Static Single Assignment Form (and dominators, post-dominators, dominance frontiers...)

CS252r Spring 2011

(Almost all slides shamelessly stolen from Jeff Foster)
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

\[ x := 3 \]

\[ y := a + b \]

\[ z := 2 \times y \]

\[ w := y + z \]

\[ w := w + y \]

\[ z := w + x \]

\[ y := a - b \]

\[ y := y \times 10 \]

\[ a > b \]
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 define variable
  - Safe to propagate facts about x only along SSA edges
What About Joins?

- Add $\Phi$ functions/nodes to model joins
  - One argument for each incoming branch
  - Operationally: selects one of the arguments based on how control flow reach this node
- Dataflow analysis: 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
  • $C(y) := C(y) \cap C(x)$
  • Add to worklist SSA edges from $y$ if $C(y)$ changed
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)

Def-Use Chains

```
case (...) of
  0: a := 1;
  1: a := 2;
  2: a := 3;
end
```

SSA Form

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

Quadratic vs. (in practice) linear behavior
Conditional Constant Propagation

• So far, we assume that all branches can be taken
  • But what if some branches are never taken in practice?
    • Debugging code that can be enabled/disabled at run time
    • Macro expanded code with constants
    • Optimizations

• Idea: use constant propagation to decide which branches might be taken
  • Fits in neatly with SSA form
Nodes versus Edges

• So far, we’ve been hazy about whether data flow facts are associated with nodes or edges
  • Advantage of nodes: may be fewer of them
  • Advantage of edges: can trace differences on multiple paths to same node

• For this problem, we’ll associate facts with edges
Conditional Execution

• Keep track of whether edges may be executed
  • Some may not be because they’re on not-taken branch
  • Initially, assume no edges taken
  • At joins, don’t propagate information from not-taken in-edges

• Side comment: Notice that we always, always start with the optimistic assumption
  • We need proof that a pessimistic fact holds
  • We’re computing a greatest fixpoint
Example

\[ x_1 := 3 \]

\[ x_1 > 2 \]

\[ j_1 := 1 \]

\[ j_2 := 4 \]

\[ j_3 := \Phi(j_1, j_2) \]

\[ j_3 = 1 \]

\[ z \]
Computing SSA Form

- Step 1: Place $\Phi$ nodes
  - Naive, impractical step 1: put a $\Phi$ function for every variable at the beginning of every block
- Step 2: Rename variables so only one definition per name
Computing SSA Form

• Step 1a: Compute the dominance frontier
• Step 1b: Use dominance frontier to place $\Phi$ nodes
  • 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 2: Rename variables so only one definition per name
Dominators

• Let X and Y be nodes in the CFG
  • Assume single entry point Entry

• X dominates Y (written X≥Y) if
  • X appears on every path from Entry to Y

• Write X>Y (X strictly dominates Y) when X dominates Y but X≠Y
  • Note ≥ is reflexive
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

• An 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
“A Simple, Fast Dominance Algorithm” by Cooper, Harvey, Kennedy, 2001

- Shows $O(N^2)$ algorithm runs faster in practice than Lengauer and Tarjan
- Intuitive algorithm, phrased as dataflow equations, solved with standard (reverse-postorder) iterative dataflow
- Requires carefully engineered data structures

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Iterative Algorithm</th>
<th>Lengauer-Tarjan/Cytron et al.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dominance</td>
<td>Postdominance</td>
</tr>
<tr>
<td></td>
<td>Dom</td>
<td>DF</td>
</tr>
<tr>
<td>&gt; 400</td>
<td>3148</td>
<td>1446</td>
</tr>
<tr>
<td>201–400</td>
<td>1551</td>
<td>716</td>
</tr>
<tr>
<td>101–200</td>
<td>711</td>
<td>309</td>
</tr>
<tr>
<td>51–100</td>
<td>289</td>
<td>160</td>
</tr>
<tr>
<td>26–50</td>
<td>156</td>
<td>86</td>
</tr>
<tr>
<td>&lt;= 25</td>
<td>49</td>
<td>26</td>
</tr>
</tbody>
</table>

Average times by graph size, measured in $\frac{1}{100}$’s of a second

Table 1: Runtimes for 10,000 Runs of Our Fortran Test Suite, aggregated by Graph Size
Why Are Dominators Useful?

- Computing static single assignment form
- Computing control dependencies
- Identify (natural) 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

Dominated by X

Dominance Frontier of X
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 1a: Compute the dominance frontier
• Step 1b: Use dominance frontier to place $\Phi$ nodes
• Step 2: Rename variables so only one definition per name
Step 1b: Placing $\Phi$ Functions for $v$

- Let $S$ be the set of nodes that define $v$.
- Need to place $\Phi$ function in every node in $\text{DF}(S)$.
  - 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 $v$!
  - $v := \Phi(v, \ldots, v)$
- So, iterate
  - $\text{DF}_1 = \text{DF}(S)$
  - $\text{DF}_{i+1} = \text{DF}(S \cup \text{DF}_i)$
Example

Entry

1: x := 3

2

3

5: x := 4

6

7

8: x := 5

9

10

11

Exit

= need Φ function
Step 2: 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 \( \mathcal{O}(E+|DF|) \) 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

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

\[
\begin{align*}
    w_2 &:= y_1 + z_1 \\
    w_3 &:= w_1 + y_3 \\
    w_4 &:= w_2
\end{align*}
\]

\[
\begin{align*}
    w_2 &:= y_1 + z_1 \\
    w_3 &:= w_1 + y_3 \\
    w_4 &:= w_3
\end{align*}
\]
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
  • No correlation between ratio and program size

<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 89 327</td>
<td>Dense matrix eigenvectors and values</td>
</tr>
<tr>
<td>FLO52</td>
<td>2,054</td>
<td>9 54 351</td>
<td>Flow past an airfoil</td>
</tr>
<tr>
<td>SPICE</td>
<td>14,093</td>
<td>8 43 753</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>Totals</td>
<td>23,181</td>
<td>8 55 753</td>
<td>221 FORTRAN procedures</td>
</tr>
</tbody>
</table>

Efficiency in Practice (cont’d)

Fig. 21. Number of φ-functions versus number of program statements.

• 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., \( \text{foo(int A[], int B[]){}} \ldots \text{foo(a,a)} \)
  • Could have \( i=j \)

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

• One possibility: make arrays **immutable**
  • Then don’t need to worry about updates to them

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

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

• Convincing?
Structures

• Can treat structures as sets of variables or as an array
  • with field name like an index into array

\[
\begin{align*}
* & := A.f; \\
A.g & := V; \\
* & := A.f + A.g \\
* & := X; \quad // X = A.f \\
Y & := V; \quad // Y = A.g \\
* & := X + Y
\end{align*}
\]

• Problems?
• For each statement $S$, let
  • $\text{MustMod}(S) =$ variables always modified by $S$
  • $\text{MayMod}(S) =$ variables sometimes modified by $S$
    • So if $v \not\in \text{MayMod}(S)$, 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 in the course
Control Dependence

- **Y** is control dependent on **X** if whether **Y** is executed depends on a test at **X**

\[ \text{X} \]

\[ \text{A} \]

\[ \text{B} \]

\[ \text{C} \]

- **A**, **B**, and **C** are control dependent on **X**
Postdominators and Control Dependence

• **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 $X \rightarrow Z_1 \rightarrow \ldots \rightarrow Z_n \rightarrow Y$ 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
Dominance Frontiers, Take 2

- 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 $DF(Y)$ in the reverse CFG!