Outline

Review

Dominance Frontiers and Dominator Trees

Emitting code for SSA form

Postscript
Outline

Review

Dominance Frontiers and Dominator Trees

 Emitting code for SSA form

Postscript
A node \( n \) in the CFG dominates a node \( m \) iff:

- \( n \) is on all paths from entry to \( m \)
- by definition, a node \( n \) always dominates itself
- if \( n \neq m \), then \( n \) strictly dominates \( m \)

Computed using a dataflow-style analysis

- Each node annotated with a set of its dominators
Static Single Assignment Form

- Simple algorithm to generate SSA form
  - Introduce $\phi$ functions
  - Rename variables using Reaching Definitions
- Algorithm can generate excessive $\phi$ functions
  - TODAY: Use dominance frontiers to place the minimal number of $\phi$ functions
- Also today: Removing $\phi$ functions
  - Machines don’t support $\phi$ functions, so we must emulate them
Maximal SSA Form

- Insert $\phi$ nodes for each definition at every join node
- Rename LHS
- Rename RHS using reaching definitions
Reducing the number of \( \phi \) nodes

- Why insert \( \phi \) nodes at only join nodes?
- Can we skip inserting \( \phi \) nodes for a definition at some join node?
• The dominance frontier of a node $n$ ($\text{DF}(n)$) is a set of nodes
• A node $m \in \text{DF}(n)$ iff:
  • $n$ does not strictly dominate $m$
  • $n$ dominates $q$ where $q \in \text{pred}(m)$
• Note that dominance frontiers only contain join nodes
  • I.e. nodes with multiple predecessors
• Computing the dominance frontier of each node:
  • Iterative Data-flow analysis?
Dominance Frontiers: Direct algorithm

Direct calculation of dominance frontiers using dominator trees.
Immediate Dominators

- The *immediate* dominator of a node $m$ (IDOM$(m)$) is the node $n$:
  - such that $n$ strictly dominates $m$, and
  - $n$ does not strictly dominate $o$ where $o \in (\text{DOM}(m) - \{m\})$
  - in some sense, $n$ is the “closest” dominator in the CFG to $m$.

- By definition, ENTRY has no immediate dominator
Not Strictly Dominates

- $n$ strictly dominates $m$
  - $SDOM(n, m) = n \in DOM(m) \land n \neq m$
- $n$ does not strictly dominate $m$
  - $\neg SDOM(n, m) = n \notin DOM(m) \lor n = m$
Note that each node in the CFG can have only one immediate dominator
- Can you see why?

Create a graph $G = (V, E)$, where:
- $V$ is the set of basic blocks
- There is an edge $(n, m)$ in $E$ if $n$ is the immediate dominator of $m$ (i.e. $\text{IDOM}(m) = n$)
Example: CFG and its dominator tree
Computing the dominance frontier

- Find all join nodes in CFG, e.g. $j$
- For all nodes $n$ that dominate predecessors of $j$ (in the CFG)
  - If $n$ does not strictly dominate $j$, add $j$ to $DF(n)$
- This last step can be operationalized over all predecessors $p$ of $j$ in the CFG:
  - Start traversing the dominator tree $p$
  - If $p$ is $IDOM(j)$, stop. Otherwise add $j$ to $DF(p)$
  - Repeat by moving up the dominator tree until you reach $IDOM(j)$
Example: Non-redundant $\phi$ functions

ENTRY

\[ y_0 = x_0 + 1 \]
\[ x_1 = 2 \]

\[ y_1 = \phi(y_0, y_4) \]
\[ y_1 > 3 \]

\[ y_2 = 3 \]
\[ a = 3 \]

\[ y_3 = \phi(y_1, y_2) \]
\[ y_4 = x_1 + y_3 + 2 \]

EXIT
Placing $\phi$ functions

- For each definition $d$ in basic block $n$:
  - Place a $\phi$ function for $d$ in all nodes $m$ where $m \in DF(n)$
  - Note that each $\phi$ function is also a definition!
  - Repeat, until no more $\phi$ functions need to be inserted

- This is the minimal number of $\phi$ functions for a definition $d$ structurally
  - Can we further reduce the overall number of $\phi$ functions?

- (Figure 9.9 in Cooper and Turczon)
Other optimizations

- Dead definitions
  - Definitions that are not read (i.e. overwritten) do not need $\phi$ functions

- Two forms:
  - *Semi-pruned* SSA form, using “globals” names (those variables that are live in to a block)
  - *Pruned* SSA form, using \texttt{LIVEOUT} information
Outline

Review

Dominance Frontiers and Dominator Trees

Emitting code for SSA form

Postscript
Renaming variables

- SSA form introduced “subscripts” for each variable
- Should we drop them when generating code?

```
a_0 = x_0 + y_0
b_0 = a_0
a_1 = 17
c_0 = a_0
```
Problem with dropping subscripts

\[
a = x + y \\
b = a \\
a = 17 \\
c = a \quad \# \ \text{WRONG!}
\]
Handling subscripts

- Each definition becomes a new variable
  - I.e. Do NOT drop subscripts
- Preserves data dependences
  - Esp. important when we aggressively move code from basic blocks (e.g. very busy expressions, loop invariant code motion, etc.)
Code for $\phi$ functions

- Introduce copies along each incoming edge to a join node

```
i_2 = 1 
i_3 = a + b
i_4 = \phi(i_2, i_3)
...
```
Inserting appropriate copies along incoming edges

\begin{align*}
i_2 &= 1 \\
i_4 &= i_2 \\
i_3 &= a + b \\
i_4 &= i_3
\end{align*}
Critical edges

- Executing \( \phi \) functions by inserting copies into predecessor blocks is not always correct.
- If such a predecessor block has multiple successors, then the \( \phi \) function may execute when it shouldn’t.
  - This *may* be harmless, but not always.
- Edges connecting such predecessors to the block containing the \( \phi \) function are called *critical* edges.
Critical Edges: Example

\[ i_2 = 1 \]
\[ i_4 = i_2 \]

\[ i_3 = a + b \]
\[ i_4 = i_3 \]

...A...

...B...
Such edges need to be split by inserting a block on that edge. See the discussion in Cooper and Turczon for more details and an example.
References

- Chapter 9 of Cooper and Turczon
  - Section 9.2.1
  - Section 9.3