Announcements

- I’m back!
- Office Hours
  - 11:30–12:30, Monday and Wednesday
  - Also by appointment
- EE 324A

Static Single Assignment (SSA)

Use-def chains

- Structure which shows, for each use of a variable, which definitions could reach it
- A use may be reached by multiple definitions
- Example:
  - \(a_1 \rightarrow\)
  - \(b_1 \rightarrow\)
  - \(a_2 \rightarrow\)
- Can also build def-use chains

Calculating use-def chains

- Easy!
- Perform a reaching-definitions dataflow analysis
- At each variable use, look for definitions of that variable that reach the statement
- Construct use-def chains

Why use-def chains?

- Capture dependence information
- Use-def chains represent flow of data through program
- Can speed up optimizations
- Consider constant propagation

Sparse constant propagation

- Consider what happens when a variable gets updated during constant propagation using worklist algorithm
  - e.g., process \(x = 2\); \(x\) moves from \(\perp \rightarrow 2\)
- Put all successors of CFG node into worklist
- But what if \(x\) isn’t used in immediate successor nodes?
- Spend a lot of time propagating data and processing nodes for no reason
- Update of \(x\) only matters at last node
Using use-def chains

• Instead of propagating data along CFG edges, what if we just propagate data along use-def edges?
  • When \( x \) is updated, propagate data directly to last node, bypassing all the intermediate nodes!
• Can we run same CP algorithm?
  • Originally initialize with just start node. No uses of definitions \( \rightarrow \) Algorithm terminates early
  • Need to change initialization: Add all statements with constant RHS to initial worklist
• Upside: original CP algorithm \( O(EV^2) \); sparse algorithm \( O(N^2V) \)
• \( N \) is number of CFG nodes

Problems with u/d chains

• Can be very expensive to represent
  • CFG with \( N \) nodes can have \( N^2 \) u/d chains
• Each use can have multiple definitions associated with it
  • Can make it difficult to keep u/d information accurate as optimizations are performed and code is transformed
• Multiple defs can make optimizations harder (will see this when we return to CP)

Solution: SSA

• Static Single Assignment form
• Compact representation of use/def information
• Key feature: No variable is defined more than once (single assignment)
  • Eliminates anti/output dependences \( \rightarrow \) more optimizations possible
• SSA enables more efficient versions of optimizations
• Used in many compilers
  • e.g., LLVM

SSA for straight line code

• Each assignment to a variable is given a unique name
• All of the uses reached by that assignment are renamed to match
• Easy for straight line code:
  
  \[
  a = 4; \quad \quad a_1 = 4; \\
  \ldots = a + 5; \quad \ldots = a_1 + 5; \\
  a = 7; \quad \quad a_2 = 7; \\
  \ldots = a + 6; \quad \quad \ldots = a_2 + 6; 
  \]

SSA for control flow

• Easy when only one definition reaches a use
• What do we do for code with branches/loops?
  • Multiple definitions reach a single use

\[
\text{if } ( \ldots ) \\
\text{\quad } x = 5 \quad \quad \quad \quad \quad \quad \text{\quad } x = 7 \\
\quad y = x 
\]

\( \phi \) functions

• Dummy function that represents merging of two values
• Part of IR, but not actually emitted as code
• Inserted at merge points to combine two definitions into one

\[
\text{if } ( \ldots ) \\
\text{\quad } x_1 = 5 \quad \quad \text{\quad } x_2 = 7 \\
\quad x_3 = (x_1, x_2) \quad \quad y = x_3
\]
Loops

How would you put this loop into SSA form?

\[
\begin{align*}
x &= 1 \\
x &= x + 1
\end{align*}
\]

Converting to SSA form

Two steps to convert a program to SSA form

- \(\varphi\) function placement
  - Where do we place the \(\varphi\) functions?
  - Variable renaming
    - Rename variable definitions and uses to satisfy single-assignment property

\(\varphi\) function placement

Need to place \(\varphi\) functions wherever two definitions of a variable might merge

- Safe: place a \(\varphi\) function at every join point in CFG
- Clearly too many functions

Minimal placement

Condition:

- If \(\exists\) CFG nodes \(X, Y, Z\) such that there are paths \(X \rightarrow^* Z\) and \(Y \rightarrow^* Z\) which converge at \(Z\), and \(X\) and \(Y\) contain assignments to some variable \(v\) (in the original program), then a \(\varphi\)-node must be inserted in \(Z\) (in the new program)

Options:

- \textit{minimal}: As few \(\varphi\)-nodes as possible subject to condition
- \textit{Briggs-minimal}: Do not insert \(\varphi\)-nodes if \(V\) is not live across basic blocks
- \textit{pruned}: Remove “dead” \(\varphi\)-nodes
Example

Red nodes represent nodes which satisfy condition

Finding minimal placement

- Could trace every path from assignments to find convergence points
- This is expensive!
- Intuition: what if, for each assignment, we can find the set of nodes which could result in a convergence of definitions?
- Then only need to place \( \varphi \)-nodes there!

Detour: dominance

- Recall some terms from CFG analysis
- A node \( X \) dominates a node \( Y \) if \( X \) appears on all paths from entry to \( Y \)
  - \( X \in \text{DOM}(Y) \)
- A node \( X \) strictly dominates \( Y \) if \( X \neq Y \) and \( X \in \text{DOM}(Y) \)
- A node \( X \) is the immediate dominator of \( Y \) if \( X \) is the closest dominator of \( Y \)
- \( X = \text{IDOM}(Y) \)
- Note: \( X = \text{IDOM}(Y) \Rightarrow \forall X' \in \text{DOM}(Y), X' \in \text{DOM}(X) \)

Dominance trees

- Dominance tree induced by \( \text{IDOM} \)
- If \( X = \text{IDOM}(Y) \), \( X \) is \( Y \)'s parent in dominance tree

Dominance trees

- Dominance tree induced by \( \text{IDOM} \)
- If \( X = \text{IDOM}(Y) \), \( X \) is \( Y \)'s parent in dominance tree

Dominance frontier

- The dominance frontier of a node \( X \) is the set of nodes \( \text{DF}(X) \) such that for all \( Y \in \text{DF}(X) \), \( X \) dominates a predecessor of \( Y \), but does not strictly dominate \( Y \)
- What are the dominance frontiers for the nodes in this CFG?
Finding dominance frontiers

- Start by building dominance tree (see algorithm in Cooper et al.), then run algorithm:

  ```
  forall v
  if (number of predecessors of v ≥ 2) then
    forall predecessors p of v
    runner = p
    while (runner = IDOM(v))
      add v to DF(runner)
      runner = IDOM(runner)
  ```

- Intuition:
  - v can only be in a DF if it has 2 or more preds
  - Predecessors must have v in DF unless they dominate v (by definition).
  - Dominators of predecessors must have v in DF unless they dominate v.

Example

```
1  2
5  4
3  7
6  8
9
```

Iterated dominance frontier

\[
DF(L) = \bigcup_{X \in L} DF(X)
\]

\[
DF^+(L) = \text{limit of sequence}

DF_1 = DF(L)

DF_{i+1} = DF(L \cup DF_i)
\]

Theorem:
The set of nodes that need \( \varphi \)-nodes for a variable \( v \) is the iterated dominance frontier \( DF^+(L) \) where \( L \) is the set of nodes with assignments to \( v \).

Inserting \( \varphi \)-nodes

```
foreach variable v
  HasAlready = { }
  EverOnWorklist = { }
  Worklist = { }

foreach node X containing assignment to v
  EverOnWorklist = EverOnWorklist \cup X
  Worklist = Worklist \cup X
  while Worklist not empty
    remove X from Worklist
    foreach Y \# DF(X)
      if Y \& HasAlready
        insert \( \varphi \)-node for v at (Y)
        HasAlready = HasAlready \cup (Y)
      if Y \& EverOnWorklist
        Worklist = Worklist \cup (Y)
        EverOnWorklist = EverOnWorklist \cup (Y)
```

Converting to SSA form

- Two steps to convert a program to SSA form
  - \( \varphi \) function placement
    - Where do we place the \( \varphi \) functions?
  - Variable renaming
    - Rename variable definitions and uses to satisfy single-assignment property

Variable renaming

- At this point, \( \varphi \)-nodes are of the form \( v = \varphi(v, v) \)
- Need to rename each variable to satisfy SSA criteria
- High level idea:
  - At every \( \varphi \)-node, rename "target" of \( \varphi \), then replace all names in the block with new name
  - Change names in successor blocks to match new name, unless successor block has a \( \varphi \)-node
  - In which case, generate new name for target, and continue
Algorithms

**Stacks**: an array of stacks, one for each variable

**Counters**: an array of counters, one for each variable

Procedure `GenName` (Variable v)
- i = Counters[v]++
- if i = Top(Stacks[v]) then replace v with v_i
- Push i onto Stacks[v]

Procedure `Rename` (Block X)
- if X visited, return
- foreach phi-node P in X
  - GenName(LHS(P))
- foreach statement A in X
  - foreach Variable v in RHS(A)
    - replace v with v_i where i = Top(Stacks[v])
  - foreach Variable v in LHS(A)
    - GenName(v)
- foreach Y in successors(X)
  - Rename(Y)
- foreach phi-node or statement A in X
  - foreach v_i in LHS(A)
    - Pop(Stacks[v_i])

Example:
```
while (...) do
  read v
  w = v + w
  v = 6
  w = v + w
end
```

```
while (...) do
  v_1 = 5
  v_2 = 7
  y = x_3
end
```

```
if ( ... )
  x_1 = 5
  x_2 = 7
  x_3 = x_1
  x_2 = x_2
  y = x_3
```

```
x_1 = 5
x_2 = x_1
x_3 = x_2
y = x_3
```

```
x = 2
y = x
```

```
x = 2
y = x
```

```
x = 4
y = x
```

Pruning phi-nodes

- Can eliminate phi-nodes that occur because of variables that are not live across basic blocks
- These "block local" variables won't be used later, so do not need to be merged
- Can eliminate phi-nodes that are dead
- Merged variable isn't used again

Translating out of SSA form

- Cannot just remove phi-nodes and restore variables to original names
- Can mess up optimizations that assume variables use separate storage

```
while (...) do
  read v
  w = v + w
  v = 6
  w = v + w
end
```

```
while (...) do
  w_3 = (w_0, w_2)
  v_3 = (v_0, v_2)
  w_1 = v_1 + w_3
  v_2 = 6
  w_2 = v_2 + w_1
```

```
if ( ... )
  x_1 = 5
  x_2 = 7
  x_3 = x_1
  x_2 = x_2
  y = x_3
```

```
x_1 = 5
x_2 = x_1
x_3 = x_2
y = x_3
```

```
x = 2
y = x
```

```
x = 4
y = x
```

Returning to CP

- Use-def chains: 16
- In SSA form: place phi node in middle

```
x = ...  
  x = ...
  x = ...
  x = ...
```

```
x = 2
y = x
```

```
x = 4
y = x
```

Problems with u/d CP

- What happens if we know which way a branch will resolve?
- Do not need to propagate information from that branch
- Easy to do with CFGs
- What does this mean when we're using u/d chains?
- Can be very hard to tell which definitions to ignore!
Use/def CP with SSA

- SSA form shortens u/d chains
- Chains terminate at merge points, rather than crossing them
- Can simply ignore information merged from un-taken branches
- Much easier to account for irrelevant information
- Complexity: O(EV)