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

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?
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 $\forall x, y \in P$
  - $x \leq x$ (reflexive)
  - $x \leq y$ and $y \leq x$ implies $x = y$ (symmetric)
  - $x \leq y$ 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 $\forall y \in S, y \leq x$ (upper bound)
  - $x \in P$ 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 $S$
  - $\lor$ - join, least upper bound, lub, supremum, sup
    - $\lor S$ is the least upper bound of $S$
    - $x \lor y$ 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 $\forall y \in S, x \leq y$ (lower bound)
  - $x \in P$ 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 $S$
  - $\land$ - meet, greatest lower bound, glb, infimum, inf
    - $\land S$ is the greatest lower bound of $S$
    - $x \land y$ is the greatest lower bound of $\{x, y\}$

Covering

- $x < y$ if $x \leq y$ and $x \not\leq y$
- $x$ 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 ($x$ bitwise and $y$) = $x$

Hasse Diagram

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

- If \( x \land y \) and \( x \lor y \) exist for all \( x, y \in P \), then \( P \) is a lattice.
- If \( S \mathbin{\text{and}} \lor S \) 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 \( x, y \in I \), \( x \land y = \max(x, y) \), \( x \lor y = \min(x, y) \)
  - But \( 0 \lor \infty \) and \( 0 \land \infty \) do not exist
  - \( I \bigcup \{ \infty, -\infty \} \) is a complete lattice

Connection Between \( \leq \), \( \land \), and \( \lor \)

- The following 3 properties are equivalent:
  - \( x \leq y \)
  - \( x \lor y = y \)
  - \( x \land y = x \)
- Will prove:
  - \( x \leq y \) implies \( x \lor y = y \) and \( x \land y = x \)
  - \( x \lor y = y \) implies \( x \leq y \)
  - \( x \land y = x \) implies \( x \leq y \)
- Then by transitivity, can obtain
  - \( x \lor y = y \) implies \( x \land y = x \)
  - \( x \land y = x \) implies \( x \lor y = y \)

Connecting Lemma Proofs

- Proof of \( x \leq y \) implies \( x \lor y = y \)
  - \( x \leq y \) 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 \( x \lor y = y \)
- Proof of \( x \leq y \) implies \( x \land y = x \)
  - \( x \leq y \) 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 \( x \land y = x \)

Lattices as Algebraic Structures

- Have defined \( \lor \) and \( \land \) in terms of \( \leq \)
- Will now define \( \leq \) in terms of \( \lor \) and \( \land \)
  - Start with \( \lor \) and \( \land \) as arbitrary algebraic operations that satisfy associative, commutative, idempotence, and absorption laws
  - Will define \( \leq \) using \( \lor \) and \( \land \)
  - Will show that \( \leq \) is a partial order
Algebraic Properties of Lattices
Assume arbitrary operations $\lor$ and $\land$ such that
\[
- (x \lor y) \lor z = x \lor (y \lor z) \quad \text{(associativity of $\lor$)}
- (x \land y) \land z = x \land (y \land z) \quad \text{(associativity of $\land$)}
- x \lor y = y \lor x \quad \text{(commutativity of $\lor$)}
- x \land y = y \land x \quad \text{(commutativity of $\land$)}
- x \lor (x \land y) = x \quad \text{(absorption of $\lor$ over $\land$)}
- x \land (x \lor y) = x \quad \text{(absorption of $\land$ over $\lor$)}
\]

Connection Between $\land$ and $\lor$
- $x \lor y = y$ if and only if $x \land y = x$
- Proof of $x \lor y = y$ implies $x = x \land y$
  \[
  x = x \land (x \lor y) \quad \text{(by absorption)}
  = x \land y \quad \text{(by assumption)}
  
  \]
- Proof of $x \land y = x$ implies $y = x \lor y$
  \[
  y = y \lor (y \land x) \quad \text{(by absorption)}
  = y \lor (x \lor y) \quad \text{(by commutativity)}
  = y \lor x \quad \text{(by assumption)}
  = x \lor y \quad \text{(by commutativity)}
  
  \]

Properties of $\leq$
- Define $x \leq y$ if $x \lor y = y$
- Proof of transitive property. Must show that $x \lor y = y$ and $y \lor z = z$ implies $x \lor z = z$
  \[
  x \lor y = y \quad \text{and} \quad y \lor z = z \implies x \lor z = z
  = x \lor (y \lor z) \quad \text{(by assumption)}
  = (x \lor y) \lor z \quad \text{(by associativity)}
  = y \lor z \quad \text{(by assumption)}
  = z \quad \text{(by assumption)}
  
  \]

Properties of $\leq$
- Proof of asymmetry property. Must show that $x \lor y = y$ and $y \lor x = x$ implies $x = y$
  \[
  x = x \lor x \quad \text{(by assumption)}
  = x \lor y \quad \text{(by commutativity)}
  = y \quad \text{(by assumption)}
  
  \]
- Proof of reflexivity property. Must show that $x \lor x = x$
  \[
  x \lor x = x \quad \text{(by idempotence)}
  
  \]

Proof of $x \lor y = \sup \{x, y\}$
- Consider any upper bound $u$ for $x$ and $y$.
- Given $x \lor u = u$ and $y \lor u = u$, must show $x \lor y \leq u$, i.e., $(x \lor y) \lor u = u$
  \[
  u = x \lor u \quad \text{(by assumption)}
  = X \lor (y \lor u) \quad \text{(by assumption)}
  = (x \lor y) \lor u \quad \text{(by associativity)}
  
  \]
Proof of $x \land y = \inf \{x, y\}$
- Consider any lower bound $l$ for $x$ and $y$.
- Given $x \land l = l$ and $y \land l = l$, must show $l \leq x \land y$, i.e., $(x \land y) \land l = l$
  \[
  l = x \land l \quad \text{(by assumption)} \hfill \\
  = x \land (y \land l) \quad \text{(by assumption)} \hfill \\
  = (x \land y) \land l \quad \text{(by associativity)}
  \]

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 $\lor$ to combine values at control-flow join points

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:
    $f(x \lor y) = f(x) \lor f(y)$
  - Distributivity implies monotonicity

Distributivity Implies Monotonicity
- Proof of distributivity implies monotonicity
- Assume $f(x \lor y) = f(x) \lor f(y)$
- Must show: $x \lor y = y$ implies $f(x) \lor f(y) = f(y)$
  \[
  f(y) = f(x \lor y) \quad \text{(by assumption)} \hfill \\
  = f(x) \lor f(y) \quad \text{(by distributivity)}
  \]
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 = \lor \{ out_m. m \in \text{pred}(n) \}
  – in_{n_0} = \perp

Dataflow Equations

• Compiler processes program to obtain a set of dataflow equations
  \begin{align*}
  \text{out}_n &:= f_n(\text{in}_n) \\
  \text{in}_n &:= \lor \{ \text{out}_m. m \in \text{pred}(n) \}
  \end{align*}
• 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 \emptyset do
  remove a node n from worklist
  in_n := \lor \{ out_m. m \in \text{pred}(n) \}
  out_n := f_n(in_n)
  if out_n changed then
    worklist := worklist \cup \text{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_n changes, put succ(m) on worklist. Consider any node n \in \text{succ}(m). It will eventually come off worklist and algorithm will set
  \begin{align*}
  \text{in}_n &:= \lor \{ \text{out}_m. m \in \text{pred}(n) \}
  \end{align*}
to ensure that in_n = \lor \{ out_m. m \in \text{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)
- $\lor = \cup$ (order is $\subseteq$)
- $\bot = \emptyset$
- $F$ = 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
  - $f(x) = \text{GEN} \cup (x-\text{KILL})$

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

Does Reaching Definitions Framework Satisfy Properties?

- Identity
- Distributivity
- Composition

Available Expressions

- $P$ = powerset of set of all expressions in program (all subsets of set of expressions)
- $\lor = \land$ (order is $\subseteq$)
- $\bot = P$ (but $\text{in}_{\bot} = \emptyset$
- $F$ = 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

General Result

All GEN/KILL transfer function frameworks satisfy
- Identity
- Distributivity
- Composition

Properties
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
  - \( \text{in}_n \) – value at program point before \( n \)
  - \( \text{out}_n \) – value at program point after \( n \)
  - \( f_n \) – transfer function for \( n \) (given \( \text{out}_n \), computes \( \text{in}_n \))
- Require that solution satisfy
  - \( \forall n. \text{in}_n = f_n(\text{out}_n) \)
  - \( \forall n \not\in N_{\text{final}}. \text{out}_n = \cup \{ \text{in}_m. m \in \text{succ}(n) \} \)
  - \( \forall n \in N_{\text{final}}. \text{out}_n = \bot \)

Worklist Algorithm for Solving Backward Dataflow Equations

```plaintext
for each \( n \) do
  \( \text{in}_n := f_n(\bot) \)
  \( \text{worklist} := N \)
while \( \text{worklist} \neq \emptyset \) do
  remove a node \( n \) from \( \text{worklist} \)
  \( \text{out}_n := \cup \{ \text{in}_m. m \in \text{succ}(n) \} \)
  \( \text{in}_n := f_n(\text{out}_n) \)
if \( \text{in}_n \) changed then
  \( \text{worklist} := \text{worklist} \cup \text{pred}(n) \)
```

Live Variables

- \( P = \text{powerset of set of all variables in program} \)
  (all subsets of set of variables in program)
- \( \cup = \cup \) (order is \( \subseteq \))
- \( \bot = \emptyset \)
- \( F = \text{all functions } f \) 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:
    - \( s_0; s_1; \ldots; s_n \), where each \( s_j \in ST \)
    - \( s_{n+1} \) generated from \( s_n \) 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 \( s \)
  \( AF(s) \leq \text{in}_n \)
  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 \([-,0,+]\)
  \[
  \begin{array}{c}
  \text{TOP} \\
  - \\
  0 \\
  + \\
  \text{BOT}
  \end{array}
  \]
- Actual lattice records a value for each variable
  - Example element: \([a\to+], b\to0, c\to-]\)

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

Operation \( \odot \) on Lattice

<table>
<thead>
<tr>
<th>( \odot )</th>
<th>BOT</th>
<th>-</th>
<th>0</th>
<th>+</th>
<th>TOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOT</td>
<td>BOT</td>
<td>-</td>
<td>0</td>
<td>+</td>
<td>TOP</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>TOP</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>-</td>
<td>0</td>
<td>+</td>
<td>TOP</td>
</tr>
<tr>
<td>TOP</td>
<td>TOP</td>
<td>TOP</td>
<td>0</td>
<td>TOP</td>
<td>TOP</td>
</tr>
</tbody>
</table>

Transfer Functions

- If \( n \) of the form \( v = c \)
  - \( f_c(x) = x[v\to+] \) if \( c \) is positive
  - \( f_c(x) = x[v\to0] \) if \( c \) is 0
  - \( f_c(x) = x[v\to-] \) if \( c \) is negative
- If \( n \) of the form \( v_1 = v_2 \star v_3 \)
  - \( f_n(x) = x[v_1\to x[v_1] \odot x[v_3]] \)

Abstraction Function

- \( \text{AF}(s)[v] = \text{sign of } v \)
  - \( \text{AF}(s)[a\to-5, b\to0, c\to2)] = [a\to+, b\to0, c\to+] \)
- Establishes meaning of the analysis results
  - Always has that sign in actual execution
- Correctness condition:
  - \( \forall v. \text{AF}(s)[v] \leq n[v] \) (\( n \) is node for \( s \))
- Two sources of imprecision
  - Abstraction Imprecision - concrete values (integers) abstracted as lattice values (-0, and +)
  - Control Flow Imprecision - one lattice value for all different possible flow of control possibilities

Imprecision Example

Abstraction Imprecision:
\([a\to-1] \) abstracted as \([a\to+]\)
\[
\begin{array}{c}
[a\to+] \\
b = -1 \\
[a\to+, b\to+] \\
[a\to+, b\to-] \\
c = a^b
\end{array}
\]
Control Flow Imprecision:
\([b\to \text{TOP}] \) summarizes results of all executions. In any execution state \( s \), \( \text{AF}(s)[b] = \text{TOP} \)
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 $x \leq y$, 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

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 $p = n_0, n_1, \ldots, n_k$ 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: $f_p(\bot) = (f_{n_k}(f_{n_{k-1}}(\ldots f_{n_1}(f_{n_0}(\bot)) \ldots)) \leq \text{in}_n$
• So the solution must have the property that $\vee\{f_p(\bot) : p \text{ is a path to } n\} \leq \text{in}_n$
  and ideally $\vee\{f_p(\bot) : p \text{ is a path to } n\} = \text{in}_n$

Soundness Proof of Analysis Algorithm

• Property to prove:
  For all paths $p$ to $n$, $f_p(\bot) \leq \text{in}_n$
• Proof is by induction on length of $p$
  – Uses monotonicity of transfer functions
  – Uses following lemma
• Lemma:
  Worklist algorithm produces a solution such that $f_p(\text{in}_n) = \text{out}_n$
  if $n \in \text{pred}(m)$ then $\text{out}_n \leq \text{in}_m$

Proof

• Base case: $p$ is of length 1
  – Then $p = n_0$ and $f_p(\bot) = \bot = \text{in}_{n_0}$$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

- \( p = n_0, \ldots, n_k, n \)
- Must show \( f_k(f_{k-1}(\ldots f_{n_1}(f_{n_0}(\ldots)) \ldots)) \leq \text{in}_n \)
  - By induction, \( f_{k-1}(\ldots f_{n_1}(f_{n_0}(\ldots)) \ldots) \leq \text{in}_{n_k} \)
  - Apply \( f_k \) to both sides, by monotonicity we get \( f_k(f_{k-1}(\ldots f_{n_1}(f_{n_0}(\ldots)) \ldots)) \leq f_k(\text{in}_{n_k}) \)
  - By lemma, \( f_k(\text{in}_{n_k}) = \text{out}_{n_k} \)
  - By lemma, \( \text{out}_{n_k} \leq \text{in}_n \)
  - By transitivity, \( f_k(f_{k-1}(\ldots f_{n_1}(f_{n_0}(\ldots)) \ldots)) \leq \text{in}_n \)

Distributivity

- Distributivity preserves precision
- If framework is distributive, then worklist algorithm produces the meet over paths solution
  - For all \( n \):
    \[ \vee \{ f_p(\bot) \mid p \text{ is a path to } n \} = \text{in}_n \]

Lack of Distributivity Example

- Constant Calculator
- Flat Lattice on Integers

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

Lack of Distributivity Anomaly

- Transfer Functions
  - If \( n \) of the form \( v = c \)
    - \( f_n(x) = x[v \downarrow c] \)
  - 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 \( c = a + b \)
      - \( f([a \rightarrow 3, b \rightarrow 2]) \vee f([a \rightarrow 2, b \rightarrow 3]) = [a \rightarrow \text{TOP}, b \rightarrow \text{TOP}, c \rightarrow 5] \)
      - \( f([a \rightarrow 3, b \rightarrow 2] \cdot [a \rightarrow 2, b \rightarrow 3]) = f([a \rightarrow \text{TOP}, b \rightarrow \text{TOP}]) = [a \rightarrow \text{TOP}, b \rightarrow \text{TOP}, c \rightarrow \text{TOP}] \)

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 \( n \), \( AF(s) \leq \text{in}_n \)
  - Meet over paths solutions, distributivity