Static Single Assignment Form
Announcements

- HW3 is out
  - Standard homework: XTA
    - Submitty page is up
  - Alternative homework: Points-to over Pcode
    - No Submitty page
  - Please choose one and let me know by the end of week

- HW2 questions?
Announcements

Office hours

- Mondays 4-5pm on Webex
- Fridays 4-5pm on Webex
- Mondays and Thursdays 2pm - in SAGE 3713
Outline of Today’s Class

- SSA, introduction and motivation

Constructing SSA
  - Step 1: Place \textit{phi} functions
  - Step 2: Renaming variables
Static Single Assignment (SSA)

- SSA form transforms the 3-address code of a program so that each variable is defined exactly once (statically)
  - A sparse, efficient representation
  - Universally applied technique in compilers and static analysis
- Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”, TOPLAS 1991
Multiple definitions for a given use
- E.g., def-use chains for `local_24`: (5,6) and (9,6)
- Inconvenient
Without SSA

- Multiple definitions of a variable
  - Potentially expensive \((O(n^2))\) def-use chains

```
1. A = ...
2. A = ...
3. A = ...
4. A = ...
5. A = ...
6. A = ...
7. ...
8. ... = A
9. ... = A
10. ... = f(A)
11. ... = A
12. ... = A+5
```
With SSA

- Single definition per variable
  - O(n) # def-use chains

1. A1 = ...
2. A2 = ...
3. A3 = ...
4. A4 = ...
5. A5 = ...
6. A6 = ...
7. A7 = phi(A1,A2,A3,A4,A5,A6)
8. ... = A7
9. ... = A7
10. ... = f(A7)
11. ... = A7
12. ... = A7+5
Def-Use Reasoning

- a
- b
- c
- d
- e

1. a = 1
2. b = 2
3. c = a + b
4. d = c - a
5. d = b + d
6. d = a + b
7. e = e + 1
8. b = a + b
9. e = c - a
10. a = b * d
11. b = a - d
SSA Form

Easy case

\[ A = \text{INPUT}; \quad A_1 = \text{INPUT}; \]
\[ Y = A + 10; \quad \text{becomes} \quad Y = A_1 + 10; \]
\[ A = A + Y; \quad A_2 = A_1 + Y; \]

The 2 definitions of A become definitions of A_1 and A_2 respectively.
Phi Functions

If-then-else

\[
\begin{align*}
\text{if } (B > 0) & \quad \text{if } (B > 0) \\
A &= 5; & A_1 &= 5; \\
\text{else} & \quad \text{becomes} \quad \text{else} & A_2 &= 10; \\
A &= 10; & A_3 &= \phi(A_1, A_2)
\end{align*}
\]

Phi function introduces a (static) definition of \(A, A_3\). If control took True arm, then \(A_3\) is \(A_1\), otherwise it is \(A_2\).
Phi Functions

Loops

\[ A = 0; \quad A1 = 0; \]
\[ \text{while } (A <= n) \quad \text{while } (A2 <= n) \]
\[ A = A + 1; \quad A3 = A2 + 1; \]

If control took forward edge, \( A2 \) is \( A1 \); otherwise, i.e., control took back edge, it is \( A3 \).
Phi Functions

- But how do we know where to place those phi functions?
Constructing SSA

- **Step 1: Place phi functions**
  - Step 1a: Compute *dominance frontier* of each CFG node $N$, $DF(N)$
  - Step 1b: Place *phi* function at dominance frontier
    - If a node $N$ contains assignment to variable $A$, put a *phi* function for $A$ at $DF(N)$: $A \ldots = \phi(\ldots)$

- **Step 2: Rename variables**
  - $A$ at $N$ is indexed: e.g., $A_k$
  - $A$ at $DF(N)$ is indexed; *phi* function arguments fill up: e.g., $A_m = \phi(A_k, \ldots)$
Dominators

- Let $M$ and $N$ be nodes in CFG
  - Recall that CFG has a single entry node 1
- $M$ dominates $N$ (written $M \geq N$) if every path from entry node 1 to $N$ goes through $M$
- $M$ strictly dominates $N$ if $M \geq N$ and $M \neq N$

- If $M > N$ and there is no $K$ such that $M > K$ and $K > N$ we say that $M$ immediately dominates $N$
Dominator Tree

- Dominator relation gives rise to the **dominator tree**
  - Nodes are the CFG nodes
  - There is an edge from M to N if M is the immediate dominator of N (Note: Not the CFG edges!)
Dominator Tree

- Known algorithms
- Other applications
- Tree:

1. \( a = 1 \)
2. \( b = 2 \)
3. \( c = a + b \)
4. \( d = c - a \)
5. \( d = b + d \)
6. \( d = a + b \)
7. \( e = e + 1 \)
8. \( b = a + b \)
9. \( e = c - a \)
10. \( a = b * d \)
11. \( b = a - d \)
Where Do We Place phi’s?

- A naïve approach: place a phi function for variable A at every merge node
  - May end up with many unnecessary phi’s!

- A better approach: place a phi function for A only at those nodes that merge two (or more) distinct paths that define A
Dominance Frontier

- N is in the dominance frontier of M iff
  - M dominates a predecessor of N
  - M does not dominate N
Dominance Frontier

- DF(B₂):

- DF(B₃):

- DF(B₄):

- DF(B₅):

- DF(B₆):

1. \( a = 1 \)
2. \( b = 2 \)
3. \( c = a + b \)
4. \( d = c - a \)
5. \( d = b + d \)
6. \( d = a + b \)
7. \( e = e + 1 \)
8. \( b = a + b \)
9. \( e = c - a \)
10. \( a = b \times d \)
11. \( b = a - d \)
Constructing SSA

- **Step 1: Place phi functions**
  - **Step 1a:** Compute dominance frontier of each CFG node \( N \), \( DF(N) \)
  - **Step 1b:** Place phi function at dominance frontier
    - If a node \( N \) contains assignment to variable \( A \), put a phi function for \( A \) at \( DF(N) \): \( A \ldots = \text{phi}(\ldots) \)

- **Step 2: Rename variables**
  - \( A \) at \( N \) is indexed: e.g., \( A_k \)
  - \( A \) at \( DF(N) \) is indexed; phi function arguments fill up: e.g., \( A_m = \text{phi}(A_k, \ldots) \)
Place \( \text{phi} \) Functions

- Let \( S \) be set of nodes that define variable \( A \)
- Place \( A \ldots = \text{phi}(\ldots) \) at each node in \( \text{DF}(S) \)
  - But the \( \text{phi} \) node adds a definition of \( A \)

Thus, we have to iterate

- \( \text{DF}_1 = \text{DF}(S) \)
- \( \text{DF}_2 = \text{DF}(S \cup \text{DF}_1) \)
- \( \ldots \)
- \( \text{DF}_n = \text{DF}(S \cup \text{DF}_{n-1}) \)
Place phi Functions

- DF(B₂):
- DF(B₃):
- DF(B₄):
- DF(B₅):
- DF(B₆):

1. \( a = 1 \)
2. \( b = 2 \)
3. \( c = a + b \)
4. \( d = c - a \)
5. \( d = b + d \)
6. \( d = a + b \)
7. \( e = e + 1 \)
8. \( b = a + b \)
9. \( e = c - a \)
10. \( a = b \times d \)
11. \( b = a - d \)
Place phi Functions

- DF(B_3): \{B_2\}
- DF(B_4): \{B_3\}
- DF(B_5): \{B_2\}

- DF(S(b)):
  - DF(S(d)):
  - DF(S(e)):
Place phi Functions

- DF(B₃): \{B₂\}
- DF(B₄): \{B₃\}
- DF(B₅): \{B₂\}
- DF(S(b)): \{B₂\}
- DF(S(d)): \{B₂, B₃\}
- DF(S(e)): \{B₂, B₃\}

1. a = 1
2. b = 2

b = phi(…)
d = phi(…)
e = phi(…)
3. c = a+b
4. d = c-a
5. d = b+d
d = phi(…)
e = phi(…)

6. d = a+b
7. e = e+1
8. b = a+b
9. e = c-a
10. a = b*d
11. b = a-d
Another Round?

- DF(B₃): {B₂}
- DF(B₄): {B₃}
- DF(B₅): {B₂}
- DF⁺(S(b)): {B₂}
- DF⁺(S(d)): {B₂, B₃}
- DF⁺(S(e)): {B₂, B₃}

1. a = 1
2. b = 2
3. c = a + b
4. d = c - a
5. d = b + d
6. d = a + b
7. e = e + 1
8. b = a + b
9. e = c - a
10. a = b * d
11. b = a - d
1. $A = \ldots$
   if $X > Y$

2. \ldots

3. \ldots

4. $A = 1$

5. $A = 1$

6. \ldots

7. \ldots
Constructing SSA

- **Step 1: Place phi nodes**
  - Step 1a: Compute dominance frontier of each CFG node $N$, $DF(N)$
  - Step 1b: Place phi nodes at dominance frontier
    - If a node $N$ contains assignment to variable $A$, put a phi node for $A$ at $DF(N)$: $A... = \phi(...)$

- **Step 2: Rename variables**
  - $A$ at $N$ is indexed: e.g., $A!k$
  - $A$ at $DF(N)$ is indexed; phi function arguments fill up: e.g., $A!m = \phi(A!k,...)$
Rename Variables

- Depth-first traversal of dominator tree
  - Keep stack Stack(\(A\)) with current def per variable
  - SEARCH(\(N\)) visits dominate tree node \(N\)
    - If \(N\) defines \(A\), push new \# on top of Stack(\(A\))
    - Recursive calls SEARCH(Child) use that \#
    - Before SEARCH(\(N\)) exits, pop \# off Stack(\(A\))

- Amazingly, it works!
1. $a = 1$
2. $b = 2$

$b = \phi(, )$
$d = \phi(, )$
$e = \phi(, )$

3. $c = a + b$
4. $d = c - a$

5. $d = b + d$
6. $d = a + b$
7. $e = e + 1$
8. $b = a + b$
9. $e = c - a$
10. $a = b \times d$
11. $b = a - d$
Rename Variables

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( c = a + b \)
4. \( d = c - a \)
5. \( d = b + d \)
6. \( d = a + b \)
7. \( e = e + 1 \)
8. \( b = a + b \)
9. \( e = c - a \)
10. \( a = b \times d \)
11. \( b = a - d \)

SEARCH(\( B_1 \))
Rename Variables

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( b_2 = \phi(b_1, ?) \)
4. \( d_1 = \phi(?, ?) \)
5. \( c_1 = a_1 + b_2 \)
6. \( d = c_1 - a_1 \)
7. \( e = \phi(e_1, ?) \)
8. \( d = b + d \)
9. \( e = e + 1 \)
10. \( b = a + b \)
11. \( e = c - a \)
12. \( a = b * d \)
13. \( b = a - d \)
1. \(a_1 = 1\)
2. \(b_1 = 2\)

\[
\begin{align*}
b_2 &= \phi(b_1,?) \\
d_1 &= \phi(?,?) \\
e_1 &= \phi(?,?) \\
c_1 &= a_1 + b_2 \\
d_2 &= c_1 - a_1 \\
d_3 &= \phi(d_2,?) \\
e_2 &= \phi(e_1,?) \\
d_4 &= b_2 + d_3 \\
d &= a + b \\
e &= e + 1 \\
b &= a + b \\
e &= c - a \\
a &= b \times d \\
b &= a - d
\end{align*}
\]
Rename Variables

1. $a_1 = 1$
2. $b_1 = 2$

3. $c_1 = a_1 + b_2$
4. $d_2 = c_1 - a_1$

5. $d_4 = b_2 + d_3$
6. $d_5 = a_1 + b_2$
7. $e_3 = e_2 + 1$
8. $b = a + b$
9. $e = c - a$
10. $a = b \cdot d$
11. $b = a - d$

$\phi$ denotes a function or operator that takes one or more arguments and returns a value.
1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( c_1 = a_1 + b_2 \)
4. \( d_2 = c_1 - a_1 \)
5. \( d_4 = b_2 + d_3 \)
6. \( d_5 = a_1 + b_2 \)
7. \( e_3 = e_2 + 1 \)
8. \( b_3 = a_1 + b_2 \)
9. \( e_4 = c_1 - a_1 \)
10. \( a = b \cdot d \)
11. \( b = a - d \)
1. $a_1 = 1$
2. $b_1 = 2$
3. $c_1 = a_1 + b_2$
4. $d_2 = c_1 - a_1$
5. $d_4 = b_2 + d_3$
6. $d_5 = a_1 + b_2$
7. $e_3 = e_2 + 1$
8. $b_3 = a_1 + b_2$
9. $e_4 = c_1 - a_1$
10. $a_2 = b_3 * d_4$
11. $b_4 = a_2 - d_4$
Properties of SSA

- Every variable has a single assignment
- Assignment dominates (non-phi) uses
- Huge impact on compiler design