KT-2

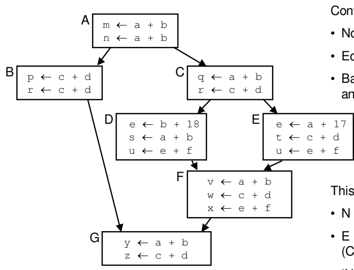
P3 / 2006

Foundations of Dataflow Analysis

Terminology: Program Representation

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

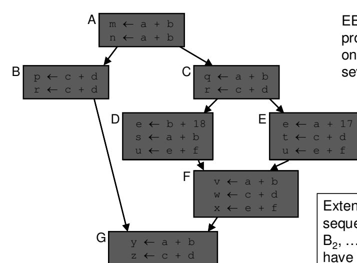
Terminology: Control-Flow Graph



Control-flow graph (CFG)
- Nodes for basic blocks
- Edges for branches
- Basis for much of program analysis & transformation

This CFG, G = (N,E)
- N = {A,B,C,D,E,F,G}
- E = {(A,B),(A,C),(B,G),(C,D),(C,E),(D,F),(E,F),(F,E)}
- |N| = 7, |E| = 8

Terminology: Extended Basic Block



EBB: Conceptually it is a program sequence with only one entry point but possibly several exit points.

Extended Basic Block (EBB): A sequence of basic blocks B_1, B_2, …, B_n where all B_i (i > 1) have a unique predecessor from the set B_1, …, B_{i-1}.

Path of an EBB: A sequence of basic blocks B_1, B_2, …, B_n where B_i is the predecessor of B_{i+1}.

Terminology: Program Points

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

Dataflow Analysis

Compile-Time Reasoning About

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

Dataflow Analysis: Basic Idea

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

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)

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 $\{x,y\}$

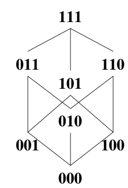
Coverings

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

Kostis Sagonas 13 Spring 2006

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



We can visualize a partial order with a Hasse Diagram

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

Kostis Sagonas 14 Spring 2006

Lattices

- If $x \wedge y$ and $x \vee y$ exist (i.e., are in P) 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*.
- Theorem: All finite lattices are complete
- Example of a lattice that is not complete
  - Integers Z
  - For any $x, y \in Z$, $x \vee y = \max(x,y)$, $x \wedge y = \min(x,y)$
  - But $\vee Z$ and $\wedge Z$ do not exist
  - $Z \cup \{+\infty, -\infty\}$ is a complete lattice

Kostis Sagonas 15 Spring 2006

Top and Bottom

- Greatest element of P (if it exists) is *top* ($\top$)
- Least element of P (if it exists) is *bottom* ($\bot$)

Kostis Sagonas 16 Spring 2006

Connection between $\le$, $\wedge$, and $\vee$

The following 3 properties are equivalent:
- $x \le y$
- $x \vee y = y$
- $x \wedge y = x$

- Will prove:
  - $x \le y$ implies $x \vee y = y$ and $x \wedge y = x$
  - $x \vee y = y$ implies $x \le y$
  - $x \wedge y = x$ implies $x \le y$
- By Transitivity,
  - $x \vee y = y$ implies $x \wedge y = x$
  - $x \wedge y = x$ implies $x \vee y = y$

Kostis Sagonas 17 Spring 2006

Connecting Lemma Proofs (1)

- 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 $x \wedge y = x$

Kostis Sagonas 18 Spring 2006

Connecting Lemma Proofs (2)

- 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

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 $\vee$

Theorem: $x \vee y = y$ if and only if $x \wedge y = x$

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

  $x = x \wedge (x \vee y)$ (by absorption)
  $= x \wedge y$ (by assumption)

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

  $y = y \vee (y \wedge x)$ (by absorption)
  $= y \vee (x \wedge y)$ (by commutativity)
  $= y \vee x$ (by assumption)
  $= x \vee y$ (by commutativity)

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 $x \vee z = z$

  $x \vee z = x \vee (y \vee z)$ (by assumption)
  $= (x \vee y) \vee z$ (by associativity)
  $= y \vee z$ (by assumption)
  $= z$ (by assumption)

Properties of $\leq$

- Proof of asymmetry property. Must show that

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

  $x = y \vee x$ (by assumption)
  $= x \vee y$ (by commutativity)
  $= y$ (by assumption)

- Proof of reflexivity property. Must show that

  $x \vee x = x$

  $x \vee x = x$ (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 $x_n = x_{n+1} = \ldots$

Transfer Functions

- Assume a lattice of abstract values P
- Transfer function $f: P \to P$ for each node in control flow graph
- f models effect of the node on the program information

Properties of Transfer Functions

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

- 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:

- Assume $f(x \vee y) = f(x) \vee f(y)$
- Must show: $x \vee y = y$ implies $f(x) \vee f(y) = f(y)$

$f(y) = f(x \vee y)$ (by assumption)

$= f(x) \vee f(y)$ (by distributivity)

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 solutions satisfy
  - $\forall n, out_n = f_n(in_n)$
  - $\forall n \neq n_0, in_n = \vee \{ out_m \mid m \text{ in } pred(n) \}$
  - $in_{n0} = \perp$

Dataflow Equations

- Result is a set of dataflow equations

  $out_n := f_n(in_n)$

  $in_n := \vee \{ out_m \mid m \text{ in } pred(n) \}$
- Conceptually separates analysis problem from program

Worklist Algorithm for Solving Forward Dataflow Equations

for each n do $out_n := f_n(\perp)$
worklist := N
while worklist $\neq \varnothing$ do
  remove a node n from worklist
  $in_n := \vee \{ out_m \mid m \text{ in } pred(n) \}$
  $out_n := f_n(in_n)$
  if out_n changed then
    worklist := worklist $\cup$ succ(n)

Correctness Argument

Why result satisfies dataflow equations?

- Whenever we 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 the worklist and the algorithm will set

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

  to ensure that $in_n = \vee \{ out_m \mid m \text{ in } pred(n) \}$

Termination Argument

Why does the algorithm terminate?

- Sequence of values taken on by in_n or out_n is a chain. If values stop increasing, the worklist empties and the algorithm terminates.
- If the lattice has the ascending chain property, the algorithm terminates
  - Algorithm terminates for finite lattices
  - For lattices without the ascending chain property, we must use a *widening* operator

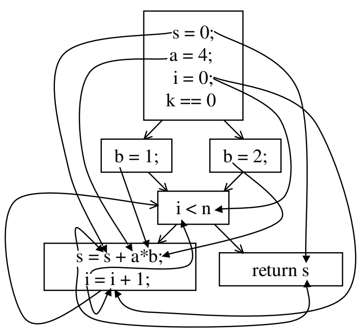
Widening Operators

- Detect lattice values that may be part of an infinitely ascending chain
- Artificially raise value to least upper bound of the chain
- Example:
  - Lattice is set of all subsets of integers
  - Widening operator might raise all sets of size n or greater to TOP
  - Could be used to collect possible values taken on by a variable during execution of the program

Reaching Definitions

- Concept of *definition* and *use*
  - z = x+y
  - is a definition of z
  - is a use of x and y
- A definition reaches a use if
  - the value written by definition
  - may be read by the use.

Reaching Definitions



Reaching Definitions Framework

- P = powerset of set of all definitions in program (all subsets of set of definitions in program)
- $\vee = \cup$ (order is $\subseteq$)
- $\perp = \emptyset$
- 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

- $f(x) = GEN \cup (x-KILL)$

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)
  - $x \subseteq x$ (reflexivity)
- 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 the properties:
- Identity
- Distributivity
- Compositionality

Available Expressions Framework

- P = powerset of set of all expressions in program (all subsets of set of expressions)
- $\vee = \cap$ (order is $\supseteq$)
- $\perp$ = P (but $in_{n0} = \emptyset$)
- 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 reach along ALL paths
- Optimizations must conservatively take all possible executions into account.
- Structure of analysis varies according to the way the results of the analysis are to be used.

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 solutions satisfy
  - $\forall n.\ in_n = f_n(out_n)$
  - $\forall n \notin N_{final}.\ out_n = \vee \{\ in_m \mid m \text{ in } succ(n)\ \}$
  - $\forall n \in N_{final} = out_n = \perp$

Worklist Algorithm for Solving Backward Dataflow Equations

for each n do $in_n := f_n(\perp)$
worklist := N
while worklist $\neq \emptyset$ do
  remove a node n from worklist
  $out_n := \vee \{\ in_m \mid m \text{ in } succ(n)\ \}$
  $in_n := f_n(out_n)$
  if in_n changed then
    worklist := worklist $\cup$ pred(n)

Live Variables Analysis Framework

- P = powerset of set of all variables in program (all subsets of set of variables in program)
- $\vee = \cup$ (order is $\subseteq$)
- $\perp = \emptyset$
- F = all functions f of the form $f(x) = a \cup (x\text{-}b)$
  - b is set of variables that the node kills
  - a is set of variables that the node reads

Meaning of Dataflow Results

- Connection between executions of program and dataflow analysis results
- Each execution generates a trajectory of states:
  - $s_0;s_1;\ldots;s_k$,where each $s_i \in ST$
- Map current state s_k to
  - Program point n where execution located
  - Value x in dataflow lattice
- Require $x \le in_n$

Abstraction Function for Forward Dataflow Analysis

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

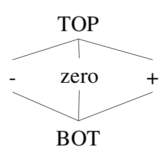
  $$AF(s) \le in_n$$

  where n is program point where the execution is located in state s, and in_n is the abstract value before that point.

Sign Analysis Example

Sign analysis - compute sign of each variable v

- Base Lattice: flat lattice on {-,zero,+}



- Actual lattice records a value for each variable
  - Example element: [a→+, b→zero, c→-]

Interpretation of Lattice Values

If value of v in lattice is:

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

Operation ⊗ on Lattice

⊗	BOT	-	zero	+	TOP
BOT	BOT	-	zero	+	TOP
-	-	+	zero	-	TOP
zero	zero	zero	zero	zero	zero
+	+	-	zero	+	TOP
TOP	TOP	TOP	zero	TOP	TOP

Transfer Functions

Defined by structural induction on the shape of nodes:

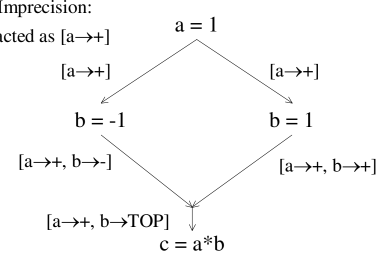
- If n of the form v = c
  - $f_n(x) = x[v \rightarrow +]$ if c is positive
  - $f_n(x) = x[v \rightarrow zero]$ if c is 0
  - $f_n(x) = x[v \rightarrow -]$ if c is negative
- If n of the form $v_1 = v_2 * v_3$
  - $f_n(x) = x[v_1 \rightarrow x[v_2] \otimes x[v_3]]$

Abstraction Function

- AF(s)[v] = sign of v
  - AF([a→5, b→0, c→-2]) = [a→+, b→zero, c→-]
- Establishes meaning of the analysis results
  - If analysis says a variable v has a given sign
  - then v always has that sign in actual execution.
- Two sources of imprecision
  - Abstraction Imprecision – concrete values (integers) abstracted as lattice values (-,zero, and +)
  - Control Flow Imprecision – one lattice value for all different possible flow of control possibilities

Imprecision 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
  - Lattice values less precise than execution values
  - Abstraction function throws away information
- Control Flow Imprecision
  - 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 $x \le y$, then x is more precise than y

Why Have Imprecision?

ANSWER: To make analysis tractable

- Conceptually infinite sets of values in execution
  - Typically abstracted by finite set of lattice values
- Execution may visit infinite set of states
  - Abstracted by computing joins of different paths

Augmented Execution States

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

Meet Over All 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:
  $f_p(\perp) = (f_{nk}(f_{nk-1}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \le in_n$
- So the solution must have the property that
  $\vee\{f_p(\perp) \mid p \text{ is a path to } n\} \le in_n$
  and ideally
  $\vee\{f_p(\perp) \mid p \text{ is a path to } n\} = in_n$

Soundness Proof of Analysis Algorithm

Property to prove:

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

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

Lemma:

The worklist algorithm produces a solution such that if $n \in pred(m)$ then $out_n \le in_m$

Proof

- Base case: p is of length 0
  - 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 $(f_k(f_{k-1}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \le in_n$
  - By induction, $(f_{k-1}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \le in_{nk}$
  - Apply f_k to both sides. By monotonicity, we get: $(f_k(f_{k-1}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \le f_k(in_{nk}) = out_{nk}$
  - By lemma, $out_{nk} \le in_n$
  - By transitivity, $(f_k(f_{k-1}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \le in_n$

Distributivity

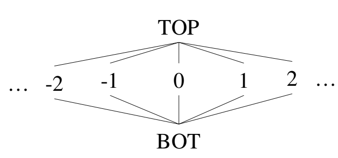
- Distributivity preserves precision
- If framework is distributive, then the worklist algorithm produces the meet over paths solution
  - For all n:

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

Lack of Distributivity Example

Integer Constant Propagation (ICP)

- Flat lattice on integers

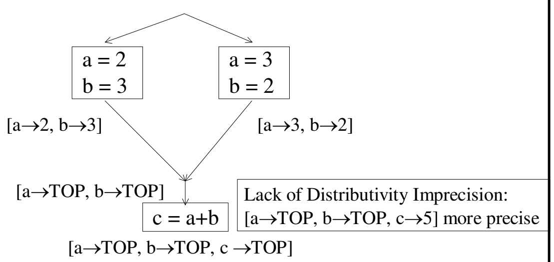


- Actual lattice records a value for each variable
  - Example element: $[a \to 3, b \to 2, c \to 5]$

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 of ICP
  - 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



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 paths solutions, distributivity