MIT 6.035
Foundations of Dataflow Analysis

Martin Rinard
Laboratory for Computer Science
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
    - pred(n) = set of all predecessors of n
    - 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

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

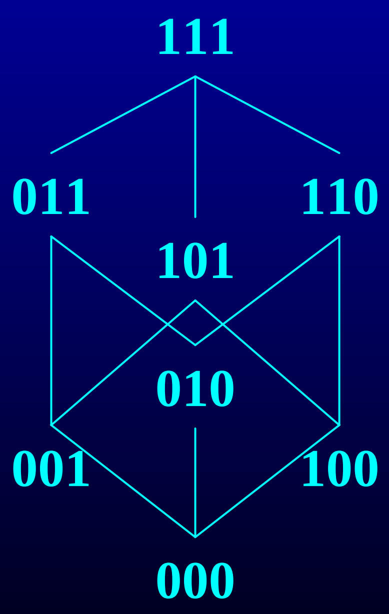
- If $S \subseteq P$ then
  - $x \in P$ is a lower bound of S if $\forall y \in S.\ x \leq y$
  - $x \in P$ is the greatest lower bound of S if
    - x is a lower bound of S, and
    - $y \leq x$ for all lower bounds y of S
  - $\wedge$ - meet, greatest lower bound, glb, infimum, inf
    - $\wedge S$ 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)
- $x \le y$ if (x 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 $x \le y$ implies $x \vee y = y$
  - $x \le y$ implies y is an upper bound of $\{x,y\}$.
  - Any upper bound z of $\{x,y\}$ must satisfy $y \le z$.
  - So y is least upper bound of $\{x,y\}$ and $x \vee y = y$
- Proof of $x \le y$ implies $x \wedge y = x$
  - $x \le y$ implies x is a lower bound of $\{x,y\}$.
  - Any lower bound z of $\{x,y\}$ must satisfy $z \le x$.
  - 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

$$x \vee y = y \text{ and } y \vee z = z \text{ implies } x \vee z = z$$

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

Properties of $\leq$

- Proof of asymmetry property. Must show that $x \vee y = y$ and $y \vee x = x$ implies $x = y$

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

- Proof of reflexivity property. Must show that $x \vee x = x$

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

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 $x_1 \leq x_2 \leq \ldots$ there exists n such that $x_n = x_{n+1} = \ldots$

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

Dataflow Equations

- Compiler processes program to obtain a set of dataflow equations

  $out_n := f_n(in_n)$

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

- Conceptually separates analysis problem from program

Worklist Algorithm for Solving Forward Dataflow Equations

for each n do $out_n := f_n(\perp)$

$in_{n0} := I$; $out_{n0} := f_{n0}(I)$

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

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

  to ensure that $in_n = \vee \{ out_m \text{ . m in pred(n)} \}$
- 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 $\subseteq$)
- $\perp = \varnothing$
- $I = in_{n0} = \perp$
- 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 $\leq$
  - $x \subseteq y$ and $y \subseteq z$ implies $x \subseteq z$ (transitivity)
  - $x \subseteq y$ and $y \subseteq x$ implies $y = x$ (asymmetry)
  - $x \subseteq x$ (idempotence)
- 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 $\supseteq$)
- $\perp = P$
- $I = in_{n0} = \varnothing$
- 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
    $out_n := \vee \{ in_m \,.\, m \text{ in } succ(n) \}$
    $in_n := f_n(out_n)$
    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

- Concept of program state s for control-flow graphs
  - Program point n where execution 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 $s_i \in ST$
  - s_{i+1} generated from s_i by executing basic block to
    - Update variable values
    - Obtain new program point n

Relating States to Analysis Result

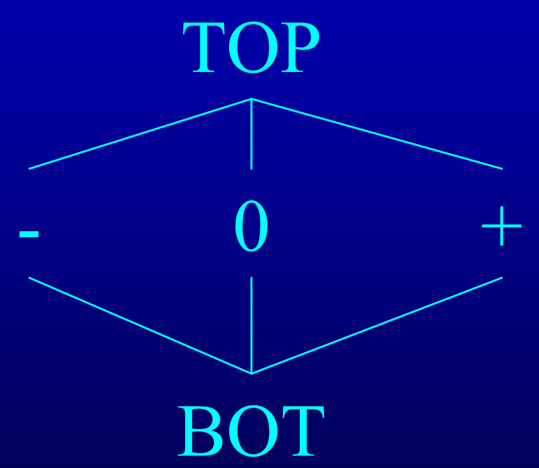
- Meaning of analysis results is given by an abstraction function $AF:ST \rightarrow P$
- Correctness condition: require that for all states s

$$AF(s) \leq in_n$$

where n is the next statement to execute in state s

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 sign of v
  - -: variable v is negative
  - 0: variable v is 0
  - +: variable v is positive
  - TOP: v may be positive or negative
- What is abstraction function AF?
  - $AF([x_1,\ldots,x_n]) = [sign(x_1), \ldots, sign(x_n)]$
  - Where $sign(x) = 0$ if $x = 0$, $+$ if $x > 0$, $-$ if $x < 0$

Operation ⊗ on Lattice

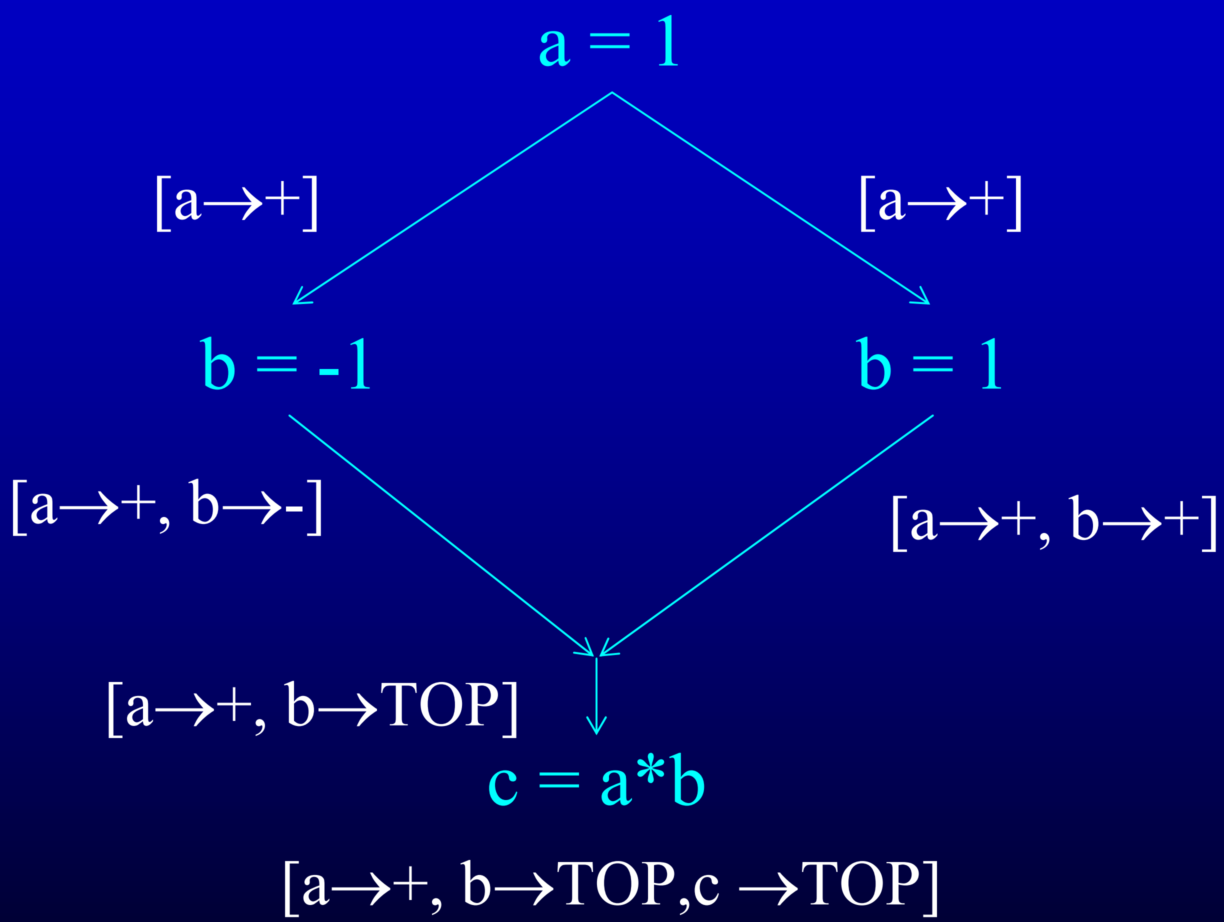
⊗	BOT	-	0	+	TOP
BOT	BOT	-	0	+	TOP
-	-	+	0	-	TOP
0	0	0	0	0	0
+	+	-	0	+	TOP
TOP	TOP	TOP	0	TOP	TOP

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 = TOP

  (uninitialized variables may have any sign)

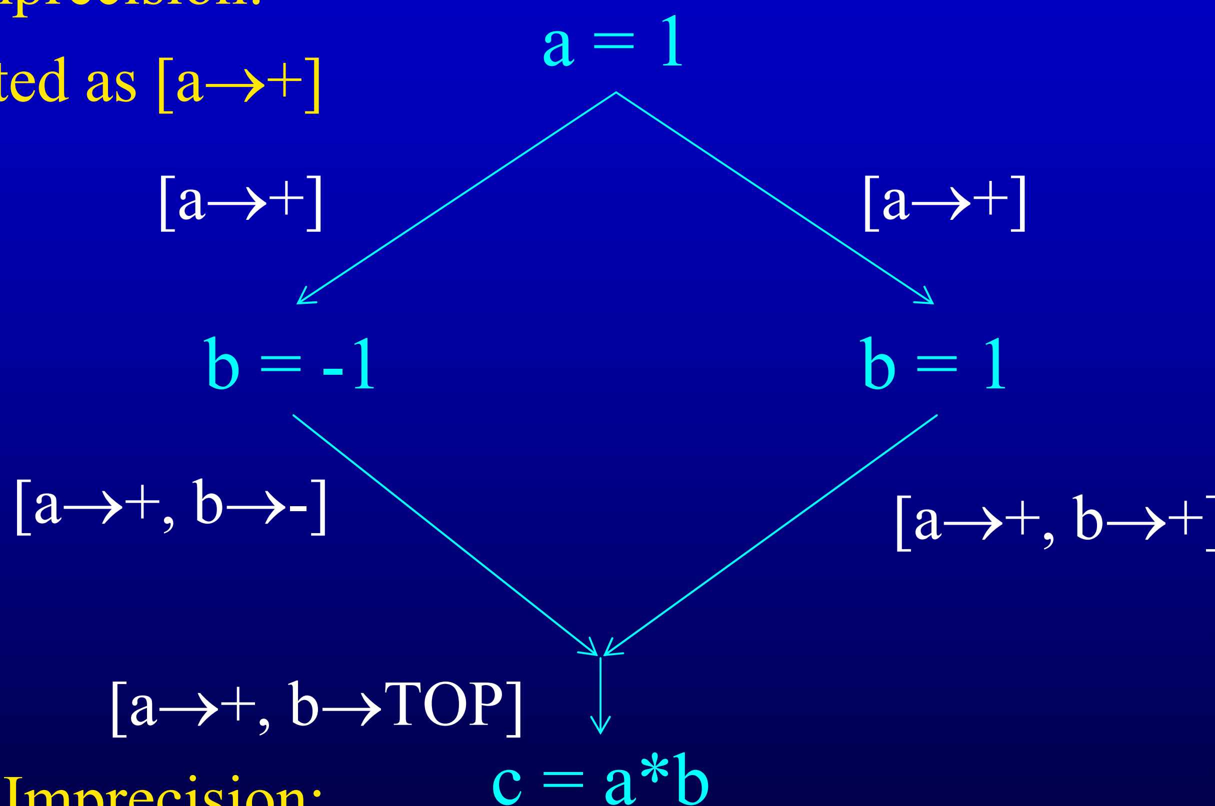
Example



Imprecision In Example

Abstraction Imprecision:

[a→1] abstracted as [a→+]



Control Flow Imprecision:

[b→TOP] summarizes results of all executions. In any 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 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] = sign of v
  - AF(n,[a→5, b→0, c→-2]) = [a→+, b→0, c→-]
- Establishes meaning of the analysis results
  - If analysis says variable has a given sign
  - Always has that sign in actual execution
- Correctness condition:
  - $\forall$ v. AF(s)[v] $\leq$ in_n[v] (n is node for s)
  - Reflects possibility of imprecision

Abstraction Function Soundness

- Will show
  $\forall$ v. AF(s)[v] $\leq$ in_n[v] (n is node for s)
  by induction on length of computation that produced s
- Base case:
  - $\forall$ v. in_{n0}[v] = TOP, which implies that
  - $\forall$ v. AF(s)[v] $\leq$ TOP

Induction Step

- Assume $\forall$ v. $AF(s)[v] \leq in_n[v]$ for computations of length k
- Prove for computations of length k+1
- Proof:
  - Given s (state), n (node to execute next), and in_n
  - Find p (the node that just executed), s_p(the previous state), and in_p
  - By induction hypothesis $\forall$ v. $AF(s_p)[v] \leq in_p[v]$
  - Case analysis on form of n
    - If n of the form v = c, then
      - $s[v] = c$ and $out_p\,[v] = sign(c)$, so
        $AF(s)[v] = sign(c) = out_p\,[v] \leq in_n[v]$
      - If $x \neq v$, $s[x] = s_p\,[x]$ and $out_p\,[x] = in_p[x]$, so
        $AF(s)[x] = AF(s_p)[x] \leq in_p[x] = out_p\,[x] \leq in_n[x]$
    - Similar reasoning if n of the form $v_1 = v_2 * v_3$

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 $p = n_0, n_1, \ldots, n_k, n$ to a node n
  (note that for all i $n_i \in pred(n_{i+1})$)
- The solution must take this path into account:
  $f_p(\perp) = (f_{nk}(f_{nk-1}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \leq in_n$
- So the solution must have the property that
  $\vee\{f_p(\perp) \,.\, p \text{ is a path to } n\} \leq in_n$
  and ideally
  $\vee\{f_p(\perp) \,.\, p \text{ is a path to } n\} = in_n$

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

  $f_n(in_n) = out_n$

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

Proof

- Base case: p is of length 1
  - Then $p = n_0$ and $f_p(\perp) = \perp = in_{n0}$
- Induction step:
  - Assume theorem for all paths of length k
  - Show for an arbitrary path p of length k+1

Induction Step Proof

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

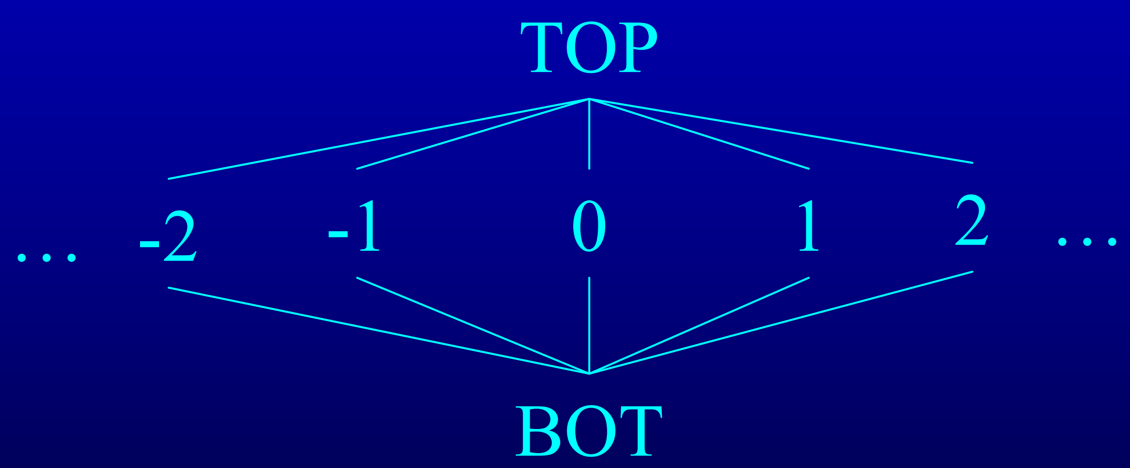
Distributivity

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

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

Lack of Distributivity Example

- Constant Calculator
- Flat Lattice on Integers



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

Transfer Functions

- If n of the form v = c
  - $f_n(x) = x[v \to c]$
- If n of the form $v_1 = v_2 + v_3$
  - $f_n(x) = x[v_1 \to x[v_2] + x[v_3]]$
- Lack of distributivity
  - Consider transfer function f for c = a + b
  - $f([a \to 3, b \to 2]) \vee f([a \to 2, b \to 3]) = [a \to TOP, b \to TOP, c \to 5]$
  - $f([a \to 3, b \to 2] \vee [a \to 2, b \to 3]) = f([a \to TOP, b \to TOP]) = [a \to TOP, b \to TOP, c \to TOP]$

Lack of Distributivity Anomaly

a = 2
b = 3

a = 3
b = 2

[a→2, b→3]

[a→3, b→2]

[a→TOP, b→TOP]

c = a+b

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

[a→TOP, b→TOP, c →TOP]

What is the meet over all paths solution?

How to Make Analysis Distributive

- Keep combinations of values on different paths

a = 2
b = 3

a = 3
b = 2

{[a→2, b→3]}

{[a→3, b→2]}

{[a→2, b→3], [a→3, b→2]}

c = a+b

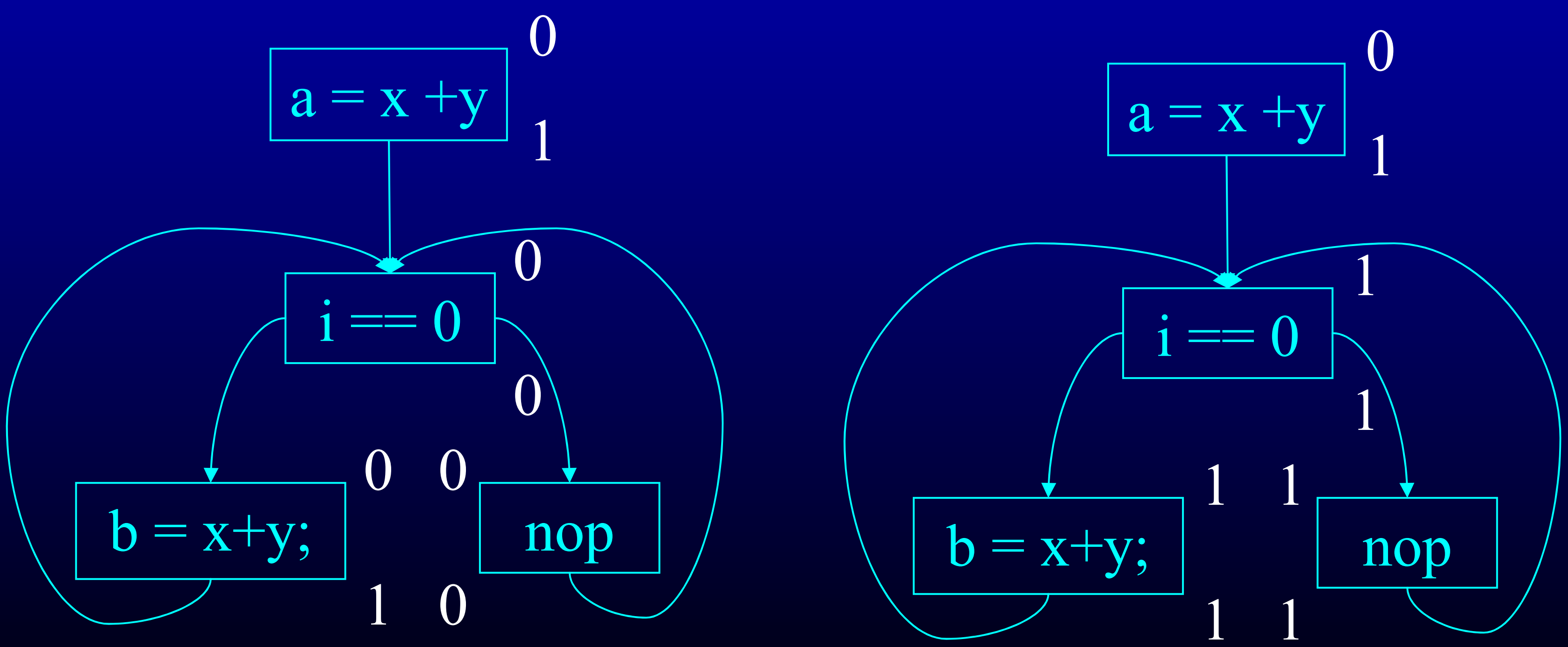
{[a→2, b→3,c→5], [a→3, b→2,c→5]}

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



Pessimistic vs. Optimistic Analyses

- Available expressions is optimistic (for common sub-expression elimination)
  - Assumes expressions are available at start of analysis
  - Analysis eliminates all that are not available
  - If analysis result $in_n \leq e$, can use e for CSE
  - Cannot stop analysis early and use current result
- Live variables is pessimistic (for dead code elimination)
  - Assumes all variables are live at start of analysis
  - Analysis finds variables that are dead
  - If $e \leq$ analysis result in_n, can use e for dead code elimination
  - Can stop analysis early and use current result
- Formal dataflow setup same for both analyses
- Optimism/pessimism depends on intended use

Summary

- Formal dataflow analysis framework
  - Lattices, partial orders
  - 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