ABSTRACT DATA TYPES IN EUCLID[1]

by Ernest Chang, Neil E. Kaden, and W. David Elliott
Department of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A7
Canada

Abstract: The programming language Euclid provides features that support abstract data types, but does not strictly speaking provide a true data abstraction mechanism. This paper assesses the data abstraction facilities that Euclid does provide, examines the two ways of instantiating Euclid modules, and discusses other features of modules that the designers of Euclid chose not to include. In particular, the paper addresses the issues of (1) enforceable separation between abstract definition and representation, (2) specifying the relationship between abstract definition and representation, (3) type parameters in modules, (4) operator extensions, and (5) scope restrictions on identifiers.

Contents

[1] This paper also appears in Computer Systems Research Group Technical Report 82, "Notes on Euclid," edited by W. David Elliott and David T. Barnard. Since the final version of the Euclid Report appeared after these papers were completed, the editors have effected changes and have tried to keep the papers as coherent as possible in spite of this sometimes radical surgery. Any contribution these papers have to make is a tribute to Jim Horning, whose consistent insights and enthusiasm motivated the authors (and editors) throughout.

ABSTRACT DATA TYPES IN EUCLID

1 Introduction

The programming language Euclid [Lampson et al.77] was designed to support the writing of verifiable system software, with the subgoal, as a near-term project, to differ from Pascal as little as possible. Hence Euclid was expressly not intended to be a research vehicle for new ideas in programming languages. One notable exception the Euclid designers made was in the area of data abstraction.

Virtually all data abstraction mechanisms claim lineage from the Simula 67 class [Dahl et al.68]. Descendent data abstraction mechanisms have not in general been used long enough for us to have much experience with their use, and as a result much of the area of data abstraction still belongs in the realm of research.

Although it was not absolutely necessary for Euclid to provide a data abstraction facility, its designers felt that such a programming tool would greatly contribute to the ability to decompose large programs so that they could be verified with existing verification methods. A data abstraction mechanism also encourages writing modular and structured programs, and thus helps meet other Euclid design goals. ([Horning76] provides an excellent summary of the advantages provided by abstraction mechanisms, both data and procedural.)

2 Euclid Modules

The module in Euclid is an encapsulation mechanism whereby the representation of an abstract object and the implementation of associated operations can be hidden from the enclosing scope. Multiple instances of an abstraction can be realized from the definition of a module type by declaring variables of that type. Closed scopes, and modules in particular, provide explicit control over the visibility of identifiers. Objects, operations, and types defined within the module must be explicitly exported in order to be used; similarly, values of variables declared outside a module must be imported explicitly to be known inside.

2.1 Modules as Abstraction Mechanisms

A data type is defined by a set of values and a set of operations on those values. An abstract data type is a data type with a representation-independent definition. Thus, abstract data types permit access by outside routines only to the abstract values and operations, and not to any of the underlying representation. In this sense, clusters in CLU [Liskov et al.77] and forms in Alphard [Wulf et al.76] are abstract data types, whereas classes in Simula 67 [Dahl et al.68] are not, since all data structures in the outermost scope of a class are accessible. Palme [73] has shown, however, how the necessary protection could be added to Simula 67 in a straightforward way.

For reasons similar to those for Simula 67 classes, Euclid modules are not true abstract data types. Access to identifiers

within a module is severely restricted by the import and export clauses as well as the Euclid scope rules, but access is allowed.

Euclid modules can be used, however, to implement abstract data types. This would require additional programmer discipline, unenforceable by the language itself, to ensure that the only entities accessible to outside routines are those abstract entities being defined.

2.2 Two Ways of Instantiating Modules

In Euclid there are two distinct methods of defining equivalent objects and their operations. A module type can be defined, and instances of the module type declared in the enclosing scope. Alternatively, a type definition can be exported from a module, and objects of that type declared in the enclosing scope.

Since no self-respecting abstract data type paper would be complete without an example of a stack, we will use the stack example to illustrate the two instantiation methods. Consider the following implementation of a bounded stack of integers in which multiple instances of the module will be instantiated. The procedure "Pop" pops the top value from the stack and assigns it to the parameter pased.

```
type Stack(StackSize: unsignedInt) = module
     exports(Pop,Push)
     var IntStack: array 1..StackSize of signedInt
     var StackPtr: 0..StackSize := 0

     procedure Push(X: signedInt) =
          imports(var IntStack, var StackPtr, StackSize)
          begin
          procedure Overflow = ... end Overflow
          if StackPtr = StackSize then
               Overflow
          else
               StackPtr := StackPtr+1
               IntStack(StackPtr) := X
          end if
          end Push

     procedure Pop(var X: signedInt) =
          imports(var IntStack, var StackPtr)
          begin
          procedure Underflow = ... end Underflow
          if StackPtr = 0 then
               Underflow
          else
               X := IntStack(StackPtr)
               StackPtr := StackPtr-1
          end if
          end Pop

     end Stack
```

The user would access the stack by code such as:

```
var A,B: Stack(100)
var Element: signedInt
...
B.Push(3);
A.Pop(Element);
```

Note: Because functions cannot have side effects and "Pop" alters the stack as well as returning a value, we cannot use the more natural form "Element := A.Pop".

Alternatively, if the module "Stack" exported a type definition "Stk", we could have the following module:

```
type Stack = module
     exports(Stk, Pop, Push)
     type Stk(StackSize: unsignedInt) = record
          var StackPtr: 0..StackSize := 0
          var Body: array 1..StackSize of signedInt
          end Stk

     procedure Push(var IStk: Stk(parameter),
                    X: signedInt) =
          begin
          procedure Overflow = ... end Overflow
          if IStk.StackPtr = IStk.StackSize then
               Overflow
          else
               IStk.StackPtr := IStk.StackPtr+1
               IStk.Body(IStk.StackPtr) := X
          end if
          end Push

     procedure Pop (var IStk: Stk(parameter),
                    var X: signedInt) =
          begin
          procedure Underflow = ... end Underflow
          if IStk.StackPtr = 0 then
               Underflow
          else
               X := IStk.Body(IStk.StackPtr)
               IStk.StackPtr := IStk.StackPtr-1
          end if
          end Pop

     end Stack
```

Corresponding user code might be:

```
var S1: Stack                    {instantiates the module}
var A: S1.Stk(100)               {instantiates the object}
var B: S1.Stk(299)               {and another object}
var Element: signedInt
...
S1.Push(B,3)
S1.Pop(A,Element)
```

The differences between these two stack implementations are largely stylistic. There are, however, situations in which the second instantiation method is more powerful, as shown in the following example.

We would like to define a data type "CharString" that contains a procedure "Append" that operates on two strings passed as arguments. The first method of instantiation requires that the data representation be declared inside the module, and requires code such as:

```
type CharString = module
     imports (CharString)
     exports (Append)
     var X: array 1..250 of char
     procedure Append (var S: CharString) =
          imports (X) ...
          end Append
     ...
     end CharString
```

Since "Append" must have access to the representation of the character string "X", it must be located inside the module "CharString". The requirement that the module import itself, in order for "Append" to be able to access the other string, however, presents an illegal situation in Euclid.

Using the second instantiation technique, this problem is easily dealt with:

```
type CharStringModule = module
     exports (CharString, Append)
     type CharString = record ... end CharString
     procedure Append (var S1,S2: CharString) =
          ...
          end Append
     ...
     end CharStringModule
```

with user code

```
var S: CharStringModule
var Sa, Sb: S.CharString
...
S.Append(Sa,Sb)
```

The existence of the two instantiation methods adds to the complexity of the language, especially since combinations of the two methods are possible. The Euclid designers, however, felt that there were situations in which each of the two methods provided a more natural solution. Instantiating modules avoids the bother of re-importing variables of an exported type. And, as we saw above, instantiating a type exported from a module provides capabilities not provided by simply instantiating the module.

3 What Euclid Left Undone

Although modules were a notable exception to the Euclid design guideline of no innovation, somewhat conservative design decisions were made concerning modules. This section discusses areas where the Euclid designers could have provided further module capabilities.

3.1 Separation of Abstraction and Representation

In CLU [Liskov et al.77], Mesa [Geschke et al.77], and Alphard [Wulf et al.76], it is possible to write the specification of an abstract data type as an entity completely distinct from its implementation. In this way, it is possible to change representations quite easily and to implement libraries of data abstractions and implementations, both of which capabilities are highly desirable.

Euclid, however, provides no syntactic mechanism to ensure that this separation is preserved. It is possible in Euclid to implement interchangeable modules for the same abstract specifications, but the specification would have to be textually copied between modules in order to do so. Also, there would be no way within the language of ensuring that the textually copied specifications remain the same.

3.2 Axiomatic Specification

In order to be able to prove that a particular representation of an abstract data type does indeed implement the specified data abstraction, a language must provide a formal means of specifying the relationship between the concrete representation and abstract definition. Although Euclid did not originally provide such a mechanism, the ill-defined abstraction function now fulfills that role, in much the same way that the (somewhat misnamed) rep function does in CLU. In Alphard this relationship between concrete and abstract is specified in the representation section of the form. None of these languages, however, provide an adequate means for axiomatic specification [Guttag et al.76].

3.3 Schemes

Types in Euclid are allowed to have formal parameters. Such parameters are typed constants, but need not be manifest. It is possible to defer fixing the value of a parameter by specifying it as any, unknown, or parameter, but it is not possible to pass types as parameters to a parameterized type.

Mitchell and Wegbreit [76] have coined the term "scheme" as a generalization of parameterized types in which type values can be passed as parameters to the definition mechanism. The instantiation of a scheme is a (possibly parameterized) abstract data type. Thus, for example, passing type "integer" as a parameter to a scheme definition could produce a bounded stack of integers data type. This data type can in turn be instantiated

to produce a particular abstract data object. Such facilities exist in both CLU and Alphard, as well as in EL1 [Wegbreit74].

3.4 Operator Extensions

Operator extensibility in Euclid is strictly procedural in nature. The generic operators equality and assignment are identical for all modules, though to be used they must be explicitly exported. A new operator defined on a data type can be viewed only as a routine, not as a new infix or prefix operator.

Operators can be redefined in both CLU and Alphard, and even equality and assignment can be redefined in CLU. Alphard also allows operators to be defined as infix or prefix, which contributes to readability.

3.5 On Inheriting Scope

The Euclid designers took much of the advice of Wulf and Shaw [73] to heart in attacking the problems of aliasing and global variables. Unlike the designers of Gypsy [Ambler et al.77], who discarded the Algol notion of nested scopes, the Euclid designers chose to reinforce Algol-like block structuring with further restrictions. In particular, Euclid requires import lists for closed scopes, prohibits redeclaration of variables within a scope, prohibits "sneak access" to variables via procedure calls, and disallows functions with side effects. Euclid allows different types of access (e.g., read, write) to be associated with an exported or imported variable, as does Alphard.

Both Euclid and Alphard require that an identifier be passed through all intervening scopes in order to be known within an inner scope. (Similarly, in Euclid all ancestors of a machine-dependent module must be made machine-dependent.) Since Gypsy does not permit a hierarchy of routine declarations, there are no intervening scopes that need simply pass an identifier along.

4 Summary

Euclid provides features that support abstract data types, but does not strictly speaking provide a true data abstraction mechanism. With modules, the Euclid designers struck a balance between providing abstraction capabilities and ensuring that the capabilities provided could be fairly easily implemented. In particular, Euclid could have provided further capabilities in the areas of enforceable separation between abstract definition and representation, specifying the relationship between abstract definition and representation, type parameters in modules, operator extensions, and scope restrictions on identifiers.

Acknowledgements: We appreciate the efforts of Inge Weber, who patiently edited this document. Ric Holt provided helpful comments.

References

[Ambler et al.77]
A.A. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch, and R.E. Wells; Gypsy: A Language for Specification and Implementation of Verifiable Programs; SIGPLAN Notices 12,3 (March 1977) pp. 1-10.

[Dahl et al.68]
O.-J. Dahl, B. Myhrhaug, and K. Nygaard; The Simula 67 Common Base Language; Norwegian Computing Center, Oslo (1968).

[Geschke et al.77]
C.M. Geschke, J.H. Morris, and E.H. Satterthwaite; Early Experience with Mesa; to appear in CACM.

[Guttag et al.76]
J.V. Guttag, E. Horowitz, and D.R. Musser; Abstract Data Types and Software Validation; USC Information Sciences Institute Technical Report (1976).

[Horning76]
J.J. Horning; Some Desirable Properties of Data Abstraction Facilities; Proceedings of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11, Special Issue (March 1976) pp. 60-62.

[Lampson et al.77]
B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek; Report on the Programming Language Euclid; SIGPLAN Notices 12,2 (February 1977) pp. 1-79.

[Liskov et al.77]
B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert; Abstraction Mechanisms in CLU; to appear in CACM.

[Mitchell and Wegbreit76]
J.G. Mitchell and B. Wegbreit; A Next Step in Data Structuring for Programming Languages; Proceedings of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11, Special Issue (March 1976) pp. 69-70.

[Palme73]
J. Palme; Protected Program Modules in Simula 67; Research Institute of National Defense, Stockholm (1973).

[Wegbreit75]
B. Wegbreit; The Treatment of Data Types in EL1; CACM 17,5 (May 1974) pp. 251-264.

[Wulf and Shaw73]
W.A. Wulf and M. Shaw; Global Variable Considered Harmful; SIGPLAN Notices 8,2 (February 1973) pp. 28-34.

[Wulf et al.76]
W.A. Wulf, R.L. London, and M. Shaw; Abstraction and Verification in Alphard; Carnegie-Mellon University (also USC Information Sciences Institute) Technical Report (1976).