Induction variable elimination

Algorithm DIV: Detection of Induction Variables

Our objective here is to associate with each induction variable, `j`, a triple (`i`,*c*,*d*) where `i` is a basic induction variable and *c* and *d* are constants such that the value of `j` is *c**`i`+*d*.

We define as *basic induction* variables those scalar variables whose only assignments in loop L have the form `i = i ± c`.

```
begin
    Find the basic induction variables. Associate with each
    basic induction variable i, the triple (i,1,0).

    for each variable k with a single assignment to k within
    L of the form k = j ± b or k = j * b where b is
    a constant and j is an induction variable do
        if j is a basic i.v. then
            associate to k, the triple (j,1,±b) in the first
            case and (j,b,0) in the second case.
        else
            let j be associated with the triple (i,c,d)
            if
            there is no assignment to i from the lone
            point of definition of j to the assignment of
            k and
            no definition of j from outside L reaches k
            then
                associate (i,c,d±b) with k in the first
                case and (i,b*c,b*d) in the second case
            fi
        fi
    od
end
```

Strength Reduction

The purpose of strength reduction is to replace multiplications inside a loop by additions.

```
begin
    for each basic induction variable i do
        for each non-basic i.v. j associated with a triple of
        the form (i,c,d) do
            Create a new variable s (unless another i.v.,
            say m, associated with the same triple was
            processed before. In this case use the same
            variable created for m)
            Replace the assignment to j with j=s
            after each assignment i = i + n append
            s=s+c*n (notice that c*n is a constant).
            Insert s=c*i+d just before the loop.
        od
    od
end
```

Another definition of basic induction variable

The previous discussion does not deal with some important cases. for example, coupled induction variables (i= j +2 ... j = i+1)

An alternative approach introduced by Cocke and Kennedy (CACM Nov. 1977) is to define as induction variable those scalar variables within a strongly connected region (if n_0 and n_m are two blocks in a strongly connected region, then there is a path from n_0 to n_m within the region) that are defined only by simple instructions of the from:

```
i = j ± k
i =± j
```

where `j` and `k` a re either induction variables or region constants

These induction variables can be detected by the following algorithm

Algorithm A^{2}DIV: Another Algorithm for the Detection of Basic Induction Variables

Input:

1. A strongly connected region R of the flow graph
2. The set RC of constants within the region (see loop-invariant detection above)

Output: The set IV of basic induction variables

Method:

```
begin
    IV = ∅
    for each instruction I in R do
        if I is of the form i=j±k or i=±j then
            IV += {i}
        fi
    od
    change = true
    while change do
        change = false
        for each instruction I in R whose lhs is in IV do
            if I is not of the form i=j±k or i=±j or
            any operand ∉ IV ∪ RC then
                remove i from IV
                change = true
            fi
        od
    od
end
```

This is based on the following observations

1. It is simpler to define what is *not* an induction variable than is to define what is
2. if `x= op(x,z)` and op is not one of **store**, **negative**, **add** or **subtract**, then `x` is not an induction variable.
3. if `x = op(y,z)` and `y` and `z` are not both elements of IV ∪ RC, then x is not an induction variable

Another Strength-Reduction Algorithm

Candidates for strength reduction

1. Multiplication by a constant

```
        loop
            n=i*a
            ...
            i=i+b
```

after strength reduction

```
        loop
            n=t1
            ...
            i=i+b
            t1=t1+a*b
```

after loop invariant removal (`b` is a constant because `i` is an induction variable)

```
        c= a * b
        loop
            n=t1
            ...
            i=i+b
            t1=t1+c
```

2. Multiplication by a constant plus a term

```
loop
    n=i*a+c
    ...
    i=i+b
```

after strength reduction

```
loop
    n=t1
    ...
    i=i+b
    t1=t1+a*b
```

Notice that the update to `t1` does not change by the addition of the contant. However, the initialization assignment before the loop should change.

3. Two induction variables multiplied by a constant and added

```
loop
    n=i*a+j*b
    ...
    i=i+c
    ...
    j=j+d
```

after strength reduction

```
loop
    n=t1
    ...
    i=i+c
    t1=t1+a*c
    j=j+d
    t1=t1+b*d
```

4. Multiplication of one induction variable by another

```
loop
    n=i*j
    ...
    i=i+a
    ...
    j=j+b
```

After strength reduction of `i*j`

```
loop
    n=t1
    ...
        --------- t1=i*j
    i=i+a
        --------- new t1 should be (i+a)*j=t1+a*j
    t1=t1+a*j
        ...
    j=j+b
        -------- new t1 should be i*(j+b)=t1+b*i
    t1=t1+b*i
```

After strength reduction of `a*j`

```
loop
    n=t1
    ...
```

```
        i=i+a
        t1=t1+t2

        ...
        j=j+b
        t1=t1+b*i
        t2=t2+a*b
```

`b*i` is handled similarly.

5. Multiplication of an induction variable by itself

```
    loop
        n=i*i
        ...
        i=i+a
```

After strength reduction

```
    loop
        n=t1
        ...
        i=i+a
            -------- new t1 should be (i+a)*(i+a)=t1+2*a*i+a*a
        t1=t1+2*a*i+a*a
```

Now strength reduce `2*a*i+a*a`

```
    loop
        n=t1
        ...
```

```
i=i+a
t₁=t₁+t₂
    -------- new t₂ should be 2*a*(i+a)+a*a=t₂+2*a*a
t₂=t₂+2*a*a
```

```
begin
    PILE={S| rhs of S is i*x with i∈ IV and x ∈ IV∪RC}
        while PILE ≠∅ do
        S from PILE (e=i*x)
        if e is of the form $v_{ix}$ then
            delete S
        else
            create a new variable $v_{ix}$
            replace S with e = $v_{ix}$
        fi
        for each T in DEFS(i,S) ∪ DEFS(x,S) do
            if there is a definition of $v_{ix}$ in T then
                next
            elseif T is outside the loop then
                insert $v_{ix}$=i * x just before the loop
            elseif T is of the form  i = k then
                    replace T with the sequence
                    [R: $v_{ix}$ = k*x,
                        i = k]
                    PILE += {R}
            elseif T is of the form i=k+l and x ≠ i then
                    replace T with the sequence
                    [R1: $v_{kx}$=k*x,
                    R2:  $v_{lx}$=l*x,
                         $v_{ix}$=$v_{kx}$+$v_{lx}$,
                         i=k+l]
                    PILE += {R1,R2}
            elseif  x≡i and T is of the form  i=k+l
                then
```

```
                                        replace T with the sequence
                                        R1:  $v_{kk}$=k*k
                                        R2:  $v_{ll}$= l*l
                                        R3:  $v_{lk}$=l*k
                                        R4:  $v_{2lk}$=2*$v_{lk}$
                                             $v_{2lk+ll}$=$v_{2lk}$+$v_{ll}$
                                             $v_{ii}$= $v_{kk}$+ $v_{2lk+ll}$
                                             i=k+l
                                        PILE+={R1,R2,R3,R4}
                              fi
                         od
                    od
               end
```

Aliasing and Pointers

When assignments of the form `*p=a` or of the form `x=*p` are found in a block the analysis algorithm has to take into account their possible effects.

In the case of reaching definitions, the most naive (conservative) approach is to assume that `*p=a` may define any program variable and does not kill any definitions (i.e. it is not a *must define* of any variable). However such assumptions result in more reaching definitions than is realistic.

In the case of live variable analysis, the most naive (conservative) approach is to assume that `*p=a` does not define any program variable and that `x=*p` may access (use) any variable. However such assumptions result in more live variables than is realistic.

We show next a simple algorithm to improve the accuracy of the analysis in the presence of pointers. The algorithm will compute for each block the set of variables that a pointer may point to at the beginning of the block. This set will contain pairs of the form (p,a) where p is a pointer and a is a variable or array.

With this information, we can compute the set S of possible values of a pointer at a particular statement and use it as follows:

1. For reaching definitions *p=a will generate a definition of every variable b such that (p,b) is in S. Also, *p=a kills definitions of b only if b is not an array and is the only variable that p could point to.
2. For live variable analysis, *p=a uses only a and p. It may also be assumed to define b if b is the only variable that p might point to. a=*p represents a definition of a and a use of p. It should also be assumed to access (use) any variable that p could point to.

Computing the effect of pointer assignments

The algorithm assumes that only additions and subtractions of constant are valid operations on pointers.

Also, if a pointer points to an array, say `a`, adding or subtracting a constant will not change the fact that the pointer points to `a`.

However, adding or subtracting a constant is not valid for the case of scalars.

A function (*trans$_I$(S)*) is defined that defines the effect of an instruction I on the set S of possible variables and arrays where the pointers used in the program may point to.

trans$_I$(S) is computed as follows:

1. if I is `p=&a` or `p=&a+c` where `a` is an array, then the result is
   (S-{(`p,b`)|(`p,b`) is in S}) + {(`p,a`)}
2. If I is `p=q±c` for pointer q and nonzero constant `c`, then the result is
   (S-{(`p,b`)|(`p,b`) is in S}) + {(`p,b`)|(`q,b`) is in S *and* `b` *is an array*)}
3. If I is `p=q`, the the result is
   (S-{(`p,b`)|(`p,b`) is in S}) + {(`p,b`)|(`q,b`) is in S)}
4. If I assigns to pointer `p` any other expression, then the result is
   (S-{(`p,b`)|(`p,b`) is in S})
5. If I does not assign to a pointer, then the result is
   S (i.e. there is no change)

If a block B consists of the instructions I1, I2, ...,In, then $trans_B(S)$ is defined as $trans_{In}(...trans_{I2}(trans_{I1}(S))...)$.

Now, we define $out[B] = trans_B(in[B])$ and

$$in[B] = \bigcup_{P \in PRED[B]} out[P]$$

This can be solved iteratively.

Interprocedural Data Flow Analysis

The objective here is to determine how each procedure influences the sets gen, kill, use, and def and then compute the data flow information for each procedure independently.

Consider

```
        subroutine p(x,y)
        ...
        a=b+x
        ...
        y=c
        ...
        d=b+x
        ...
        end
```

Is `b+x` available at the last statement? It will depend on whether `y=c` kills the expression or not. If `call p(z,z)` is possible, or the following sequence is possible

```
        subroutine q(u,v)
        ...
        call p(u,v)
        ...
        end
        ...
        call q(z,z)
```

The `y=c` would kill the expression `b+x`

Alias Computation

In some situations it is conservative not to regard variables as aliases of one another. For example, for reaching definitions. In other cases the conservative choice is to assume aliasing when in doubt. For example in available expressions.

We will assume that the language has global variables and parameters, and that a global variable can be a parameter. We do not distinguish between occurrences of a variable at different points of the program containing calls to the same procedure. That is, although the same variable may represent totally different values at two different points of the program, it will be considered the same parameter in all cases. Also, the computation is for the whole program. that is, we will assume that if two variable could be aliased at a certain point, we will assume they always could be.

Algorithm AC: Alias Computation.

Input: A collection of procedures and global variables

Output: An equivalence relation with the property that whenever there is a position in the program where x and y are aliases on one another, x R y; the converse need not be true.

Method:

```
begin
    Rename variables v local to each procedure p (including
    formal parameters) as p$v.

    For each call p(y1,y2,..yn) to procedure
    p(x1,x2,...,xn) set xi R yi.

    Take the transitive and reflexive closure of R
end
```

The change[p] set

We now compute change[p], the set of globals or formal parameters that can be changed by calling p.

Let def(p) the set of formal parameters and globals changed within p itself.

```
for each procedure p do change[p]=def(p) od
while changes to any change[p] occur do
    for each procedure q called by p do
            add any global variables in change[q] to
            change[p]
            for each call to q add to change[p] the actual
            parameters whose formal equivalents are in
            change[q].
    od
od
```

The change information, together with the alias information can be sued to do data flow analysis. For example, to compute ekill in the available expression problem, a call to a procedure q can be safely assumed to assign (define) only those variables aliased to a variable in change[q].

Monotone Data Flow Analysis Framework

Definition. A relation R on a set S is
(a) reflexive iff $(\forall\ x \in S)[xRx]$,
(b) antisymmetric iff $[xRy \wedge yRx \rightarrow x=y]$
(c) transitive iff $(\forall\ x,y,z \in S)\ [xRy \wedge yRz \rightarrow xRz]$

Definition. A *partial ordering* on a set *S*, which we denote by the symbol $\leq$, is a reflexive, antisymmetric, and transitive relation on S. The pair $(S,\leq)$ is called a *partially ordered set*.

$\top$

We write $x < y$ iff $x \leq y$ and $x \neq y$.

Definition. Let $(S, \leq)$ be a partially ordered set, and let a and b be elements of S. A join (AKA least upper bound, lub) of a and b (denoted $a \vee b$) is an element $c \in S$ such that [$a \leq c$ and $b \leq c$ and there is no $x \in S$ such that ($a \leq x < c$ and $b \leq x < c$)].

A meet (AKA greatest lower bound, glb) of a and b (denoted $a \wedge b$) is an element $d \in S$ such that [$d \leq a$ and $d \leq b$ and there is no $x \in S$ such that ($d < x \leq a$ and $d < x \leq b$)].

A *minimal* element of a partially ordered set, T, is an element $a \in T$ such that there is no $x \in T$ for which $x < a$. The maximal element is similarly defined.

The bottom (AKA zero, minimum, least) element of S is an element $\perp \in S$ such that $\perp \leq x$ for all $x \in S$. The top (AKA one, maximum, greatest) element of S is an element $\top \in S$ such that $x \leq \top$ for all $x \in S$.

Observation. Both meet and join are idempotent $x \wedge x = x, x \vee x = x$

Definition A lattice is a partially ordered set, any two elements of which have a unique join and meet.

Definition A semilattice is a pair (S,*) where S is a nonempty set and * is a binary operation on S which is idempotent, commutative, and associative. If (S, $\wedge$, $\vee$) is a lattice, then both (S, $\wedge$) and (S, $\vee$) are semilattices.

A data flow analysis framework consists of:

1. A semilattice $(L,\wedge)$. Where L is a nonempty set, and $\wedge$ is a binary operation on L. $\wedge$ represents the confluence operator.

2. A set F of transfer functions $f \in F$, $f:L \rightarrow L$.

The elements of L are the values at the top of the nodes of a flow graph. For example, the **in[B]** sets in the reaching definitions problem would be elements of L. Thus, $L=2^D$ where D is the set of definitions in the program.

The transfer functions define the effect of a node of the flow graph on the elements of L. In the case of *reaching definitions*, the set F is the set of functions of the form $f(X)=A\cup(X-B)$ where $A, B \in L$. A is gen and B is kill. ($\wedge$ is set union).

For *available expressions*, $L=2^E$ where E is the set of expressions computed by the program. F has the same form as the function above, but A and B are now sets of expressions. ($\wedge$ is now set intersection).

For constant propagation each element of L is a function (or table)

$$\psi : V \rightarrow R \cup \{\text{nonconstant, undefined}\}.$$

Where V is the set of variables in the program.

The transfer functions are created from the type of operation in the flowgraph:

(a) if no definitions then f is the identity.
(b)if x=c then $f(\mu)=\nu$ with $\nu(w)=\mu(w)\ \forall\ w \neq x$ and $\nu(x) = c$.
(c)if x=y+z then $f(\mu)=\nu$ with $\nu(w)=\mu(w)\ \forall\ w \neq x\ \nu(x)=\mu(y)+\mu(z)$
[where

nonconstant+a=nonconstant,
undefined + a=undefined,
noncontant+undefined=noncontant]

(d) if read(x) then $f(\mu)=\nu$ with $\nu(w)=\mu(w)\ \forall\ w \neq x$. $\nu(x)$=nonconstant...

The meet operation of two tables m and n is defined with the following table

	nc	**d**	**undef**
nc	nc	nc	nc
c	nc	if c=d then c else nc	c
undef	nc	d	undef

In semilattices: $\mu \le \nu$ iff $\mu \wedge \nu = \mu$.

Also: $\mu \wedge \nu \le \mu$ and $\mu \wedge \nu \le \nu$

Thus, in reaching definitions, $x \le y$ is equivalent to $y \subseteq x$.

In available expressions $x \le y$ is equivalent to $x \subseteq y$.

Basic assumptions

Definition. Given a semilattice $(L,\wedge)$ of finite length with a top element, a set of operations F on L is said to be a *monotone operation space associated with L* iff the following four conditions are satisfied.

1. Each $f\in F$ is monotonic. That is

$(\forall f\in F)(\forall x,y\in L)\ [x\leq y\rightarrow f(x)\leq f(y)]$.

This is equivalent to:

$(\forall f\in F)(\forall x,y\in L)[f(x\wedge y)\leq f(x)\wedge f(y)]$

2. There is $I \in F$ such that $\forall\ x \in L$, $I(x)=x$

3. $\forall f,g \in F$ there is $fg \in F$ such that $\forall x\in L$ $fg(x)=f(g(x))$

4. For each $x \in L$ there exists $f \in F$ such that $x=f(\perp)$

Definition Given a semilattice $(L,\wedge)$ with a top element $\top$ in L (i.e., and element $\top$ such that $\top\wedge m=m\ \forall\ m \in L$) , a *monotone data flow analysis framework* is a triple $(L,\wedge,F)$ where F is a monotone operation space associated with L.

Recall that $\wedge$ is associative, commutative, and idempotent.

Definition. Distributivity: $f(\mu\wedge\nu)=f(\mu)\wedge f(\nu)$

The Meet-Over-All-Paths Solution to Data-Flow Problems

Consider a monotone dataflow framework $(L,\wedge,F)$. Assume that the functions $f_B \in F$ represent the effect of a basic block on the sets conatining data for a particular dataflow problem.

Let $f_P(x)=f_{Bk-1}(\ldots f_{B1}(f_{B0}(x) \ldots)$ for a path P = (B0, B1, ..., Bk-1)

The solution to the dataflow problem is

mop(B) = $\bigwedge$(over all paths from B0 to B) $f_p(T)$, where $T \geq s$ for all elements of the semilattice L.

When the f_B are montone and distributive, the mop can be computed as follows

```
foreach node B do
        out[B] = $f_B[T]$
od
while changes to any out occur do
        foreach block B in depth-first order do
                in[B] = $\wedge$ (over all predecessors P of B) out[P]
                out[B]=$f_B$(in[B])
        od
od
```

Avaialble expressions, live analysis, reaching definitions have monotone distributive f's.

Constant propagation is not distributive.

Computing Live Variable Using Interval Analysis

Let in[x] be the live definitions at the beginning of block x. This block could be a basic block or represent an interval.

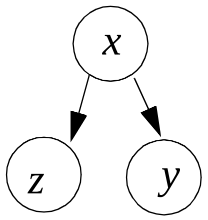
Live variable will be computed in two phases. First, we will present the second phase.

Algorithm LVIA-2:Live variable using interval analysis-Pass 2.

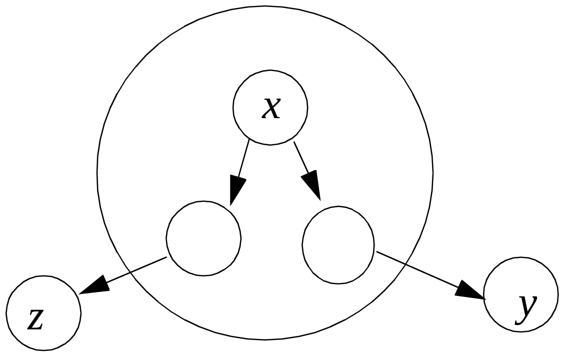
Input: The derived sequence of control flow graphs G_0, G_1, ..., G_m. Where G_0 is the original graph, and G_m is the trivial graph.

The set use[x] of variables with upwardly exposed uses in block x. Block x could be a basic block (graph G_0) or represent an interval (graphs G_1 thru G_m).

The set notdef[x,y] of variables not necessarily defined when block x is traversed towards y.



In the single node case*:*
notdef(x,y) = notdef(x,z)= U-def(x)



For a supernode the definition is more involved.

Output: in[x] the set of live definitions for all blocks x.

Method:

```
begin
    in[N] = use[N] /* N is the single node in G_m */
    for each G = G_{m-1}, ... G_0 do
        for each interval I of G do
            in[head(I)] = in [I]
            for each J in SUCC(I) do
                in [head(J)] = in [J]
            od
            /* in [I] and in[J] are available from the previous
            iteration */
            for each x in I - head(I) in reverse interval order do
                $in[x] = use[x] \cup \bigcup_{y \in SUCC(x)} (notdef[x,y] \cap in[y])$
            od
        od
    od
end
```

To compute use[x] and notdef[x,y] we apply the following algorithm:

Algorithm LVIA-1:Live variable using interval analysis-Pass 1.

Input: The derived sequence of control flow graphs G_0, G_1, ..., G_m. Where G_0 is the original graph, and G_m is the trivial graph.

The set *use[x]* of variables with upwardly exposed uses in basic block *x* of graph G_0.

The set *notdef[x]=U-def[x]* of variables not necessarily defined in basic block *x* of graph G_0. Notice that, since *x* is a basic block, *notdef[x,y]=notdef[x]* for all *y* in *SUCC(x)*.

Intermediate: For each *x* in interval *I*, path[x], the set of variables *V* for which there is a clear path (not containing a store into *V*) from the entry of *I* to the entry of *x*.

Output: *use[x]* for all blocks *x* in graphs G_1, ..., G_m. *notdef[x,y]* for all pairs of connected blocks *x* and *y* (i.e. *y* is a successor of *x* in some graph G_i).

Method:

```
begin
    for each $G = G_0, \ldots G_{m-1}$ do
        for each interval $I$ of $G$ do
            $use[I] = use[head(I)]$
            $path[head(I)] = \mathbf{U}$ /* U is the set of all variables */
            /*use[head(I)] is available from the previous iteration */
            for each $x$ in $I$ - $head(I)$ in interval order do
                $path[x] = \bigcup_{y \in PRED(x)} (path[y] \cap notdef[y, x])$
                $use[I] = use[I] \cup (path[x] \cap use[x])$
            od
            for each $J$ in $SUCC(I)$ do
                $notdef[I, J] = \bigcup_{y \in PRED(head(J)) \cap I} (path[y] \cap notdef[y, head(J)])$
            od
        od
    od
end
```

Reaching Definitions Using T1 & T2

T1 and T2 generate regions possibly nested within each other.

To compute the reaching definitions proceed as follows:

For each region and basic block B compute from the inside-out

gen[R,B] and *kill[R,B]*

which correspond to the definitions generated and killed, respectively, along paths within the region from the header to the end of block B.

Start with: *gen[B,B] = gen[B]* and *kill[B,B]=kill[B]*

Every time that T2 is applied to have region R1 consume region R2 we have:

- If B is in R1, then *gen[R,B]=gen[R1,B]* and *kill[R,B]=kill[R1,B]*
- If B is in R2 then

$$gen[R, B] = gen[R2, B] \cup \left(\left(\bigcup_{C \in PRED(head(R2))} gen[R1, C]\right) - kill[R2, B]\right)$$

$$kill[R, B] = kill[R2, B] \cup \left(\left(\bigcap_{C \in PRED(head(R2))} kill[R1, C]\right) - gen[R2, B]\right)$$

Every time that T1 is applied to transform region R1 with a self loop into region R we have:

$$gen[R, B] = gen[R1, B] \cup \left(\bigcup_{C \in PRED(head(R1))} gen[R1, C] \right) - kill[R1, B]$$

$$kill[R, B] = kill[R1, B]$$

Once the process terminates with the trivial graph, *U* and

$$out[B] = gen[U, B]$$

$$in[B] = \bigcup_{C \in PRED[B]} gen[U, C]$$

Computing Reaching Definitions with Interval Analysis

As in the case of live variables, we proceed in two passes. Again, the result will be in in[x] the set of all definitions reaching block x.

Algorithm RDIA-2:Reaching Definitions with interval analysis-Pass 2.

Input: The derived sequence of control flow graphs G_0, G_1, ..., G_m. Where G_0 is the original graph, and G_m is the trivial graph.

The set notkill[x,y] of definitions not necessarily killed when block x is traversed towards y.

The set gen[x,y] of definitions generated when block x is traversed towards y.

Output: in[x] the set of all definitions reaching block x.

```
begin
    in[N] = $\Phi$
    for each G= $G_{m-1}$, ... $G_0$ do
        for each interval I of G do
            in[head(I)] = in [I]
            /* in [I] is available from the previous iteration */
            for each x in I - head(I) in interval order do
                $in[x] = \bigcup_{y \in PRED(x)} in[y] \cap notkill[y,x] \cup gen[y,x]$
            od
        od
end
```

To compute gen[x,y] and notkill[x,y] we apply the following algorithm:

Algorithm RDIA-1: Reaching Definitions with interval analysis-Pass 1.

Input: The derived sequence of control flow graphs $G_0, G_1, ..., G_m$. Where G_0 is the original graph, and G_m is the trivial graph.

The set *notkill[x]=U-kill[x]* of definitions not necessarily killed in basic block *x* of graph G_0. Notice that, since *x* is a basic block, *notkill[x,y]=notkill[x]* for all *y* in *SUCC(x)*.

The set *gen[x]* of definitions generated in basic block *x* of graph G_0. Notice that, since *x* is a basic block, *gen[x,y]=gen[x]* for all *y* in *SUCC(x)*.

Intermediate: For each *x* in interval *I*, *path[x]*, the set of variables *V* for which there is a clear path (not containing a store into *V*) from the entry of *I* to the entry of *x*.

For each block *x*, *rdtop[x]*, the set of defintions that reach the top of *x* from nodes within the interval.

Output: *notkill[x,y] and gen[x,y]*.

Method:

```
begin
    for each $G = G_0, ... G_{m-1}$ do
    for each interval $I$ of $G$ do
        $rdtop[head(I)] = \Phi$
        $path[head(I)] = \mathbf{U}$ /* $\mathbf{U}$ is the set of all variables */
        for each $x$ in $I - head(I)$ in interval order do
            $path[x] = \bigcup_{y \in PRED(x)} (path[y] \cap notkill[y, x])$
            $rdtop[x] = \bigcup_{y \in PRED(x)} rdtop[y] \cap notkill[y, x] \cup gen[y, x]$
        od
        for each $J$ in $SUCC(I)$ do
            $notkill[I, J] = \bigcup_{y \in PRED(head(J)) \cap I} (path[y] \cap notkill[y, head(J)])$
            $gen[I, J] = \bigcup_{y \in PRED(head(J)) \cap I} A(y, J) \cup B(I, J)$
  Where
      $A(y, J) = rdtop[y] \cap notkill[y, head(J)] \cup gen[y, head(J)]$
   and
      $B(I, J) = \bigcup_{z \in PRED(head(I)) \cap I} ((rdtop[z] \cap notkill[z, head(I)]) \cup gen[z, head(I)]) \cap notkill[I, J]$
        od
    od
  od
end
```