1-1-1986

Uses of Generics in Ada

Mark Sherman
Dartmouth College

USES OF GENERICS IN ADA

Mark Sherman

Technical Report PCS-TR86-104

Uses of Generics in Ada

Mark Sherman
Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH. 03755

October 1984

Abstract

This paper discusses how Ada generic procedures and packages can be used in novel ways to provide general initialization of records, expressions using discriminants, record field hiding and explicit control of parameter binding.

I. Why Worry About Using Generics in Ada

Although several languages contain features that resemble the generic facility of Ada (e.g., Clu [Liskov]), there is no general catalog of programming techniques for using generic features. Therefore programmers learning Ada cannot turn to experience with other languages to learn how to use Ada's generic features. Further, a review of some current Ada textbooks reveals a rather limited view of Ada's generic facilities: a means of adding type parameters to abstract data types and procedures [Barnes, Habermann]. In this paper, I claim that there are more uses for generics than stack packages, swap procedures and numerical integration functions.

The sections of the paper are organized around two strategies for using Ada's generic features: generalized records and parameter control. The next section outlines how generic packages can be viewed and used as a record. Section 3 discusses how generics can be used to control parameter ppassing. Section 4 summarizes the paper. The paper provides only the principles involved. An interested reader should study the references in Section 5 to see some complete examples of applying these principles.

II. Using Generics to Form Reords

II.1. A Simple Example

Generic packages in Ada are a direct analog to record types. With a slight change of syntax, they can be

interchanged almost exactly [Wegner]. A simple example will illustrate the similarities. Suppose we wish to save the information about an employee in a record. A typical set of declarations using a record would be:

```
--We assume declarations for Gender, Department_Code, and
--Benefit_Code appear elsewhere

type Employee (G:Gender) is record
        ID: Integer;
        Department: Department_Code;
        Benefit_Class: Benefit_Code;
        end record;

a: Employee(Male);
b: Employee(Female);

a.ID:=23894;
b.Benefit_Class:=Exempt;
```

A corresponding program fragment using generic packages would be:

```
generic
        G: in Gender;
package Employee is
        ID: Integer;
        Department: Department_Code;
        Benefit_Class: Benefit_Code;
end;

package a is new Employee(Male);
package b is new Employee(Female);

a.ID:=23894;
b.Benefit_Class:=Exempt;
```

The two fragments above illustrate the following correspondences between Ada's generic packages and Ada's records:

Record Features	Generic Package Features
Records	Generic Packages
Type Definitions	Generic Package Definitions
Record Components	Variables in Packages
Record Variables	Package Instantiations
Component Selection	Variable Selection
Discriminants	Generic Parameters

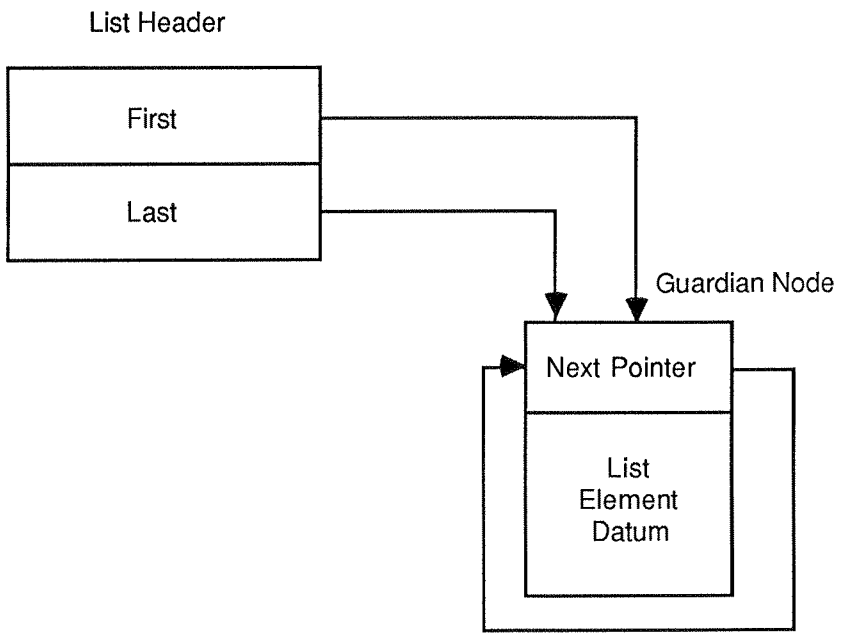
Although similar, there are important differences between record types and generic packages that will govern which feature a programmer should use for a particular problem. The next two sections discuss the advantages of each approach.

II.2. Advantages of Generic Packages over Records

There are three advantages of using generic packages instead of record types: sophisticated initialization, selective component hiding with procedure components and parameter expressions. Each advantage is discussed in turn.

II.2.1. Initialization

The initialization of components in a record declaration is limited in two ways: initialization occurs only once per elaboration of the type declaration and there is no convenient interaction between the initializations of separate components [ANSI]. These limitations can be illustrated by an implementation of a list that uses a guardian header with a circular, singly-linked list of nodes [Grogono], where the header should refer to the first and last elements of the list. Upon creation, the list header should denote an empty list, which in this representation is a single node pointed at by both the first and last fields of the record (as illustrated below):



A record declaration for the list header might be:

```
type List_Head is record
        First: Ref_Node:=Some_Init;
        Last: Ref_Node:=Some_Init;
        end record;
```

The first problem is that the declaration for *List_Head* will have its initialization expressions evaluated only once during eleaboration of the type definition. However, each list header should have its own guardian, not all list headers using the same one. Similarly, we would like both the *First* and *Last* components of the record to point at the same guardian. Without some obscure coding tricks, one could not be sure that the same access value would be returned by the two calls of *Some_Init*. By contrast, a generic package may include a package body that can contain statements to perform arbitrarily complex initialization. For example, the generic package corresponding to *List_Head* would be:

```
generic
package List_Head is
        First: Ref_Node;
        Last:Ref_Node;
end;

package body List_Head is
begin
        First:=Some_Init;
        Last:=First;
end;
```

Because the initialization statements in the package body are elaborated each time the package is instantiated, each list header "variable" will have its own guardian. Further, the requirement that *First* and *Last* refer to the same note is easily met by the statements in the package body.

II.2.2. Parameter Expressions

A second advantage of generic packages over record definitions is the ability to use, without restrictions, generic parameters in local declarations. The corresponding feature of records, discriminants, is restricted. Specifically, record definitions do not permit discriminants to be used in expressions of components. Thus

the following record declaration is erroneous:

```
type Queue(Queue_Size: Natural) is record
        LastElt: Integer:=0;
        Elts:array(0..Queue_Size-1) of integer;
end record;
```

To allow the discriminant *Queue_Size* to be used in an expression, a generic package declaration must be used:

```
generic
        Queue_Size: in Natural:=0;
package Queue is
        LastElt: Integer:=0;
        Elts: array (0..Queue_Size-1) of integer;
end;
```

II.2.3. Procedures and Selective Field Hiding

A third advantage of generic packages over record types is the ability to limit the programmer's access to components in a record. One may wish to record some performance, debugging or registry service in a record [Hilfinger]. As a simple example, suppose that each record contains a unique serial number for tracing purposes. If the serial number were a component in a record, there would be nothing that would prevent the serial number from being altered (constant components could not be initialized with unique serial numbers).

One approach for providing a protected serial number would be to place the variable that holds the serial number in a package body. Unfortunately, this approach would also prevent the user from seeing any components hidden in the body. If hidden components were used in tandem with functions, then a user could gain access to the hidden components by calling an appropriate access function. This is illustrated below:

```
generic
package Registered_Integer is
        -- This is a package that merely registers its use with a serial number.

        The_Integer: Integer;        --only "component" in record
        function Serial_Number return integer;
end;

package body Registered_Integer is

        Local_Serial_Number: Integer;

        function Serial_Number return Integer is
        begin
                return Local_Serial_Number;
        end;
begin
        --Last_Serial_Number is a global variable (could be
        --protected if necessary by placing it in a package or function).
        Last_Serial_Number:=Last_Serial_Number + 1;
        Local_Serial_Number:=Last_Serial_Number;
end;
```

Using the declarations above, a user could declare a "registered" integer as follows:

```
package My_Integer is new Registered_Integer;
```

The user could alter the *The_Integer* component like any other component but could only read the serial number by calling the *Serial_Number* function.

Once one admits the use of subprograms as "record components," then one can start viewing protected components as being more than simple values. For example, one can use functions to replace redundant information in records. A variable-length string illustrates the principle. If we assume that strings are represented as an array of characters that is terminated by a null character (ascii code 0), then we could use the following declaration for a string type:

```
type Var_String is record
        Chars: array (1..100) of character;
        Length: integer;
        end record;
```

The characters would be stored in the *Chars* array (terminated by a character with code 0). Every routine

that manipulates a variable-length string would be responsible for keeping the value of *Length* consistent with the characters stored in the array. A generic package could provide the same functionality in a simpler form by using a function as a record component:

```
generic
package Var_String is
        Chars: array (1..100) of character;
        function Length return integer;
end;

package body Var_String is
        function Length return integer is
        begin
                --find where character code "0" is in "chars" array
        end;
end;
```

With either implementation, the user of a variable-length string can find out the string's length by selecting the *Length* field, but with the generic package, selecting the *Length* component will result in a function being executed. This approach relieves all other string procedures from being burdened with keeping the *Length* component consistent with the *Chars* component.

II.3. Advantages of Records over Generic Packages

Although generic packages do have several advantages over record declarations, there are several features that record types have which generic packages lack: the use of assignment, the ability to be used in parameters and the ability to be composed in types other than records. Each of these advantages of record types is discussed below.

II.3.1. Assignment

If two variables are declared as a record type, then one may assign one variable to another (ignoring for the moment any constraint errors and limited types that might be present). Although two instances of a generic package might be thought of as two variables, there is no syntax for assigning one to another. Using the example of *Var_String* from before, we cannot write:

```
package String1 is new Var_String;
package String2 is new Var_String;
```

String1:=String2;

II.3.2. Parameter Passing

One might suggest that the addition of a "copy" procedure to the generic package for *Var_String* could solve the assignment problem for generic packages. Unfortunately, there is no way to pass a package as a parameter to subprogram. Although we could declare the procedure *Copy* in *Var_String*, there is no way to pass the value to be copied. Records, on the other hand, can be declared and passed as parameters.

II.3.3. Type Composition

A final advantage of record types over generic packages is the ability to use record types in defining access and array types. One can declare packages within packages, so there is a natural substitute for using record types in record definitions, but there is no simple way to declare an array of packages or a pointer to a package (although proposals for adding the type "pointer-to-package" has been made).

If one does not need the generic parameters (as, for example, if one just wants to hide certain record fields) and one is willing to use a subprogram to simulate each record component, then one can use a task type in much the same way as a record type. In such an approach, task entries in a select statement would serve the role of access subprograms.

III. Controlling Parameter Binding

The generic features of Ada permit one to pass parameters in a more flexible way than with subprograms. In this section, we consider several ways that one may exploit generic features to control parameter passing to procedures.

III.1. The S-M-N Theorem

The S-M-N theorem states, roughly, that one can bind some parameters of a computable function before binding others (and still have the same computable function). A generic procedure in Ada permits one to apply this theorem practically in two situations: precomputations and ordered evaluations.

III.1.1. Precomputation

One may have values in a procedure that are constant over several calls of a procedure. For example, one might build a table and then search it many times for a particular value. Normally one would pass the table as a parameter to the search procedure. In some circumstances, there may be some substantial computations needed to determine the table parameter. For example, the table might be part of a record in an array which is referred to by an access value. Instead of recomputing the parameter each time (especially with non-optimizing compilers), one might choose to "bind" the table into the search procedure [Hibbard, Lamb]. This can be accomplished by making a non-generic procedure into a generic procedure. For example, we might change:

```
procedure Search(t:in out table; s:string);
```

into

```
generic
        t: in out table;
procedure Search(s:string);
```

Before we might have had the series of statements:

```
Search(Master.Names(CurYear).CurTable,Str1);
Search(Master.Names(CurYear).CurTable,Str2);
Search(Master.Names(CurYear).CurTable,Str3);
```

These can be replaced by:

```
declare
        procedure MasterSearch is new Search(Master.Names(CurYear).CurTable);
begin
        MasterSearch(Str1);
        MasterSearch(Str2);
        MasterSearch(Str3);
end;
```

III.1.2. Order of Parameter Elaboration

Using a similar technique from the previous section, we can resolve a possible problem posed by Ada's definition of parameter passing: the unspecified order of parameter elaboration. The reference manual does not guarantee which parameter of many will be elaborated first (or rather, defines programs that

depend on a certain order to be erroneous). However, a programmer may wish to control parameter evaluation order, for example, when actual parameters call a function that manipulates global variables. Generic parameters allow one to control the order of evaluation. For example, if we have a procedure to two parameters, such as:

```
procedure P(a,b:integer);
```

and we wish a call to *P* to have the left parameter evaluated before the right parameter, we can make *P* into a generic procedure:

```
generic
        a: integer;
procedure P(b:integer);
```

and change a call of *P* from *P(L,R)* into:

```
declare
        procedure PL is new P(L);
begin
        PL(R);
end;
```

The sequencing of actions in the fragment above is controlled by the order of statement elaboration. Because Ada requires the procedure declaration of *PL* to be elaborated before the call of *PL*, we know that the parameter *L* will be elaborated before the elaboration of *R*. The extension to arbitrary numbers of parameters requires the encapsulation of the subprogram in a package, but is otherwise straightforward.

III.2. Guaranteed Parameter Passing (Pass-by-Reference vs Pass-by-Value)

The generic facilities of Ada can be used to force a specific kind of parameter passing. Normally parameters to subprograms will be passed by value-result when scalar and by an unspecified method when structured. In the presence of aliasing, one may wish to force Ada to use pass-by-reference for *in out* parameters. This can be accomplished by using generic parameters instead of conventional parameters. Specifically, Ada requires that *in* parameters to generics must always be copied into the instance while *in out* parameters must always be "renamed" inside of the instance. In the following example:

```
generic
        --I assume that ArrayType is an array type declared previously
        a: in out ArrayType;
```

```
        b: in out ArrayType;
procedure Mod;

procedure Mod is
begin
        --I assume that Modify is a procedure declared previously
        Modify(a);
        Modify(b);
end;
```

Mod is a procedure that will modify each array once. If *Mod* were written as a conventional procedure, one could not be certain that a call of *Mod* with the same parameter would cause the passed variable to be modified once or twice. Using the generic form of *Mod*, one can guarantee that passing the same parameter for both *a* and *b* would cause that parameter to be modified twice.

IV. Summary

In this paper I have shown that the generic features in Ada can be used in some novel ways not discussed in recent textbooks. Generic packages can be used as records with more flexible components, and generic procedures can be used to provide some alternative methods of controlling parameter binding. This paper continues a growing catalog of techniques for using Ada's generic facilities. Over time, other uses of generics will undoubtedly appear.

V. References

[ANSI] *American National Standard Reference Manual for the Ada Programming Language*, American National Standards Institute, 1430 Broadway, New York, NY. 10018, ANSI/MIL-STD-1815A-1983, 1983.

[Barnes] J.G.P.Barnes, *Programming in Ada*, Addison-Wesley International Computer Science Series, 2nd edition, 1984.

[Grogono] Peter Grogono, *Programming in Pascal*, Addison-Wesley Series in Computer Science, 1978.

[Habermann] A. Nico Habermann and Dewayne E. Perry, *Ada for Experience Programmers*, Addison-Wesley Series in Computer Science, 1983.

[Hibbard] Peter Hibbard, Andy Hisgen, Jonathan Rosenberg, Mary Shaw and Mark Sherman, *Studies in Ada Style*, 2nd edition, Springer-Verlag, 1983.

[Hilfinger] Paul N. Hilfinger, *Abstraction Mechanisms and Language Design*, Technical Report CMU-CS-81-147, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 15213.

[Lamb] David A. Lamb and Paul N. Hilfinger, "Simulation of Procedure Variables Using Ada Tasks," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 1, January 1983, p. 13-15.

[Liskov] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Robert Scheifler and Alan Snyder, *CLU Reference Manual*, Springer-Verlag Lecture Notes in Computer Science, Vol. 114, 1981.

[Wegner] Peter Wegner, "On the unification of data and program abstraction in Ada," *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, p. 256-264.