

Data Flow Analysis

15-745

3/25/08

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
- $\text{In}(s) = \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$ **must**
- $\text{Out}(s) = \text{Gen}(s) \wedge (\text{In}(s) - \text{Kill}(s))$ **forward**
 - Note these are also called **transfer functions**

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

$\text{Kill}(s)$ = set of facts no longer true after/before s
forward/backward

Forward Data Flow, Again

$\text{Out}(s) = \text{Top}$ for all statements s

$W := \{\text{all statements}\}$ (worklist)

Repeat

Take s from W

- $\text{temp} := f_s(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s'))$ (f_s monotonic transfer fn)
- if ($\text{temp} \neq \text{Out}(s)$) {
 - $\text{Out}(s) := \text{temp}$
 - $W := W \wedge \text{succ}(s)$
- }

until $W = \emptyset$

What we would like to know:

Does it terminate?

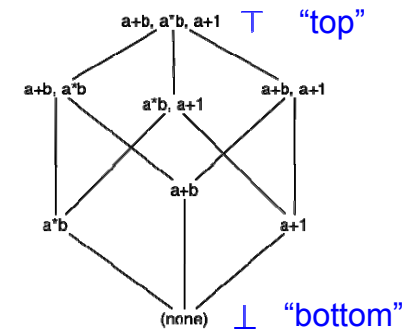
Is it accurate?

How long does it take?

Data Flow Facts and lattices

Typically, data flow facts form a lattice

Example, Available expressions



Partial Orders

- A *partial order* is a pair (P, \preceq) such that
 - \preceq is a \subseteq relation on P
 - \preceq is *reflexive*: $x \preceq x$
 - \preceq is *anti-symmetric*: $x \preceq y$ and $y \preceq x$ implies $x = y$
 - \preceq is *transitive*: $x \preceq y$ and $y \preceq z$ implies $x \preceq z$

Lattices

- A partial order is a lattice if \wedge and \vee are defined so that
 - \wedge is the *meet* or *greatest lower bound* operation
 - $x \wedge y \preceq x$ and $x \wedge y \preceq y$
 - If $z \preceq x$ and $z \preceq y$ then $z \preceq x \wedge y$
 - \vee is the *join* or *least upper bound* operation
 - $x \preceq x \vee y$ and $y \preceq x \vee y$
 - If $x \preceq z$ and $y \preceq z$, then $x \vee y \preceq z$

Lattices (cont.)

A finite partial order is a **lattice** if meet and join exist for every pair of elements

A lattice has unique elements **bot** and **top** such that

$$x \times B = B \quad x \vee B = x$$

$$x \times A = x \quad x \vee A = A$$

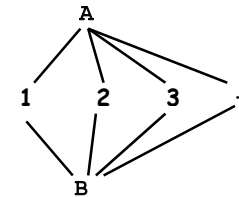
In a lattice

$$x \leq y \text{ iff } x \times y = x$$

$$x \leq y \text{ iff } x \vee y = y$$

Useful Lattices

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



Monotonicity

- A function f on a partial order is **monotonic** if

$$x \leq y \text{ implies } f(x) \leq f(y)$$

- Easy to check that operations to compute In and Out are monotonic

- $In(s) = \prod_{s' \in \text{pred}(s)} Out(s')$
- $Temp = Gen(s) \wedge (In(s) - Kill(s))$

- Putting the two together

- $Temp = f_s (\prod_{s' \in \text{pred}(s)} Out(s'))$

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, ≤)

Available expressions

- P = sets of expressions
- $S1 \sqcap S2 = S1 \cap S2$
- Top = set of all expressions

Reaching Definitions

- P = set of definitions (assignment statements)
- $S1 \sqcap S2 = S1 \wedge S2$
- Top = empty set

Fixpoints

We always start with Top

- Every expression is available, no defs 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, ≤), cont'd

Live variables

- P = sets of variables
- $S1 \sqcap S2 = S1 \wedge S2$
- Top = empty set

Very busy expressions

- P = set of expressions
- $S1 \sqcap S2 = S1 \cap S2$
- Top = set of all expressions

Forward vs. Backward

$Out(s) = Top$ for all s
 $W := \{ \text{all statements} \}$
repeat
 Take s from W
 $temp := f_s(\prod_{s' \in pred(s)} Out(s'))$
 if (temp != $Out(s)$) {
 $Out(s) := temp$
 $W := W \wedge succ(s)$
 }
until $W = \emptyset$

$In(s) = Top$ for all s
 $W := \{ \text{all statements} \}$
repeat
 Take s from W
 $temp := f_s(\prod_{s' \in succ(s)} In(s'))$
 if (temp != $In(s)$) {
 $In(s) := temp$
 $W := W \wedge pred(s)$
 }
until $W = \emptyset$

Termination Revisited

How many times can we apply this step:

$$\text{temp} := f_s(\bigsqcap_{s' \in \text{pred}(s)} \text{Out}(s'))$$

if (temp != Out(s)) { ... }

Claim: Out(s) only shrinks

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

Termination Revisited (cont'd)

A *descending chain* in a lattice is a sequence

- $x_0 \supseteq x_1 \supseteq x_2 \supseteq \dots$

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

$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

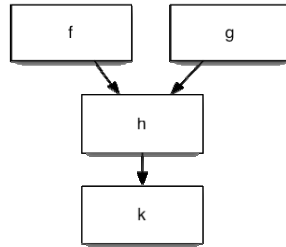
A function f is distributive if

$$f(x \sqcap y) = f(x) \sqcap f(y)$$

Benefit of Distributivity

Joins lose no information

$$\begin{aligned} k(h(f(T) \sqcap g(T))) &= \\ k(h(f(T)) \sqcap h(g(T))) &= \\ k(h(f(T))) \sqcap k(h(g(T))) \end{aligned}$$



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 $\{s_1, \dots, s_n\}$, let $f_p = f_n; \dots; f_1$
- Let $\text{path}(s)$ be the set of paths from the entry to s

$$\text{MOP}(s) = \sqcap_{p \in \text{path}(s)} f_p(T)$$

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?

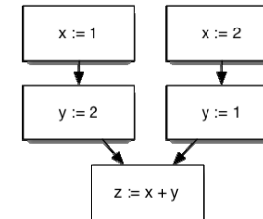
Analyses of *how* the program computes

- 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 $Q = \max \#$ back edges on cycle-free path

- Nesting depth
- Back edge is from node to ancestor on DFS tree

Then if $f(x) \leq x$ (sufficient, but not necessary)

- Running time is $O((Q + 1) |E|)$
 - Note direction of req't depends on top vs. bottom

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

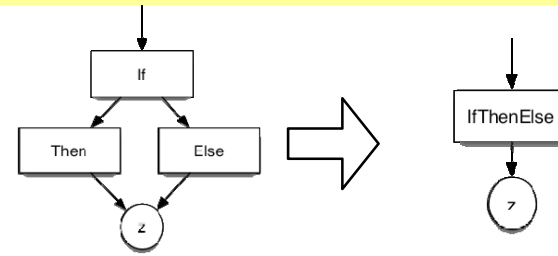
Another Approach: Elimination

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



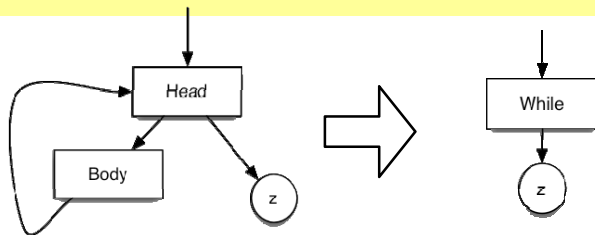
$$f_{ite} = (f_{then} \circ f_{if}) \sqcap (f_{else} \circ f_{if})$$

$$\text{Out}(\text{if}) = f_{if}(\text{In}(\text{ite}))$$

$$\text{Out}(\text{then}) = (f_{then} \circ f_{if})(\text{In}(\text{ite}))$$

$$\text{Out}(\text{else}) = (f_{else} \circ f_{if})(\text{In}(\text{ite}))$$

Elimination Methods: Loops



$$f_{while} = f_{head} \sqcap f_{head} \circ f_{body} \circ f_{head} \sqcap f_{head} \circ f_{body} \circ f_{head} \circ f_{body} \circ f_{head} \sqcap \dots$$

Elimination Methods: Loops (cont)

Let $f^i = f \circ f \circ \dots \circ f$ (i times)

- $f^0 = \text{id}$

Let

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

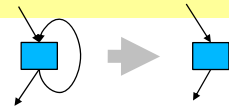
Need to compute limit as j goes to infinity

- Does such a thing exist?

Observe: $g(j+1) \leq g(j)$

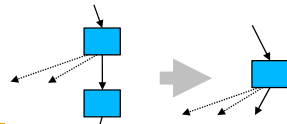
Forming regions: T1-T2 Reduction

Oldest and simplest



T1: self loop

Can reduce all well-structured graphs!



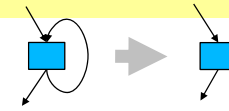
T2: two-block sequence

only requirement for T2:

second block has single predecessor

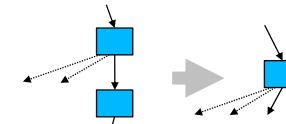
T1-T2 Reduction

Oldest and simplest



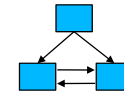
T1: self loop

Can reduce all well-structured graphs!



T2: two-block sequence

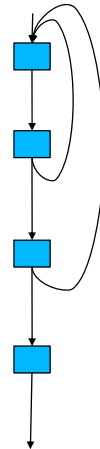
But...cannot reduce irreducible graphs!



--end up w/ "limit flow graph"

T1-T2 Example

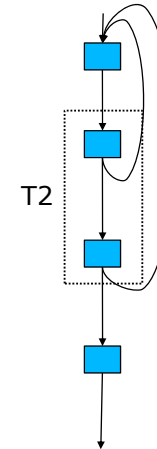
Hierarchy can seem strange....



T1-T2 Example

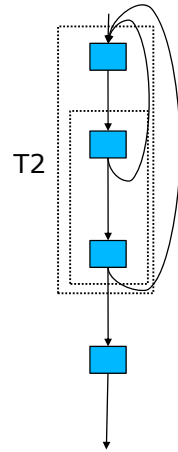
Hierarchy can seem strange....

(out edges from new region get merged - not shown)



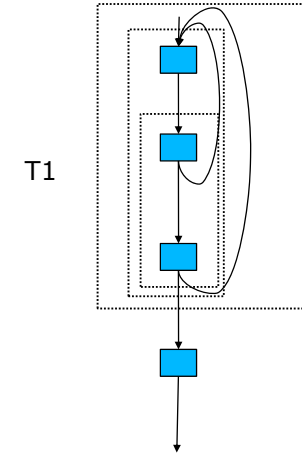
T1-T2 Example

Hierarchy can seem strange....



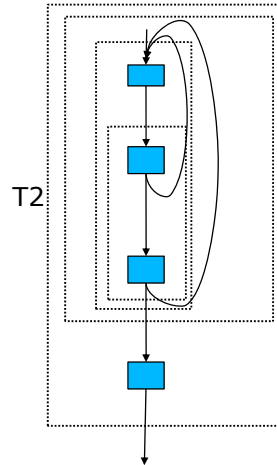
T1-T2 Example

Hierarchy can seem strange....



T1-T2 Example

Hierarchy can seem strange....

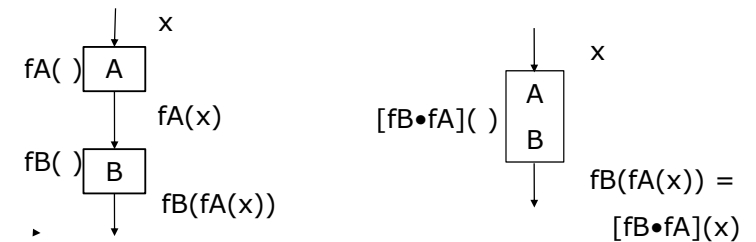


Why?

An alternate approach to dataflow analysis

- before, we iterated on basic blocks

Now, each time we form a region ->
form a composite transfer function that
summarizes the effect of that region



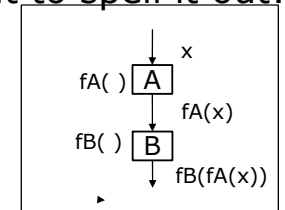
Dataflow Analysis on the Control Tree

- 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 $f_B \bullet f_A$?

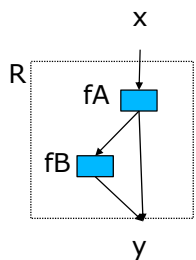
Well, we have already done this when computing the transfer function of a block that is a sequence of instructions...but to spell it out:



$$f_A(x) = \text{GenA} \cup (x - \text{KillA})$$

$$\begin{aligned} f_B(f_A(x)) &= \text{GenB} \cup (f_A(x) - \text{KillB}) \\ &= \text{GenB} \cup ((\text{GenA} \cup (x - \text{KillA})) - \text{KillB}) \\ &= \underline{\text{GenB} \cup (\text{GenA} - \text{KillB})} \cup (x - \underline{(\text{KillA} \cup \text{KillB})}) \end{aligned}$$

More Sample Calculations

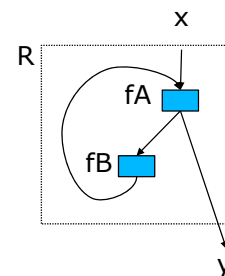


$$\begin{aligned} f_R(x) &= f_B(f_A(x)) \wedge f_A(x) \\ &= [(f_B \bullet f_A) \wedge f_A](x) \\ &= [(f_B \wedge I) \bullet f_A](x) \end{aligned}$$

\wedge 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



$$\begin{aligned} y = f_R(x) &= f_A(x) \wedge [f_A \bullet f_B \bullet f_A](x) \wedge \dots \\ &= [f_A \bullet (f_B \bullet f_A)^*](x) \end{aligned}$$

* is Kleene ("clay-nee") closure:

$$f^* = I \wedge f \wedge f \bullet f \wedge f \bullet f \bullet f \wedge \dots$$

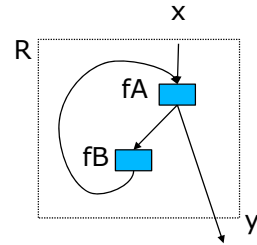
top-down calculations:

- $\text{in}(f_A) = [(f_B \bullet f_A)^*](x)$
- $\text{in}(f_B) = f_A(\text{in}(f_A))$

Example closure for gen/kill

$$fR(x) = I \cap (\prod_{n>0} f^n)$$

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



$$\begin{aligned} f^2(x) &= f(f(x)) \\ &= \text{gen} \cup (\text{gen} \cup (x - \text{kill}) - \text{kill}) \\ &= \text{gen} \cup (x - \text{kill}) \end{aligned}$$

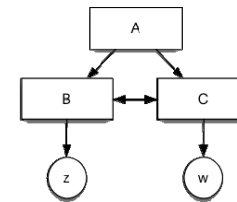
$$\begin{aligned} \text{So, } fR(x) &= I \cup (\text{gen} \cup (x - \text{kill})) \\ &= x \cup \text{gen} \end{aligned}$$

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