July 16, 1971
K. Kennedy, P. Owens

An Algorithm for Use-Definition Chaining:

Several forms of optimization depend on knowing which definitions in a program can affect the environment at a given point in the control flow graph of a program. By a definition of a variable x, we mean an assignment of a value to x. This newsletter presents an algorithm which computes the set, reaches(b), of all definitions of variables for which there is a definition-clear path to the entry to block b. This algorithm uses the interval technique and might be considered the "dual" of the dead-variable analysis algorithm (SETL Newsletter Number 28).

For the purposes of the analysis, we need the following sets and functions.

1. defs - the set of all definitions in the program.
2. var(defn) - the function which maps a definition onto the variable it defines.
3. thru(b,sb) - the set of variables for which there is a definition - clear path through b to sb.
4. def(b,sb) - the set of definitions in b from which there is a definition clear path for the variable defined to an exit from b to sb.
5. initial - the set of initializing definitions made before entry to the program.

The following functions are assumed to have been provided by the interval analysis.

1. s(b) - the set of immediate successors of b.
2. p(b) - the set of immediate predecessors of b.

These functions are also defined on all intervals of the derived graphs.

The first step in our analysis is to calculate <u>thru</u> and <u>def</u> for an interval, given these sets for the nodes of the interval. To do this, we will first compute three intermediate sets.

1. <u>path(b)</u> - the set of variables for which there is a definition-clear path from interval entry to <u>b</u>.
2. <u>defint(b)</u> - the set of definitions in the interval which can reach <u>b</u> by any path not including a latch (a branch back to the head).
3. <u>defhead</u> - the set of definitions in the interval which can reach the head by a latch.

We now state some equations involving these sets. Suppose sb is the head of some successor interval <u>sint</u> of the interval <u>intv</u> that we are processing. A definition clear path through <u>intv</u> to <u>sint</u> must pass through some predecessor of sb in <u>intv</u> and through that predecessor to sb; the SETL code fragment is

$$(1)\quad \text{thru(intv,sint)} = [\underline{\text{u}}\text{: b}\in\text{p(sint(1))}\underline{\text{int}}\text{ tl [intv]]}\ \text{(path(b) } \underline{\text{int}} \text{ thru(b,sint(1)))}$$

where <u>intv</u> and <u>sint</u> are SETL sequences (in interval order) of nodes.

Computing the set <u>def(intv,sint)</u> for the interval <u>int</u> is more complicated. A definition within <u>intv</u> can reach <u>sint</u> if it is in one of two sets.

1. $[\underline{\text{u}}\text{: b}\in\text{p(s intv(1))]}\ \text{(def(b,sint(1))}\ \underline{\text{u}}\ \{\text{d}\in\text{defint(b)} \mid \text{var(d)}\in\text{thru(b,sint(1))}\}\text{);}$
   - the set of definitions in <u>b</u> with a def-clear path to

sint(1) and definitions in <u>intv</u> which reach the entrance to <u>b</u> and pass through <u>b</u> to sint(1).

2. $\{d \in \mathrm{defhead} \mid \mathrm{var}(d) \in \mathrm{thru}(\mathrm{intv}, \mathrm{sint})\}$
   - the set of definitions which reach the head of <u>intv</u> via a latch and whose variables have def-clear paths through <u>intv</u> to <u>sint</u>.

Therefore,

$$(2)\quad \mathrm{def}(\mathrm{int}, \mathrm{sint}) = \{d \in \mathrm{defhead} \mid \mathrm{var}(d) \in \mathrm{thru}(\mathrm{int}, \mathrm{sint})\}\ \underline{u}[\underline{u}\!:\ b \in p(\mathrm{sint}(1))](\mathrm{def}(b, \mathrm{sint}(1))\ \underline{u}\ \{d \in \mathrm{defint}(b) \mid \mathrm{var}(d) \in \mathrm{thru}(b, \mathrm{sint}(1))\});$$

The equation for <u>path(b)</u> is the same as the one in dead variable analysis.

$$(3)\quad \mathrm{path}(b) = [\underline{u}\!:\ pb \in p(b)](\mathrm{path}(pb)\ \underline{\mathrm{int}}\ \mathrm{thru}(pb, b));$$

where

$$(4)\quad \mathrm{path}(\mathrm{int}(1)) = \{\text{all variables}\} = \mathrm{var}[\mathrm{defs}];$$

The set <u>defint(b)</u> is computed by examining each predecessor of b in the interval. A definition will be in <u>defint(b)</u> if it is in <u>def(pb,b)</u> for some predecessor <u>pb</u> of <u>b</u>, or if it is in <u>defint(pb)</u> and its variable is in <u>thru(pb,b)</u>.

$$(5)\quad \mathrm{defint}(b) = [u\!:\! pb \in p(b)]\ (\mathrm{def}(pb, b)\ \underline{u}\ \{d \in \mathrm{defint}(pb) \mid \mathrm{var}(d) \in \mathrm{thru}(pb, b)\});$$

where

$$(6)\quad \mathrm{defint}(\mathrm{int}(1)) = \underline{\mathrm{nl}};$$

The form of these SETL code fragments suggests that we process the nodes of the interval in interval order. Since we need information concerning the predecessors of each node processed. The routine inout(intervals), has, as its only argument, a sequence of intervals, starting with the intervals of the control flow graph followed by intervals of the first derived graph and so on. Each interval in this sequence is a sequence of its nodes in interval order. For each interval, inout computes the sets, thru and def, assuming that they are available for the nodes of the interval. Their availability is assured by the order of the intervals in the sequence intervals.

```
define inout (intervals); optimizer external s,p,thru,def,var,defs;
/* process each interval */
(1≤∀j≤#intervals) intv=intervals(j);
     defint(intv(1))=nl; path(intv(1))=var[defs];
/* pass through intv to get path and defint */
(2≤∀i≤#intv)  b=intv(i);
     path(b) = [u: pb∈p(b)](path(pb)int thru(pb,b));
     defint(b) = [u: pb∈p(b)](def(pb,b)u
         {d∈defint(pb) | var(d)∈ thru(pb,b)});end ∀i;
/* compute defhead */
     defhead = [u: pb∈ (p(intv(1)) int tl[intv])](def(pb,intv(1))
         u{d∈defint(pb) | var(d)∈ thru(pb,intv(1))});
/* compute thru and def for interval */
(∀sint ∈ s(intv)) sb = sint(1);
     thru(intv,sint) = [u: pb∈(p(sb)int tl[intv]]
          (path(pb) int thru(pb,sb));
     def(intv,sint) ={d∈defhead | var(d)∈thru(intv,sint)}
         u [u: pb∈p(sb)](def(pb,sb) u
               (d∈ defint(pb) | var(d)∈thru(b,sb)))
     end ∀sint; end ∀i; return; end inout;
```

This routine is all we need for the first pass. It will calculate thru and def for intervals of the control flow graph and all derived graphs.

The second pass must calculate reaches(b) for every block in the program. A definition reaches b if, for some predecessor pb, it is in def(pb,b) or if it reaches pb and the variable it defines is in thru(pb,b).

$$(7) \quad \text{reaches}(b) = [\underline{u}:\ pb \in p(b)](\text{def}(pb,b)\ \underline{u}\ \{d \in \text{reaches}(pb) \mid \text{var}(d) \in \text{thru}(pb,b)\});$$

For a program entry e,

$$(8) \quad \text{reaches}(e) = \text{initial};$$

and for interval heads,

$$(9) \quad \text{reaches}(\text{intv}(1)) = \text{reaches}(\text{intv});$$

At the end of the first pass the reaches set for the single node representing the entire program is set to initial. Then the routine outin is called.

```
define usedef(intervals); optimiser external p,s,thru,def,defs
initial, reaches, var;       inout(intervals);
     reaches(interval(#intervals) = initial;
     outin(intervals); return; end usedef;
```

The subroutine outin calculates the reaches set for the nodes of each interval, given this set for the interval itself. It processes the outermost interval first, then the next outermost intervals and so on, passing through intervals in reverse order. Within an interval, nodes are processed in interval order using equation (7).

```
define outin(intervals); optimizer external s,p,thru,def,var,
     reaches;
/* process intervals in reverse order */
(#intervals≥∀j≥1) intv=intervals(j);
     reaches(intv(1))=reaches(intv);
/* process nodes in interval order */
(2≤∀i≤#intv) b=intv(i);
     reaches(b)=[u: pb∈p(b)](def(pb,b) u
       {d∈ reaches(pb) | var(d)∈ thru(pb,b)});
end ∀i; end ∀j; return; end outin;
```

On exit from this routine, <u>reaches(b)</u> will have been computed for every block b in the program, which was the desired result. Note that the form of equation (7) forces us to process in interval order if we wish to always have the required <u>reaches</u> sets.