Forward Data Flow, Again

- Out(s) = Top for all statements s
- W := { all statements } (worklist)
- repeat
  - Take s from W
  - temp := f_s (\forall s' \in \text{pred(s)} Out(s')) \text{ (} f_s \text{ monotonic transfer fn)}
  - if (temp \neq \text{Out(s)}) {
    - Out(s) := temp
    - W := W \cup \text{succ(s)}
  }
- until W = \emptyset

Fixpoints

- We always start with Top
  - Every expression is available, no defns reach this point
  - Most optimistic assumption
  - Strongest possible hypothesis
    - = true of fewest number of states
- Revise as we encounter contradictions
  - Always move down in the lattice (with meet)
- Result: A greatest fixpoint

Lattices (P, \leq)

- Available expressions
  - P = sets of expressions
  - S_1 \cap S_2 = S_1 \cap S_2
  - Top = set of all expressions
- Reaching Definitions
  - P = set of definitions (assignment statements)
  - S_1 \cap S_2 = S_1 \cup S_2
  - Top = empty set

Forward vs. Backward

\text{Out(s) = Top for all } s
\text{W := \{ all statements \}} \text{ (worklist)}
\text{repeat}
\text{Take s from W}
\text{temp := } f_s (\forall s' \in \text{pred(s)} \text{Out(s')) \text{ (} f_s \text{ monotonic transfer fn)}
\text{if (temp \neq \text{Out(s)}) {
  - Out(s) := temp
  - W := W \cup \text{succ(s)}
}}
\text{until W = \emptyset}

Lattices (P, \leq), cont’d

- Live variables
  - P = sets of variables
  - S_1 \cap S_2 = S_1 \cup S_2
  - Top = empty set
- Very busy expressions
  - P = set of expressions
  - S_1 \cap S_2 = S_1 \cap S_2
  - Top = set of all expressions

Termination Revisited

- How many times can we apply this step:
  - \text{temp := } f_s (\forall s' \in \text{pred(s)} \text{Out(s'))
  - if (temp \neq \text{Out(s)}) {...
- Claim: \text{Out(s)} only shrinks
  - Proof: \text{Out(s)} starts out as top
    - So \text{temp} must be \leq than \text{Top} after first step
    - Assume \text{Out(s') shrinks for all predecessors } s' \text{ of } s
    - Then \forall s' \in \text{pred(s)} \text{Out(s')} shrinks
      - Since \text{f_s monotonic, } f_s (\forall s' \in \text{pred(s)} \text{Out(s')) shrinks}
Termination Revisited (cont’d)

• A descending chain in a lattice is a sequence
  - x0 ⊑ x1 ⊑ x2 ⊑ ...
  - The height of a lattice is the length of the longest descending chain in the lattice

• Then, dataflow must terminate in O(nk) time
  - n = # of statements in program
  - k = height of lattice
  - Assumes meet operation takes O(1) time

Distributive Data Flow Problems

• By monotonicity, we also have
  \[ f(x \sqcap y) \leq f(x) \sqcap f(y) \]

• A function \( f \) is distributive if
  \[ f(x \sqcap y) = f(x) \sqcap f(y) \]

Least vs. Greatest Fixpoints

• Dataflow tradition: Start with Top, use meet
  - To do this, we need a meet semilattice with top
  - Meet semilattice = meets defined for any set
  - Computes greatest fixpoint

• Denotational semantics tradition: Start with Bottom, use join
  - Computes least fixpoint

Accuracy of Data Flow Analysis

• Ideally, we would like to compute the meet over all paths (MOP) solution:
  - Let \( f_i \) be the transfer function for statement \( s \)
  - If \( p \) is a path \( \{s_1, ... , s_n\} \), let \( f_p = f_{s_1} \odot ... \odot f_{s_n} \)
  - Let \( \text{path}(s) \) be the set of paths from the entry to \( s \)
  \[ \text{MOP}(s) = \sqcap_{p \in \text{path}(s)} f_p(\top) \]

• If a dataflow problem is distributive, then solving the dataflow equations in the standard way yields the MOP solution

Benefit of Distributivity

• Joins lose no information

\[
k(h(f(\top) \sqcap g(\top))) =
k(h(f(\top)) \sqcap h(g(\top))) =
k(h(f(\top))) \sqcap h(g(\top))
\]

What Problems are Distributive?

• Analyses of how the program computes
  - Live variables
  - Available expressions
  - Reaching definitions
  - Very busy expressions

• All Gen/Kill problems are distributive
A Non-Distributive Example

- Constant propagation

- In general, analysis of what the program computes is not distributive

A Non-Distributive Example

Practical Implementation

- Data flow facts = assertions that are true or false at a program point

- Represent set of facts as bit vector
  - Fact, represented by bit i
  - Intersection = bitwise and, union = bitwise or, etc

- “Only” a constant factor speedup
  - But very useful in practice

Practical Implementation

Basic Blocks

- A basic block is a sequence of statements s.t.
  - No statement except the last in a branch
  - There are no branches to any statement in the block except the first

- In practical data flow implementations,
  - Compute Gen/Kill for each basic block
    - Compose transfer functions
  - Store only In/Out for each basic block
  - Typical basic block ~5 statements

Basic Blocks

Order Matters

- Assume forward data flow problem
  - Let $G = (V,E)$ be the CFG
  - Let $k$ be the height of the CFG

- If $G$ acyclic, visit in topological order
  - Visit head before tail of edge
  - Running time $O(|E|)$
  - No matter what size the lattice

Order Matters

Order Matters — Cycles

- If $G$ has cycles, visit in reverse postorder
  - Order from depth-first search

- Let $Q = \text{max} \# \text{back edges on cycle-free path}$
  - Nesting depth
  - Back edge is from node to ancestor on DFS tree

- Then if $\forall x.f(x) \leq x$ (sufficient, but not necessary)
  - Running time is $O((Q + 1)|E|)$
    - Note direction of req’t depends on top vs. bottom

Order Matters — Cycles

Flow-Sensitivity

- Data flow analysis is flow-sensitive
  - The order of statements is taken into account
    - I.e., we keep track of facts per program point

- Alternative: Flow-insensitive analysis
  - Analysis the same regardless of statement order
    - Standard example: types
      - `/ x : int /* x := ... */ x : int /*`
**Terminology Review**

- Must vs. May
  - (Not always followed in literature)
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Distributive vs. Non-distributive

**Another Approach: Elimination**

- Recall in practice, one transfer function per basic block
- Why not generalize this idea beyond a basic block?
  - “Collapse” larger constructs into smaller ones, combining data flow equations
  - Eventually program collapsed into a single node!
  - “Expand out” back to original constructs, rebuilding information

**Lattices of Functions**

- Let $(P, \leq)$ be a lattice
- Let $M$ be the set of monotonic functions on $P$
- Define $f \leq g$ if for all $x, f(x) \leq g(x)$
- Define the function $f \land g$ as
  - $(f \land g)(x) = f(x) \land g(x)$
- Claim: $(M, \leq)$ forms a lattice

**Elimination Methods: Conditionals**

- $f_{ite} = (f_{then} \circ f_{if}) \cap (f_{else} \circ f_{if})$
- $Out(if) = f_{if}(In(ite)))$
- $Out(then) = (f_{then} \circ f_{if})(In(ite)))$
- $Out(else) = (f_{else} \circ f_{if})(In(ite)))$

**Elimination Methods: Loops**

\[
\begin{align*}
\text{While} & \quad \Rightarrow \\
\text{Head} & \quad \Rightarrow \quad \text{Body} \quad \Rightarrow
\end{align*}
\]

\[
\begin{align*}
\ell_{\text{while}} & = \ell_{\text{head}} \cap \\
& \quad \ell_{\text{head}} \circ \ell_{\text{body}} \circ \ell_{\text{head}} \cap \\
& \quad \ell_{\text{head}} \circ \ell_{\text{body}} \circ \ell_{\text{head}} \circ \ell_{\text{body}} \circ \ell_{\text{head}} \cap \cdots
\end{align*}
\]

**Elimination Methods: Loops (cont’d)**

- Let $f^i = f \circ f \circ \ldots \circ f$ (i times)
  - $f^0 = \text{id}$
- Let
  - $g(j) = \bigcap_{i}[0, j](\ell_{\text{head}} \circ \ell_{\text{body}})^i \circ \ell_{\text{head}}$
- Need to compute limit as $j$ goes to infinity
  - Does such a thing exist?
- Observe: $g(j+1) \leq g(j)$
Height of Function Lattice

- Assume underlying lattice \((P, \leq)\) has finite height
  - What is height of lattice of monotonic functions?
  - Claim: At most |\(P|\times\text{Height}(P)\)

- Therefore, \(g(j)\) converges

Non-Reducible Flow Graphs

- Elimination methods usually only applied to reducible flow graphs
  - Ones that can be collapsed
  - Standard constructs yield only reducible flow graphs
  - Unrestricted goto can yield non-reducible graphs

Comments

- Can also do backwards elimination
  - Not quite as nice (regions are usually single entry but often not single exit)
  - For bit-vector problems, elimination efficient
    - Easy to compose functions, compute meet, etc.
  - Elimination originally seemed like it might be faster than iteration
    - Not really the case

Data Flow Analysis and Functions

- What happens at a function call?
  - Lots of proposed solutions in data flow analysis literature
  - In practice, only analyze one procedure at a time
  - Consequences
    - Call to function kills all data flow facts
    - May be able to improve depending on language, e.g., function call may not affect locals

More Terminology

- An analysis that models only a single function at a time is intraprocedural
- An analysis that takes multiple functions into account is interprocedural
- An analysis that takes the whole program into account is...guess?

- Note: global analysis means “more than one basic block,” but still within a function

Data Flow Analysis and The Heap

- Data Flow is good at analyzing local variables
  - But what about values stored in the heap?
  - Not modeled in traditional data flow

- In practice: \(\%x := e\)
  - Assume all data flow facts killed (!)
  - Or, assume write through \(x\) may affect any variable whose address has been taken

- In general, hard to analyze pointers
Data Flow Analysis and Optimization

- **Moore’s Law**: Hardware advances double computing power every 18 months.

- **Proebsting’s Law**: Compiler advances double computing power every 18 years.

- We’ll focus on other uses of data flow analysis in this class (later in the semester)