SETL Newsletter Number 38 January 13, 1972

Algorithm for Live-Dead Analysis Including Node-Splitting

K. Kennedy

A path in the control flow graph of a program is said to be definition-clear (or def-clear) with respect to a variable A if there is no definition of A between the beginning and end of the path. Here, a definition of A is an operation which sets A to a new value. A variable A is said to be live at a point p in the control flow graph if there is a definition-clear path for A from p to a use of A. If there is no such path, A is said to be dead at p.

To perform certain optimizations it is useful to know, for each point in the control flow graph, which variables are live and which variables are dead. The algorithm described in this newsletter will provide this information for each block in the program in the form of a set of variables, live(block), which are live on entry to that block. To compute this set we will need some auxilliary information about each block. In particular we will need two sets.

1. For each block, we will need the set of all variables which are live on entry to that block by virtue of a definition-clear path from the block entry to a use within the block itself. We shall call this set inside(block).

2. For each block, we will need the sets thru(block,sblock), one for each sblock $\in$ S(block), which are the sets of variables for which there is a definition-clear path from the block entry through the block to sblock. (Recall that s(b) is the set of all basic blocks which are immediate successors of block b and that s(b) is the set of all immediate predecessors of b.)

There is an important relation among these three sets (live,thru,inside), which can be explained as follows: a variable will be live on entry to a block if it is live by virtue of a definition-clear path to a use within the block or if there is a definition-clear path through the block to a successor block

at which it is live on entry. This relation can be expressed by the following set relation:

$$(1)\quad live(block) = inside(block) \cup \left(\bigcup_{sb \in s(block)} (thru(block, sb) \cap live(sb)) \right)$$

or as a SETL "code fragment":

```
(1')  live(block) = inside(block) u [u: sb ∈ s(block)]
                    (thru(block,sb) int live(sb));
```

It is easy to see that, using this relation, we will be able to find out which variables are live on entry to a block if we know which variables are live on entry to its successors. This idea is the basis for our algorithm.

The algorithm will proceed in two passes over the nodes of the control flow graph and all its derived graphs.

1. The first pass will compute the thru and inside sets for intervals of the derived graphs.

2. The second pass will compute the live set, first for the single node of the last derived graph, then for each node in the underlying interval. It will continue in this manner until the live sets have been computed for each node in the control flow graph.

The "thru" and "inside" information for basic blocks can be derived by examining these blocks. The difficult part of the first pass is computing these sets for an interval, given the sets for each node in that interval. This can be done as follows. Suppose we have two auxilliary sets, path(block) and insidesofar. The set path(block) contains all variables for which there is a definition-clear path from the interval entry to the entry to block. The set insidesofar is an accumulator set. As we process the blocks of the interval in interval order,

when we process the block b, we add to <u>insidesofar</u> all variables for which there is a def-clear path from interval entry to b and which are in the set inside(b). These are the variables which will be in the set <u>inside(interval)</u>.

(2) insidesofar = insidesofar $\cup$ (path(b) $\cap$ inside(b))
This equation is true for all blocks in the interval. There is a def-clear path for a variable from interval entry to a block b if there is such a path from interval entry to some predecessor of b and if there is a def-clear path through the predecessor to b.

(3) $$\text{path}(b) = \bigcup_{pb \in p(b)} (\text{path}(pb) \cap \text{thru}(pb,b))$$
This relation does not hold for the head of the interval. Since the head entry is identical to the interval entry,

(4) path(head) = all variables
Suppose J is an interval which is a successor of I and that j_1 is its head. The node j_1 must be a successor of at least one block in I. We can, therefore, compute thru(I,J) as follows:

(5) $$\text{thru}(I,J) = \bigcup_{b \in p(j_1) \cap I} (\text{path}(b) \cap \text{thru}(b,j_1))$$
As previously indicated,

(6) inside(I) = insidesofar
where <u>insidesofar</u> is the version left after all nodes in the interval have been processed.

The algorithm, then, passes through the nodes of the interval I in interval order, computing the sets, <u>path</u> and <u>insidesofar</u>. Because it uses interval order, the predecessors of a node are always processed before the node itself and the path sets required by equations (2) and (3) are available when needed.

Note that we need not worry about the contribution of loops within the interval, because a loop cannot contribute any new paths. This is because path(head) is already as large as it can be.

The final step of the algorithm will be to compute the thru and inside sets for the interval, using equations (5) and (6). We can now write a SETL algorithm, process(interval), which will perform the computations specified above. The argument, interval, will be a SETL sequence of nodes with interval(1) = head.

```
define process(interval); external p, s, inside, thru,
     allvars; path = nl, insidesofar = nl;
     path(interval(1)) = allvars;
/* pass through the interval in interval order */
(2 <∀i <#interval)b=interval(i);
     path(b) = [u : pb∈p(b)](path(pb) int thru(pb,b));
     insidesofar = insidesofar u (path(b) int inside(b));
     end ∀i; /  now calculate thru and inside sets I/
     inside(interval) = insidesofar;
     (∀y ∈ s(interval)) preint = p(y(1)) int tl[interval];
     thru(interval,y)=[u : b∈preint](path(b) int thru(b,y(1)));
     end ∀y; return; end process;
```

Now the entire first pass may be described. The process, called pass1, will be presented with a sequence of intervals starting with basic intervals, then intervals of the first derived graph and so on until the last interval, which is the fully reduced control flow graph.

```
define pass1(intervals); (1 <∀i <#intervals)
     process(intervals(i)); end ∀i; return; end pass1;
```

This pass merely consists of calling process for each interval in all the derived graphs. At the end of pass 1, we will have computed the inside and thru sets for each interval in the sequence of derived graphs, including the interval which reduces the program to a single node. Since an exit block has no successors the live set for an exit block is equal to the inside set for that block. In particular, the single node representing the whole program is an exit block, so we now have the set of variables which are live on entry to the program -- this is just the inside set for the node. This set will be useful in two ways. First, it will tell us which variables are improperly initialized in the program and, second, it will be important in computing the live sets for nodes of the underlying interval. The driving routine for the dead variable trace can now be specified.

```
define livevars(intervals); external live, thru,
     inside, s, p; pass1(intervals);
/* set live set for the single node */
     live(intervals( intervals)) = inside(intervals( intervals));
     pass2(intervals); return; end livevars;
```

The driving routine calculates the live set for the program entry and calls pass2. Pass2 is a simple routine which calls the routine liveint for each interval in the list starting with the last and going forward. This order will insure that we always process outer intervals before we process inner intervals.

```
define pass2(intervals);
     (#intervals > ∀i > 1) liveint(intervals(i)); end ∀i;
     return; end pass2;
```

The routine liveint merely calculates the live sets for every node in the interval, given the live sets for the entry to the

interval, and for the entries to all successors of the interval. It does this by passing through the interval in reverse interval order and calculating the live set for each node as it is encountered. The formula used is (1). In order to use this formula, we must have live sets for each successor of the node we are processing. But we do have these sets. Suppose that we are examining node b_i and a particular successor sb. There are three possibilities.

1. If sb is not in the interval, it must be the head of some successor interval y. Since we have computed the live sets for every successor interval and since the entry to an interval is identical to the entry to its head, we can use live(y) for live(sb).

2. If sb is in the interval but is not the head, we have computed live(sb) because we are processing nodes in reverse interval order, which means that we must have already processed sb.

3. If sb is the head of the interval, we can use live(interval) since the entry to the interval is identical to the entry to its head. Recall that we always have live(interval) before we process the nodes of that interval.

Thus we have the required live sets and we can always apply formula (1). Notice how important the interval order is to this process. The routine liveint is coded in SETL as follows:

```
define liveint(interval); external s, p, thru, inside, live;
/* pass through the interval in reverse order */
     live(interval(1)) = live(interval);
(#interval > ∀i > 2) b = interval(i); live(b) = inside(b);
     (∀ sb ∈ s(b)) live(b) = live(b) u (thru(b,sb) int
          (if ∃ [y] ∈ s(interval) | sb eq y(1) then live(y)
            else live(sb)); end ∀sb; end ∀i;
            return; end liveint;
```

This subroutine is called for each interval in all the derived graphs and the basic control flow graph. On completion of pass2, it will have computed a live set for each block in the control flow graph. The dead variable analysis algorithm will, therefore, pass twice through all the nodes of the control flow graph and its derived graphs - which is reasonable efficiency.

We are now ready to incorporate nodesplitting into the dead variable analysis algorithm and to present the final form of that algorithm. The basis for the nodesplitting considerations discussed here is the treatment of this subject found in the SETL notes. In that treatment, if one of the derived graphs G_j cannot be reduced further, it is transformed into G_j in which several of the nodes of G_j are split into more than one copy. The nodes of the graph G_j are SETL pairs, where the first item of each pair is a node of the original graph G_j and the second item of each pair is either nl, in the case of an unsplit node, or another node of the graph G_j, in the case of a split node. The successor function for this transformed graph is described in [5]. What we need to know to modify the dead variable analysis algorithm is

1. How to derive thru and inside sets for the nodes of G_j given these sets for the nodes of G_j (this information is needed in the first pass), and

2. How to derive the live sets for the nodes of G_j given these sets for the nodes of G_j (this information is needed in the second pass).

To answer these questions we must look at the nature of nodes of G_j. Let b and sb be a pair of nodes in G_j such that sb s(b). Now b represents a copy of hd b (the first element of the pair) and sb represents a copy of hd sb. If there is a definition-clear path to a use of a variable in hd b, there

must be a definition-clear path to a use in any copy of that node. Therefore,

(1) inside(b) = inside(hd b);

Similarly, if there is a definition-clear path for a variable through hd b to the exit to hd sb, then there must be such a path in any copy.

(2) thru(b,sb) = thru(hd b, hd sb);

These represent the transformations which must be made on the first pass. The second-pass transformation is slightly more complicated. If b_1, b_2, ..., b_n are nodes in G_j which are all copies of hd b_1, then if a variable is live on entry to any of the nodes b_1, b_2, ..., b_n it must be live on entry to the node hd b_1 in G_j. Thus,

(3) live(hd b_1) = [u : $1 < i < n$] live(b_i)

We shall make this transformation in the routine liveint. Whenever we calculate live(b) we will execute the following SETL instruction.

(4) if type b eq pair then live(hd b) = live(b) u
if live(hd b) eq Λ then nl else live(hd b)

The transformations (1) and (2) will be made in an initialization block at the beginning of the routine process.

Is this all we need to do? The answer to this question is yes. Consider the routine process. In normal processing each interval processed will consist of a sequence of nodes from the previous derived graph. However, if some nodes of the previous graph were split, the interval will be a sequence of pairs. The sets thru and inside will not be defined for the pairs, but will be defined for the first elements of those pairs. These sets can therefore be defined for the pairs by

transformations (1) and (2). If the interval processed by the routine liveint consists of pairs, the live sets are computed for each pair. But at the next level, live sets will be needed for the nodes represented by these pairs. These live sets are computed by transformation (3).

Here then is the modified SETL algorithm. The reader will note that it is identical to the algorithm presented earlier except for the insertion of the transformations described above.

```
define process(interval); external p, s, inside, thru,
     allvars; path = nl; insidesofar = nl;
     path(interval (1)) = allvars;
/* test for split nodes */
     if type interval(1) eq pair then
     (1 <∀i <#interval) b = interval(i); inside(b) = inside(hd b);
     (∀sb ∈ s(b)) thru(b,sb) = thru(hd b, hd sb); end ∀sb;
     end ∀i; end if; insidesofar = inside(interval(1));
/* pass through the interval in interval order */
     (2 <∀i <#interval) b = interval(i);
       path(b) = [u : pb ∈ p(b)](path(pb) int thru(pb,b));
       insidesofar = insidesofar u (path(b) int inside(b));
       end ∀i;
/* now calculate thru and inside for the interval */
      inside(interval) = insidesofar;
      (∀y ∈ s(interval)) preint = p(y(1)) int tl[interval];
        thru(interval,y) = [u : b∈preint](path(b)int thru(b,y(1)));
        end ∀y; return; endprocess;

define passl(intervals); (1 <∀i <#intervals)
    process(intervals(i)); end ∀i; return; end passl;

define livevars(intervals); external live, thru,
    inside, s, p; passl(intervals);
/* set live set for the single node */
     live(intervals(#intervals)) = inside(intervals(#intervals));
     pass2(intervals); return; end livevars;
```

```
define pass2(intervals); (#intervals > ∀i > 1)
     liveint(intervals(i)); end ∀i; return; end pass2;

define liveint(intervals); external s, p, thru, inside, live;
     /* pass through the interval in reverse order */
     live(interval(1)) = live(interval);
     (#interval > ∀i > 2) b=interval(i); live(b) = inside(b);
        (∀sb ∈ s(b)) live(b) = live(b) u (thru(b,sb) int
         (if ∃ [y] ∈ s(interval) | sb eq y(1) then live(y)
             else live(sb)); end ∀ sb;
     /* test for split nodes */
       if type b eq pair then live(hd b) = live(b) u
         (if live(hd b) eq Ωthen nl else live(hd b));;
     end  i; return; end liveint;
```

This is the complete algorithm with node-splitting included. It is now in a form which is suitable for implementation as part of an optimizing compiler based on interval techniques.