15-745

SSA Dominators

Copyright © Seth Copen Goldstein 2001-7

The $\Phi$ function

- $\Phi$ merges multiple definitions along multiple control paths into a single definition.
- At a BB with $p$ predecessors, there are $p$ arguments to the $\Phi$ function.
  
  $x_{\text{new}} \leftarrow \Phi(x_1, x_1, x_1, \ldots, x_p)$

- How do we choose which $x_i$ to use?
  - Most compiler writers don’t really care!
  - If we care, use moves on each incoming edge
    (Or, as in pegasus use a mux)

Trivial SSA

- Each assignment generates a fresh variable.
- At each join point insert $\Phi$ functions for all live variables.

```
x <- 1
y <- x
y <- 2

z <- y + x
```

Minimal SSA

- Each assignment generates a fresh variable.
- At each join point insert $\Phi$ functions for all variables with multiple outstanding defs.

```
x <- 1
y <- x
y <- 2

z <- y + x
```
When do we insert \( \Phi \)?

- Insert a \( \Phi \) function for variable \( A \) in block \( Z \) iff:
  - \( A \) was defined more than once before (i.e., \( A \) defined in \( X \) and \( Y \) AND \( X \neq Y \))
  - \( Z \) is the first block that joins the paths from \( X \) to \( Z \) and \( Y \) to \( Z \)

- Entry block implicitly defines all vars
- Note: \( A = \Phi(...) \) is a def of \( A \)
When do we insert $\Phi$?

**Def-use property of SSA**

- If $x_i$ is used in $x \leftarrow \Phi(..., x_i, ...)$, then NO BBs in any path from BB($x_i$) to BB($\Phi$) include def of $x$ except BB($x_i$) and BB($\Phi$)
- If $x$ is used in $y \leftarrow ...x ...$, then no BBs in path from BB($x$) to BB($y$) define $x$ except BB($x$)

Another way to say this: Definitions dominate uses
Dominance Property of SSA

• In SSA definitions dominate uses.
  - If \( x_i \) is used in \( x \leftarrow \Phi(..., x_i, ...) \), then \( BB(x_i) \) dominates \( i \)th pred of \( BB(\Phi) \).
  - If \( x \) is used in \( y \leftarrow ... \times ... \), then \( BB(x) \) dominates \( BB(y) \).

• Use this for an efficient alg to convert to SSA.

A little side trip

• Computing dominators

• \( d \) dom \( n \) iff every path from \( s \) to \( n \) goes through \( d \).
• \( n \) dom \( n \) for all \( n \).

Some definitions:
- immediate dominator: \( d = idom(n) \) iff
  - \( d \neq n \)
  - \( d \) dom \( n \)
  - \( d \) doesn't dominate any other dominator of \( n \).
- strictly dominates: \( s = sdom(n) \) iff
  - \( s \) dom \( n \)
  - \( s \neq n \).

Examples

• \( d \) dom \( n \) iff every path from Entry to \( n \) contains \( d \).
  
  1 dom 1 ; 1 dom 2 ; 1 dom 3 ; 1 dom 4 ;
  2 dom 2 ; 2 dom 3 ; 2 dom 4 ; 3 dom 3 ;
  4 dom 4

• \( s \) strictly dominates \( n \), \( s = sdom(n) \), iff \( s \) dom \( n \) and \( s \neq n \).
  
  1 sdom 2 ; 1 sdom 3 ; 1 sdom 4 ;
  2 sdom 3 ; 2 sdom 4

• \( d \) immediately dominates \( n \), \( d = idom(n) \), iff \( d \) sdom \( n \) and there is no node \( x \) such that \( d \) dom \( x \) and \( x \) dom \( n \).
  
  1 idom 2 ; 2 idom 3 ; 2 idom 4

Properties of dominators

• idom(\( n \)) is unique

• The dominance relation is a partial ordering; that is, it is reflexive, anti-symmetric and transitive:
  - reflexive:
    \( x \) dom \( x \)
  - anti-symmetric:
    \( x \) dom \( y \) and \( y \) dom \( x \) \( \rightarrow \) \( x = y \)
  - transitive:
    \( x \) dom \( y \) and \( y \) dom \( z \) \( \rightarrow \) \( x \) dom \( z \)

(adapted from: http://www.eecg.toronto.edu/~voss/ece540/)
The dominator tree

- One can represent dominators in a CFG as a tree of immediate dominators.
- In dominator tree, edge from parent to child if parent idom child in the CFG.
- The set of dominators of a node are the nodes from the root to the node.

Computing Dominators

- \( d \ \text{dom} \ n \) iff every path from \( s \) to \( n \) goes through \( d \)
- Note: \( n \ \text{dom} \ n \) for all \( n \)
- If \( s \ \text{dom} \ d \ & d \neq n \ & p_i \ \in \ \text{pred}(n) \ & d \ \text{dom} \ p_i \), then \( d \ \text{dom} \ n \)
- How can we use this?

Simple iterative alg

- \( \text{dom}(\text{Entry}) = \text{Entry} \)
- for all other nodes, \( n \), \( \text{dom}(n) = \) all nodes changed = true
  while (changed) {
    changed = false
    for each \( n, n \neq \text{Entry} \) {
      old = \( \text{dom}(n) \)
      \( \text{dom}(n) = \{n\} \cup \bigcap_{p \in \text{pred}(n)} \text{dom}(p) \)
      if (\( \text{dom}(n) \neq \text{old} \)) changed = true
    }
  }
Finding immediate dominators

- idom(n) dominates n, isn't n, and, doesn't strictly dominate any other sdom n
- Init idom(n) to nodes which sdom n
- foreach x ∈ idom(n) 
  foreach y ∈ idom(n) - {x} 
  if (y ∈ sdom(x)) idom(n)=idom(n)-{y}

Example (immediate dominators)

- dom(1) = {Entry}
- dom(2) = {Entry,1}
- dom(3) = {Entry,1,2}
- dom(4) = {Entry,1,2,3}
- dom(5) = {Entry,1,2,3,4}
- dom(6) = {Entry,1,2,3,4,5}
- dom(7) = {Entry,1,2,3,4,5,6}
- dom(8) = {Entry,1,2,3,4,5,7}
- dom(9) = {Entry,1,2,3,4,9}
- dom(10) = {Entry,1,2,10}

Dominance Property of SSA

- In SSA definitions dominate uses.
  - If xᵢ is used in x ← Φ(..., xᵢ, ...), then BB(xᵢ) dominates ith pred of BB(PHI)
  - If x is used in y ← ...x ..., then BB(x) dominates BB(y)

- Use this for an efficient alg to convert to SSA
Dominance

x strictly dominates w (s sdom w) iff x dom w AND x ≠ w

If there is a def of a in block 5, which nodes need a Φ()

Dominance Frontier

The dominance Frontier of a node x = { w | x dom pred(w) AND !(x sdom w)}

Dominance Frontier & path-convergence

c is an example of the successors of n not strictly dominated by n

Computing DF(n)
Computing DF(n)

\[
\text{DF}(n) = \{ w \mid x \text{ dom pred}(w) \text{ AND } !(x \text{ sdom } w) \}
\]

Computing the Dominance Frontier

\[
\begin{align*}
\text{compute-DF}(n) \\
S &= \{ \} \\
\text{foreach node } y \text{ in succ}[n] \\
&\quad \text{if idom}(y) \neq n \\
&\quad \quad S = S \cup \{ y \} \\
\text{foreach child of } n, c, \text{ in D-tree} \\
&\quad \text{compute-DF}(c) \\
&\quad \text{foreach } w \text{ in DF}[c] \\
&\quad \quad \text{if } !n \text{ sdom } w \\
&\quad \quad \quad S = S \cup \{ w \} \\
\text{DF}[n] &= S
\end{align*}
\]

Using DF to compute SSA

- place all \( \Phi() \)
- Rename all variables

Using DF to Place \( \Phi() \)

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DF(defsite)
      - if we haven't put \( \Phi() \) in node put one in
      - If this node didn't define the variable before: add this node to the defsites

- This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of \( \Phi() \) necessary
Using DF to Place $\Phi()$

```plaintext
foreach node n {
    foreach variable v defined in n {
        orig[n] $\cup$ = \{v\}
        defsites[v] $\cup$ = \{n\}
    }
    foreach variable v {
        W = defsites[v]
        while W not empty {
            foreach y in DF[n]
            if y $\notin$ PHI[v] {
                insert "v $\leftarrow$ $\Phi(v,v,\ldots)$" at top of y
                PHI[v] = PHI[v] $\cup$ \{y\}
                if v $\notin$ orig[y]: W = W $\cup$ \{y\}
            }
        }
    }
}
```

Renaming Variables

- Walk the D-tree, renaming variables as you go
- Replace uses with more recent renamed def
  - For straight-line code this is easy
  - If there are branches and joins?

Renaming Variables

```
• Walk the D-tree, renaming variables as you go
• Replace uses with more recent renamed def
  – For straight-line code this is easy
  – If there are branches and joins?

Easy implementation:
  – for each var: rename (v)
  – rename(v): replace uses with top of stack
    at def: push onto stack
    call rename(v) on all children in D-tree
    for each def in this block pop from stack
```

```
foreach var a
    a.count = 0
    a.stack = empty
    a.stack.push(0)
rename(entry)
rename(n){
    foreach s in block n
    if s isn't $\Phi$
        foreach use of x in S
        replace x with x.stack.top()
    foreach def of x in S
        i = ++x.count
        x.stack.push(i)
        replace x with x,
```
rename(n) {
    foreach s in block n
        if s isn't Φ
            foreach use of x in S
                replace x with x \_stack.top()
            foreach def of x in S
                i = ++x.count
                x \_stack.push(i)
                replace x with x
            foreach y ∈ succ(n)
                j = pred # of n in y
                foreach Φ in y
                    i ← var-j \_stack.top()
                    replace var-j with var-j
                foreach child X of n in D-tree: rename(X)
            foreach def, x, in S: x \_stack.pop()
}

Compute D-tree

Compute Dominance Frontier
Insert $\Phi()$

1. $i \leftarrow 1$
2. $j \leftarrow 1$
3. $k \leftarrow 0$

4. $j \leftarrow \Phi(j,j)$
5. $k < 100?$
6. $j < 20?$
7. return $j$

DFs
var $i$: $W=\{1\}$

DF$\{1\}$, DF$\{5\}$

---

Insert $\Phi()$

1. $i \leftarrow 1$
2. $j \leftarrow 1$
3. $k \leftarrow 0$

4. $j \leftarrow \Phi(j,j)$
5. $k < 100?$
6. $j < 20?$
7. return $j$

DFs
var $j$: $W=\{1,5,6\}$

DF$\{1\}$, DF$\{5\}$

---

Insert $\Phi()$

1. $i \leftarrow 1$
2. $j \leftarrow 1$
3. $k \leftarrow 0$

4. $j \leftarrow \Phi(j,j)$
5. $k < 100?$
6. $j < 20?$
7. return $j$

DFs
var $j$: $W=\{1,5,6\}$

DF$\{1\}$, DF$\{5\}$, DF$\{6\}$
Insert $\Phi()$

1. $i \leftarrow 1$
2. $j \leftarrow 1$
3. $k \leftarrow 0$

4. $j \leftarrow \Phi(j,j)$
5. $k \leftarrow \Phi(k,k)$
6. $k < 100?$
7. $j < 20?$

return $j$

Rename Vars

1. $i \leftarrow 1$
2. $j \leftarrow 1$
3. $k \leftarrow 0$
4. $j \leftarrow \Phi(j,j)$
5. $k \leftarrow \Phi(k,k)$
6. $k < 100?$
7. $j < 20?$

return $j$

var $k$: $W = \{1,5,6\}$

Var | Count | stack
---|---|---
i | 0 | 0
j | 0 | 0
k | 0 | 0

rename(1) - defs & uses
### Rename Vars

```
1. i₁ ← 1
   j₁ ← 1
   k₁ ← 0

2. j₂ ← Φ(j₁,j)
   k₂ ← Φ(k₁,k)
   k₂ < 100?

3. j₂ < 20?
   return j₂

4. j₂ ≥ 20?

5. j₃ ← i₂
   k₃ ← k₂ + 1

6. j₄ ← k
   k₄ ← k + 2

7. j ← Φ(j₃,j₄)
   k ← Φ(k₃,k₄)
```

Variable Rename Table:

<table>
<thead>
<tr>
<th>Var</th>
<th>Count</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
<td>0 1</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>0 1 2</td>
</tr>
<tr>
<td>k</td>
<td>3</td>
<td>0 1 2</td>
</tr>
</tbody>
</table>

### SSA Properties

- Only 1 assignment per variable
- Definitions dominate uses
Dead Code Elimination

W <- list of all defs
while !W.isEmpty {
    Stmt S <- W.removeOne
    if |S.users| != 0 then continue
    if S.hasSideEffects() then continue
    foreach def in S.definers {
        def.users <- def.users - {S}
        if |def.uses| == 0 then
            W <- W UNION {def}
    }
}

Since we are using SSA, this is just a list of all variable assignments.

Aggressive Dead Code Elimination

Assume a stmt is dead until proven otherwise.

init:
    mark as live all stmts that have side-effects:
    - I/O
    - stores into memory
    - returns
    - calls a function that MIGHT have side-effects
As we mark S live, insert S.defs into W

while (|W| > 0) {
    S <- W.removeOne()
    if (S is live) continue;
    mark S live, insert S.defs into W
}
**Example DCE**

```
B0 i0<-0
j0<-0

B1 j1 ← Φ(j0, j2)
i1 ← Φ(i0, i2)
i2<-i1*2
j2<-j1+1
j2<10?
B2 return j2
```

```
B0 i0<-0
j0<-0

B1 j1 ← Φ(j0, j2)
i1 ← Φ(i0, i2)
i2<-i1*2
j2<-j1+1
j2<10?
B2 return j2
```

**Problem!**

**Fixing DCE**

If $S$ is live, then
forall users of $S$.def
if user is a branch -> mark user as live

**Control Dependence**

$Y$ is control-dependent on $X$ if
- $X$ branches to $u$ and $v$
- $\exists$ a path $u$→exit which does not go through $Y$
- $\forall$ paths $v$→exit go through $Y$

IOW, $X$ can determine whether or not $Y$ is executed.

**Aggressive Dead Code Elimination**

Assume a stmt is dead until proven otherwise.

```
while (|W| > 0) {
    S ← W.removeOne()
    if (S is live) continue;
    mark S live, insert
    - forall operands, S.operand.definers into W
    - S.CD^1 into W
}
```
Example DCE

```
B0: i0 <- 0
   j0 <- 0

B1:
   j1 <- (j0, j2)
   i1 <- (i0, i2)
   i2 <- i1 * 2
   j2 <- j1 + 1
   j2 < 10?
   B2: return j2
```

CCP Example

```
i <- 1
j <- 1
k <- 0

k < 100?

j < 20?
return j

j <- i
k <- k + 1
j <- k
k <- k + 2
```

• Does block 6 ever execute?
  • Simple CP can’t tell
  • CCP can tell:
    • Assumes blocks don’t execute until proven otherwise
    • Assumes Values are constants until proven otherwise

CCP -> DCE

```
i1 <- 1
j1 <- 1
k1 <- 0

k2 <- (k3, 0)

k2 < 100?

k3 < k2 + 1
return 1
```

Small problem.
Finding the CDG

Y is control-dependent on X if
• X branches to u and v
• ∃ a path u→exit which does not go through Y
• ∀ paths v→exit go through Y

IOW, X can determine whether or not Y is executed.

Finding the CDG

• Construct CFG
• Add entry node and exit node
• Add (entry, exit)
• Create G', the reverse CFG
• Compute D-tree in G' (post-dominators of G)
• Compute DF_{G'}(y) for all y ∈ G' (post-DF of G)
• Add (x, y) ∈ G to CDG if x ∈ DF_{G'}(y)
Summary

• In SSA definitions dominate uses
• Use Dominance Frontier to create minimal SSA
  - Compute Dominance Tree
  - Compute DF
  - Compute Iterated Dominance Frontier
• Use D-Tree to rename variables
• Control Dependence can be computed by inspecting post-dominators, IOW, DF on reverse graph