SETL Newsletter #31

May 10, 1971

J. Schwartz

An additional preliminary remark on the importance of "object types" for SETL, with some reflections on the notion of "data structure language".

Newsletter 26 notes that the general 'decision postponement' principle makes it desirable to include various programmer definable "object types" in SETL, and to allow the various basic SETL operations to be interpreted in a manner depending on the type of the object concerned. This point deserves to be underscored, as important issues relating abstract to concrete algorithms are involved.

If programmer definable object types are provided, then fixed-size (or even variable-size) bit-strings, when designated as having some particular object-type, can be taken as the arguments of particular SETL operations. Example: a bit string 2000 words in length and having some appropriate initial layout might be designated as being of the type 'set-of-string-triples-with-hashed-access'. Then the operations

$$a \ \underline{\text{with}}\ x, \ni a,$$

and perhaps

$$(\forall x \in a) \quad ,$$

etc., could be defined for objects of this type. A given SETL algorithm might then run considerably more efficiently if certain of its key objects were merely <u>initialized</u> to be of an appropriate special type, rather than of the standard SETL <u>default</u> type 'Ω'. This could be a large step toward briding the gap between SETL and languages of lower level.

1. Substantially increased efficiency might result from appropriate <u>supplements to</u>, rather than <u>changes in</u>, the text of a given SETL algorithm.

2. A 'supplemented algorithm' consisting of a basic SETL algorithm plus the extra text defining the manner in which the operations of that algorithm were to be realized brings one much closer to the full specification of an algorithm in a lower level

language than does a SETL algorithm unsupplemented. Thus 'object types' may provide an important intermediate step to the 'two-stage' programming technique envisaged in connection with SETL.

Note that 'error returns' having no significance at the basic SETL level may be associated with specialized object-types which in other respects are intended as replacements for more general SETL objects. Thus it may be impossible to insert more than 1000 triples in a given 'set-like' object, it may be impossible to insert any string triple containing more than 10 characters in total, etc. Then error conditions might produce characteristic 'overflow' and 'illegal condition' messages, and terminate execution.

To work out the program that the above remarks suggest would be to build up an 'object-type library' consisting of various useful object types, together with code defining the manner in which the basic SETL operations are to be applied to those object types. Some of the issues and problems which such a plan might imply will be touched upon below. Before that, however, some words concerning optimization, in the situation that would result from the implementation of such a plan. A crude implementation would involve a great many conditional transfers during execution, corresponding to tests for the types of the data objects involved in operations. An optimizer might deduce the types from the code, thereby bypassing many of these tests. Supplementary type-declarations, which could aid the optimizer in its work, would be a reasonable feature. An optimizer ought also look for combinations of operations which can be performed with special efficiency, especially when such combinations are of very frequent use. E.g. indexed stores deserve better treatment than is implied by the sequence

$$f = f\ \underline{\text{lesf}}\ x\ \underline{\text{with}}\ \langle x,a \rangle$$

for $f(x) = a$, etc.

Note also that if the scheme described is to allow gains in efficiency over pure SETL, it is important to provide efficient modes of access to portions of atoms of type bit-string and character-string. The presently specified SETL operations are deficient in this regard. A string-hashing function is desirable also.

Plainly, the object types will have to allow numerical and other type parameters, so that for example should be able to call an object a "hash table of k-character words with 8-bit hash entry to an array of $\ell$ bytes" etc. Another example is "bit-string represented set of objects indexed by table t". Note in this last case the importance of treating union, intersection, and complement directly, and not merely as programmed compound operations: a special case of the observation concerning optimization made above.

Consider as a more specific example of what is envisaged the possibility of declaring a set to be a 'sequence' <u>seq</u> of 'character strings of length at most 10 characters, of total size at most 1000 items, with implicit first pair components'. This might correspond to a SETL character string 10,000 characters long, broken into 10-character fields. The allowable 'element' type for this 'sequence' would then be a pair <n,string>, n being an integer and <u>string</u> a character string no more than 10 characters long. We suppose for the sake of simplicity that the sequence can have no 'gaps'. Then implicitly associated with the sequence is an integer <u>jtop</u> . The various basic SETL operations now should have the following interpretation:

```
A.                  set with <n,string>;
```

means

```
   if n ne (jtop+1) or (len string) gt 10 or n gt 1000
           then return error; else jtop = jtop+1;
      seq[(10*n-9): 10*n ] = string + (10 - len string) * ' ';
      return seq;
```

```
B.                    ∋ seq
```

means

```
   return <jtop, seq[(10*jtop-9): 10*jtop]>;
```

```
C.                    seq less <n,string>
```

means

```
   if n lt jtop then return error; else if n eq 0 then
      then return error; else if n eq jtop then jtop=jtop-1;
      return seq; else return seq;
```

```
D.                    seq(n) = x;
```

means

```
   if n gt (jtop+1) or n lt 0 then callerror;
   else if n eq(jtop+1)then seq = seq with <n,x>; return;
   else seq[(10*n-9): 10*n] = x+(10-len x)* ' ' ;
   return;
```

and so forth.

If it is safe to assume that none of the errors guarded against can actually occur, or if built-in SETL features would in any case give sufficient error indication, some of the error tests might be omitted.

Note that important questions emerge here concerning the situations in which 'independent copies' must be created; questions with profound efficiency implications. A few words will be said about these questions below.

The technique suggested above stands in an interesting relationship to some of the 'memory management' ideas which are under consideration for implementation in LITTLE. 'Low-level' object types of the kind that we have been discussing will normally be represented by bit-strings, either fixed or variable in length, and if variable probably growing by addition at their upper boundary. (Since this is the case efficiently implementable in a lower level language.) Bit strings used in this way may be called *arrays*. One might then imagine declarations (belonging, it is true, to an implementation level sufficiently low as to be barely visible from SETL) which specify that certain of these arrays are to be treated as *merged*, i.e., to be 'grown' and 'shrunk', 'allocated', 'disallocated', 'paged', etc. all within some common area of an underlying memory. Various other aspects of memory treatment might then be specified, as for example stack-like treatment ('window' arrays), inverted stack treatment, access pattern (including number of distinct addresses within a bit-string likely to be current), etc. Standard memory management algorithms might then be applied, allowing tolerably good performance to be reached without a full resort to lower-level programming practices. This approach would in particular provide the following important simplification: one could use a large, perhaps indefinitely large, number of independent arrays, grouping items together in a single array only when some logical relationship between the items required this to be done.

All these considerations serve to raise the question as to whether an intermediate 'data structure language' can usefully be defined. More or less equivalently: is there any semantically useful notion of 'combination' of data structures, on which such a language might be based? This question surely deserves investigation; plainly, such an investigation ought to include a systematic survey of the concrete algorithmic designs appropriate to various of the existing SETL algorithms, to see what phenomena are typical. Similar questions are also implicit in some of the discussions concerning "data description languages" which form part of the literature on data-base systems design.

The following remarks are intended as a preliminary survey of this interesting data-structure language question.

a. If one set f is always accessed by indexing on another set b (i.e., if f appears only in combinations $f(x)$, where $x \in a$) then its elements need not be collectively recorded but may be treated as 'attributes' of the elements of b. More generally, if a set is referenced only in certain restrictive patterns it may be logically unnecessary to record the totality of its elements in any explicit way; it may be sufficient to record certain of its subsets as 'attributes' of other elements.

b. Once all the logical reductions suggested by a are applied to a problem, there results a family of 'essential' sets, which may be considerably smaller than the totality of all sets mentioned in an algorithm. These sets have actually to be represented in some appropriate fashion. They will normally have some kind of 'regular' structure, i.e., consist of elements either all of the same kind or at any rate of some very limited number of kinds, each of these elements in turn having some regular structure.

The general principle involved in point (a) above is worth some comment. Since all stored items are in fact ultimately stored within a word-organized addressable memory, each stored item has in fact an address. (Either a word address or a bit-address.) The address is known when the item is accessed; conversely, the item is accessible when the address is known. This means that with each stored item one somewhat special integer attribute (memory location) may always be associated, this association costing nothing. Any component of the elements of a set which is used merely for addressing these elements can therefore be suppressed in the stored representation of the set.

This suggests, as a possible intermediate logical construct to be used in the development of a data-structure language, the following concept, to which I give the name 'range'. A *range* consists of a collection of logically non-overlapping *items*, each having some unique numerical address. Each item has various *attributes*; attributes can be either *literals*, *references* (to some other range), *multiple* (either *multiple literal* or *multiple reference*), or *flagged*, in which case the attribute can be some combination of the above types, a flag being maintained to show its structure. The basic operatins affecting ranges and items are

i. setting an attribute of an item (if non-multiple)

ii. adding an attribute-value to an attribute (if multiple)

ii'. deleting an attribute-value from an attribute (if multiple)

iii. adding an item to a range

iv. deleting an item from a range.

Note that if operation iv. is provided for a given range, then the existence of some standard garbage-collection scheme is implied.

Part of the data-structure language envisaged would then be a *language of ranges*, which allowed the specification of ranges, of the item-types which a given range could contain, of the attributes of these item types, the physical manner in which these attributes were to be represented, with any special default or common-case conventions, etc. The declaratory data-structure language would then be expanded into standard code-sequences implementing the various basic range-item-attribute relationships.

A second part of the total data structure language would serve to specify the manner in which sets are to be represented by the elements of ranges. The principal possibilites seem to be as follows.

i. Chained (unilaterally or bilaterally) within a range, or within several ranges.

i'. Chained with marks; the items of a set are then all those reachable along a given chain for which a given 'marking' attribute (or combination of marking attributes) has a specified value.

ii. Delimited within a range. The items of a set represented in this way are then all those with addresses lying between two fixed limits.

ii'. Delimited with marks within a range. The items of the set are all those within a delimited section of a range for which a given combination of marking attributes has a specified value.

These two sublanguages used together should suffice to describe most of the principal concrete algorithmic techniques.

For example, a bit-vector technique may be described as the representation of sets by delimited items within a range, these items being multiple fixed-length groups of bits.

To additional observations deserve to be made.

a. If one of the attributes of an item is an integer, we may omit to store this attribute if it is merely used for accessing the item or its value can be calculated from the address of the item within its range. This device will normally be available for *standard sequences*, i.e. sets of pairs whose first element is an integer, and which always includes one and only one pair for each integer n in a certain interval $k_1 \leq n \leq k_2$. In certain special cases, as for example when the items of doubly-indexed arrays are stored within a range in a pattern calculable from certain array-associated 'dimension' parameters, the preceding remark can be generalized to allow the suppression of pairs or triples of integer attributes, etc.

b. Compression will often be secured by representing certain attributes of an item, especially literal attributes, not directly but in an encoded form. The language of ranges should allow for this important possibility. Many encoding techniques are instances of the following general trick: the code for a literal element is the index, in some range, of the (often unique) item having the literal element as one of its attributes. (unique identifying attribute). The use of this trick can of course go together with techniques which expedite the process of finding the item (or sometimes item) which corresponds to a given literal; techniques such as hashing, ordered arrangement, binary or n-ary tree-storage etc.

*Iteration Control*. An important property of sets, as well as other types of compound data structures, is that they may be used as iteration controllers. This iteration-over-subparts possibility ought to be provided for programmer-defined object types, including those 'low-level' object types which might be developed in connection with a range-related data-structure language. A possible scheme for accomplishing this is as follows:

Note first that it is most natural for an iteration controller to return not actual subitems of a structure, but rather to return objects which may conveniently be used as *subitem addresses*. For example, if s is a sequence, we prefer $(\forall n \in s)$ as the associated iteration form, where n is an integer which in the course of the iteration will vary over all the integers $1 \leq n \leq$ #seq, rather than $(\forall x \in s)$, where x is s(n), and also rather than the simple set-theoretic $(\forall$ pair $\in s)$, where pair is <n,s(n)>. Indeed, the hypothetical $(\forall x \in s)$ iteration would make essential aspects of the relationship between successive x's irrecoverable; while $(\forall$ pair $\in s)$ tends toward the clumsy, and might also be too close to the basic set theoretical iteration available anyhow to be worth including as a separate feature. Taking this point as understood, we may go on to require that a function *next* be defined for each data item to be used as an iteration controller. We require this function to have the following properties:

i. When called with parameters next(s,$\Omega$), s being a data item of given type, it will return (the address of) the initial subpart of s.

ii. When called with the parameters next(s,x), x being (the address of) a subpart of s, it will return the next subpart in sequence, or, if no such subpart exists, it will return $\Omega$.

Given such a function, we may regard the iteration

$$(\forall n \in s) \text{ block;}$$

as a shorthand for

n = $\Omega$; (while (n <u>is</u> next(s,n)) <u>ne</u> $\Omega$) block;

Thus <u>iterations are defined for structured objects of a given type by defining the manner in which the function</u> next <u>applies to these objects</u>.

Two examples: for *sequences*, addresses are integers satisfying $1 \leq n \leq$ #seq. In this case, the function next(s,n) is defined by the text body

```
   return if s eq nℓ then Ω else if n eq Ω then 1 else
   if n ℓt #s then n+1 else Ω;
```

For *binary trees*, an address is specified by a sequence of nodes $a_1, a_2, \ldots, a_k$, of which each is a descendant (left or right) of the previous node. Assuming that 'top-left-right' describes the desired order of iteration over nodes, we may take the following text body to define next(tree,a) for iteration over binary trees.

```
   if a  eq Ω then return top tree;
   if (d is ℓ(a(#a))) ne Ω then a(#a+1) = d; return a;;
dright: if (d is r(a(#a))) ne Ω then a(#a+1) = d; return a;;
   (while (n is #a) gt 1) if a(n) = ℓ(a(n-1)) then a(n) =  Ω;
   go to dright; else a(n) = Ω;;
   return Ω; /*as iteration over tree is finished
    if this point is reached */
```

With the conventions suggested, the iterator ($\forall$ a $\in$ tree)block; produce a sequence of tree addresses ranging over all the nodes of *tree*, and applies *block* to each of them in turn.

Note also that other useful SETL operations such as

$$\{e(a),\ a \in \text{tree}\}$$

and

$$\exists\ [a] \in \text{tree} \mid C(a)$$

can be defined in a standard way in terms of the basic iterator.

A further generalization, and one which *brings us into contact with the notion of an infinite set*, is possible. If a set is *never used in an algorithm except as an iteration controller* we need not keep all its members as a fixed totality; a function which generates its elements, in some standard order, each element appearing only once, is all we require. Suppose then that we allow the iterator function *next* to *apply to atoms of type function*, using the very simple code

```
return f(a);
```

when

```
next(f,a)
```

is called in this way. Then any programmed function which never generates two elements twice can be regarded as defining a kind of set, irrespective of whether this set be finite or infinite. If for example we then write

```
definef integers(n); return if n eq Ω then 0 else n+1;
end integers;
```

the statement

$(\forall n \in \text{integers})$ block;

will have its expected meaning. Similarly, a prime-number generator could be used to give meaning to

$(\forall p \in \text{primes})$ block;

etc.

In the SETL notes, various combinatorial generator programs are given. With a suitable generalization of the above technique (to allow the transmission of additional parameters to generator functions) one could employ such useful constructions as

$(\forall \text{ perm} \in \text{perms}(n))$ block;

where *perms*(n) generates the set of all permutations of the first n integers.

Note that this possibility is connected with an optimization also available for finite sets: the iteration

$$(\forall x \in f\{y\}) \text{ block}$$

should for efficiency be taken as

$$(\forall z \in f \mid \underline{\text{hd}}\ z\ \underline{\text{eq}}\ y)\ x = \underline{t\ell}\ z;\ \text{block};$$

to avoid the explicit generation of an unnecessary intermediate set.

If programmer-definable object types are provided, one will want on occasion to regard an object of one type as being of another type. For example, one may occasionally want to regard a binary tree explicitly as a triple consisting of a set of nodes and two descendant maps; conversely, having constructed such a triple, one may want to regard it as a tree. For this purpose, 'type conversion' operations which change the designated type of an object but perform no other transformation on it can be used.

The question of when independent copies of complex objects need to be created can be a vexing one, both in SETL itself and during the transformation of SETL programs to lower-level forms. The SETL interpreter will use a combined reference-count and dead-trace technique to avoid logically unnecessary recopyings wherever unnecessary. An automatic optimizer may of course miss important cases, and for this reason it may be appropriate to provide more directly usable programmer aids for the suppression of unnecessary copying whenever possible. For example, a 'dead' declaration, or perhaps a generalized 'conditionally dead' statement might be desirable. Tools for monitoring the activity of behind-the-scenes copying operations would also be desirable. If SETL is linked to a 'range' language in the manner envisaged above, it may be necessary to establish conventions using which the SETL interpreter can occasionally transmit 'copy structure' signals to the range language interpreter routines.