Lecture 14
SSA-Style Optimizations

(Slides courtesy of Seth Goldstein.)

Review: Minimal SSA
- Each assignment generates a fresh variable.
- At each join point insert \( \Phi \) functions for all variables with multiple outstanding defs.

\[
\begin{align*}
y & \leftarrow x \\
y & \leftarrow 2 \\
z & \leftarrow y + x
\end{align*}
\]

Review: Dominance Frontier and Path Convergence

Constant Propagation
- If "\( v \leftarrow c \)" replace all uses of \( v \) with \( c \)
- If "\( v \leftarrow \Phi(c,c,c) \)" replace all uses of \( v \) with \( c \)

\[
W \leftarrow \text{list of all defns}
\]

while !W.isEmpty {
  Stmt S <- W.removeOne
  if S has form "\( v \leftarrow \Phi(c,\ldots,c) \)"
    replace S with \( V \leftarrow c \)
  if S has form "\( v \leftarrow c \)"
    delete S
  foreach stmt U that uses v,
    replace \( v \) with \( c \) in U
  W.add(U)
}

(Slides courtesy of Seth Goldstein.)
Other Optimizations with SSA

- **Copy Propagation**
  - delete "x \leftarrow \Phi(y,y,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

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

\[ j_2 \leftarrow \Phi(j_4, 1) \]
\[ k_2 \leftarrow \Phi(k_4, 1) \]
\[ k_2 < 100? \]
\[ j_2 < 20? \]
\[ \text{return } j_2 \]

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

\[ j_4 \leftarrow \Phi(i, j_3) \]
\[ k_4 \leftarrow \Phi(k_3, k_5) \]

But, so what?

Conditional Constant Propagation

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

\[ j_2 \leftarrow \Phi(j_4, 1) \]
\[ k_2 \leftarrow \Phi(k_4, 1) \]
\[ k_2 < 100? \]
\[ j_2 < 20? \]
\[ \text{return } j_2 \]

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

\[ j_4 \leftarrow \Phi(i, j_3) \]
\[ k_4 \leftarrow \Phi(k_3, k_5) \]

Conditional Constant Propagation Algorithm

Keeps 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)

Lattice for representing variables:

\[
\begin{array}{c}
\text{T} & \text{not executed} \\
\text{1} & \text{we have seen evidence that the variable has been assigned a constant with the value} \\
\text{2} & \text{we have seen evidence that the variable can hold different values at different times} \\
\end{array}
\]
**Conditional Constant Propagation**

1. \( i_1 \leftarrow 1 \)
2. \( j_1 \leftarrow 1 \)
3. \( k_1 \leftarrow 0 \)
4. \( j_2 \leftarrow \Phi(j_1,1) \)
5. \( k_2 \leftarrow \Phi(k_1,0) \)
6. \( k_2 < 100? \)
7. \( j_2 < 20? \)
8. \( \text{return } j_2 \)
9. \( j_3 \leftarrow 1 \)
10. \( k_3 \leftarrow k_2 + 1 \)
11. \( j_4 \leftarrow \Phi(1,j_3) \)
12. \( k_4 \leftarrow \Phi(k_3,k_5) \)
13. \( j_5 \leftarrow k_2 \)
14. \( k_5 \leftarrow k_2 + 2 \)

**Dead Code Elimination**

```java
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.users| == 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: \( i \leftarrow 0 \)
- B1: \( i \leftarrow i*2 \)
- B2: \( \text{return } j \)

**Standard DCE leaves Zombies!**
Aggressive Dead Code Elimination
Assume a statement 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

Y is control-dependent on X if
- X branches to u and v
- 3 a path u→exit which does not go through Y
- Y paths v→exit go through Y
i.e. X can determine whether or not Y is executed.
Aggressive Dead Code Elimination

Assume a statement 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 into W
}

Example DCE

B0  i0<0
j0<0

B1  j1 <- S(1, j0)
    i1 <- S(1, i0)
    i2 <- i1 * 2
    j2 <- j1 + 1
    j2 < 10?

B2 return j2

B1  j1 <- S(1, j0)
    i1 <- S(1, i0)
    i2 <- i1 * 2
    j2 <- j1 + 1
    j2 < 10?

B2 return j2

Conditional Constant Propagation

(Recall from earlier.)

• Does block 6 ever execute?
• Simple CP can't tell
• Conditional CP can tell:
  • Assumes blocks don't execute until proven otherwise
  • Assumes values are constants until proven otherwise

1  i1 <- 1
j1 <- 1
k0 <- 0

2  j2 <- S(1, j0)
    k2 <- S(1, k0)
    k2 < 100?

3  j2 < 100?
4  return j2

5  j3 <- 1
    k3 <- k2 + 1
6  j3 <- k3
    k3 <- k2 + 2

7  j4 <- S(1, j0)
    k4 <- S(1, k0)
Applying Dead Code Elimination to the Result of CCP

After CCP

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

After DCE

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

\[ k_3 < k_2 + 1 \]

return 1

Finding the Control Dependence Graph

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

i.e. X can determine whether or not Y is executed.

Dominance Frontier and Path Convergence

Finding the Control Dependence Graph

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

i.e. 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 \( \text{DFG}'(y) \) for all \( y \in G' \) (post-DF of \( G \))
- Add \((x, y)\) to \( \text{CDG} \) if \( x \in \text{DFG}'(y) \)

CDG of example

\[
\begin{align*}
\text{entry:} & \quad \{\}\ \\
2: & \quad \{\text{entry}\} \\
1: & \quad \{1, \text{entry}\} \\
0: & \quad \{\text{entry}\} \\
\text{exit:} & \quad \{\}\ \\
\end{align*}
\]