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 $N_{\text{final}}$
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 $S$
  - $\lor$ - 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 \( \forall y \in S. \ x \leq y \)
  - \( x \in P \) 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 $x \neq y$
- $x$ 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 $x \land y$ and $x \lor y$ exist for all $x, y \in P$, then $P$ is a lattice.

• If $\land S$ 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 $x, y \in I$, $x \lor y = \max(x, y)$, $x \land y = \min(x, y)$
  - But $\lor I$ and $\land I$ 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 ($\bot$)
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 $x \lor y = y$
  – $x \leq y$ 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 $x \land y = x$
Connecting Lemma Proofs

• Proof of $x \lor y = y$ implies $x \leq y$
  – $y$ is an upper bound of $\{x, y\}$ implies $x \leq y$
• Proof of $x \land y = x$ implies $x \leq y$
  – $x$ is a lower bound of $\{x, y\}$ implies $x \leq y$
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

• Intuitive concept of \( \lor \) and \( \land \) as information combination operators (or, and)
Algebraic Properties of Lattices

Assume arbitrary operations $\lor$ and $\land$ such that

- $(x \lor y) \lor z = x \lor (y \lor z)$ (associativity of $\lor$)
- $(x \land y) \land z = x \land (y \land z)$ (associativity of $\land$)
- $x \lor y = y \lor x$ (commutativity of $\lor$)
- $x \land y = y \land x$ (commutativity of $\land$)
- $x \lor x = x$ (idempotence of $\lor$)
- $x \land x = x$ (idempotence of $\land$)
- $x \lor (x \land y) = x$ (absorption of $\lor$ over $\land$)
- $x \land (x \lor y) = x$ (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 \land 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 $x \lor z = z$

$$x \lor z = x \lor (y \lor z) \quad \text{(by assumption)}$$
$$= (x \lor y) \lor z \quad \text{(by associativity)}$$
$$= y \lor z \quad \text{(by assumption)}$$
$$= z \quad \text{(by assumption)}$$
Properties of $\leq$

• Proof of asymmetry property. Must show that $\forall x \forall y (x \lor y = y$ and $y \lor x = x$ implies $x = y$

  \[
  x = y \lor x \quad \text{(by assumption)}
  \]

  \[
  = x \lor y \quad \text{(by commutativity)}
  \]

  \[
  = y \quad \text{(by assumption)}
  \]

• Proof of reflexivity property. Must show that $\forall x (x \lor x = x$

  \[
  x \lor x = x \quad \text{(by idempotence)}
  \]
Properties of $\leq$

- Induced operation $\leq$ agrees with original definitions of $\lor$ and $\land$, i.e.,
  \[ x \lor y = \sup \{x, y\} \]
  \[ x \land y = \inf \{x, y\} \]
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., $(x \lor y) \lor u = u$

\[
\begin{align*}
u &= x \lor u & \text{(by assumption)} \\
    &= x \lor (y \lor u) & \text{(by assumption)} \\
    &= (x \lor y) \lor u & \text{(by associativity)}
\end{align*}
\]
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., $(x \land y) \land l = l$

\[
\begin{align*}
l &= x \land l \quad \text{(by assumption)} \\
    &= x \land (y \land l) \quad \text{(by assumption)} \\
    &= (x \land y) \land l \quad \text{(by associativity)}
\end{align*}
\]
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

- 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:
  $x \leq y$ 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 $f(x) \lor f(y) = f(y)$
  
  \[
  f(y) = f(x \lor y) \quad \text{(by assumption)}
  \]
  
  \[
  = f(x) \lor f(y) \quad \text{(by distributivity)}
  \]
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
  - $\forall n. \ out_n = f_n(in_n)$
  - $\forall n \neq n_0. \ in_n = \lor \{ \ out_m . \ m \ in \ pred(n) \}$
  - $in_{n_0} = I$
  - Where $I$ summarizes information at start of program
Dataflow Equations

• Compiler processes program to obtain a set of dataflow equations

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

\[
\text{in}_n := \bigvee \{ \text{out}_m \cdot m \text{ in pred}(n) \}
\]

• Conceptually separates analysis problem from program
Worklist Algorithm for Solving Forward Dataflow Equations

for each $n$ do $\text{out}_n := f_n(\perp)$

$\text{in}_{n_0} := I$; $\text{out}_{n_0} := f_{n_0}(I)$

worklist := $N - \{ n_0 \}$

while worklist $\neq \emptyset$ do

remove a node $n$ from worklist

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

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

if $\text{out}_n$ 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 $out_m$ changes, put $succ(m)$ on worklist.
  Consider any node $n \in succ(m)$. It will eventually come off worklist and algorithm will set
  
  $in_n := \lor \{ out_m . m \in pred(n) \}$
  
  to ensure that $in_n = \lor \{ out_m . m \in pred(n) \}$
- 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)
- $\lor = \cup$ (order is $\subseteq$)
- $\bot = \emptyset$
- $I = \text{in}_{n_0} = \bot$
- $F =$ 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)
  - $x \subseteq x$ (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)
    $$f(x) \cup f(y) = (a \cup (x - b)) \cup (a \cup (y - b))$$
    $$= a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b)$$
    $$= f(x \cup y)$$
Does Reaching Definitions Framework Satisfy Properties?

- What about composition?
  - 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_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))
    
    - 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

- $P = \text{powerset of set of all expressions in program (all subsets of set of expressions)}$
- $\lor = \cap$ (order is $\supseteq$)
- $\bot = P$
- $I = \text{in}_{n_0} = \emptyset$
- $F = \text{all functions } f \text{ of the form } f(x) = a \cup (x-b)$
  - $b$ 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
  – \( \text{in}_n \) – 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 satisfies
  – \( \forall n. \, \text{in}_n = f_n(\text{out}_n) \)
  – \( \forall n \notin N_{\text{final}}. \, \text{out}_n = \lor \{ \text{in}_m \. \, m \in \text{succ}(n) \} \)
  – \( \forall n \in N_{\text{final}} = \text{out}_n = O \)
  – Where \( O \) summarizes information at end of program
Worklist Algorithm for Solving Backward Dataflow Equations

for each $n$ do $\text{in}_n := f_n(\perp)$
for each $n \in N_{\text{final}}$ do $\text{out}_n := O; \text{in}_n := f_n(O)$

worklist := $N - N_{\text{final}}$

while worklist $\neq \emptyset$ do
  remove a node $n$ from worklist
  $\text{out}_n := \lor \{ \text{in}_m . m \in \text{succ}(n) \}$
  $\text{in}_n := f_n(\text{out}_n)$
  if $\text{in}_n$ changed then
    worklist := worklist $\cup$ pred($n$)
Live Variables

• \( P = \text{powerset of set of all variables in program} \) (all subsets of set of variables in program)

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

• \( \bot = \emptyset \)

• \( O = \emptyset \)

• \( F = \text{all functions } f \text{ of the form } f(x) = a \cup (x-b) \)
  - \( b \) 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:
  \(- s_0; s_1; \ldots; s_k, \text{where each } s_i \in ST\)
  \(- s_{i+1} \text{ generated from } s_i \text{ by executing basic block to} \)
    • 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: $P = \text{flat lattice on } \{-,0,+\}$
  
  ![Lattice Diagram]

- Actual lattice records a value for each variable
  - Example element: $[a \mapsto +, b \mapsto 0, c \mapsto -]$
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?
  - \( AF([x_1, ..., x_n]) = [\text{sign}(x_1), ..., \text{sign}(x_n)] \)
  - Where \( \text{sign}(x) = 0 \) if \( x = 0 \), + if \( x > 0 \), - if \( x < 0 \)
Operation $\otimes$ on Lattice

<table>
<thead>
<tr>
<th>$\otimes$</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>BOT</td>
<td>0</td>
<td>BOT</td>
<td>BOT</td>
</tr>
<tr>
<td>-</td>
<td>BOT</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>BOT</td>
<td>-</td>
<td>0</td>
<td>+</td>
<td>TOP</td>
</tr>
<tr>
<td>TOP</td>
<td>BOT</td>
<td>TOP</td>
<td>0</td>
<td>TOP</td>
<td>TOP</td>
</tr>
</tbody>
</table>
Transfer Functions

• If n of the form \( v = c \)
  - \( f_n(x) = x[v \rightarrow+] \) 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 \( v_1 = v_2 \ast v_3 \)
  - \( f_n(x) = x[v_1 \rightarrow x[v_2] \otimes x[v_3]] \)

• \( I = \text{TOP} \)
  (uninitialized variables may have any sign)
Example

\[ a = 1 \]

\[ \begin{align*}
  b &= -1 \\
  b &= 1
\end{align*} \]

\[ \begin{align*}
  c &= a \times b \\
  c &= a \times b, b \rightarrow \text{TOP}
\end{align*} \]
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 $\lor$ 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

• \( AF(s)[v] = \text{sign of } v \)
  – \( AF(n, [a \rightarrow 5, b \rightarrow 0, c \rightarrow -2]) = [a \rightarrow +, b \rightarrow 0, c \rightarrow -] \)

• Establishes meaning of the analysis results
  – If analysis says variable has a given sign
  – Always has that sign in actual execution

• Correctness condition:
  – \( \forall v. \ AF(s)[v] \leq \text{in}_n[v] \) (\( n \) is node for \( s \))
  – Reflects possibility of imprecision
Abstraction Function Soundness

- Will show
  \[ \forall v. \text{AF}(s)[v] \leq \text{in}_n[v] \] (n is node for s) by induction on length of computation that produced s

- Base case:
  - \[ \forall v. \text{in}_{n_0}[v] = \text{TOP} \], which implies that
  - \[ \forall v. \text{AF}(s)[v] \leq \text{TOP} \]
Induction Step

• Assume ∀ v. AF(s)[v] ≤ 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 ∀ v. AF(s_p)[v] ≤ 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] ≤ in_n[v]
      – If x≠v, s[x] = s_p[x] and out_p[x] = in_p[x], so
        AF(s)[x] = AF(s_p)[x] ≤ in_p[x] = out_p[x] ≤ 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 $i n_i \in \text{pred}(n_{i+1})$)

• The solution must take this path into account:
  $$f_p(\bot) = (f_{nk}(f_{nk-1}(...f_{n1}(f_{n0}(\bot)) ...)) \leq in_n$$

• So the solution must have the property that
  $$\vee\{f_p(\bot) . p \text{ is a path to } n\} \leq in_n$$
  and ideally
  $$\vee\{f_p(\bot) . p \text{ is a path to } n\} = in_n$$
Soundness Proof of Analysis Algorithm

• Property to prove:
  For all paths $p$ to $n$, $f_p(\perp) \leq \text{in}_n$

• Proof is by induction on length of $p$
  – Uses monotonicity of transfer functions
  – Uses following lemma

• Lemma:
  Worklist algorithm produces a solution such that
  \[
  f_n(\text{in}_n) = \text{out}_n
  \]
  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 \( f_p(\perp) = \perp = \text{in}_{n_0} \)

• Induction step:
  – Assume theorem for all paths of length \( k \)
  – Show for an arbitrary path \( p \) of length \( k+1 \)
Induction Step Proof

- $p = n_0, \ldots, n_k, n$
- Must show $f_k(f_{k-1}(\ldots f_{n1}(f_{n0}(\bot)) \ldots)) \leq \text{in}_n$
  - By induction $(f_{k-1}(\ldots f_{n1}(f_{n0}(\bot)) \ldots)) \leq \text{in}_{nk}$
  - Apply $f_k$ to both sides, by monotonicity we get $f_k(f_{k-1}(\ldots f_{n1}(f_{n0}(\bot)) \ldots)) \leq f_k(\text{in}_{nk})$
  - By lemma, $f_k(\text{in}_{nk}) = \text{out}_{nk}$
  - By lemma, $\text{out}_{nk} \leq \text{in}_n$
  - By transitivity, $f_k(f_{k-1}(\ldots f_{n1}(f_{n0}(\bot)) \ldots)) \leq \text{in}_n$
Distributivity

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

- Constant Calculator
- Flat Lattice on Integers

- Actual lattice records a value for each variable
  - Example element: [a→3, b→2, c→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 \( c = a + b \)
  \( f([a \rightarrow 3, \ b \rightarrow 2]) \lor f([a \rightarrow 2, \ b \rightarrow 3]) = [a \rightarrow \text{TOP}, \ b \rightarrow \text{TOP}, \ c \rightarrow 5] \)
  \( f([a \rightarrow 3, \ b \rightarrow 2] \lor [a \rightarrow 2, \ b \rightarrow 3]) = f([a \rightarrow \text{TOP}, \ b \rightarrow \text{TOP}]) = [a \rightarrow \text{TOP}, \ b \rightarrow \text{TOP}, \ c \rightarrow \text{TOP}] \)
Lack of Distributivity Anomaly

\[ a = 2 \quad a = 3 \]
\[ b = 3 \quad b = 2 \]
\[ [a \rightarrow 2, \ b \rightarrow 3] \quad [a \rightarrow 3, \ b \rightarrow 2] \]
\[ [a \rightarrow \text{TOP}, \ b \rightarrow \text{TOP}] \]
\[ c = a + b \]

Lack of Distributivity Imprecision:
\[ [a \rightarrow \text{TOP}, \ b \rightarrow \text{TOP}, \ c \rightarrow 5] \text{ more precise} \]
\[ [a \rightarrow \text{TOP}, \ b \rightarrow \text{TOP}, \ c \rightarrow \text{TOP}] \]

What is the meet over all paths solution?
How to Make Analysis Distributive

• Keep combinations of values on different paths

\[ \begin{align*}
& a = 2, b = 3 \\
& a = 3, b = 2 \\
& c = a + b
\end{align*} \]

\[ \{ [a \rightarrow 2, b \rightarrow 3], [a \rightarrow 3, b \rightarrow 2] \} \]
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

\[
a = x + y
\]

\[
i == 0
\]

\[
b = x + y;
\]

\[
nop
\]
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