CMSC 631 – Program Analysis and Understanding

Static Single Assignment Form and Dominators
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?
  ▪ $\Rightarrow$ potentially takes a lot of space/time

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

\[
x := 3
\]

\[
y := a + b
\]

\[
z := 2 \times y
\]

\[
w := y + z
\]

\[
a > b
\]

\[
y := a - b
\]

\[
y := y \times 10
\]

\[
w := w + y
\]

\[
z := w + x
\]
Example

\[
x := 3
\]

\[
y := a + b
\]

\[
z := 2 \times y
\]

\[
w := y + z
\]

\[
a > b
\]

\[
y := a - b
\]

\[
y := y \times 10
\]

\[
w := w + y
\]

\[
z := w + x
\]
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
• Add SSA edges from definitions to uses
  ▪ No intervening statements use/define variable
  ▪ Safe to propagate only along SSA edges
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
Constant Propagation Revisited

• Initialize facts at each program point
  ▪ C(n) := top

• Add all SSA edges to the worklist

• While the worklist isn’t empty,
  ▪ Remove an edge (x, y) from the worklist
  ▪ C(y) := C(y) meet C(x)
  ▪ Add SSA edges from y if C(y) changed
Def-Use Chains vs. SSA
Def-Use Chains vs. SSA

• Alternative: Don’t do renaming; instead, compute simple def-use chains (reaching definitions)
  □ Propagate facts along def-use chains
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
Def-Use Chains vs. SSA (cont’d)

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 vs. SSA (cont’d)

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

- a := 1
- a := 2
- a := 3
- b := a
- c := a
- d := a
Def-Use Chains vs. SSA (cont’d)

```plaintext
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**

```
a := 1
\downarrow
b := a
```
```
a := 2
\downarrow
c := a
```
```
a := 3
\downarrow
d := a
```

**SSA Form**

```
a_1 := 1
\downarrow
a := \Phi(a_1, a_2, a_3)
```
```
a_2 := 2
\downarrow
```
```
a_3 := 3
\downarrow
```

```
a_4 := \Phi(a_1, a_2, a_3)
\downarrow
```
```
b_1 := a_4
```
```
c_1 := a_4
```
```
d_1 := a_4
```
Def-Use Chains vs. SSA (cont’d)

```
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**

```
a := 1
```

```
a := 2
```

```
a := 3
```

```
b := a
```

```
c := a
```

```
d := a
```

**SSA Form**

```
a_1 := 1
```

```
a_2 := 2
```

```
a_3 := 3
```

```
a_4 := \Phi(a_1, a_2, a_3)
```

```
b_1 := a_4
```

```
c_1 := a_4
```

```
d_1 := a_4
```

**Quadratic vs. (in practice) linear behavior**

Monday, September 26, 2011
Computing SSA Form
Computing SSA Form

- Step 1: Compute the dominance frontier
Computing SSA Form

• Step 1: Compute the dominance frontier

• Step 2: Use dominance frontier to place $\Phi$ nodes
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
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 $X$
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
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 $Y$
  ▪ Note $\geq$ is reflexive

• $X$ strictly dominates $Y$ (written $X > Y$) if
  ▪ $X$ dominates $Y$ but $X \neq Y$
The dominator relationship forms a tree
- Edge from parent to child = parent dominates child
- Note: edges are not same as CFG edges!
The dominator relationship forms a tree

- Edge from parent to child = parent dominates child
- Note: edges are not same as CFG edges!
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 (Else) Are Dominators Useful?

• Identify loops in CFG
  - All nodes $X$ dominated by entry node $H$, where in CFG $X$ can reach $H$ and there is exactly one back edge to $H$ among the $X$
    - A construct like “continue” must be represented as jumping to the end of the loop, to ensure one back edge

• Computing control dependences
  - Details given in the last few slides
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

- \( Y \) is in the dominance frontier of \( X \) iff
  - \( X \) dominates a predecessor of \( Y \)
  - \( X \) does not strictly dominate \( Y \)
Example

DF(1) =
DF(2) =
DF(3) =
DF(4) =
DF(5) =
DF(6) =
DF(7) =
Example

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

- $DF(X) = DF_{local}(X) \cup DF_{up}(S_X)$ where
  - $DF_{local}(X) = \{Y \in \text{succ}(X) \mid X \not> Y\}$
    - Any successor of $X$ not (strictly) dominated by $X$ is in $DF(X)$
  - $S_X = \{ Z \mid \text{idom}(Z) = X \}$
    - $\text{idom}(Z)$ is the parent of $Z$ in the dominator tree
  - $DF_{up}(S_X) = \{Y \in DF(Z) \mid Z \in S_X \text{ and } X \not> Y\}$
    - Nodes from $DF(Z)$ that are not strictly dominated by $X$ are also in $DF(X)$
Why Is This Sufficient?

• Suppose $Y \in DF(X)$
  - Then there is a $U \in \text{pred}(Y)$ such that $X \geq U$, $X \not\succ Y$
  - If $U = X$, then $Y \in DF_{\text{local}}(X) = \{Y \in \text{succ}(X) | X \not\succ Y\}$
  - Otherwise $U \neq X$
    - Then there is a node $Z$ such that $\text{idom}(Z) = X$ and $Z \geq U$
    - Possibly $Z = U$
    - Since $X \not\succ Y$, $Z \not\succ Y$, hence $Y \in DF(Z)$
  - Therefore $Y \in DF_{\text{up}}(\{Z\}) = \{Y \in DF(Z) | X \not\succ Y\}$
• Let \( \text{sdom}(X) = \{Y \mid X > Y\} \)

• In a postorder traversal on dominator tree
  - \( \text{DF}(X) = \text{succ}(X) - \text{sdom}(X) \)
    - i.e., \( \text{DF}(X) = \text{DF}_{\text{local}}(X) \)
  - For each \( Z \) such that \( \text{idom}(Z) = X \) do
    - \( \text{DF}(X) = \text{DF}(X) \cup (\text{DF}(Z) - \text{sdom}(X)) \)
    - i.e., \( \text{DF}(X) = \text{DF}(X) \cup \text{DF}_{\text{up}}(Z) \)
Equivalent Algorithm

• In a postorder traversal on dominator tree
  - $\text{DF}(X) = \text{succ}(X)$
  - For each $Z$ such that $\text{idom}(Z) = X$ do
    - $\text{DF}(X) = \text{DF}(X) \cup \text{DF}(Z)$
  - $\text{DF}(X) = \text{DF}(X) - \text{sdom}(X)$

• There’s another equivalent algorithm that runs in $O(E+|DF|)$
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

1: x := 3

5: x := 4

8: x := 5

Entry

2

3

4

6

7

9

10

11

Exit

Monday, September 26, 2011
Example

Entry

1: x := 3

2

3

5: x := 4

6

7

8: x := 5

9

10

11

Exit

1

2

3

5

6

9

7

8

10

11
Example

Entry

1: x := 3

2

3

5: x := 4

6

7

8: x := 5

10

11

Exit

= need $\Phi$ function

Monday, September 26, 2011
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 value of join may come from different paths
  - So just set along each possible path

\[
\begin{align*}
  w_2 &:= y_1 + z_1 \\
  w_3 &:= w_1 + y_3 \\
  w_4 &:= \Phi(w_2, w_3) \\
  z &:= w_4
\end{align*}
\]
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
Efficiency in Practice

• Claimed:
  
  - SSA grows linearly with size of program

<table>
<thead>
<tr>
<th>Package name</th>
<th>Statements in all procedures</th>
<th>Statements per procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EISPACK</td>
<td>7,034</td>
<td>22 Min, 89 Median, 327 Max</td>
<td>Dense matrix eigenvectors and values</td>
</tr>
<tr>
<td>FLO52</td>
<td>2,054</td>
<td>9 Min, 54 Median, 351 Max</td>
<td>Flow past an airfoil</td>
</tr>
<tr>
<td>SPICE</td>
<td>14,093</td>
<td>8 Min, 43 Median, 753 Max</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>Totals</td>
<td>23,181</td>
<td>8 Min, 55 Median, 753 Max</td>
<td>221 FORTRAN procedures</td>
</tr>
</tbody>
</table>

Efficiency in Practice (cont'd)

- Convincing?
**Arrays**

- Need to handle array accesses

- Problem: How do we know whether \( A[i], A[j], \) and \( B[k] \) are all distinct?
  - Could have \( A=B \), e.g., \( \text{foo}(\text{int } A[], \text{int } B[])\) \( \ldots \) \( \text{foo}(a,a) \)
  - Could have \( i=j \)

- History: significant research on determining array dependencies, for parallelizing compilers
Arrays (cont’d)

• One possibility: treat arrays as single variables
  ▪ Then don’t need to worry about updates to them

\[
\begin{align*}
* & := A(i); \\
A(j) & := V; \\
* & := A(k) + 2;
\end{align*}
\]

• \texttt{Update}($A, j, V$) makes a copy of $A$
  ▪ Then try to collapse unnecessary copies

• Structures are arrays with constant indexes
Pointers

• For each statement \( S \), let
  - \( \text{MustMod}(S) = \) variables always modified by \( S \)
  - \( \text{MayMod}(S) = \) variables sometimes modified by \( S \)
    - So if \( v \notin \text{MayMod}(S) \), then \( S \) must not modify \( v \)
  - \( \text{MayUse}(S) = \) variables sometimes used by \( S \)

• Then assume that statement \( S \)
  - writes to \( \text{MayMod}(S) \)
  - reads \( \text{MayUse}(S) \cup (\text{MayMod}(S) - \text{MustMod}(S)) \)

• Convincing? We’ll talk more about pointers later
As mentioned earlier, dominators can be used to compute control dependences.

- \( Y \) is *control dependent* on \( X \) if whether \( Y \) is executed depends on a test at \( X \).

- \( A, B, \) and \( C \) are control dependent on \( X \).
Postdominators and Control

• \( Y \) postdominates \( X \) if every path from \( X \) to Exit contains \( Y \)
  - I.e., if \( X \) is executed, then \( Y \) is always executed

• Then, \( Y \) is control dependent on \( X \) if
  - There is a path \( X \to Z_1 \to \cdots \to Z_n \to Y \) such that \( Y \) postdominates all \( Z_i \) and
    - \( Y \) does not postdominate \( X \)
    - I.e., there is some path from \( X \) on which \( Y \) is always executed, and there is some path on which \( Y \) is not executed
• Postdominators are just dominators on the CFG with the edges reversed

• To see what $Y$ is control dependent on, we want to find the $X$s such that in the reverse CFG

  - There is a path $X \leftarrow Z_1 \leftarrow \cdots \leftarrow Z_n \leftarrow Y$ where
    - for all $i, Y \geq Z_i$ and
    - $Y \not> X$

• I.e., we want to find $\text{DF}(Y)$ in the reverse CFG!