Dead Code Elimination & Constant Propagation on SSA Form

Slides mostly based on Keith Cooper's set of slides (COMP 527 class at Rice University, Fall 2002).

Used with kind permission.

Dead Code Elimination on SSA Form

Algorithm:

for each op i
   if i is critical then
      mark i
      add i to WorkList
   while (WorkList ≠ Ø)
      remove i from WorkList
      if i has form "x y op z"
         if def(y) is not marked then
            mark def(y)
            add def(y) to WorkList
         if def(z) is not marked then
            mark def(z)
            add def(z) to WorkList
      for each b n RDF(block(i))
         mark the block-ending branch in b
         add it to WorkList
      Sweep
      for each op i
         if i is not marked then
            if i is a branch then
               rewrite with a jump to i's nearest useful post-dominator
            else
               delete i

Notes:

• Eliminates some branches
• Reconects dead branches to the remaining live code
• Finds useful post-dominator by walking post-dominator tree
• Entry & exit nodes are useful

Useless Control Flow Elimination

Handling Branches

When is a branch useful?

When a useful operation depends on its existence

In the CFG, j is control dependent on i if

1. 3 a non-null path p from i to j such that p post-dominates every node on p after i
2. j is not strictly post-dominates i

j is control dependent on i if one path from i leads to j, one doesn’t
This is the reverse dominance frontier of j (RDF(j))

Algorithm uses RDF(n) to mark branches as live

Constant Propagation

Safety

• Proves that name always has known value
• Specializes code around that value
   Move some computations to compile time (→ code motion)
   Expose some unreachable blocks (→ dead code)

Opportunity

• Value + 1 signifies an opportunity

Profitability

• Compile-time evaluation is cheaper than run-time evaluation
• Branch removal may lead to block coalescing
(→ cleanup)
• If not, it still avoids the test & makes branch predictable
Sparse Constant Propagation on SSA Form

How long does this algorithm take to halt?
• Initialization is two passes
  • $|\text{op}| = 2 \times |\text{op}|$ edges
  • Value($v$) can take on $3$ values
    - $\text{TOP}$, $\text{BOT}$
    - Each use can be on Worklist twice
    - $2 \times |\text{top} list| = 4 \times |\text{top} list|$, Evaluations, Worklist pushes & pops

This is an optimistic algorithm:
• Initialize all values to $\text{TOP}$, unless they are known constants
• Every value becomes $\text{BOT}$ or $c_v$ unless its use is uninitialized

Sparse Conditional Constant Propagation

What happens when it propagates a value into a branch?
• $\text{TOP}$ ⇒ we gain no knowledge
• $\text{BOT}$ ⇒ either path can execute
• $\text{TRUE}$ or $\text{FALSE}$ ⇒ only one path can execute

Working this into the algorithm:
• Use two worklists: SSAWorkList & CFGWorkList
• SSAWorkList determines values
• CFGWorkList governs reachability
• Don’t propagate into operation until its block is reachable

Sparse Conditional Constant Propagation

There are some subtle points
• Branch conditions should not be $\text{TOP}$ when evaluated
  > Indicates an upwards-exposed use (no initial value)
  > Hard to envision compiler producing such code
• Initialize all operations to $\text{TOP}$
  > Block processing will fill in the non-top initial values
  > Unreachable paths contribute $\text{TOP}$ to $0$-functions
• Code shows CFG edges first, SSA edges
  > Can intermix them in arbitrary order
  > Correctness
  > Taking CFG edges first may help with speed (minor effect)
Sparse Conditional Constant Propagation

More subtle points

- TOP * BOT \rightarrow TOP
  - If TOP becomes 0, then 0 * BOT \rightarrow 0
  - This prevents non-monotonic behavior for the result value
  - Uses of the result value might go irrevocably to zero
  - Similar effects with any operation that has a zero

- Some values reveal simplifications, rather than constants
  - BOT * c \rightarrow BOT, but might turn into shifts & adds (\sim 2, \sim 0)
  - Removed commutativity (\textit{reassociation})
  - BOT**2 \rightarrow BOT * BOT
    (\textit{vs.} test or call to library)

- cbr TRUE \rightarrow l_1 \rightarrow l_4, becomes br \rightarrow l_4
  - Method discovers this; it must rewrite the code, too

Using SSA Form for Optimizations

In general, using SSA conversion prior to optimization leads to

- Cleaner formulations
- Better results
- Faster algorithms

We've seen two SSA-based optimization algorithms

- Dead-code elimination
- Sparse conditional constant propagation

There are many more...