15-745 Lecture 4

SSA
CCP, ADCE
Dominance & Minimal SSA

From Before... Def-Use Chains

- ...
- for (i=0; i++; i<10) {
-   ... = ... i ...;
-   ...
- }
- for (i=j; i++; i<20) {
-   ... = i ...
- }

How is this related to RA?

Def-Use chains are expensive

foo(int i, int j) {
  ...
  switch (i) {
  case 0: x=3; break;
  case 1: x=1; break;
  case 2: x=6; break;
  case 3: x=7; break;
  default: x = 11;
  }
  switch (j) {
  case 0: y=x+7; break;
  case 1: y=x+4; break;
  case 2: y=x-2; break;
  case 3: y=x+1; break;
  default: y=x+9;
  }
  ...
}

In general,

\[ N \text{ defs} \times M \text{ uses} \Rightarrow O(NM) \text{ space and time} \]

A solution is to limit each var to ONE def site
Def-Use chains are expensive

foo(int i, int j) {
    ...
    switch (i) {
        case 0: x=3; break;
        case 1: x=1; break;
        case 2: x=6;
        case 3: x=7;
        default: x = 11;
    }
    x1 is one of the above x's

    switch (j) {
        case 0: y=x1+7;
        case 1: y=x1+4;
        case 2: y=x1-2;
        case 3: y=x1+1;
        default: y=x1+9;
    }
}

A solution is to limit each var to ONE def site

SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)

Advantages of SSA

- Makes du-chains explicit
- Makes dataflow optimizations
  - Easier
  - faster
- Improves register allocation
  - Automatically builds Webs
  - Makes building interference graphs easier
- For most programs reduces space/time requirements

SSA History

- Developed by Wegman, Zadeck, Alpern, and Rosen in 1988
- New to gcc 4.0, used in ORC, LLVM, used in both IBM and Sun Java JIT compilers
  - and others
Straight-line SSA

\[
\begin{align*}
  a & \leftarrow x + y \\
  b & \leftarrow a + x \\
  a & \leftarrow b + 2 \\
  c & \leftarrow y + 1 \\
  a & \leftarrow c + a
\end{align*}
\]

SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text.
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?

Merging at Joins

\[
\begin{align*}
  c & \leftarrow 12 \\
  \text{if (i) } \{ \\
  & a \leftarrow x + y \\
  & b \leftarrow a + x \\
  \} \text{ else } \{ \\
  & a \leftarrow b + 2 \\
  & c \leftarrow y + 1 \\
  \} \\
  a & \leftarrow c + a
\end{align*}
\]
SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text.
- Easy for a basic block:
  - Assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?
  - Use a notional fiction: A $\Phi$ function

Merging at Joins

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?
  - We don’t really care!
  - If we care, use moves on each incoming edge
**Trivial SSA**

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

```
\[\begin{align*}
x &\leftarrow 1 \\
y &\leftarrow x \\
y &\leftarrow 2 \\
x_1 &\leftarrow 1 \\
y_1 &\leftarrow x_2 \\
y_2 &\leftarrow 2 \\
x_2 &\leftarrow \Phi(x_1,x_1) \\
y_3 &\leftarrow \Phi(y_1,y_2) \\
z_1 &\leftarrow y_3 + x_2 \\
z &\leftarrow y + x
\end{align*}\]
```

Way too many $\Phi$ functions inserted.

**Minimal SSA**

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

```
\[\begin{align*}
x &\leftarrow 1 \\
y &\leftarrow x \\
y &\leftarrow 2 \\
x_1 &\leftarrow 1 \\
y_1 &\leftarrow x_2 \\
y_2 &\leftarrow 2 \\
z &\leftarrow y + x \\
y_3 &\leftarrow \Phi(y_1,y_2) \\
z_1 &\leftarrow y_3 + x_1
\end{align*}\]
```

**Another Example**

```
a &\leftarrow 0 \\
b &\leftarrow a + 1 \\
c &\leftarrow c + b \\
a &\leftarrow b * 2 \\
a &\leftarrow N \\
\text{return } c
```

**Another Example**

```
a &\leftarrow 0 \\
b &\leftarrow a + 1 \\
c &\leftarrow c + b \\
a &\leftarrow b * 2 \\
a &\leftarrow N \\
\text{return } c
```

Notice use of $c_1$
Lets optimize the following:

```plaintext
i = 1;
j = 1;
k = 0;
while (k<100) {
    if (j<20) {
        j = i;
k++;
    } else {
        j = k;
k+=2;
    }
} return j;
```

First, turn into SSA

```
i ← 1
j ← 1
k ← 0
```

```
while (k<100) {
    if (j<20) {
        j = i;
k++;
    } else {
        j = k;
k+=2;
    }
}
return j;
```

Properties of SSA

- Only 1 assignment per variable
- Definitions dominate uses
- Can we use this to help with constant propagation?

Constant Propagation

- If “v ← c”, replace all uses of v with c
- If “v ← Φ(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 ← Φ(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)
}
Constant Propagation

1
i₁ ← 1
j₁ ← 1
k₁ ← 0

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

3
j₂ < 20?
4 return j₂

5
j₃ ← 1
k₃ ← k₂ + 1
j₅ ← k₂
k₅ ← k₂ + 2

6
j₅ ← k₂
k₅ ← k₂ + 2

7
j₄ ← Φ(j₃, j₅)
k₄ ← Φ(k₃, k₅)

But, so what?
Other stuff we can do?

- **Copy propagation**
  delete "x ← Φ(y)" and replace all x with y
  delete "x ← 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**

1. \( i_1 ← 1 \)
2. \( j_1 ← 1 \)
3. \( k_1 ← 0 \)
4. \( j_2 ← Φ(j_4, 1) \)
5. \( k_2 ← Φ(k_4, 0) \)
6. \( k_2 < 100? \)
7. \( j_2 < 20? \)
8. \( return j_2 \)
9. \( j_3 ← 1 \)
10. \( k_3 ← k_2 + 1 \)
11. \( k_5 ← k_2 + 2 \)
12. \( j_4 ← Φ(1, j_3) \)
13. \( k_4 ← Φ(k_3, k_5) \)

But, so what?

**Conditional Constant Propagation**

1. \( i_1 ← 1 \)
2. \( j_1 ← 1 \)
3. \( k_1 ← 0 \)
4. \( j_2 ← Φ(j_4, 1) \)
5. \( k_2 ← Φ(k_4, 0) \)
6. \( k_2 < 100? \)
7. \( j_2 < 20? \)
8. \( return j_2 \)
9. \( j_3 ← 1 \)
10. \( k_3 ← k_2 + 1 \)
11. \( k_5 ← k_2 + 2 \)
12. \( j_4 ← Φ(1, j_3) \)
13. \( k_4 ← Φ(k_3, k_5) \)

**CCP data structures & lattice**

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

Use a lattice for variables:
- have evidence that variable can hold different values at different times
- evidence that the var has been assigned a constant
- not executed
Conditional Constant Propagation

1

\[ i_1 \leftarrow 1 \]
\[ j_1 \leftarrow 1 \]
\[ k_1 \leftarrow 0 \]

2

\[ j_2 \leftarrow \Phi(j_1, 1) \]
\[ k_2 \leftarrow \Phi(k_1, 0) \]
\[ k_2 < 100? \]

3

\[ j_2 < 20? \]

4

\[ \text{return } j_2 \]

5

\[ j_3 \leftarrow 1 \]
\[ k_3 \leftarrow k_2 + 1 \]

6

\[ j_5 \leftarrow k_2 \]
\[ k_5 \leftarrow k_2 + 2 \]

7

\[ j_4 \leftarrow \Phi(1, j_3) \]
\[ k_4 \leftarrow \Phi(k_3, k_5) \]
Conditional Constant Propagation

1

\[ i_1 \leftarrow 1 \]
\[ j_1 \leftarrow 1 \]
\[ k_1 \leftarrow 0 \]

2

\[ j_2 \leftarrow \text{Φ}(j_4, 1) \]
\[ k_2 \leftarrow \text{Φ}(k_4, 0) \]
\[ k_2 \leftarrow 100? \]

3

\[ j_2 \leftarrow \text{Φ}(j_4, 1) \]
\[ k_2 \leftarrow \text{Φ}(k_4, 0) \]
\[ k_2 \leftarrow 100? \]

4

\[ j_2 \leftarrow 20? \]
\[ \text{return } j_2 \]

5

\[ j_3 \leftarrow 1 \]
\[ k_3 \leftarrow k_2 + 1 \]
\[ k_3 \leftarrow k_2 + 2 \]

6

\[ j_3 \leftarrow 1 \]
\[ k_3 \leftarrow k_2 + 1 \]
\[ k_3 \leftarrow k_2 + 2 \]

7

\[ j_4 \leftarrow \text{Φ}(1, j_5) \]
\[ k_4 \leftarrow \text{Φ}(k_3, k_5) \]

8

\[ j_4 \leftarrow \text{Φ}(1, j_5) \]
\[ k_4 \leftarrow \text{Φ}(k_3, k_5) \]
When do we insert $\Phi$?

- We 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 $X \neq Y$)
  - There exists a non-empty path from $x$ to $z$, $P_{xz}$, and a non-empty from $y$ to $z$, $P_{yz}$ s.t.
    - $P_{xz} \cap P_{yz} = \{ z \}$
    - $z \notin P_{xq}$ or $z \notin P_{xr}$ where $P_{xz} = P_{xq} \to z$ and $P_{yz} = P_{xr} \to z$
  - Entry block contains an implicit def of all vars
  - Note: $A = \Phi(\ldots)$ is a def of $A$

Dominance Property of SSA

- In SSA definitions dominate uses.
  - If $x_i$ is used in $x \leftarrow \Phi(\ldots, x_i, \ldots)$, then $BB(x_i)$ dominates $ith$ pred of $BB(\Phi)$
  - If $x$ is used in $y \leftarrow \ldots x \ldots$, then $BB(x)$ dominates $BB(y)$
- We can use this for an efficient alg to convert to SSA
Dominance Frontier

If there is a def of a in block 5, which nodes need a \( \Phi() \)?

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

Computing Dominance Frontier

- You've probably already seen a \( O(n^3) \) iterative algorithm
- There's also a near linear time algorithm due to Tarjan and Lengauer (Chap 19.2)
  - SSA construction therefore near linear
  - SSA form makes many optimizations linear (no need for iterative data flow)
Side trip: Dominators

Dominators

- **a dom b**
  - block *a dominates* block *b* if every possible execution path from entry to *b* includes *a*
    - *entry* dominates everything
    - *0* dominates everything but entry
    - *1* dominates *2* and *1*

Dominators are useful in identifying “natural” loops

Properties of Dom

- Dominance is a partial order on the blocks of the flow graph, i.e.,
  1. Reflexivity: *a dom a* for all *a*
  2. Anti-symmetry: *a dom b* and *b dom a* implies *a = b*
  3. Transitivity: *a dom b* and *b dom c* implies *a dom c*

- NOTE: there may be blocks *a* and *b* such that neither *a dom b* or *b dom a* holds.

- The dominators of each node *n* are linearly ordered by the dom relation. The dominators of *n* appear in this linear order on any path from the initial node to *n*.

Definitions

- **a sdom b**
  - If *a* and *b* are different blocks and *a dom b*, we say that *a* strictly dominates *b*

- **a idom b**
  - If *a sdom b*, and there is no *c* such that *a sdom c* and *c sdom b*, we say that *a* is the immediate dominator of *b*
Computing dominators

- We want to compute $D[n]$, the set of blocks that dominate $n$

Initialize each $D[n]$ (except $D[entry]$) to be the set of all blocks, and then iterate until no $D[n]$ changes:

$$D[entry] = \{entry\}$$

$$D[n] = \{n\} \cup \bigcap_{p \in \text{pred}(n)} D[p], \text{ for } n \neq entry$$

Update rule:

$$D[n] = \{n\} \cup \bigcap_{p \in \text{pred}(n)} D[p]$$
Computing dominators

- Iterative algorithm is $O(n^2e)$
  - assuming bit vector sets
- More efficient algorithm due to Lengauer and Tarjan
  - $O(e \cdot \alpha(e,n))$
  - much more complicated
  - your book provides a simple algorithm that is very fast in practice
    - faster than Tarjan algorithm for any realistic CFG

Computing dominators

- Let $sD[n]$ be the set of blocks that strictly dominate $n$, then
  $$sD[n] = D[n] - \{n\}$$
- To compute $iD[n]$, the set of blocks (size $\leq 1$) that immediately dominate $n$
  $$iD[n] = sD[n]$$
- Set
- Repeat until no $iD[n]$ changes:
  $$iD[n] = iD[n] - \cup_{d \in iD[n]} sD[d]$$

Example

<table>
<thead>
<tr>
<th>block</th>
<th>Second Pass</th>
<th>Third Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>{entry}</td>
<td>{entry}</td>
</tr>
<tr>
<td>0</td>
<td>{0,entry}</td>
<td>{0,entry}</td>
</tr>
<tr>
<td>1</td>
<td>{1,0,entry}</td>
<td>{1,0,entry}</td>
</tr>
<tr>
<td>2</td>
<td>{2,1,0,entry}</td>
<td>{2,1,0,entry}</td>
</tr>
<tr>
<td>3</td>
<td>{3,0,entry}</td>
<td>{3,0,entry}</td>
</tr>
<tr>
<td>4</td>
<td>{4,0,entry}</td>
<td>{4,0,entry}</td>
</tr>
<tr>
<td>5</td>
<td>{5,4,0,entry}</td>
<td>{5,4,0,entry}</td>
</tr>
<tr>
<td>exit</td>
<td>{exit,3,0,entry}</td>
<td>{exit,3,0,entry}</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>block</th>
<th>Initialization</th>
<th>First Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>$iD[n] = sD[n]$</td>
<td>$iD[n] = \emptyset$</td>
</tr>
<tr>
<td>0</td>
<td>{entry}</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>{0,entry}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{1,0,entry}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{0,entry}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{0,entry}</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{4,0,entry}</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>{3,0,entry}</td>
<td></td>
</tr>
</tbody>
</table>

Update rule:
$$ iD[n] = iD[n] - \cup_{d \in iD[n]} sD[d] $$
In the dominator tree the initial node is the entry block, and the parent of each other node is its immediate dominator.

<table>
<thead>
<tr>
<th>block</th>
<th>ID[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>{}</td>
</tr>
<tr>
<td>0</td>
<td>(0)</td>
</tr>
<tr>
<td>1</td>
<td>(0)</td>
</tr>
<tr>
<td>2</td>
<td>(1)</td>
</tr>
<tr>
<td>3</td>
<td>(0)</td>
</tr>
<tr>
<td>4</td>
<td>(0)</td>
</tr>
<tr>
<td>5</td>
<td>(4)</td>
</tr>
</tbody>
</table>

In the dominator tree the initial node is the entry block, and the parent of each other node is its immediate dominator.

Block \(a\) post-dominates \(b\) (a \(pd\) dom \(b\)) if every path from \(a\) to the exit block includes \(b\).

\(pd\) dom on CFG is the same as \(dom\) on the reverse (all edges reversed) CFG.

- 0 post-dominates ?
- 1 post-dominates ?
- 4 post-dominates ?

If \(z\) is the first node we encounter on the path from \(x\) which \(x\) does not strictly dominate, \(z\) is in the dominance frontier of \(x\).

For some path from node \(x\) to \(z\), \(x \rightarrow \ldots \rightarrow y \rightarrow z\) where \(x\) dom \(y\) but not \(x\) sdom \(z\).

- Dominance frontier of 1?
- Dominance frontier of 2?
- Dominance frontier of 4?

Let \(dominates[n]\) be the set of all blocks which block \(n\) dominates.

- Subtree of dominator tree with \(n\) as the root.

The dominance frontier of \(n\), \(DF[n]\) is

\[
DF[n] = \left( \bigcup_{s \in \text{dominates}[n]} \text{succs}(s) \right) - (\text{dominates}[n] - \{n\})
\]
Example

First calculate dominates[n] from the dominator tree

<table>
<thead>
<tr>
<th>block</th>
<th>dominates[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>(entry,0,1,2,3,4,5,exit)</td>
</tr>
<tr>
<td>0</td>
<td>(0,1,2,3,4,5,exit)</td>
</tr>
<tr>
<td>1</td>
<td>(1,2)</td>
</tr>
<tr>
<td>2</td>
<td>(2)</td>
</tr>
<tr>
<td>3</td>
<td>(3,exit)</td>
</tr>
<tr>
<td>4</td>
<td>(4,5)</td>
</tr>
<tr>
<td>5</td>
<td>(5)</td>
</tr>
<tr>
<td>exit</td>
<td>(exit)</td>
</tr>
</tbody>
</table>

Dominator Tree

Then compute the successor set of dominates[n]

<table>
<thead>
<tr>
<th>block</th>
<th>dominates[n]</th>
<th>succ(dominates[n])</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>{entry,0,1,2,3,4,5,exit}</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>{0,1,2,3,4,5,exit}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{1,2}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{2}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{3,exit}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{4,5}</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{5}</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>{exit}</td>
<td></td>
</tr>
</tbody>
</table>

Finally, remove all the blocks from the successor set that are strictly dominated by n to get DF[n]

<table>
<thead>
<tr>
<th>block</th>
<th>sdominates[n]</th>
<th>succ(dominates[n])</th>
<th>DF[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>(entry,0,1,2,3,4,5,exit)</td>
<td>(0,1,2,3,4,5,exit)</td>
<td>{}</td>
</tr>
<tr>
<td>0</td>
<td>(0,1,2,3,4,5,exit)</td>
<td>(1,2,3,4,5,exit)</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>(1,2)</td>
<td>(2,3)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(2)</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(3,exit)</td>
<td>(exit)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(4,5)</td>
<td>(3,4,5)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(5)</td>
<td>(3,4)</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>(exit)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example
Recap

- \( a \text{ dom } b \)
  - every possible execution path from entry to \( b \) includes \( a \)
- \( a \text{ sdom } b \)
  - \( a \text{ dom } b \) and \( a \neq b \)
- \( a \text{ idom } b \)
  - \( a \) is “closest” dominator of \( b \)
- \( a \text{ pdom } b \)
  - every path from \( a \) to the exit block includes \( b \)
- Dominator trees
- Dominance frontier

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 \( \Phi() \) neccessary

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 ← \( \Phi(v,v,\ldots) \)” 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?

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 i ← 1
 j ← 1
 k ← 0

2 k < 100?

3 j < 20?
 4 return j

5 j ← i
 6 j ← k
 7 k ← k + 1
 8 k ← k + 2
```

D-tree

```
1 i ← 1
 j ← 1
 k ← 0

2 k < 100?

3 j < 20?
 4 return j

5 j ← i
 6 j ← k
 7 k ← k + 1
 8 k ← k + 2
```

DFs

Compute Dominance Frontier

```
1 i ← 1
 j ← 1
 k ← 0

2 k < 100?

3 j < 20?
 4 return j

5 j ← i
 6 j ← k
 7 k ← k + 1
 8 k ← k + 2
```

DFs
\textbf{Insert }\Phi()\textbf{ }

1. \(i \leftarrow 1\) 
2. \(j \leftarrow 1\) 
3. \(k \leftarrow 0\)

2. \(j \leftarrow \Phi(j,j)\) 
3. \(k \leftarrow k + 1\) 
4. \(j \leftarrow k + 2\)

3. \(j \leftarrow \Phi(j,j)\) 
4. \(j \leftarrow \Phi(j,j)\) 
5. \(j \leftarrow \Phi(j,j)\) 
6. \(j \leftarrow \Phi(j,j)\) 
7. \(j \leftarrow \Phi(j,j)\)

\textbf{DFs}

var \(i\): \(W=\{1\}\) 
var \(j\): \(W=\{1,5,6\}\) 
\(\text{DF}(1), \text{DF}(5)\)
Insert $\Phi()$

1. \(i \leftarrow 1\)
2. \(j \leftarrow 1\)
3. \(k \leftarrow 0\)

\[\begin{align*}
\text{j} & \leftarrow \Phi(j, j) \\
\text{k} & \leftarrow \Phi(k, k) \\
k & \leftarrow 0
\end{align*}\]

\[\begin{align*}
j & \leftarrow \Phi(j, j) \\
\text{k} & \leftarrow \Phi(k, k)
\end{align*}\]

\[\text{j} \leftarrow \Phi(j, j) \\
\text{k} \leftarrow \Phi(k, k)\]

\[\text{j} < 20?\]

\[\text{return} \ j\]

\[\begin{align*}
j & \leftarrow i \\
k & \leftarrow k + 1
\end{align*}\]

\[\begin{align*}
j & \leftarrow k \\
k & \leftarrow k + 2
\end{align*}\]

DFs

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

Rename Vars

1. \(i_1 \leftarrow 1\)
2. \(j_1 \leftarrow 1\)
3. \(k \leftarrow 0\)

\[\begin{align*}
j_2 & \leftarrow \Phi(j_4, j_1) \\
k & \leftarrow \Phi(k, k) \\
k & \leftarrow 0
\end{align*}\]

\[\begin{align*}
j & \leftarrow \Phi(j, j) \\
k & \leftarrow \Phi(k, k)
\end{align*}\]

\[\text{j} < 20?\]

\[\text{return} \ j_2\]

\[\begin{align*}
j_3 & \leftarrow i_1 \\
k & \leftarrow k + 1 \\
k & \leftarrow k + 2
\end{align*}\]

\[\begin{align*}
j & \leftarrow k \\
k & \leftarrow k + 2
\end{align*}\]

\[\text{j} \leftarrow \Phi(j_3, j_4) \\
k \leftarrow \Phi(k, k)\]

\[\text{j} < 20?\]

\[\text{return} \ j\]

\[\begin{align*}
j_3 & \leftarrow i_1 \\
k & \leftarrow k + 1 \\
k & \leftarrow k + 2
\end{align*}\]

\[\begin{align*}
j & \leftarrow k \\
k & \leftarrow k + 2
\end{align*}\]

\[\text{j} \leftarrow \Phi(j_3, j_4) \\
k \leftarrow \Phi(k, k)\]
** Rename Vars **

1. \( i_1 \leftarrow 1 \)
   \( j_1 \leftarrow 1 \)
   \( k_1 \leftarrow 0 \)

2. \( j_2 \leftarrow \Phi(j_4,j_1) \)
   \( k_2 \leftarrow \Phi(k_4,k_1) \)

3. \( j_2 < 20? \)
   return \( j_2 \)

4. \( j_3 \leftarrow i_1 \)
   \( k_3 \leftarrow k_2 + 1 \)

5. \( j_5 \leftarrow k_2 \)
   \( k_5 \leftarrow k_2 + 2 \)

6. \( j_4 \leftarrow \Phi(j_3,j_5) \)
   \( k_4 \leftarrow \Phi(k_3,k_5) \)

7. ** 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}
        delete S
    }
}

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

** Example DCE **

B0
\( i \leftarrow 0 \)
\( j \leftarrow 0 \)

B1
\( i \leftarrow i*2 \)
\( j \leftarrow j+1 \)
\( j < 10? \)

B2
return \( j \)

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

B1
\( j_1 \leftarrow \Phi(j_0,j_2) \)
\( i_1 \leftarrow \Phi(i_0,i_2) \)
\( i_2 \leftarrow i_1*2 \)
\( j_2 \leftarrow j_1+1 \)
\( j_2 < 10? \)

B2
return \( j_2 \)

Standard DCE leaves Zombies!
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 alive, insert S.defs into W

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

Example DCE

Problem!

Fixing ADCE

• If S is live, then
  If T determines if S can execute, T should be live
Fixing DCE

• If $S$ is live, then
  If $T$ determines if $S$ can execute, $T$ should be live

\[
\begin{array}{c}
\text{Live} \\
\text{Live} \\
\text{Live} \\
\end{array}
\]

Control Dependence

$Y$ is control-dependent on $X$ if
  • $X$ branches to $u$ and $v$
  • $\exists$ a path $u \rightarrow \text{exit}$ which does not go through $Y$
  • $\forall$ paths $v \rightarrow \text{exit}$ 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 $u \rightarrow \text{exit}$ which does not go through $Y$
  • $\forall$ paths $v \rightarrow \text{exit}$ 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 $y \in G'$ (post-DF of $G$)
• Add $(x, y) \in G$ to CDG if $x \in DF_G(y)$
Aggressive Dead Code Elimination

Assume a stmt is dead until proven otherwise.

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

**CCP Example**

1  i ← 1
   j ← 1
   k ← 0

2  k < 100?

3  j < 20?

4  return j

5  j ← i
   k ← k + 1

6  j ← k
   k ← k + 2

7  

• 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 ← Φ(k1, 0)
  k2 < 100?

k2 < k2+1  return 1

**Whew!**

• SSA: 1 assignment per variable. Defs dom uses
• Minimal SSA, Phi-functions, variable relabeling
• Dominators, dominator trees, dominance frontier
• CCP
• ADCE
• Control dependence