Data Flow Analysis

Recall: Data Flow Analysis

- A framework for proving facts about program
- Reasons about lots of little facts
- Little or no interaction between facts
  - Works best on properties about how program computes
- Based on all paths through program
  - including infeasible paths

Recall: Data Flow Equations

- Let $s$ be a statement
- $\text{succ}(s) = \{\text{immediate successor statements of } s\}$
- $\text{Pred}(s) = \{\text{immediate predecessor statements of } s\}$
- $\text{In}(s)$ program point just before executing $s$
- $\text{Out}(s)$ = program point just after executing $s$
- $\text{Out}(s) = \top$ for all statements $s$
- $W := \{\text{all statements}\}$ (worklist)
- Repeat
  - Take $s$ from $W$
  - $\text{temp} := f_{\text{out}}(\cap_{s' \in \text{pred}(s)} \text{Out}(s'))$ ($f_{\text{out}}$ monotonic transfer fn)
  - if ($\text{temp} \neq \text{Out}(s)$) {
    - $\text{Out}(s) := \text{temp}$
    - $W := W \setminus \text{succ}(s)$
  }$
  - until $W = \emptyset$

Forward Data Flow, Again

$\text{Gen}(s) = \text{set of facts true after/before } s$ that weren’t true before/after

$\text{Kill}(s) = \text{set of facts no longer true after/before } s$ forward/backward
What we would like to know:

Does it terminate?
Is it accurate?
How long does it take?

Data Flow Facts and lattices

Typically, data flow facts form a lattice

Example, Available expressions

![Graph showing lattice structure]

Partial Orders

• A **partial order** is a pair \((P, \leq)\) such that
  - \(\leq\) is reflexive: \(x \leq x\)
  - \(\leq\) is antisymmetric: \(x \leq y\) and \(y \leq x\) implies \(x = y\)
  - \(\leq\) is transitive: \(x \leq y\) and \(y \leq z\) implies \(x \leq z\)

Lattices

• A partial order is a lattice if \(x\) and \(w\) are defined so that
  - \(x\) is the meet or greatest lower bound operation
    - \(x \land y \leq x\) and \(x \land y \leq y\)
    - If \(z \leq x\) and \(z \leq y\) then \(z \leq x \land y\)
  - \(w\) is the join or least upper bound operation
    - \(x \lor x \geq y\) and \(y \geq x \lor y\)
    - If \(x \geq z\) and \(y \geq z\), then \(x \lor y \geq z\)
Lattices (cont.)

A finite partial order is a \textbf{lattice} if meet and join exist for every pair of elements.

A lattice has unique elements \textbf{bot} and \textbf{top} such that:

\[
\begin{align*}
\text{bot} \times \text{bot} &= \text{bot} \\
\text{top} \times \text{bot} &= \text{bot} \\
\text{bot} \times \text{top} &= \text{bot} \\
\end{align*}
\]

In a lattice:

\[
\begin{align*}
\text{bot} \land \text{top} &= \text{bot} \\
\text{top} \land \text{top} &= \text{top} \land \text{bot} \\
\text{bot} \land \text{top} &= \text{bot} \\
\end{align*}
\]

Useful Lattices

- \((2^S, \subseteq)\) forms a lattice for any set \(S\).
- \(2^S\) is the powerset of \(S\) (set of all subsets).
- If \((S, \subseteq)\) is a lattice, so is \((S, \supseteq)\).
- i.e., lattices can be flipped.
- The lattice for constant propagation.

Monotonicity

- A function \(f\) on a partial order is \textbf{monotonic} if:
  \[
  x \preceq y \text{ implies } f(x) \preceq f(y)
  \]
- Easy to check that operations to compute In and Out are monotonic.
  - \(\text{In}(s) = \cap s \subseteq \text{pred}(s) \text{ Out}(s')\)
  - \(\text{Temp} = \text{Gen}(s) \cap (\text{In}(s) - \text{Kill}(s))\)
- Putting the two together:
  - \(\text{Temp} = f_s(\cap s \subseteq \text{pred}(s) \text{ Out}(s'))\)

Termination

- We know algorithm terminates because:
  - The lattice has finite height.
  - The operations to compute In and Out are monotonic.
  - On every iteration we remove a statement from the worklist and/or move down the lattice.
### 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

### 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 \((P, \leq)\), 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

### Forward vs. Backward

**Out\((s)\) = Top for all \(s\)**
\[
W := \{ \text{all statements} \}
\]
repeat
\[
\text{Take } s \text{ from } W
\]
\[
\text{temp} := f_s(n s' e \ pred(s) \ Out(s'))
\]
\[
\text{if (temp !=} \ Out(s)) \{\]
\[
\text{Out}(s) := \text{temp}
\]
\[
W := W \setminus \text{succ}(s)
\]
\[
\}\}
\]
\[
\text{until } W = \emptyset
\]

**In\((s)\) = Top for all \(s\)**
\[
W := \{ \text{all statements} \}
\]
repeat
\[
\text{Take } s \text{ from } W
\]
\[
\text{temp} := f_s(n s' e \ succ(s) \ In(s'))
\]
\[
\text{if (temp !=} \ In(s)) \{\]
\[
\text{In}(s) := \text{temp}
\]
\[
W := W \setminus \text{pred}(s)
\]
\[
\}\}
\]
\[
\text{until } W = \emptyset
\]
Termination Revisited

How many times can we apply this step:
\[
\text{temp} := f_s(\Pi_{s' \in \text{pred}(s)} \text{Out}(s'))
\]
if (temp != Out(s)) { ... }

Claim: Out(s) only shrinks
- Proof: Out(s) starts out as top
  - So temp must be ≤ than Top after first step
- Assume Out(s') shrinks for all predecessors s' of s
- Then \( \Pi_{s' \in \text{pred}(s)} \text{Out}(s') \) shrinks
- Since \( f_s \) monotonic, \( f_s(\Pi_{s' \in \text{pred}(s)} \text{Out}(s')) \) shrinks

Termination Revisited (cont’d)

A descending chain in a lattice is a sequence
\[
\text{x_0 \sqsupseteq x_1 \sqsupseteq x_2 \sqsupseteq ...}
\]
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

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

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
\[
f(x \sqcap y) = f(x) \sqcap f(y)
\]
**Benefit of Distributivity**

Joins lose no information

\[ k(h(f(T) \cap g(T))) = k(h(f(T))) \cap k(g(T)) = k(h(f(T))) \cap k(g(T)) \]

**Accuracy of Data Flow Analysis**

Ideally, we would like to compute the meet over all paths (MOP) solution:

- Let \( f_s \) be the transfer function for statement \( s \)
- If \( p \) is a path \( \{s_1, \ldots, s_n\} \), let \( f_p = f_{s_1}; \ldots; f_{s_n} \)
- Let \( \text{path}(s) \) be the set of paths from the entry to \( s \)

\[ \text{MOP}(s) = \cap_{p \in \text{path}(s)} f_p(T) \]

If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution.

**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
Order Matters

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

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 — Cycles

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

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

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

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

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

Elimination Methods: Conditionals

Let \( f^i = f \circ f \circ \ldots \circ f \) (i times)

\[ f_{\text{ite}} = (f_{\text{then}} \circ f_{\text{if}}) \sqcap (f_{\text{else}} \circ f_{\text{if}}) \]

\[ \text{Out}(\text{if}) = f_{\text{if}}(\text{In}(\text{ite})) \]
\[ \text{Out}(\text{then}) = (f_{\text{then}} \circ f_{\text{if}})(\text{In}(\text{ite})) \]
\[ \text{Out}(\text{else}) = (f_{\text{else}} \circ f_{\text{if}})(\text{In}(\text{ite})) \]

Elimination Methods: Loops

Let \( g(j) = \sqcap_{i \in [0..j]} (f_{\text{head}} \circ f_{\text{body}})^i \circ f_{\text{head}} \)

Need to compute limit as \( j \) goes to infinity

- Does such a thing exist?

Observe: \( g(j+1) \leq g(j) \)
Forming regions: T1-T2 Reduction

Oldest and simplest

Can reduce all well-structured graphs!

only requirement for T2:
second block has single predecessor

T1: self loop

T2: two-block sequence

T1-T2 Example

Hierarchy can seem strange....

T1-T2 Reduction

Oldest and simplest

Can reduce all well-structured graphs!

But...cannot reduce irreducible graphs!
--- end up w/ “limit flow graph”

T2
(out edges from new region get merged – not shown)

T1-T2 Example

Hierarchy can seem strange....
Hierarchy can seem strange....

An alternate approach to dataflow analysis, before we iterated on basic blocks, we iterated on basic blocks. Now, each time we form a region to form a composite transfer function that summarizes the effect of that region.

\[ f_B \circ f_A(x) = f_B(f_A(x)) \]
Dataflow Analysis on the Control Tree

- After all regions are formed there is just one region for the whole proc, i.e., you get one transfer function for the whole proc.

- But what good is it to have dataflow info at the exit node?

- The rest of the story: you also build functions for distributing the results back down the control tree to each region, eventually to the leaves (basic blocks).

Details...

How to calculate $f_B \cdot f_A$?

Well, we have already done this when computing the transfer function of a block that is a sequence of instructions... but to spell it out:

$$f_A(x) = \text{Gen}_A \cup (x \cdot \text{Kill}_A)$$

$$f_B(f_A(x)) = \text{Gen}_B \cup (f_A(x) \cdot \text{Kill}_B)$$

$$= \text{Gen}_B \cup ((\text{Gen}_A \cup (x \cdot \text{Kill}_A)) \cdot \text{Kill}_B)$$

$$= \text{Gen}_B \cup (\text{Gen}_A \cdot \text{Kill}_B) \cup (x \cdot (\text{Kill}_A \cup \text{Kill}_B))$$

More Sample Calculations

$$f_R(x) = f_B(f_A(x)) \land f_A(x)$$

$$= [ (f_B \cdot f_A) \land f_A ](x)$$

$$= [ (f_B \lor I) \land f_A ](x)$$

$\land$ is the meet operator.

- gets just slightly more complicated for flow-sensitive transfer functions where $f_A_{\text{then}}$ is different than $f_A_{\text{else}}$

- distribution calculation (coming down the control tree) is obvious.

More Sample Calculations

$$y = f_R(x) = f_A(x) \land [f_A \cdot (f_B \cdot f_A)](x) \land \ldots$$

$$= [f_A \cdot (f_B \cdot f_A)\ldots](x)$$

* is Kleene (“clay-nee”) closure:

$$f^* = I \land f \land f \land f \land f \land \ldots$$

Top-down calculations:

- $\text{in}(f_A) = [(f_B \cdot f_A)\ldots](x)$

- $\text{in}(f_B) = f_A(\text{in}(f_A))$
Example closure for gen/kill

\[ f_R(x) = I \cap (\prod_{n \geq 0} f^n) \]

Suppose, \( f(x) = \text{gen} \cup (x - \text{kill}) \)  
[E.g., reachingdefs]

\[ f^2(x) = f(f(x)) = \text{gen} \cup (\text{gen} \cup (x - \text{kill})) - \text{kill} \]
\[ = \text{gen} \cup (x - \text{kill}) \]

So, \( f_R(x) = I \cup (\text{gen} \cup (x - \text{kill})) \]
\[ = x \cup \text{gen} \]

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
- But, showing new signs of life for JIT