Comput. Lang. Vol. 12, No. 1, pp. 21–25, 1987
Printed in Great Britain. All rights reserved

0096-0551/87 $3.00 + 0.00
Copyright © 1987 Pergamon Journals Ltd

TWO IMPLEMENTATION MODELS OF ABSTRACT DATA TYPES

JOHN D. GANNON and MARVIN V. ZELKOWITZ
Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, U.S.A.

(*Received March* 1986; *revision received July* 1986)

Abstract—This paper compares two implementation models for abstract data types: direct and indirect implementations. Direct implementations offer relatively cheap execution and expensive compilation costs while indirect implementations result in relatively expensive implementations and cheap compilation costs. These two models are both accommodated by Ada, and a small experiment compares their costs for a particular data type.

Keywords: abstract data type ada run-time implementation

1. INTRODUCTION

The development of user-defined types in ALGOL 68 and Pascal provided a clear break with the earlier generation of languages such as FORTRAN and Algol which allowed users only a fixed set of types. Programmers using these newer languages can declare new types which in turn can be used in declaring new data objects. Objects with user-defined types correspond more closely with problem-domain objects than to programming-language objects.

Pascal and Algol 68 type definitions are not ideal methods for defining new types since there is no way to declare the operations of a newly declared type. Any program unit within the reach of the type definition can access the components of objects with user-defined types. Programmers define new operations with functions and procedures, and turned to methodologies like information hiding [1] to remedy the shortcomings of the type definition features of these languages. Only the few routines that actually manipulated the components of objects need to know the details of an object's representation; in the remainder of a program components can be manipulated indirectly by calling one of these routines.

During the 1970s, several languages were designed to extend type definitions that encapsulated the representation of objects and permitted users to define new operations on objects. Among the better known languages from this era are CLU [2], Alphard [3], Simula [4], Mesa [5], Euclid [6] and Ada [7]. Of these languages, Ada is certain to become the most widely used. The remainder of this paper discusses the implementation of encapsulated objects in Ada, although we believe similar results can be obtained from other languages.

2. TWO IMPLEMENTATION MODELS

In languages which support abstract data types two basic implementation models have been used; for purposes of discussion we call them: direct implementations and indirect implementations. Storage for directly implemented objects is allocated in the activation record of the procedure in which they are declared. This implies that the compiler knows the underlying representation of objects during compilation of all modules that declare objects and can (possibly) generate inline code to manipulate the objects. In contrast, indirectly implemented objects are represented in the activation record of their declaration as pointers to the actual (heap) storage for the objects. The compiler generating code for operations on these objects need not know their structure, but must generally generate less efficient code. For example, if an abstract type Stack was declared as follows

```
type Values is array (1 .. 100) of integer;
type Stack is
  record
    Top: integer;
    A: Values;
  end record;
```

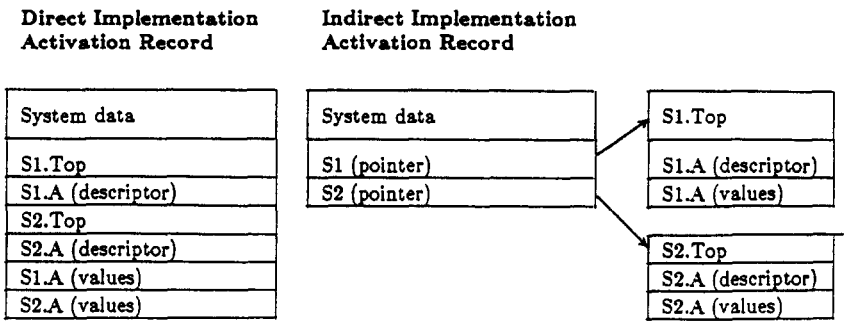

Fig. 1. Two activation records.

then instances of objects declared in a procedure

```
procedure P ...;
  var
    S1, S2: Stack;
  ...
```

could be represented in either of the ways shown in Fig. 1.

Direct implementations offer reductions in execution time (and possibly code size) over indirect implementations since it takes an extra memory cycle (and perhaps an extra instruction) to dereference the pointer to the object in indirect implementations. Attaining this advantage might depend critically on having a compiler that expands procedures inline. If data objects have to be passed (by reference) to procedures implementing abstract operations in order to perform the operation, the number of memory references in the direct and indirect implementations is likely to be similar.

Indirect implementations may lead to fewer compilations than direct implementations when changes are made to the representations of objects. When the representations of directly implemented objects change, then the module implementing the type's operations and all the compilation units that declare these objects must be recompiled. The latter units must be recompiled to effect changes in the structures of their activation records. However, in indirect implementations, all abstract objects are referenced via pointers so only the module implementing the type's operations needs to be recompiled. The recompilation costs for large systems of programs can be significant.

Thus the tradeoffs can be summarized as follows: relatively cheap execution and expensive compilation costs for direct implementations, and relatively expensive implementations and cheap compilation costs for indirect implementations. Among the better known languages, Euclid represents objects directly, while in CLU, Mesa, and Simula 67 objects are represented indirectly. We have implemented each of these approaches—the first in Simpl-D [8] and the second in PLUM [9]. Since our two implementations are for very different languages, it makes little sense to compare these two approaches empirically. However, Ada supports both these approaches so it is possible to compare them using Ada.

3. TWO MODELS IN ADA

In Ada, modules are called *packages*, and the visible part of a package that is known to other modules is specified in a *package specification*. An Ada specification contains the name of the abstract type, and procedures and functions that implement the operations of the type, as well as any other information the designer of the abstract type wishes to make known to the outside world. For example, a directly implemented stack could be specified as:

```
package Stack_Type is
  type Stack is private;
  procedure Push (S: in out Stack; X: in INTEGER);
  procedure Pop (S: in out Stack);
  function Top (S: in Stack) return INTEGER;
  function EmptyStack (S: in Stack) return BOOLEAN;
```

```
  private
    type Values is array (1 .. 100) of INTEGER;
    type Stack is
      record
        Top: INTEGER := 0;
        A: Values;
      end record;
end Stack_Type;
```

Stack objects could then be declared in another module as shown below.

```
with Stack_Type; use Stack_Type;
procedure StackTest is
  S1, S2: Stack;
  ...
end StackTest;
```

To avoid exposing the representation of a new type to many other compilation units as ALGOL68 and Pascal do, Ada provides the concept of a *private type*. While the name of a private type is available to all users of a package, access to its components is restricted to the package body with the same name as the specification. However, since the compiler reads the package specification, the benefits of both encapsulation and direct implementation are available to Ada users. Because of this, changes in the private part of the Stack_Type specification will necessitate recompilation of program units using Stack Type.

To reduce the number of compilations, indirectly implemented stacks could be specified by the package shown below.

```
package IStack_Type is
  type IStack is private;
  procedure NewStack (S: in out IStack);
  procedure Push (S: in out IStack; X: in INTEGER);
  procedure Pop (S: in out IStack);
  function Top (S: in IStack) return INTEGER;
  function EmptyStack (S: in IStack) return BOOLEAN;
  private
    type IStackRep; -- hide representation from outside
    type IStack is access IStackRep;
end IStack_Type;
```

The details of the representation of stack objects can be hidden within the package body.

```
package body IStack_Type is

  type Values is array (1 .. 100) of INTEGER;
  type IStackRep is
    record
      Top: INTEGER;
      A: Values;
    end record;


  procedure NewStack (S: in out IStack) is
    begin
      S := new IStackRep;
      S.Top := 0;
    end NewStack;
  ...
end IStack_Type;
```

Stack objects can then be declared and manipulated in separate modules.

```
with IStack_Type; use IStack_Type;
procedure StackTest is
  S: IStack;
  begin
    NewStack(S);
    ...
  end StackTest;
```

In procedure StackTest, the object named S is represented by a pointer and NewStack must be called to create the stack object. Thus changes to the representation of stack objects require that the package body be recompiled since NewStack creates these objects, but program units declaring

stack objects (like StackTest) remain unchanged. Note that we have not completely hidden the differences in representations between the two implementations. The indirect implementation still needs a call to NewStack to allocate storage. This difference can be avoided by adding a (superfluous) call to NewStack in the direct implementation. These difference could be avoided if Ada provided an initialization operation in package bodies.

Using an Ada compiler on a VAX 8600 using Berkeley UNIX 4.3, we ran two experiments to determine the expense (in terms of data space and execution time) of both indirect and direct implementations. In both experiments we used the Stack data types declared above. The first experiment shows the effects of declaring a number of stacks and doing the minimal number of operations on them (i.e. calling NewStack to create the indirectly implemented objects).

Space Used by Objects					
Implementation	Number of Objects				
	0	1	25	50	100
direct	29k	29k	48k	68k	108k
indirect	29k	31k	54k	77k	124k

The figures in this table are produced by the UNIX "time" command and represent "unshared memory" sizes of the data and stack segments of processes. As is evident from the table, the indirectly implemented objects use more storage than directly implemented objects. Each additional indirectly implemented object costs approximately 0.92 k of memory, while a directly implemented object costs an additional 0.8 k. Actual storage costs for each object is 404 bytes for indirectly implemented objects and 400 bytes for directly implemented objects so there is about 100% execution overhead per object allocation in using this compiler.

The second experiment measures the cost of operations on objects. In this case a series of pairs of Push-Pop operations was applied to a single Stack object.

Time Consumed by Operations				
Implementation	Number of Push-Pop Pairs			
	100k	250k	500k	1000k
direct	1.6	4.2	8.4	16.9
indirect (access check)	2.1	5.2	10.4	20.9
indirect (no access check)	1.7	4.3	8.7	17.3

The figures in this table are produced by the UNIX "time" command and represent "user time", the number of seconds executing the user's program. Indirectly implemented operations took approximately 25% longer to execute than the corresponding directly implemented operations. Investigation of the generated code for indirectly implemented objects revealed that the pointer to the stack object was being checked to make sure it was not null each time the stack object was referenced. Inserting the pragma SUPPRESS(ACCESS_CHECK) eliminated almost all of the performance differences between the two implementations. The small differences that remain are the result of an extra instruction generated for parameter transmission of indirectly implemented objects. Passing one of these objects as an in-out parameter results in the value of the pointer representing the object being passed to the operation and the (possibly changed) value being copied back to the object's storage. In contrast, the compiler generates an address for directly implemented objects and knows that any change in the address need not be restored. While the execution differences are small, running programs with access checking disabled is unsafe.

As predicted, expanding the operations inline results in even larger advantages for directly implemented objects.

Time Consumed by Operations				
Implementation	Number of Inline Push-Pop Pairs			
	100k	250k	500k	1000k
direct	0.6	1.5	3.0	6.2
indirect	1.1	2.8	5.6	11.2

Part of the differences in these figures results from the compiler not eliminating access checking when expanding operations on indirectly implemented objects. Even if we remove the differences found in the previous values when neither set of operations was expanded inline, we still see an advantage for directly implemented objects.

4. CONCLUSIONS

In Ada, at first glance there is no apparent reason why a private part should appear in a package specification rather than a package body since users of the package cannot directly access the representation of the type. The reason becomes clear when we consider implementation models. Since a major goal of the language is rapid execution in embedded computers, execution time is a primary design goal and compilation costs have a relatively low priority. By putting the private parts into the specification, an Ada compiler can implement objects directly rather than indirectly saving both storage space and execution time. By offering both implementation possibilities, Ada permits its users to save on compilation costs by using indirect implementations during development, and to save on execution costs by switching to direct implementations during production.

Acknowledgement—This work was supported by the Air Force Office of Scientific Research under grant F49620-85-K-0018.

REFERENCES

1. Parnas D. L., On the criteria used in decomposing systems into modules. *Commun. ACM* **15**(12), 1053–1058 (December 1972).
2. Liskov B., Abstraction mechanisms for CLU. *Commun. ACM* **20**, 564–576 (1977).
3. Wulf W., London R. and Shaw M., An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Engng* **2**, 253–264 (1976).
4. Dahl O.-J., Myhrhaug B. and Nygaard K., *The SIMULA 67 Common Base Language*. Norwegian Computing Center, Oslo, Publication No. S-22 (1970).
5. Geschke C. M., Morris J. H. and Satterthwaite E. H., Early experience with Mesa. *Commun. ACM* **20**, 540–553 (1977).
6. Holt R. C. and Wortman D. B., A model for implementing Euclid modules and prototypes. *ACM TOPLAS* **4**(4), 552–562 (October 1982).
7. *Reference Manual for the Ada Programming Language*. United States Department of Defense, Draft Revised MIL-STD 1815 (July 1982).
8. Gannon J. D. and Rosenberg J., Implementing data abstraction features in a stack-based language. *Software Pract. Exper.* **9**, 547–560 (1979).
9. Zelkowitz M. V. and Lyle J. R., Implementation of language enhancements. *Comput. Lang.* **6**, 139–153 (1981).

About the Author—JOHN D. GANNON received the A. B. degree in mathematical economics and the M.S. degree in applied mathematics from Brown University in 1970 and 1972, respectively, and the Ph.D. degree in computer science from the University of Toronto in 1975. He is currently an Associate Professor in the Department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland. Dr Gannon's research centers on language and compiler design to increase the reliability of programs. Initially his work focussed on the design of less error-prone programming languages. His interests in program proving and testing have led him to investigate formal specifications, test oracles, and test coverage metrics. Most recently, Dr Gannon has studied atomic remote procedure call in response to the problems of distributed and fault-tolerant computing.

About the Author—MARVIN V. ZELKOWITZ is Associate Professor of Computer Science at the University of Maryland. He received a Ph.D. degree in Computer Science from Cornell University in 1971. He is a past chairman of both ACM SIGSOFT and IEEE Computer Society Technical Committee on Software Engineering and is currently on the Executive Committee of both of these. His major research interests are in programming language design, language implementation and environment design, and he has published many papers on these topics.