Dataflow Analysis: Non-distributive Analysis

Outline of Today's Class

- Dataflow frameworks, conclusion
  - Lattices (last time)
  - Transfer functions (last time)
  - Worklist algorithm
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis

Monotone Dataflow Framework

- 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
  - Computes the so-called MFP solution

Worklist Algorithm for Forward Dataflow Problems

/* Initialize to initial values; 1 is entry node of CFG */

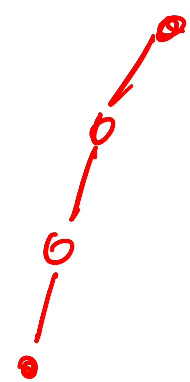
in(1) = InitialValue; out(1) = f_1(in(1))

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 */

$in^t(j) \le in^{t+1}(j)$

$out^t(j) \le out^{t+1}(j)$



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

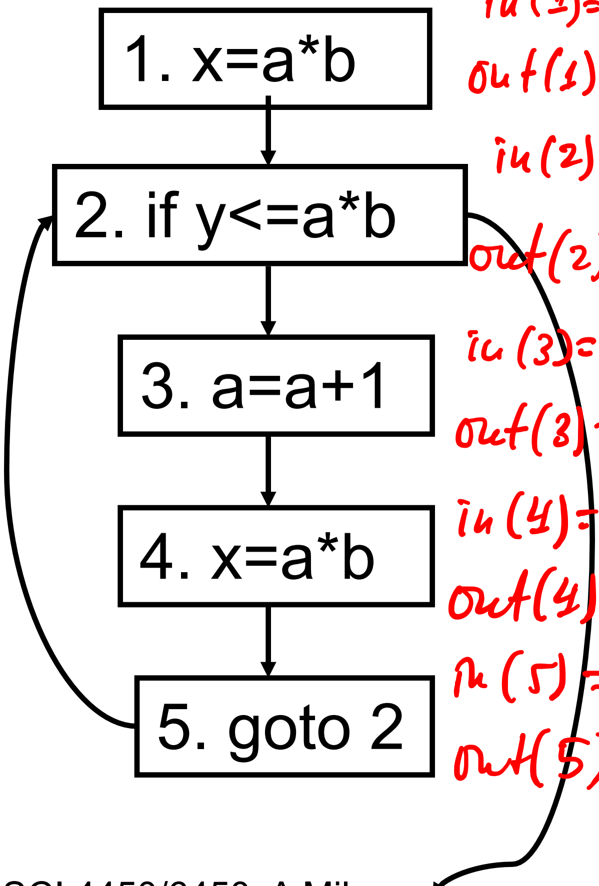
        W = W U { k | k is successor of j }

}

Worklist Algorithm on *Reach*

D = all definitions:{(x,1),(x,4),(a,3)}

Poset is 2^D, ≤ is the subset relation ⊑



in(1)={}
out(1)={(x,1)}
in(2)={}
out(2)={}
in(3)={}
out(3)={(a,3)}
in(4)={}
out(4)={(x,4)}
in(5)={}
out(5)={}

W={2,3,4,5}

Iter 1
remove 2
in(2)={(x,1)} out(2)={(x,1)}

Iter 2
remove 3
in(3)={(x,1)} out(3)={(x,1),(a,3)}

Iter 3
remove 4
in(4)={(x,1),(a,3)} out(4)={(x,4),(a,3)}

Iter 4
remove 5 W={}
in(5)={(x,4),(a,3)} out(5)={(x,4),(a,3)}
W={2}

Termination Argument

- Theorem: the algorithm terminates. Why?

- Sketch of argument:

  A node k is placed on worklist only if the out(j) of a predecessor j changes. Monotonicity of **f** guarantees $in^t(j) \leq in^{t+1}(j)$ and $out^t(j) \leq out^{t+1}(j)$. (Here $in^t(j)$, $out^t(j)$ are the sets at iteration t.)

  in and out sets are elements of **L** and **L** satisfies the *Ascending Chain Condition*; thus, there is only a finite number of times each out(j) changes.

Correctness Argument

- Theorem: Worklist algorithm computes a solution that satisfies the dataflow equations. Why?

- Sketch of argument:

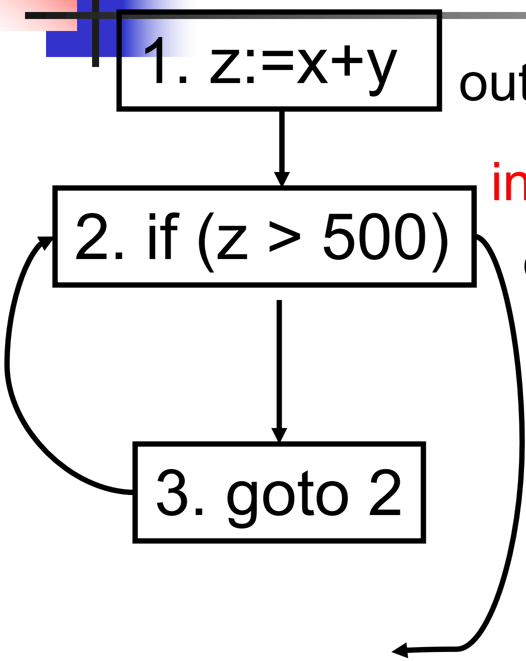
  Suppose either (1) $Vout(i) \neq in(j)$ or (2) $out(j) \neq f_j(in(j))$

  For (1) to hold we must have "grown" out(i) in some iteration and not added successor j to worklist; this is impossible.

Precision Argument

- Theorem: Worklist algorithm computes the least solution of the dataflow equations.
  - Historically, solution computed by worklist algorithm is called the **maximal fixpoint solution (MFP solution)**
  - For every node j, worklist algorithm computes a solution MFP(j) = (in(j),out(j)), such that for every solution (in'(j),out'(j)) of the dataflow equations we have in(j) ≤ in'(j) and out(j) ≤ out'(j)

Example (*Avail*)



in(1) = Ø

out (1) = (in(1)-E_z) U {x+y}

in(2) = out(1) **V** out(3)

out(2) = in(2)

in(3) = out(2)

out(3) = in(3)

{ }

|

{x+y}

$X = \{x+y\} \cap X$

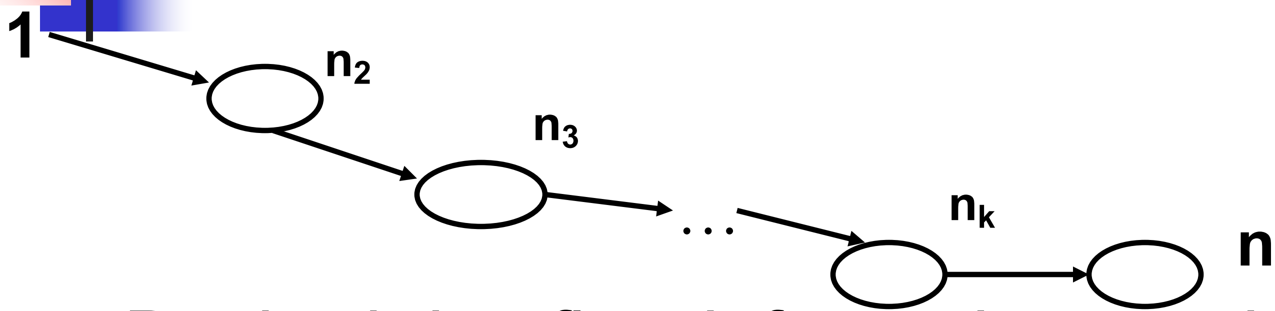
	MFP Solution1	Solution2
in(1)	Ø	Ø
out(1)	{x+y}	{x+y}
in(2)	{x+y}	Ø
in(3)	{x+y}	Ø

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

Outline of Today's Class

- Dataflow frameworks, conclusion
  - Lattices (last time)
  - Transfer functions (last time)
  - Worklist algorithm
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis

Meet Over All Paths (**MOP**)



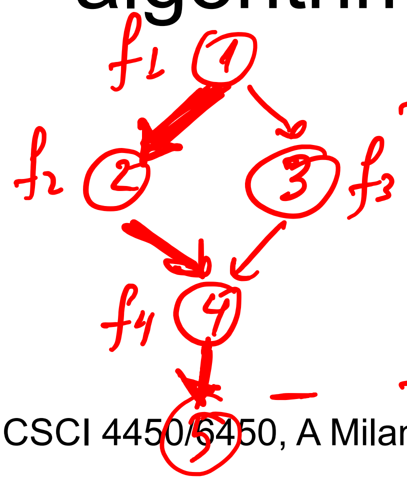
- Desired dataflow information at **n** is obtained by traversing ALL PATHS from 1 (entry node) to **n**.

- For every path **p=(**1, $\mathbf{n_2}$, $\mathbf{n_3}$ **...,** $\mathbf{n_k}$**)** we compute $\mathbf{f_{n_k}(\ldots f_{n_2}(f_1(}\mathrm{InitialValue}\mathbf{)))}$
- The MOP at *entry* of **n** is $\mathbf{V}$ $\mathbf{f_{n_k}(\ldots f_{n_2}(f_1(}\mathrm{InitialValue}\mathbf{)))}$

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

MOP vs. MFP

- MOP is an abstraction of the best solution computable with dataflow analysis
  - It is a common assumption in dataflow analysis that *all program paths are executable*
- MFP is the solution computed by the worklist algorithm



MOP:

$f_4(f_2(f_1(Init))) \lor$

$f_4(f_3(f_1(Init)))$

$f_4(out(2)) \lor f_4(out(3))$ vs

MFP:

$out(4) = f_4(in(4))$

$in(4) = out(2) \lor out(3)$

$out(2) = f_2(f_1(Init))$

$out(3) = f_3(f_1(Init))$

$f_4(out(2) \lor out(3))$

MOP vs. MFP

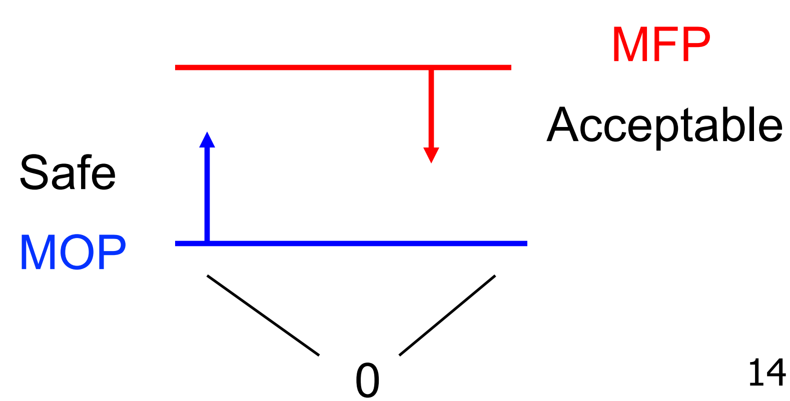
- For *distributive* problems MFP = MOP!

- Unfortunately, for *monotone* problems this is not true. But we still have a safe solution: it is a theorem that for monotone problems, MFP $\geq$ MOP

Safety of a Dataflow Solution

- A safe (also, correct or sound) solution X overestimates the "best" possible dataflow solution, i.e., $X \geq MOP$

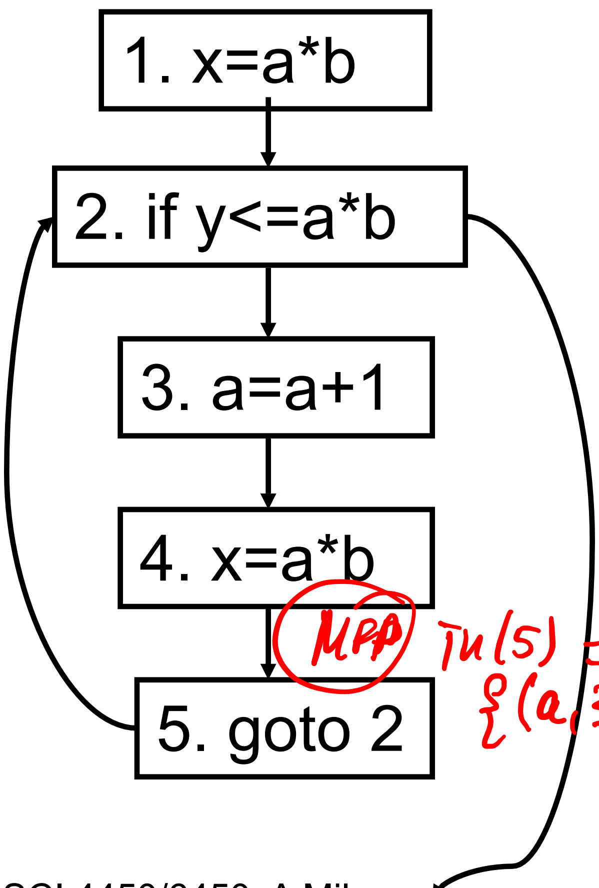
- Historically, an acceptable solution X is one that is better than what we can do with the MFP, i.e., $X \leq MFP$



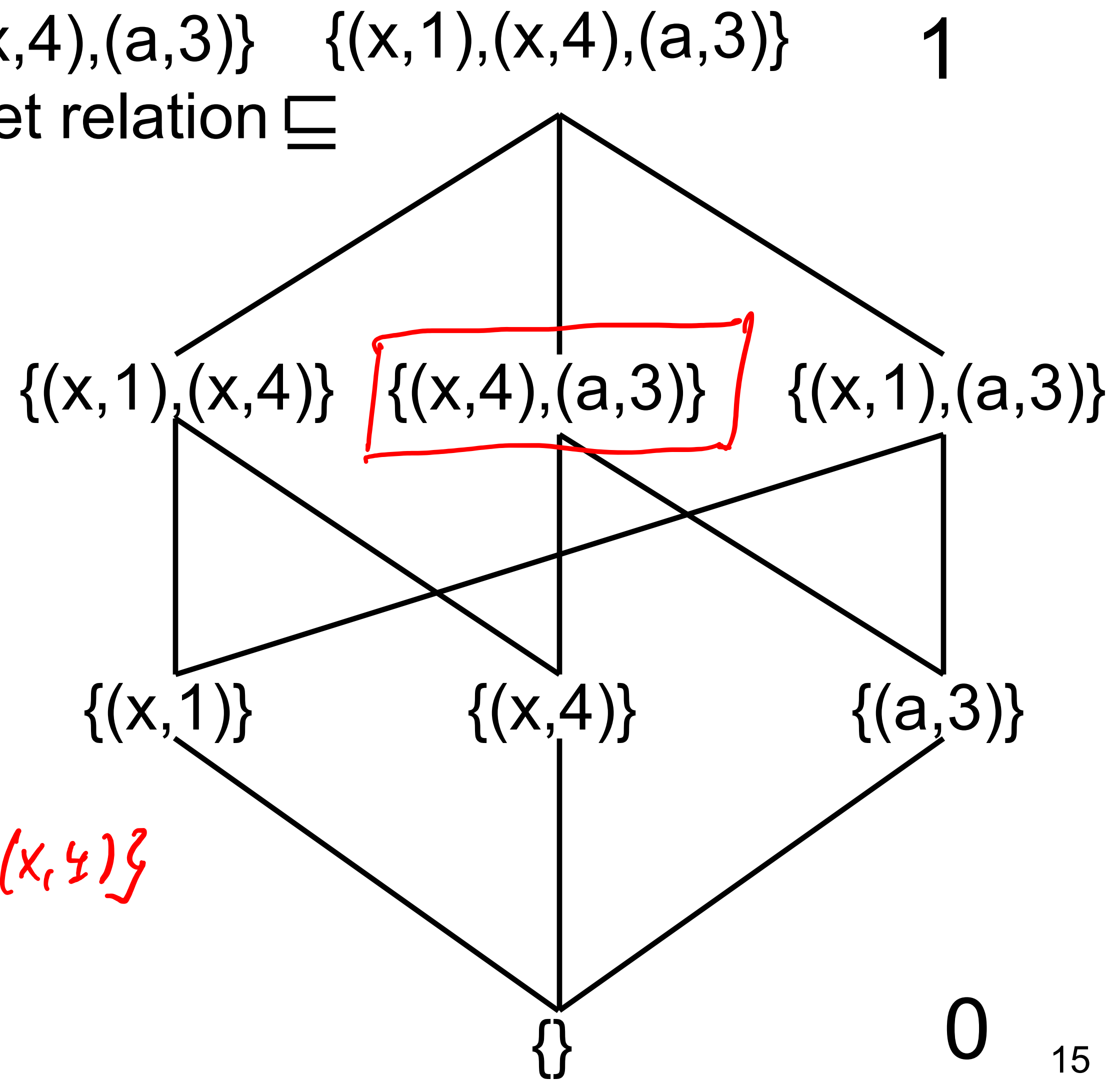
Safe Solutions: *Reach*

U = all definitions:{(x,1),(x,4),(a,3)}

Poset is 2^U, $\leq$ is the subset relation $\sqsubseteq$



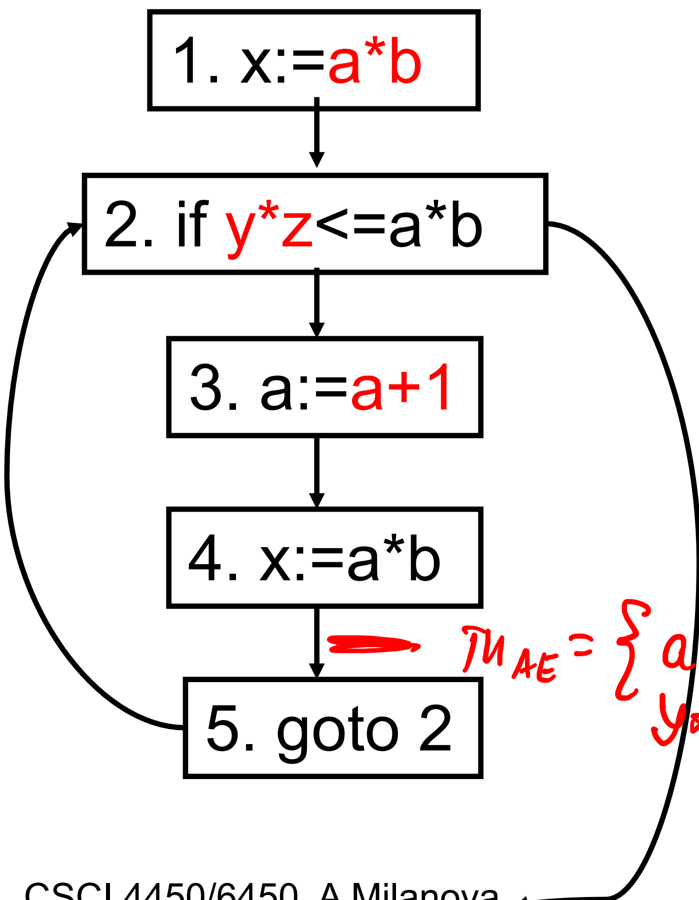
MFP Ju(5) = {(a,3),(x,4)}



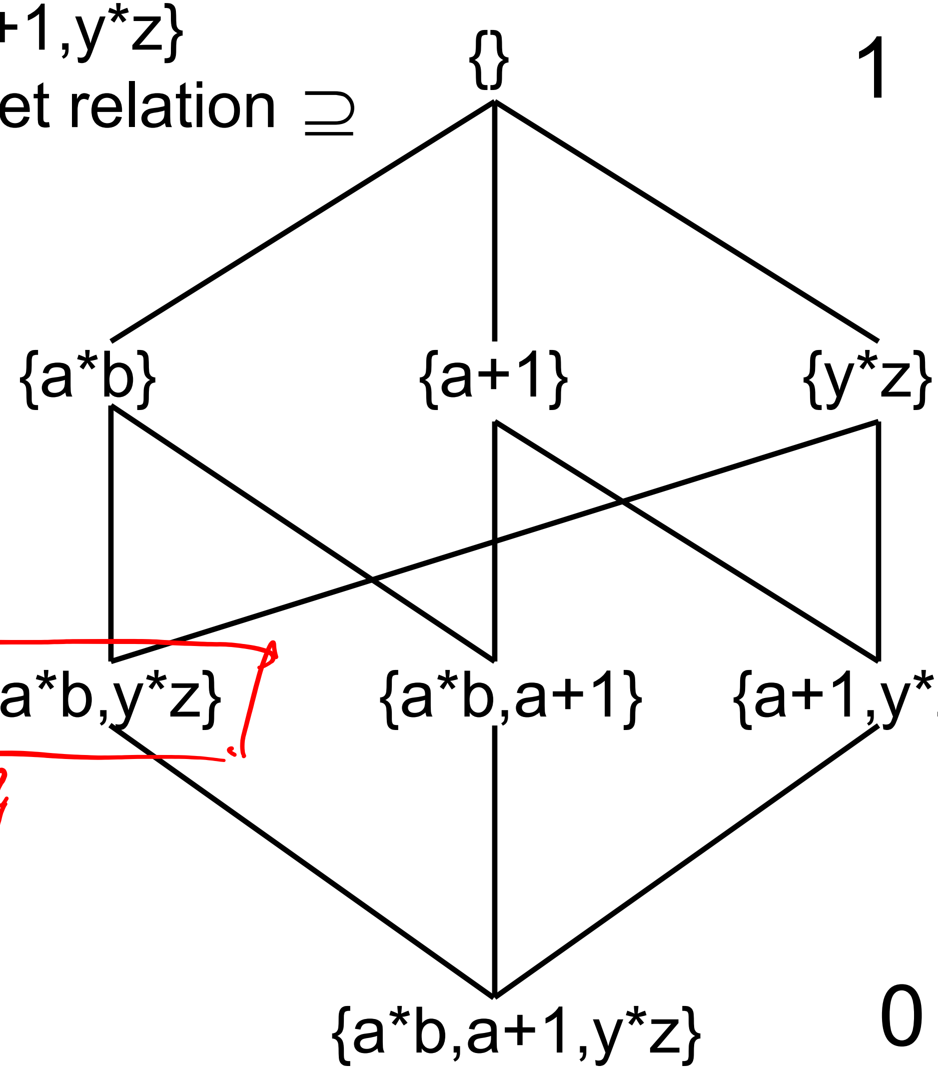
Safe Solutions: *Avail*

U = all expressions: {a*b,a+1,y*z}

Poset is 2^U, ≤ is the superset relation ⊇



$IN_{AE} = \{a*b, y*z\}$

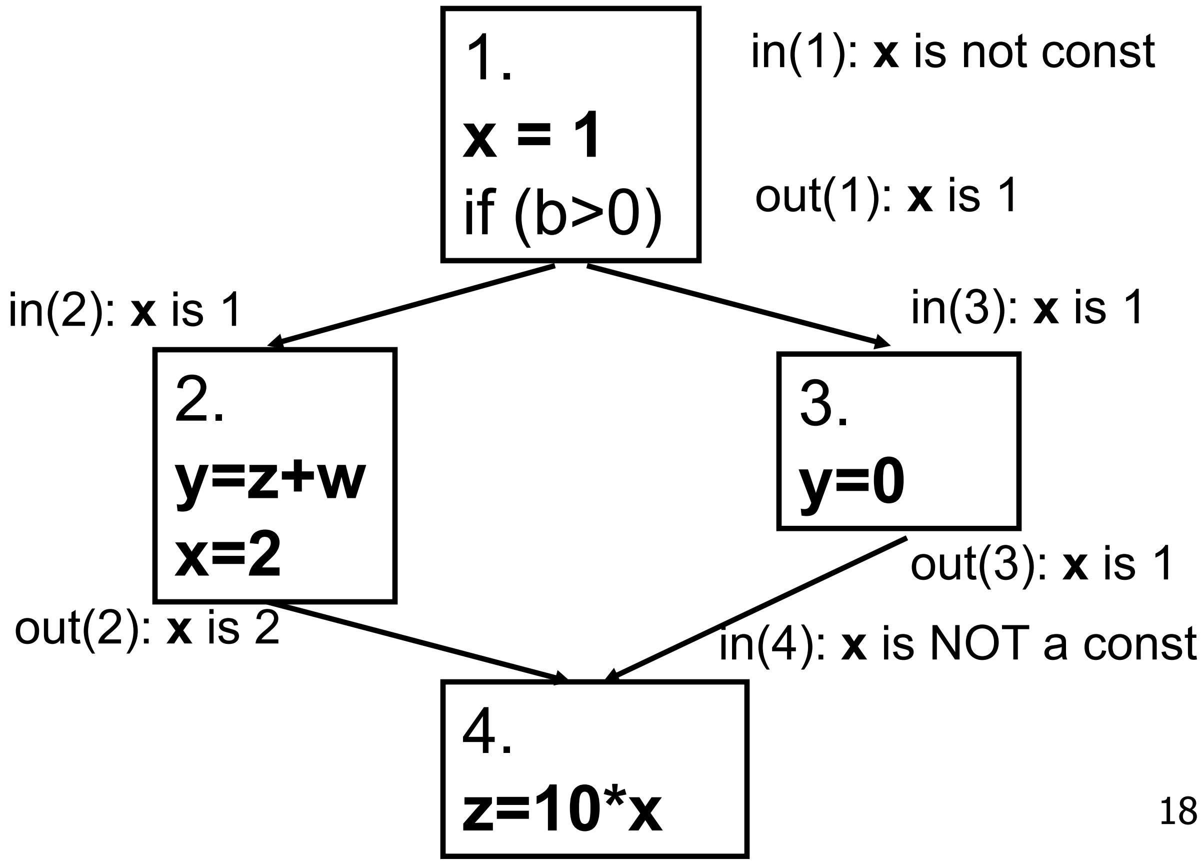


Outline of Today's Class

- Dataflow frameworks, conclusion
  - Lattices (last time)
  - Transfer functions (last time)
  - Worklist algorithm
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis

Constant Propagation (Simple)

- Problem statement: Can variable **x** hold a constant value at a given program point?
- Example:



Let’s Fit Analysis into Monotone Dataflow Framework

- If property space has desired properties
  - it is a lattice **L, ≤** that satisfies the *Ascending Chain Condition*
  - its merge operator 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: Property Space

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

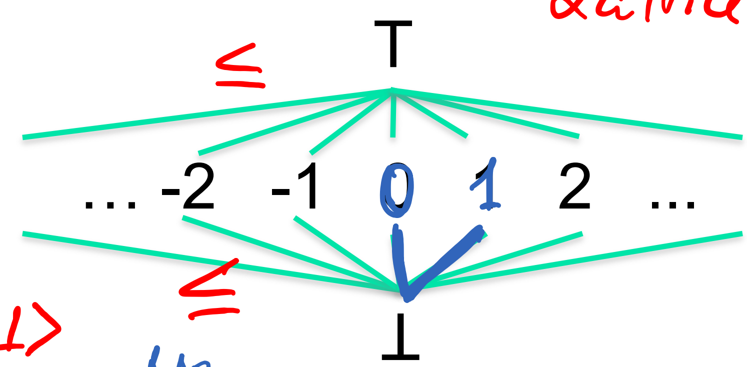
value	meaning
1 (or T)	**x** is NOT a constant
C	**x** has constant value **C**
0 (or $\perp$)	**x** is unknown

Constant Propagation: Lattice

Lattice of constants

- Lattice $\mathbf{L_x}, \leq$

$\bot \leq 1$, $0 \leq T$



$c1 \vee c2 = T$

$c1 \wedge c2 = \bot$

$l1 = \langle x \to \bot, y \to T, z \to \bot \rangle$

$l2 = \langle x \to \bot, y \to 5, z \to 0 \rangle$

No

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

Product Lattice

- E.g.,

<**x**→⊥, **y**→**1**, **z**→T>, <**x**→**1**, **y**→**2**, **z**→**3**>, etc. are lattice elements

- E.g.,

<x→1, y→2, z→T> **≤** **<x→T, y→2, z→**T>

$\ell 1_x = 1 \le \ell 2_x = T$ ✓ $\ell 1_z = T \le \ell 2_z = T$ ✓

$\ell 1_y = 2 \le \ell 2_y = 2$ ✓

- E.g.,

<x→1, y→3, z→T> **V** **<x→T, y→2, z→**T> **=**

<T, T, T>

Product Lattice

Does Product Lattice satisfy the ACC?

$\langle x \to T, y \to T \rangle$

|

$\langle x \to T, y \to 1 \rangle$

|

$\langle x \to T, y \to \bot \rangle$

|

$\langle x \to 1, y \to \bot \rangle$

|

$\langle x \to \bot, y \to \bot \rangle$

Length of maximal ascending chain is 2*N where N is the number of variables in the tuple.

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** = **y** *Op* **z**

  f_j: kill: **x** → **val**

  gen:

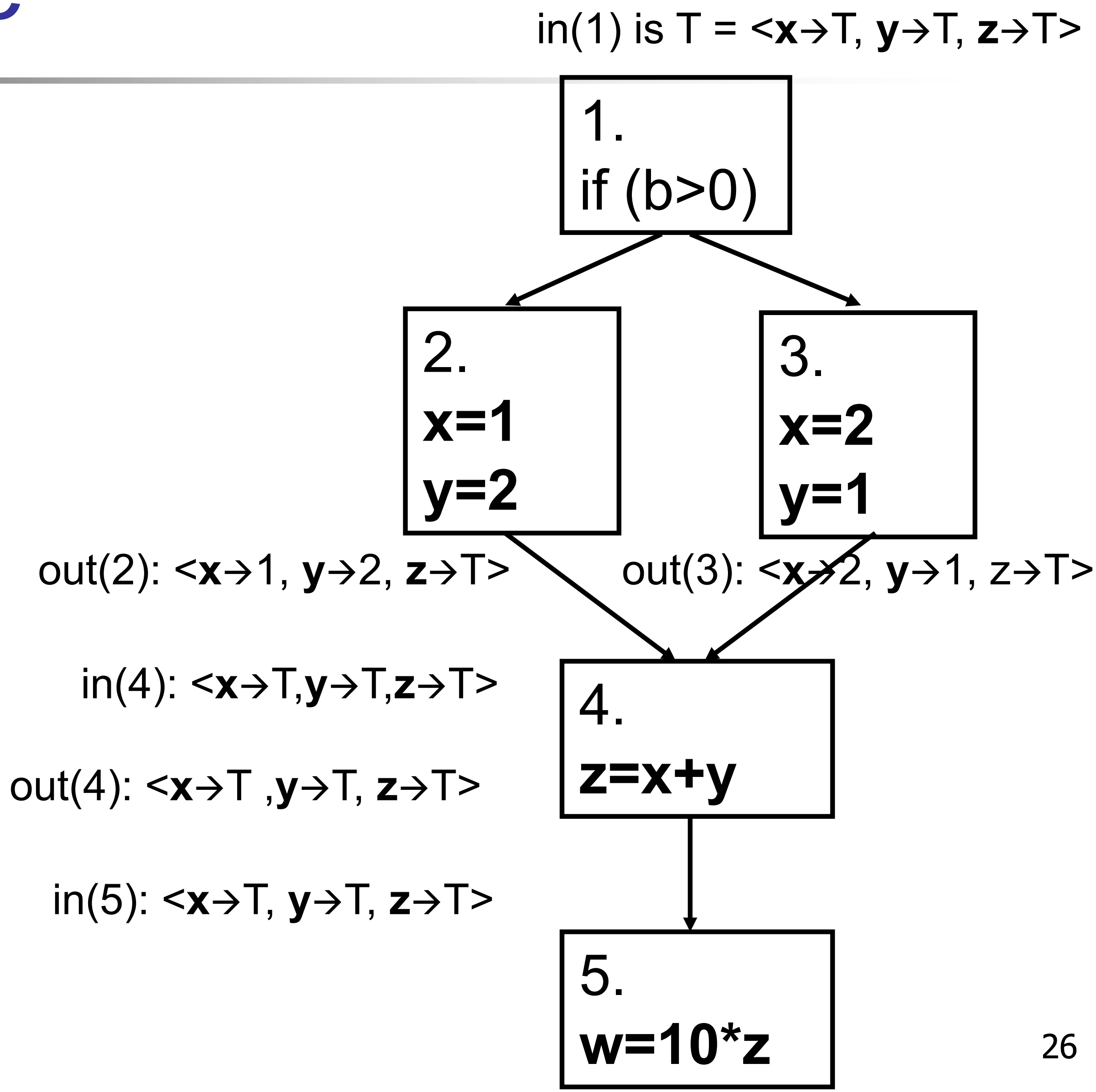
If **y** → c_1 and **z** → c_2 in in(j), then **x** → c_1 *Op* c_2

else if **y** → T or **z** → 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



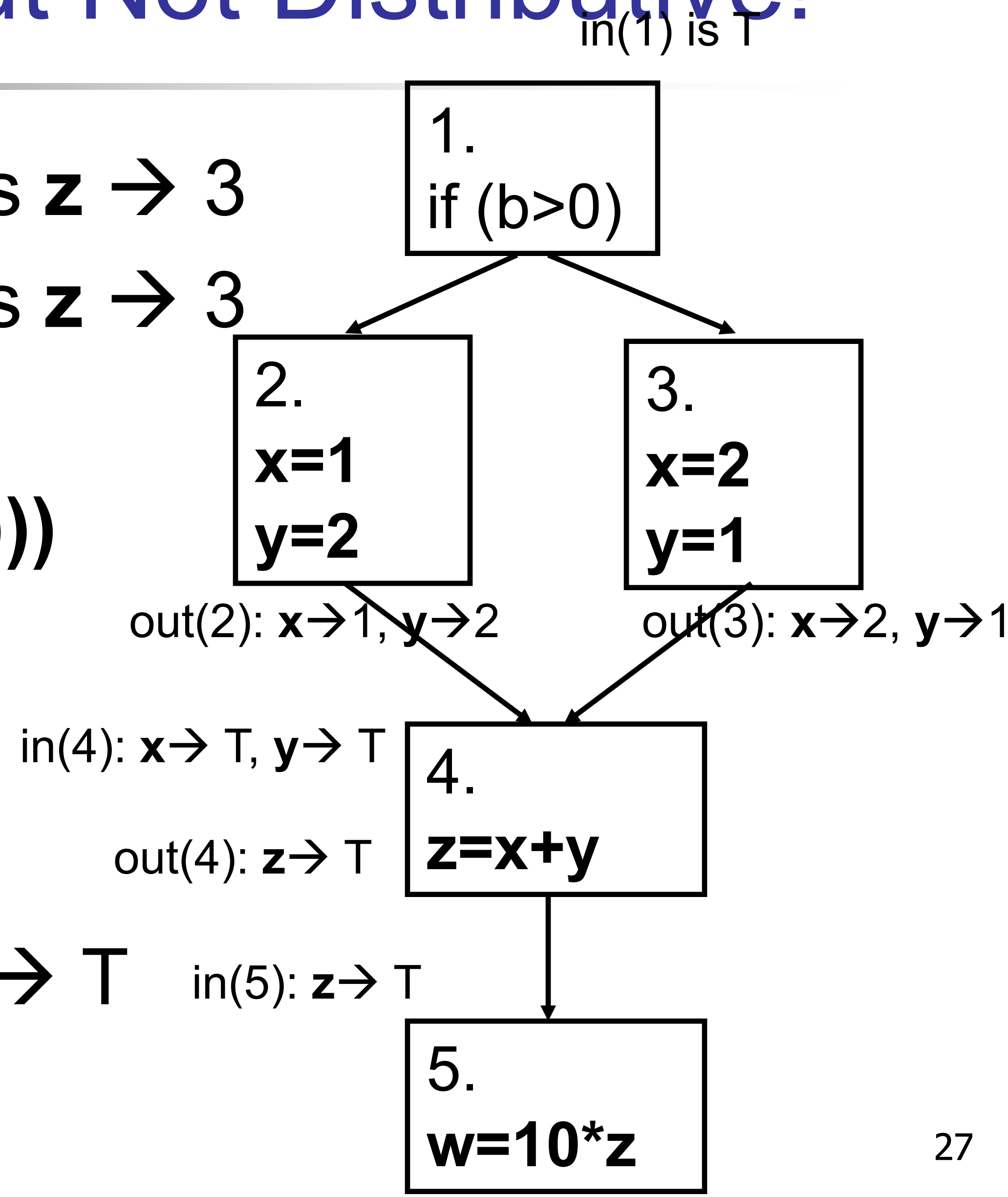
in(1) is T = <x→T, y→T, z→T>
1.
if (b>0)
2.
x=1
y=2
3.
x=2
y=1
out(2): <x→1, y→2, z→T>
out(3): <x→2, y→1, z→T>
in(4): <x→T,y→T,z→T>
4.
z=x+y
out(4): <x→T ,y→T, z→T>
in(5): <x→T, y→T, z→T>
5.
w=10*z

Constant Propagation is Monotone but Not Distributive!

- $\mathbf{f_4(f_2(f_1(}T\mathbf{)))}$ computes **z** → 3
- $\mathbf{f_4(f_3(f_1(}T\mathbf{)))}$ computes **z** → 3
- Thus, MOP at 5

$\mathbf{f_4(f_2(f_1(}T\mathbf{)))} \vee \mathbf{f_4(f_3(f_1(}T\mathbf{)))}$

computes **z** → 3

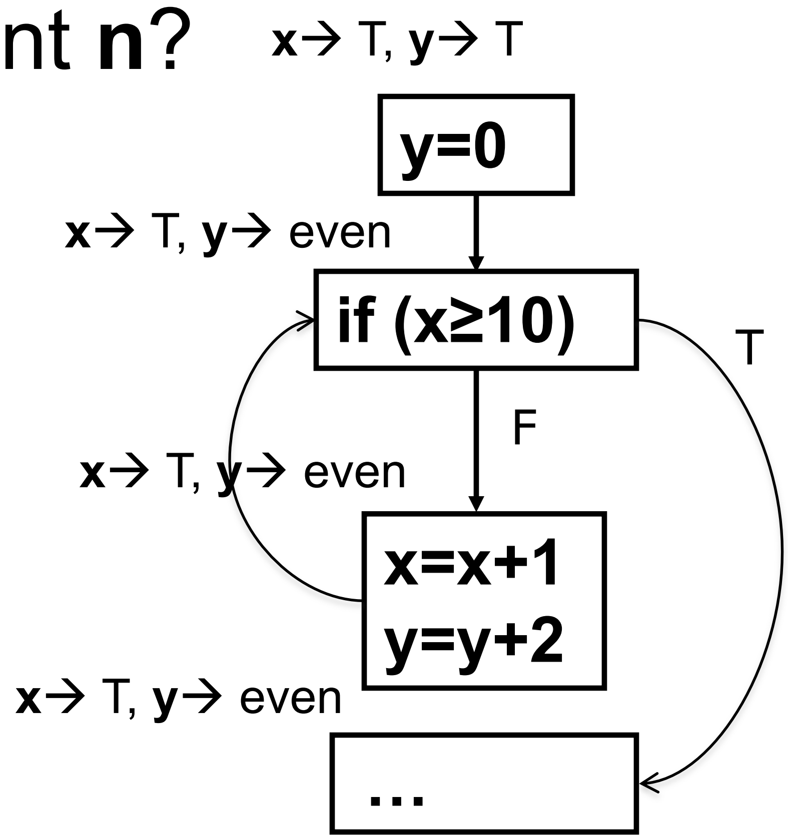


MFP at 5 computes **z** → T

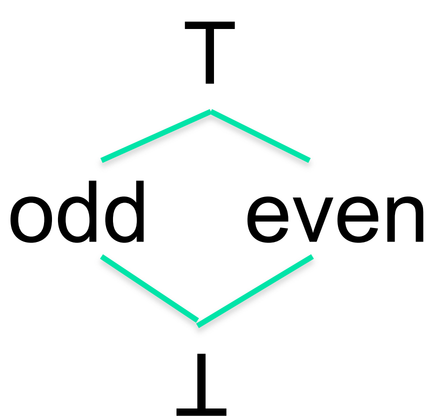
(i.e., **z** is NOT a const)

More Product Lattices

- Problem statement: Is integer variable **x** odd or even at program point **n**?

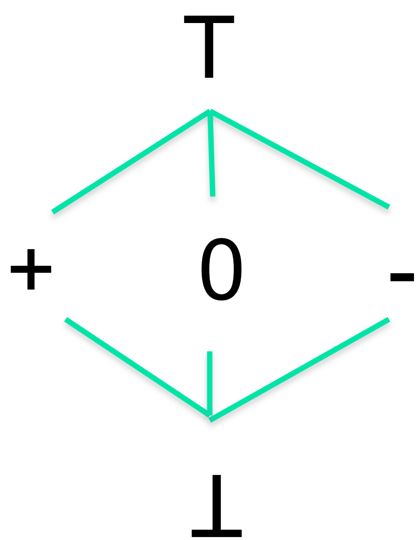


- L_x:



More Product Lattices

- Problem statement: What sign does a variable hold at a given program point, i.e., is it positive, negative, or 0
- $\mathbf{L_x}$:



E.g., **< x→+,y→T,z→0 >**

Outline of Today's Class

- Dataflow frameworks, conclusion
  - Lattices (last time)
  - Transfer functions (last time)
  - Worklist algorithm
- MOP solution vs. MFP solution
- Non-distributive analyses
  - Constant propagation
  - Points-to analysis

Points-to Analysis

- Problem statement: What memory locations may a pointer variable point to?

- Many applications!
  - Enables compiler optimizations

```
1. a = 1;          1. a = x*y*z+x;
2. *p = b;         2. *p = b;
3. s = a*a;        3. s = x*y*z+x;
```

  - Static debugging and taint analysis tools

Points-to Graph: Example

Example 1:

```
int a, b;
int *p1, *p2;
p1 = &a;
p2 = p1;
*p2 = 1;
```

Points-to Graph: Example

Example 2:

```
int a, b = 15;
int *p1, *p2;
int **p3;
p3 = &p1;
p1 = &a;
p2 = *p3;
*p2 = b;
```

Points-to Analysis (for a C-like language)

- Assume the following 4 simple statements

(1) address taken **p = &q**

(2) propagation **p = q**

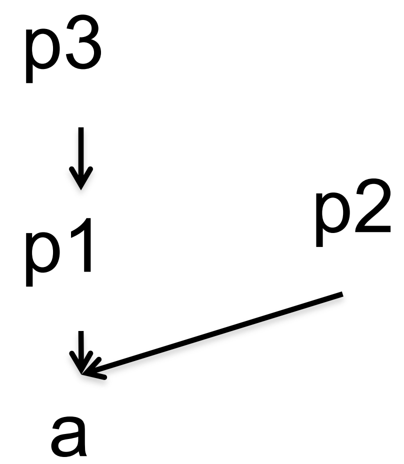
(3) indirect read **p = *q**

(4) indirect write (update) ***p = q**

- We can preprocess any C program into a sequence of statements of these kinds

Points-to Analysis: Property Space

- Lattice **L,≤**
  - Lattice of the subsets over edges **p** ⟶ **q** where **p** and **q** are (names of) memory locations
  - … or in simpler terms, lattice elements are points-to graphs, e.g.,
  - **V** is points-to graph union
  - **0** of L is empty graph
  - **1** of L is complete graph



Points-to Graphs **Pt**

- Nodes are names of memory locations
  - Program variables, a, p:
    - p = &a
  - But also heap locations:
    - p = malloc(sizeof(int)) // h1
- Edges represent points-to relations
  - E.g., p $\longrightarrow$ a, read: “p points to a”
  - E.g., p $\longrightarrow$ h1, read: “p points to heap location h1”

Points-to Analysis: Transfer Functions

(1) $\mathbf{f_{p=\&q}}$: "kill" all points-to edges from **p** and "generate" a new points-to edge from **p** to **q**

(2) $\mathbf{f_{p=q}}$: "kill" all points-to edges from **p**; "generate" new points-to edges from **p** to every **x**, such that **q** points to **x** in incoming points-to graph in(j)

Points-to Analysis: Transfer Functions

(3) $\mathbf{f_{p=*q}}$: "kill" all points-to edges from **p**; "generate" new points-to edges from **p** to every **x**, s.t. there is **y** where **q** points to **y**, and **y** points to **x** in in(j)

(4) $\mathbf{f_{*p=q}}$: Do not kill! Can you think of a reason why? "Generate" new points-to edges from every **y** to every **x**, such that **p** points to **y** and **q** points to **x**

Points-to Analysis is Monotone

- To argue monotonicity we must show that if $\mathbf{Pt_1}$ is **≤** (subset of) $\mathbf{Pt_2}$, then $\mathbf{f(Pt_1) \leq f(Pt_2)}$ for each transfer function **f**

(1) $Pt_1 \leq Pt_2$ then $\mathbf{f_{p=\&q}}(Pt_1) \leq \mathbf{f_{p=\&q}}(Pt_2)$

(2) $Pt_1 \leq Pt_2$ then $\mathbf{f_{p=q}}(Pt_1) \leq \mathbf{f_{p=q}}(Pt_2)$

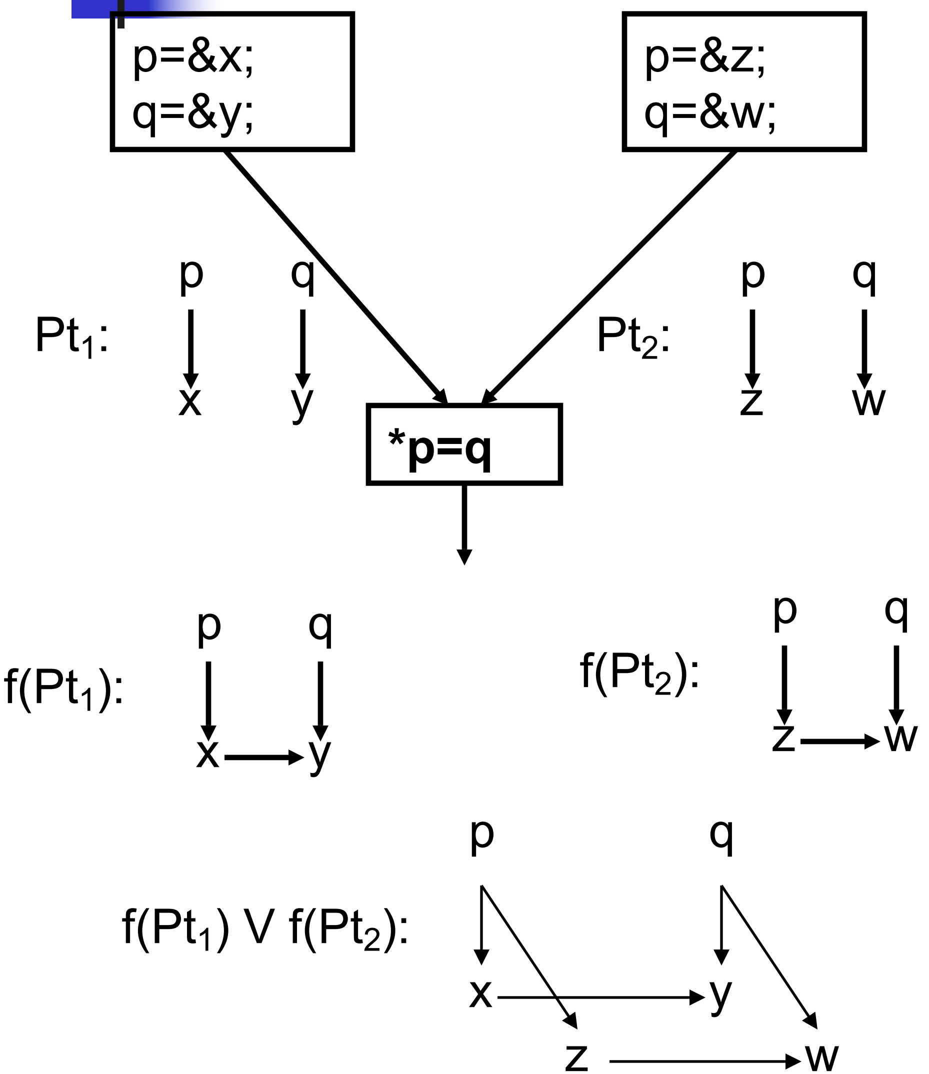
(3) $Pt_1 \leq Pt_2$ then $\mathbf{f_{p=*q}}(Pt_1) \leq \mathbf{f_{p=*q}}(Pt_2)$

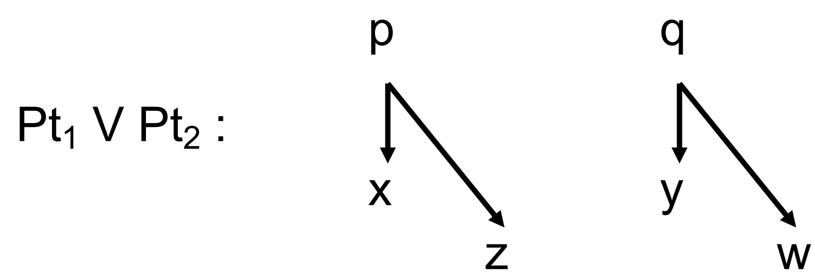
(4) $Pt_1 \leq Pt_2$ then $\mathbf{f_{*p=q}}(Pt_1) \leq \mathbf{f_{*p=q}}(Pt_2)$

… but it is not distributive!

- Because of updates!

Points-to Analysis is Not Distributive

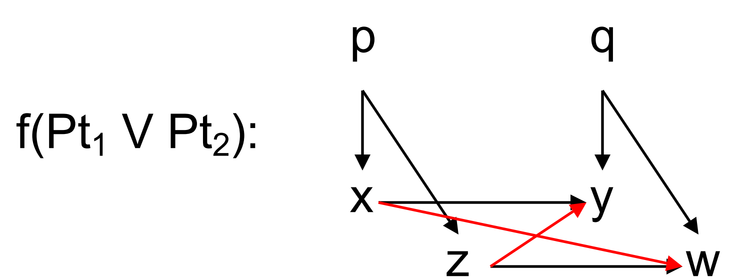




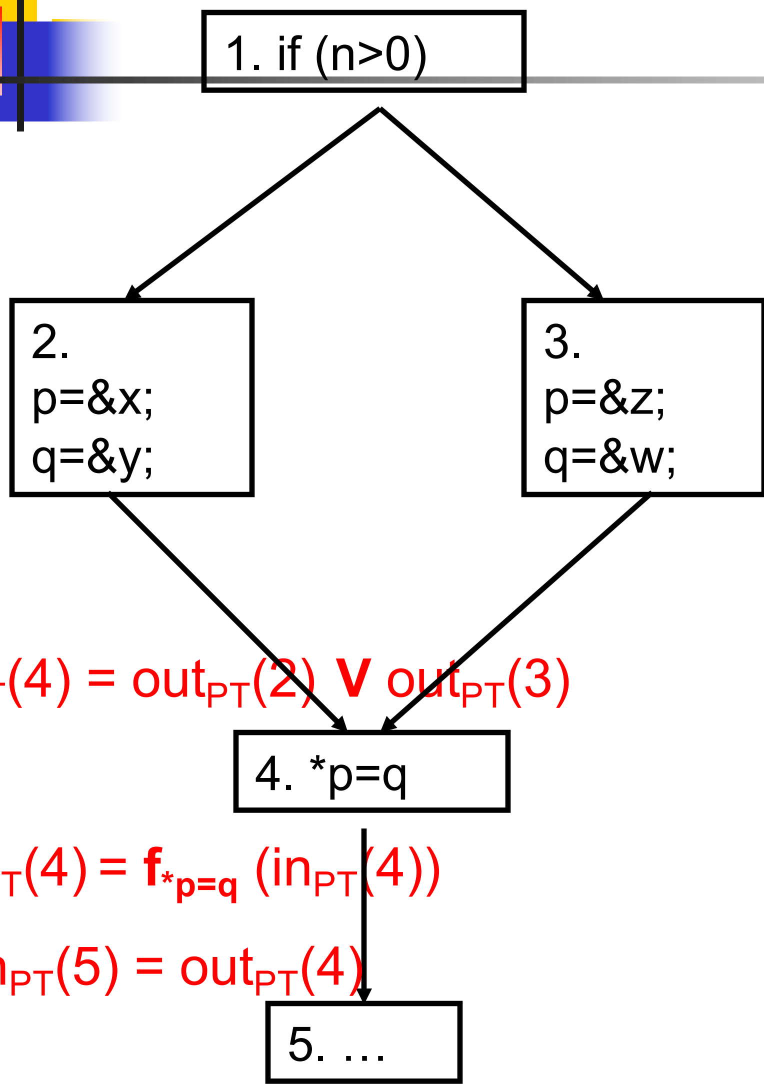
What **f** for ***p = q** does: Adds edges from each variable that **p** points to (**x** and **z**), to each variable that ***q*** points to (**y** and **w**). Result is 4 new edges:

from **x** to **y** and to **w**

and from **z** to **y** and to **w**



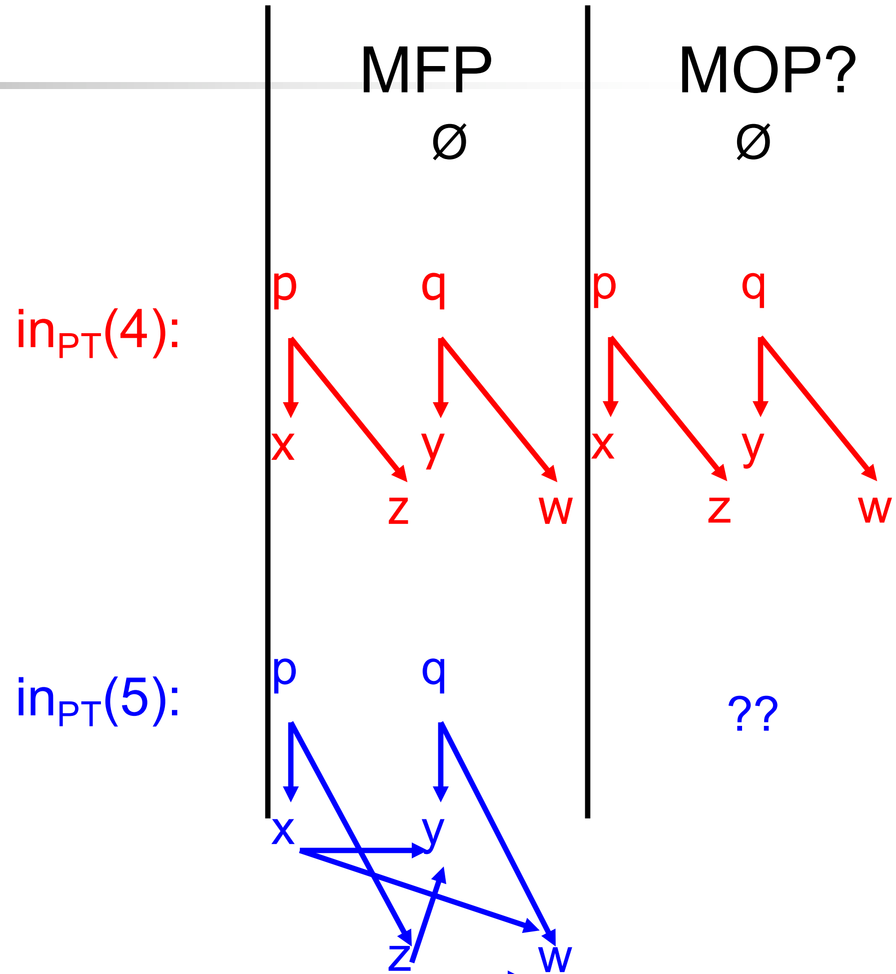
MFP vs. MOP for Points-to



$in_{PT}(4) = out_{PT}(2)$ **V** $out_{PT}(3)$

$out_{PT}(4) = \mathbf{f}_{*p=q}\ (in_{PT}(4))$

$in_{PT}(5) = out_{PT}(4)$



Next Time

- Putting this into practice
- Program analysis frameworks
  - Soot
  - Ghidra