SETL Newsletter #28 May 17, 1971

An Algorithm for Common Subexpression Elimination and Code Motion

K. Kennedy

This paper will describe an overall common-subexpression elimination algorithm which includes both safety and profitability considerations. The approach we shall use will be a new one: we shall handle all code motion and hoisting by strategically inserting computations at entries to nodes and allowing the common-subexpression elimination routine to do the rest. The algorithm initially 'inserts' computations rather freely, aiming in this way to find useful cases where optimizations of the 'hoisting' type are possible. Uselessly inserted computations are understood to be removed subsequently by a 'dead computation' analysis.

As in all algorithms of this type there will be two passes: an information-gathering pass which works from inner intervals to outer intervals and an optimization pass which uses the information from the first pass to perform optimizing code transformations.

First, we must describe the data structures associated with this algorithm. In keeping with SETL terminology, I shall describe these as sets. The reader should keep in mind that these sets can appropriately be implemented as bit vectors. First, we need some global sets.

1. comps is the set of all computations being considered.
2. safop is the set of all computations which cannot cause an interrupt because of divide check, overflow, etc.

Associated with each block entry we have the following sets. (Note that some of these sets will be available when the optimization algorithm we intend to describe begins to work; others will be calculated by the algorithm. Note also that our algorithm will associate similar sets not only with basic blocks but ultimately with intervals, 'second order' intervals whose elements are intervals, etc.)

1. insert(b) is the set of computations to be inserted at entry to b.
2. avail(b) is the set of computations which will be 'available' on entry to b, given the actual set of computations available on entry to the interval containing b. Note that we say that a computation is available at a given point in a program if, along every path leading to this point, there will be found an instance of the computation not follwed by any reassignment of the value of one of its inputs. This implies that the computation is redundant, since its previously computed value is immediately available for reloading.
3. defavail(b) is the set of computations which would be available on entry to b if no computations were available on entry to the interval containing b.
4. posavail(b) is the set of computations which would be available on entry to b assuming that all computations were available on entry to the interval containing b.
5. safe(b) is the set of computations which may be safely inserted at entry to b because they will not cause any interrupts which would not happen anyway. Thus a computation is in safe(b) if every path from the entry to b contains an instance of that computation before a kill or a program exit.
6. upkill(b) is the set of all computations for which there is an upwards exposed kill (a redefinition of one of the inputs to the calculation before any instance of the calculation) in b.

Associated with blocks and their successors we have the following sets.

1. movable(b,s) is the set of calculations which are upwards exposed on a path through b leading to s and which are movable to the entry to b.
2. nokill(b,s) is the set of calculations which are available on exit to s if they are available on entry to b.
3. expdown(b,s) is the set of calculations which are always available on exit to s from b.
4. nocalc(b,s) is the set of computations for which there is a path through b to s which contains no instance of the computation.

There are a number of important relations among these sets. We express these set-theoretic relationships using SETL 'code-fragments' which will eventually form part of the complete algorithm to be presented below.

First, some relations between the sets of computations 'available' in one or another sense.

1. defavail(b) = [int: p ∈ predecessors(b)]
   (defavail(p) int nokill(p,b) u expdown(p,b))
2. posavail(b) = [int: p ∈ predecessors(b)]
   (posavail(p) int nokill(p,b) u expdown(p,b))

These relations reflect the fact that a computation is available on entry to a block b if for all predecessors p of b it is either available on entry to p and not killed from p to b or it is always available on exit from p to b.

The set of computations actually available on entry to b has a simple expression in terms of these last two sets:

3. avail(b) = posavail(b) int avail(interval) u defavail(b)

The preceding formula is true because of the nature of the posavail and defavail sets. Recall that a computation is in posavail(b) if it is available on entry to the block whenever it is available on entry to the interval. A computation is

in defavail(b) if it is always available on entry to b no matter what was available on entry to the interval. Therefore, a computation is available on entry to b if it is a member of defavail(b) or if it is a member of posavail(b) and is available on entry to the interval.

Next we give a formula determining the set of calculations, which may safely be placed at the entry to b:

4. safe(b) = (comps diff upkill(b)) int
[int: s ∈ successors(b)](comps diff nocalc(b,s) u safe(s))

This says that a computation is safe on entry to b if there is no upwards exposed kill in b and for every successor s of b there is either no path to s which does not contain an instance of the calculation or the computation is safe at s.

Next we give a formula which defines the set of those calculations which, if they are available on entry to an interval, remain available on entry to a successor interval.

5. nokill(interval,sint) =
[int: b ∈ predecessors(head(sint)) int interval]
(posavail(b) int nokill(b,head(sint)) u expdown(b,head(sint)))

This last says that the availability of a computation is not spoiled between entry to an interval and exit to a successor interval sint if for every block of the interval which has an exit to sint the availability of the computation is not spoiled between interval entry and block entry and is also not spoiled in passing through the block to the exit.

Now we give a formula describing these computations which are always available on exit from an interval I to a specified successor, irrespective of whether they are available on entry to I.

6. expdown(interval,sint) =
[int: b ∈ predecessors(head(sint)) int interval]
(defavail(b) int nokill(b,head(sint)) u expdown(b,head(sint)))

This says that a computation is always available on exit from interval to sint if for every predecessor b of head(sint) it is either always available on exit from b or it is always available on entry to b and it is not killed in b.

Next we give a formula which characterizes those calculations which may be moved out of an interval.

7. movable(interval,sint) = [u: b ϵ interval]
(posavail(b) diff defavail(b) int
[u: s ϵ successors(b) int interval|sint ϵ path(s)]movable(b,s))

Here the auxiliary set path(s) is the set of all successor intervals to which there is a path from s. The formula says that a computation is movable from a point on a path in the interval which leads to the successor interval sint if it is movable out of some block b on that path and if the computation is available on entry to b if and only if it is available on entry to the interval. If this condition is satisfied we can insert the computation at the entry to the interval and then be sure that its value will always be available at the point in b at which it was formerly computed.

Now we can give a formula which determines the set of computations to be inserted (on a 'trial' basis) at the entry to a block b. This formula will as a matter of fact only be used to insert code at the entry to intervals; thus the "b" of the formula represents an interval.

8. insert(b) = comps diff avail(b) int
([int: s ϵ successors(b)] movable(b,s) int(safop u safe(b)))

This formula expresses several conditions. First, we do not insert any computation which will be available on entry to b. Subject to that restraint, we insert all computations which are movable out of b and which are safe, either because they cannot, by their nature, cause an error interrupt or because any interrupt they might cause would take place anyway.

Note that we move calculations out of b only if an instance of the calculation appears on a path leading to every successor of b. This restriction is derived from profitability considerations. Ideally, we would only move code out of strongly connected regions. Any computation which is in a strongly connect region

is on a path to a latching node which has a branch to the head node of the interval. Since there is a path from the head to every successor of the interval, any computation in a strongly connected region is on a path to every successor of the interval. Therefore, the restriction never prevents code from moving out of strongly connected regions. It does, however, severely restrict code motion out of other regions of the interval.

The preceding relations concern the principal sets used in the algorithm given below. A few auxiliary sets are also used in this algorithm. The rules governing the calculation of these auxiliary sets are as follows:

9. somepath(b) = [u: pb ∈ predecessors(b)]
(somepath(pb) int nocalc(pb,b))

10. upkill(interval) = upkill(interval) u
(somepath(b) int upkill(b))

11. nocalc(interval,sint) =
[u: pb ∈ predecessors(head(sint)) int interval]
(somepath(pb) int nocalc(pb,head(sint)))

Here, the auxiliary set somepath(b) can be defined as the set of all computations for which there is a calculation-free path (a path crossing no instance of the computation) from the interval entry to the entry to b. Formula 10 merely says that there is an upwards exposed kill of a computation in the interval if there is a calculation-free path for that computation to a block in which there is an upwards exposed kill. Formula 11 is analogous to formula 9 and says that there is a calculation-free path for a computation through the interval to a successor interval sint if one of the predecessors of head(sint) in the interval has a calculation-free path to its entry and a calculation free path through to head(sint).

The relations which have just been described individually are basic to the overall algorithm which is given below. This algorithm is, as we have already said, structured into two passes.

The first pass has two subpasses for each interval. The forward subpass proceeds through the nodes of an interval in interval order and computes posavail and defavail for each block using formulas 1 and 2. This subpass also computes upkill and nocalc for the interval. The backward subpass goes through the nodes of the interval in reverse interval order updating posavail for each block and finding computations which are movable out of the interval (using formula 7).

These two passes are contained in the routine process which has as its only explicit formal parameter an interval. This will be passed as a SETL sequence (the nodes of the interval in interval order with interval(1) representing the head).

Here then is the SETL algorithm.

```
define process(interval); external successors,
   predecessors, movable, nocalc, nokill, expdown,
   upkill, defavail, posavail, comps; somepath = nl
   head = interval(1); somepath(head) = comps;
   upkill(interval) = nl; defavail(head) = nl,
   posavail(head) = comps;
/* the forward pass to get defavail, posavail,
   upkill, and somepath */
(2 ≤ ∀i ≤ #interval) b = interval(i); pred = predecessors(b);
   defavail(b) = [int: pb ∈ pred]
   (defavail(pb)  int nokill(pb,b) u expdown(pb,b));
   posavail(b) = [int: pb ∈ pred]
    (posavail(pb) int nokill(pb,b) u expdown(pb,b));
   somepath(b) = [u: pb ∈ pred]
    (somepath(pb)int  nocalc(pb,b));
   upkill(interval) = upkill(interval) u
    (somepath(b) int upkill(b)); end ∀i;
```

```
/* some initialization for pass2 */
   intnodes = tℓ[interval]; path = nℓ; latchpath = nℓ;
   sints = successors(interval);
   (∀y ∈ sints) movable(interval,y) = nℓ; end ∀y;
   latch = [int: b ∈ predecessors(head) int intnodes]
           posavail(b);
/* the backward pass to update posavail and calculate movable*/
 (#interval ≥ ∀i ≥ 1) b = interval(i); path(b) = nℓ;
   succint = successors(b) int(intnodes less head);
   latchpath(b) = (if head ∈ successors(b) then t else f)
           or [or: s ∈ succint] latchpath(s);
/* if latchpath(b) is true then b is in the strongly
   connected region and there is a path to every successor */
 (∀y ∈ sints | y(1) ∈ successors(b) or latchpath(b))
           y in path(b); end ∀y;
   path(b) = path(b) u [u: s ∈ succint] path(s);
   posavail(b) = posavail(b) int latch u defavail(b);
   diffset = posavail(b) diff defavail(b);
   (∀s ∈ successors(b)) if (∃[y] ∈ sints | s eq y(1))
     then movable(interval,y) = movable(interval,y) u
       (diffset int movable(b,s)); else if s eq head
     then (∀y ∈ sints)movable(interval,y) = movable(interval,y)
       u (diffset int movable(b,s)); end ∀y;
     else (∀y ∈ path(s)) movable(interval,y) = movable(interval,y)
       u (diffset int movable(b,s)); end ∀y;
      end ∀s;  end ∀i;
/* final calculations to get desired sets */
 (∀y ∈ sints) predint = predecessors(y(1)) int intnodes;
   expdown(interval,y) = [int: pb ∈ predint]
     (defavail(pb) int nokill(pb,y(1)) u expdown(pb,y(1)));
   nokill(interval,y) = [int: pb ∈ predint]
     (posavail(pb) int nokill(pb,y(1)) u expdown(pb,y(1)));
   nocalc(interval,y) = [u: pb ∈ predint]
     (somepath(pb) int nocalc(pb,y(1)); end ∀y; return; end process;
```

The driving routine for the inner-outer process begins with a sequence of intervals going from basic intervals to more complex so that intervals(#intervals) is the single interval which includes the entire control flow graph.

```
define inout(intervals);
   (1 ≤ ∀i ≤ #intervals) process(intervals(i)); end ∀i;
         return; end inout;
```

When this routine returns we have all the information we need for the outer-inner pass. This pass will treat the intervals in reverse order, doing insertions and deletions as needed.

```
define outin(intervals); external exits, safe,
   upkill, nocalc, comps, avail;
   avail = nℓ; avail(intervals(#intervals))=nℓ; safe=nℓ
   (∀b ∈ exits) safe(b) = nℓ;
   safe(interval(#intervals)) = comps diff
     (upkill(interval(#intervals)) u [u: b ∈ exits]
           nocalc(interval(#intervals,b)));
   (#intervals ≥ i ≥ 1) optcess(intervals(i)); end ∀i;
     return; end outin;
```

The routine "optcess" computes the insert and new avail sets for the interval and then computes safe and avail sets for each block of the interval. It does this in a backwards pass through the nodes of the interval.

```
define optcess(interval); external insert, avail,
   successors, predecessors, nocalc, upkill,
   comps, defavail, posavail, movable, safop;
/* calculate insert and modified avail sets */
   insert(interval) = comps diff avail(interval)
      int([int: s ∈ successors(interval)]
           movable(interval,s) int(safop u safe(interval)));
   newavail = avail(interval) u insert(interval);
/* find avail and safe vectors for each block */
   safe(interval(1)) = safe(interval);
   avail(interval(1)) = newavail;
 (#intervals ≥ ∀i ≥ 2) b = interval(i);
   avail(b) = newavail int posavail(b) u defavail(b);
   safe(b) = (comps diff upkill(b)) int
    [int: s ∈ successors(b)] (comps diff nocalc(b,s)
     u if (∃[y] ∈ successors(interval) | s eq y(1))
           then safe(y) else safe(s)); end ∀i;
   return; end optcess;
```

This routine will provide us with the insert and avail sets for every interval and will prepare us for the basic block optimization.