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 \( V \times N \)
  - Construct the lattice starting from a lattice for \( N \)
- Problem: \( (N, \leq) \) is not a complete lattice!
  ... why?

Constant Folding Lattice

- Second try: lattice \( (\mathbb{N} \cup \{T, \bot\}, \leq) \)
  - Where \( \bot \leq n \), for all \( n \in \mathbb{N} \)
  - And \( n \leq T \), for all \( n \in \mathbb{N} \)
  - Is complete!
- Meaning:
  - \( v = T \): don’t know if \( v \) is constant
  - \( v = \bot \): \( v \) is not constant
- Solution: flat lattice \( L = (\mathbb{N} \cup \{T, \bot\}, \leq) \)
  - Where \( \bot \leq n \), for all \( n \in \mathbb{N} \)
  - And \( n \leq T \), for all \( n \in \mathbb{N} \)
  - And distinct integer constants are not comparable

- Note: meet of any two distinct numbers is \( \bot \)!
**CF: Transfer Functions**

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

**Example:**
\[ \{x=2,y=3,z=\top\} \]
\[ \text{At join point, apply meet operator} \]
\[ \text{Then use transfer function for } z=x+y \]

**CF: Distributivity**

- Example:
  \[ \{x=2,y=3,z=\top\} \]
  \[ z = x+y \]

**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: $\forall x \in U, T = \emptyset$
  - 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: $\forall x \in U, T = S$
  - Hence, initialize info to the whole set
  - Examples: available expressions

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<=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 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

Dominator

- 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”

- 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 \( d_1 \) 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 \( n_0 \)
  - \( m \) 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 who dominate \( n \)
  - \( \text{dom}(n) = \{ n \} \)
  - \( \text{dom}(n) = \cap \{ \text{dom}(m) | m \in \text{pred}(n) \} \)

- 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

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++)
  
  \[ a[i] = 10\cdot i + x\cdot x; \]

- Expression \( x\cdot x \) produces the same result in each iteration; move it of the loop:
  
  \[ t = x\cdot x; \]
  
  for (i=0; i<10; i++)
  
  \[ a[i] = 10\cdot i + t; \]

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

\[ \text{INV} = \emptyset \]

Repeat
  
  for each instruction \( i \in \text{INV} \)
  
  if operands are constants, or have definitions outside the loop, or have exactly one definition \( d \in \text{INV} \)
  
  then \( \text{INV} = \text{INV} \cup \{i\} \)

Until no changes in \( \text{INV} \)

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++)
  
  \{ \]
  
  \[ x = y+z; \]
  
  \[ x = y+z; \]
  
  \[ t = x\cdot x; \]
  
  \[ \text{for} (i=0; i<N; i++) \]
  
  \[ a[i] = 10\cdot i + x\cdot x; \]
  
  \[ \text{for} (i=0; i<M; j++) \]
  
  \[ t1 = x\cdot x; \]
  
  \[ \text{for} (j=0; j<M; j++) \]
  
  \[ a[i][j] = x\cdot x + 10\cdot i + 100\cdot j; \]
  
  \[ t2 = t1 + 10\cdot j; \]
  
  \[ a[i][j] = t2 + 100\cdot j; \]