, D. Douglas. "ALEXI—A Case Study in Design Issues for Lisp Capabilities in Ada". *Wash. Ada Symp. 5* (June 1988),109-116.

Steele, Guy L. *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization).* AI-TR-474, Artificial Intelligence Laboratory, MIT, May 1978.

Steele, Guy L. *Common Lisp, The Language; 2nd Ed.* Digital Press, Bedford, MA, 1990,1029p.

Taft, Tucker, *et al. [Ada-9X] DRAFT Mapping Document*. Ada-9X Proj. Rep., Feb. 1991.

Taft, Tucker, *et al. [Ada-9X] DRAFT Mapping Rationale Document*. Ada-9X Proj. Rep., Feb. 1991.

Wand, M. "A Semantic Prototyping System". *Proc. '84 Symp. on Compiler Constr., Sigplan Not. 19*,6 (June 1984),213-221.

Yen, Mike. "Adapting an AI-Based Application from its Lisp Environment into a Real-Time Embedded System". *Proc. AIAA Comps. in Aerospace VII,* Monterey, CA, (Oct. 1989),1114-1122.

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

