COMP 150-AVS
Fall 2018

Static Single Assignment Form
Motivation

• Data flow analysis needs to represent facts at every program point

• What if
  ▪ There are a lot of facts and
  ▪ There are a lot of program points?
  ▪ ⇒ potentially takes a lot of space/time

• Most likely, we’re keeping track of irrelevant facts
Example

\[
x := 3
\]

\[
y := a + b
\]

\[
z := 2 \times y
\]

\[
w := y + z
\]

\[
y := a - b
\]

\[
y := y \times 10
\]

\[
w := w + y
\]

\[
z := w + x
\]
Sparse Representation

• Instead, we’d like to use a sparse representation
  ▪ Only propagate facts about $x$ where they’re needed

• Enter *static single assignment* form
  ▪ Each variable is defined (assigned to) exactly once
  ▪ But may be used multiple times
Example: SSA

- Add SSA edges from definitions to uses
  - No intervening statements use/define variable
  - Safe to propagate only along SSA edges
What About Joins?

• Add $\Phi$ functions/nodes to model joins
  - Intuitively, takes meet of arguments
  - At code generation time, need to eliminate $\Phi$ nodes
Def-Use Chains vs. SSA

- Alternative: Don’t do renaming; instead, compute simple def-use chains (reaching definitions)
  - Propagate facts along def-use chains

- Drawback: Potentially quadratic size
case (...) of
  0:  a := 1;
  1:  a := 2;
  2:  a := 3;
end

case (...) of
  0:  b := a;
  1:  c := a;
  2:  d := a;
end

Def-Use Chains

SSA Form

Quadratic vs. (in practice) linear behavior
Computing SSA Form

• Step 1: Compute the dominance frontier

• Step 2: Use dominance frontier to place $\Phi$ nodes
  - Naive, impractical step 2: put a $\Phi$ function for every variable at the beginning of every block
  - Better: If node $X$ contains assignment to $a$, put $\Phi$ function for $a$ in dominance frontier of $X$
    - Adding $\Phi$ fn may require introducing additional $\Phi$ fn

• Step 3: Rename variables so only one definition per name
Dominators

• Let $X$ and $Y$ be nodes in the CFG
  - Assume single entry point $\text{Entry}$

• $X$ dominates $Y$ (written $X \geq Y$) if
  - $X$ appears on every path from $\text{Entry}$ to $Y$

• Write $X > Y$ when $X$ dominates $Y$ but $X \neq Y$
  - Note $\geq$ is reflexive
Dominator Tree

- The dominator relationship forms a tree
  - Edge from parent to child = parent dominates child
  - Note: edges are not same as CFG edges!

![Dominator Tree Diagram]

```plaintext
Dominator Tree
• The dominator relationship forms a tree
  ▪ Edge from parent to child = parent dominates child
  ▪ Note: edges are not same as CFG edges!
```

![Dominator Tree Diagram]
Computing Dominator Tree

• Standard algorithm due to Lengauer and Tarjan

• Runs in time $O(E\alpha(E, N))$
  - $E = \# \text{ of edges, } N = \# \text{ of nodes}$
  - where $\alpha(\cdot)$ is the inverse Ackerman’s function
  - Very slow growing; effectively constant in practice

• Algorithm quite difficult to understand
  - But lots of pseudo-code available
Why Are Dominators Useful?

- Computing static single assignment form
- Computing control dependencies
- Identify loops in CFG
  - All nodes $X$ dominated by entry node $H$, where $X$ can reach $H$, and there is exactly one back edge (head dominates tail) in loop
Where do $\Phi$ Functions Go?

- We need a $\Phi$ function at node $Z$ if
  - Two non-null CFG paths that both define $v$
  - Such that both paths start at two distinct nodes and end at $Z$
Dominance Frontiers: Illustration
Dominance Frontiers

• $Y$ is in the dominance frontier of $X$ iff
  - There exists a path from $X$ to Exit through $Y$ such that $Y$ is the first node not strictly dominated by $X$

• Equivalently:
  - $Y$ is the first node where a path from $X$ to Exit and a path from Entry to Exit (not going through $X$) meet

• Equivalently:
  - $X$ dominates a predecessor of $Y$
  - $X$ does not strictly dominate $Y$
Example

DF(1) = \{1\}
DF(2) = \{7\}
DF(3) = \{6\}
DF(4) = \{6\}
DF(5) = \{1, 7\}
DF(6) = \{7\}
DF(7) = \emptyset
Computing SSA Form

• Step 1: Compute the dominance frontier

• Step 2: Use dominance frontier to place $\Phi$ nodes

• Step 3: Rename variables so only one definition per name
Step 2: Placing $\Phi$ Functions for $v$

- Let $S$ be the set of nodes that define $v$
- Need to place $\Phi$ function in every node in $DF(S)$
  - Recall, those are all the places where the definition of $v$ in $S$ and some other definition of $v$ may meet
- But a $\Phi$ function adds another definition of $v$!
  - $v := \Phi(v, ..., v)$
- So, iterate
  - $DF_1 = DF(S)$
  - $DF_{i+1} = DF(S \cup DF_i)$
Example

Entry

1: x := 3

2

3

5: x := 4

6

7

8: x := 5

9

10

11

Exit

= need \( \Phi \) function
Step 3: Renaming Variables

• Top-down (DFS) traversal of dominator tree
  - At definition of \( v \), push new \# for \( v \) onto the stack
  - When leaving node with definition of \( v \), pop stack
  - Intuitively: Works because there’s a \( \Phi \) function, hence a new definition of \( v \), just beyond region dominated by definition

• Can be done in \( O(E+|DF|) \) time
  - Linear in size of CFG with \( \Phi \) functions
Eliminating $\Phi$ Functions

- Basic idea: $\Phi$ represents facts that the value of join may come from different paths
  - So just set along each possible path

$$w_2 := y_1 + z_1 \quad w_3 := w_1 + y_3$$

$$w_4 := \Phi(w_2, w_3)$$

$$w_2 := y_1 + z_1 \quad w_3 := w_1 + y_3$$

$$w_4 := w_2$$

$$w_4 := w_3$$
Eliminating $\Phi$ Functions in Practice

• Copies performed at $\Phi$ fns may not be useful
  - Joined value may not be used later in the program
    - (So why leave it in?)

• Use dead code elimination to kill useless $\Phi$s

• Subsequent register allocation will map the (now very large) number of variables onto the actual set of machine register