CS412/413

Introduction to Compilers
Radu Rugina

Lecture 28: Control Flow Analysis
09 Apr 04

Problem 4: Constant Folding

- Compute constant variables at each program point
- Constant variable = variable having a constant value on all program executions
- Dataflow information: sets of constant values
- Example: {x=2, y=3} at program point p
- Is a forward analysis
- Let V = set of all variables in the program, nvar = |V|
- Let N = set of integer constants
- Use a lattice over the set V x N
- Construct the lattice starting from a lattice for N
- Problem: (N, ≤) is not a complete lattice!
  … why?

Constant Folding Lattice

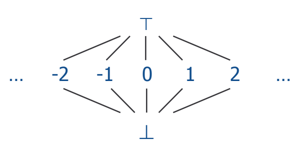
- Second try: lattice (N∪{⊤,⊥}, ≤)
  - Where ⊥≤n, for all n∈N
  - And n≤⊤, for all n∈N
  - Is complete!
- Meaning:
  - v=⊤: don't know if v is constant
  - v=⊥: v is not constant

Constant Folding Lattice

- Second try: lattice (N∪{⊤,⊥}, ≤)
  - Where ⊥≤n, for all n∈N
  - And n≤⊤, for all n∈N
  - Is complete!
- Problem:
  - Is incorrect for constant folding
  - Meet of two constants c≠d is min(c,d)
  - Meet of different constants should be ⊥
- Another problem: has infinite height …

Constant Folding Lattice

- Solution: flat lattice L = (N∪{⊤,⊥}, ⊑)
  - Where ⊥ ⊑ n, for all n∈N
  - And n ⊑ ⊤, for all n∈N
  - And distinct integer constants are not comparable



- Note: meet of any two distinct numbers is ⊥!

Constant Folding Lattice

- Denote N*=N∪{⊤,⊥}
- Use flat lattice L=(N*, ⊑)
- Constant folding lattice: L'=(V → N*, $\sqsubseteq_C$)
- Where partial order on V → N* is defined as:

  X $\sqsubseteq_C$ Y iff for each variable v: X(v) ⊑ Y(v)

- Can represent a function in V → N* as a set of assignments: { {v1=c1}, {v2=c2}, …, {vn=cn} }

CF: Transfer Functions

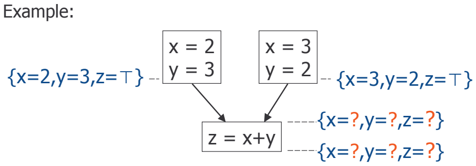
- Transfer function for instruction I:
  $F_I(X) = (X - kill[I]) \cup gen[I]$
  where:
  kill[I] = constants "killed" by I
  gen[I] = constants "generated" by I
- $X[v] = c \in N^*$ if $\{v=c\} \in X$
- If I is v = c (constant): gen[I] = {v=c} kill[I] = {v} x N*
- If I is v = u+w: gen[I] = {v=e} kill[I] = {v} x N*
  where e = X[u] + X[w], if X[u] and X[w] are not $\top$, $\bot$
  e = $\bot$, if X[u] = $\bot$ or X[w] = $\bot$
  e = $\top$, if X[u] = $\top$ and X[w] = $\top$

CF: Transfer Functions

- Transfer function for instruction I:
  $F_I(X) = (X - kill[I]) \cup gen[I]$
- Here gen[I] is not constant, it depends on X
- However transfer functions are monotonic (easy to prove)
- … but are transfer functions distributive?

CF: Distributivity

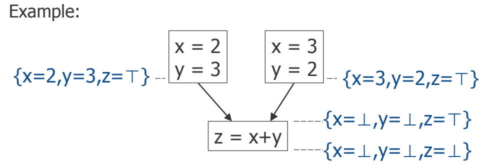
- Example:



- At join point, apply meet operator
- Then use transfer function for z=x+y

CF: Distributivity

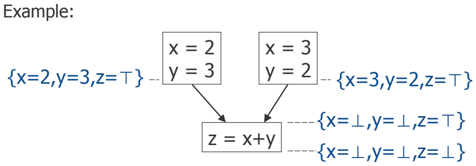
- Example:



- Dataflow result (MFP) at the end: $\{x=\bot,y=\bot,z=\bot\}$
- MOP solution at the end: $\{x=\bot,y=\bot,z=5\}$!

CF: Distributivity

- Example:



- Reason for MOP $\neq$ MFP:
  transfer function F of z=x+y is not distributive!
  $F(X1 \sqcap X2) \neq F(X1) \sqcap F(X2)$
  where X1 = $\{x=2,y=3,z=\top\}$ and X2 = $\{x=3,y=2,z=\top\}$

Classification of Analyses

- Forward analyses: information flows from
  - CFG entry block to CFG exit block
  - Input of each block to its output
  - Output of each block to input of its successor blocks
  - Examples: available expressions, reaching definitions, constant folding
- Backward analyses: information flows from
  - CFG exit block to entry block
  - Output of each block to its input
  - Input of each block to output of its predecessor blocks
  - Example: live variable analysis

Another Classification

- "may" analyses:
  - information describes a property that MAY hold in SOME executions of the program
  - Usually: $\sqcap = \cup$, $\top = \emptyset$
  - Hence, initialize info to empty sets
  - Examples: live variable analysis, reaching definitions
- "must" analyses:
  - information describes a property that MUST hold in ALL executions of the program
  - Usually: $\sqcap = \cap$, $\top = S$
  - Hence, initialize info to the whole set
  - Examples: available expressions

Next

- Control flow analysis
  - Detect loops in control flow graphs
  - Dominators
- Loop optimizations
  - Code motion
  - Strength reduction for induction variables
  - Induction variable elimination

Program Loops

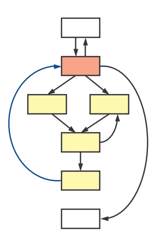
- Loop = a computation repeatedly executed until a terminating condition is reached
- High-level loop constructs:
  - While loop: while(E) S
  - Do-while loop: do S while(E)
  - For loop: for(i=1, i<=u, i+=c) S
- Why are loops important:
  - Most of the execution time is spent in loops
  - Typically: 90/10 rule, 10% code is a loop
- Therefore, loops are important targets of optimizations

Detecting Loops

- Need to identify loops in the program
  - Easy to detect loops in high-level constructs
  - Difficult to detect loops in low-level code or in general control-flow graphs
- Examples where loop detection is difficult:
  - Languages with unstructured "goto" constructs: structure of high-level loop constructs may be destroyed
  - Optimizing Java bytecodes (without high-level source program): only low-level code is available

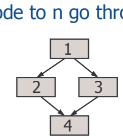
Control-Flow Analysis

- Goal: identify loops in the control flow graph
- A loop in the CFG:
  - Is a set of CFG nodes (basic blocks)
  - Has a loop header such that control to all nodes in the loop always goes through the header
  - Has a back edge from one of its nodes to the header



Dominators

- Use concept of dominators to identify loops:
  "CFG node d dominates CFG node n if all the paths from entry node to n go through d"



1 dominates 2, 3, 4
2 doesn't dominate 4
3 doesn't dominate 4

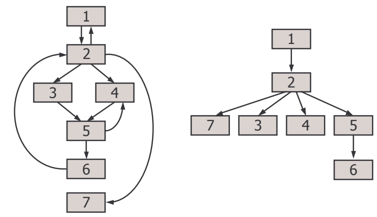
- Intuition:
  - Header of a loop dominates all nodes in loop body
  - Back edges = edges whose heads dominate their tails
  - Loop identification = back edge identification

Immediate Dominators

- Properties:
  1. CFG entry node n_0 in dominates all CFG nodes
  2. If d1 and d2 dominate n, then either
  - d1 dominates d2, or
  - d2 dominates d1
- Immediate dominator idom(n) of node n:
  - idom(n) $\neq$ n
  - idom(n) dominates n
  - If m dominates n, then m dominates idom(n)
- Immediate dominator idom(n) exists and is unique because of properties 1 and 2

Dominator Tree

- Build a dominator tree as follows:
  - Root is CFG entry node n_0
  - m is child of node n iff n=idom(m)
- Example:



Computing Dominators

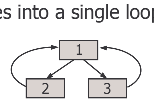
- Formulate problem as a system of constraints:
  - dom(n) is set of nodes who dominate n
  - $dom(n_0) = \{n_0\}$
  - $dom(n) = \cap \{ dom(m) \mid m \in pred(n) \}$
- Can also formulate problem in the dataflow framework
  - What is the dataflow information?
  - What is the lattice?
  - What are the transfer functions?
  - Use dataflow analysis to compute dominators

Natural Loops

- Back edge: edge $n \rightarrow h$ such that h dominates n
- Natural loop of a back edge $n \rightarrow h$:
  - h is loop header
  - Loop nodes is set of all nodes that can reach n without going through h
- Algorithm to identify natural loops in CFG:
  - Compute dominator relation
  - Identify back edges
  - Compute the loop for each back edge

Disjoint and Nested Loops

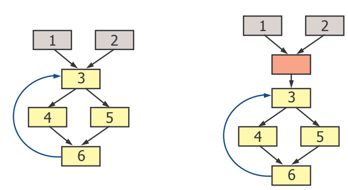
- Property: for any two natural loops in the flow graph, one of the following is true:
  1. They are disjoint
  2. They are nested
  3. They have the same header
- Eliminate alternative 3: if two loops have the same header and none is nested in the other, combine all nodes into a single loop



Two loops: {1,2} and {1,3}
Combine into one loop: {1,2,3}

Loop Preheader

- Several optimizations add code before header
- Insert a new basic block (called preheader) in the CFG to hold this code



Loop optimizations

- Now we know the loops in the program
- Next: optimize loops
  - Loop invariant code motion
  - Strength reduction of induction variables
  - Induction variable elimination

Loop Invariant Code Motion

- Idea: if a computation produces same result in all loop iterations, move it out of the loop
- Example:
```
for (i=0; i<10; i++)
    a[i] = 10*i + x*x;
```
- Expression x*x produces the same result in each iteration; move it of the loop:
```
t = x*x;
for (i=0; i<10; i++)
    a[i] = 10*i + t;
```


Loop Invariant Computation

- An instruction a = b OP c is loop-invariant if each operand is:
  - Constant, or
  - Has all definitions outside the loop, or
  - Has exactly one definition, and that is a loop-invariant computation
- Reaching definitions analysis computes all the definitions of x and y which may reach t = x OP y

Algorithm

```
INV = ∅
Repeat
    for each instruction i ∉ INV
      if operands are constants, or
        have definitions outside the loop, or
        have exactly one definition d ∈ INV
      then INV = INV U {i}
Until no changes in INV
```


Code Motion

- Next: move loop-invariant code out of the loop
- Suppose a = b OP c is loop-invariant
- We want to hoist it out of the loop
- Code motion of a definition d: a = b OP c in pre-header is valid if:
  1. Definition d dominates all loop exits where a is live
  2. There is no other definition of a in loop
  3. All uses of a in loop can only be reached from definition d

Other Issues

- Preserve dependencies between loop-invariant instructions when hoisting code out of the loop
```
for (i=0; i<N; i++) {        x = y+z;
   x = y+z;                  t = x*x;
   a[i] = 10*i + x*x;        for(i=0; i<N; i++)
}                               a[i] = 10*i + t;
```
- Nested loops: apply loop invariant code motion algorithm multiple times
```
                                    t1 = x*x;
for (i=0; i<N; i++)                 for (i=0; i<N; i++) {
  for (j=0; j<M; j++)                 t2 = t1+ 10*i;
    a[i][j] = x*x + 10*i + 100*j;     for (j=0; j<M; j++)
                                        a[i][j] = t2 + 100*j; }
```