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
• Let \( N \) := set of integer numbers
• The lattice is a map from \( V \) to \( N \)
• Construct the lattice starting from a lattice for \( N \)

Constant Folding Lattice

- **Second try:** lattice \( (N \cup \{\top, \bot\}, \leq) \)
  - Where \( \bot \leq m \), for all \( m \in N \)
  - And \( m \leq \top \), for all \( m \in N \)
  - Is complete!
  - **Meaning:**
    - \( v = \top \): don’t know if \( v \) is constant
    - \( v = \bot \): \( v \) is not constant
  - **Problem:**
    - Is incorrect for constant folding
    - Meet of two constants \( c \land d \) is \( \min(c, d) \)
    - Meet of different constants should be \( \bot \)
  - **Another problem:** has infinite height

Constant Folding Lattice

- **Solution:** flat lattice \( L = (N \cup \{\top, \bot\}, \sqsubseteq) \)
  - Where \( \bot \sqsubseteq m \), for all \( m \in N \)
  - And \( m \sqsubseteq \top \), for all \( m \in N \)
  - And distinct integer constants are not comparable

Note: meet of any two distinct numbers is \( \bot \)

CF: Transfer Functions

- Transfer function for node \( n \):
  \[ F_n(X) = (X - \text{kill}[n]) \cup \text{gen}[n] \]
- Dataflow information \( X \) is a map from \( V \) to \( N \cup \{\top, \bot\} \)
  - Represent it as a set of pairs \( (vax \mapsto a) \)
  - Denote by \( X[vax] = a \) the value of var in this mapping

- If \( n \) is \( v = c \) (constant):
  - \( \text{gen}[n] = (v \mapsto c) \cap \text{kill}[n] = (v \mapsto \bot) \)
- If \( n \) is \( v = u \lor v \):
  - \( \text{gen}[n] = (v \mapsto u) \cap \text{kill}[n] = (v \mapsto \bot) \)
where \( e = x[a] + x[b] \), if \( x[a] \) and \( x[b] \) are not \( T, \bot \)
- \( e = \bot \), if \( x[a] = \bot \) or \( x[b] = \bot \)
- \( e = T \), if \( x[a] = T \) or \( x[b] = T \)
CF: Transfer Functions

- Transfer function for node n:
  \[ F_n(X) = (X - \text{kill}[n]) \cup \text{gen}[n] \]
- Here gen[n] is not constant, it depends on X
- Exercise: prove that transfer functions are monotonic
- However, transfer functions are not distributive

CF: Distributivity

- Example:
  \[
  \begin{align*}
  x &= 2 \\
  y &= 3 \\
  z &= x + y
  \end{align*}
  \]
- MFP and MOP yield different solutions

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 \textit{MAY} hold in \textit{SOME}
    executions of the program
  - Usually: \( \cap = \cup \), \( T = \emptyset \)
  - Hence, initialize info to empty sets
  - Examples: live variable analysis, reaching definitions
- "must" analyses:
  - Information describes a property that \textit{MUST} hold in \textit{ALL}
    executions of the program
  - Usually: \( \cap = \cap \), \( T = S \)
  - 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: \textbf{while}(E) S
  - Do-while loop: \textbf{do} S \textbf{while}(E)
  - For loop: \textbf{for}(i=1, i<\text{n}, 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

- Examples:
  - 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 \( n_e \) dominates all CFG nodes
  2. If \( d_1 \) and \( d_2 \) dominate \( n \), then either
     - \( d_1 \) dominates \( d_2 \) or
     - \( d_2 \) dominates \( d_1 \)

- Immediate dominator \( \text{idom}(n) \) of node \( n \):
  - \( \text{idom}(n) \neq n \)
  - \( \text{idom}(n) \) dominates \( n \)
  - If \( m \) dominates \( n \), then \( m \) dominates \( \text{idom}(n) \)

- Immediate dominator \( \text{idom}(n) \) exists and is unique because of properties 1 and 2

Dominator Tree

- Build a dominator tree as follows:
  - Root is CFG entry node \( n_e \)
  - \( m \) is child of node \( n \) iff \( n = \text{idom}(m) \)

- Example:

Computing Dominators

- Formulate problem as a system of constraints:
  - \( \text{dom}(n) \) is set of nodes that dominate \( n \)
  - \( \text{dom}(n) = \{ n_{o} \} \)
  - \( \text{dom}(n) = ( \cap \{ \text{dom}(p) \mid p \in \text{pred}(n) \} ) \cup \{ n \} \)

- 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→h such that h dominates n
- Natural loop of a back edge n→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

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

- 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:
  \[ t = x^2; \]
  for (i=0; i<10; i++)
  a[i] = 10*i + t;

Loop Invariant Computation

- An instruction \( a = b \text{ OP } c \) 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 \( t = x \text{ OP } y \)
Algorithm

INV = ∅
Repeat
  for each instruction $\notin$ INV
  if operands are constants, or
  have definitions outside the loop, or
  have exactly one definition $d \in$ INV
    then
      INV = INV ∪ {i}
  Until no changes in INV

Other Issues

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