PROCEDURAL SPECIFICATIONS AND IMPLEMENTATIONS FOR ABSTRACT DATA TYPES

A. L. Furtado and P. A. S. Veloso
Pontificia Universidade Catolica do R. J.
22453 Rio de Janeiro RJ
Brasil

1. Introduction

When working with an abstract data type it is often convenient to have the ability to execute its operations before a final implementation is available [Guttag, Horowitz, Musser '78], [Goguen et al '79].

Here we propose a procedural formalism, which is based on canonical forms and amounts to an implementation of the data type by a data type of terms. It is close enough to the algebraic specification, from which it can be easily derived.

Its simple translation into some string manipulation language (here SNOBOL) provides executable versions of the operations.

2. Canonical form

Throughout the paper we shall use as a running example the well-known (parametric) data type finite sets of elements. It has the following operations

```
EMPTY - creates an empty set
INSERT(X,S) - inserts element X into set S
REMOVE(X,S) - removes element X from set S
HAS(X,S) - tests whether element X belongs to set S
CARD(S) - gives the number of elements in set S
```

We assume that we can test equality of elements via EQ(x,y) and the availability of logical and natural with the usual operations.

Any set-object of the data type is constructed by iterated applications of the operation INSERT and REMOVE to EMPTY. However, distinct manners of construction can yield the same object. For instance , the four terms

```
(a) INSERT(E2,REMOVE(E1,INSERT(E3,INSERT(E1,EMPTY))))
(b) INSERT(E3,INSERT(E2,INSERT(E3,EMPTY)))
(c) INSERT(E3,INSERT(E2,EMPTY))
(d) INSERT(E2,INSERT(E3,EMPTY))
```

represent the set { E2, E3 }.

We want to have a canonical form so that each set has a unique representative from the class of equivalent terms which denote it. Then two terms will be equivalent iff their canonical representatives are (syntactically) identical.

In our simple example it is clear that zero or more INSERT's applied to EMPTY suffice to generate all the set-objects. Thus we can rule out term (a) above. For ordering the various applications of INSERT we place a strict total order (say lexicographic) on the elements and require larger elements to be INSERTed first (as in (d) assuming E2 < E3).

Notice that this does not require the order relation to be available in the data type, nor does it imply that we now have ordered sets. It may be also worth mentioning that the strict ordering criterion prohibits "redundant" applications as in (b). According to the criterion chosen, the canonical term for { E2, E3 } is (d), as in (c) the elements are INSERTed out of order.

Thus, we take as canonical representatives the terms of the form

```
INSERT(E1,INSERT(E2,...,INSERT(En,EMPTY)...))
where E1 < E2 < ... < En, with n ≥ 0
```

3. Algebraic specification

An algebraic presentation for our example, assuming the availability of logical and natural with their usual operations is given in fig. 1. The format is a variation of [Guttag, Horowitz, Musser '78], [Gaudel '77], [Tompa '80]. The equations were obtained by the methodology proposed by [Pequeno, Veloso '79]; they ought to be able to transform each term of the form

operation(canonical terms)

into the canonical form for the result. However, it is important to bear in mind that the algebraic specification, as it stands, has to consist of equations satisfied by any term, canonical or not. That is why we have to consider the possibility of "redundant" INSERT's in the equations, e.g. for CARD.

```
Type Set[Element]
Sorts Set, Element / logical,natural
Operations:
     ( ) -> Set : EMPTY
     (Element, Set) -> Set : INSERT, REMOVE
     (Element, Set) -> logical : HAS
     (Set) -> natural : CARD
     (Element, Element) -> logical : EQ
Axioms
  declare x, y:Element, s:Set
     INSERT(x,INSERT(y,s)) = if EQ(x,y) then INSERT(y,s)
                             else INSERT(y,INSERT(x,s))
     REMOVE(EMPTY) = EMPTY
     REMOVE(x,INSERT(y,s)) = if EQ(x,y) then REMOVE(x,s)
                             else INSERT(y,REMOVE(x,s))
     HAS(x,EMPTY) = F
     HAS(x,INSERT(y,s)) = if EQ(x,y) then T
                          else HAS(x,s)
     CARD(EMPTY) = 0
     CARD(INSERT(x,s)) = if HAS(x,s) then CARD(s)
                         else 1 + CARD(s)
```

Fig. 1: Algebraic presentation

4. Procedural specification

A procedural specification for our example appears in fig. 2. In order to improve readability we write the (canonical) terms with square brackets instead of parentheses and "|" instead of comma. The (high-level) language features are self-explanatory, but perhaps for "?", which stands for any argument, as in PLANNER [Hewitt '72]. Thus,

```
card(INSERT[E2|INSERT[E3|EMPTY]]) => 1 + card(INSERT[E3|EMPTY])
=> ... => 1 + 1 + 0,
```

and

```
insert(E3,INSERT[E2|EMPTY]) => INSERT[E2|insert(E3,EMPTY)] =>
=> INSERT[E2|INSERT[E3|EMPTY]]
```

A striking difference between the algebraic presentation and its procedural counterpart lies in the occurrence of "=>" instead of "=" ; the rewriting rules are now applied in a single direction. This feature, together with appropriate conditionals, ensure that unending computations will not appear. For instance, two calls to insert will have their sequence inverted (if and) only if they are not in the correct order.

```
type Set[Element]

  op empty():Set
     => EMPTY
  endop

  op insert(x:Element,s:Set):Set
     var y:Element,s':Set
     match s
       EMPTY => INSERT[x|EMPTY]
       INSERT[y|s'] => if x = y then s
                       else if x > y then INSERT[y|insert(x,s')]
                            else INSERT[x|s]
       otherwise => fail
     endmatch
  endop

  op remove(x:Element,s:Set):Set
     var y:Element,s':Set
     match s
       EMPTY => EMPTY
       INSERT[y|s'] => if x = y then s'
                       else if x > y then INSERT[y|remove(x,s')]
                            else s
       otherwise => fail
     endmatch
  endop

  op has(x:Element,s:Set):logical
     var y:Element,s':Set
     match s
       EMPTY => F
       INSERT[y|s'] => if x = y then T
                       else if x > y then has(x,s')
                            else F
       otherwise => fail
     endmatch
  endop

  op card(s:Set):natural
     var s':Set
     match s
       EMPTY => 0
       INSERT[?|s'] => 1 + card(s')
       otherwise => fail
     endmatch
  endop

endtype
```

Fig. 2: Procedural specification

Moreover, it is easy to show by induction that all canonical terms and only these will be generated by successive calls starting with a call to empty. Accordingly, the procedures are somewhat simpler than the corresponding equations, since we have made use of the fact that only canonical terms will be received as input parameters (and given as results). Under this assumption, the "otherwise => fail" alternative of the case-like match statement will never be taken.

In the terminology of [Goguen, Thatcher, Wagner '78] the procedures specify the operations of our canonical term algebra. Alternatively, we can regard the procedures as specifying an implementation of our example by the data type consisting of the terms of the language together with term-manipulating operations [Guttag, Horowitz, Musser '78].

5. A SNOBOL implementation

There exist already some computerized aids to be used in connection with algebraic specifications [Guttag, Horowitz, Musser '78], [Goguen et al '79], [Gannon et al '80]. Here we propose the less ambitious, but still quite rewarding, goal of implementation of the procedural specification in some programming language by means of hand translation.

We chose SNOBOL for its wide availability and good pattern-matching features (for an interesting recent alternative, see [Griswold, Hanson '80]). In particular, the BAL feature was found handy to define a pattern ARG matching any argument of the form <operator> '(' <argument list> ')' or consisting of a single constant or variable.

The translation from the procedural specification into SNOBOL functions required only minor adaptations, often stemming from the desire to take advantage of certain powerful features of the language (as in HAS and the usage of "#" for CARDINALITY, for instance). Expressions to be executed are distinguished from terms used for representation by using quotes for the latter.

5. Additional operations

We have included other usual set-theoretic operations (union, set equality, etc.) [Tompa '80], leaving their specifications to the reader. Also included are CHOOSE and RESTRICTION, which deserve some explanation.

CHOOSE corresponds to the nondeterministic instruction x from s, as in [Hoare '72, p. 125], which selects, if possible, some element (to be assigned to x) of set s and

removes it from s. This kind of operation can be described as an "impure" procedure [Guttag, Horowitz, Musser '78] or as a function with a composite range

CHOOSE: Set -> (Element X Set) U { failure },

where failure signals that the set is empty. For a nonempty s the result of CHOOSE(s) is a pair <e,s'> where s' = REMOVE(e,s) and HAS(e,s) = T. We wrote CHOOSE as a SNOBOL function with a side-effect , requiring that the argument be passed by name.

RESTRICTION(p,s) yields the subset of s consisting exactly of those elements of s that satisfy the unary predicate p, viewed as a function from Element into logical. Thus its algebraic specification would be

RESTRICTION(Pred,Set) -> Set

and, with p:Pred, s:Set, x:Element,

```
RESTRICTION(p,EMPTY) = EMPTY
RESTRICTION(p,INSERT(x,s)) = if p(x) then INSERT(x,
                                        RESTRICTION(p,s))
                             else RESTRICTION(p,s)
```

In our SNOBOL implementation, p is passed as an unevaluated expression to RESTRICTION, where it is repeatedly evaluated for each element of the set s.

6. Concluding remarks

The representation of data instances by means of a canonical form appears to support a useful methodology for obtaining not only an algebraic (or axiomatic) presentation but also a procedural specification. The latter, besides being close to the former, allows an easy translation into a programming language, thus permitting early testing and experimental usage.

Implementation by terms provides a way to bridge the gap between two approaches to abstract data types: algebraic presentations, on the one hand, and programming language encapsulating constructs, such as clusters [Liskov et al '77], on the other. In fact, this implementation comes directly from the procedural specification and enables the designer to produce a first cluster, considering only questions of behavior, thus postponing the choice of a representation adequate with respect to efficiency.

We are currently applying these ideas to the area of data bases. We have been able to describe formally the implementation of data base applications by a data model , and the establishment of set-structured access paths.

References

. Gannon, J., McMullin, P., Hamlet, R., Ardis, M. - Testing traversable stacks. SIGPLAN Notices , 15(1), 1980.

. Gaudel,M. C. - Algebraic specification of abstract data types. Res. Rept. 360, IRIA, 1979.

. Goguen, J. A., Thatcher, J. W., Wagner, E. G. - An initial algebra approach to the specification, correctness and implementation of abstract data types. R. T. Yeh (ed). Current trends in programming methodology IV, Prentice-Hall, 1978.

. Goguen, J., Tardo, J., Williamson, N., Zamfir, M. - A practical method for testing algebraic specifications. The UCLA Computer Sci. Dept. Quarterly - vol. 7(1), Jan. 1979.

. Griswold, R. E., Hanson, D. R. - An alternative to the use of patterns in string processing. ACM TOPLAS, 2(2), 1980.

. Guttag, J. V., Horowitz, E., Musser, D.R. - Abstract data types and software validation - CACM, v. 21(12), Dec. 1978.

. Hewitt, C. - Description and theoretical analysis (using schemata) of PLANNER. PhD thesis, MIT, 1972.

. Hoare, C. A. R. - Notes on data structuring. O. Dahl, E. Dijkstra, C. A. R. Hoare (eds). Structured Programming, Academic Press, 1972.

. Liskov, B, Snyder, H., Atkinson, R., Schaffert, C. - Abstraction mechanisms in CLU. CACM 20(8), 1977.

. Pequeno, T. H. C., Veloso, P. A. S. - Do not write more axioms than you have to. Proc. International Computing Symposium, 1978.

. Tompa, T. W. - A practical example of the specification of abstract data types. Acta Informatica, 13, 1980.

```
*                APPENDIX - listing of SNOBOL functions
*
        &FULLSCAN = 1
        &ANCHOR = 1
        OUTPUT('OUTPUT',6,'(1X,72A1)')
        ARG = BREAK(',(') '(' BAL ')' | BREAK(',)')
*
*
        DEFINE('INSERT(X,S)Y,S1')                          :(INSEND)
INSERT  INSERT = IDENT(S,'EMPTY')  'INSERT(' X
+               ',EMPTY)'                                  :S(RETURN)
        S  'INSERT(' ARG . Y ',' ARG . S1 ')'  :F(FRETURN)
           INSERT = IDENT(X,Y)  S                          :S(RETURN)
           INSERT = LGT(X,Y)  'INSERT(' Y ','
+                             INSERT(X,S1) ')'              :S(RETURN)
           INSERT = 'INSERT(' X ',' S ')'                  :(RETURN)
INSEND
*
*
        DEFINE('REMOVE(X,S)Y,S1')                          :(REND)
REMOVE  REMOVE = IDENT(S,'EMPTY'  'EMPTY'                  :S(RETURN)
        S  'INSERT(' ARG . Y ',' ARG . S1 ')'  :F(FRETURN)
           REMOVE = IDENT(X,Y)  S1                         :S(RETURN)
           REMOVE = LGT(X,Y)  'INSERT(' Y ','
+                             REMOVE(X,S1) ')'              :S(RETURN)
           REMOVE = S                                      :(RETURN)
REND
*
*
        DEFINE('HAS(X,S)')                                 :(HEND)
HAS     S  ARB 'INSERT(' X                                 :S(RETURN)F(FRETURN)
HEND
*
*
        DEFINE('CARDINALITY(S)S1')                         :(CAEND)
CARDINALITY CARDINALITY = IDENT(S,'EMPTY')  0              :S(RETURN)
        S  'INSERT(' ARG ',' ARG . S1 ')'                  :F(FRETURN)
           CARDINALITY = 1 +  CARDINALITY(S1)              :(RETURN)
CAEND
        OPSYN('#','CARDINALITY',1)
*
*
        DEFINE('CHOOSE(S)')                                :(CHEND)
CHOOSE  HAS(ARG . CHOICE,$S)                               :F(FRETURN)
        $S = REMOVE(CHOICE,$S)
        CHOOSE = CHOICE                                    :(RETURN)
CHEND
*
*
        DEFINE('UNION(SA,SB)X,S1')                         :(UNEND)
UNION   UNION = IDENT(SA,'EMPTY')  SB                      :S(RETURN)
        SA  'INSERT(' ARG . X ',' ARG . S1 ')'  :F(FRETURN)
           UNION = HAS(X,SB)  UNION(S1,SB)                 :S(RETURN)
           UNION = INSERT(X,UNION(S1,SB))                  :(RETURN)
UNEND
*
```

```
      DEFINE('DIFFERENCE(SA,SB)X,S1')                    :(DIEND)
DIFFERENCE DIFFERENCE = IDENT(SA,'EMPTY')  'EMPTY' :S(RETURN)
           SA  'INSERT(' ARG . X ',' ARG . S1 ')' :F(FRETURN)
               DIFFERENCE = HAS(X,SB)  DIFFERENCE(S1,SB) :S(RETURN)
               DIFFERENCE = INSERT(X,DIFFERENCE(S1,SB))  :(RETURN)
DIEND
*
*
      DEFINE('INTERSECTION(SA,SB)')                      :(INTEND)
INTERSECTION INTERSECTION = DIFFERENCE(SA,DIFFERENCE(SA,SB)) :(RETURN)
INTEND
*
*
      DEFINE('CONTAINS(SA,SB)')                          :(COEND)
CONTAINS   IDENT(DIFFERENCE(SB,SA),'EMPTY')     :S(RETURN)F(FRETURN)
COEND
*
*
      DEFINE('EQSET(SA,SB)')                             :(EQEND)
EQSET      (CONTAINS(SA,SB)  CONTAINS(SB,SA))   :S(RETURN)F(FRETURN)
EQEND
*
*
      DEFINE('RESTRICTION(P,S)X,S1')                     :(RESTEND)
RESTRICTION  RESTRICTION = IDENT(S,'EMPTY')  'EMPTY' :S(RETURN)
           S  'INSERT(' ARG . X . ELMT ',' ARG . S1 ')' :F(FRETURN)
              '' P                                     :F(REST1)
              RESTRICTION = INSERT(X,RESTRICTION(P,S1)) :(RETURN)
REST1         RESTRICTION = RESTRICTION(P,S1)   :(RETURN)
RESTEND
*
* SAMPLE EVALUATIONS
*
      SC = INSERT('E2', INSERT('E1',INSERT('E3',INSERT('E1','EMPTY'))))
      OUTPUT = 'SC = ' SC
      SX1 = RESTRICTION(*DIFFER(ELMT,'E1'),SC)
      OUTPUT = "RESTRICTION(*DIFFER(ELMT,'E1'),SC) = "
      OUTPUT = '   SX1 = ' SX1
      OUTPUT = 'SX1 HAS ' #SX1 ' ELEMENTS :'
      ST = SX1
ITER  OUTPUT = CHOOSE('ST')                              :S(ITER)F(FINIS)
FINIS
      SD = INSERT('E4',INSERT('E1','EMPTY'))
      OUTPUT = 'SD = ' SD
      SX2 = UNION(SC,SD)
      OUTPUT = 'UNION(SC,SD) = '
      OUTPUT = '   SX2 = ' SX2
      SX3 = DIFFERENCE(SC,SD)
      OUTPUT = 'DIFFERENCE(SC,SD) = '
      OUTPUT = '   SX3 = ' SX3
      SX4 = INTERSECTION(SC,SD)
      OUTPUT = 'INTERSECTION(SC,SD) = '
      OUTPUT = '   SX4 = ' SX4
      OUTPUT = CONTAINS(SC,SX1) 'SC CONTAINS SX1'
      OUTPUT = EQSET(SX1,SX3) 'SX1 EQUALS SX3'
END
```

```
SC = INSERT(E1,INSERT(E2,INSERT(E3,EMPTY)))
RESTRICTION(*DIFFER(ELMT,'E1'),SC) =
   SX1 = INSERT(E2,INSERT(E3,EMPTY))
SX1 HAS 2 ELEMENTS :
E2
E3
SD = INSERT(E1,INSERT(E4,EMPTY))
UNION(SC,SD) =
   SX2 = INSERT(E1,INSERT(E2,INSERT(E3,INSERT(E4,EMPTY))))
DIFFERENCE(SC,SD) =
   SX3 = INSERT(E2,INSERT(E3,EMPTY))
INTERSECTION(SC,SD) =
   SX4 = INSERT(E1,EMPTY)
SC CONTAINS SX1
SX1 EQUALS SX3
```