HIERARCHICAL DESIGN AND EFFICIENT IMPLEMENTATION IN SETL : A CASE STUDY.

Edmond Schonberg
Illinois Institute of Technology
Chicago, Ill. 60616

I. INTRODUCTION.

The passage from formal system specification to actual implementation continues to be a subject of intense study, particularly in the context of formal verification systems and of languages with abstract data-types. (see e.g [1], [4], [6]) .The paradigm which emerges from these studies(and which was clearly set forth by Dijkstra [2]) is a follows : one starts from a description of system modules, presented in a very-high level (non-executable) specification language, which is designed to be particularly amenable to formal proof of correctness. This description constitutes the specification of some abstract machine, whose implementation is then produced by successively lower-level machines which realize the primitives of the original one, until the lowest level machine in this sequence is chosen to correspond to an existing set of language primitives. A particularly lucid example of this approach is presented in [5]. The purpose of this short note is to examine the system described therein, and present an approach to its design and implementation in the framework of SETL [3]. This short case study will emphasize the advantages of using set primitives at the specification level, and the ease with which a programming language which includes these primitives (in their most general form) can be made to yield a reasonably efficient implementation by judicious choice of data-structure descriptions. (See [3] for a full description of the SETL data-description sublanguage). The fact that SETL does not include lists among its datatypes,but provides a perfectly general associative store, adds to the interest of this example, whose design goal is the realization of a non-trivial list management system.

II. THE DESIGN GOAL.

The system to be implemented is fully described in [5]. It can be sketched as follows : a hybrid list management system, consisting of two modules : LIST AND ULIST, is to be produced. LIST is a standard list manipulation system, which includes the usual primitives CAR, CDR, CONS, and the predicates ATOMP and ISCELL. ULIST manipulates unique lists, that is to say, the lists created by the ULIST operations are such that no two of them are ever isomorphic. The operations in ULIST are similar to those of LIST, and include UCAR, UCDR, UCONS, ATOMP and ISUCELL. The major difference between the two modules lies in the action of UCONS. UCONS differs from CONS in that, instead of creating a new cell upon each invocation, the execution of UCONS(x,y) will yield the unique cell whose UCAR is x and whose UCDR is y, if such a cell was already created by previous action in ULIST. In this fashion lists are made to share storage, and no redundant copies of list structures are ever produced within this module.

III. SETL DESCRIPTION.

1. The LIST module.

List elements, or cells, are objects characterized by a unique address, or more abstractly by a NAME(called -blank atom- in SETL). Cells are further characterized by the fact that two single-valued relations (maps) are defined on them : CAR and CDR. The range values of these maps are either cells or atomic objects (i.e. objects belonging to a primitive data-type such as Integer, Real, etc. whose internal representation does not concern us). The LIST module is thus fully described by specifying its action on three data-objects : a set CELLS and two maps, CAR and CDR, whose domain is CELLS and whose range is the set CA = CELLS + ATOMS.

The primitive CONS creates a new cell and specifies its CAR and CDR. A new cell is produced by the creation of a new name. This is achieved in SETL by means of the primitive NEWAT (analogous to the LISP primitive GENSYM). The code for CONS follows :

```
PROC CONS(X,Y)  ;

    NEWCELL := NEWAT ;    $ Create new cell.
    CELLS WITH NEWCELL ;  $ insert it in the set of all cells.
    CAR(NEWCELL) := X ;   $ and define its CAR and CDR.
    CDR(NEWCELL) := Y ;
     RETURN NEWCELL ;
END  ;
```

The predicate ISCELL ascertains whether a given object X is an existing cell. This corresponds to performing a membership test for X in CELLS:

```
PROC ISCELL(X) ;   RETURN (X IN  CELLS ) ;   END ;
```

Equivalently, ISCELL(X) can be stated as the following predicate :

```
PROC ISCELL(X) ; RETURN (CAR(X) /= OM  AND CDR(X) /= OM) ; END ;
```

i.e. X is a cell if both its CAR and CDR are defined. (OM is the SETL 'undefined value', obtained when trying to access a map value on a point outside of its domain). Note that if the membership test is implemented in fully protected fashion (i.e. if the test (X IN CELLS) can be performed regardless of the type of X), then there is no need to specify any exception conditions. The same holds for the handling of the maps CAR and CDR: if X is outside of the current domain of definition of CAR, then CAR(X) yields OM. Delicate boundary conditions and exception conditions need not be examined, because the SETL processor itself provides the required environment protection.

2. THE ULIST module.

The cells manipulated by the ULIST module (ucells) are characterized by the fact that for a given pair of values (x,y) (taken from the range values of CAR and CDR) there is at most one cell whose CAR is x and CDR is y. A set UCELLS, is the first data object needed to describe ULIST. The uniqueness condition is expressed by the existence of a single-valued map ,

UCELL : CA x CA -> UCELLS

The predicate ISUCELL(X) is simply a membership test for X in UCELLS. UCONS is implemented as follows :

```
PROC UCONS(X, Y)  ;

    IF (ISUCELL(X) OR ATOMP(X)) AND (ISUCELL(Y) OR ATOMP(Y)) THEN
        IF UCELL(PAIR := [X,Y]) = OM THEN   $ Create new cell.
            NEWCELL := CONS(X,Y) ;          $ Using standard CONS .
            UCELLS WITH NEWCELL;            $ Enlarge set of unique
                                            $ cells.
            UCELL(PAIR) := NEWCELL ;        $ Update map from pairs
                                            $ to unique cells.
        END IF ;                            $ Else such ucell exists
        RETURN UCELL(PAIR) ;                $ already.
    ELSE                                    $ Invalid arguments.
        RETURN OM ;
    END IF ;
END PROC ;
```

Note that the semantics of SETL insure that UCELL is a single-valued map, as long as the only updates on it take the form UCELL(X) := Y ; as in the code above.

Finally, UCAR and UCDR are simply the restrictions of CAR and CDR to a subset of their domain (i.e. UCELLS rather than CELLS). The code for UCONS above insures that Ucells are built only out of atoms or other Ucells (as specified in [5]).

IV. EFFICIENCY CONSIDERATIONS - INTRODUCING BASED REPRESENTATIONS.

The sets CELLS and UCELLS, and the maps CAR, CDR AND UCELL, together with the procedures CONS, UCONS, UCAR, UCDR and the membership predicates ISCELL and ISUCELL, provide an implementation of the proposed system. The correctness of this implementation (which , as it stands, is indeed executable in SETL),follows from the informal remarks made in the previous sections, and from the fact that the SETL primitives realize faithfully the semantics of the theory of finite sets.

The implementation thus produced is clearly very inefficient, both in terms of storage and execution speed since

a) A given cell may appear in as many as 9 instances or copies : as an element of CELLS, an element of UCELLS, an element of the domain or/and range of CAR,etc.

b) To evaluate the CAR or the CDR of a given cell, a general associative retrieval operation must be performed, instead of the simple dereferencing operation which would be used in standard list manipulation sytems.

Both these glaring inefficiencies can be eliminated, by introducing based representations. Very briefly, based representations are specified for a SETL program by introducing special sets, called BASES,in terms of which actual program objects (sets, maps,etc.) are specified. This specification can take a number of forms : a set may be a subset of a base, a map may have a subset of a base for its domain (and/or range), an object may be an element of a base, etc. (see [3] for an extensive discussion of the syntax and semantics of base declarations). The following points can be made :

a) We note first that UCELLS $\subseteq$ CELLS, than DOMAIN(CAR) = DOMAIN(CDR) = CELLS, and that RANGE(UCELL) $\subseteq$ CELLS. This leads us to choose CELLS as a base set, in terms of which the other data-objects can be described.

b) The set UCELLS is used for membership tests (predicate ISUCELL) and for insertions (by means of UCONS). As no other operations are performed on it, it can most economically be represented as a local subset of CELLS, i.e. by attaching to each element of CELLS a single membership bit specifying its presence in UCELLS.

c) The maps CAR and CDR are total over CELLS. They are best described as local maps, i.e. their values are attached to the element block of each cell in CELLS. (This element block can be thought of, in LISP terms, as a fixed-length property list).

The range of CAR and CDR includes both cells and atoms and is simply described as having general (i.e. unrestricted) type.

The representations for our data-objects are now specified by the following declarations :

```
     BASE CELLS  ;

REPR
     UCELLS : LOCAL SET(∈ CELLS ) ;

     CAR, CDR : LOCAL SMAP (∈ CELLS) GENERAL ;
     NEWCELL :  CELLS ;

     UCELL : SMAP ( [GENERAL, GENERAL])∈ CELLS ;

END ;
```

The code for CONS and UCONS is unmodified, except for the removal of the explicit insertion of NEWCELL in CELLS in line 2 of CONS. This insertion is performed automatically by virtue of the declaration for NEWCELL as a element of the base set CELLS and the semantics of base sets. [3].

The effect of the declarations above is to realize for CAR and CDR the usual pointer structure on the set of cells. UCELL remains of course an associative structure, as it must in any efficient implementation. (See the careful discussion of its realization in INTERLISP in [5]) .

V. REMAINING INEFFICIENCIES.

There are three remaining inefficiencies in storage usage in the system we have obtained.

a) The element blocks in CELLS are larger than a list cell need be, because the range of CAR and CDR has been described as an object of general type, and this will force the allocation of a full word for each.Without more precise information about the range values of CAR and CDR, there is no way for the SETL processor to realize that each of these values occupies only the space required for a pointer. In some machines, this may be (tolerably) wasteful.

This situation indicates, however, a weakness in the current set of basing declarations : there is no way to specify that the range of CAR is EITHER an element of the base set CELLS, or an element of the set ATOMS(which might actually be useful as a base set also).Union types are in any case absent from the data-description language of SETL.

b) The element blocks in CELLS are also larger than usual list cells because they hold the unique NAME (produced by NEWAT whenever CONS is invoked)that identifies it. This added field is indispensable if we want to preserve the usual semantics of CONS. The definition of a cell as an ordered pair [X,Y] would lead to a similar implementation of ULIST as the one proposed above, but LIST itself could not be realized, because the set

CELLS would contain no duplicate pairs. In the absence of the concept of Machine address in the language, there is no other way but to create arbitrary tags to refer to different set elements whose structure may be identical.

c) . The third inefficiency appears in the storage of the map UCELL : its domain is a set of pairs [X, Y], where X and Y are the CAR and CDR of some ucell. The values of X and Y must be stored in a pair , as a point in the domain of UCELL, even though these values also appear as the CAR and CDR of the corresponding element block in the base CELLS. Careful hand-coding of the map (associative store) UCELL will use the element block directly, i.e. the entry corresponding to a certain hash-value will point to an element block in CELLS rather than to a separate ordered pair [X,Y]. Such refinements are of course beyond the capabilities of the fixed formats provided by the SETL data-structuring mechanisms.

REFERENCES.
---

[1] ACM conference on language design for reliable software, O Wortman,Ed.(1977)

[2] Dahl, O,J., Dijkstra, E.W., and Hoare, C.A.R : structured programming. Academic Press, New York, 1972.

[3] Dewar,R.B.K., Grand,A., Liu, S.C., Schonberg,E., and Schwartz,J.T.: Programming by refinement, as exemplified by the SETL representation sublanguage. TOPLAS,1,1 (July 1979) to appear.

[4] Liskov,B., and Zilles,S. : specification techniques for data abstraction. IEEE trans. Software Eng. SE-1,1 (march 1975) 7-19.

[5] Spitzen,J, Leavitt, K., and Robinson,L. : An example of hierarchical design and proof. CACM, Vol. 21,no. 12(Dec 1978) pp 1064-1075.

[6] Wulf.,W.a, London,R.L., and Shaw, M. : and introduction to the construction and verification of ALPHARD programs. IEEE trans. Software eng. SE-2,4 (dec. 1976) pp 253-265.