Data Flow Analysis

15-745

3/24/09

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
  - succ(s) = {immediate successor statements of s}
  - Pred(s) = {immediate predecessor statements of s}
  - In(s) program point just before executing s
  - Out(s) = program point just after executing s
- In(s) = $\cap_{s' \in pred(s)}$ Out(s') **must**
- Out(s) = Gen(s) ^ (In(s) – Kill(s)) **forward**
  - Note these are also called transfer functions

Gen(s) = *set of facts true after/before* s *that weren't true before/after*

Kill(s) = *set of facts no longer true after/before* s

forward/backward

Forward Data Flow, Again

Out(s) = Top for all statements s

W := { all statements } (worklist)

Repeat

Take s from W

- temp := $f_s(\sqcap_{s' \in pred(s)}$ Out(s')) (f_s monotonic *transfer fn*)
- if (temp != Out(s)) {
  - Out(s) := temp
  - W := W ^ succ(s)
- }

until W = ∅

What we would like to know:

Does it terminate?

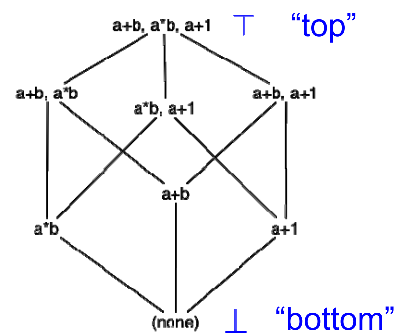
Is it accurate?

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, ‰) such that
  - ‰ #› P § P
  - ‰ #s *reflexive*: x ‰ x
  - ‰ ⫲is *anti-symmetric*: x ‰ y and y ‰ x implies x = y
  - ‰ ⫲is *transitive*: x ‰ y and y ‰ z implies x ‰ 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 x y ‰ x and x x y ‰ y
    - If z ‰ x and z ‰ y then z ‰ x x y
  - w is the join or least upper bound operation
    - x ‰ x w y and y ‰ x w y
    - If x ‰ z and y ‰ z, then x w y ‰ 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 x B = B   x w B =x

x x A = x   x w A = A

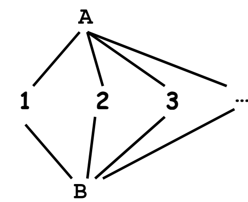
In a lattice

x ‰ y iff x x y = x

x ‰ y iff x w y = y

Useful Lattices

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



Monotonicity

- A function f on a partial order is monotonic if

  x ‰ y implies f(x) ‰ f(y)

- Easy to check that operations to compute In and Out are monotonic
  - In(s) = ⊓ $_{\mathbf{s' 5\ pred(s)}}$ Out(s')
  - Temp = Gen(s) ^ (In(s) – Kill(s))
- Putting the two together
  - Temp = f_s (⊓ $_{\mathbf{s' 5\ pred(s)}}$ 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 ⊓ S2 = S1 ∩ S2
- Top = set of all expressions

Reaching Definitions

- P = set of definitions (assignment statements)
- S1 ⊓ S2 = S1 ^ S2
- 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, ≤), cont'd

Live variables

- P = sets of variables
- S1 ⊓ S2 = S1 ^ S2
- Top = empty set

Very busy expressions

- P = set of expressions
- S1 ⊓ S2 = S1 ∩ S2
- Top = set of all expressions

Forward vs. Backward

```
Out(s) = Top  for all s
W := { all statements }
repeat
    Take s from W
    temp := f_s(⊓_{s' ∈ pred(s)} Out(s'))
    if (temp != Out(s)) {
     Out(s) := temp
     W := W ^ succ(s)
    }
until W = ∅
```

```
In(s) = Top  for all s
W := { all statements }
repeat
    Take s from W
    temp := f_s(⊓_{s' ∈ succ(s)} In(s'))
    if (temp != In(s)) {
     In(s) := temp
     W := W ^ pred(s)
    }
until W = ∅
```

Termination Revisited

How many times can we apply this step:

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

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 $\sqcap_{s' \in pred(s)}$ Out(s') shrinks
- Since f_s monotonic, $f_s(\sqcap_{s' \in pred(s)} Out(s'))$ 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

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

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

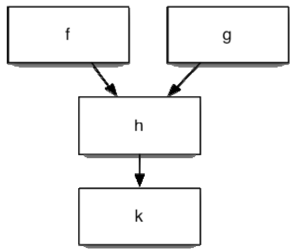
A function f is distributive if

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

Benefit of Distributivity

Joins lose no information

$$k(h(f(\top) \sqcap g(\top))) = \\ k(h(f(\top)) \sqcap h(g(\top))) = \\ k(h(f(\top))) \sqcap k(h(g(\top)))$$



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 $f_p = f_n;\ldots;f_1$
- Let path(s) be the set of paths from the entry to s

$$\mathrm{MOP}(s) = \sqcap_{p \in \mathrm{path}(s)} f_p(\top)$$

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?

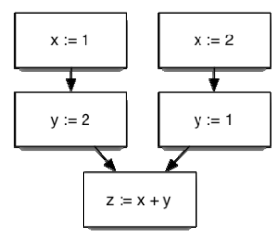
Analyses of *how* the program computes

- 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 ; x. f(x)‰ 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

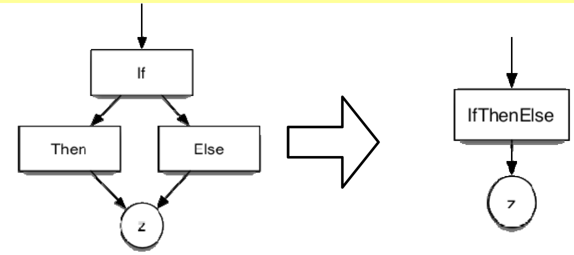
Another Approach: Elimination

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

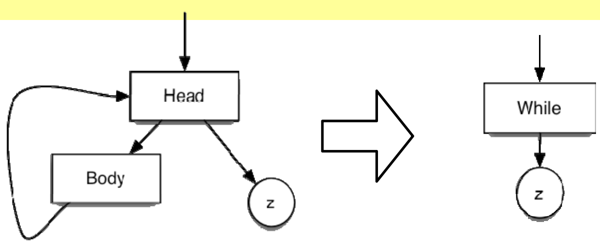
Elimination Methods: Conditionals



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

$$\begin{aligned} \text{Out(if)} &= f_{\text{if}}(\text{In(ite)}) \\ \text{Out(then)} &= (f_{\text{then}} \circ f_{\text{if}})(\text{In(ite)}) \\ \text{Out(else)} &= (f_{\text{else}} \circ f_{\text{if}})(\text{In(ite)}) \end{aligned}$$

Elimination Methods: Loops



$$\begin{aligned} f_{\text{while}} \quad = \quad & f_{\text{head}} \sqcap \\ & f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \sqcap \\ & f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \sqcap \cdots \end{aligned}$$

Elimination Methods: Loops (cont)

Let $f^{i} = f \circ f \circ \ldots \circ f$ (i times)

- $f^{0} = id$

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

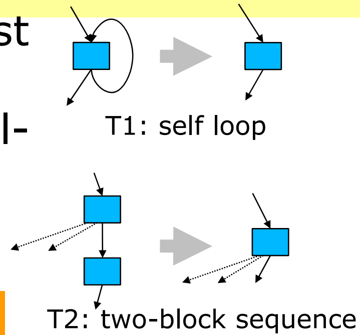
- Does such a thing exist?

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:

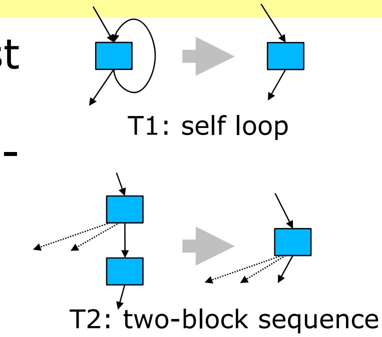
second block has

single predecessor

T1-T2 Reduction

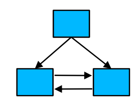
Oldest and simplest

Can reduce all well-structured graphs!



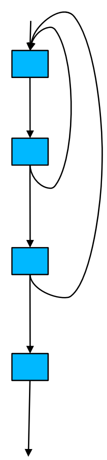
But...cannot reduce irreducible graphs!

--end up w/ "limit flow graph"



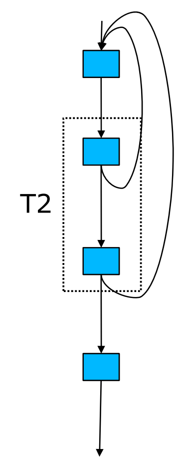
T1-T2 Example

Hierarchy can seem strange....



T1-T2 Example

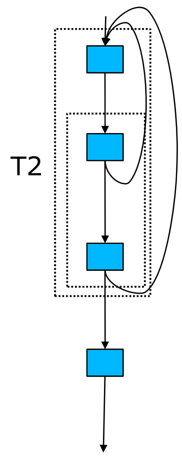
Hierarchy can seem strange....



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

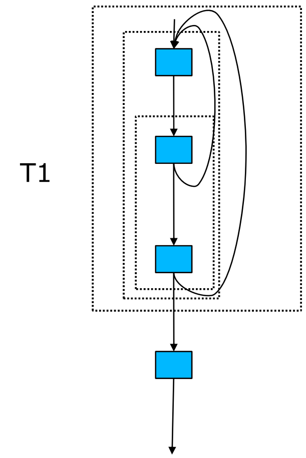
T1-T2 Example

Hierarchy can seem strange....



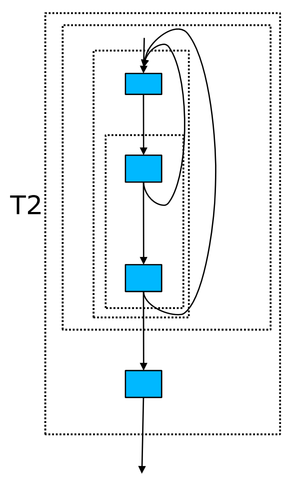
T1-T2 Example

Hierarchy can seem strange....



T1-T2 Example

Hierarchy can seem strange....

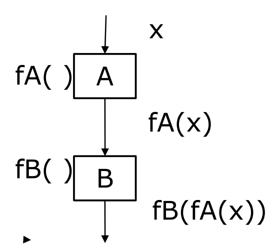


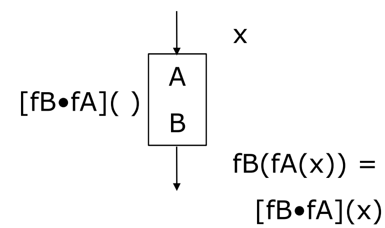
Why?

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
<u>summarizes the effect of that region</u>





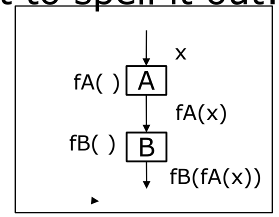
Dataflow Analysis on the Control Tree

- 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 fB•fA?

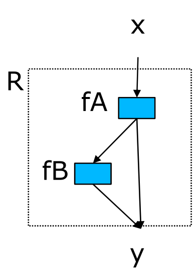
Well, we have already done this when computing the transfer function of a block that is a sequence of instructions...but to spell it out:



fA(x) = GenA U (x-KillA)

fB(fA(x)) = GenB U (fA(x) – KillB)
= GenB U ((GenA U (x-KillA)) – KillB)
= <u>GenB U (GenA – KillB)</u> U (x – <u>(KillA U KillB)</u>)

More Sample Calculations



fR(x) = fB(fA(x)) ^ fA(x)

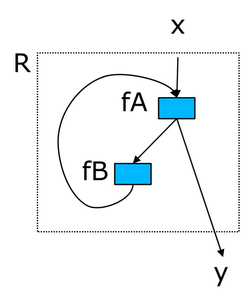
= [(fB•fA) ^ fA](x)

= [(fB ^ I) • fA](x)

^ is the meet operator

- gets just slightly more complicated for flow-sensitive transfer functions where fA_{then} is different than fA_{else}
- distribution caluclation (coming down the control tree) is obvious

More Sample Calculations



y = fR(x) = fA(x) ^ [fA•fB•fA](x) ^ ....

= [fA•(fB•fA)*] (x)

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

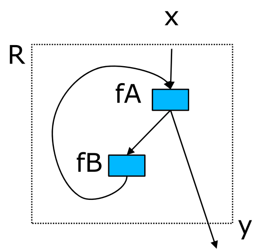
f* = I ^ f ^ f•f ^ f•f•f ^ ....

top-down calculations:

- in(fA) = [(fB•fA)*](x)
- in(fB) = fA(in(fA))

Example closure for gen/kill

$fR(x) = I \sqcap (\sqcap_{n>0} f^n)$



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

$f^2(x) = f(f(x))$
$= gen \cup ((gen \cup (x - kill)) - kill)$
$= gen \cup (x - kill)$

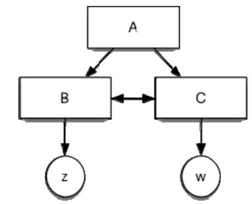
So, $fR(x) = I \cup (gen \cup (x - kill))$
$= x \cup gen$

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