Abstract Data Types with Shared Operations.

Kasper Østerbye

Abstract:
It is argued that the concept of abstract datatypes can be extended in a way that allows for operations to be shared between two (or more data- types), and that this will often give a better and more natural way of structuring programs. Shared operations can access the representation of more than one abstract datatype.

1. Introduction.

The notion of abstract data types is a powerful concept that allows us to make abstractions, and to use complex program components without having to worry about how they are implemented. On the other hand, everyone who has used a language that forces the use of abstract datatypes will have experienced the problem of where to put things. In this paper an extension of the abstract data type concept is presented. Using this extended concept it is possible to let one operation be part of two (or more) abstract data types. Such an operation will be referred to as a shared operation.

Shared operations are proposed because it is sometimes necessary to let an operation have access to the representation details of more than one abstract data type in order to provide an efficient implementation. The argument for introducing shared operations is efficiency gain and better structuring without compromising the information hiding principles behind abstract datatypes.

In this paper the example of multiplying a vector with a matrix has been chosen, as it is necessary for the multiplication operation to have access to both the representation of the vector and the matrix to provide an efficient multiplication.

Figure 1.a to 1.c illustrates the point. If we place the multiplication as in fig. 1.a, we must provide some means for the vector to access the representation of the matrix. Such a representation providing operation, is a de facto violation of information hiding, and should be avoided. In figure 1.b the symmetrical problem arises, and in figure 1.c where we have placed the multiplication outside both datatypes is worse, as there must be a representation providing operation in both the vector and the matrix.

Address: Department of Computer Science, building 460, Stanford University, Stanford California 94305.
EMAIL:Internet: kasper@csil.stanford.edu
If we have a datatype concept that does not allow for shared operations, we cannot place the multiplication operation so that we have access to the representation of both datatypes.

What we propose is to extend the notion of an abstract datatype in such a way that operations can be thought of as operating on more than one abstract data type at a time. The basic idea is illustrated in figure 2. When Vector and Matrix is designed, it is anticipated that there is an instruction (Multiplication) that will need access to both representations. This way we are not forced into writing representation providing operation's in either vector or matrix.

2. Existing Languages.

In ADA, abstract data types are made by making a package that contains a type declaration, and a number of procedures and functions that can manipulate the type. To solve the problem of access to representation in ADA, one can define a package that
contains both the definition of matrix and of vector. Placing the multiplication operation in that same package will solve the problems. The multiplication operation will have access to the representation of both the vector and the matrix, and multiplication is defined structurally together with vector and matrix. This is illustrated in figure 3.

![Combined Package Diagram](image)

Figure 3.

The problem is that vectors might be multiplied by both complex- and rational numbers as well. By that argument the package will have grown rather large and the idea behind using packages is lost. Also the operation for multiplying vectors with matrices might end up in the same package as the operation to add points, and the intention of using packages for abstract data types is lost. (an other example, used by Ingalls (DHI), is where to put the print operation, in the object to be printed, or in the media to print on)

In object-oriented languages, that protects the representation of classes (like SmallTalk [ST] and CLU [CLU]), dataprotection is so strongly part of the language that it is not possible to define a operation that can access the representation of other than the class in which it is defined. The other arguments of the operation can only be accessed via their message protocols. In other object-oriented languages (like Simula and Loops[Loops]), the representations of all objects are accessible to all other objects, and it is therefore arguable whether or not these languages support abstract datatypes. However the problem of representation access does not exist.

Use of (multiple) inheritance has been considered, but it does not seem to offer any solutions to representation access.

In [Olm] it is shown how one can sometimes solve some problems like this using block structure in object oriented languages. In this case however, his proposal will end up with the same problems as outlined under ADA.

3. A solution in CLU.
Shared operations can be incorporated into CLU with what seems to be minimal changes. An abstract datatype in CLU takes the form:

Vector = cluster is create, copy, similar, add, mult
rep = Real_List
create = proc() returns (Vector) ... end create
copy = proc(v:Vector) returns(Vector) ... end copy
...
mult = proc(v:Vector, m:Matrix) returns(Vector) ... end mult
end Vector.

Within, say the copy operation the representation of "v" can be examined by the use of "down(v)" which type-converts "v" from "Vector" to "Real_List". The operation "down" is only allowed for the type of the cluster (datatype) in which we are working (here "Vector"). That is, in the "mult" operation we can get to the representation of "v" by "down(v)", while "down(m)" is not allowed.

If one were to extend CLU to include shared operations, all that is needed is to allow for the "down" operator to be used on the matrix too. This can of course not be allowed in general. A possible way to restrict it, would be to include a special syntax in the cluster header.

Vector = cluster is create, ..., mult shared with (Matrix)
...
end Vector
Matrix = cluster is create, ..., mult shared with (Vector)
...
end Matrix

If the "X shared with (Y_1, ..., Y_n)" is included, it means that the "down" operation will be allowed for the types Y_1, ..., Y_n in the operation X. By requiring all datatypes to include the "shared with" clause, it is assured that unintentional access to the representation is not possible. It is the designer of the datatypes vector and matrix that decides "mult" is shared. If someone later wants to write an operation, that operation can not access the representation of vector or matrix, as there is no shared clause for that new operation.
The reason for choosing CLU as an example language was that it was the language that needed the least twisting to allow for shared operations. Other languages have been considered too. ADA requires only minor changes to allow for shared operations. Also the build-in polymorphism gives a very nice syntax for calling shared operations. Most object-oriented languages have a special syntax for operation called "receiver.opr(arg)" that introduces an unnatural asymmetry when we want to think of an operation as belonging to more than one class ("(matrix,vector).mult" ?). In [KMMN] an environment that handles program fragments is presented. The environment allows for parse trees, that are actually acyclic graphs. In such an environment it will be quite natural to have shared operations.

4. Further examples.

In his paper Rumbaugh [JRu] argues that relations should be included as an extension to object oriented languages. One of his arguments is that if we want to model an employed-by relation between person objects and company objects, we would normally do that by letting each person have a pointer to a company object, and have the company object have a set of pointers to persons employed by that same company. Now Rumbaugh correctly points out, that if we want to make an operation "firePerson", and if we place that operation in the company-class, we can not update the pointer in the person object, and vice versa (Due to data protection). However, if we are using the shared operation abstract data type model, it is possible to define an operation that updates both pointers at the same time, thus keeping the relation consistent.

Ingalls [DHI] uses the example of displaying various objects on a number of media to argue that polymorphism is useful in more than a single argument. He does not address the problem of representation access, and it is arguable whether or not it is needed in the case of printing pictures on files.

5. Conclusion.

It is sometimes necessary for an operation to have access to the representation of more than one abstract data type, and therefore breaking the usual conception of an abstract data type in order to provide for an efficient implementation. One definition of an abstract data type is:

"An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects." Liskov and Zilles in [LZ]

However, the above definition does not indicate that the same operation can not be used to characterize several abstract data types. In this paper it has been argued that such a situation is
sometimes interesting, and a way shared operations can be implemented has been outlined.

While the concept of shared operations proposed in this paper does increase inter-modular dependencies, it is our belief that it does so in a structured way. The present alternative, to provide representation exporting operations seems more likely to increase inter-modular dependencies as it gives no control over which datatypes will depend on others.

6. Acknowledgements.
This idea developed from discussions on whether or not abstract data types were imposing too strict and unnatural restrictions on the programmer. I would like to thank Terry Winograd, Kristen Nygaard, Bent Brun Kristensen, and Liam Peyton for providing good comments on this idea.

References:


[ST] Smalltalk-80: The language and its implementation. By A. Goldberg and D. Robson. Addison-Wesley, 1983