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.
  
$$x_{new} \leftarrow \Phi(x_1, x_1, x_2, \ldots, x_p)$$

- How do we choose which $x_i$ to use?
  - We don’t really care!
  - If we care, use moves on each incoming edge

**Minimal SSA**

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

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
  - 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 $\Phi()$

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

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?

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

Compute D-tree

$1 \leftarrow 1$
$1 \leftarrow k = 0$

$k < 100?$

$2 \leftarrow 2$
$2 \leftarrow 1$
$2 \leftarrow k = 1$

$j < 20?$

$3 \leftarrow 3$
$3 \leftarrow 1$
$3 \leftarrow j = 1$

$k < k + 1$

$4 \leftarrow 4$

$5 \leftarrow 5$
$5 \leftarrow j = k$

$k = k + 1$

$6 \leftarrow 6$
$6 \leftarrow j = k$

$k = k + 1$

$7 \leftarrow 7$

$1 \leftarrow 1$
$1 \leftarrow k = 0$

$k < 100?$

$2 \leftarrow 2$
$2 \leftarrow 1$
$2 \leftarrow k = 0$

$j < 20?$

$3 \leftarrow 3$
$3 \leftarrow 1$
$3 \leftarrow j = 1$

$k < k + 1$

$4 \leftarrow 4$

$5 \leftarrow 5$
$5 \leftarrow j = k$

$k = k + 1$

$6 \leftarrow 6$
$6 \leftarrow j = k$

$k = k + 2$

$7 \leftarrow 7$

DFs
SSA & Opts

Insert $\Phi()$

1 1 \{ \}
2 (i,j,k)
3 \{ \}
4 (j,j)
5 (k,k)
6 \{ \}
7 (j,j)

var j: W={1,5,6}
Df(1), Df(5)

Rename Vars

1 i←←←← 1
2 j←←←← 1
3 k←←←← 0
4 j←←←← ΦΦΦΦ (j,j)
5 k←←←← ΦΦΦΦ (k,k)
6 1
7 2
8 3
9 4
10 5
11 6
12 7

var k: W={1,5,6}
**Rename Vars**

1.  
2.  
3.  
4.  
5.  
6.  
7.  

**Computing DF(n)**

n dom a  
n dom b  
\( \text{n dom c} \)

**Computing the Dominance Frontier**

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

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

**SSA Properties**

- Only 1 assignment per variable
- definitions dominate uses

**Constant Propagation**

- If "v ← c", replace all uses of v with c
- If "v ← \( \Phi(c,c,c) \)" replace all uses of v with c

W ← list of all defs
while !W.isEmpty {
    Stmt S ← W.removeOne
    if S has form "v ← \( \Phi(c,c,c) \)"
        replace S with V ← c
    if S has form "v ← c" then
        delete S
    foreach stmt U that uses v,
        replace v with c in U
    W.add(U)
}
Other stuff we can do?

- Copy propagation
  - delete "$x \leftarrow \Phi(y)$" and replace all $x$ with $y$
  - delete "$x \leftarrow y$" and replace all $x$ with $y$
- Constant Folding
  (Also, constant conditions too!)
- Unreachable Code
  Remember to delete all edges from unreachable block

Constant Propagation

But, so what?
Conditional Constant Propagation

- 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

```
1 i1 ← 1 j1 ← 1 k1 ← 0
2 j2 ← (j4, 1) k2 ← (k4, 0) k2 < 100?
3 j3 ← 1 k3 ← k2 + 1
4 return j3
j4 ← (1, j5)
k4 ← (k3, k5)
5 j5 ← k2
6 k5 ← k2 + 2
7 j6 ← (1, k6)
k6 ← (k3, k5)
```

Tracks:
- Blocks (assume unexecuted until proven otherwise)
- Variables (assume not executed, only with proof of assignments of non-constant value do we assume not constant)

Use a lattice for variables:
- TOP = we have evidence that variable can hold different values at different times
- integers = we have seen evidence that the var has been assigned a constant with the value
- BOT = not executed
Conditional Constant Propagation

i1 ← 1
j1 ← 1
k1 ← 0

j1 ← (j2, 1)
k1 ← (k2, 0)
k1 < 100?

j2 < 20?
return j2

j3 ← 1
k3 ← k2 + 1
j5 ← k2
k5 ← k2 + 2

j4 ← (1, j5)
k4 ← (k3, k5)

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.

Example DCE

B0: i <- 0
    j <- 0
B1: i <- i * 2
    j <- j + 1
    j < 10?
B2: return j

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) {
    Stmt S ← W.removeOne
    if |S.users| == 0 then continue;
    mark S live, insert S.defs into W
}
Example DCE

Example DCE

Fixing DCE

Fixing DCE

Control Dependence

Control Dependence

Aggressive Dead Code Elimination

Aggressive Dead Code Elimination

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

B0: \( i_0 \leftarrow 0 \)  
\( j_0 \leftarrow 0 \)

B1: \( j_1 \leftarrow (j_0, j_2) \)  
\( i_1 \leftarrow (i_0, i_2) \)  
\( l_1 \leftarrow i_1 \cdot j_1 \)  
\( j_1 \leftarrow j_1 + 1 \)  
\( j_1 \leftarrow 107 \)

B2: \( \text{return } j_1 \)

**CCP Example**

i \( \leftarrow 1 \)
\( j \leftarrow 1 \)
\( k \leftarrow 0 \)

k < 100?  
\( k_2 \leftarrow \Phi(k_3, 0) \)  
\( k_2 < 100? \)

\( j < 20? \)
\( \text{return } j \)

\( j \leftarrow i \)
\( k \leftarrow k + 1 \)
\( k \leftarrow k + 2 \)

**CCP -> DCE**

\( l_1 \leftarrow 1 \)
\( j_1 \leftarrow 1 \)
\( k_1 \leftarrow 0 \)

k_2 \leftarrow \Phi(k_3, 0) \)  
\( k_2 < 100? \)

\( j_1 < j_1 + 1 \)  
\( j_1 < 107 \)

\( \text{return } j_1 \)

**Finding the CDG**

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.
Finding the CDG

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.

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)

CDG of example

exit: {}
2: {entry}
1: {1,entry}
0: {entry}
entry: {}