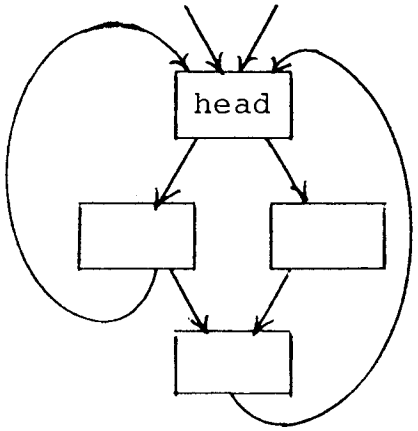
SETL Newsletter No. 112 August 14, 1973

An Algorithm to Compute Compacted Use-Definition Chains

K. Kennedy

This newsletter describes a modification of the use-definition chaining algorithm presented in SETL Newsletter No. 37 [1]. The purpose of this modification is to achieve storage economy by a somewhat compacted form of these chains. We shall assume that the reader has a familiarity with the Cocke-Allen *interval* method of control flow analysis. Expositions of this method are contained in [2,3,4,5]. Basically, an interval is a set of blocks in the program control flow graph such that a particular block, called the *head*, is the only one in the interval with predecessors outside the interval and all loops within the interval pass through the head. Thus an interval might be thought of as a generalized looping structure. A typical interval is diagrammed below.



The interval forms a programming locality which may contain variable names not defined or used elsewhere in the program. The compacted use-definition chaining algorithm attempts to take advantage of this hypothesis by restricting use-definition chains to these localities. Putting it another way, a variable used inside an interval may have been defined in one of two ways:

1. It may have been defined by an instruction within the same interval, in which case the chain from use to definition is local to the interval; or
2. It may have been defined in some larger containing interval, in which case the chain is a global one.

This breakdown allows us to distinguish between two types of use-definition chains -- local and global -- which we treat quite differently. To handle local use-definition chains, we compute, for each block b in the interval, the set *reaches(b)* of all definitions (defining instructions) within the interval containing b which produce a value which is still available on entry to b. In other words, if the value computed by a definition can "reach" b that definition will be in *reaches(b)*.

To handle global chains, we break them up into parts based on the following observation: since an interval **I is** a single-entry region, a definition which reaches a use within I must also reach the entry of I. Thus a global definition which sets variable x reaches a use of x within I if

1. that definition reaches the entry of interval I, and
2. there is a path from the entry of I to the use which contains no redefinition of x. (Such a path is said to be *definition-clear* for x or *x-clear*.)

If *reaches* is defined for intervals as well as blocks, we can use *reaches(I)* to decide the first condition. For the second, we can use the set *path(b)* which contains all variables x for which there is a definition-clear path to b from the entry of the interval containing b. The following algorithm uses these two sets to determine the set of all definitions in the program which reach a given block b.

1. Begin with the set *reaches(b)* of all definitions within I (the immediately containing interval) which reach b.
2. For all variables x in *path(b)*, add to the above set, all definitions which reach I and compute x.

The set of definitions which reach I is computed by applying the same algorithm recursively to I. Let *var(d)* denote the variable which is defined by instruction d, and let *interval(b)* be the interval immediately containing b. The following function computes the set of all definitions which can reach its argument block b.

```
definef defswhichreach(b);
     /* reaches, path, interval, and var are global */
     return (reaches(b) + {d ∈ defswhichreach(interval(b)) |
                           var(d) ∈ path(b)});
end defswhichreach;
```

A serious application of this information to global dead computation elimination is contained in another newsletter [6]. The remainder of this newsletter is devoted to the computation of the sets *reaches* and *path*.

The control flow analysis of the program will provide us with the following functions, defined for each block (and each interval) in the program.

1. *pred(b)* the set of all immediate predecessors of b
2. *succ(b)* the set of all immediate successors of b
3. *interval(b)* the interval immediately containing b.

Note here that blocks and intervals are represented by blank atoms which have certain functions defined on them.

1. *contents(b)* is the set of instructions in b if b is a block, and the set of nodes in b if b is an interval.
2. *order(b)*, where b is an interval, is a mapping from the integers {1,2,3,...} to the blocks in *contents(b)* which orders these blocks

in *interval order* (see [5]). The special element order(b,1) denotes the head of b.

Finally, an initial scan of the program text will provide us with three important quantities.

1. *allvars* is the set of all variables in the program
2. *thru(b,sb)* defined for each block b and sb ∈ succ(b), is the set of all variables for which there is a definition-clear path through b to sb.
3. *def(b,sb)* is the set of all definitions contained in b from which there is a definition-clear path for the variable defined to an exit from b leading to sb.

This completes the list of data structures we will need.

Suppose we are considering an interval *intv* which contains block b. There is an x-clear path from interval entry to b if there is such a path to some predecessor pb of b and through that predecessor to b. As a SETL code fragment, this might be written

```
(1)  path(b) = [+: pb ∈ pred(b)](path(pb) * thru(pb,b));
```

For the special case where b is the head of the interval

```
(1') path(head) = /* all variables */  allvars;
```

since the entry to the head of an interval is identical to the entry of the interval.

In order to apply equation (1), we must be sure that whenever we apply it to a block b, we have already applied it to all predecessors of the block. This condition is assured if we process the blocks in interval order [5]. Thus the *path* sets can be computed in a single pass, using equation (1') for the head and equation (1) for the rest of the blocks (processed in interval order).

This simple one-pass method will not work for *reaches(b)*, because of the presence of *latches* -- branches from within the interval to the head. We must divide all definitions within *intv*, which reach a block b, into two classes.

1. Those which reach b along a definition-clear path which does not contain a latch, and
2. Those which reach b along a definition-clear path which does contain a latch.

The definitions in class 1 can be computed by a method similar to the one used for *path(b)*: a definition reaches b if it is in a predecessor of b or if it reaches a predecessor and passes through.

```
(2)  reaches(b) = [+: pb ∈ pred(b)](def(pb,b)
                    + {d ∈ reaches(b) | var(d) ∈ thru(pb,b)});
```

The initializing definition is obvious.

```
(2')          reaches(head) = nl;
```

However, after the first pass, we must take the class 2 definitions into account. Let *latchdef* be the set of definitions which reach the head via a latch, computed in the natural way:

```
(3)  latchdef = [+: pb ∈ (pred(head) * contents(intv)]
       (def(pb,head)+{d∈reaches(pb) | var(d)∈thru(pb,head)});
```

Now any definition of variable x which reaches the head can reach b if there is an x-clear path from the head to b.

```
(4)  reaches(b) = reaches(b) + {d ∈ latchdef | var(d) ∈ path(b)};
```

A second pass (applying equation (4)) will therefore be required to compute the desired sets.

Once we have processed an interval in this manner we may wish to process the interval of which it is a part. To do this we need the sets *thru* and *def* for intervals. These can be computed during the same processing using some simple considerations. Suppose *sintv* is an immediate successor of *intv*

and *shead* is the head of *sintv*. Then an x-clear path through *intv* to *sintv* must lead through *intv* to b, an immediate predecessor of *shead*, and through b.

```
(5)  thru(intv,sintv) = [+: b∈contents(intv)|b∈pred(shead)]
                          (path(b) * thru(b,shead));
```

Similarly, a definition reaches *sintv* from within *intv* if the definition is in some predecessor b of *shead* or if it reaches b and passes through.

```
(6)  def(intv,sintv) = [+: b∈contents(intv)|b∈pred(shead)]
       (def(b,shead)+{d∈reaches(b)|var(d)∈thru(b,shead)});
```

We are now ready to present the algorithm *chains*, which computes the sets *reaches* and *path* for every interval. Its only argument is the sequence *intervals* which contains all the intervals in the program, beginning with the lowest-level intervals, followed by the next lowest level, and so on. (This order insures that we will process all intervals of one level before proceeding to a higher level.)

```
define chains(intervals);
   /* this routine uses the global data items: pred, succ,
      contents, order, thru, def, allvars, and var, which
      were described earlier. its results, reaches and path, are
      are also global */
   (1 ≤ ∀j ≤ #intervals) intv = intervals(j);
   /* initialize path, reaches, thru, and def */
   head = order(intv,1);
   nodes = contents(intv);
   path(head) = allvars;
   reaches(head) = nl;
   (∀sintv ∈ succ(intv)) shead = order(sintv,1);
      thru(intv,sintv)= if shead ∈ succ(head)
         then thru(head,shead) else nl;
      def(intv,sintv) = nl;
   end ∀sintv;
```

```
      /* first pass. compute thru, path, and the initial reaches */
      (2 ≤ ∀i ≤ #nodes) b = order(intv,i);
         path(b) = [+: pb ∈ pred(b)](path(pb)*thru(pb,b));
         reaches(b) = [+: pb∈pred(b)](def(pb,b)
               +{d∈reaches(pb)|var(d)∈thru(pb,b)});
         /* compute thru */
         (∀sintv ∈ succ(intv)|b∈pred(order(sintv,1) is shead))
            thru(intv,sintv) = thru(intv,sintv) +
               (path(b) * thru(b,shead));
         end ∀sintv;
      end ∀i;  /* end of first pass */
      /* now compute latchdef */
      latchdef = [+: pb∈(pred(head) * nodes)]
         (def(pb,head)+{d∈reaches(pb)|var(d)∈thru(pb,head)});
      /* second pass -- compute def and the final version of reaches*/
      (∀b ∈ nodes)
         reaches(b) = reaches(b)+{d∈latchdef|var(d)∈path(b)};
         (∀sintv∈succ(intv)|b∈pred(order(sintv,1) is shead))
         def(intv,sintv) = def(intv,sintv) +
            def(b,shead)+{d∈reaches(b)|var(d)∈thru(b,shead)};
         end ∀sintv;
      end ∀b;  /* end of second pass */
      /* all quantities have now been computed */
   end ∀j;  /* all intervals have been processed */
   return;
end chains;
```

Acknowledgement. This form of compaction was suggested by J. Schwartz in relation to constant propagation.

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. Schwartz, J., Abstract Algorithms and a Set-Theoretic Language for Their Expression, Courant Inst. Math. Sci., New York 1971.
3. Cocke, J., and Schwartz, J., Programming Languages and Their Compilers, Chapter 6, Preliminary Notes, Sec. Rev. Vers., Courant Inst. Math. Sci., New York, 1970.
4. Allan, F. E., "Control Flow Analysis," Sigplan Proceedings, Vol. 5, No. 7 (July 1970), pp. 1-19.
5. Kennedy, K., "A Global Flow Analysis Algorithm," International Journal of Computer Mathematics, Vol. III, No. 1, Gordon and Breach, pp. 5-15.
6. Kennedy, K., "Global Dead Computation Elimination," SETL Newsletter No. 111, Courant Inst. Math. Sci., New York, August 1973.