SETL Newsletter No. 111 — August 7, 1973

Global Dead Computation Elimination

K. Kennedy

As a result of optimizations such as reduction in strength coupled with linear function test replacement, many instructions in a program may compute values which are never used. To really reap the advantages of earlier optimizations these computations should be eliminated at some point. This newsletter proposes a global scheme which makes use of "use-definition chains" described in other SETL Newsletters [1,2].

Intermediate Code

We assume an intermediate code of the form proposed in SETL Newsletter #102 [3] where each instruction is represented by a unique blank atom along with several mappings:

1. *op(inst)*	the operation code
2. *targ(inst)*	the name of the target variable
3. *args(inst)*	a tuple containing the names of the arguments to the instruction
4. *next(inst)*	the next instruction to be taken.

In addition to these functions we propose two more functions.

5. *do(inst)*	a logical variable which indicates whether or not the instruction is dead. If *do(inst)* has the value "true" the instruction is not dead; if it has the value "false" it is dead and can be eliminated.
6. *reaches(inst)*	the set of instructions in the program whose values can reach the instruction *at* without passing through a subsequent definition of the value.

The Basic Idea

We will assume that all output is done through calls to subroutines. With this in mind we must list the set of values which are always live.

1. All values which appear as arguments to subroutines are assumed to be live at the call to that subroutine.
2. All values which are arguments to conditional branch instructions are live at the branch instruction.

The reasoning behind this algorithm is simple: any instruction which computes a value used by a live instruction is itself live. Initially we assume all instructions to be dead. Into the set *livecomps* we gather all subroutine calls, function calls, and conditional branches, which we know to be live from the discussions above. The algorithm then proceeds by removing an instruction from *livecomps*, setting its do-flag, and adding to *livecomps* any instructions which may compute values used by this instruction. Specifically, if the selected instruction is *inst*, we examine each instruction in *reaches(inst)* and add any such instruction whose do-flag has not been set and which sets an argument of *inst*. When *livecomps* is exhausted, the do-flags of all live computations have been set.

The following SETL routine will perform this computation. We assume that the set *program* contains all nodes in the program and that all functions on these nodes except *do* are defined prior to entry.

```
define deadcomp(program);
   /* targ, op, args, do, and reaches are externally defined */
   /* first set all do-flags to false and collect instructions
      which are subroutine calls or conditional branches into
      livecomps */
   livecomps = nℓ;
   (∀n ∈ program)
      do(n) = f;
      if op(n) ∈ {bsr,bfn brc}
         then livecomps = livecomps with n;;
      if op(n) ∈ {br,hlt} then
              do(n) = t;;   end ∀n;
```

```
/* now  proceed through livecomps examining each argument */
(while livecomps ne nℓ)
   x from livecomps;  a = args(x);
   do(x) = t;
   (1 ≤ ∀i ≤ #a,  y ∈ reaches(x)|targ(y)=a(i) and do(y)=f)
   livecomps = livecomps with y;
      end ∀i;  end while;
return;  end deadcomp;
```

This elegant and simple routine has a major disadvantage: the sets *reaches(x)* are defined for every instruction in the program and may be very large. In the interests of storage economy, we will sacrifice elegance and efficiency to use a more compact form of the use-definition information.

Block-Level Method

To achieve a more compact representation, we must take the control-flow structure of the program into account. Let us assume that a control-flow analysis pass has provided us with a set *blocks* of one blank atom for each basic block in the program. The function *contents(b)*, defined for each b ∈ *blocks*, denotes the set of instructions in the block b. Because of the linear nature of basic blocks, each argument of a given instruction is computed by a unique instruction earlier in the block, or by one of a number of instructions outside the block. A new function defined on each instruction *inst* expresses this observation.

> *computedby(inst)* is a tuple of the same length as *args(inst)*. The i-th component of *computedby(inst)* is the source of the corresponding component of *args(inst)*. If the argument is computed by an instruction in the same block the component of *computedby(inst)* is that instruction; if the argument is computed outside the block then the associated component of *computedby(inst)* is the blank atom representing the block itself. This function can be determined by a preliminary scan of the program text.

The function *reaches*, previously defined for each instruction, is now defined only for blocks. If b is a block, then reaches(b) contains the set of all instructions in the program whose computed value can "reach" the entry of b. In other words, if there is a path from the definition to the entry of b which contains no redefinition of the value defined then that definition is in *reaches(b)*.
The live instruction marking algorithm now proceeds as follows.

1. Begin with the set *livecomps* of all initially live instructions (subroutine calls, function calls, conditional branches).
2. Select and remove an element x from *livecomps*. The element may be an instruction or a pair. If it is an instruction
   a) mark it, i.e., set do(x) = t;
   b) for each argument of x, if the argument is computed by an instruction y within the same block and do(y) eq f, then add y to *livecomps*. Otherwise, the argument must be computed outside the block, in which case form the pair *<block,arg>* (the block node and the argument) and add it to *livecomps*.
3. If x (the element selected) is a pair *<block,value>* ,
   a) form the set *comps* of all instructions in *reaches(block)* which compute *value*.
   b) add each element y of *comps* for which *do(y)* eq f to *livecomps*.
4. If *livecomps* is exhausted then stop.
   Otherwise, repeat steps 2 through 4.

The advantage of this method is that the *reaches* sets need be maintained only for each block rather than for each instruction -- a substantial reduction in the size of this data structure.
The *reaches* sets required here can be computed by an algorithm given in SETL Newsletter #37 [1].

Here is the SETL code for the block-level method. It requires the sets *program* and *blocks* as arguments.

```
define deadcomp(program,blocks);
   /* targ, op, args, do, computedby, and reaches
      are external quantities */
   /* first set all do-flags to false and collect initially
      live instructions into livecomps */
   livecomps = nℓ;
   (∀n ∈ program)  do(n) = f;
      if op(n) ∈ {bsr,bfn,brc}
         then livecomps = livecomps with n;;
      /* we always perform branches and halts */
      if op(n) ∈ {br,hlt} then do(n) = t;;
   end ∀n;
   /* now pass through livecomps applying steps 2 through 4
      of the algorithm above */
   (while livecomps ne nℓ)
      x from livecomps;
      if pair x then
         /* get block and value */
         <b,val> = x;
         /* add all instructions which set val to livecomps */
         livecomps = livecomps+{n∈reaches(b)|targ(n) eq val
                                       and do(n) eq f};
         else /* x is an instruction */
            do(x) = t;  arglist = args(x);
            complist = computedby(x)
            /* pass through arguments */
            (1 ≤ ∀i ≤ #arglist)
               val = arglist(i);  /* the value */
               new = complist(i); /* the computing inst */
               if new ∈ blocks then /* add a new pair */
                  livecomps=livecomps with <new,val>;
               else if do(new) eq f then
                  /* add an instruction */
                  livecomps=livecomps with new;
               end if new;
            end ∀i;
         end if pair x;
      end while livecomps;
      return;  /* all flags set */
end deadcomp;
```

The *reaches* sets used by this algorithm may still be too large, so one more attempt to reduce their size will be made.

Interval-Level Method

Suppose we assume that the control flow analysis pass provides the Cocke-Allen interval structure of the program (see [4]), in the form of an expanded set *blocks* which now contains not only basic blocks but intervals as well. The function *interval(b)* denotes the interval immediately containing b.

The interval structure allows us to divide use-definition chains into two classes:

1. Those chains which are entirely contained within an interval, and
2. Those chains which pass from a definition in an outer interval to a use within an inner interval.

This division is the basis for a partition of the use-definition information. For each element of the set blocks, two functions are defined.

1. *reaches(b)* is the set of all instructions within *interval(b)* which reach the entry of b. Notice that we have substantially reduced the size of the reaches sets by restricting them to one interval.
2. *path(b)* is the set of all variables for which there is a path from the entry of *interval(b)* to b which contains no redefinition of the variable. In other words, the variables in *path(b)* are those which may be defined outside the interval.

The marking algorithm which uses this information is essentially the same as the block-level algorithm with one modification: if the element selected from *livecomps* in step 2 is a pair *<b,value>* and *value* is an element of *path(b)*, we must consider definitions which occur outside the interval so we add the pair *<interval(b),value>* to *livecomps*. Thus, the definitions for that value in the containing interval will eventually be added - in a cascade-like effect. The SETL code now looks like this:

```
define deadcomp(program,blocks);
   /* targ, op, args, do, computedby, reaches, path,
      and interval  are external quantities*/
   /* set do-flags false and initialize livecomps */
   livecomps = nℓ;
   (∀n ∈ program) do(n) = f;
      if op(n) ∈ {bsr,bfn,brc}
         then livecomps = livecomps with n;;
      /* we always perform branches and halts */
      if op(n) ∈ {br,hlt} then do(n) = t;;
   end ∀n;
   /* now pass through livecomps applying steps 2 through 4
      modified for intervals */
   (while livecomps ne nℓ)
      x from livecomps;
      if pair x then
         /* get block and value */
         <b,val> = x;
         /* add instructions which reach b */
         livecomps = livecomps+{n∈reaches(b)|targ(n) eq val
                                          and do(n) eq f};
         /* add a new pair to livecomps if the value can reach b
            from outside the interval */
         if val ∈ path(b) then livecomps=livecomps with
                                              <interval(b),val>;
         end if val;
      else /* an instruction */
         do(x) = t; arglist = args(x);
         complist = computedby(x);
         /* pass through arguments */
         (1 ≤ ∀i ≤ #arglist)
            val = arglist(i);  /* the value */
            new = complist(i); /* the computing inst */
            if new ∈ blocks then /* add a pair */
               livecomps = livecomps with <new,val>;
            else if do(new) eq f then
               /* add an instruction */
               livecomps = livecomps with new;
            end if new;
         end ∀i;
      end if pair x;
   end while livecomps;
   return;  /* all flags set */
end deadcomp;
```

The algorithm which computes the required sets, *reaches* and *path*, is described in SETL Newsletter #112 [2]. This completes our description of the marking algorithm. Dead computations (marked f) may be actually eliminated during code generation.

References

1. Kennedy, K. and Owens, P., "An Algorithm for Use-Definition Chaining," SETL Newsletter #37, Courant Inst. Math. Sci., New York, July 1971.

2. Kennedy, K., "An Algorithm to Compute Compacted Use-Definition Chains," SETL Newsletter #112, Courant Inst. Math. Sci., New York, August 1973.

3. Kennedy, K., "Reduction in Strength Using Hashed Temporaries," SETL Newsletter #102, Courant Inst. Math. Sci., New York, March 1973.

4. Schwartz, J., Abstract Algorithms and a Set-Theoretic Language for their Expression, Courant Inst. Math. Sci., New York 1971.