SETL Newsletter # 178

J.T. Schwartz
August 6, 1976

Motion of Range Checks Out of Loops;

Optimization of Integer Arithmetic

Languages of the FORTRAN class given some of their efficiency simply by not checking indexed loads and stores for 'index out of range' conditions. In SETL we can certainly not afford to let all indexed accesses to tuples go without checking (if we did, pointers could be over-written) but the cost of systematic checking is sufficiently high for the elimination of checks by global analysis techniques to be well worth our while. The present newsletter will explore the cases in which range-check operations can be moved out of loops.

To begin with, it is worth examining the machine code sequence, say for the 6600, which represents the operation u = v(i) in the general case. We assume that v is represented by a word held in a register XV, and that this word contains a right-justified pointer to the body of v, plus an additional field LIMIT which will overflow (to become negative) if and only if an integer in access of the limit for v is added to its value (we suppose that the integer value i is known not to exceed 17 bits). The access sequence, where here as throughout we assume that the integer v is known to be no less than 1, is then as follows:

```
        SB4     X1              INTEGER VALUE TO RREG
        LX0     XV, LIMITOFFS   RIGHT-JUSTIFY LIMIT FIELD
        SB3     X0 + B4         B3 NEGATIVE IF OVERFLOW
         NG     B3, OK
        SAU     OMEGADR         LOAD OMEGA
         JP     AROUND
OK      SAU     XV + B4         PERFORM LOAD.
AROUND  BSS     0
```

In numerous cases e.g. in the context u = v(i) + 1,'backwards' type analysis will show that the value Ω is unacceptable for v(i), so that an out of range load can be treated as a fatal error. In this case we can test the out of range condition by forcing a fatal error, and the preceeding sequence simplified to

```
SB4          XI
LX0          LIMITOFFS
SA4          X0 + B4        GENERATE INTERRUPT IF OVERFLOW
SAU          XV + B4
```

This four-instruction sequence is not bad, but FORTRAN would generate a considerably better sequence consisting of not more than the first and last instructions of this sequence. The two middle instructions, i.e.,

```
LX0          LIMITOFFS
SA4          X0 + B4
```

can be considered to assign the value 1 to an auxiliary shadow variable associated with the pair i,v and named iokasvindex. This observation makes it plain that some of the checks that would otherwise be necessary can be eliminated using standard redundancy analysis.

If the value being retrieved by the operation v(i) is known to be an arithmetic quantity, and if this quantity will be used only arithmetically, so that its misuse cannot cause endless looping or lead to any pointer failure in SETL's garbage-collected environment, then a substantially more effective treatment becomes possible. Specifically, we can allow out-of-range references to occur within loops, and attempt to check them only on loop exit. For this to be allowable, we must insist that an out-of-range condition can never occur within the loop unless an out of range condition is detected on loop exit, and conversely that an out-of-range condition must never occur on exit from a loop L unless such a condition also occurs within the loop.

Endless looping aside, the conditions that have just been stated will be implied by the following more easily checked conditions:

i. The length of the tuple v is not changed within the loop L;

ii. The index i is only increased within L;

iii. Any path in L from an instruction which changes the value of i to an exit of L passes through at least one use of i as an index of the tuple v.

As an example of this, consider the loop

```
(1 ≤ ∀i ≤ # v) c(i) = u(i) + v(i);;
```

where we suppose that u and v are known by declaration to be real vectors. This expands as

```
        i = 1.
        lim = #  v
loop:   c(i) = u(i) + v(i);
        if i ≥ lim then go to out;;
        i = i + 1;
        go to loop;
```

Here our conditions (i - iii) are satisfied, and hence the suitability of i as an index of ii and v need not be checked until after loop exit. Note also that the loop-terminator sequence

```
if i ≥ lim then go out;
i = i + 1;
go to loop;
```

can immediately be optimized to

```
i = i + 1;
if i ≤ lim then go to loop;
```

provided that it is remembered that the value of i upon exit from the first sequence is one less than upon exit from the second sequence.

The matrix multiply loop

```
(1 ≤ ∀n ≤ k, 1 ≤ ∀m ≤ ℓ) a(n,m) = [+: 1 ≤ j ≤ n] b(n,j)*c(j,m);;
```

furnishes another signficant example. After expansion, reduction of two-dimensional matrix macros to corresponding one-dimensional form, and strength reduction of integer arithmetic, this loop can become

```
                    (copy a)
                    n = 1;
                    dimcn = 0;
                    dimbn = 0;
loopn:              m = 1;
                    nj = dimbn + 1;
loopm:              s = 0;
                    nm = dimcn + m;
                    jm = m;
                    j = 1;
loopj:              s = s + bb(nj) * cc(jm);
                    if j ≥ r then go to outj;;
                    j = j + 1;
                    jm = jm + dimc;
                    nj = nj + 1;
                    go to loopj;
outj:               aa (nm ) = s;
                    if m ≥ ℓ then go to outm;;
                    m = m + 1;
                    nm = nm + 1;
                    jm = jm + 1;
                    go to loopn;
outm:               if n ≥ k then go to outn;;
                    n = n + 1;
                    dimcn = dimcn + dimc;
                    dimbn = dimbn + dimb;
                    nm = nm + dima;
                    nj = nj + dimb;
                    go to loopm;
outn:               ...
```

The integers which appear in the preceeding code are all initialized using quantities which we may assume to have been declared to be of a size reasonable for vector indices; none of them can be incremented inordinately often. Using these facts its follows readily (by arguments which will be given in additional detail later in the present newsletter) that none of the integers appearing in the preceeding code can be larger than machine word size. Thus all can be carried in machine format, and all integer arithmetic can be done at machine speeds without any special necessity for integer overflow checking. None of the loops shown in the preceeding code change the size of the tuples bb or cc, or diminish any of the indices nj, jm, or nm. Moreover, neither nj nor jm is ever incremented without being used as an index of bb (resp. cc) subsequently, and before exit from the preceeding nest of loops. Hence all range checks for the indices nj and jm can be moved entirely out of this nest of loops.

Range checks for indices appearing in indexed store operations can also be moved, but the justifying arguments needed in this case are somewhat more complex than in the indexed load case. Even though we may not be able to run the risk of overwriting any area that contains pointers, tuples containing untyped reals and integers only can be segregated at the top of memory, so that an out-of-range indexed store cannot destroy any pointer. However, still more serious problems arise in connection with out-of-range indexed stores. In SETL, an out-of-range store (with a non-negative index) is legal, since it can be used legitimately to extend the size of a tuple. Moreover, in making such a store, we may have to check a tuple's share bit, and to perform a copy operation if this share bit is on. (Note that repeated share-bit checking within an important loop can ultimately put one to considerably more expanse than occasional copy operations, which will tend to take place before loops are entered, or on the first iteration of a loop.)

Especially if applied interprocedurally, the technique outlined in newsletter 176 should eliminate most unnecessary share-bit checking (especially by moving copy operations out of loops at compile time.)

Cases in which an out-of-range index store operation can be treated as (movable) error can be defined in the following way. When we write a declaration such as

v: tuple (real) (100),

we mean to assert that no value other than a real tuple, and no value in which there exists a non-Ω component with index in excess of 100, can ever be assigned to v. This clearly excludes out-of-range stores except in situation in which v has been passed as a parameter. Now suppose that v is passed as a parameter to a procedure p, and that

(a) no indexed assignment v(i) = x to v can have an Ω right hand side. Thus no such assignment can diminish the length of v.

(b) Neither p, nor any procedure called directly or indirectly by p, can exit normally, i.e., by a non-error-exit. (We leave open the possibility that these procedures may call an error exit.)

Then if the length of v is increased by p or some procedure called directly or indirectly, an error is inevitable, and will take place no later than upon return from p. Hence we are free to consider such an assignment as an error at its moment of occurence, and to diagnose it as an error at any subsequent moment prior to program exit. This observation allows out of range checks for indexed checks to be moved in much the same way as load checks. Note that an argument much like that which we have just given can also be used to exclude stores of quantities of the wrong type into tuples originally declared to have members of some other type, e.g., to exclude stores of non-real quantities into a t initially declared to be tuple(real).

However, in moving a store check we inevitably permit a store to a somewhat unpredictable heap location. If an upper memory area is reserved for tuples (especially tuples containing untyped reals and integers exclusively) then we can be sure that an out of range store can only affect either a component of a tuple or the stated size, length, or hashcode/hashokbit of a tuple. Call a quantity tuple-dependent if it is obtained either by accessing a component, or the length of some tuple t which might have been declared to contain untyped quantities only, or is obtained by performing a set theoretical operation involving t either as a member, number of member, component, etc. Then if none of the quantities appearing in tests within a loop L are tuple-dependent, it follows that the gross features of the flow of control within L cannot be influenced by out-of-range indexed stores, and consequently range checks for these store can be moved out of L. Out of range indexed store operations occuring within L can therefore be tolerated temporarily, since they cannot prevent exit from the loop L, and a check for the occurence of such a store, with appropriate diagnostic action if one has occured, will in any case take place immediately after exit from L.

Note in particular that all tests occuring in the loops given as examples above are tuple independent; thus for these loops all range checking of tuple indices, whether for loads or stores, can be postponed until loop exit.

In addition to the check-elimination techniques that have been suggested, other methods, based on global analysis, which exploit available facts concerning relative extent of tuples and size of indices can be devised and used to eliminate redundant checks. By combining available approaches in suitable fashion, it should be possible to eliminate almost all of the checks which the SETL tuple semantics would seem to demand, at least in dealing with tuples known *a priori* not to contain any pointers.

2. Optimization of Integer Arithmetic.

If a SETL quantity i is an integer we will generally know this fact, since either typefinding or direct declaration will reveal it. In some cases, we will know upper and lower bounds for i since these bounds can be declared. Of course, declared bounds must occasionally be checked; however techniques like those used in the preceeding section can often be used to move such checks out of loops. An important possibility that arises in connection with the optimization of integer arithmetic is to keep integers i in their standard machine format (up to 59 bits of absolute value in the 6600). For this to be possible, we need to be sure *a priori* that none of the integers to be kept in this format can exceed $2^{59} - 1$ in absolute value. We shall now describe techniques which can be used to establish this fact. Our procedure is as follows:

(1) First, we establish that certain integers do not exceed $2^n - 1$ in absolute value ('n-bit' integers). We can know that this is true for i either because it is declared, in which case the checks that we apply to i will in any case have to validate the declaration, or because every assignment to i has the form i = *const*, where *const* is not too large, or i = j $\pm$ k, where j and k are both known to be integers of n - 1 bits at most, or i = j/k, where j is at most n bits, or i = j * k, where the sum of the lengths of j and k do not exceed n - 1 bits.

(2) Next, we find groups of integers i_1, i_2,... all assignments to which have either one of the forms considered above, or have the form $i_1 = i_2 \pm j$, where the 'increment integer' j has already been shown to have at most n bits. Suppose also that, if assignments of the form $i_1 = i_2 \pm j$ were left out of consideration, the integers i_1, i_2,... could themselves be shown (by application of the arguments of paragraph (1) above) to be of at most n bits.

Then, if we assume that no SETL program executes more than 2^{37} incrementation operations (which would require at least six hours running time on the 6600 and probably much more),at most 2^{37} incrementation could have intervened before all the integers i_k were set by operations other than an incrementation; hence each i_k consists of at most n + 37 bits.

In the matrix multiply example shown in the preceeding section of this newsletter, this argument suffices to establish that all the quantities nj, dimbn, nm, dimcn, and jm are of 54 bits at most. Thus,on the 6600, all these quantities can be carried in their 'hardware' form. Note that (nj, dimbn) and (nm, dimcn) form 'groups' in the sense of the preceeding paragraph.

Within-loop checks of the size of an integer (of declared size) are only necessary if the integer appears as an incrementation constant; and, in the example all such checks can be moved out of all loops.

In some cases it will be important to us to know a priori that a positive integer is 'short' in the sense of the SETL run-time data structures. There may also be significant machine-dependent size limits n, such that integers of n bits or less can be handled with special efficiency. (For example, lengths of 17 and 18 bits can be significant on the 6600). There do exist a few simple cases in which this can be assumed *a priori*. Programmer defined integers will often be supplied with range declarations.When we form the sum or difference of two integers, both supplied with declared limits, then limits will be available for the result. Some operations, and in particular division by constants known to be positive, will reduce an integer's range of variation, and this can be exploited.

Each time a value is assigned to an integer declared to lie in a certain range,we must in principle check the value to ensure that it belongs to this range. In some cases, the fastest way of doing this on the 6600 will be to use an operation which generates a machine exception for integers out of range; in

in other cases, a slower sequence containing a conditional branch operation will have to be used. Note also that in some cases the techniques for moving range checks out of loops that were described earlier in the present newsletter can be adapted and will allow integer range checks to be moved out of loops. This remark applies in particular to range-checking of integers which steadily increase or decrease within a loop.

3. An additional remark concerning 'covariant integers'.

Two integers i,j are said to be 'covariant' in a loop if every basic block within the loop increments each of i, j by the same loop-invariant amount. In this case, the difference of the two integers is loop invariant, and only one,say j, of the integers needs to be carried in the loop. The other, i, can be derived from j by addition. Moreover, if all the contexts in which i is used within the relate i to a loop invariant context, we will often be able to obtain the same information by relating j to a somewhat different, but still loop-invariant context, thus making it unnecessary to culculate i at all. In such situations we will always want to carry the variable, i or j, which it is most difficult to eliminate; and to eliminate the other. By applying this observation (which slightly generalizes the classical technique of 'linear test replacement' described some years ago by Frances Allen) to the matrix multiplication routine shown in section 1, we can obtain the following improved code:

```
                    (copy aa)
                    dimcn = 0;
                    dimbn = 0;
                    dimcnlim = k * dimc  + 2 * dimc;
loopn:              nj = dimbn + 1;
                    nmlim  = l + dimcn;
                    nm = dimcn + 1;
```

```
loopm:      s = 0 ;
            jm = nm - dimcn;
            njlim = nj - 1 + r;
loopj:      s = s + bb(nj) * cc(jm);
            nj = nj + 1;
            jm = jm + dimc;
            if nj ≤ njlim then go to loopj;;
            a(nm) = s;
            nm = nm + 1;
            nj = nj - 1;
            if nm ≤ nmlim then go to loopm;;
            dimcn = dimcn + dimc;
            dimbn = dimbn + dimb;
            nm = nm + (dima - 1);
            jm = jm + dimc;
            if dimcn < dimcnlim  then go to loopm;;
            nj = nj - 1;  /* these instructions prepare */
            nm = nm -dima; /* for the application of   */
                           /* out-of range tests */
```

It is less obvious in this code version than in the preceeding version of the same code that the array indices increase steadily between successive indexing operations; but since this code is equivalent to the preceeding version, it is still true. Hence, it is also true that in-range checks only need to be performed on loop exit.