Dataflow Frameworks, conclusion

Announcements

- Homework is due Monday
  - Problem 6(b) is Extra credit
  - Submit in Submitty
  - You can lock your team later. Maximal size is 3. Ideal size is 2.
- Quiz 1 today

Outline of Today's Class

- Dataflow frameworks, conclusion
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation

Monotone Dataflow Frameworks

- A problem fits into the dataflow framework if
  - its property space is a lattice **L, ≤** that satisfies the *Ascending Chain Condition*
  - its merge operator V is the join of **L**

  and
  - its transfer function space **F: L→ L** is monotone
- Thus, we can make use of a generic solution procedure, known as the worklist algorithm or the maximal fixpoint algorithm or the fixpoint iteration algorithm

Catch up: Transfer Functions

- **The transfer functions: f: L→ L**. Formally, function space **F** is such that
  1. **F** contains all $\mathbf{f_j}$
  2. **F** contains the identity function **id(x) = x**
  3. **F** is closed under composition
  4. Each **f** is **monotone**

Catch up: Monotonicity

- **F: L→ L** is monotone if and only if:

  (1) **a,b** in **L**, **f** in **F** then **a ≤ b** $\Longrightarrow$ **f(a) ≤ f(b)**

  or (equivalently):

  (2) **x,y** in **L**, **f** in **F** then **f(x) V f(y) ≤ f(x V y)**
- Theorem: Definitions (1) and (2) are equivalent.
  - Show that (1) implies (2)
  - Show that (2) implies (1)

Catch up: Distributivity

- **F: L → L** is distributive if and only if **x,y** in **L**, **f** in **F** then **f(x V y) = f(x) V f(y)**
- Every distributive function is also monotone but not the other way around
- Distributivity is a very nice property!

Catch up: Monotonicity and Distributivity

- Is classical *Reach* distributive?
  - Yes
  - To show distributivity:

For each **j ((X U Y) ∩ pres(j)) U gen(j)** =
((X∩pres(j)) U gen(j)) U ((Y∩pres(j)) U gen(j))

((X U Y) ∩ pres(j)) U gen(j) =
((X ∩ pres(j)) U (Y ∩ pres(j))) U gen(j) =
((X ∩ pres(j)) U gen(j)) U ((Y ∩ pres(j)) U gen(j))

Monotone Dataflow Frameworks

- A problem fits into the dataflow framework if
  - its property space is a lattice **L, ≤** that satisfies the *Ascending Chain Condition*
  - its merge operator V is the join of **L**

  and
  - its transfer function space **F: L→ L** is monotone
- Thus, we can make use of a generic solution procedure, known as the worklist algorithm or the maximal fixpoint algorithm or the fixpoint iteration algorithm

Worklist Algorithm for Forward Dataflow Problems

```
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue;                  in_Reach (1) = UNDEF
for m = 2 to n do in(m) = 0            in_Reach (m) = {}
W = {1,2,…,n} /* put every node on the worklist */

while W ≠ Ø do {
   remove j from W
   out(j) = f_j(in(j))                 out_Reach(j) = (in_Reach(j)∩pres(j))Ugen(j)
   for i in successors(j)
     if out(j) ≰ in(i) then {           if out_Reach(j) ⊈ in_Reach (i)
        in(i) = out(j) V in(i)            in_Reach(i) = out_Reach(j) U in_Reach(i)
        W = W U { i }
     }
}
```

Worklist Algorithm for Forward Dataflow Problems (slightly different)

```
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue; out(1) = f_1(in(1))
for m = 2 to n do in(m) = 0; out(m) = f_m(0)
W = {2,…,n} /* put every node but 1 on the worklist */

while W ≠ Ø do {
   remove j from W
   in(j) = V { out(i) | i is predecessor of j }
   out(j) = f_j(in(j))
   if out(j) changed then
      W = W U { k | k is successor of j }
}
```


Termination Argument

- Why does the algorithm terminate?
- Sketch of argument:

  At each "phase", at least one out(j) changes. Monotonicity of $\mathbf{f_j}$ entails that change of is up the chain: out'(j)≥out(j). Since out(j) in **L**, and **L** satisfies the *Ascending Chain Condition*, out(j) changes at most ***O(h)*** times where **h** is the height of the lattice **L.**

Correctness Argument

- Theorem: The worklist algorithm computes a solution that satisfies the dataflow equations
- Why?
- Sketch of argument:

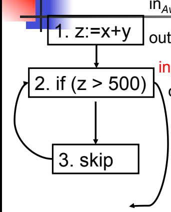
  Whenever **j** is processed, algorithms sets out(j) = $\mathbf{f_j}$(in(j)). Whenever out(j) changes, algorithm puts successors on the list, so in(j) = **V** { out(i) }.

  So final solution will satisfy equations.

Precision Argument

- Theorem: The algorithm computes the least solution of the dataflow equations.
  - Historically though, this solution is called the **maximal fixpoint solution (MFP)**
  - I.e., For every node **j**, the worklist algorithm computes a solution **MFP(j)** = {in(j),out(j)}, such that every other solution {in'(j),out'(j)} of the dataflow equations is in(j) ≤ in'(j), out(j) ≤ out'(j)

Example



	Solution1	Solution2
$in_{Avail}(1) = \varnothing$	∅	∅
$out_{Avail}(1) = (in_{Avail}(1)-E_z) \cup \{x+y\}$	{x+y}	{x+y}
$in_{Avail}(2) = out_{Avail}(1)$ **V** $out_{Avail}(3)$	{x+y}	∅
$out_{Avail}(2) = in_{Avail}(2)$		
$in_{Avail}(3) = out_{Avail}(2)$	{x+y}	∅
$out_{Avail}(3) = in_{Avail}(3)$		

Equivalent to: $in_{Avail}(2) = \{x+y\}$ **V** $in_{Avail}(2)$ and recall that **V** is ∩ (i.e., set intersection).

Question

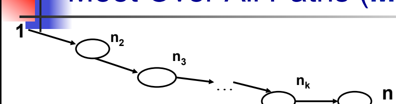
Willy Wazoo changed the worklist algorithm to initialize to 1.

```
/* Initialize to initial values; 1 is entry node of CFG */
in(1) = InitialValue;                      in_Reach(1) = UNDEF
for m = 2 to n do in(m) = 1                in_Reach(m) = ALL_DEFS
W = {1,2,…,n} /* put every node on the worklist */

while W ≠ Ø do {
   remove j from W
   out(j) = f_j(in(j))                     out_Reach(j) = (in_Reach(j)∩pres(j))Ugen(j)
   for i in successors(j)
     if out(j) ≰ in(i) then {              if out_Reach(j) ⊈ in_Reach(i)
        in(i) = out(j) V in(i)                in_Reach(i) = out_Reach(j) U in_Reach(i)
        W = W U { i }
     }
}
```

1. Does Willy's algorithm compute an over-approximation or an under-approximation of the MFP?

Meet Over All Paths (**MOP**)



- Desired dataflow information at **n** is obtained by traversing ALL PATHS from 1 (entry node) to **n**. For every path **p=(1, $\mathbf{n_2}$, $\mathbf{n_3}$ ..., $\mathbf{n_k}$)** we compute $\mathbf{f_{n_k}(\ldots f_{n_2}(f_1(init(1))))}$
- The MOP at *entry* of **n** is $\mathbf{V\ f_{n_k}(\ldots f_{n_2}(f_1(init(1))))}$

  over all paths **p** from **1** to **n**

MOP vs. MFP

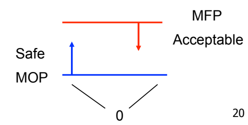
- The MOP is an abstract model for the best solution computable with this kind of static analysis
  - It is a common assumption in this kind of static analysis that *all program paths are executable*
  - (Abstract interpretation and axiomatic semantics abstract state more precisely, and can rule out some paths)
- The MFP is the solution computed by the worklist algorithm

MOP vs. MFP

- For *distributive* problems MFP = MOP!
- Unfortunately, for *monotone* problems this is not true. But we still have a safe solution: it is a theorem that for monotone problems, MFP ≥ MOP

Safety of a Dataflow Solution

- A safe (also, correct or sound) solution X overestimates the "best" possible dataflow solution, i.e., X ≥ MOP
- Historically, an acceptable solution X is one that is better than what we can do with the MFP, i.e., X ≤ MFP

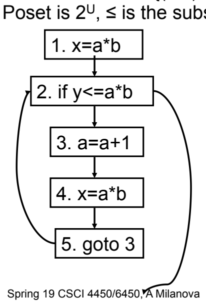


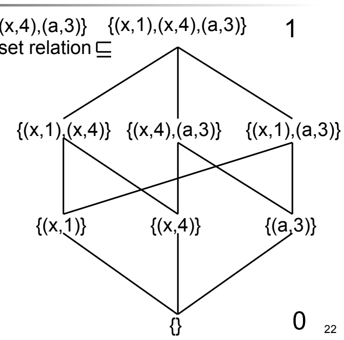
Safe Solutions

- In *may problems* 1 is the universal set of facts, the merge operator is set union. It is safe to err by saying that a fact reaches a node when in fact it doesn't
  - E.g., intuitively, it is safe to err by saying that a definition **(x,k)** reaches a node, when in fact it MAY NOT REACH that node
  - Safe entails "larger" then the MOP under our partial order. Our definition of ≤ is subset inclusion (which is natural). So "safer" solutions are larger sets

Safe Solutions: *Reach*

U = all definitions:{(x,1),(x,4),(a,3)}
Poset is 2^U, ≤ is the subset relation ⊑



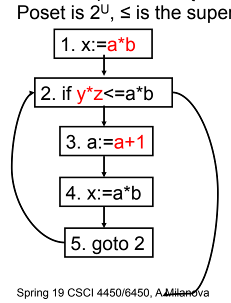


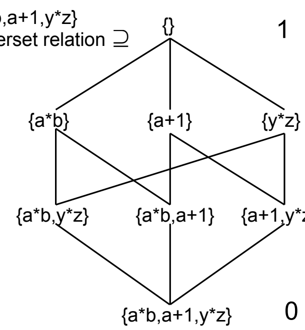
Safe Solutions

- In *must problems* the 1 is the empty set, and the merge operator is set intersection. It is safe to err by saying that a fact does not reach a node when in fact it does
  - E.g., intuitively, it is safe to err by saying that an expression is NOT AVAILABLE when in fact it may be available
  - Safe means larger then the MOP under our partial order. Under our definition of ≤, which is superset, "safer" solutions end up being smaller sets

Safe Solutions: *Avail*

U = all expressions: {a*b,a+1,y*z}
Poset is 2^U, ≤ is the superset relation ⊒





Precision of a Dataflow Solution

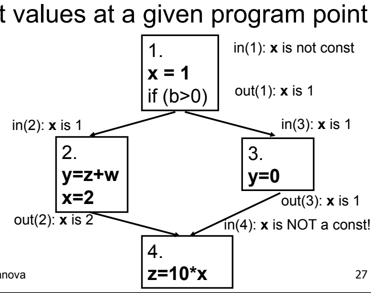
- Precise solution is one that is “close” to MOP
  - A precise solution contains few spurious dataflow facts (spurious facts is what we call noise)
  - Unfortunately, for most problems even the MOP (an approximation itself!) is undecidable
- MOP ≤ X ≤ Y: X is more precise than Y
  - Usually we can compare two solutions X and Y
  - But, for most problems, we have no way of knowing the “ground truth”

Outline of Today’s Class

- Dataflow frameworks
- MOP vs. MPF
- Non-distributive analyses
  - Constant propagation

Constant Propagation (Simple)

- Problem statement: What variables always hold constant values at a given program point
- Example:



Aside: Defining an Analysis

- Define program syntax
  - In practice, we deal with a lot more than the simple abstraction ☺
- Define property space
  - The *abstract program state* that approximates the concrete program state
- Define transfer function space over syntax
  - Symbolically execute program over abstract state

Aside: Defining an Analysis

- If property space has desired properties
  - is a lattice **L, ≤** that satisfies the *Ascending Chain Condition*
  - merge operator V is the join of **L**

  and
- Function space **F: L→ L** is monotone

then analysis fits the monotone dataflow framework and can be solved using the worklist algorithm

Constant Propagation: Syntax

S ::= *S*; *S* | **while (b) {** *S* **}** | **if (b) {** *S* **} else {** *S* **}**
| **x =** *V* | **x =** *V Op V*

Op ::= + | - | * | /

V ::= **x** | **y** | **z** | **C**

- **x, y, z** are program variables
- **C** is constant
- We have to define transfer functions for **x =** *V* and **x =** *V Op V*

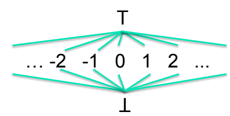
Constant Propagation: Property Space

- Associate one of the following values with variable **x** at each program point

value	meaning
1 (or T)	**x** is NOT a constant
c	**x** has constant value **c**
0 (or ⊥)	**x** is unknown

Constant Propagation: Lattice

- Lattice **L_x,≤**



- Dataflow lattice **L** is the product of **L_x**
  - **l1,l2** in **L**, **l1 ≤ l2** iff **$l1_x$ ≤ $l2_x$** for every variable **x**
  - **l1** V **l2** amounts to **$l1_x$** V **$l2_x$** for every variable **x**
  - Merge operator is join of **L**
- Does the product lattice satisfy the ACC?

Constant Propagation: Transfer Functions

- **j: x = c**

  f_j: kill **x** → **val,** generate **x** → **c**
- **j: x = y**

  f_j: kill **x** → **val,** add **x** → **val'**, s.t. **y** → **val'** in in(j).
- **val** and **val'** are one of
  - ⊥: bottom (unknown)
  - c: constant
  - T: top (not a constant)

Constant Propagation: Transfer Functions

- **j**: **x =** V_1 *Op* V_2

  f_j: kill: **x** → **val**

  gen:

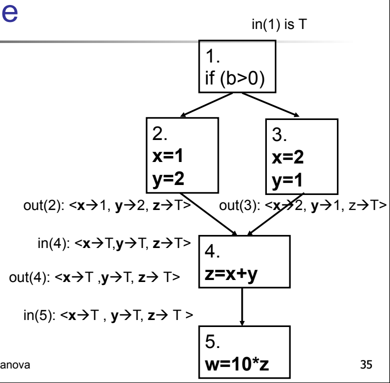
If V_1 → **c_1** and V_2 → **c_2** in in(j), then **x** → **c_1** *Op* **c_2**

else if V_1 → T or V_2 → T in in(j), then **x** → T

else **x** → ⊥

- Next, we'll argue monotonicity which would give us that Constant Propagation is solvable by the Worklist algorithm

Example



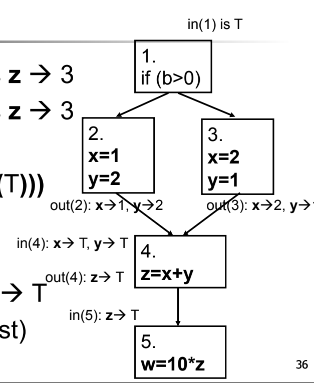
Not Distributive! A Counter Example

- **f_4(f_2(f_1(T)))** implies **z** → 3
- **f_4(f_3(f_1(T)))** implies **z** → 3
- Thus, MOP at 5

f_4(f_2(f_1(T))) V **f_4(f_3(f_1(T)))** implies **z** → 3

MFP at 5 gave us **z** → T

(i.e., **z** is NOT a const)



Next Class

- We'll continue with non-distributive analyses
  - Constant propagation
  - Points-to analysis

- Introduction to Soot