Static Single Assignment

Size of SSA graph

- potentially large (pathological cases)
- in practice appears linear in size of program

Applications for SSA

- loop invariant code motion
  (hoist invariant live ranges)
- induction variable detection
  (find cycles in SSA graph)
- constant propagation
  (sparse conditional constant)
- many more...

Constant Propagation Revisited

Sparse simple constant

- optimistic assumptions
- propagation along sparse graph (SSA)

Algorithm

1. Initialization
   (a) mark values for LHS of unknown expressions as $\perp$ (e.g., READ stmts)
   (b) mark values for LHS of constant assignment as having its constant value
   (c) everything else is marked with $\top$

2. Add to Worklist all SSA edges where definition is not $\top$

3. Iterate until worklist is empty
   (a) meet values at definition and use points
   (b) if resulting value at use is different, replace value at use
   (c) recompute the value of expression at use
   (d) if new value, update and add outgoing SSA edges from this def to Worklist
Sparse Conditional Constant

Enhanced constant propagation

- ignore edges in CFG that are not executable

Conditional definition

- when expression in conditional branch is constant, determine direction of branch
- only propagate definitions for executable code
- at join points, ignore edges not marked as executable

```plaintext
i ← 1
if (i = 1)
    then j ← 1
    else j ← 2
k ← j
```

Wegman and Zadeck, “Constant propagation with conditional branches,” ACM Transactions on Programming Languages and Systems (TOPLAS), 13(2), Apr 1991
Algorithm

1. \( \text{FlowWorkList} \leftarrow \text{edges leaving Entry in CFG} \)
   \( \text{SSAWorkList} \leftarrow \emptyset \)
   each flow graph edge is marked “non-executable”
   each value is initially set to \( T \)

2. \( \text{while } (\text{FlowWorkList} \cup \text{SSAWorkList}) \neq \emptyset \)

   (a) remove an item from one of the WorkLists
    (b) if it is a program flow graph edge from \( \text{FlowWorkList} \) marked “non-executable” then
       i. mark the edge “executable”
       ii. perform visit-\( \phi \) for all \( \phi \) functions of destination node
       iii. if this is first visit, perform visit-expression
    iv. if node has 1 succ., add it to \( \text{FlowWorkList} \)

   (c) if it is an SSA edge from \( \text{SSAWorkList} \) and its destination is a \( \phi \)-function, perform visit-\( \phi \)

   (d) if it is SSA edge from \( \text{SSAWorkList} \) and its destination is an expression, check entering program flow edges. If any is marked “executable”, perform visit-expression
Sparse Conditional Constant

\textit{visit-}φ()\textit{ }

for each operand of the φ-function, examine the corresponding edge in the flow graph

1. if the edge is “executable”, set operand’s value to value of its definition

2. if the edge is “non-executable”, set operand’s value to \(\top\)

\textit{visit-expression()}\textit{ }

1. evaluate the expression
   
   operands are calculated as the meet of incoming values on SSA edges

2. if the expression’s value changes
   
   (a) if expression is RHS of an assignment, add outbound SSA edges to \texttt{SSAWorkList}
   
   (b) if expression controls a conditional branch, add “some” outgoing flow graph edges to \texttt{FlowWorkList}.
   
   If value is \(\bot\), add all outgoing edges.
   
   If value is \(C_i\), add only edges executed as a result
Sparse Conditional Constant

Complexity

1. edges in SSA graph examined at most twice
2. edges in flow graph examined once

⇒ linear in edges of SSA graph + flow graph

Advantages of SSA

• ideal for analyses where uses and definitions are examined

• particularly profitable if program contains definitions and uses for lots of variables, and different variables are referenced in different parts of program

• may also be more suitable for algorithms obtaining more precise analysis results

Savings

• time during propagation

• space requirements for analysis
  (better match to how information is used)
Postdominators

Question

Can we generalize the concept of conditional definitions?

Postdominance

• $X$ dominates $Y$ if $X$ appears on every path from Entry to $Y$

• $X$ postdominates $Y$ if $X$ appears on every path from $Y$ to Exit

• $X$ strictly postdominates $Y$ if $X$ postdominates $Y$ and $X \neq Y$

• $Y$ is in the postdominance frontier of $X$ if
  – $X$ postdominates a successor of $Y$
  – but $X$ does not strictly postdominate $Y$
**Postdominance Frontiers Algorithm**

for each X in a bottom-up traversal of the postdominator tree

\[
\text{PDF}(X) \leftarrow \emptyset \\
\text{for each } Y \in \text{Pred}(X) \quad /* \text{local} */ \\
\quad \text{if } X \text{ does not strictly postdominate } Y \text{ then} \\
\quad \text{PDF}(X) \leftarrow \text{PDF}(X) \cup \{Y\} \\
\text{for each } Z \in \text{Children}(X) \quad /* \text{up} */ \\
\text{for each } Y \in \text{PDF}(Z) \\
\quad \text{if } X \text{ does not strictly postdominate } Y \text{ then} \\
\quad \text{PDF}(X) \leftarrow \text{PDF}(X) \cup \{Y\}
\]

Pred = immediate predecessors in the CFG
Children = descendents in the postdominator tree
Control Dependence

Properties

• Y is *control dependent* on X if
  – on a path from X to Y, Y postdominates every node on that path except X
  – Y does not strictly postdominate X

• informally, whether Y is executed may depend on branch taken at X

Theorem

Y is control dependent on X if X is in the postdominance frontier of Y

Finding postdominators and control dependences

• add CFG edge from Entry to Exit

• reverse CFG and find dominators, dominator tree, and dominance frontiers on reversed graph
Control Dependence Example

CFG

Postdominance Frontier (PDF)

CDG

PDOM
Control Dependence Example
Dead Code Elimination Revisited

WorkList approach

- mark specific instructions as critical
- follow use-def chains to mark defs
- follow control dependence to mark branches
- iterate until stable, remove unmarked code

Handling control flow

- older work marks all control flow as critical
- control dependence is more precise
- a conditional branch is critical iff a critical instruction is control dependent on it

Conditional branches

- replace non-critical branch with jump to closest live postdominator
- peephole optimization cleans up the debris (fold blocks, jump-to-jump, empty blocks)
- loops, cases, if-then-else all fall this way
Dead Code Elimination Algorithm

— initialization phase —
for each instruction $i$ do
  if $i$ is “critical” then
    $\text{live}(i) \leftarrow \text{true}$
    add $i$ to WorkList
  else $\text{live}(i) \leftarrow \text{false}$

— marking phase —
while ($\text{WorkList} \neq \emptyset$) do
  remove $i$ from WorkList
  for each use $u$ in $i$ do
    for each def $d$ of $u$ do  (1 unless $\phi$-node)
      $\text{live}(d) \leftarrow \text{true}$
      add $d$ to WorkList
  for each block $b$ on which $i$ is control dependent
    $\text{live}($last$(b)) \leftarrow \text{true}$
    add last$(b)$ to WorkList

— removal phase —
for each instruction $i$ do
  if ($\text{live}(i)$ is false & $i$ is a branch) then
    make $i$ a jump to $i$’s nearest live postdominator
  if $i$ is a jump then
    $\text{live}(i) \leftarrow \text{true}$
  if $\text{live}(i)$ is false then delete $i$