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, \( \text{ivar} = |V| \)
- 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

- \text{Second try}: lattice \( (N \cup \{ \bot, \top \}, \leq) \)
  - Where \( \bot \leq n \), for all \( n \in N \)
  - And \( n \leq \top \), for all \( n \in N \)
  - Is complete!

- Meaning:
  - \( v = \top \): don't know if \( v \) is constant
  - \( v = \bot \): \( v \) is not constant

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

Constant Folding Lattice

- Denote \( N^* = N \cup \{ \top \} \)
- Use flat lattice \( L = (N^*, \sqsubseteq) \)
- Constant folding lattice: \( L^* = (V \rightarrow N^*, \sqsubseteq_C) \)
- Where partial order on \( V \rightarrow N^* \) is defined as:
  \[ X \sqsubseteq_C Y \text{ iff for each variable } v: X(v) \sqsubseteq Y(v) \]
- Can represent a function in \( V \rightarrow N^* \) as a set of assignments: \( \{ (v1=c1), (v2=c2), ..., (vn=cn) \} \)
**CF: Transfer Functions**

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

**CF: Distributivity**

- Example:
  \[ \{x = 2, y = 3, z = \top\} \rightarrow \begin{cases} x = 3, y = 2, z = \top \\quad \text{if } z = x + y \end{cases} \]
- At join point, apply meet operator
- Then use transfer function for \( z = x + y \)

**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: \( \cap \neq \cup, T \neq \emptyset \)
  - 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: \( \cap = \cap, T = S \)
  - Hence, initialize info to the whole set
  - Examples: available expressions

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

Program Loops

- Loop = a computation repeatedly executed until a terminating condition is reached

- High-level loop constructs:
  - While loop: \( \text{while}(E) \ S \)
  - Do-while loop: \( \text{do } S \text{ while}(E) \)
  - For loop: \( \text{for}(i=1, i\leq n, i+=c) \ S \)

- 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

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 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:
  - $\text{dom}(n)$ is set of nodes who dominate $n$
  - $\text{dom}(n) = \{ n \}$
  - $\text{dom}(n) = \cap \{ \text{dom}(m) \mid m \in \text{pred}(n) \}$

- 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