Static Program Analysis
Part 3 – lattices and fixpoints

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University
Flow-sensitivity

• Type checking is (usually) flow-insensitive:
  – statements may be permuted without affecting typability
  – constraints are naturally generated from AST nodes

• Other analyses must be flow-sensitive:
  – the order of statements affects the results
  – constraints are naturally generated from control flow graph nodes
Sign analysis

• Determine the sign (+, −, 0) of all expressions

• The Sign lattice:

States are then modeled by the map lattice $\text{Vars} \rightarrow \text{Sign}$ where $\text{Vars}$ is the set of variables in the program
Generating constraints

1. var a, b;
2. a = 42;
3. b = a + input;
4. a = a - b;

\[
x_1 = [a \mapsto ?, b \mapsto ?]
\]
\[
x_2 = x_1[a \mapsto +]
\]
\[
x_3 = x_2[b \mapsto x_2(a)+?]
\]
\[
x_4 = x_3[a \mapsto x_3(a) - x_3(b)]
\]
Sign analysis constraints

• The variable $[v]$ denotes a map that gives the sign value for all variables at the program point after node $v$

• For variable declarations:
  $[\text{var } x_1, \ldots, x_n] = \text{JOIN}(v)[x_1 \mapsto ?, \ldots, x_n \mapsto ?]$

• For assignments:
  $[x = E] = \text{JOIN}(v)[x \mapsto \text{eval}(\text{JOIN}(v), E)]$

• For all other nodes:
  $[v] = \text{JOIN}(v)$

where $\text{JOIN}(v) = \bigsqcup [w]_{w \in \text{pred}(v)}$ combines information from predecessors (explained later…)

Evaluating signs

- The \( \text{eval} \) function is an \textit{abstract evaluation}:
  - \( \text{eval}(\sigma, x) = \sigma(x) \)
  - \( \text{eval}(\sigma, \text{intconst}) = \text{sign}(\text{intconst}) \)
  - \( \text{eval}(\sigma, E_1 \text{ op } E_2) = \overline{\text{op}}(\text{eval}(\sigma, E_1), \text{eval}(\sigma, E_2)) \)

- \( \sigma: \text{Vars} \rightarrow \text{Sign} \) is an abstract state

- The \( \text{sign} \) function gives the sign of an integer

- The \( \overline{\text{op}} \) function is an abstract evaluation of the given operator
### Abstract operators

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>↓</th>
<th>0</th>
<th>-</th>
<th>+</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>↓</th>
<th>0</th>
<th>-</th>
<th>+</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>*</th>
<th>↓</th>
<th>0</th>
<th>-</th>
<th>+</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>/</th>
<th>↓</th>
<th>0</th>
<th>-</th>
<th>+</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>&gt;</th>
<th>↓</th>
<th>0</th>
<th>-</th>
<th>+</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>==</th>
<th>↓</th>
<th>0</th>
<th>-</th>
<th>+</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Increasing precision

• Some loss of information:
  – \((2 > 0) == 1\) is analyzed as ?
  – ++/ is analyzed as ?, since e.g. \(\frac{1}{2}\) is rounded down

• Use a richer lattice for better precision:

• Abstract operators are now 8x8 tables
Partial orders

• Given a set $S$, a partial order $\sqsubseteq$ is a binary relation on $S$ that satisfies:
  – reflexivity: $\forall x \in S: x \sqsubseteq x$
  – transitivity: $\forall x, y, z \in S: x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  – anti-symmetry: $\forall x, y \in S: x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y$

• Can be illustrated by a Hasse diagram (if finite)
Upper and lower bounds

• Let $X \subseteq S$ be a subset

• We say that $y \in S$ is an upper bound $(X \sqsubseteq y)$ when
  $\forall x \in X: x \sqsubseteq y$

• We say that $y \in S$ is a lower bound $(y \sqsubseteq X)$ when
  $\forall x \in X: y \sqsubseteq x$

• A least upper bound $\sqcup X$ is defined by
  $X \subseteq \sqcup X \land \forall y \in S: X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$

• A greatest lower bound $\sqcap X$ is defined by
  $\sqcap X \subseteq X \land \forall y \in S: y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$
A (complete) lattice is a partial order where $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$.

A lattice must have:
- a unique largest element, $\top = \sqcup S$
- a unique smallest element, $\bot = \sqcap S$

If $S$ is a finite set, then it defines a lattice iff:
- $\top$ and $\bot$ exist in $S$
- $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in S$ (where $x \sqcup y$ is notation for $\sqcup \{x, y\}$)
These partial orders are lattices
These partial orders are not lattices
The powerset lattice

- Every finite set $A$ defines a lattice $(2^A, \subseteq)$ where
  - $\bot = \emptyset$
  - $\top = A$
  - $x \sqcup y = x \cup y$
  - $x \sqcap y = x \cap y$
Lattice height

- The *height* of a lattice is the length of the longest path from $\bot$ to $\top$.
- The lattice $(2^A, \subseteq)$ has height $|A|$.
Map lattice

• If A is a set and L is a lattice, then we obtain the map lattice:

\[ A \to L = \{ [a_1 \mapsto x_1, a_2 \mapsto x_2, \ldots] \mid A=\{a_1, a_2, \ldots\} \land x_1, x_2, \ldots \in L_i \} \]

ordered pointwise

• ⊔ and ⊓ can be computed pointwise

• \( \text{height}(A \to L) = |A| \cdot \text{height}(L) \)
Product lattice

- If $L_1, L_2, ..., L_n$ are lattices, then so is the product:

$$L_1 \times L_2 \times ... \times L_n = \{ (x_1, x_2, ..., x_n) \mid x_i \in L_i \}$$

where $\sqsubseteq$ is defined pointwise.

- Note that $\sqcup$ and $\sqcap$ can be computed pointwise.

- $\text{height}(L_1 \times L_2 \times ... \times L_n) = \text{height}(L_1) + ... + \text{height}(L_n)$
Sum lattice

• If \( L_1, L_2, \ldots, L_n \) are lattices, then so is the sum:

\[
L_1 + L_2 + \ldots + L_n = \{ (i,x_i) \mid x_i \in L_i \setminus \{ \bot, \top \} \} \cup \{ \bot, \top \}
\]

where:

– \( \bot \) and \( \top \) are as expected
– \((i,x) \sqsubseteq (j,y)\) if and only if \( i=j \) and \( x \sqsubseteq y \)

• \( \text{height}(L_1 + L_2 + \ldots + L_n) = \max\{\text{height}(L_i)\} \)
If $A$ is a set, then $\text{flat}(A)$ is a lattice:

\[
\begin{array}{c}
\top \\
\downarrow & \searrow & \swarrow \\
a_1 & a_2 & \ldots & a_n \\
\downarrow & \swarrow & \nearrow \\
\bot
\end{array}
\]

- $\text{height}(\text{flat}(A)) = 2$
Lift lattice

• If $L$ is a lattice, then so is $\text{lift}(L)$, which is:

\[ \text{height}(\text{lift}(L)) = \text{height}(L) + 1 \]
Generating constraints, again

```
1   var a,b;
2   a = 42;
3   b = a + input;
4   a = a - b;
```

\[
x_1 = [a \mapsto ?, b \mapsto ?]
\]
\[
x_2 = x_1[a \mapsto +]
\]
\[
x_3 = x_2[b \mapsto x_2(a)+?]
\]
\[
x_4 = x_3[a \mapsto x_3(a)-x_3(b)]
\]
Sign analysis constraints, revisited

- The variable $⟦v⟧$ denotes a map that gives the sign value for all variables at the program point after node $v$

- $⟦v⟧ ∈ States$ where $States = Vars → Sign$

- For variable declarations:
  
  $⟦\text{var } x_1, \ldots, x_n ⟧ = JOIN(v)[x_1 ↦ ?, \ldots, x_n ↦ ?]$

- For assignments:
  
  $⟦x = E⟧ = JOIN(v)[x ↦ eval(JOIN(v), E)]$

- For all other nodes:
  
  $⟦v⟧ = JOIN(v)$

where $JOIN(v) = \bigcup_{w ∈ pred(v)} [w]$ combines information from predecessors
Constraints

• From the program being analyzed, we have constraint variables $x_1, \ldots, x_n \in L$ and a collection of constraints:
  
  $x_1 = f_1(x_1, \ldots, x_n)$
  
  $x_2 = f_2(x_1, \ldots, x_n)$
  
  $\ldots$
  
  $x_n = f_n(x_1, \ldots, x_n)$

• These can be collected into a single function $f: L^n \rightarrow L^n$:
  
  $f(x_1, \ldots, x_n) = (f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n))$

• How do we find the least (i.e. most precise) value of $x_1, \ldots, x_n$ such that $x_1, \ldots, x_n = f(x_1, \ldots, x_n)$ (if that exists)???
Monotone functions

- A function $f : L \rightarrow L$ is *monotone* when:
  \[ \forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \]
- A function with several arguments is monotone if it is monotone in each argument
- Monotone functions are closed under composition
- As functions, $\sqcup$ and $\sqcap$ are both monotone
- A function is *extensive* when:
  \[ \forall x \in L : x \sqsubseteq f(x) \]
- Monotone is different from extensive
  - e.g. all constant functions are monotone
Monotonicity for the sign analysis

- The $\sqcup$ operator and map updates are monotone
- Compositions preserve monotonicity
- Are the abstract operators monotone?
- This is verified by a tedious manual inspection
- Or better, run an $O(n^3)$ algorithm for an $n \times n$ table:
  - $\forall x, y, x' \in L: x \sqsubseteq x' \Rightarrow x \overline{\text{op}} y \sqsubseteq x' \overline{\text{op}} y$
  - $\forall x, y, y' \in L: y \sqsubseteq y' \Rightarrow x \overline{\text{op}} y \sqsubseteq x \overline{\text{op}} y'$

  For variable declarations:
  $\llbracket \text{var } x_1, ..., x_n \rrbracket = \text{JOIN}(v)[x_1 \mapsto ?, ..., x_n \mapsto ?]$

  For assignments:
  $\llbracket x = E \rrbracket = \text{JOIN}(v)[x \mapsto \text{eval}(\text{JOIN}(v), E)]$
The fixed-point theorem

\[ x \in L \text{ is a fixed-point of } f: L \rightarrow L \text{ iff } f(x) = x \]

In a lattice with finite height, every monotone function \( f \) has a unique least fixed-point:

\[ \text{fix}(f) = \bigsqcup_{i \geq 0} f^i(\bot) \]
Proof of existence

• Clearly, $\bot \sqsubseteq f(\bot)$
• Since $f$ is monotone, we also have $f(\bot) \sqsubseteq f^2(\bot)$
• By induction, $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$
• This means that

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots \ f^i(\bot) \ldots$$

is an increasing chain
• $L$ has finite height, so for some $k$: $f^k(\bot) = f^{k+1}(\bot)$
• But then $\text{fix}(f) = f^k(\bot)$
Proof of unique least

• Assume that \( x \) is another fixed-point: \( x = f(x) \)
• Clearly, \( \bot \sqsubseteq x \)
• By induction, \( f^i(\bot) \sqsubseteq f^i(x) = x \)
• In particular, \( \text{fix}(f) = f^k(\bot) \sqsubseteq x \), i.e. \( \text{fix}(f) \) is least

• Uniqueness then follows from anti-symmetry
Computing fixed-points

The time complexity of $\text{fix}(f)$ depends on:

- the height of the lattice
- the cost of computing $f$
- the cost of testing equality

```plaintext
x = ⊥;
do {  
t = x;
x = f(x);
}
while (x ≠ t);
```
Intuition of monotonicity

• Recall that a function $f: L \rightarrow L$ is monotone when
  \[ \forall x, y \in L: x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \]

• $x \sqsubseteq y$ can be interpreted as “$x$ is at least as precise as $y$”
• $f$ is an “approximation improvement function”
• When $f$ is monotone:
  “more precise input cannot lead to less precise output”
Summary: lattice equations

• Let $L$ be a lattice with finite height

• A equation system is of the form:

\[
\begin{align*}
    x_1 &= f_1(x_1, \ldots, x_n) \\
    x_2 &= f_2(x_1, \ldots, x_n) \\
    \vdots \\
    x_n &= f_n(x_1, \ldots, x_n)
\end{align*}
\]

where $x_i$ are variables and each $f_i: L^n \rightarrow L$ is monotone

• Note that $L^n$ is a product lattice
Solving equations

• Every equation system has a *unique least solution*, which is the least fixed-point of the function $f: \mathbb{L}^n \rightarrow \mathbb{L}^n$ defined by

$$f(x_1,\ldots,x_n) = (f_1(x_1,\ldots,x_n), \ldots, f_n(x_1,\ldots,x_n))$$

• A solution is always a fixed-point (for any kind of equation)

• The least one is the most precise
Solving inequations

- A *inequation system* is of the form

\[
x_1 \sqsubseteq f_1(x_1, \ldots, x_n) \\
x_2 \sqsubseteq f_2(x_1, \ldots, x_n) \\
\vdots \\
x_n \sqsubseteq f_n(x_1, \ldots, x_n)
\]

\[
x_1 \sqsupseteq f_1(x_1, \ldots, x_n) \\
x_2 \sqsupseteq f_2(x_1, \ldots, x_n) \\
\vdots \\
x_n \sqsupseteq f_n(x_1, \ldots, x_n)
\]

- Can be solved by exploiting the facts that

\[
x \sqsubseteq y \iff x = x \sqcap y
\]

and

\[
x \sqsupseteq y \iff x = x \sqcup y
\]
The monotone framework (1/2)

- A CFG to be analyzed, nodes $\text{Nodes} = \{v_1, v_2, \ldots, v_n\}$
- A finite-height lattice $L$ of possible answers
  - fixed or parametrized by the given program
- A constraint variable $\lbrack v \rbrack \in L$ for every CFG node $v$
- A dataflow constraint for each syntactic construct
  - relates the value of $\lbrack v \rbrack$ to the variables for other nodes
  - typically a node is related to its neighbors
  - the constraints must be monotone functions:
    \[
    \lbrack v_i \rbrack = f_i(\lbrack v_1 \rbrack, \lbrack v_2 \rbrack, \ldots, \lbrack v_n \rbrack)
    \]
The monotone framework (2/2)

• Extract all constraints for the CFG

• Solve constraints using the fixed-point algorithm:
  – we work in the lattice $L^n$ where $L$ is a lattice describing abstract states
  – computing the least fixed-point of the combined function:
    $$f(x_1,\ldots,x_n) = (f_1(x_1,\ldots,x_n), \ldots, f_n(x_1,\ldots,x_n))$$

• This solution gives an answer from $L$ for each CFG node
Generating and solving constraints

CFG

constraints

fixed-point solver

solution

[p] = &int
[q] = &int
[malloc] = &int
[x] = φ
[foo] = φ
[&n] = &int
[main] = () -> int
Lattice points as answers

- the trivial, useless answer
- our answer (the least fixed-point)
- the true answer
- safe answers
- unsafe answers

Conservative approximation...
The naive algorithm

\[ x = (\bot, \bot, \ldots, \bot); \]
\[ \text{do } \{ \]
\[ t = x; \]
\[ x = f(x); \]
\[ \} \text{ while } (x \neq t); \]

• Correctness ensured by the fixed point theorem
• Does not exploit any special structure of \( L^n \) or \( f \)
  (i.e. \( x \in L^n \) and \( f(x_1, \ldots, x_n) = (f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)) \))
The naive algorithm

<table>
<thead>
<tr>
<th></th>
<th>( f^0(\perp, \perp, \ldots, \perp) )</th>
<th>( f^1(\perp, \perp, \ldots, \perp) )</th>
<th>( \ldots )</th>
<th>( f^k(\perp, \perp, \ldots, \perp) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \perp )</td>
<td>( f_1^1(\perp, \perp, \ldots, \perp) )</td>
<td>( \ldots )</td>
<td>( f_1^k(\perp, \perp, \ldots, \perp) )</td>
</tr>
<tr>
<td>2</td>
<td>( \perp )</td>
<td>( f_2^1(\perp, \perp, \ldots, \perp) )</td>
<td>( \ldots )</td>
<td>( f_2^k(\perp, \perp, \ldots, \perp) )</td>
</tr>
<tr>
<td>\ldots</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( n )</td>
<td>( \perp )</td>
<td>( f_n^1(\perp, \perp, \ldots, \perp) )</td>
<td>( \ldots )</td>
<td>( f_n^k(\perp, \perp, \ldots, \perp) )</td>
</tr>
</tbody>
</table>

Computing each new entry is done using the previous row

- Without using the entries in the current row that have already been computed!
- And many entries are likely unchanged from row to row!
Chaotic iteration

Recall that $f(x_1,\ldots,x_n) = (f_1(x_1,\ldots,x_n), \ldots, f_n(x_1,\ldots,x_n))$

\[
x_1 = \perp; \ldots \ x_n = \perp;
\]
while $((x_1,\ldots,x_n) \neq f(x_1,\ldots,x_n))$ {
    pick $i$ nondeterministically such that $x_i \neq f_i(x_1, \ldots, x_n)$
    $x_i = f_i(x_1, \ldots, x_n)$;
}\}

We now exploit the special structure of $L^n$
– may require a higher number of iterations,
    but each iteration is generally cheaper
Correctness of chaotic iteration

- Let $x^j$ be the value of $x=(x_1, \ldots, x_n)$ in the j’th iteration of the naive algorithm
- Let $\underline{x}^j$ be the value of $x=(x_1, \ldots, x_n)$ in the j’th iteration of the chaotic iteration algorithm
- By induction in $j$, show $\forall j: \underline{x}^j \subseteq x^j$
- Chaotic iteration eventually terminates at a fixed point
- It must be identical to the result of the naive algorithm since that is the least fixed point
Towards a practical algorithm

• Computing $\exists i : \ldots$ in chaotic iteration is not practical

• Idea: predict $i$ from the analysis and the structure of the program!

• Example:
  In sign analysis, when we have processed a CFG node $v$, process $\text{succ}(v)$ next
The worklist algorithm (1/2)

- Essentially a specialization of chaotic iteration that exploits the special structure of $f$

- Most right-hand sides of $f_i$ are quite sparse:
  - constraints on CFG nodes do not involve all others

- Use a map:

  $$\text{dep}: \text{Nodes} \rightarrow 2^{\text{Nodes}}$$

  that for $v \in \text{Nodes}$ gives the variables $w$ where $v$ occurs on the right-hand side of the constraint for $w$
\( x_1 = \bot; \ldots \ x_n = \bot; \)
\( W = \{v_1, \ldots, v_n\}; \)
\( \text{while} \ \ (W \neq \emptyset) \ {\}
  \quad v_i = W.\text{removeNext}(); \\
  \quad y = f_i(x_1, \ldots, x_n); \\
  \quad \text{if} \ (y \neq x_i) \ {\}
    \quad \text{for} \ (v_j \in \text{dep}(v_i)) \ W.\text{add}(v_j); \\
    \quad x_i = y; \\
  \}
\}
Further improvements

• Represent the worklist as a priority queue
  – find clever heuristics for priorities

• Look at the graph of dependency edges:
  – build strongly-connected components
  – solve constraints bottom-up in the resulting DAG
Transfer functions

• The constraint functions in dataflow analysis usually have this structure:

\[
\llbracket \nu \rrbracket = t_\nu(JOIN(\nu))
\]

where \( t_\nu : States \rightarrow States \) is called the \textit{transfer function} for \( \nu \)

• Example:

\[
\llbracket x = E \rrbracket = JOIN(\nu)[x \mapsto eval(JOIN(\nu),E)]
\]

\[
= t_\nu(JOIN(\nu))
\]

where

\[
t_\nu(s) = s[x \mapsto eval(s,E)]
\]