SETL Newsletter # 162

J.T. Schwartz
January 20, 1976

Improved Target Code Forms Available in the Presence of Global Information Concerning a SETL Program.

This newsletter is a preliminary attempt to define the target code which will be generated by our first optimizing SETL compiler, i.e., on the assumption that information derived from global analysis is available, but that directives for elaborate data structure reorganization is not. The principal forms of information that will be available are as follows:

i. variable types known ;
ii. unnecessary copy operations flagged;
iii. 'reoccurences' of objects known through *crthis* and *crpart* functions;
iv. single valued maps known;
v. strict positivity of integers know in some cases;
vi. relationships of inclusion and membership determined.

Some significant operations for which improved code can be generated are as follows:

1. Tuple Indexing.

(a) In the case v (constant_integer), where the length of v is known to exceed the constant, direct retrieval can be used.

(b) In the case v(j), where the types are known, we can combine the test 'j as short and non-negative' with the test that $j \leq \#\ v$. (It is assumed that short integers have a displaced sign bit on.) The code, in schematic, 6600-like

machine code, is

	cycles	oplength
load R1, vpointer;	1	2
load R2, jvalue;	1	2
load R3, vinfmask;	1	2
mask R4, vinfo;	1	1
R3 = R3 and R4; /* R3 is vlen */	1	1
R4 = R3 - R2;	1	1
if R4 < 0 go to longroutine;	2	2
if R2 = 0 go to longroutine;	2	2
R1 = R1 + R2;	1	1
load R1, desiredcomponent;	1	2
	12 cycles	16
FORTRAN cycles	3	6
FORTRAN factor	4 x	3 x
MIDL cycles	11	14
MIDL factor	1.2 x	1.2 x

If this code is generated offline, 4 cycles will be added, but the code expansion factor will shrink to 1. A reasonable strategy is to compile inline within innermost loops, otherwise compile offline calls. The second test can be omitted if the index is shown to be strictly positive.

The basic typechecking sequence is

	extra cycles
load R1, pointer;	0
load R2, info;	0
mask R3, type;	1
R3 = R3 and R2;	1
shift R3;	1
load address indexed by R3;	1
indexed jump;	2
total checking	6

For some operations two types will have to checked. But in cases like the present, in which there is a 'likely' second operand type which can be implicitly checked, the added work of checking is only 10 cycles (counting 4 cycles for call and return.) The overall timings to be expected should be roughly as follows:

	cycles	FORT. Factor
offline, with typechecking	22	7. 5 x
online, no typechecking	12	4 x
compiled MIDL, with overflow check	12	3. 5 x
FORTRAN	3	

2. Arithmetic, Iteration.

(a) Integer addition. Here the case worth treating as frequent is addition of short positive integers. We can in fact test for the 'extra-short' case (1 guard bit) to exclude overflow in the addition. The code sequence is

	cycles	oplength
load R1, ival;	1	2
load R2, jval;	1	2
R3 = R3 or R2;	1	1
mask R4;	1	1
R3 = R3 and R4;	1	1
ifnonzero R3 go to longroutine;	2	2
R1 = R1 + R2;	1	1
	8 cycles	10
FORTRAN cycles	3	5
FORTRAN factor	3 x	2 x
MIDL cycles, factor	same as FORTRAN	

Overall timings are then:

		FORT factor
offline, with typechecking	18	6 x
online, no typechecking	8	3 x
FORTRAN and MIDL	3	

(b) Integer subtraction. This resembles addition, except that a test must be made for a negative result, adding two cycles. Overall timings are

Offline, with typechecking	20	7 x
Online, no typechecking	10	3 x
FORTRAN and MIDL	3	

(c) Equality test. The following sequence can be used

	Cycles	Code
load R1, i;	1	2
load R2, j;	1	2
R3 = R1 exor R2;	1	1
loadmask R4, typeandvalue;	1	2
R3 = R1 and R4;	1	1
if zero R3 goto equal;	2	2
R3 = R1 and R2;	1	1
shift R3, longbit;	1	1
freg R3 goto longroutine;	2	2
R1 = false;	1	2
	12 cycles	16 (+4)
FORTRAN	5	8

Overall timings are as follows:

Online	12	2.5 x
FORTRAN and MIDL	5	

(d) Integer iteration. The range limits should be checked outside the loop to verify that they are short. If this is done, the iteration overhead will reduce to that of FORTRAN.

(c) Iteration over a set not used as mapping. If as set is not used as a mapping, its elements, including tuple elements, will be stored without application of the transformation of tuples used in the standard SETL representation. An address in the set will then be simply an inyeger index (into its hashtable) and a pointer (to the current list element). The next element procedure is then

```
    loop:       /* start of iteration */
                ... /* loop body */
                ptr = nextfield. ptr;
                if ptr eq nil go to advanceindex;
                load element (ptr);
                go to loop;
advanceindex:   index = index + 1;
                if index gt hashsize go to outofloop;
                load e = hashelt (index);
                if e not void go to loop;
                go to advanceindex;
```

The speed of this is within a factor of two of the ordinary fast iteration over a list.

3. Single valued mappings of one parameter, sets used only for membership testing.

(a) Suppose that f is a single-valued mapping, and that it is the value of a variable which is never either assigned to another variable or made part of another composite, so that the only operations addressing f are f =, f(x) =, and = f(x). Then we can proceed as follows:

(i) Determine all the ovariables o at which elements x that can eventually appear in the context f(x)= or =f(x) are created. Any other one-parameter map g for which an x created at one of these same o can appear in the context g(x)= or =g(x) may be called *cousin* to f. Using the transitive closure of this relationship, we can divide the set of all single-valued maps appearing in our program into equivalence classes C. If an ovariable o is one at which we create an element x that can eventually appear in the context f(x) = or = f(x), where f is some member of the equivalence class C, then we can say that o is *associated with* C.

(ii) Create a standardised *implicit base* s for all the objects x created at an ovariable associated (in the sense just explained) with the equivalence class C. This set s is represented by a hashtable into which objects x created at such o are inserted (or within which they are located) as soon as they are created. We represent each such object x by an *auxiliary block* B, which will consist of 2 n + 1 machine words (or more precisely, of 2 n + 1 fields packed within some appropriate number of machine words, where n is the number of maps f in the equivalence class C. The first word of B will contain the standard SETL representation of x; the next n fields contain the values of f(x) for each of the n f in C; the remaining n words will contain *validating serial numbers* for the stored values f(x) (the way in which these serial numbers are used will be explained immediately below.) A pointer to the auxiliary block representing x is allocated as soon as x is created by evaluation of one of the ovariables o associated with C; this pointer is then used everywhere else as the representation of x.

(iii) Let us write the f-value stored in the block associated with x as *x.fval*, and write the associated serial number as *x.fval-serial*. We use x.fval-serial in conjuction with a global serial number *f.serial* that we associate with f itself, as follows. The integer f.serial is incremented wherever an assignment f = *expn* is made to f. (But indexed assignments to f do not change f.serial.) The lookup operation y = f(x) is performed as follows:

```
if x.fval-serial eq f.serial then return x.fval
   else  y = of(f,x); /* i.e., perform normal SETL hashed lookup *
         x.fval-serial = f.serial;
         x.fval = y; return y;
end if;
```

The assignment operation f(x) = y is performed as follows:

```
x.fval = y;
x.fval-serial = f.serial;
```

(iv) When an ovariable associated with the equivalence class C is encountered and a new value x is created, the auxiliary block B associated with x must be allocated. Whenever this is done, x.fval-serial should be initialised to zero for each f in C; this will force the hashed lookup operation of <f,x> to be performed the first time that the value f(x) is subsequently required. (However, if x is a blank atom, we may instead initialise x.fval-serial to f.serial, and x.fval to Ω, when x is created.)

The machine - level code sequence required for calculation of f(x) is then approximately

	Cycles	Code
load R1, x;	1	2
load R2, x.fval-serial;	1	2
load R3, f.serial;	1	2
R3 = R3 exor R2;	1	1
if notzero R3 goto longroutine;	2	2
load R2, x.fval;	1	2
Total cycles	7	11
FORTRAN cycles	2	4
FORTRAN factor	3.5 x	3
MIDL cycles (without overflow check)	3	6
MIDL factor	3 x	2 x

Overall timings to be expected are therefore as follows:

	Cycles	FORT. factor
Offline	11	5. 5 x
Online	7	3. 5 x
Compliled MIDL, with overflow check	12	6 x
Compiled MIDL, no overflow check	3	1. 5 x.

It appears reasonable to compile online code in inner loops, offline code elsewhere.

The machine level sequence for the store operation is even more favorable, namely

	Cycles	Code
load R1, x;	1	2
load R2, y;	1	2
load R3, f.serial;	1	2
store R2, x.fval;	1	2
store R3, x.fval-serial;	1	2
Total cycles	5	10
FORTRAN cycles	3	6
FORTRAN factor	2 x	2 x
MIDL cycles (without overflow check)	4	8
MIDL factor	1.3 x	1.3 x

It appears reasonable to compile this code online in all cases.

(v) Implicit bases s (cf. point (iii) above) should be treated somewhat differently from other sets during garbage collection: they should not be used as starting points for garbage collector marking. (To ensure this, we have only to ensure that the garbage collector ignores the existence of s during the marking phase.) At the end of the marking phase, all dead (unmarked) elements x should be deleted from every implicit base s. During the garbage collectors relocation phase, it should no longer ignore the existence of the bases s.

This ensures that all elements x which can no longer be referenced and all the associated map values f(x) are deleted from all implicit bases each time that garbage collection occurs.

(vi) If a variable st has a value which is a set used only for insertion and for membership testing, and which is never assigned to any other variable or made part of a large composite, then st may be treated in much the same way as the map f of the preceeding discussion.

Essentially, we treat st as a boolean-valued map which assigns t to each of its elements, but assigns f to non-elements. Any set treated in this way will be a member of some appropriate equivalence class C (cf. paragraph (i) above), and if x is produced at an ovariable associated with C, then the auxiliary block used to represent x will contain both a 1 - bit field which is set to 1 if x belongs to st, and a serial number used to test the validity of this 1 - bit field.

If in addition to insertion and variable testing the set *st* must support iteration, or if *st* is addressed by the ∋ or from operators, then among the *st*-representing fields of x we must include a pointer field allowing the elements of *st* to be grouped together in a 1-way linked list. To delete an element x from *st*, we can simply drop the *st*-associated bit in the representation of x, leaving x temporarily linked into the *st*-representing list; then, when we iterate over *st*, all elements x with dropped *st*-associated bits can be removed from this list.

A straightforward list representation of the sort just described is only available if all direct assignments to *st* have the form st = nl; or, somewhat more generally, st = {y}; since this latter assignment can be treated as the combination

st = nl; y in st.

If more general assignments st = *expn* appear, then to represent st we might have to use a dual structure one part of which was a list while the other represented the value of *expn*; this would be clumsier and less efficient, and is not clearly worth while.

(vii) Space can be saved at the expense of speed by using short serial number fields. Whenever a serial number overflows its field, a garbage-collection-like operation must examine all items containing serial numbers, reduce all non-current serial numbers to zero, and reduce every current serial number to 1.

(viii) Input operations can be handled in the following way, provided that we suppose that the structure of the object read is known. Represent the input operation, in a formal way, using a collection of formal ovariables which represent the various structural levels and components of the object x actually read. That is, use several formal ovariables to represent the 'formation' of the elementary parts of x,several more to represent the parts of x involving these first level sets/tuples as members or components, etc. Then global analysis will associate each one of these formally introduced ovariables with some equivalence class C. Immediately after x is read, its representation should be modified so as to ensure that each part in the total structure that is x is represented by an auxiliary block of appropriate size.

(b) Next suppose that f is a single-valued mapping which is used for assignment to another variable, or which is made an element of some larger composite. Then the following approach, which is not much less advantageous than the treatment just sketched for case (a), can be used. As in case (a), use an auxiliary block B to represent each x which can appear in a context f(x) = ... or ... = f(x). However, in this case the word x.fval of B will contain not the value f(x), but rather will contain a pointer to the pair <x, f(x)> in the standard SETL representation of f. The lookup operation ... = f(x) can then be performed much as in case (a), except that an extra step of indirection is involved, adding 1 cycle and a double-length instruction to the case (a) code sequence. The overall machine level timings associated with calculations of f(x) are therefore approximately

	Cycles	FORT. factor
offline	12	6 x
online	8	4 x.

It appears reasonable to compile online code in inner loops, offline code elsewhere.

In performing the store operation f(x) = y, we must remember to update the pair <x, f(x)> in the standard SETL representation of f. This is necessary, since a standard representation of f must always be available, in case f is either assigned to some other variable or made part of a composite. The machine-level sequence for the store operation is

	Cycles	Code
load R1, x;	1	2
load R2, y;	1	2
load R3, x.fval-serial;	1	2
if R3 = 0 then go to longroutine;	2	2
/*note that R3 = 0 if <x, f(x)> */		
/* must be allocated */		
load R3, fserial;	1	2
store R3, x.fval-serial;	1	2
load R3, x.val	1	2
store R2, (x.val + 1)	1	2
Total cycles	9	16
FORTRAN cycles	3	6
FORTRAN factor	3 x	3 x
MIDL cycles (without overflow check)	4	8
MIDL factor	2 x	2 x

This code could be compiled online in inner loops, offline in other cases. Overall timings would then be as follows:

	Cycles	FORT. factor
Offline	13	4 x
Online	9	3 x
Compiled MIDL, with overflow check	12	4 x
Compiled MIDL, no overflow check	4	1.3 x

(c) Next suppose that f is a possibly multivalued map of one parameter. If this is the case, we expect f to be addressed by the following operations, and only by these:

$$= f(x),\ = f\{x\},\ f(x) = y,\ \langle x,y\rangle \text{ in } f,\ \langle x,y\rangle \in f.$$

Maps of this sort can be treated in a manner closely paralleling the treatment of single-valued sets as already outlined. More specifically, we introduce a single-valued map ff such that ff(x) = f{x} for all x such that f{x} ne nl, and treat the five operations in the preceeding list as

```
= if ff(x)  ne Ω and (# ff(x)) eq 1 then ∋ff(x) else Ω;
= if ff(x) ne Ω then ff(x) else nl;
  ff(x) = {y};
  ff(x) = ff(x) with y;
  if ff(x) eq Ω then false else y ∈ ff(x);
```

respectively. The sets ff(x) will be represented in their ordinary SETL form.

2. The program-graph derivation sequence algorithm as an example of the foregoing.

As an example illustrating the power of the optimization procedures outlined in the preceeding paragraphs, we shall consider a program-graph derivation sequence algorithm like that considered in O.P. II, pp. 269-270; cf. also NL 130, pp. 33 ff, and NL 159, item (d). We take our graph to be defined by a set *nodes*, an entry node *ent*, and a node-to-set-of-nodes map *cesor*. The following is a base form of the algorithm:

```
definef interval (nodes,x); /* calculates the interval with head  x */
int = <x>;
(while ∃y ∈ cesor[range(int)]-range(int) | {z ∈ (nodes-range(int)) |
                                          y ∈ cesor(z)} eq nl)
     int = int + <y>;
end while;
(∀y(n) ∈ int) intov(y) = int;;  /* intov is global */
return int;
end interval;

definef intervals (nodes, ent); /* ent is the program graph entry node*
ints = {interval (nodes, ent)};
(while  ∃nd ∈ cesor [[+: int ∈ ints] range(int) is intnds]-intnds)
               interval (nodes,nd) in ints;
end while;
return ints;
end intervals;
definef dg (nodes, entry);
intov = nl;  /* intov is global */
(∀i ∈ (intervals (nodes, entry) is ints))
     cesor(i) = intov [[+: z(n) ∈ i] cesor(z)-range(int)];
                                   /* cesor is global */
end ∀i;
return ints;
end dg;
```

```
definef dseq (nodes, entry);
<n,e>  =  <nodes, entry>
seq = <<n,e>>;
(while (#(dg(n,e) is der)) lt # n doing <n,e> = seq (#seq);)
     seq (# seq + 1) = <der, intov(e)>;
end while;
return seq;
end dseq;
```

It is reasonable to suppose that formal differentiation will be applied manually to the first three of these routines, giving them the following form (after fusion of the second and third routines.)

```
1   definef interval (nodes, x);
2   int = nult; followsint = nl;
3   nullcount = {x};
4   (while nullcount ne nl)
5        y from nullcount; y in allintnodes; /* all int nodes is golbal */
6        y out followsint; y out allfollows; /* allfollows is global */
7        int = int + <y>;
8        (∀z ∈ cesor(y) | y not ∈ allintnodes)
9             z in followsint; z in allfollows;
10            count(z) = count(z) - 1;  /* count is global */
11            if count(z) eq 0 then z in nullcount;;
12       end ∀z;
13  end while;
14  (∀y(n) ∈ int) intov(y) = int;;
15  (∀z ∈ followsint) count(z) = count(z) + 1;;
16  follows(int) = followsint;  /* follows is global */
17  return int;
18  end interval;
```

```
1    definef dg (nodes, ent);
2    allfollows = {ent}; ints = nl; follows = nl; allintnodes = nl;
3    count = {<y,0>, n ∈ nodes};
4    (∀z ∈ nodes, z ∈ cesor(y)) count(z) = count(z) + 1;;
5    (while allfollows ne nl)
6         nd from allfollows; interval(nodes, nd) in ints;
7    end while;
8    (∀i ∈ ints)
9         cesor(i) = intov [follows(i)];
10   end ∀i;
11   return ints;
12   end dg;
```

The maps (all single-valued) which appears in this code are *cesor*, *intov*, *follows*, and *count*; the sets are *followsint*, *allintnodes*, *allfollows*, *nullcount*, and *ints*. The set *followsint* becomes part of a composite, so (as far as can be told by the crude methods described in the present newsletter) the sets *nullcount*, *allfollows*, and *ints* must all support iterations, and the first two of these sets are arguments of tests s ne nl. Moreover, all direct assignments to these variables have one of the forms s = nl or s = {x}. Therefore, as indicated in (vi), p. 8, elements of *nullcount*, *allfollows*, and *ints* will be linked together in 1-way lists, and counts of the total number of elements of *nullcount* and *allfollows* will be kept.

All the elements x which appear either as indices of *cesor*, *intov*, *follows*, or *count* are either created by execution of line 7 of the *interval* routine appearing above, or are elements of the set *nodes*, created before the outermost routine *dseq* of our group of routines is called. Wherever these elements x are created, we will want to allocate a block large enough to hold fields representing the values cesor(x), intov(x), follows(x), count(x), plus link fields for holding together lists representing *nullcount*, *allfollows*, and *ints*, plus a bit flagging membership in *allintnodes*; plus validating serial numbers as required.

The efficiency-crucial inner loop which appears as lines 8 thru 12 of the *interval* routine contains six instructions. The data structure and target code choices that we have outlined will allow four of these to execute at roughly 1/4 of FORTRAN speeds, while two more will execute in the normal compiled SETL manner, which we estimate as 1/30 of FORTRAN speed. The overall speed of the loop should therefore be substantially better than 1/13 of the speed of a carefully worked out, logically equivalent optimized FORTRAN loop.

An optimization possibility missed by the procedures outlined in this newsletter, but one which would be caught by the more elaborate semi-manual data structure choice system being developed is that of representing the sets follows(int), followsint, and cesor(z) as lists. After this is done no operations of a fully SETL inefficiency level will remain in the inner loop that we have been discussing, and this loop might run at something like 1/4 the speed of a corresponding FORTRAN loop.

3. Bitvectoring

Optimization by the use of bitvectors can be an important technique for the representation of small sets, sets dense in some numerical range, and families of sets on which boolean operations will be performed frequently, provided that all these sets are subsets of some common base. Although this sort of optimization can be handled quite nicely if appropriate structure declarations are used, it does not seem to fit comfortably into the more fully automatic optimization scheme outlined in the preceeding pages. However, a few relatively easy revisions of the present SETL semantic of bitstrings could make direct programming of algorithms using bitvectors relatively easy. What we want are bitstring operations directly analogous to the operations on sets of integers which they would be used to represent, and easy ways of converting between bit and set versions of sets of integers.

The following conventions seem appropriate:

s + s' denotes booleans 'or' of bitstrings (changes existing convention)

s * s' " " " 'and' " (")

s/s' " " " 'exor' " (new convention)

s(n) is 0 if n > length(s) (changes existing convention)

n ∈ s denotes s(n) (new convention)

s with n forms same bitstring as s(n) = 1 (")

s + iset, where s is a bitstring and *iset* a set of integers forms same set as

(∀n ∈ iset) s(n) = 1;; (new convention)

s denotes the number of 1-bits in the bitstring s (changes existing convention)

len s denotes the length of s (new convention)

$s_1 || s_2$ denotes the concatenation of s_1 and s_2 (new convention; should probably be adopted for character strings also)

∀n ∈ s iterates over the n for which s(n) = 1.

f[s] forms {f(n), n ∈ s} for n a bit-string; f must be either a tuple or a map (new convention).

More generally, the operations s + s', s * s', s/s' should be allowed for all cases in which one of the operands is a bit-string and the other is a set of integers he result being either set or a bit-string, depending on the type of the first argument s. Note that, if these conventions are adopted, a set *st* of integers can be converted to a bit-string by writing nulb + st (for which we may prefer to write 'bstring st'), and that a bit-string b may be converted to a set of integers by writing nl + b.

4. A Note on Hardware Implementation: Rough Estimates of the Speed of a 'SETL Machine'.

When a language like SETL is implemented, and assuming that some reasonable degree of optimization has been performed, the code present at run-time will consist of two main parts:

(a) A code text, consisting either of items to be interpreted, or of fully expanded machine level sequences, to be interpreted directly by the hardware. The operations represented directly in this code text will generally be housekeeping operations (such as stack manipulation, and counting operations) which implement the space allocation and call semantics of the language, plus a series of 'elementary' calls to the routines of:

(b) A sizeable run-time library. The routines of this library implement the primitives of the language in all the important variants which an optimizer distinguishes. For efficiency, i.e., to save call overhead, important short fragments of this library may be compiled in-line.

In order for an operation to be worth putting into hardware (or loop-free microcode) it must meet the following requirement:

(i) The body of (ordinary assembly-language) code needed to realise the operation must be small;

(ii) The number of parameters of the operation, the size of the result or results which it produces, and the amount of intermediate data which it requires must all be small.

As may be seen from the code fragments outlined in the preceeding pages, many of the most important routines of the SETL run-time library will consist of short initial sections which test for and dispose of common cases of important operations, and longer 'backing' sections which dispose of more complex but rarer cases.

A reasonable convention for linking hardware implementations of efficiency-critical code sections to longer, less critical sections implemented in software is the following. The parameters of an operation will be loaded into registers (which can either be standard, or can be designated in the operation code); the results will likewise appear in registers. Suppose that the operation can either fail in one of n possible ways, or can succeed. Then execution of the operation will cause a forward skip of n + 1 locations if the operation succeeds; but if it fails in the j-th possible way the address of this skip target will be loaded into a (standard or designated) register, and a skip forward of j locations will be caused.

The internal parallelism of hardware will generally allow any amount of processing consistent with constraints (i) and (ii) above to be performed in one or two hardware cycles. If the number of loads of non-contiguous data required to set up for an operation is n, then the total operation time should be n + 2 cycles (assuming that the operation does not fail.) This implies the following comparisons in the case of the operations considered above.

	SETL mach. cycles	FORTRAN cycles
vector indexing	4	3
integer addition, subtraction	3	3
equality test, short quantities	3	3
map value retrieval	5	2
map value store	5	3

This implies that optimized SETL codes in which operation failure cases were almost totally avoided could run well within a factor of 2 of the speed of their FORTRAN counterparts. Of course, this factor would be cut down seriously if the more complex operation failure had to be executed with any substantial frequency. Of course, intelligently chosen special hardware would speed up the execution of these subsidiary sequences also. Overall, a 1/4 SETL/FORTRAN speed ratio does not seem too much to hope for.

By instrumenting our optimized SETL system suitably, we should be able to make a farly accurate estimate of the percentage of operation failure cases which are typically encountered, and thus to estimate the speed of SETL running on specially designed hardware.