SETL NEWSLETTER # 200

J.T. Schwartz
February 12, 1978

Possible Additional REPRs for the New SETL System

The new SETL system is now beginning to approach operational status. As the present library of data representations, and the routines which support them become operational, it may become feasible to add various significant new representations, with corresponding code, to the library. Some of these representations may be easily implementable by short routines which invoke existing facilities. This note will comment on two potentially significant representations not currently provided: list and B-trees.

1. List representations

Sets can be represented as lists with bitvector supplements, and tuples can be represented by lists. For sets, we can begin by introducing two new REPRs

local list (∈b)

and remote list (∈b).

A set s having the first of these representations would be represented by a one-way list of its elements, supplemented by one-bit fields in the base (these are exactly the bits that would represent s as a local set.) Similarly, if s has remote list representation, then it is represented by a list, and also by a bit-vector of exactly the sort that would represent s if it were remote set (∈b).

The bits associated with a set support membership and equality testing, whereas the list supports iteration and the from operation. The operation s less x can be performed simply by dropping the bit associated with x; but then one will want to check for deleted elements during iterations over s, and remove them from the list (a list form which allows this test to be merged with the end-of-list test necessary in any case might therefore be appropriate.) The operation s with x tests

the bit associated with x and attaches x to the end of a list if necessary.

For tuples t we can provide

listed tuple mode,

this represented t as a 1-way list. Such lists support efficient iteration, and also concatenation and adjunction of elements. The indexing operation t(i) and indexed assignment t(i):=x one very expensive, but their expense can be reduced by storing the index value i associated with the last list component examined, plus a pointer to the preceeding component, as part of the representation of a listed tuple. Then indexed retrievals t(i) can be performed by comparing i to the current component index, and proceeding forward as many list elements as necessary from the current position. This technique can be especially effective if we use the trick of storing the exclusive or of a backward and a forward pointer in each list element; then either t(i+1) or t(i-1) can be retrieved rapidly once t(i) has been accessed.

It is conceivable that an automatic analyser may be able to detect cases in which the elements added to a set s having list REPR are known (statically) to be outside s, and in which s can always be used destructively. The necessary analysis will not be easy, but will be facilitated by the fact that attempts to establish this kind of disjointness can be focused on sets for which a list REPR has been declared. When these conditions can be established, the bit-flag or bit-vector part of the representation of s can be abandoned, and s can represented as a simple list. In some cases it may even be possible to show that two lists s and s' represent disjoint sets, allowing the union operation s + s' to be implemented, in maximally efficient fashion, simply by concatenating lists.

2. B-tree representations

If a tuple t frequently addressed by insertions and/or deletions, then it may be advantageous to use a B-tree representation for t. If destructive use is permissible, concatenation of tuples and separation of tuples into two parts can also be handled effectively in this representation. Acceptable syntactic representations for these operations are already available:

for element insertion: t(i:0):= [x];
for subtuple insertion: t(i:0):= t';
for subtuple deletion: t(i:) := t(i+j:);
for concatenation: t || t' (or t+t');
for separation into two parts: [t, t']:= [t(1:i), t(i+1:)].

It is possible for the parser to special-case these situations for objects t declared to have B-tree representation.
A plausible declaration form is

tree tuple *mode*.

We must also provide an effective way of searching a tuple having B-tree representation. A plausible approach is to store the index i of the last two components accessed, together with a pointer to the last component accessed, as part of each tuple t having B-tree representation. Then we can provide three additional primitives

right t, left t, and mid t.

Our aim in defining these primitives is to allow an efficient binary search of a sorted vector t having B-tree representation to be written as follows:

```
i := mid t;
(while  i ≠  Ω)
     if key-field (t(i)) = x then
                quit;        /* element found */
     else
```

```
            i := if key-field (t(i)) <x then left t else right t;
        end if;
    end while;
    /* now the condition i ≠ Ω determines whether a key-field */
    /* with value x has been found or not, and the value      */
    /* t(i) is immediately accessible if x has been found     */
```

The logical advantage of proceeding in this way is that no notion of sorted order need be inherent in the B-tree REPR itself.

If we assume a 2-3 tree representation for t, the primitives right and left will act as follows. Suppose that the last addressed component of t is stored as the k-th element a node N of the tree representing t, and that N stores a total of K components of t. (Of course, K has either the value 2 or the value 3, and $1 \leq k \leq K$.) Then

(a) If $k \leq K$, then right t is the k + 1'st element stored in N, unless this was the last component but one accessed, in which case right t is the middle element of the k-th child node N' of N. (But if in this latter case N is a twig, then right t is Ω; if N' exists but stores only two elements of t, then right t is the first of these elements.)

(b) If k = k, then right t is the middle item stored in N' (as in (a)), or, if N' stores only two elements, is the first of these two elements. If N is a twig, so that N' does not exist, then right t is Ω.

(c) The rules for the primitive left t are symmetric to those for right t.

(d) The primitive mid t returns the middle element stored in the topmost node N of the tree representing t (or the first element in N, if N stores only two elements.)

3. More specialised representations

Various specialized representations for set-theoretic objects have come to play a large role in the design of particular classes of high-efficiency algorithms. Some of these might be made available as REPRs. The common obstacle to doing so is that the pattern of operations that these REPRs support can be somewhat fragmentary, and may involve several objects in combination. Here we shall only consider one such specialized REPR, the 'compressed balanced virtual tree representation' for mappings that gives a highly advantageous way of handling equivalence classes. (This structure and its use is described by Aho Hopcroft-Ullman.)

We can put the essential facts connected with this data structure as follows. Let f be a single-valued map, defined on all or part of a set s. We suppose that no sequence of values x, f(x), $f^2(x)$,... will ever cycle; thus every such sequence will end with a unique $y = f^n(x)$ such that f(y) = Ω. Introduce the notation $f^\infty(x)$ for this y (an infix notation for $f^\infty(x)$ might be f lim x.) The REPR we have in mind makes the calculation of $f^\infty(x)$ very fast (essentially, a fixed number of machine cycles), and also supports the following operations:

Retrieval of f(x).

Assignment or reassignment f(x) := y, subject to restrictions described below.

Overall reassignment f := nl or f := h.

The technique is to store both a standard form map representation of f, and a subsidiary 'compressed virtual tree' representation.

This subsidiary representation involves the following objects:

(i) A map *ff* whose domain includes that of f, which is essentially the parent mapping in an auxiliary tree T.

(ii) An integer-valued map *ndesc* with the same domain as f, which is the number of descendants function for T.

(iii) A boolean valued function *flag*, which defines the exact significance of values of ff.

We can evaluate $f^{\infty}(x)$ by executing the loop

```
z := x; (while flag (z)) z := ff (z);;
```

and then by returning ff (z). However, to achieve efficiency, this basic procedure is modified to 'compact' paths in T as they are traversed: Each node encountered along such a tree is made a direct child of the ancestor node $f^{\infty}(x)$ ultimately reached. The full procedure used to evaluate $f^{\infty}(x)$ is therefore

```
z := x; s := nℓ;
(while flag (z));
        s with z;  z := ff (z);
end while;
(∀u∈s)  ff(u)  := z;;
/* and now return if ff(z)≠ Ω then ff(z) else z*/
```

An assignment f(x) := y will only be accepted if $f(x) = \Omega$ (so that no element z with $f(z) = x$ can have $ff(z) \neq x$) and if $x \neq f^{\infty}(y)$ (so that the assignment creates no cycles). The effect of this assignment on ff is to cause either $ff(x) := f^{\infty}(y)$ or $ff(f^{\infty}(y)) := x$ to be executed; of these two possible operations we choose that one which will keep the tree T balanced. The procedure is simply:

```
if f(x) ≠ Ω or (z := f^∞(y)) = x then error;;
f(x) := y;      /* keep the map up to date */
ff(x) := z;     /* keep value f^∞(x) up to date */
if  ndescs(x) ≥ ndescs(z) then  /* make z a T-child of x */
     ndescs(x) + ndescs(z);  /* here take Ω as a code for 1 */
     ff(z) := x; flag(z) := true;  /* x is now the T-parent of z */
else
     ndescs(z) + ndescs(x);  /* make x a T-child of z */
     flag(x) := true;        /* z is now the T-parent of x */
end if;
```

To make this representation available as a REPR, we can simply use

```
limit smap (mode) mode
```

as a generalization of the present smap REPR. A typical case might be

limit smap (∈b) ∈b.

We can also allow the (sparse) form

limit sparse smap (mode) mode.

Generalizations of the compressed balanced tree data structure are also very useful for cases in which values v(x) belonging to an associative semigroup are defined on the nodes x of T, and in which products

$$v(x) \cdot v(f(x)) \cdots v(f^{\infty}(x))$$

need to be calculated with very high efficiency. However, since here three distinct logical objects, namely f, the map v, and the binary operation which combines values v(x) are involved, it is less clear that a situation of this kind can be described simply by a REPR.