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 \( c \times i + d \).

We define as basic induction variables those scalar variables whose only assignments in loop L have the form \( i = i \pm c \).
begin
Find the basic induction variables. Associate with each
basic induction variable \(i\), the triple \((i,1,0)\).

if for each variable \(k\) with a single assignment to \(k\)
within \(L\) of the form \(k = j \pm b\) or \(k = j \times b\)
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 \((j,1,\pm b)\) 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 \(j\) from outside \(L\) reaches \(k\)
then
associate \((i,c,d\pm b)\) 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
    \( s = s + c \times n \) (notice that \( c \times n \) is a constant).
    Insert \( s = c \times i + d \) just before the loop.
  od
od
end
Another definition of 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:

\[
\begin{align*}
i &= j \pm k \\
i &= \pm j
\end{align*}
\]

where \(j\) and \(k\) are either induction variables or region constants.

These induction variables can be detected by the following algorithm:

Algorithm A²DIV: Another Algorithm for the Detection of 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 induction variables
Method:

begin
    IV = ∅
    for each instruction I in R do
        if I is of the form \(i = j \pm k\) or \(i = \pm j\) 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 \(i = j \pm k\) or \(i = \pm j\) or any operand \(\not\in IV \cup RC\) 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 \(x = \text{op}(x, z)\) and op is not one of store, negative, add or subtract, then \(x\) is not an induction variable.
3. if \(x = \text{op}(y, z)\) and \(y\) and \(z\) are not both elements of \(IV \cup RC\), then \(x\) is not an induction variable
Another Strength-Reduction Algorithm

begin
    PILE={S | rhs of S is \(i \times x\) with \(i \in IV\) and \(x \in IV \cup RC\)}
    DEF_{ij} = \{S | i or j are defined\}
    while PILE \(\neq \emptyset\) do
        S from PILE (\(e = i \times x\))
        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) \(\cup\) DEFS(x,S) do
        if there is a definition of \(v_{ix}\) in T then
            next
        elseif T is outside the loop then
            insert \(v_{ix} = i \times x\) just before the loop
        elseif T is of the form \(i = k\) then
            replace T with the sequence
            \([R: v_{ix} = k \times x,\]
            \(i = k]\)
            PILE += \{R\}
        elseif T is of the form \(i = k + 1\) and \(x \neq i\) then
            replace T with the sequence
            \([R1: v_{kx} = k \times x,\]
            \(R2: v_{1x} = 1 \times x,\]
            \(v_{ix} = v_{kx} + v_{1x},\]
            \(i = k + 1]\)
PILE += \{R1,R2\}

elseif \(x \equiv i\) and \(T\) is of the form \(i = k + 1\) then
replace \(T\) with the sequence

R1: \(v_{kk} = k \ast k\)
R2: \(v_{11} = 1 \ast 1\)
R3: \(v_{1k} = 1 \ast k\)
R4: \(v_{21k} = 2 \ast v_{1k}\)
    \(v_{21k+1} = v_{21k} + v_{11}\)
    \(i = k + 1\)

PILE += \{R1,R2,R3,R4\}

fi

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 that \( x = *p \) does not kill any definitions. 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 \( *p = a \) is deemed to 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 substracting 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 
   \( (S-\{(p, b)|\{p, b\} \text{ is in } S\}) + \{(p, a)\} \)
2. If \( I \) is \( p=q \pm c \) for pointer \( q \) and nonzero constant \( c \), then the result is
   \( (S-\{(p, b)|\{p, b\} \text{ is in } S\}) + \{(p, b)|\{q, b\} \text{ is in } S \text{ and } b \text{ is an array}\} \)
3. If \( I \) is \( p=q \), the the result is
   \( (S-\{(p, b)|\{p, b\} \text{ is in } S\}) + \{(p, b)|\{q, b\} \text{ is in } S\} \)
4. If \( I \) assigns to pointer \( p \) any other expression, then the result is
   \( (S-\{(p, b)|\{p, b\} \text{ is in } S\}) \)
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 $I_1, I_2, ..., I_n$, then $\text{trans}_B(S)$ is defined as $\text{trans}_{I_n}(\text{trans}_{I_{n-1}}(...\text{trans}_{I_2}(\text{trans}_{I_1}(S))...))$.

Now, we define $\text{out}[B] = \text{trans}_B(\text{in}[B])$ and

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

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

```plaintext
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 \( \text{call } p(z, z) \) is possible, or the following sequence is possible

```plaintext
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. Also, we will not distinguish among occurrences of a variable in different calls to the same procedure. 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.

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

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].
Computing Live Variable Using Interval Analysis

Let $\text{in}[x]$ 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 $\text{use}[x]$ 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 $\text{notdef}[x,y]$ of variables not necessarily defined when block $x$ is traversed towards $y$.

In the single node case:

$$\text{notdef}(x,y) = \text{notdef}(x,z) = U-\text{def}(x)$$

For a supernode the definition is more involved.

*Output:* $\text{in}[x]$ the set of live definitions for all blocks $x$. 
Method:

\[
\text{begin } \\
\quad \text{in}[N] = \text{use}[N] /* N is the single node in } G_m */ \\
\quad \text{for each } G = G_{m-1}, \ldots, G_0 \text{ do} \\
\quad \quad \text{for each interval } I \text{ of } G \text{ do} \\
\quad \quad \quad \text{in[head(I)] = in [I]} \\
\quad \quad \quad \text{for each } J \text{ in } \text{SUCC}(I) \text{ do} \\
\quad \quad \quad \quad \text{in [head(J)] = in [J]} \\
\quad \quad \quad \text{od} \\
\quad \quad /* \text{in} [I] \text{ and in[J] are available from the previous iteration */} \\
\quad \text{for each } x \text{ in } I - \text{head}(I) \text{ in reverse interval order do} \\
\quad \quad \text{in}[x] = \text{use}[x] \cup \bigcup_{y \in \text{SUCC}(x)} (\text{notdef}[x, y] \cap \text{in}[y]) \\
\quad \text{od} \\
\text{od} \\
\text{end}
\]
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 containint 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
      \( \text{use}[I] = \text{use}[\text{head}(I)] \)
      \( \text{path}[\text{head}(I)] = U /* U \) is the set of all variables */
      /*\( \text{use}[\text{head}(I)] \) is available from the previous iteration */
      for each \( x \) in \( I - \text{head}(I) \) in interval order do
        \( \text{path}[x] = \bigcup_{y \in \text{PRED}(x)} (\text{path}[y] \cap \text{notdef}[y, x]) \)
      od
      \( \text{use}[I] = \text{use}[I] \cup (\text{path}[x] \cap \text{use}[x]) \)
    od
    for each \( J \) in \( \text{SUCC}(I) \) do
      \( \text{notdef}[I, J] = \bigcup_{y \in \text{PRED}(\text{head}(J)) \cap I} (\text{path}[y] \cap \text{notdef}[y, \text{head}(J)]) \)
    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

\[ \text{gen}[R,B] \text{ and } \text{kill}[R,B] \]

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: \( \text{gen}[B,B] = \text{gen}[B] \text{ and } \text{kill}[B,B] = \text{kill}[B] \)

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

- If B is in R1, then \( \text{gen}[R,B] = \text{gen}[R1,B] \text{ and } \text{kill}[R,B] = \text{kill}[R1,B] \)
- If B is in R2 then

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

\[
\text{kill}[R,B] = \text{kill}[R2,B] \cup \left( \bigcup_{C \in \text{PRE}(\text{head}(R2))} \text{kill}[R1,C] \right) - \text{gen}[R2,B]
\]
Every time that $T_1$ is applied to transform region $R_1$ with a self loop into region $R$ we have:

$$gen[R, B] = gen[R_1, B] \cup \left( \bigcup_{C \in \text{PRED}(\text{head}(R_1))} gen[R_1, C] \right) - kill[R_1, B]$$

$$kill[R, B] = kill[R_1, B]$$

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

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

$$in[B] = \bigcup_{C \in \text{PRED}[B]} gen[U, C]$$
Computing Reaching Definitions with Interval Analysis

As in the case of live variables, we proceed in two passes. Again, the result will be in \( \text{in}[x] \) 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 \( G_0, G_1, \ldots, G_m \).
Where \( G_0 \) is the original graph, and \( G_m \) is the trivial graph.

The set \( \text{notkill}[x,y] \) of definitions not necessarily killed when block \( x \) is traversed towards \( y \).

The set \( \text{gen}[x,y] \) of definitions generated when block \( x \) is traversed towards \( y \).

*Output:* \( \text{in}[x] \) the set of all definitions reaching block \( x \).
begin

\[ in[N] = \Phi \]

for each \( G = G_{m-1}, \ldots, G_0 \) do

for each interval \( I \) of \( G \) do

\[ in[\text{head}(I)] = in[I] \]

/* \( in[I] \) is available from the previous iteration */

for each \( x \) in \( I - \text{head}(I) \) in interval order do

\[ in[x] = \bigcup_{y \in PRED(x)} in[y] \cap \text{notkill}[y, x] \cup \text{gen}[y, x] \]

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 $G_0, G_1, \ldots, G_m$. Where $G_0$ is the original graph, and $G_m$ is the trivial graph.

The set $notkill[x]=U\text{-}kill[x]$ 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 definitions that reach the top of $x$ from nodes within the interval.

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

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