

6.110 Computer Language Engineering

Re-lecture 5

April 17, 2024

MIT 6.1100

Foundations of Dataflow Analysis

Martin Rinard

Massachusetts Institute of Technology

Dataflow Analysis

- Compile-Time Reasoning About
- Run-Time Values of Variables or Expressions
- At Different Program Points
 - Which assignment statements produced value of variable at this point?
 - Which variables contain values that are no longer used after this program point?
 - What is the range of possible values of variable at this program point?

Program Representation

- Control Flow Graph
 - Nodes N – statements of program
 - Edges E – flow of control
 - $\text{pred}(n)$ = set of all predecessors of n
 - $\text{succ}(n)$ = set of all successors of n
 - Start node n_0
 - Set of final nodes N_{final}

Program Points

- One program point before each node
- One program point after each node
- Join point – point with multiple predecessors
- Split point – point with multiple successors

Basic Idea

- Information about program represented using values from algebraic structure called lattice
- Analysis produces lattice value for each program point
- Two flavors of analysis
 - Forward dataflow analysis
 - Backward dataflow analysis

Forward Dataflow Analysis

- Analysis propagates values forward through control flow graph with flow of control
 - Each node has a transfer function f
 - Input – value at program point before node
 - Output – new value at program point after node
 - Values flow from program points after predecessor nodes to program points before successor nodes
 - At join points, values are combined using a merge function
- Canonical Example: Reaching Definitions

Backward Dataflow Analysis

- Analysis propagates values backward through control flow graph against flow of control
 - Each node has a transfer function f
 - Input – value at program point after node
 - Output – new value at program point before node
 - Values flow from program points before successor nodes to program points after predecessor nodes
 - At split points, values are combined using a merge function
 - Canonical Example: Live Variables

Summary

- Dataflow analysis computes some *information* (say, of type \mathbf{I}) at each statement (or basic block)
- Each statement has a transfer function $\mathbf{f}: \mathbf{I} \rightarrow \mathbf{I}$
 - Given what information we have at the program point before, and what is at the statement, what information do we have at the program point after?
- At each merge points, we combine information from the paths using a *join* function $\mathbf{v}: \mathbf{I} \times \mathbf{I} \rightarrow \mathbf{I}$
- Lattices are a way to formalize all this and prove that a dataflow analysis always terminates (assuming some properties of \mathbf{I} , \mathbf{f} and \mathbf{v})

Partial Orders

- Set P
- Partial order \leq such that $\forall x, y, z \in P$
 - $x \leq x$ (reflexive)
 - $x \leq y$ and $y \leq x$ implies $x = y$ (asymmetric)
 - $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitive)
- Can use partial order to define
 - Upper and lower bounds
 - Least upper bound
 - Greatest lower bound

Upper Bounds

- If $S \subseteq P$ then
 - $x \in P$ is an upper bound of S if $\forall y \in S. y \leq x$
 - $x \in P$ is the least upper bound of S if
 - x is an upper bound of S , and
 - $x \leq y$ for all upper bounds y of S
 - \vee - join, least upper bound, lub, supremum, sup
 - $\vee S$ is the least upper bound of S
 - $x \vee y$ is the least upper bound of $\{x, y\}$

Lower Bounds

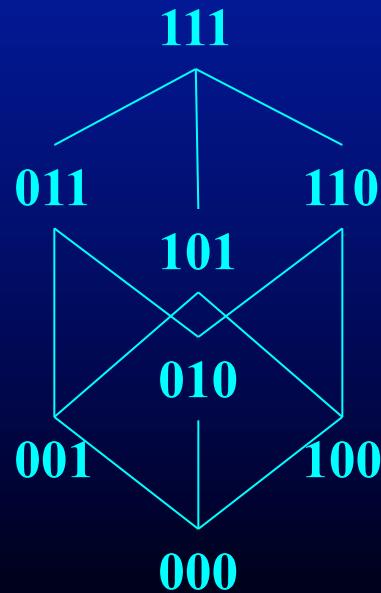
- If $S \subseteq P$ then
 - $x \in P$ is a lower bound of S if $\forall y \in S. x \leq y$
 - $x \in P$ is the greatest lower bound of S if
 - x is a lower bound of S , and
 - $y \leq x$ for all lower bounds y of S
 - \wedge - meet, greatest lower bound, glb, infimum, inf
 - $\wedge S$ is the greatest lower bound of S
 - $x \wedge y$ is the greatest lower bound of $\{x, y\}$

Covering

- $x < y$ if $x \leq y$ and $x \neq y$
- x is covered by y (y covers x) if
 - $x < y$, and
 - $x \leq z < y$ implies $x = z$
- Conceptually, y covers x if there are no elements between x and y

Example

- $P = \{ 000, 001, 010, 011, 100, 101, 110, 111 \}$
(standard boolean lattice, also called hypercube)
- $x \leq y$ if $(x \text{ bitwise and } y) = x$



Hasse Diagram

- If y covers x
 - Line from y to x
 - y above x in diagram

Lattices

- If $x \wedge y$ and $x \vee y$ exist for all $x, y \in P$, then P is a lattice.
- If $\wedge S$ and $\vee S$ exist for all $S \subseteq P$, then P is a complete lattice.
- All finite lattices are complete

Lattices

- If $x \wedge y$ and $x \vee y$ exist for all $x, y \in P$, then P is a lattice.
- If $\wedge S$ and $\vee S$ exist for all $S \subseteq P$, then P is a complete lattice.
- All finite lattices are complete
- Example of a lattice that is not complete
 - Integers I
 - For any $x, y \in I$, $x \vee y = \max(x, y)$, $x \wedge y = \min(x, y)$
 - But $\vee I$ and $\wedge I$ do not exist
 - $I \cup \{+\infty, -\infty\}$ is a complete lattice

Top and Bottom

- Greatest element of P (if it exists) is top
- Least element of P (if it exists) is bottom (\perp)

Connection Between \leq , \wedge , and \vee

- The following 3 properties are equivalent:

- $x \leq y$

- $x \vee y = y$

- $x \wedge y = x$

- Will prove:

- $x \leq y$ implies $x \vee y = y$ and $x \wedge y = x$

- $x \vee y = y$ implies $x \leq y$

- $x \wedge y = x$ implies $x \leq y$

- Then by transitivity, can obtain

- $x \vee y = y$ implies $x \wedge y = x$

- $x \wedge y = x$ implies $x \vee y = y$

Connecting Lemma Proofs

- Proof of $x \leq y$ implies $x \vee y = y$
 - $x \leq y$ implies y is an upper bound of $\{x, y\}$.
 - Any upper bound z of $\{x, y\}$ must satisfy $y \leq z$.
 - So y is least upper bound of $\{x, y\}$ and $x \vee y = y$
- Proof of $x \leq y$ implies $x \wedge y = x$
 - $x \leq y$ implies x is a lower bound of $\{x, y\}$.
 - Any lower bound z of $\{x, y\}$ must satisfy $z \leq x$.
 - So x is greatest lower bound of $\{x, y\}$ and $x \wedge y = x$

Connecting Lemma Proofs

- Proof of $x \vee y = y$ implies $x \leq y$
 - y is an upper bound of $\{x,y\}$ implies $x \leq y$
- Proof of $x \wedge y = x$ implies $x \leq y$
 - x is a lower bound of $\{x,y\}$ implies $x \leq y$

Lattices as Algebraic Structures

- Have defined \vee and \wedge in terms of \leq
- Will now define \leq in terms of \vee and \wedge
 - Start with \vee and \wedge as arbitrary algebraic operations that satisfy associative, commutative, idempotence, and absorption laws
 - Will define \leq using \vee and \wedge
 - Will show that \leq is a partial order
- Intuitive concept of \vee and \wedge as information combination operators (or, and)

Algebraic Properties of Lattices

Assume arbitrary operations \vee and \wedge such that

- $(x \vee y) \vee z = x \vee (y \vee z)$ (associativity of \vee)
- $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ (associativity of \wedge)
- $x \vee y = y \vee x$ (commutativity of \vee)
- $x \wedge y = y \wedge x$ (commutativity of \wedge)
- $x \vee x = x$ (idempotence of \vee)
- $x \wedge x = x$ (idempotence of \wedge)
- $x \vee (x \wedge y) = x$ (absorption of \vee over \wedge)
- $x \wedge (x \vee y) = x$ (absorption of \wedge over \vee)

Connection Between \wedge and \vee

- $x \vee y = y$ if and only if $x \wedge y = x$
- Proof of $x \vee y = y$ implies $x = x \wedge y$
$$\begin{aligned} x &= x \wedge (x \vee y) && \text{(by absorption)} \\ &= x \wedge y && \text{(by assumption)} \end{aligned}$$
- Proof of $x \wedge y = x$ implies $y = x \vee y$
$$\begin{aligned} y &= y \vee (y \wedge x) && \text{(by absorption)} \\ &= y \vee (x \wedge y) && \text{(by commutativity)} \\ &= y \vee x && \text{(by assumption)} \\ &= x \vee y && \text{(by commutativity)} \end{aligned}$$

Properties of \leq

- Define $x \leq y$ if $x \vee y = y$
- Proof of transitive property. Must show that

$x \vee y = y$ and $y \vee z = z$ implies $x \vee z = z$

$$x \vee z = x \vee (y \vee z) \text{ (by assumption)}$$

$$= (x \vee y) \vee z \text{ (by associativity)}$$

$$= y \vee z \text{ (by assumption)}$$

$$= z \text{ (by assumption)}$$

Properties of \leq

- Proof of asymmetry property. Must show that

$x \vee y = y$ and $y \vee x = x$ implies $x = y$

$$x = y \vee x \quad (\text{by assumption})$$

$$= x \vee y \quad (\text{by commutativity})$$

$$= y \quad (\text{by assumption})$$

- Proof of reflexivity property. Must show that

$$x \vee x = x$$

$$x \vee x = x \quad (\text{by idempotence})$$

Properties of \leq

- Induced operation \leq agrees with original definitions of \vee and \wedge , i.e.,
 - $x \vee y = \sup \{x, y\}$
 - $x \wedge y = \inf \{x, y\}$

Proof of $x \vee y = \sup \{x, y\}$

- Consider any upper bound u for x and y .
- Given $x \vee u = u$ and $y \vee u = u$, must show $x \vee y \leq u$, i.e., $(x \vee y) \vee u = u$

$$\begin{aligned} u &= x \vee u && \text{(by assumption)} \\ &= x \vee (y \vee u) && \text{(by assumption)} \\ &= (x \vee y) \vee u && \text{(by associativity)} \end{aligned}$$

Proof of $x \wedge y = \inf \{x, y\}$

- Consider any lower bound l for x and y .
- Given $x \wedge l = l$ and $y \wedge l = l$, must show $l \leq x \wedge y$, i.e., $(x \wedge y) \wedge l = l$

$$\begin{aligned} l &= x \wedge l && \text{(by assumption)} \\ &= x \wedge (y \wedge l) && \text{(by assumption)} \\ &= (x \wedge y) \wedge l && \text{(by associativity)} \end{aligned}$$

Chains

- A set S is a chain if $\forall x, y \in S. y \leq x$ or $x \leq y$
- P has no infinite chains if every chain in P is finite
- P satisfies the ascending chain condition if for all sequences $x_1 \leq x_2 \leq \dots$ there exists n such that $x_n = x_{n+1} = \dots$

For the quiz, you should know:

- Definition of posets, lattices
- Properties of lattices
 - Operations: \leq , \wedge , \vee
 - Lower/upper bounds, top \top , bottom \perp
 - Algebraic properties
 - Completeness

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
- Will use \vee to combine values at control-flow join points

Transfer Functions

- Transfer function $f: P \rightarrow P$ for each node in control flow graph
- f models effect of the node on the program information

Transfer Functions

Each dataflow analysis problem has a set F of transfer functions $f: P \rightarrow P$

- Identity function $i \in F$
- F must be closed under composition:
 $\forall f, g \in F. \text{ the function } h = \lambda x. f(g(x)) \in F$
- Each $f \in F$ must be monotone:
 $x \leq y \text{ implies } f(x) \leq f(y)$
- Sometimes all $f \in F$ are distributive:
 $f(x \vee y) = f(x) \vee f(y)$
- Distributivity implies monotonicity

Distributivity Implies Monotonicity

- Proof of distributivity implies monotonicity
- Assume $f(x \vee y) = f(x) \vee f(y)$
- Must show: $x \vee y = y$ implies $f(x) \vee f(y) = f(y)$

$$\begin{aligned} f(y) &= f(x \vee y) && \text{(by assumption)} \\ &= f(x) \vee f(y) && \text{(by distributivity)} \end{aligned}$$

Putting Pieces Together

- Forward Dataflow Analysis Framework
- Simulates execution of program forward with flow of control

Forward Dataflow Analysis

- Simulates execution of program forward with flow of control
- For each node n , 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)
- Require that solution satisfy
 - $\forall n. out_n = f_n(in_n)$
 - $\forall n \neq n_0. in_n = \vee \{ out_m . m \text{ in } pred(n) \}$
 - $in_{n_0} = I$
 - Where I summarizes information at start of program

Dataflow Equations

- Compiler processes program to obtain a set of dataflow equations

$$\text{out}_n := f_n(\text{in}_n)$$

$$\text{in}_n := \vee \{ \text{out}_m . m \text{ in pred}(n) \}$$

- Conceptually separates analysis problem from program

Worklist Algorithm for Solving Forward Dataflow Equations

```
for each n do  $out_n := f_n(\perp)$   
 $in_{n_0} := I$ ;  $out_{n_0} := f_{n_0}(I)$   
 $worklist := N - \{ n_0 \}$   
while  $worklist \neq \emptyset$  do  
    remove a node n from worklist  
     $in_n := \vee \{ out_m . m \text{ in } pred(n) \}$   
     $out_n := f_n(in_n)$   
    if  $out_n$  changed then  
         $worklist := worklist \cup succ(n)$ 
```

Correctness Argument

- Why result satisfies dataflow equations
- Whenever process a node n , set $out_n := f_n(in_n)$
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

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

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
 - Could be used to collect possible values taken on by variable during execution of program
 - Widening operator might raise all sets of size n or greater to TOP (likely to be useful for loops)

Reaching Definitions

- P = powerset of set of all definitions in program (all subsets of set of definitions in program)
- $\vee = \cup$ (order is \subseteq)
- $\perp = \emptyset$
- $I = \text{in}_{n0} = \perp$
- F = all functions f of the form $f(x) = a \cup (x-b)$
 - b is set of definitions that node kills
 - a is set of definitions that node generates
- General pattern for many transfer functions
 - $f(x) = \text{GEN} \cup (x-\text{KILL})$

Does Reaching Definitions Framework Satisfy Properties?

- \subseteq satisfies conditions for \leq
 - $x \subseteq y$ and $y \subseteq z$ implies $x \subseteq z$ (transitivity)
 - $x \subseteq y$ and $y \subseteq x$ implies $y = x$ (asymmetry)
 - $x \subseteq x$ (reflexive)
- F satisfies transfer function conditions
 - $\lambda x. \emptyset \cup (x - \emptyset) = \lambda x. x \in F$ (identity)
 - Will show $f(x \cup y) = f(x) \cup f(y)$ (distributivity)
 - $$\begin{aligned} f(x) \cup f(y) &= (a \cup (x - b)) \cup (a \cup (y - b)) \\ &= a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b) \\ &= f(x \cup y) \end{aligned}$$

Does Reaching Definitions Framework Satisfy Properties?

- What about composition?
 - Given $f_1(x) = a_1 \cup (x - b_1)$ and $f_2(x) = a_2 \cup (x - b_2)$
 - Must show $f_1(f_2(x))$ can be expressed as $a \cup (x - b)$
$$\begin{aligned}f_1(f_2(x)) &= a_1 \cup ((a_2 \cup (x - b_2)) - b_1) \\&= a_1 \cup ((a_2 - b_1) \cup ((x - b_2) - b_1)) \\&= (a_1 \cup (a_2 - b_1)) \cup ((x - b_2) - b_1) \\&= (a_1 \cup (a_2 - b_1)) \cup (x - (b_2 \cup b_1))\end{aligned}$$
 - Let $a = (a_1 \cup (a_2 - b_1))$ and $b = b_2 \cup b_1$
 - Then $f_1(f_2(x)) = a \cup (x - b)$

General Result

All GEN/KILL transfer function frameworks satisfy

- Identity
- Distributivity
- Composition

Properties

Available Expressions

- P = powerset of set of all expressions in program (all subsets of set of expressions)
- $\vee = \cap$ (order is \supseteq)
- $\perp = P$
- $I = \text{in}_{n0} = \emptyset$
- F = all functions f of the form $f(x) = a \cup (x-b)$
 - 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 reach along ALL paths
- Optimizations must conservatively take all possible executions into account. Structure of analysis varies according to way analysis used.

Backward Dataflow Analysis

- Simulates execution of program backward against the flow of control
- For each node n , 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 $in_n := f_n(\perp)$

for each $n \in N_{final}$ do $out_n := O$; $in_n := f_n(O)$

worklist := $N - N_{final}$

while worklist $\neq \emptyset$ do

 remove a node n from worklist

$out_n := \bigvee \{ in_m . m \text{ in } succ(n) \}$

$in_n := f_n(out_n)$

 if in_n changed then

 worklist := worklist \cup 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)
- $\perp = \emptyset$
- $O = \emptyset$
- 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

- Control flow graph and set of variables v in V
- Concept of program state s in ST
 - s is a map that stores values of variables v in V
 - $s[v]$ is the value of v in state s
- Concept of pair $\langle s, n \rangle$ - program state s at node n
- n executes in s to produce $\langle s', n' \rangle$
 - s' stores values of variables after n executes
 - n' is next node to execute

Execution of Program

(program represented as control flow graph)

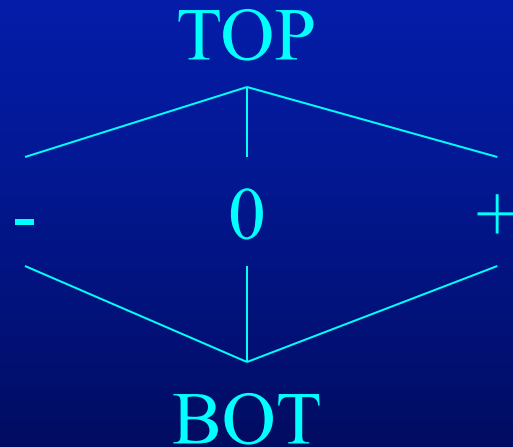
- Concept of a program execution
- Execution is a sequence (trajectory) of $\langle s, n \rangle$ pairs
 - $\langle s_0, n_0 \rangle; \langle s_1, n_1 \rangle; \dots; \langle s_k, n_k \rangle$
 - $\langle s_{i+1}, n_{i+1} \rangle$ generated from $\langle s_i, n_i \rangle$ by
 - executing n_i in state s_i
 - n_i updates variable values in s_i to produce s_{i+1}
 - control then flows to n_{i+1}
 - n_{i+1} is next node to execute after n_i

Relating Program Executions to Dataflow Analysis Results

- Meaning of program analysis result is given by an abstraction function $AF:ST \rightarrow P$
 - $p = AF(s)$
 - s in ST is a program state
 - p in P is an element of dataflow lattice P
- Correctness condition: given any program execution $\langle s_0, n_0 \rangle; \dots; \langle s_k, n_k \rangle$ and pair $\langle s, n \rangle$ where $s = s_i$ and $n = n_i$ for some $0 \leq i \leq k$ then $AF(s) \leq in_n$ where in_n is result that program analysis produces at program point before n

Sign Analysis Example

- Sign analysis - compute sign of each variable v
- Base Lattice: $P = \text{flat lattice on } \{-,0,+\}$



Actual Lattice

- Actual lattice records a sign for each variable
 - Example element: $[a \rightarrow +, b \rightarrow 0, c \rightarrow -]$
- Function lattice
 - Elements of lattice are functions (maps) from variables to base sign lattice
 - For function lattice elements f_1 and f_2
 - $f_1 \leq f_2$ if $\forall v \text{ in } V. f_1(v) \leq f_2(v)$

Interpretation of Lattice Values

- If value of v in lattice is:
 - BOT: no information about sign of v
 - -: variable v is negative
 - 0: variable v is 0
 - +: variable v is positive
 - TOP: v may be positive, negative, or zero
- What is abstraction function AF ?
 - $AF([v_1, \dots, v_n]) = [\text{sign}(v_1), \dots, \text{sign}(v_n)]$
 - Where $\text{sign}(v) = 0$ if $v = 0$, $+$ if $v > 0$, $-$ if $v < 0$

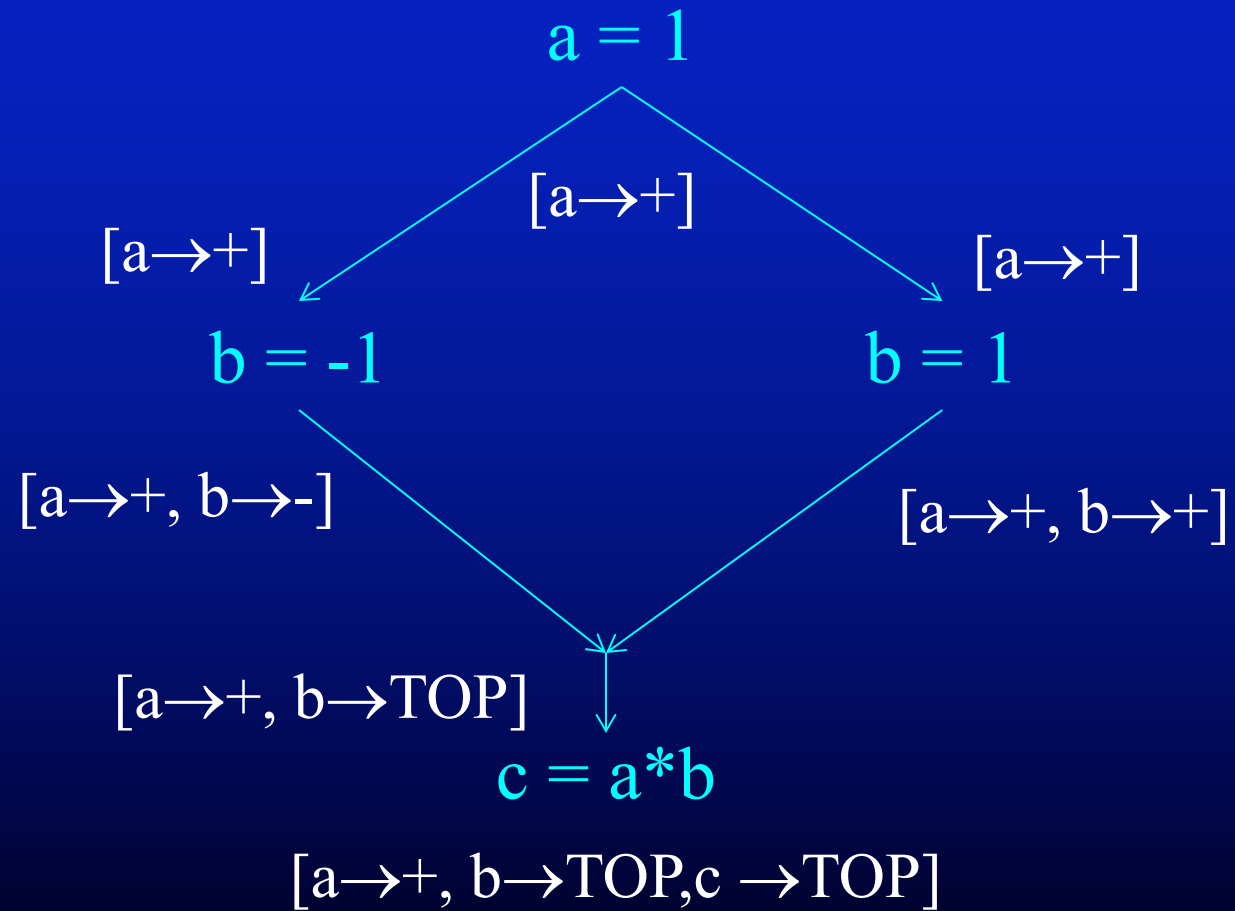
Operation \otimes on Lattice

\otimes	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

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 $v_1 = v_2 * v_3$
 - $f_n(x) = x[v_1 \rightarrow x[v_2] \otimes x[v_3]]$
- $I = \text{TOP}$ (if variables not initialized)
- $I = [v_1 \rightarrow 0, \dots, v_n \rightarrow 0]$ (if variables initialized to 0)

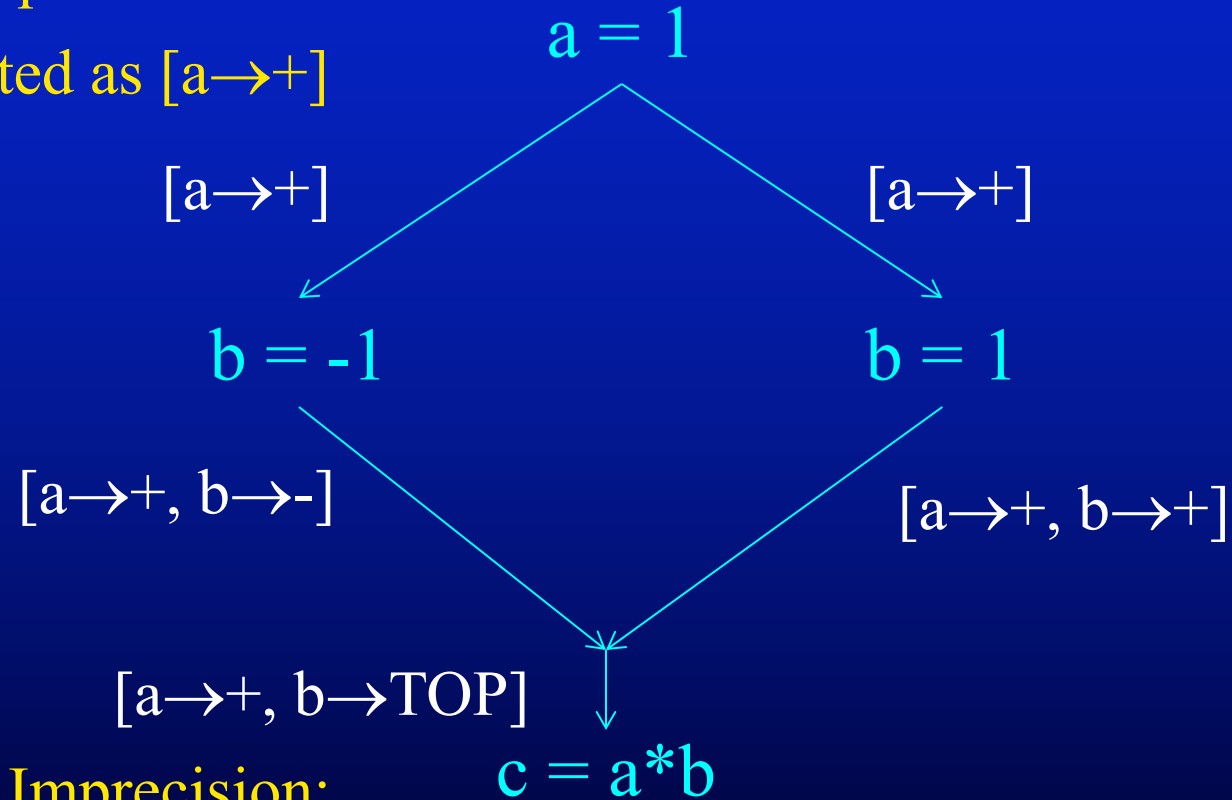
Example



Imprecision In Example

Abstraction Imprecision:

$[a \rightarrow 1]$ abstracted as $[a \rightarrow +]$



Control Flow Imprecision:

$[b \rightarrow \text{TOP}]$ summarizes results of all executions. In any execution state s , $\text{AF}(s)[b] \neq \text{TOP}$

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 Have 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

Abstraction Function

- $AF(s)[v] = \text{sign of } v$
 - $AF([a \rightarrow 5, b \rightarrow 0, c \rightarrow -2]) = [a \rightarrow +, b \rightarrow 0, c \rightarrow -]$
- Establishes meaning of the analysis results
 - If analysis says variable has a given sign
 - Always has that sign in actual execution
- Correctness condition:
 - program execution $\langle s_0, n_0 \rangle; \dots; \langle s_k, n_k \rangle$ and pair $\langle s, n \rangle$
 - where $s = s_i$ and $n = n_i$ for some $0 \leq i \leq k$
 - $\forall v \text{ in } V. AF(s)[v] \leq in_n[v]$ (n is node for s)
 - Reflects possibility of imprecision

Correctness Condition

Start with

program execution $\langle s_0, n_0 \rangle; \dots; \langle s_k, n_k \rangle$ and pair $\langle s, n \rangle$

where $s = s_i$ and $n = n_i$ for some $0 \leq i \leq k$

then $AF(s) \leq in_n$ where

in_n is result that program analysis produces

at program point before n

For sign analysis, $AF(s)$ is a map that gives sign of each variable v

$$\forall v. AF(s)[v] \leq in_n[v]$$

Sign Analysis Soundness

Given

program execution $\langle s_0, n_0 \rangle; \dots; \langle s_k, n_k \rangle$ and pair $\langle s, n \rangle$

where $s = s_i$ and $n = n_i$ for some $0 \leq i \leq k$

then $\forall v. AF(s)[v] \leq in_n[v]$ where

in_n is result that program analysis produces
at program point before n

Will prove by induction on i

(length of execution that produced $\langle s_i, n_i \rangle$)

Base Case of Induction

- For base case
 - $i = 0, n = n_0$
 - $\forall v. \text{in}_{n_0}[v] = \text{TOP}$
- Then $\forall v. \text{AF}(s)[v] \leq \text{TOP}$

Induction Step

- Assume $\forall v. AF(s)[v] \leq in_n[v]$ for executions of length k
- Prove for computations of length $k+1$
- Proof:
 - Given $s = s_{k+1}$ (state), $n = n_{k+1}$ (node to execute next), and in_n
 - Find s_k (the previous state), n_k (the previous node), and in_{n_k}
 - By induction hypothesis $\forall v. AF(s_k)[v] \leq in_{n_k}[v]$
 - Case analysis on form of n_k
 - If n_k of the form $v = c$ (other cases are similar), then
 - $s[v] = c, \quad out_{n_k}[v] = sign(c),$
 - $s[x] = s_k[x], out_{n_k}(x) = in_{n_k}(x)$ for $x \neq v$
 - By induction hypothesis, $\forall x. AF(s_k)[x] \leq out_{n_k}[x]$
 - $out_{n_k} \leq in_n$ (because n_k in $pred(n)$ and in_n is least upper bound of set that includes out_{n_k})
 - Therefore $\forall x. AF(s)[x] \leq in_n[x]$ (transitivity)

Augmented Execution States

- Abstraction functions for some analyses require augmented execution states
 - Reaching definitions: states are augmented with definition that created each value
 - Available expressions: states are augmented with expression for each value

Meet Over Paths Solution

- What solution would be ideal for a forward dataflow analysis problem?
- Consider a path $p = n_0, n_1, \dots, n_k, n$ to a node n (note that for all i $n_i \in \text{pred}(n_{i+1})$)
- The solution must take this path into account:
$$f_p(\perp) = (f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots)) \leq in_n$$
- So the solution must have the property that $\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} \leq in_n$
and ideally
$$\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} = in_n$$

Soundness Proof of Analysis Algorithm

- Property to prove:

For all paths p to n , $f_p(\perp) \leq in_n$

- Proof is by induction on length of p
 - Uses monotonicity of transfer functions
 - Uses following lemma

- Lemma:

Worklist algorithm produces a solution such that

$$f_n(in_n) = out_n$$

$$\text{if } n \in \text{pred}(m) \text{ then } out_n \leq in_m$$

Proof

- Base case: p is of length 1
 - Then $p = n_0$ and $f_p(\perp) = \perp = \text{in}_{n_0}$
- Induction step:
 - Assume theorem for all paths of length k
 - Show for an arbitrary path p of length $k+1$

Induction Step Proof

- $p = n_0, \dots, n_k, n$
- Must show $f_k(f_{k-1}(\dots f_{n1}(f_{n0}(\perp)) \dots)) \leq in_n$
 - By induction $(f_{k-1}(\dots f_{n1}(f_{n0}(\perp)) \dots)) \leq in_{nk}$
 - Apply f_k to both sides, by monotonicity we get $f_k(f_{k-1}(\dots f_{n1}(f_{n0}(\perp)) \dots)) \leq f_k(in_{nk})$
 - By lemma, $f_k(in_{nk}) = out_{nk}$
 - By lemma, $out_{nk} \leq in_n$
 - By transitivity, $f_k(f_{k-1}(\dots f_{n1}(f_{n0}(\perp)) \dots)) \leq in_n$

Distributivity

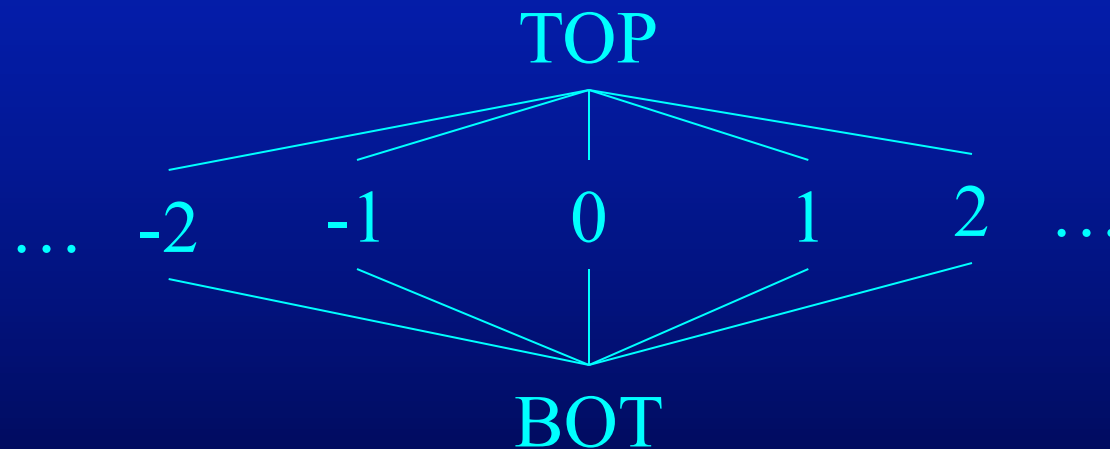
- Distributivity preserves precision
- If framework is distributive, then worklist algorithm produces the meet over paths solution

– For all n :

$$\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} = in_n$$

Lack of Distributivity Example

- Constant Calculator
- Flat Lattice on Integers

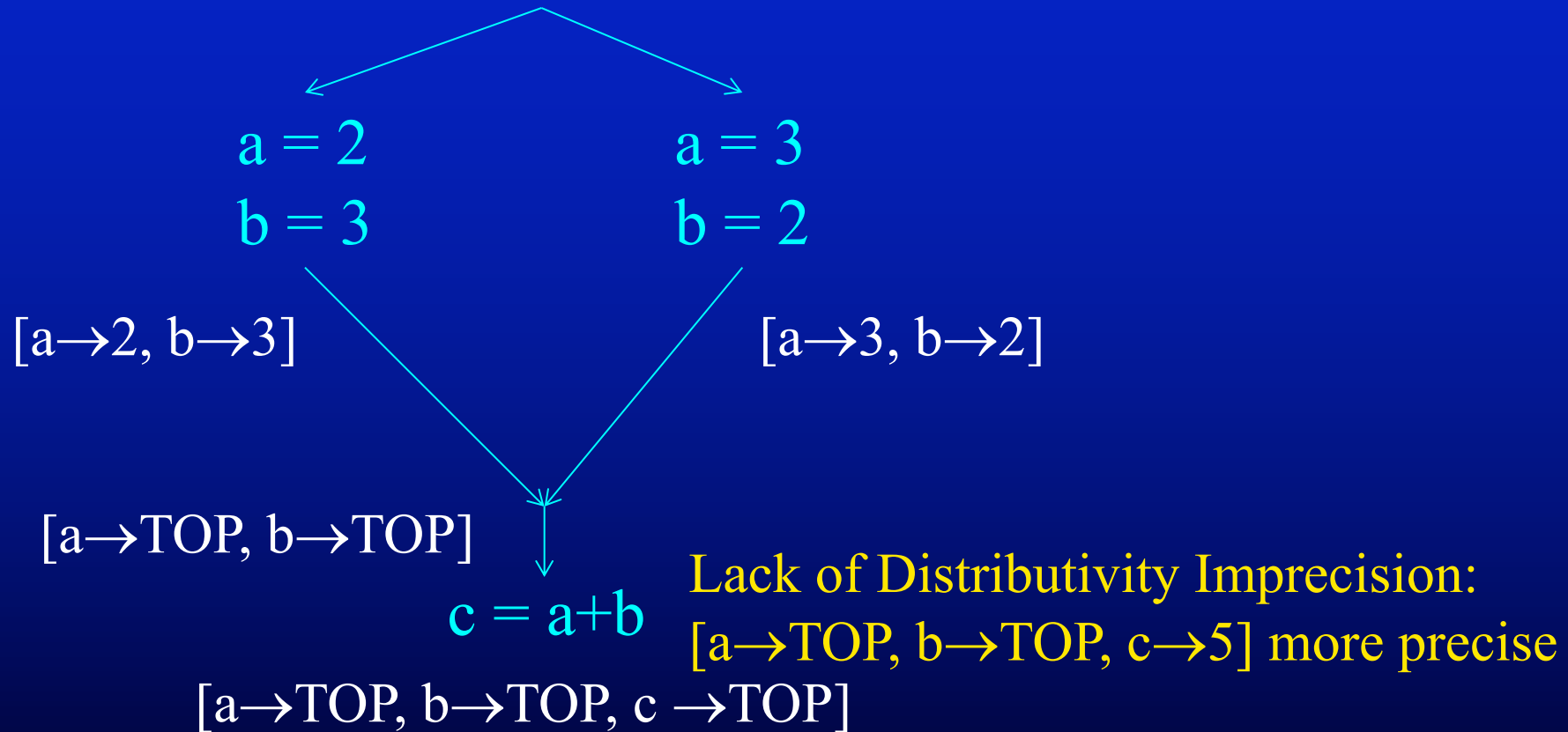


- Actual lattice records a value for each variable
 - Example element: $[a \rightarrow 3, b \rightarrow 2, c \rightarrow 5]$

Transfer Functions

- If n of the form $v = c$
 - $f_n(x) = x[v \rightarrow c]$
- 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
 - Consider transfer function f for $c = a + b$
 - $f([a \rightarrow 3, b \rightarrow 2]) \vee f([a \rightarrow 2, b \rightarrow 3]) = [a \rightarrow \text{TOP}, b \rightarrow \text{TOP}, c \rightarrow 5]$
 - $f([a \rightarrow 3, b \rightarrow 2] \vee [a \rightarrow 2, b \rightarrow 3]) = f([a \rightarrow \text{TOP}, b \rightarrow \text{TOP}]) = [a \rightarrow \text{TOP}, b \rightarrow \text{TOP}, c \rightarrow \text{TOP}]$

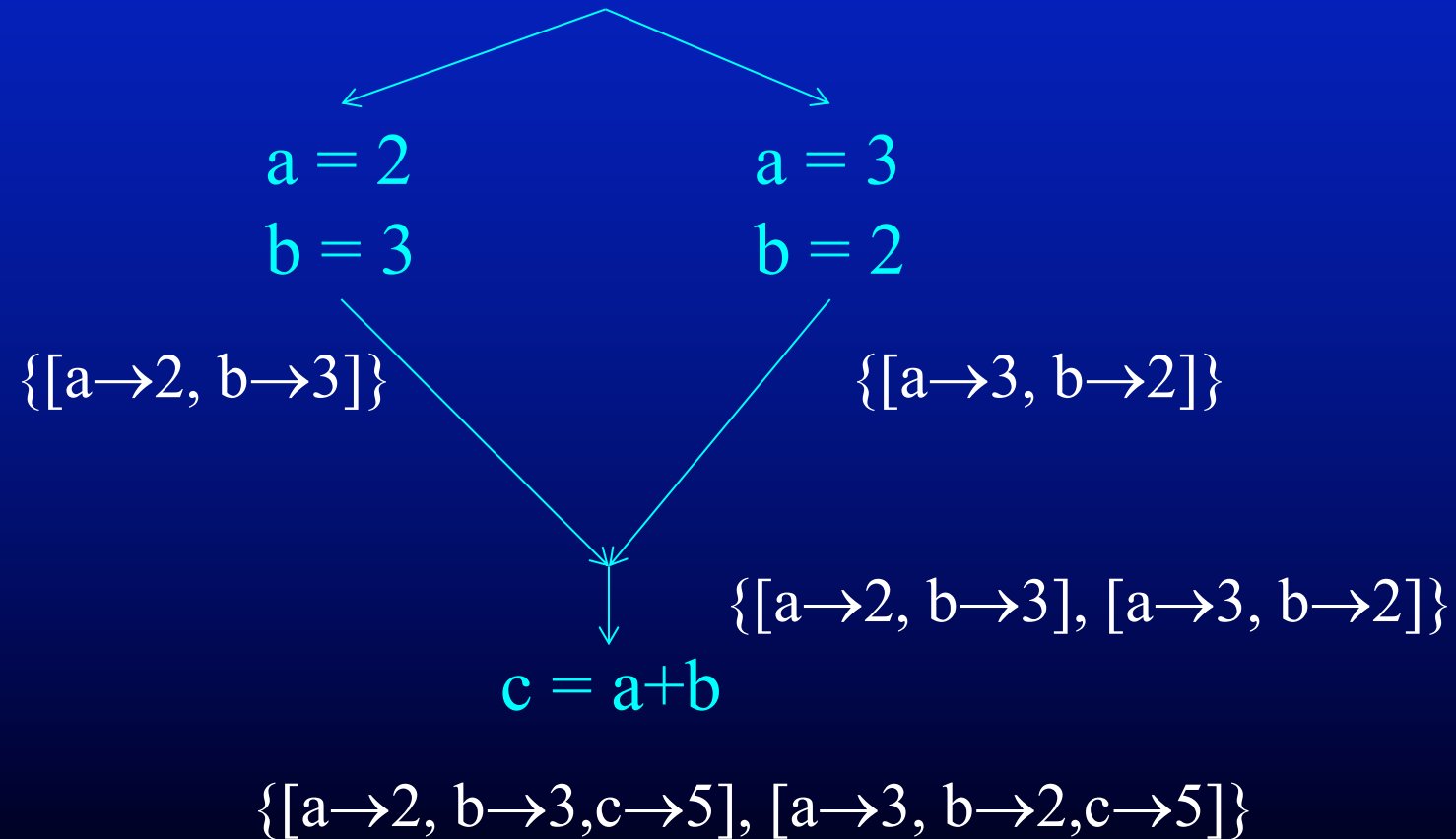
Lack of Distributivity Anomaly



What is the meet over all paths solution?

How to Make Analysis Distributive

- Keep combinations of values on different paths

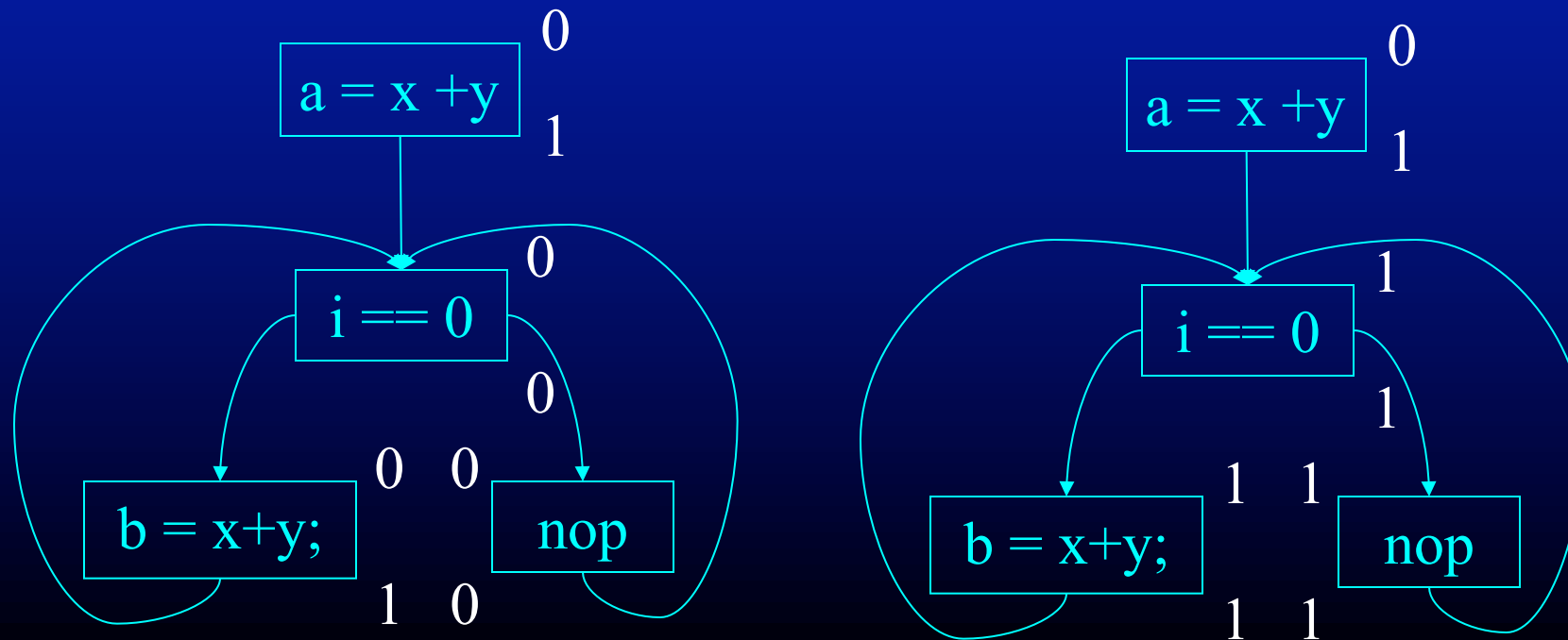


Issues

- Basically simulating all combinations of values in all executions
 - Exponential blowup
 - Nontermination because of infinite ascending chains
- Nontermination solution
 - Use widening operator to eliminate blowup (can make it work at granularity of variables)
 - Loses precision in many cases

Multiple Fixed Points

- Dataflow analysis generates least fixed point
- May be multiple fixed points
- Available expressions example



Summary

- Formal dataflow analysis framework
 - Lattices, partial orders, least upper bound, greatest lower bound, ascending chains
 - Transfer functions, joins and splits
 - Dataflow equations and fixed point solutions
- Connection with program
 - Abstraction function $AF: S \rightarrow P$
 - For any state s and program point n , $AF(s) \leq in_n$
 - Meet over all paths solutions, distributivity

For the quiz, you should know:

- How to give transfer functions for simple lattices and nodes
- Abstraction functions
- Meet over paths solution
- Causes of imprecision