SETL Newsletter No. 188

Leonard Vanek

May 31, 1977

A LIMITED FORM OF COMMON SUBEXPRESSION ELIMINATION FOR SETL PROGRAMS

Although the complicated algebraic expressions which are the source of the usually-studied form of common subexpression elimination do not appear frequently in SETL code, there is an area in which greater efficiency can be achieved if common subexpressions are removed. In fact, it turns out that at the source level the cause of the original inefficiency is analogous to the cause of most inefficiencies in algebraic languages. For example, in FORTRAN the most troublesome redundant expressions are those used to compute array displacements from lists of subscripts. The analogue to this type of computation in SETL is the operation of hashing to find the value of a map.

Consider the following SETL code, which is taken from actual code written for part of the SETL optimizer.

```
If not crthis{fromoi} subset crthis{i} then
   crthis{i} = crthis{i} + crthis{fromoi};
   changed = true;
end if;
```

With unoptimized code, the expressions crthis{fromoi} and crthis{i} must each be evaluated twice. However, by saving the temporary from the first evaluations, we can avoid repeating the expensive hashing operations.

In order to have a relatively fast common subexpression detection algorithm, we will not try to locate all common subexpressions, but merely those which are most likely to appear and which the programmer might be tempted to remove by explicitly saving the value in a programmer-defined temporary. These expressions are essentially just those that are identical in the source code, although the algorithm may catch a few others as well.

The Method

Unlike Kildall's algorithm [Ki], which detects common subexpressions by building up equivalence classes of expressions which are equal in value at each point in the program, this algorithm constructs equivalence classes of variable occurrences. The immediate motivation for this approach is that the SETL optimizer uses the occurrence chaining map (ffrom or its transitive closure, du) [Va]

[Ki] Kildall, G.A., A Unified Approach to Global Program Optimization, First ACM Symposium on Principles of Programming Languages, 1973.

[Va] Vanek, L.I., Global Analysis Techniques for the Optimizing SETL Compiler, Proceedings of the First Moscow Conference on Very High Level Languages, September 1977.

instead of the flow graph to represent the flow of control of the program being analyzed; thus, it is easy to talk about variable occurrences but difficult to talk about points in the program. Furthermore, this scheme requires only one partition for the entire program, not a partition for every block as Kildall's does.

For completeness, we review the following preliminary definitions.

Definition 1 -

The sets alldefs and alluses are, respectively, the sets of all definitions (occurrences of variables used as outputs in an instruction) and all uses (input occurrences.)

Definition 2 -

For any occurrence, oi, of a variable v, ffrom(oi) is the set of uses of v which can be reached from oi on a path containing no intervening occurrences of v.

Definition 3 -

The map bfrom is the inverse of ffrom.

Definition 4 -

The map du is the transitive closure of ffrom with its domain restricted to alldefs.

Definition 5 -

The map ud is the transitive closure of bfrom with its range restricted to alldefs.

Our algorithm initially puts each occurrence into an equivalence class by itself, and then merges classes when it finds two that must have the same value. If this merging procedure is done just once, examining the occurrences in the order of a topological sort of the flow graph (or the order of the code vector, if you prefer not to spend time sorting the graph), most of the removable redundant computations will be found. This is by no means an optimal strategy because it does not trace the looping structure of the program, but since the kind of situations we are interested in do not involve looping it does quite well.

The initial partition consists of equivalence classes containing a single definition and all the uses which have only that definition linked to them via the ud map. Also in the same class are uses which have more than one definition in common but which, for any execution path passing through both uses, must have been created by the same definition. Clearly in both cases the occurrences in a class must represent the same value - namely the value assigned at the definition. Two examples of such equivalence classes are illustrated in Figure 1. The initializations are accomplished by the following formulas:

RULE 1 -

```
(∀ o ∈ alldefs)
   eqclass(o) = {o} + {i ∈ du(o) | #ud(i) eq 1};
(∀ i ∈ alluses)
    eqclass(i) = if #ud(i) eq 1 then eqclass(arb ud(i))
                 else {i' ∈ du[ud(i)] | ud(i) eq ud(i') and
                                    i reaches instof(i') and
                                    i' reaches instof(i) };
```

The predicate reaches is defined as follows:

Definition 6 -

An occurrence oi reaches an instruction q iff there exists no path to q from the instruction containing oi (but not including either end point) which contains a redefinition of var(oi). It is vacuously true if there is no path from oi to q.

Subsequently, the equivalence classes of two definitions, o_1 and o_2, are merged if the following three conditions are satisfied:

RULE 2 -

1. The quadruples defining the occurrences are -

$$o_1 = \theta(i_{11}, i_{12}, \ldots i_{1n})$$

$$o_2 = \theta(i_{21}, i_{22}, \ldots i_{2n})$$

2. θ is a deterministic operator with no side effects. This means that outputs of operators such as read, random, function call/return, next element of set, and increment cannot have their equivalence classes merged.

3. $\forall$ j=1, 2 ... n: $i_{1j} \in$ eqclass(i_{2j})

Figure 2 illustrates how this class merging procedure works.

Removing Redundant Operations

If two definitions are in the same equivalence class it may or may not be possible to eliminate the computation of one of them. For example, considering the quadruples discussed in (1.) above, the instruction defining o_2 can be replaced by the simple assignment $o_2 = o_1$ only if all paths in the program which lead to o_2 pass through the instruction defining o_1, and var(o_1) is not redefined on any of the paths between the definition of o_1 and the definition of o_2. This condition is guaranteed to to be satisfied if o_1 and o_2 are in the same procedure and var(o_1) is a temporary variable.

Before making a more precise statement of the conditions necessary for removing redundant subexpressions, another predicate must be defined.

Definition 6 -

An instruction q_1 dominates another instruction q_2 iff all execution paths from the entry of the program to q_2 must pass through q_1.

A quadruple is a redundant subexpression if the following condition holds.

RULE 3 -

Let o_1 and o_2 be definitions in the same equivalence class. If instof(o_1) dominates instof(o_2) and o_1 reaches instof(o_2) then the quadruple defining o_2 can be replaced by either

1. the assignment var(o_2) = var(o_1) - if the variables of the two occurrences are distinct, or
2. a no-op instruction - if the variables are the same.

This code elimination will be carried out on an intraprocedural basis because it is not possible to keep a local variable of one procedure available for copying within another procedure.

Reducing Storage Requirements

Because the reaches predicate is a function of both the occurrences and the instructions in a program, it requires a very large amount of storage. In fact, it requires more than is generally feasible for it to be granted for any reasonably large program. Hence, a version of the above formulation which does not depend on reaches would be more useful. Unfortunately, without reaches the algorithm will not be as powerful, but it is still possible to devise such an algorithm.

If we restrict ourselves to only reusing subexpressions which are stored in temporaries, the second use of the predicate can be eliminated. Since temporaries are never reused there is no need to check that a variable has not been killed on the path from where the subexpression is first computed to the instruction where it is recomputed. All that is necessary is that the first instruction dominate the second.

The other application of reaches is less easily done away with and can be accomplished by using a more refined initial partition. With this initialization not as many classes will end up being merged, but the merging procedure will be much less costly.

The classes of occurrences constructed will no longer create a partition of the set of all occurrences. With the new initialization it may happen that two occurrences i' and i are both in class(i) but are not in each other's class. This will happen, for example, when i' and i" are in different alternatives of an "if" statement but can both be reached from i. The new initialization is accomplished by the following SETL code:

RULE 1A -

```
(∀ oi ∈ alluses + alldefs)

   class(oi) = {oi};        i = oi;

   (while #ffrom(i) eq 1 and i notin bfromdead)
    $  Trace unique path forward, ignoring merges,
     $ and stopping at the first fork
      i = arb ffrom(i);        class(i) = class(i) with i;
   end;

   fchain = ffrom(i);   $   use workpile after path diverges

   (while fchain ne nl)
   $   Trace all forward paths, but pass merges only if
    $  all predecesors to the targets of merges are
     $ already in class(oi)
      i from fchain;
      if bfrom(i) subset iclass then
         class(oi) = class(oi) with i;
         fchain = fchain + ffrom(i) - class(oi);
      end if;
   end while;

   i = oi;
```

```
    $   The preceding two loops are repeated with the
    $   forward and backward directions, as well as
    $   references to merges and forks, interchanged.
    $   The reference to bfromdead can be eliminated.

end ∀;
```

Since the classes defined by rule 1A are not equivalence classes, it might seem that the third part of rule 2 may require some modifications. For equivalence classes, $x \in eqclass(y)$ implies that $eqclass(x) = eqclass(y)$ and, hence, that $y \in eqclass(x)$. But for the new classes, it is not necessarily true that $x \in class(y)$ implies $y \in class(x)$. Figure 3 illustrates this. The first two parts of rule 2 remain the same in rule 2A, but part 3 now becomes

3. $\forall\ j=1, 2 \ldots n: i_{1j} \in class(i_{2j})$ or $i_{2j} \in class(i_{1j})$

Also, although it is clear what is meant by merging equivalence classes, it is necessary to define precisely what it means to merge value classes. The following rule accomplishes this.

RULE 4 -

The procedure for merging value classes is

$(\forall\ oi \in class(o_1))\ class(oi) = class(oi) + class(o_2);$

$(\forall\ oi \in class(o_2))\ class(oi) = class(oi) + class(o_1);$

This rule preserves the property that occurrences in the same value class (and lying on the same path of execution) must represent the same value.

Finally, rule 3 simplifies to become the following:

RULE 3A -

Let o_1 and o_2 be definitions in the same value class. If instof(o_1) dominates instof(o_2) and var(o_1) is a temporary variable, then the quadruple defining o_2 can be replaced by

$$\text{var}(o_2) = \text{var}(o_1).$$

Further Improvements

If we allow ourselves the ability to insert code and create new temporaries, it is possible to further liberalize rule 3A. Rather than insist that var(o_1) be a temporary, we create new temporaries when needed to hold the value of a common subexpression until it is again needed. Thus, to rule 3A we add the following:

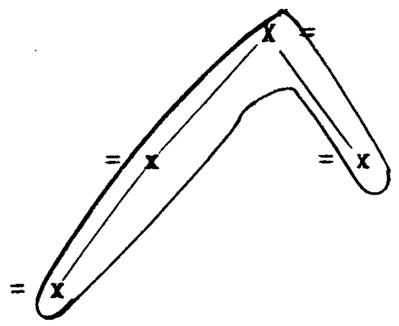
RULE 3B -

Let o_1 and o_2 be definitions in the same value class. If instof(o_1) dominates instof(o_2) and var(o_1) is not a temporary variable, then if the following quadruple is inserted after instof(o_1) on all possible paths from that instruction

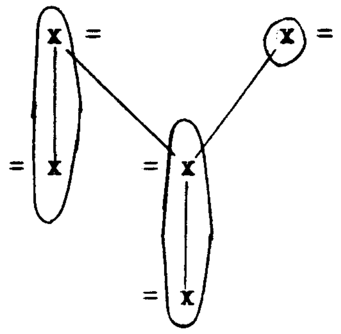
$$newtemp = var(o_1)$$

then the quadruple defining o_2 can be replaced by

$$var(o_2) = newtemp.$$



(a) All Occurrences of x in the same eqivalence class



(b) Definitions in Different Classes from Uses

FIGURE 1: INITIAL EQUIVALENCE CLASSES USING RULE 1

```
x = f1(g1(h1(y)))
        .
        .
        .
z = f2(g2(h2(y)))
```

(a) Source Code - corresponding arguments are equivalent

```
t1 = h1(y)
t2 = g1(t1)
x = f1(t2)
    .
    .
    .
t3 = h2(y)    t3 ∈ class(t1)
t4 = g2(t3)          therefore t4 ∈ class(t2)
z = f2(t4)                  therefore z ∈ class(x)
```

(b) Q1 Quadruples - classes are merged

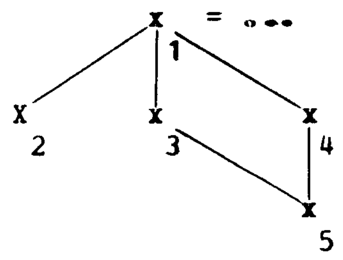
```
t3 = t1   -   useless since t3 and t4 are never used.
t4 = t2   -   both quads can be removed (replaced by no-op.)
z = x     -   assumes that x reaches z and that
              instof(x) dominates instof(z).
```

(c) Best Possible Replacement for Second Computation

FIGURE 2: CLASS MERGER and REDUNDANT CODE REMOVAL

```
1.    x = ...
2.    if P1 then return f(x);
3.    elseif P2 then y = g(x);
4.                  else y = h(x);    end if;
5.    return x+y;
```

(a) Program Segment



(b) "ffrom" Graph for x

$\text{class}(x_1) = \{x_1, x_2, x_3, x_4, x_5\}$

$\text{class}(x_2) = \{x_1, x_2\}$

$\text{class}(x_3) = \{x_1, x_3, x_5\}$

$\text{class}(x_4) = \{x_1, x_4, x_5\}$

$\text{class}(x_5) = \{x_3, x_4, x_5\}$

(c) Value Classes from Rule 1A

FIGURE 3: INITIAL VALUE CLASSES