SETL Newsletter # 122B

J. Schwartz
July 24, 1974

Still More Miscellaneous Optimisations

This short newsletter extends NL 122 and 122A.

(a) Existential quantifier extended over set expressions. (peephole)

An existential quantifier appearing in the context $\exists$ y ε *setexpn*|C(y) should be modified to avoid the formation of an explicit set, perhaps in general, but certainly if C(y) is an expression simple enough for its evaluation to be obviously fast. This applies to such cases as

$$\exists y \in \{e(u,v), u\in s, v\in t(u) | .. \} | C(y),$$

which should be compiled as

```
[result = f; (∀uεs, vεt(u) |...)
                    if C(e(u,v)) then result = t; quit;;
                end ∀; return result;]
```

The same remark applies to the construction $\exists$ y ε f{x}, but in this case our remark is a special instance of remark xv of NL 122.

Note also that *setexpn* ne nℓ should be compiled as $\exists$yε *setexpn*; then the optimisation which we have just described can be applied.

(b) 'Overallocation' for tuple and set-valued variables. (Global)

An ovariable o in a SETL program is said to be *potentially growing* if the evaluation of o creates a tuple, string, or set v, and if there exists some ivariable iε du(o) at which the value v will be used destructively in a manner which increases the size of v (its length if v is a tuple or string; the number of its elements if v is a set). If o is potentially growing, an exceptionally large initial block can be allocated when o is evaluated; this can be trimmed when one moves to a program point at which v will no longer be increased. (To 'trim' a set we reduce the size of its hash table, which is a fast operation.) This suggestion is exemplified by the code

```
x = nult;
(while ...) ... x = x + expn;...;;
...f(n) = x;...
```

A large heap block (perhaps a hundred or so words) should be allocated when x = nult is executed; then the *while* loop will probably execute without any reallocation of x becoming necessary. On exit from the loop the block allocated for x ought to be 'trimmed', and its unused portions returned to the garbage collector.

Tuples and sets treated in this way can be allocated blocks of a hundred or so words; for character and bit strings, blocks of several hundred characters or several thousand bits can be allocated. Note that 'extra large' space allocations will only be made for objects which are direct values of (programmer and compiler-generated) set, tuple, or string-valued variables; and only for some of these variables. Hence the amount of space required for these allocations need not be excessive.

If the value of a potentially growing ovariable o is supplied by an extraction operator, as in the example

```
x = f(n);
(while...) ... x = x + expn;...;;
...f(n) = x;
```

then it will not be convenient to attach any allocation operat[illegible] to the evaluation of o. In such cases, we can reserve a large block if reallocation becomes necessary at a subsequent destructive use of the value of o. In the example shown above, this would mean reserving a large block for x if its destructive use in x = x + *expn* forced reallocation.

(c) <u>Finding 'Short' integers.</u> (Global, Uses Typefinder)

Most of the integers used for routine bookeeping operations in SETL will be 'short' in any likely implementation of the language. On the CDC 6600, where integers as large as $.5\times10^{18}$ can be stored, and where sets or tuples are most unlikely to contain as many as 10^6 elements, it is even reasonable to distinguish between very short, short, and long integers. A very short integer is $\leq 10^6$; a short integer is $\leq .5\times10^{18}$. These can be considered as types and found by Tenenbaum's typefinder, if the following rules are applied: #s is always very short, as are constants $\leq 10^6$. If n is short and n' very short, then $n \pm n'$ is short. Upper and lower iteration limits are short. Note that at least $.5\times10^6$ seconds, i.e. at least 100 hours of continuous execution, must pass before 'overflow' of an integer classified as 'short' can occur.

'Backward' typefinding of very short integers becomes possible if we adopt the reasonable rule that an integer which is not very short can never be used to index a tuple or a character string. (Compare also NL. 122, xii; and 122 A .i).

(d) <u>Certain operations involving tuples.</u> (peephole)

The test

```
        <a,b> eq <c,d>
```

can be done as

```
        a eq c and b eq d.
```

More generally,

```
        x eq <c,d>
```

can be done as

```
    if type x ne tupl then f else if (#x) ne 2 then f else
                                 x(1) eq c and x(2) eq d;
```

of course, typefinding may allow some of the clauses of such a test to be eliminated.

(e) <u>Precalculated destination of concatenated tuples.</u> (global)

Code sequences such as

```
      w = <y> + z;
      u = x + w;
```

should be detected, and a test made to see if the 'intermediate' variable w is dead after the 'final' result vector u has been formed. If this is the case, w can be formed directly within the (generously large) space allocated for u.

(f) <u>Tuples formed for transmission and unpacking.</u> (global)

Tuples of known length will sometimes be formed at one point of a program P only to be transmitted to some other point of P and unpacked. For example, this technique may be used to transmit one rather than several arguments to a subprocedure, or to allow use of a functional notation where there are actually several seperate values to be returned.

Cases of this kind can be found, and the components of the tuples involved stored in some appropriate collection of compiler generated variables, no actual tuple being formed in the heap. When applied to subprocedures, this technique will in effect increase the number of subprocedure arguments.

(g) Optimisations depending on the absence of side effects from function calls (global).

After performing a global interprocedural analysis of a program P we will be able to certify that certain expressions E occuring in P can be evaluated without side effect. Information of this kind is useful in various ways. For example, it is needed to optimise existential quantifiers $\exists\, x \in \{e(y),\ y \in s \mid E(y)\}$ in the manner explained above (cf. (b)). If we know that E and E_1 can both be evalusted without side effects, we may be in a position to interchange the order of execution of $x = E;\ x_1 = E_1$ if advantage can be secured by doing so. If E_1 is expersive to evaluate, then the booleans

E or E_1, E and E_1

can be converted to

if E then t else E_1, if E then E_1 else f

respectively.

(h) Applied operator tracing as applied to tuples.

The standardised conventions of the SETL run-time library are seen at their weakest in the sections which deal with sets of tuples. Much of this weakness can be cured by determining the operators which will be applied to tuples (and to sets of tuples) and using code sequences which reflect intended usage.

i. Sets $s = \{t,\ldots\}$ of tuples used only for membership testing should be stored as they would be if $\{t,\ldots\}$ were instead $\{<t>,\ldots\}$; but the hash function applied to tuples t belonging to such sets should be precalculated from all the elements of t and kept with t, rather than being defined simply as the hash of t(1). This same remark applies to tuples generated in order to be made members of such sets.

ii. Set $s = \{t,\ldots\}$ of n-tuples used as functions of k variables should be stored as if they were instead

$$s' = \{<t(1:k), \{u(k+1:), u \in s \,|\, u(1:k) \underline{eq}\ t(1:k)\}>\}$$

Tuples belonging to such sets, and also tuples generated in order to be made members of such sets, should have hashes precalculated from their first k components stored with them. Note that the test $t \in s$ can be made as $t(k+1:) \in s'(t(1:k))$.