REPRESENTATION OF ALMOST CONSTANT VECTORS

by

J. Steensgaard-Madsen*

TR79-375

J. Steensgaard-Madsen
Department of Computer Science
Cornell University
Ithaca, New York 14853

*On leave from University of Copenhagen, DIKU, Sigurdsgade 41, DK-2200 Copenhagen N. Denmark, with support from the Danish Natural Research Council.

Representation of almost constant vectors.

J. Steensgaard-Madsen *

Department of Computer Science
Cornell University
Ithaca, New York 14853

ABSTRACT

An example in a recent report on the programming language Russell has illustrated difficulties related to user defined storage management. Here is demonstrated how the dynamic approach to encapsulation earlier proposed by the author provides means to solve the particular storage management problem. The method used is, however, easily generalized to other similar cases.

In addition to the example a number of notational conveniences are introduced. One that allows abbreviated references to components of record-like structures is called controlled coercion. Another allows a function-like use of classes.

Keywords: Classes, abstract data types, storage management, programming languages.

---

* On leave from University of Copenhagen, DIKU, Sigurdsgade 41, DK-2200 Copenhagen N, Denmark, with support from the Danish Natural Research Council.

Representation of almost constant vectors.

J. Steensgaard-Madsen *

1. Introduction.

In a previous report I have introduced the following constructs for use in a Pascal-like programming language: Types as parameters, classes (often considered to be abstract data types) and objects. The constructs for class definition and object application can in their basic form be explained by simple rewriting rules transforming them into procedure declarations and calls. In this sense class and object usage can be considered a simple shorthand for a programming technique relying on procedures. A fairly obvious idea is to look for a similar technique related to functions. A very trivial result comes out of this search, basically a notational convenience.

A. Demers and J. Donahue have in a recent report [1] defined a programming language called Russell. An example in the report shows a definition of a type, "sparse", intended for representation of sparse matrices. The solution to the problem of representing sparse matrices should illustrate the implementation of a type with its own storage management. The solution should also enhance facilities of Russell particular to that language. Unfortunately the example in the available preliminary version of the report is in error because storage may be released while still in use with obvious bad effects.

The fundamental reason why the example fails is that a programmer has no means to express actions to be performed after an access right to a capsule (class or abstract data type) has been exercised. The facility to do this by means of nested class definitions is fundamental in the dynamic approach to class semantics in my proposal. This report shows how a class similar to "sparse" may be defined

2. Controlled coercion.

Consider a variable declaration as in Pascal:

```
var x : record
          n : integer;
          s : array [ 1..80 ] of char
        end
```

This allows field references like x.n and x.s [10]. Now, one may want the convenience of omitting some field identifiers. Context requirements will often provide enough information for automatic selection of the proper field - e.g., x [10]. Such inferences from context requirements to automatically invoked operations are called coercions. If only one path from one level of nested definitions to another may be followed by coercions, common problems with backtracking across several levels can be avoided. This means that if some context requirements are not fulfilled in a particular situation only one field - or sequence of fields - can be produced in any attempt to fulfill the requirements. In this report brackets around an identifier will be used to indicate a field identifier that may be omitted from a reference. As coercions thus is under the control of the programmer the term <u>controlled</u> <u>coercions</u> will be used.

The example declaration above may be modified to

```
var x : record
          n : integer;
        [s] : array [ 1..80 ] of char
        end
```

allowing constructions like x.n := 0 and x [3] := 'a', but not x := x + 1. The same mechanism may be used in selecting facilities provided by an object.

3. Anonymous objects.

The application of classes has in an earlier presentation [3] been restricted to a context called an object statement:

```
object x : <class call>;
<block>
```

Within the block of an object statement as above, facilities of the object x are accessible much like fields of record - e.g., x.reset(20) denotes selection of the facility reset of the object x. If x is used only once we have a situation like

```
object x : <class call>;
begin  S ( x ) end
```

This can, however, be expressed more clearly by allowing a class call to appear wherever an object identifier may be used - i.e., the above situation specifies the meaning of

```
S ( <class call> )
```

We say that an anonymous object is used within S.

Because one could expect that usually more than one facility of an object will be used it might seem counterintuitive to provide a special notation for the case when an object is referred to only once. Some justification can, however, be found in ordinary expressions if left operands of an infix operator is considered as an object. Usually, a particular operator can be seen as a choice from a set of possible operators.

Anonymous objects are especially useful in with-statements. Examples given with the proposal of the class and object notation gain in clarity by the use of anonymous objects in with-statements - for example, declaration of a shared variable and its use in a conditional critical region simplifies to

```
{ see explanation below }
object x : sharing of record ch : char; ... end;
...
with x.when ( B ) do begin  shared.ch := 'A' ... end
```

thus establishing an even closer analogy to Hoare's original proposal.

Above, "sharing" is a class identifier requiring one type parameter following <u>of</u>. Any object of class sharing provides a facility "when", which is a class. The benefit of using an anonymous object here is in the class call x.when (B). The identifier "shared" denotes a facility provided by objects of class when. "Shared" gives access to the shared variable of the type given in the call of "sharing".

If with-statements are allowed as statement part of a block, yet another advantage will be obtained. Blocks, and programs in particular, then have a prefix property in so far as a new, protected environment can be created for a block by just adding text to precede the normal text of the block, very much like SIMULA 67.

4. <u>Almost constant vectors</u>.

This section gives a class definition in the notation of [3] with which the reader should be familiar. However, the notation is in good agreement with other proposals and for the benefit of those who might go on without the detailed knowledge of [3] the overall structure of a simplified heading is given:

```
class <class identifier> (<import parameter list>)
: <export control identifier> (<export parameter list>)
```

As seen from the example below, the import parameter list and the enclosing parenthesis may be omitted. The export control identifier - always "def" in the example below - and the associated export parameter list is in fact an additional procedure-parameter to the class. It is set off to the right of the other parameters because no actual parameter is substituted explicitly. Implicitly, however, an actual parameter is constructed from every object-statement. The parameter identifiers in the export parameter list are called access identifiers because they are used to access facilities provided by an object of the class. Within the class definition the additional parameter is used as a procedure and a call of the procedure corresponds to the activation of an object-statement. The actual parameters in such an activation binds dynamically access-identifiers of the object to definitions (probably) local to the class definition. Local definitions that are not bound to access identifiers are thus hidden to the object statement.

The example from the report on the programming language Russell that triggered this report is on sparse matrices. A sparse matrix could be represented by a two dimensional array. In that case most components would have identical values - e.g., zero. Thus, better storage economy might be obtained by keeping an explicit representation of only non-zero values. Because the set of components with non-zero values may vary in time an explicit storage management is required.

In order to reduce the size of the example and concentrate on the storage management issue, the following specialize to the similar problem for vectors. Letting the index-type be reals emphasizes that an array representation is inadequate. The vector will be represented by a linear linked list of components with non-zero values, sorted on increasing values of indices. A component record contains in addition to index, value, and link fields a field "unused" that is true when the component is in use from outside the class.

The storage management works as follows. When a component of the vector is referenced, via an object of class access, the field "unused" is copied into a hidden variable, "first_access". One such variable is allocated at each reference to a vector component. An essential part of the invariant for access-objects (cf. [2]) is

first_access = (number of references to component p^ = 0)

The number of references may of course change during the actions represented by the call "def (p^.value)", but at completion of the call the relation is valid again. The proof is simple: use of the parameters of "def" cannot change the invariant, and later references to the component p^ must be dynamically contained in calls of class access that will be completed when "def (p^.value)" is completed.

Overhead in this solution amounts to a) one boolean variable for each value represented in a vector, and b) some local variables and block administration information for each active call on class "access" - typically only a few, say 2 or 3 calls.

```
     class thin_vector
      :def ( class [access] ( pos : real )
            : def ( var [v] : real )      );
       type
          link      = ^ component;
          component = record
                         index, value : real;
                         next         : link;
                         unused       : boolean
                      end;
       var
          first     : link;
          { first^.index < first^.next^.index <
                           first^.next^.next^.index < ...  }

       class access ( position : real )
         : def ( var v : real );
       {------------------------------------------------
       }  var first_access : boolean; p : link;          {
       }    { --- }                                      {
       }  begin                                          {
       }     { find a component p^ such that p^.index }  {
       }     { = position, possibly by allocating a   }  {
       }     { new component, setting p^.value to 0   }  {
       }     { and p^.unused to true                  }  {
       }                                                 {
NB:    }     first_access := p^.unused;                  {
 !     }     p^.unused := false;                         {
 !!    }   { I, where I = ( first_access =          }    {
 !!    }   {                ( no. of ref's to p^ = 0) ) } {
 !!    }     def ( p^.value );                           {
 !!    }   { I }                                         {
 !     }     if first_access then                        {
 !     }        if p^.value = 0 then                     {
 !     }           { --- Free p --- }                    {
 !     }        else p^.unused := true                   {
 !     }  end;                                           {
       ------------------------------------------------}
    begin
       first := nil;
       def ( access );
       { --- Free all remaining components --- }
    end
```

Given the definition above, an object of class thin_vector may be created and used in an object-statement like

```
object x : thin_vector;
<block>
```

In the block of this statement one may find parts as:

```
x.access ( 0.5 ).v := 1.5;                          { 1' }
x ( 0.7 ) := x ( 0.5 ) - 1.5;                       { 2' }
```

with the following meaning according to the sections on anonymous objects and controlled coercions:

```
object NN : x.access ( 0.5 );
begin NN.v := 1.5 end;                              { 1 }

object NN1 : x.access ( 0.7 );
       NN2 : x.access ( 0.5 );
begin NN1.v := NN2.v - 1.5 end;                     { 2 }
```

Note that the order of allocation of NN1 and NN2 is irrelevant.

In 1 the compound statement will be executed corresponding to the procedure call def (p^.value) in the definition of "access". Hence, the component identified by position = 0.5 will remain represented in x after execution of 1. Later, in 2, the component identified by position = 0.7 will be removed since zero is assigned as value. Note especially, that if more than one reference to a particular component is in effect at one time only the dynamically first may cause the disposal of that component.

5. Conclusions

The use of classes has been extended so that class calls may appear in places where a corresponding object denotation might have been used. Further, a notation that explicitly allows coercion in the form of default selection of a facility has been introduced.

Using these means it has been shown how a programmer may solve a particular storage management problem. However, the solution is obviously typical for a family of problems. The problem of representing sparse matrices was originally selected by the authors of the report on Russell in order to focus on unusual facilities in that language. This paper has shown that the particular problem can be solved with the class definition method proposed by myself. It remains to be seen how the problem will eventually be solved by use of Russell. For the time being the dynamic class interpretation seems to provide better control than does the Russell facilities.

References.

[1] Demers, A. and J. Donahue: "Report on the Programming Language Russell", TR 79-371, Computer Science Department, Cornell University, 1979.

[2] Hoare, C.A.R.: "Proof of corectness of data representation",Acta Informatica, 1, p. 271-281, 1972.

[3] Steensgaard-Madsen, J.: "Classes and Objects - a dynamic approach", TR 78-356, Computer Science Department, Cornell University, 1978.