SETL Newsletter # 122

J. T. Schwartz
January 20,1974

More Local and Semi-Local SETL Optimisations.

Optimisations may be divided into three classes:

A. *Local* optimisations, which can be performed when certain strictly local code features are detected.

B. *Semi-local* optimisations, which involve only local code transformation but which can only be performed when certain local 'code' features are detected and in addition one has verified that a program to be optimised possesses certain necessary global properties.

C. *Global* optimisations, which involve global program properties and make use of non-local code transformations.

In this newsletter, which continues D. Shields' Newsletter 45 on the same subject, we will enumerate various optimisations of type A and B. Note that optimisations of type A can be built directly into the new SETLB to BALM translator; optimisations of type B must wait intil some degree of global program analysis is available in the SETL system.

To get full benefit even from the strictly local optimisations, they may have to be carried through all the levels of translation which the SETL system is now using. That is, the SETLB to BALM translator may notice a feature, and translate it using a specially introduced BALM operation, which the BALM to LITTLE translator is able to detect and convert into an appropriately efficient LITTLE sequence. As an example of this observation, note that the n appearing in optimisation (iii) below is known to be an integer, and even a 'short' integer; to exploit this fact properly, it must be reflected even at the LITTLE level. A similar remark applies to optimisation (v) below.

I. Local optimisations: (Some of which are quoted from Newsletter 45).

i. Compile ∃{e(x), xεa | C(x)} as
if ∃ xεa | C(x) then e(x) else Ω

ii. Compile yε {e(x), xεa | C(x)} as
temp = f; (∀ xεa |C(x)) if y eq e(x) then temp = t; quit;;

iii. Compile # {xεa | C(x)} as
n = 0; (∀xεa | C(x)) n = n + 1;;

(note that the SRTL level *next* function can be used explicitly)

iv. Calculation of all constant vectors and sets can be moved to an initialisation section; constants being assigned to global names. If possible, this should be done also for constants involving labels; if this is not possible, a 'once only' switch should be prefixed to the calculation of a constant which involves labels.

v. Special equality and inequality tests such as x eq nl, x ne nl, type x eq int etc. should be done directly by efficient BALM, or even by specially added LITTLE macros.

vi. The copy operation normally applied to the first argument of s + t (if s is a set) can be omitted if s is an expression. Since '+' is commutative, this remark applies also if t is an expression. It also applies to s - t if s is an expression.

vii. The union operation

s + {e(x), xεa | C(x)}

can be compiled as

(∀xεa |C(x)) x in s;;

viii. The intersection operation

s ∩ {e(x), xεa | C(x)}

can be compiled as

t = nl; (∀xεa | C(x))if (e(x) is elt) εs then elt in t;;

ix. {xεa | C(x)} ne nl becomes ∃xεa|C(x)

x. set - {e(x), xεa | C(x)} becomes
(∀xεa | C(x)) e(x) out set;;

xi. Multiple assignments of the form

$$\langle a_1, \ldots, a_k \rangle = \langle expn_1, \ldots, expn_k \rangle$$

can be compiled as

$$temp_1 = expn_1; \ldots; temp_{k-1} = expn_{k-1}; a_k = expn_k;$$

$$a_1 = temp_1; \ldots; a_{k-1} = temp_{k-1};$$

Note also that the SETLB syntax should be upgraded to permit constructions of the form <f(i),f(j)> = <x,y>, and should be brought in line with the current definition of multiple assignments.

xii. Quantities, such as do-loop indices, which are known to be 'short' or which can be shown to be short by a test performed on loop entry ought to be handled as LITTLE integers for purposes of incrementation. For example, in compiling

(m $\leq \forall$k $\leq$ n) *body;*

it is acceptable to assume that m and n are $\leq 2^{27}$ and to attempt to produce the following LITTLE code:

```
        m1 = intval(m); /* where intval strips m of its
                           type information */
        n1 = intval(n);
        if |m1| gt 2**27  or |n1| gt 2**27  then error;
loop:   if (m1 gt n1)  then exit;
        K = setlint(m1);  /* where setlint attaches type
                             information to m1 */
```

```
body;
m1 = m1 + 1;
go to loop;
```

The same optimisation can be applied to descending iterations.

xiii. In calculating

$$[\ +:\ x \varepsilon s\ |\ C(x)]\ \langle e(x)\rangle,$$

pre-estimate the number of components of the resulting vector as being equal to # s; pre-allocate a vector of this size, and build up the required result by component insertion, finally throwing away excess space. More generally, when an expression of this form $[\ +:\ x \varepsilon s\ |\ C(x)]\ e(x)$ occurs in the context $[\ +:\ x \varepsilon s\ |\ C(x)]\langle e(x)\rangle + [+:x \varepsilon s_1 | C_1(x)]\langle e_1(x)\rangle$ calculation of the whole ought to be arranged to avoid the creating of unnecessary vector fragments and unnecessary copying.

xiv. Enhance the SETLB syntax so as to allow iterators over tuples, i.e. iterators $(\ \forall\ x(k) \varepsilon t) \ldots$. Iterations of this sort can employ the optimisation described as (xii) above. Moveover, within an iterative loop headed by $(\ \forall x(k)\ \varepsilon t) \ldots$, the indexed access x(k) can be direct and efficient. The same optimisation should of course be applied to forms such as $\exists\, x(k) \varepsilon t, \{e(x),\ x(k) \varepsilon t\ | \ldots\}$, etc.

xv. Iterations of the form $(\ \forall x \varepsilon f\{s\})$ can be handled in a special way. Specifically, the first element address of a tuple whose first element is s can be located, and a routine called to reconstruct the tail of this tuple. Then the present *nextelt* routine can be used to advance the element address, and iteration can proceed until all tuples starting with s have been exhausted. Note that the set f{s} never has to be constructed explicitly.

A similar approach can be applied to iterators $\forall x \in f\{s_1, s_2\}$, and to iterator-based constructions such as $\{e(x),\ x \in f\{s\} | C(x)\}$, etc.

II. Semi-Local Optimisations.

Some of the optimisations noted in this section depend on the availability of live-dead and object-type information. Of course, these are harder optimisations than those described in the preceeding section.

i. Quantities known to be integers might be handled in a special way as follows:
Make available two compile time declarations NOERROR (permitting the supperssion of certain costly types of run-time error checking, and also permitting the use of type information derived by A. Tenenbaum's 'backward' method) and SHORT (declaring that no integers in excess of an implementation-defined limit will occur.) In the presence of these declarations, integers can be maintained as LITTLE integers, and be converted to SETL form only when they are put in sets, become vector components, or assigned to a variable. For LITTLE integers, fast arithmetic comparison operations are available.

ii. Sets which never become set elements or vector components, and which are never assigned to more than one variable, will be dead when the sole variable x of which they are the value becomes dead. Value semantics can be guaranteed for such sets even without maintaining dynamic reference counts. When such sets s are used in constructions such as $s + s_1$, copying of s can be omitted. Similar remarks applies to tuples t appearing in the context $t + t_1$.

iii. In constructions like

$$s = \{e(x),\ x \in a \mid C(x)\}\ ;$$
$$\ldots$$
$$t = s + \{e_1(x),\ x \in a_1 \mid C_1(x)\}\ ;$$

it may be possible to detect that s is used only to form a union, and hence to suppress the explicit construction of s. This same remark applies to sets used only to form intersections.