SETL Newsletter No. 123 K. Kennedy, Rice University

Variable Subsumption with Constant Folding February 1, 1974

One of the difficulties of a thorough strength reduction algorithm which works on an intermediate language of quadruples [1] is that it leaves many assignments of the form

$$X = Y$$

inside loops. A good register allocation algorithm might take care of this problem but many of these assignments can be eliminated during the machine-independent phase. Suppose that after the above assignment the only use of X comes before Y is assigned. In that case we could eliminate the assignment by using Y instead of X -- the assignment to X would then be a dead computation and could be eliminated by a dead computation elimination algorithm [2]. This type of optimization is known as "variable subsumption" and has been treated by a number of authors [3,4]. A global solution is presented here as part of a series of reduction in strength algorithms begun in SETL Newsletter No. 102, "Reduction in Strength Using Hashed Temporaries" [1]. The intermediate language, which consists of simple quadruples, will not be described here; the reader should see [1] for a complete introduction.

Basic Considerations

To eliminate an assignment of the form X = Y we must be able to replace all uses of X which can be reached from this assignment by uses of Y. The algorithm we develop here will attempt to do this as often as possible although its success will be judged by the success of the dead computation elimination algorithm invoked later.

In the assignment X = Y we call X the *primary* and Y the *alternate*. At any point in our processing we will have a number of primaries and corresponding alternates (each primary will have only one alternate) which we will maintain as a *subsumption list* of ordered pairs

<primary,alternate> .

We say that this pair is *covalued* at that point because the two variables will always have the same value there.

Suppose we are processing straight-line code, say within a basic block. We move forward through the code of the block performing the following steps at each instruction.

1. If the instruction is an assignment x = y, all pairs with x as an element must be removed from the subsumption list and the new pair <x,y> must be added.
2. Whenever an operand of the instruction is a primary in the subsumption list, that operand is replaced by its alternate.
3. If z is the target (the variable assigned) of the instruction, all pairs with z as an element must be removed from the subsumption list because these pairs are no longer covalued. (The exception is that of the assignment, covered in case 1.)

An interesting point about this method is its resemblance to constant folding. If we wish to perform constant folding in basic blocks, we keep a list of <variable,constant value> pairs and perform the following steps at each instruction.

1. If the instruction is an assignment x = constant, we remove any pair with x as its first element and insert the pair <x,constant> in the folding list.
2. Replace all operands of the instruction by constants where applicable.
3. If z is the target of the instruction remove any pair with z as its first element from the folding list.
4. If the instruction is of the form z = x op y and x and y are both constants, perform op on x and y (at compile time) to produce constant c, replace the instruction by z = c, and apply case 1 (above).

The similarity of these methods hints that we might be able to do constant folding by generalizing a subsumption algorithm slightly. The algorithm developed in this newsletter will do a complete job of global variable subsumption and a somewhat incomplete job of constant folding.

Basic Block Algorithm

The fundamental tool in our system will be an algorithm which, given an input subsumption-folding list *subinput*, performs subsumption and folding in the basic block and computes two output sets.

1. *subout* - the output subsumption list, and
2. *killedout*- the set of all variables to which an assignment is made within the block.

This second set is important for the global analysis, discussed later. The routine is written along lines suggested in the previous section.

```
definef subfold(block,subinput);
/* a number of quantities are global:
   contents is a function which produces the instructions in a block
   next is a sequencing function for the block
   op, args, and targ produce parts of an instruction
   common is the set of common variables
   constants is the set of atoms which are constant values
   val maps a constant onto its value
   instruction mnemonics are also global */
   subout = inputsub;
   killedout = nl;
/* find the first instruction */
   inst = if ∃b ∈ contents(block) |
              b n ∈ next[contents(block)]
              then b else Ω;
/* loop through the block */
   (while inst ∈ contents(block) doing inst= next(inst);)
/* first check for a call */
   if op(inst) ∈ {bfn,bsr} then
      arg = args(inst);
/*    assume all arguments killed */
```

```
      newkill = {arg(i), 1 ≤ i ≤ #arg}
                 + common /* all common vars */
                 + if op eq bfn
                        then {targ(inst)} else nℓ;
      killedout = killedout + newkill;
/*       remove appropriate pairs */
      (∀p ∈ subout | hd p ∈ newkill or tℓ p ∈ newkill)
      subout = subout less p;  end ∀p;
   else /* a normal instruction */
      const = t;  arg = args(inst);
      (1 ≤ ∀i ≤ #arg) x = arg(i)
/*       check for possible replacements */
         if subout(x) ne Ω then arg(i) = subout(x);;
         if arg(i) n ∈ constants then const = f;;
   end ∀i;
/*    now compute values for constant operations */
   if const then /* compute  value  */
      go to {<add,plus>,
             <sub,minus>,
             <mul,mpy>,
             <div,dvd>,
             <exp,power>,
             <xld,noth>,
             <sto,noth>,
             <neg,chs>,
             <xst,noth>,
             <br,noth>,
             <brc,noth>,
             <hlt,noth>}(op(inst));
   else go to noth;  end if const;
/*   code for constant computations */
plus: value = val(arg(1)) + val(arg(2));
      go to subst;
minus: value = val(arg(1)) - val(arg(2));
      go to subst;
```

```
mpy:   value = val(arg(1)) * val(arg(2));
       go to subst;
dvd:   value = val(arg(1))/val(arg(2));
       go to subst;
power: value = val(arg(1)) exp val(arg(2));
       go to subst;
chs:   value = - val(arg(1));
/*     insert value in constant table and change instruction */
       subst: if ∃x ∈ constants|val(x) = value
                 then c = x;
                 else c = newat;
                    constants = constants with c;
              end if;
              op(inst) = sto;
              args(inst) = <c>;
/*     remove killed pairs */
       noth: (∀p ∈ subout|targ(inst) ∈ {hd p, tℓ p})
                subout = subout less p;  end ∀p;
/*     add new pair if appropriate */
       if op(inst) eq sto then
                subout(targ(inst)) = arg(1);;
/*     update killedout */
       killedout = killedout with targ(inst);
   end if op(inst);
   return <subout,killedout>;
end subfold;
```

Although this routine is long, it is nevertheless straightforward.

Global Considerations

Suppose that a basic block b has a number of predecessors, blocks from which transfers to b can be made. Then the input subsumption list *subin(b)* is just the intersection of the output subsumption lists from these predecessor blocks. In order to do global variable subsumption, we must have correct input subsumption lists for every block in the program. This can be achieved in two passes. The first pass gets output subsumption lists for each of the blocks, then output subsumption lists for intervals, then for higher order intervals and so on. When all output subsumption lists have been computed, we can apply the "intersection" principle on an outer-to-inner basis until we have correct subsumption lists at the entry of each block. We then invoke *subfold* once again to perform the final subsumption (and folding).

As simple as this seems, there remain some tricky problems to be solved. First, we wish to invoke *subfold* only twice for each block in the program -- once on the first pass and once on the second pass. If we are to be able to do this, we must be able to exactly determine what the subsumption list (on the second pass) is for input to the head of an interval if we know what it is on input to the interval. There are two cases to consider.

1. A subsumption pair is active on entry to the interval head if it is active on entry to the interval and if it is not killed on any path in the interval which leads back to the head.
2. A subsumption pair is active on entry to the head if it is active on entry to the interval and it would be a member of the output subsumption list of every block which branches back to the head, even if no substitutions were active on entry to the interval.

The information required to determine these two conditions must be computed on the first pass. In particular, we need to compute *killedin(head)* -- the set of variables killed on some path through the interval leading back to the head, which will be used to determine condition 1 -- and *subaround(interval)* -- the set of subsumption pairs which are in all the output subsumption lists of blocks that branch back to the head, which will be used to determine condition 2. If *inputsub* is the subsumption list on interval entry, then *subin(head)*, the subsumption list on entry to the head, is given by the SETL code fragment

```
(1)   subin(head) = /* condition 1 */
         {p ∈ inputsub|hd p n ∈ killedin(head)
                   and tℓ p n ∈ killedin(head)}
         + /* condition 2 */
              inputsub * subaround(interval);
```

The second problem arises when we attempt to compute *subout* and *killedout* for intervals. If we are to compute *subaround* (to solve the first problem above) we must assume that on entry to every interval the set *inputsub* is nℓ; however, we must also know what the output subsumption list is for a given input subsumption list -- a seeming contradiction. Fortunately, this second output subsumption list may be computed from the first by using a method similar to the solution of problem 1. Given the input list *subin(b)* for an interval (or block) *b*, a pair will be in the general output subsumption list if

1. if is in *subout(b)*, i.e. it is produced in *b* assuming the null input list to *b*, or
2. it is in *subin(b)* and neither of its elements is killed in *b*, i.e. it is in the set

```
   {p ∈ subin(b)|hd p n ∉ killedout(b)
                and tℓ p n ∈ killedout(b)} .
```

These observations will be the basis for a general subsumption-list jump function to be discussed in the next section.

Pass 1

We are now ready to present an algorithm which passes through an interval, computes *subout* (assuming the null input list) and *killedout* for each entry-exit pair of the interval, and computes the sets *subaround* and *killedin(head)* needed in pass 2. If the interval to which this algorithm is applied is in fact a basic block, the algorithm will call *subfold* and then convert the output to entry-exit pair form.

Two important intermediate variables are maintained.

a. *subin(b)*, for each block *b* in the interval, is the input substitution list for that block assuming the null substitution list on interval entry.

b. *killedin(b)*, for each block *b*, is the set of variables which are killed on some path leading from interval entry to b. This information will be saved for use by pass 2 since it never changes.

From the discussion in the last section, we can define functions which compute *subin* and *killedin* for a given block *b*. First to compute *subin(b)* we will need to look at each predecessor *pb* of *b*, take the union of *subout(pb)* and

```
{p ∈ subin(pb)|hd p n ∈ killedout(pb)
              and tl n ∈ killedout(pb)}
```

and intersect these for all such predecessors. The argument *cont* restricts the set of blocks that we will consider.

```
definef jumpsub(b,cont);
/*  pred, subout, killedout, and subin are global */
   return ([*: pb∈pred(b)|pb∈cont]
           (subout(pb) +
             {p∈subin(pb)|hd p n∈killedout(pb)
                and tl p n ∈ killedout(pb)}));
end jumpsub;
```

A similar function can be coded to compute the set *killedin(b)*. Here the consideration is simpler -- a variable is killed along a path from interval entry to *b* if, for some predecessor *pb*, it is either killed on a path to *pb* or killed within *pb*.

```
definef jumpkill(b,cont);
   /* killedin, killedout, pred are global */
   return ([+: pb ∈ pred(b) | pb ∈ cont]
               (killedin(pb) + killedout(pb)));
end jumpkill;
```

Using these two functions, we can now code the routine *subpass1* which computes the desired quantities for an interval. Note that the killedin sets must be modified to take looping paths into consideration.

```
definef subpass1(interval,inputsub)
   /* contents, order, blocks, killedin, subin, subaround,
      pred, succ  are global */
   cont = contents( interval);
   /* isinterval really a block? */
   if cont * blocks eq nl then
      /* call subfold and use the input subsumption list */
      <x,y> = subfold(interval,inputsub);
      /* convert to entry-exit pair form */
      (∀sb ∈ succ(interval))
         subout(interval,sb) = x;
         killedout(interval,sb) = y;  end ∀sb;
   /* return the pair */
   return<subout{interval}, killedout{interval}>;
   else /* we have an interval */
      head = order(interval,1)
       subin(head) = nl;
       killedin(head) = nl;
       <subout{head}, killedout{head}>=subpass1(head,subin(head));
```

```
    /* now pass through the interval in interval order */
    (2 ≤ ∀i ≤ #cont) b = order(interval,i);
       /* apply jump functions */
       subin(b) =jumpsub(b,cont);
       killedin(b)= jumpkill(b,cont);
       /* call subpass1 recursively to get subout, killedout for b*/
       <subout{b},killedout{b}> = subpass1(b, subin(b));
    end ∀i;
    /* recompute killedin for head */
    killedin(head) = jumpkill(head,cont);
    /* recompute killedin for every block */
    (∀b ∈ cont - {head})
       killedin(b) = killedin(b) + killedin(head);
    end ∀b;
    /* compute subaround */
    subaround(interval) = jumpsub(head,cont);
    /* now compute the output sets, killedout and subout for the
       interval */
    (∀sint ∈ succ(interval))
       hsint = order(sint,1);
       killedout(interval,sint) = /* apply jump functions*/
          jumpkill(hsint,cont);
       subout(interval,sint) = jumpsub(hsint, cont);
    end ∀sint;
    return <subout{interval}, killedout{interval}>;
  end if cont ≠ blocks;
end subpass 1;
```

When this routine is applied to the interval representing the entire program it will compute the *killedin* sets for every block and interval in the program and will compute the *subaround* sets for every interval in the program, preparing the way for the second pass. Note that *killedout* and *subout* are not global.

Pass 2

The second pass is similar to the first except that correct subsumption lists are passed into intervals and distributed to the various contained blocks. The input subsumption list *subinput* to an interval must be modified along the lines suggested in the "Global Considerations" section before being passed to the head of the interval. The following function, essentially a transcribed version of code fragment (1) from that section, performs the task.

```
definef convertsub(interval,inputsub);
   /* killedin, subaround, order are global */
   killedinhead = killedin(order(interval,1));
   return( /* condition 1 */
     {p ∈ inputsub|hd p n ∈ killedinhead
           and tℓ p n ∈ killedinhead}
      + /* condition 2 */
          (inputsub * subaround(interval)));
end convertsub;
```

The other main differences from pass 1 are
a. *killedin* sets are not recomputed;   and
b. therefore nothing is done about branches back to the head.
Here then is the SETL code.

```
definef subpass2(interval,inputsub);
/* contents, order, blocks, killedin, subaround, pred, succ
   are all global */
   cont = contents(interval);
/*    is interval really a block? */
   if cont * blocks eq nℓ then  /* call subfold */
      <x,y> = subfold(interval,inputsub);
/*    convert to entry-exit pair form */
      (∀sb ∈ succ(interval))
         subout(interval,sb) = x;
         killedout(interval,sb) = y;  end ∀sb;
```

```
      /* return the resulting pair */
      return <subout{interval}, killedout{interval}>;
   else /* we have an interval */
      head = order(interval,1);
      subin(head) = convertsub(interval,inputsub);
      <subout{head}, killedout{head}>=subpass2(head,subin(head));
      /* now pass through in interval order */
      (2 ≤ ∀i ≤ #cont) b = order(interval,i);
         subin(b) = jumpsub(b,cont);
         /* call subpass 2 recursively */
         <subout{b}, killedout{b}> =subpass2(b,subin(b));  end ∀i;
      /* note that killedin is not recomputed and no
         loop considerations are needed */
      /* now compute output quantities */
      (∀sint ∈ succ(interval))
         hsint = order(sint,1);
         /* use jump functions */
         subout(interval,sint) = jumpsub(hsint,cont);
         killedout(interval,sint) = jumpkill(hsint,cont);
      end ∀sint;
      return<subout{interval},killedout{interval}>;
   end if cont * blocks;
end subfold2;
```

Suppose *progint* is the high-order interval which represents the whole program. The entire variable subsumption process can be invoked by two calls

```
dummy = subpass1(progint,nℓ);
dummy = subpass2(progint,nℓ);
```

Discussion

The storage required by this method is not large. Between passes, we must save *killedin* for every block and interval in the program and *subaround* for every interval in the program. The recursive routines can be converted to an iterative form if intervals are processed in the correct order. (otherwise storage requirements go up because more versions of *killedout* and *subout* must be kept on hand simultaneously).

This method does some constant folding but by no means all of it. This is because each folding pass may create new constant assignments which must then be distributed globally. However, this may be all the folding we need because it gets two important cases. First, the algorithm should do a reasonable job of local constant folding, since two passes are made through each first-level interval and the results of the first pass are folded in on the second pass. Second, a constant assigned in any block will always be folded into the blocks it predominates. Thus, constant parameter initializations will usually be taken care of.

Since a good code motion algorithm will remove from loops most expressions which might be eliminated by constant folding a complete algorithm based on data flow analysis might not be needed or even desirable, given the time and space requirements of such an algorithm.

Acknowledgment. This research was supported by the National Science Foundation, Grant GJ-40585.

References.

1. Kennedy, K., "Reduction in Strength Using Hashed Temporaries," SETL Newsletter #102, Courant Inst. Math. Sci., New York, 3/73.
2. Kennedy, K., "Global Dead Computation Elimination," SETL Newsletter #111, Courant Inst. Math. Sci., New York, August 1973.
3. Allen, F. E., and Cocke, J. "A Catalogue of Optimizing Transformations," Design & Optimization of Compilers, Prentice-Hall, '72.
4. Lowry, E. S., and Medlock, C. W., "Object Code Optimization," Comm. ACM 12, 1 (January 1969) 13-22.