

Dead Code Elimination & Constant Propagation on SSA form

Slides mostly based on Keith Cooper's set of slides (COMP 512 class at Rice University, Fall 2002).
Used with kind permission.

Using SSA - Dead Code Elimination

Dead code elimination

- Conceptually similar to mark-sweep garbage collection
 - > Mark *useful* operations
 - > Everything not marked is useless
- Need an efficient way to find and to mark useful operations
 - > Start with critical operations
 - > Work back up SSA edges to find their antecedents

Operations defined as critical

- I/O statements, linkage code (*entry & exit blocks*), return values, calls to other procedures

Algorithm will use post-dominators & reverse dominance frontiers

Using SSA – Dead Code Elimination

Mark

```
for each op i
  clear i's mark
  if i is critical then
    mark i
    add i to WorkList
while (WorkList ≠ ∅)
  remove i from WorkList
  (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 ∈ 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
    if i is not a jump then
      delete i
```

Notes:

- Eliminates some branches
- Reconnects dead branches to the remaining live code
- Find useful post-dominator by walking post-dominator tree
 - > Entry & exit nodes are useful

KT2, 2004

3

Using SSA – Dead Code 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. $\exists$ a non-null path p from i to j $\ni j$ post-dominates every node on p after i
2. j does not strictly post-dominate i

- j control dependent on $i \Rightarrow$ one path from i leads to j , one doesn't
- This is the reverse dominance frontier of j ($\text{RDF}(j)$)

Algorithm uses $\text{RDF}(n)$ to mark branches as live

KT2, 2004

4

Using SSA – Dead Code Elimination

What's left?

- Algorithm eliminates useless definitions & some useless branches
- Algorithm leaves behind empty blocks & extraneous control-flow

Algorithm from: Cytron, Ferrante, Rosen, Wegman, & Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM TOPLAS 13(4), October 1991

with a correction due to Rob Shillner

Two more issues

- Simplifying control-flow
- Eliminating unreachable blocks

Both are CFG transformations (no need for SSA)

KT2, 2004

* 5

Constant Propagation

Safety

- Proves that name always has known value
- Specializes code around that value
 - > Moves some computations to compile time ($\Rightarrow$ code motion)
 - > Exposes some unreachable blocks ($\Rightarrow$ dead code)

Opportunity

- Value $\neq \perp$ signifies an opportunity

Profitability

- Compile-time evaluation is cheaper than run-time evaluation
- Branch removal may lead to block coalescing (CLEAN)
 - > If not, it still avoids the test & makes branch predictable

KT2, 2004

6

Using SSA - Sparse Constant Propagation

$\forall$ expression, e $\left\{ \begin{array}{ll} \text{TOP} & \text{if its value is unknown} \\ c_i & \text{if its value is known} \\ \text{BOT} & \text{if its value is known to vary} \end{array} \right.$
 $\text{Value}(e) \leftarrow$
 $\text{WorkList} \leftarrow \emptyset$

$\forall$ SSA edge $s = \langle u, v \rangle$
 if $\text{Value}(u) \neq \text{TOP}$ then
 add s to WorkList

while ($\text{WorkList} \neq \emptyset$)
 remove $s = \langle u, v \rangle$ from WorkList
 let o be the operation that uses v
 if $\text{Value}(o) \neq \text{BOT}$ then
 $t \leftarrow$ result of evaluating o
 if $t \neq \text{Value}(o)$ then
 $\forall$ SSA edge $\langle o, x \rangle$
 add $\langle o, x \rangle$ to WorkList

i.e., o is " $a \leftarrow b \text{ op } v$ " or " $a \leftarrow v \text{ op } b$ "

Evaluating a Φ -node:

$\Phi(x_1, x_2, x_3, \dots, x_n)$ is
 $\text{Value}(x_1) \wedge \text{Value}(x_2) \wedge \text{Value}(x_3)$
 $\wedge \dots \wedge \text{Value}(x_n)$

Where

$\text{TOP} \wedge x = x \quad \forall x$
 $c_i \wedge c_j = c_i \quad \text{if } c_i = c_j$
 $c_i \wedge c_j = \text{BOT} \quad \text{if } c_i \neq c_j$
 $\text{BOT} \wedge x = \text{BOT} \quad \forall x$

Same result, fewer $\wedge$ operations
 Performs $\wedge$ only at Φ nodes

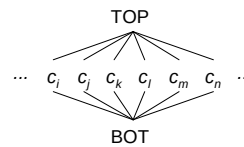
KT2, 2004

7

Using SSA - Sparse Constant Propagation

How long does this algorithm take to halt?

- Initialization is two passes
 - $|\text{ops}| + 2 \times |\text{ops}|$ edges
- $\text{Value}(x)$ can take on 3 values
 - TOP , c_i , BOT
 - Each use can be on WorkList twice
 - $2 \times |\text{args}| = 4 \times |\text{ops}|$ evaluations, WorkList pushes & pops



This is an optimistic algorithm:

- Initialize all values to TOP , unless they are known constants
- Every value becomes BOT or c_i , unless its use is uninitialized

KT2, 2004

8

Sparse Conditional Constant Propagation

Optimism

$12 \quad i_0 \leftarrow 12$
 $\text{while } (\dots)$
 $\text{TOP } i_1 \leftarrow \Phi(i_0, i_3)$
 $\text{TOP } x \leftarrow i_1 * 17$
 $\text{TOP } i \leftarrow i_1$
 $\text{TOP } i_2 \leftarrow \dots$
 $\text{TOP } i_3 \leftarrow j$

Optimistic initializations
Leads to:
 $i_1 \equiv 12 \wedge \text{TOP} \equiv 12$
 $x \equiv 12 * 17 \equiv 204$
 $j \equiv 12$
 $i_3 \equiv 12$
 $i_1 \equiv 12 \wedge 12 \equiv 12$

Optimism

- This version of the algorithm is an optimistic formulation
- Initializes values to TOP
- Prior version used $\perp$ (implicit)

In general

- Optimism helps inside loops
- Largely a matter of initial value

M.N. Wegman & F.K. Zadeck, Constant propagation with conditional branches, ACM TOPLAS, 13(2), April 1991, pages 181–210.

KT2, 2004

* 9

Sparse Conditional Constant Propagation

What happens when it propagates a value into a branch?

- TOP $\Rightarrow$ we gain no knowledge
 - BOT $\Rightarrow$ either path can execute
 - TRUE or FALSE $\Rightarrow$ only one path can execute
- But, the algorithm does not use this ...

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

KT2, 2004

10

Sparse Conditional Constant Propagation

```
SSAWorkList  $\leftarrow \emptyset$ 
CFGWorkList  $\leftarrow n_0$ 
 $\forall$  block b
  clear b's mark
   $\forall$  expression e in b
    Value(e)  $\leftarrow$  TOP
```

Initialization Step

To evaluate a branch
 if arg is BOT then
 put both targets on CFGWorklist
 else if arg is TRUE then
 put TRUE target on CFGWorkList
 else if arg is FALSE then
 put FALSE target on CFGWorkList

To evaluate a jump
 place its target on CFGWorkList

```
while ((CFGWorkList  $\cup$  SSAWorkList)  $\neq \emptyset$ )
  while(CFGWorkList  $\neq \emptyset$ )
    remove b from CFGWorkList
    mark b
    evaluate each  $\Phi$ -function in b
    evaluate each op in b, in order
  while(SSAWorkList  $\neq \emptyset$ )
    remove s = <u,v> from WorkList
    let o be the operation that contains v
    t  $\leftarrow$  result of evaluating o
    if t  $\neq$  Value(o) then
      Value(o)  $\leftarrow$  t
       $\forall$  SSA edge <o,x>
        if x is marked, then
          add <o,x> to WorkList
```

Propagation Step

KT2, 2004

11

Sparse Conditional Constant Propagation

There are some subtle points

- Branch conditions should not be TOP when evaluated
 - > Indicates an upwards-exposed use (*no initial value*)
 - > Hard to envision compiler producing such code
- Initialize all operations to TOP
 - > Block processing will fill in the non-top initial values
 - > Unreachable paths contribute TOP to Φ -functions
- Code shows CFG edges first, then SSA edges
 - > Can intermix them in arbitrary order (*correctness*)
 - > Taking CFG edges first may help with speed (*minor effect*)

KT2, 2004

12

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 irretrievably to
 - > Similar effects with any operation that has a "zero"
- Some values reveal simplifications, rather than constants
 - > $BOT * c_i \rightarrow BOT$, but might turn into shifts & adds ($c_i = 2, BOT \geq 0$)
 - > Removes commutativity (reassociation)
 - > $BOT ** 2 \rightarrow BOT * BOT$ (vs. series or call to library)
- $cbr\ TRUE \rightarrow L_1, L_2$ becomes $br \rightarrow L_1$
 - > Method discovers this; it must rewrite the code, too!

KT2, 2004

13

Sparse Conditional Constant Propagation

Unreachable Code

```

17 i ← 17
   if (i > 0) then
10 j1 ← 10
   else
20 j2 ← 20
10 j3 ← Φ(j1, j2)
170 k ← j3 * 17
  
```

With SCC
executing
blocks

Optimism

- Initialization to TOP is still important
 - Unreachable code keeps TOP
 - $\wedge$ with TOP has desired result
- Cannot get this any other way
- DEAD code cannot test ($i > 0$)
 - DEAD marks j_2 as useful

In general, combining two optimizations can lead to answers that cannot be produced by any combination of running them separately.

This algorithm is one example of that general principle.

Combining register allocation & instruction scheduling is another ...

KT2, 2004

*14

Using SSA Form for Optimizations

In general, using SSA conversion leads to

- Cleaner formulations
- Better results
- Faster algorithms

We've seen two SSA-based algorithms.

- Dead-code elimination
- Sparse conditional constant propagation

We'll see more...