CS 526

Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526

DATAFLOW ANALYSIS

The slides adapted from Martin Rinard and Vikram Adve

Application to Dataflow Analysis

Dataflow information will be lattice values

- **Transfer functions** operate on lattice values
- Solution algorithm will generate **increasing sequence of values** at each program point
- Ascending chain condition will ensure **termination**

We will use $\vee$ to combine values at control-flow join points

Transfer Functions

Transfer function f: P→P for each node in control flow graph models the effect of the node on the program information

Transfer Functions

Each dataflow analysis problem has a **set F of transfer functions** f: $P \rightarrow P$, i.e.,

- **Identity function** belongs to the set, $i \in F$
- F must be **closed under composition**:
  $\forall f,g \in F$. the function $h = \lambda x.f(g(x)) \in F$
- Each $f \in F$ must be **monotone**:
  $x \leq y$ implies $f(x) \leq f(y)$
- Sometimes all $f \in F$ are **distributive**:
  $f(x \vee y) = f(x) \vee f(y)$
- Note that distributivity implies monotonicity

Putting the Pieces Together…

Forward Dataflow Analysis

Simulates execution of program forward with flow of control

For each node n, we have

- in_n – value at program point before n
- out_n – value at program point after n
- f_n – transfer function for n (given in_n, computes out_n)

Requires that solution satisfird

- $\forall n.\ out_n = f_n(in_n)$
- $\forall n \neq n_0.\ in_n = \vee \{\ out_m\ .\ m \text{ in } pred(n)\ \}$
- $in_{n0} = I$, I summarizes information at start of program

Dataflow Equations

Compiler processes program to obtain a set of dataflow equations

$$out_n = f_n(in_n)$$

$$in_n = \vee \{ out_m \mid m \in pred(n) \}$$

Conceptually separates analysis problem from program

Worklist Algorithm for Solving Forward Dataflow Equations

```
for each n do out_n := f_n(⊥)

in_n0 := I; out_n0 := f_n0(I)
worklist := N - { n_0 }

while worklist ≠ ∅ do
      remove a node n from worklist
      in_n  := ∨ { out_m . m in pred(n) }
      out_n := f_n(in_n)
      if out_n changed then
            worklist := worklist ∪ succ(n)
```

Correctness Argument

Why does result satisfy dataflow equations?

- Whenever process a node n, algorithm sets $\texttt{out}_n$:= $\texttt{f}_n(\texttt{in}_n)$
  Therefore, the algorithm ensures that $out_n = f_n(in_n)$
- Whenever out_m changes, put succ(m) on worklist. Consider any node $n \in succ(m)$. It will eventually come off worklist and algorithm will set

  $$in_n := \vee \{ out_m \,.\, m \text{ in } pred(n) \}$$

  to ensure that $in_n = \vee \{ out_m \,.\, m \text{ in } pred(n) \}$
- So final solution will satisfy dataflow equations
- Need also to ensure that the dataflow equalities correspond to the states in the program execution (this comes later!)

Termination Argument

Why does algorithm terminate?

Sequence of values taken on by IN_n or OUT_n is a chain. If values stop increasing, worklist empties and algorithm terminates.

If lattice has ascending chain property, algorithm terminates

- **Algorithm terminates for finite lattices**
- For lattices without ascending chain property, use **widening operator**

Termination Argument (Details)

- For lattice **(L, ≤)**
- Start: each node n∈CFG has an initial IN set, called $IN_0[n]$
- When F is **monotone,** for each n, successive values of IN[n] form a non-decreasing sequence.
  - Any chain starting at x ∈ L has at most c_x elements
  - x=IN[n] can increase in value at most c_x times
  - Then $C = \max_{n \in CFG} c_{IN[n]}$ is finite
- On every iteration, at least one In set must increase in value
  - If loop executes N × C times, all IN sets would be ⊤
  - The algorithm terminates in O(N × C) steps

Speed of Convergence

Loop Connectedness d(G): for a reducible CFG G, it is the maximum number of back edges in any acyclic path in G.

Rapid: A Data-flow framework (L, ≤, F) is called **Rapid** if

$$\forall f \in F, \forall x \in L . \quad x \leq f(x) \wedge f(\top)$$

Kam & Ullman, 1976: Data-flow Framework is Rapid

- The depth-first version of the iterative algorithm halts in at most d(G) + 3 passes over the graph
- If the lattice L has ⊤, at most d(G) + 2 passes are needed

In practice:

- d(G) < 3, so the algorithm makes less than 6 passes over the graph
- The rapid condition implies that the information around the loop stabilizes in 2 steps

Widening Operators

Detect lattice values that may be part of infinitely ascending chain

Artificially raise value to least upper bound of chain

Example:

- Lattice is set of all subsets of integers
- E.g, it can collect possible values of the variables during the execution of program
- Widening operator might raise all sets of size n or greater to TOP (likely to be useful for loops)

Reaching Definitions Algorithm

(Reminder)

```
for all nodes n in N
    OUT[n] = emptyset; // OUT[n] = GEN[n];
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
       choose a node n in Changed;
       Changed = Changed - { n };

       IN[n] = emptyset;
       for all nodes p in predecessors(n)
       IN[n] = IN[n] U OUT[p];

       OUT[n] = GEN[n] U (IN[n] - KILL[n]);

       if (OUT[n] changed)
          for all nodes s in successors(n)
          Changed = Changed U { s };
```

General Worklist Algorithm

(Reminder)

```
for each n do out_n := f_n(⊥)

in_n0 := I; out_n0 := f_n0(I)
worklist := N - { n_0 }

while worklist ≠ ∅ do
      remove a node n from worklist
      in_n  := ∨ { out_m . m in pred(n) }
      out_n := f_n(in_n)
      if out_n changed then
            worklist := worklist ∪ succ(n)
```

Reaching Definitions

P = powerset of set of all definitions in program (all subsets of set of definitions in program)

$\vee = \cup$ (order is $\subseteq$)

$\perp = \varnothing$

$I = in_{n0} = \perp$

F = all functions f of the form $f(x) = a \cup (x\text{-}b)$

- b is set of definitions that node kills
- a is set of definitions that node generates

General pattern for many transfer functions

- $f(x) = GEN \cup (x\text{-}KILL)$

Does Reaching Definitions Framework Satisfy Properties?

$\subseteq$ satisfies conditions for $\leq$

- **Reflexivity:** $x \subseteq x$
- **Assymetry:** $x \subseteq y$ and $y \subseteq x$ implies $y = x$
- **Transitivity:** $x \subseteq y$ and $y \subseteq z$ implies $x \subseteq z$

F satisfies transfer function conditions

- **Identity:** $\lambda x.\emptyset \cup (x - \emptyset) = \lambda x.x \in F$
- **Distributivity:** Will show $f(x \cup y) = f(x) \cup f(y)$

$$f(x) \cup f(y) = (a \cup (x - b)) \cup (a \cup (y - b))$$
$$= a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b)$$
$$= f(x \cup y)$$

Does Reaching Definitions Framework Satisfy Properties?

What about composition of F?

Given $f_1(x) = a_1 \cup (x\text{-}b_1)$ and $f_2(x) = a_2 \cup (x\text{-}b_2)$
we must show $f_1(f_2(x))$ can be expressed as $a \cup (x - b)$

$$f_1(f_2(x)) = a_1 \cup ((a_2 \cup (x\text{-}b_2)) - b_1)$$
$$= a_1 \cup ((a_2 - b_1) \cup ((x\text{-}b_2) - b_1))$$
$$= (a_1 \cup (a_2 - b_1)) \cup ((x\text{-}b_2) - b_1))$$
$$= (a_1 \cup (a_2 - b_1)) \cup (x\text{-}(b_2 \cup b_1))$$

- Let $a = (a_1 \cup (a_2 - b_1))$ and $b = b_2 \cup b_1$
- Then $f_1(f_2(x)) = a \cup (x - b)$

Reaching Definitions is Rapid

Convergence Is Fast

$$
\begin{aligned}
f(x) &\overset{?}{\geq} x \wedge f(\top) \\
a_f \cup (x \cap b_f) &\overset{?}{\geq} x \cap (a_f \cup (\top \cap b_f)) \\
a_f \cup (x \cap b_f) &\overset{?}{\geq} x \cap (a_f \cup b_f) \\
a_f \cup (x \cap b_f) &\overset{?}{\geq} (x \cap a_f) \cup (x \cap b_f)
\end{aligned}
$$

$$
\begin{aligned}
a_f &\geq x \cap a_f \\
x \cap b_f &= x \cap b_f
\end{aligned}
$$

General Result

All GEN/KILL transfer function frameworks satisfy the three properties:

- Identity
- Distributivity
- Composition

And all of them converge rapidly

Available Expressions

P = powerset of set of all expressions in program (all subsets of set of expressions)

$\vee = \cap$ (order is $\supseteq$)

$\perp = P$

$I = in_{n0} = \varnothing$

F = all functions f of the form $f(x) = a \cup (x\text{-}b)$

- b is set of expressions that node kills
- a is set of expressions that node generates

Another GEN/KILL analysis

Concept of Conservatism

Reaching definitions use $\cup$ as join

- Optimizations must take into account all definitions that reach along **ANY path**

Available expressions use $\cap$ as join

- Optimization requires expression to be available along **ALL paths**

Optimizations must **conservatively take all possible executions into account.**

Backward Dataflow Analysis

- Simulates execution of program backward against the flow of control
- For each node n, we have
  - in_n – value at program point before n
  - out_n – value at program point after n
  - f_n – transfer function for n (given out_n, computes in_n)
- Require that solution satisfies
  - $\forall n.\ in_n = f_n(out_n)$
  - $\forall n \notin N_{final}.\ out_n = \vee \{\ in_m\ .\ m \text{ in succ}(n)\ \}$
  - $\forall n \in N_{final} = out_n = O$
  - Where O summarizes information at end of program

Worklist Algorithm for Solving Backward Dataflow Equations

```
for each n do in_n := f_n(⊥)
for each n ∈ N_final do out_n := ⊥; in_n := f_n(⊥)
worklist := N - N_final

while worklist ≠ ∅ do
   remove a node n from worklist
   out_n := ∨ { in_m . m in succ(n) }
   in_n := f_n(out_n)
   if in_n changed then
       worklist := worklist ∪ pred(n)
```

Live Variables

P = powerset of set of all variables in program (all subsets of set of variables in program)

$\vee = \cup$ (order is $\subseteq$)

$\bot = \varnothing$

$\mathbf{O} = \varnothing$

F = all functions f of the form $f(x) = a \cup (x-b)$

- b is set of variables that node kills
- a is set of variables that node reads

Meaning of Dataflow Results

Concept of **program state s** for control-flow graphs

- **Program point n** where execution is located (n is node that will execute next)
- Values of variables in program

Each execution generates a trajectory of states:

- $s_0;s_1;\ldots;s_k$, where each $s_i \in S$
- s_{i+1} generated from s_i by executing basic block to
  1. Update variable values
  2. Obtain new program point n

Relating States to Analysis Result

- Meaning of analysis results is given by an abstraction function $AF:ST \rightarrow P$

- Correctness condition: require that for all states s

$$AF(s) \leq in_n$$

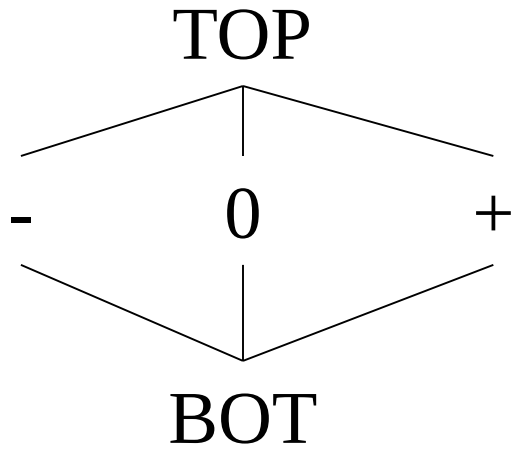
where n is the next statement to execute in state s

[See e.g., Nielson; Nielson; Hankin. Principles of Program Analysis (2004), for full formal treatment (with operational semantics)]

Sign Analysis Example

Sign analysis - compute sign of each variable v

Base Lattice: P = flat lattice on {-,0,+}



Actual lattice records a value for each variable

- Example element: [a→+, b→0, c→-]

Interpretation of Lattice Values

If value of v in lattice is:

- BOT: no information about the sign of v
- -: variable v is negative
- 0: variable v is 0
- +: variable v is positive
- TOP: v may be positive or negative or zero

What is abstraction function AF?

- AF($[v_1,\ldots,v_n]$) = [sign(v_1), …, sign(v_n)]
- $\text{sign}(x) = \begin{cases} 0 \text{ if } v = 0 \\ + \text{ if } v > 0 \\ - \text{ if } v < 0 \end{cases}$

Transfer Functions

If n of the form $v = c$

- $f_n(x) = x[v \rightarrow +]$ if c is positive
- $f_n(x) = x[v \rightarrow 0]$ if c is 0
- $f_n(x) = x[v \rightarrow -]$ if c is negative

If n of the form $v_1 = v_2 * v_3$

- $f_n(x) = x[v_1 \rightarrow x[v_2] \otimes x[v_3]]$

I = TOP (uninitialized variables may have any sign)

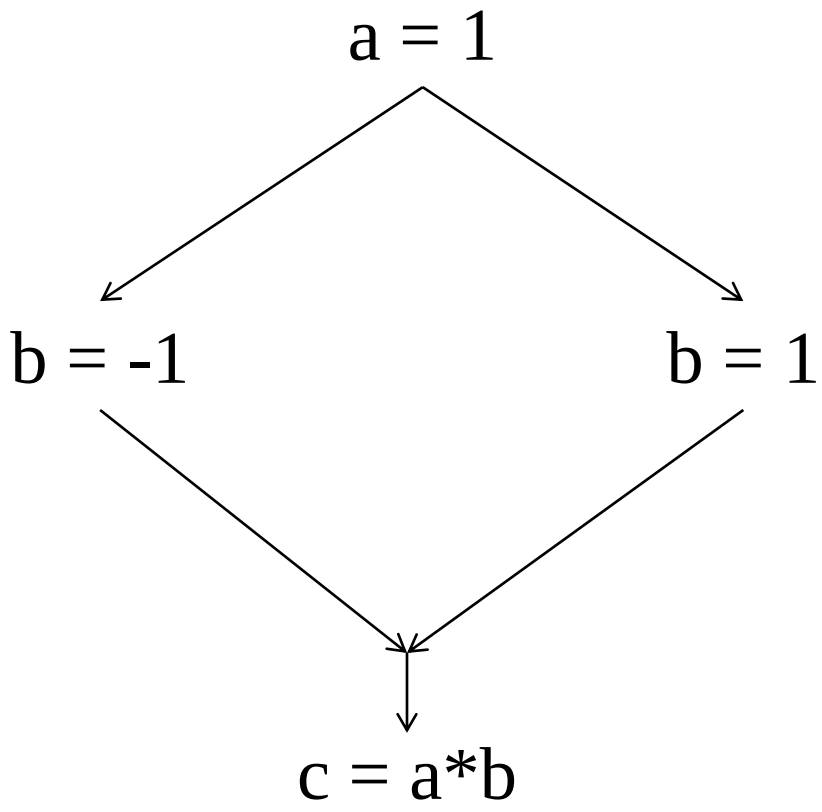
Operation ⊗ on Lattice

⊗	BOT	-	0	+	TOP
BOT					
-					
0					
+					
TOP					

Operation ⊗ on Lattice

⊗	BOT	-	0	+	TOP
BOT	BOT	BOT	0	BOT	BOT
-	BOT	+	0	-	TOP
0	0	0	0	0	0
+	BOT	-	0	+	TOP
TOP	BOT	TOP	0	TOP	TOP

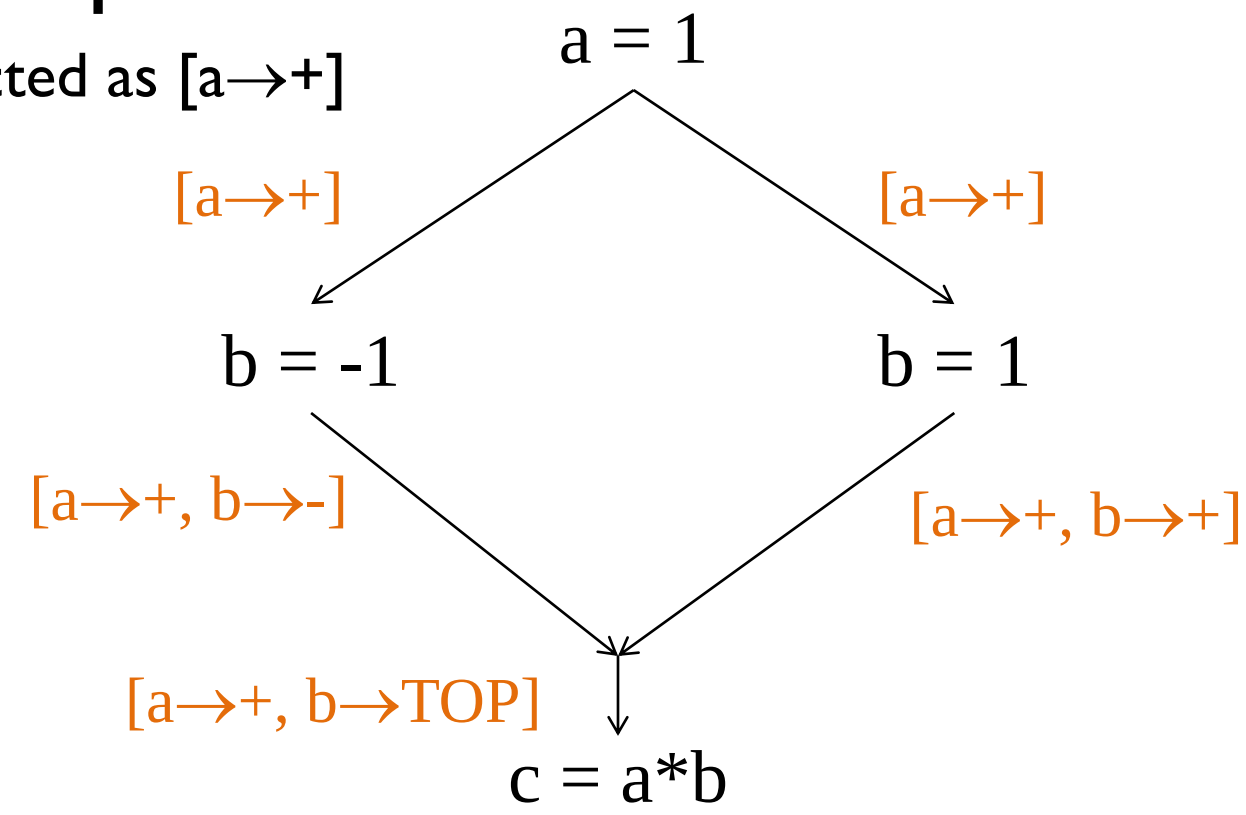
Sign Analysis Example



Imprecision In Example

Abstraction Imprecision:

[a→1] abstracted as [a→+]



Control Flow Imprecision:

[b→TOP] summarizes results of all executions.

(In any concrete execution state s, AF(s)[b]≠TOP)

General Sources of Imprecision

Abstraction Imprecision

- Concrete values (integers) abstracted as lattice values (-,0, and +)
- Lattice values less precise than execution values
- Abstraction function throws away information

Control Flow Imprecision

- One lattice value for all possible control flow paths
- Analysis result has a single lattice value to summarize results of multiple concrete executions
- Join operation $\vee$ moves up in lattice to combine values from different execution paths
- Typically if $x \leq y$, then x is more precise than y

Why To Allow Imprecision?

Make analysis tractable

Unbounded sets of values in execution

- Typically abstracted by finite set of lattice values

Execution may visit unbounded set of states

- Abstracted by computing joins of different paths

Correctness of Solution

Correctness condition:

- $\forall$ v . AF(s)[v] $\leq$ in_n[v] (n is node, s is state)
- Reflects possibility of imprecision

Proof:

- By the induction on the structure of the computation that produces s

Meet Over Paths* Solution

What solution would be ideal for a forward dataflow problem?

Consider a path p = $\mathbf{n_0}, n_1, \ldots, n_k, \mathbf{n}$ to a node n
(note that for all i, $n_i \in pred(n_{i+1})$)

The solution must take this path into account:

$f_p(\bot) = (f_{nk}(f_{nk-1}(\ldots f_{n1}(f_{n0}(\bot)) \ldots)) \leq in_n$

So the solution must have the property that

$$\vee\{f_p(\bot) \,.\, p \text{ is a path to } n\} \leq in_n$$

and ideally

$$\vee\{f_p(\bot) \,.\, p \text{ is a path to } n\} = in_n$$

* Name exists for historical reasons; this is really a join

Soundness Proof of Analysis Algorithm

Property to prove: For all paths p to n, $f_p(\bot) \leq in_n$

- Proof is by induction on length of p
- Uses monotonicity of transfer function

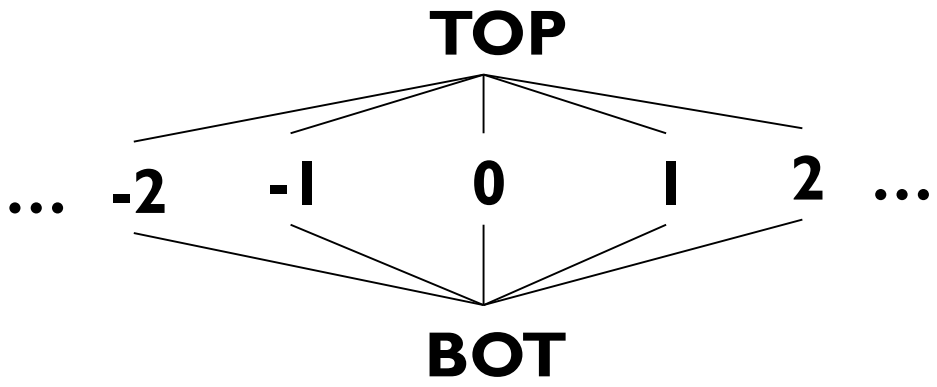
Connections between MOP and worklist solution:

- [Kildall, 1973] The iterative worklist algorithm: (1) converges and (2) computes a MFP (maximum fixed point) solution of the set of equations using the worklist algorithm
- [Kildall, 1973] **If F is distributive, MOP = MFP**

  $\vee\{f_p(\bot)$. p is a path to n$\} = in_n$
- [Kam & Ullman, 1977] If F is monotone, MOP $\leq$ MFP

Lack of Distributivity Example

Constant Calculator: Flat Lattice on Integers



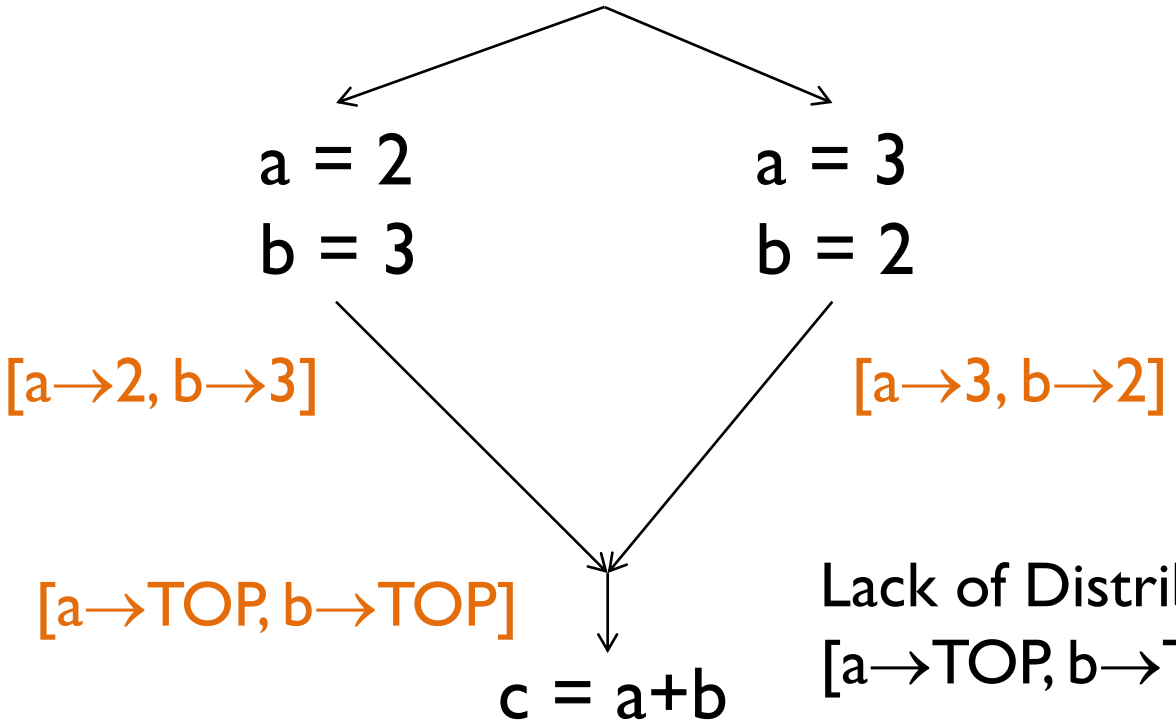
Actual lattice records a value for each variable

- Example element: [a→3, b→2, c→5]

Transfer function:

- If n of the form v = c, then $f_n(x) = x[v \rightarrow c]$
- If n of the form $v_1 = v_2 + v_3$, $f_n(x) = x[v_1 \rightarrow x[v_2] + x[v_3]]$

Lack of Distributivity Anomaly

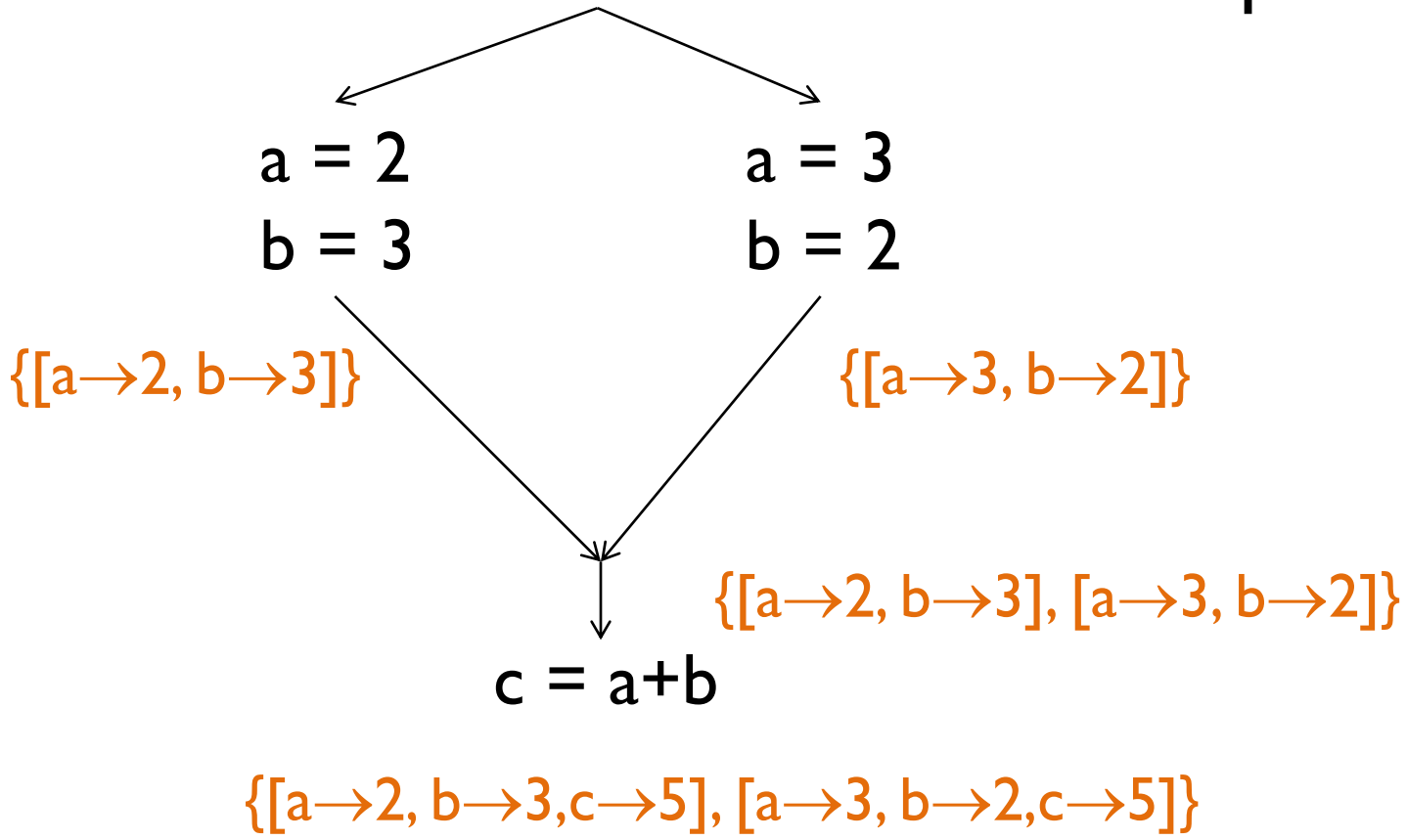


Lack of Distributivity Imprecision:
[a→TOP, b→TOP, c→5] more precise

What is the meet over all paths solution?

Make Analysis Distributive

Keep combinations of values on different paths



Discussion of the Solution

It basically simulates **all combinations** of values in **all executions**

- Exponential blowup
- Nontermination because of infinite ascending chains

Terminating solution:

- Use widening operator to eliminate blowup
  (can make it work at granularity of variables)
- However, loses precision in many cases
- Not trivial to select optimal point to do widening

Look Forward

We will return to these problems later in the semester

- **Interprocedural analysis:** how to handle function calls and global variables in the analysis?
- **Abstract interpretation:** how to automate analysis with infinite chains and rich abstract domains?

Additional readings:

- Long comparison: Flemming Nielson; Hanne R. Nielson; Chris Hankin. Principles of Program Analysis (2004). Springer. (available online)
- Short comparison: Wolfgang Woegerer. A Survey of Static Program Analysis Techniques (available online)