SETL Newsletter Number 9

December 1, 1970
Malcolm Harrison

Implementation and Language Design

1. The "long forms" of

A <u>with</u> X

A <u>less</u> X

can be made practically as efficient as the "short forms". This can be done by implementing a set as an ordered tree structure, and permitting a subtree to be common to many sets. Modification of a set is then done by building a new path from the root without changing the original. The resulting tree will have at least $\#A - \log\#A$ nodes in common with the original, and at most $\log\#A$ different internal nodes, assuming balanced trees.

The time to determine if X is a member of A is proportional to $\#A$, and the time to rebuild the appropriate tree path is also proportional to $\#A$. The distinction between the two "forms" is therefore of marginal value.

2. The important property of the above implementation is that the operations do not <u>change</u> existing data-structures but instead build new data-structures with parts of old ones. This presents a different approach to the manipulation of complex data-structures, and has the following properties:

a.) data-structures may overlap, so less memory is required.

b.) assignment can be done without copying.

c.) construction operators need not necessarily copy their component sub-structures.

In general, a structure may be a substructure of more than one other structure, so we have the following additional properties

d.) data-structures should not be modified unless their usage as sub-structures is known.

e.) garbage collection must be used to determine reusable memory.

3. In a tree representation of a set, each set-element relationship is represented by one node. If two sets are not permitted to share memory, the best we can do for memory utilization is an amount of memory which is proportional to the number of such set-element relationships. The more copying is done by the primitive operations the worse this becomes.

If, on the other hand, we permit common subtrees, memory requirements can be reduced considerably. The following strategies can be used to make use of this possibility, in addition to implementing primitive operations as suggested above:

a.) when building a new node, determine if it already exists in memory, and if it does use the old one rather than create a new one.

b.) periodically reorganize structures to use minimum memory - perhaps at garbage collection time.

Note that a.) includes the strategy suggested in Schwartz' original description. I do not know of an optimum algorithm for b.).

4. The above considerations also affect the language design. In particular, the user should be discouraged from writing procedures

which modify data-structures. This can be done conveniently by insisting that arguments are passed by value, not by reference. From the linguistic point of view this implies that functions should be used instead of subroutines. For example, we would write x = f(x,y) instead of call f(x,y). Note that if a variable is a data-type, as suggested below, many of the operations implemented as subroutines and making essential use of call-by-reference can be implemented by passing the variable as an argument, and permitting the function to change the value assigned to this variable. Accordingly, the following changes to the language are suggested:

a.) subroutines be eliminated.

b.) function arguments be passed by value.

c.) the side effects of operations be restricted to assigning new values to variables.

Copies and References

As pointed out by Pat Goldberg in Newsletter Number 2, the ability to handle data structures with common sub-structures is often useful, and can save both execution time and memory. The advantages of this facility would seem to be somewhat less in a high-level language such as SETL, and in some cases we might expect an optimizing compiler to be able to make use of such representations internally without the programmer's knowledge. However, some algorithms are simpler if common sub-structures are permitted. Examples are algebraic manipulation, when it is convenient to do substitution for a variable by modifying a single sub-structure which is referenced many times in the structure; and in program interpretation, in which assignment is conveniently done by a single modification of the structure.

The usual mechanism for providing this facility uses a reference data-type. A reference would be an atom so that when a structure containing a reference is copied the structure it references is not copied. Two additional operations are normally provided. If e is an expression,

<u>ref</u>(e)

would have as its value a reference to the value of e. If r is a reference,

<u>inf</u>(r)

would have as its value the structure referenced.

A generalization of this scheme is used in BALM, and will be available automatically in BALM-SETL. I suggest its incorporation into SETL. Its essential characteristics are:

A <u>variable</u> is a legitimate data-type. It is regarded as an atom, and there are three operations associated with it. If v is an expression whose value is a variable

↑v

gives the current value of the variable, and

↑v = x

changes the value of the variable to x. If s is an expression whose value is a character-string,

<u>variable</u> s

gives the variable whose name is s. This variable is identical with the variable used in the program with the same name, so the commands:

```
ABC = X
$ variable 'ABC' = X
```

are identical in their effect.

With these facilities we can implement a reference as a variable whose name can be either computed by the program, or delivered by a built-in routine such as the GENSYM function of LISP and BALM. The expression <u>inf</u>(r) could be written as $r and the <u>ref</u> function defined as:

```
ref = proc(e), begin(v),
      refvarnumb = refvarnumb + 1,
      v = variable('*' cat dec refvarnumb),
      $v = e, return v
      end end;
unary("ref, "ref, 500);
```

in BALM-SETL, or using <u>GENSYM</u> as

```
ref = proc(e), begin(v), v = gensym( ), $v = e, return v end end;
```

The equivalent in current SETL form is:

```
definef ref e;
        external refvarnumb;
        refvarnumb = refvarnumb + 1;
        v = variable ('*' cat dec refvarnumb);
        $v = e;
        return v;
        end ref;
```