Static Program Analysis

Part 3 – lattices and fixpoints

https://cs.au.dk/~amoeller/spa/

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

Flow-sensitivity

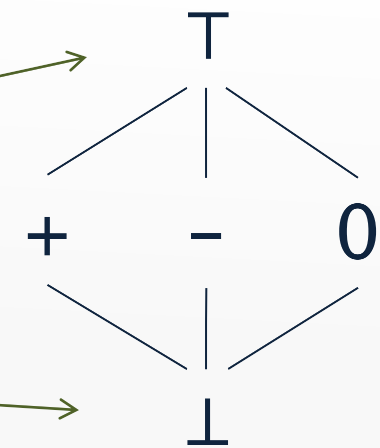
- Type checking is (usually) flow-*insensitive*:
  - statements may be permuted without affecting typability
  - constraints are naturally generated from *AST nodes*

- 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:

“any number”



“not of type number”
(or, “unreachable code”)

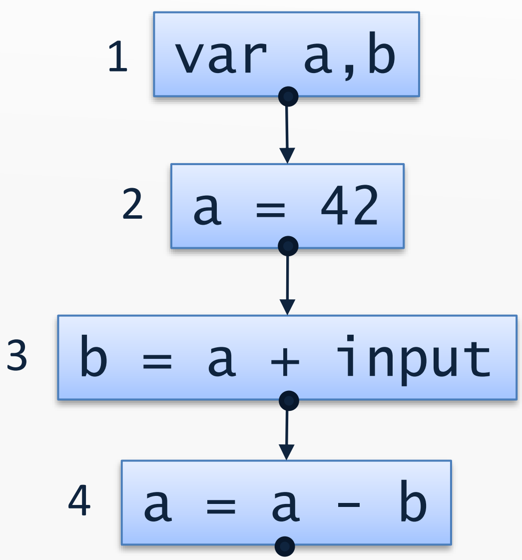
The terminology will be defined later – this is just an appetizer…

- States are modeled by the map lattice *Var* → *Sign*
  where *Var* is the set of variables in the program

Implementation: `TIP/src/tip/analysis/SignAnalysis.scala`

Generating constraints

```
var a,b;
a = 42;
b = a + input;
a = a - b;
```



$$x_1 = [a \mapsto \top, b \mapsto \top]$$

$$x_2 = x_1[a \mapsto +]$$

$$x_3 = x_2[b \mapsto x_2(a) + \top]$$

$$x_4 = x_3[a \mapsto x_3(a) - x_3(b)]$$

Sign analysis constraints

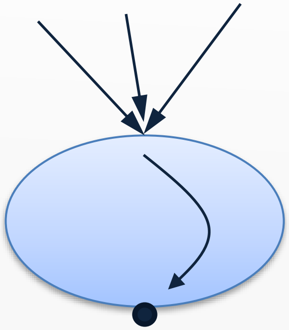
- The variable $[\![v]\!]$ denotes a map that gives the sign value for all variables at the program point *after* CFG node v

- For assignments:

  $[\![\, x = E \,]\!] = \mathit{JOIN}(\mathrm{v})[x \mapsto \mathit{eval}(\mathit{JOIN}(\mathrm{v}), E)]$
- For variable declarations:

  $[\![\, \mathtt{var}\ x_1, \ldots, x_n \,]\!] = \mathit{JOIN}(\mathrm{v})[x_1 \mapsto \top, \ldots, x_n \mapsto \top]$
- For all other nodes:

  $[\![\mathrm{v}]\!] = \mathit{JOIN}(\mathrm{v})$



where $\mathit{JOIN}(\mathrm{v}) = \bigsqcup_{\mathrm{w} \in \mathit{pred}(\mathrm{v})} [\![\mathrm{w}]\!]$

combines information from predecessors
(explained later…)

Evaluating signs

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

- σ: $Var \rightarrow Sign$ is an abstract state

- The *sign* function gives the sign of an integer

- The $\overline{\text{op}}$ function is an abstract evaluation of the given operator op

Abstract operators

+	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	-	+	⊤
-	⊥	-	-	⊤	⊤
+	⊥	+	⊤	+	⊤
⊤	⊥	⊤	⊤	⊤	⊤

-	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	-	⊤
-	⊥	-	⊤	-	⊤
+	⊥	+	+	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊤

*	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	0	0	0
-	⊥	0	+	-	⊤
+	⊥	0	-	+	⊤
⊤	⊥	0	⊤	⊤	⊤

/	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	⊥	0	0	⊤
-	⊥	⊥	⊤	⊤	⊤
+	⊥	⊥	⊤	⊤	⊤
⊤	⊥	⊥	⊤	⊤	⊤

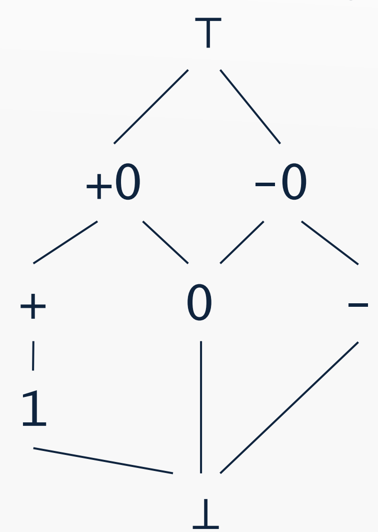
>	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	0	⊤
-	⊥	0	⊤	0	⊤
+	⊥	+	+	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊤

==	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	+	0	0	⊤
-	⊥	0	⊤	0	⊤
+	⊥	0	0	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊤

(assuming the subset of TIP with only integer values)

Increasing precision

- Some loss of information:
  - `(2>0)==1` is analyzed as $\top$
  - $+/+$ is analyzed as $\top$, since e.g. ½ is rounded down
- Use a richer lattice for better precision:

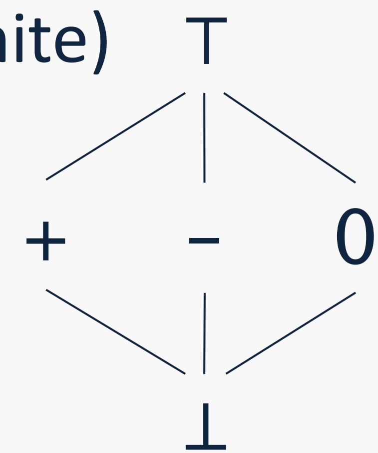


- Abstract operators are now 8×8 tables

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 $\bigsqcup X$ is defined by

  $X \sqsubseteq \bigsqcup X \wedge \forall y \in S: X \sqsubseteq y \Rightarrow \bigsqcup X \sqsubseteq y$
- A *greatest* lower bound $\bigsqcap X$ is defined by

  $\bigsqcap X \sqsubseteq X \wedge \forall y \in S: y \sqsubseteq X \Rightarrow y \sqsubseteq \bigsqcap X$

Lattices

- A *lattice* is a partial order where
  $x \sqcup y$ and $x \sqcap y$ exist for all $x,y \in S$ ($x \sqcup y$ is notation for $\bigsqcup\{x,y\}$)
- A *complete lattice* is a partial order where
  $\bigsqcup X$ and $\bigsqcap X$ exist for all $X \subseteq S$

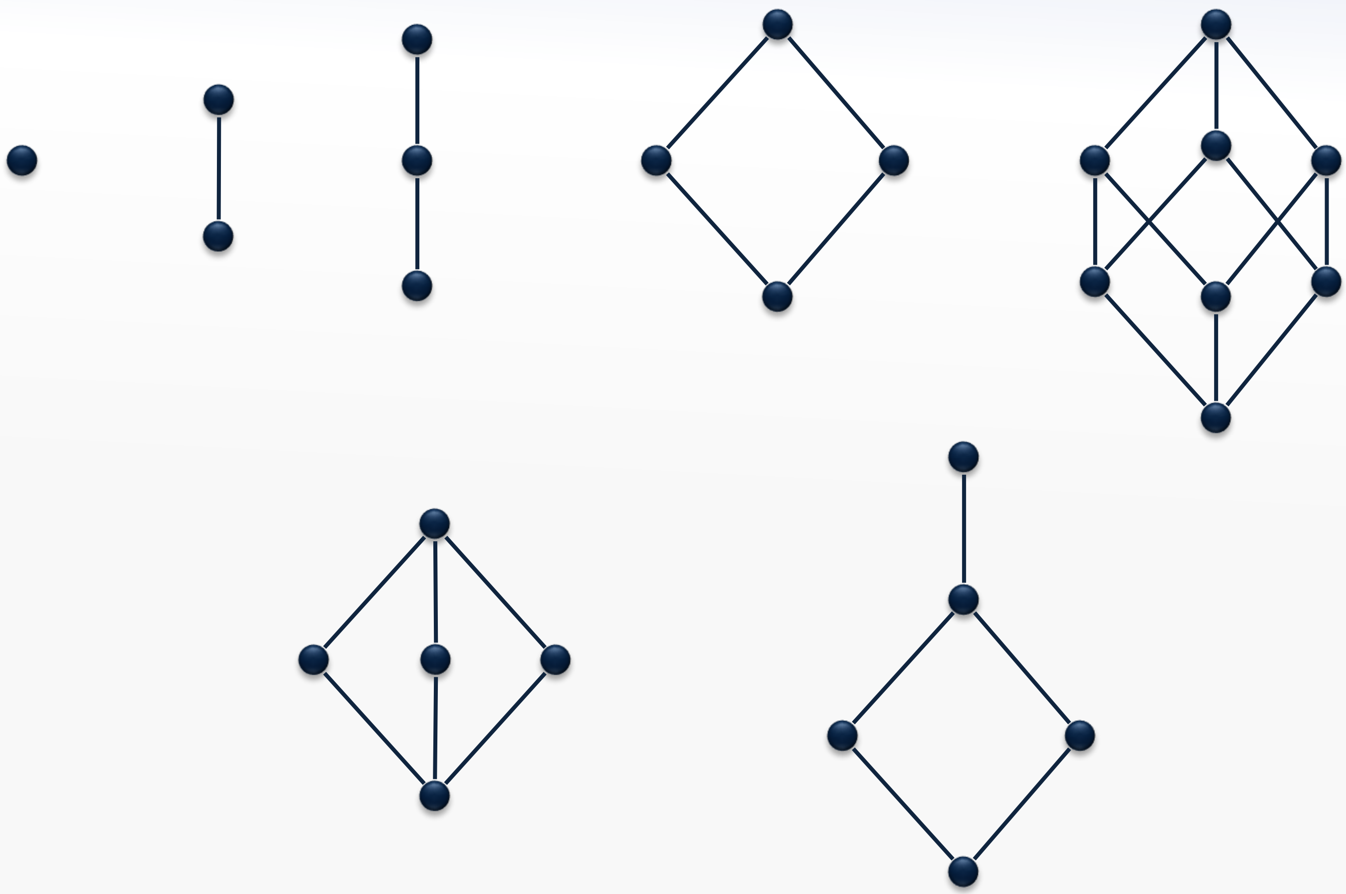
- A complete lattice must have
  - a unique largest element, $\top = \bigsqcup S$
  - a unique smallest element, $\bot = \bigsqcap S$

  (exercise)

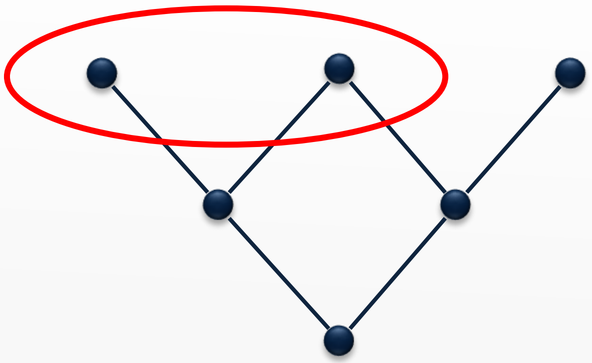
- A finite lattice is complete if $\top$ and $\bot$ exist

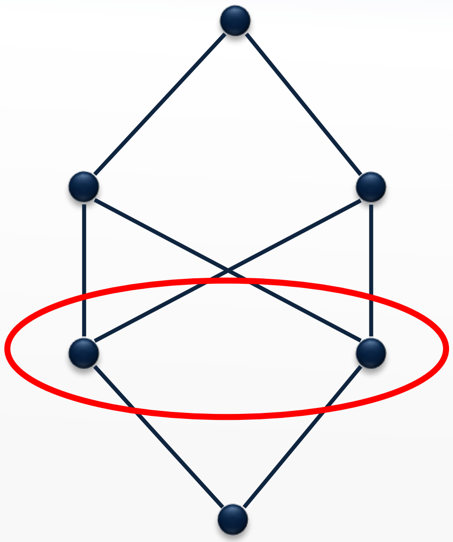
Implementation: `TIP/src/tip/lattices/`

These partial orders are lattices



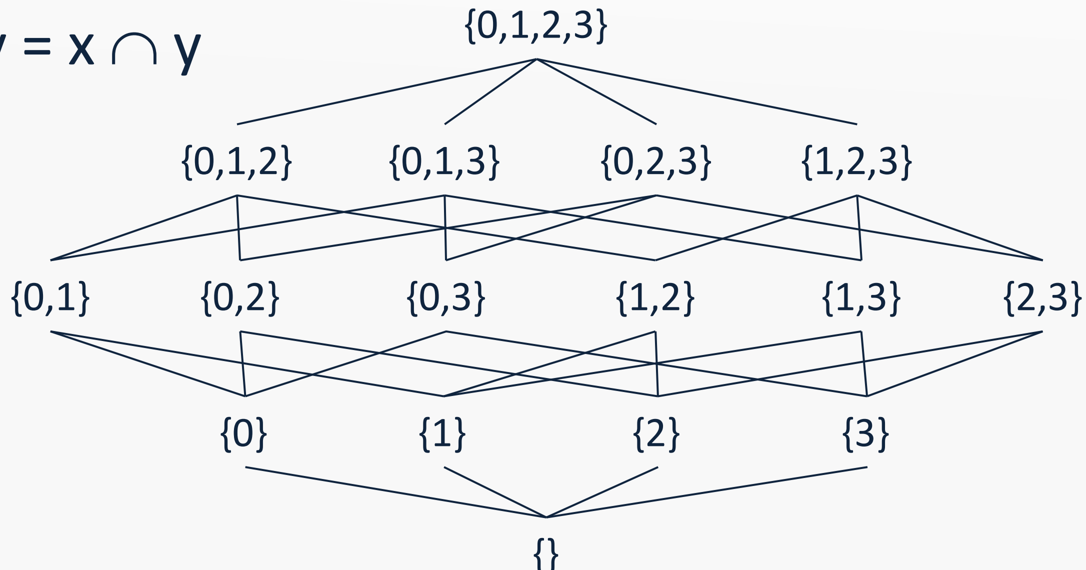
These partial orders are *not* lattices





The powerset lattice

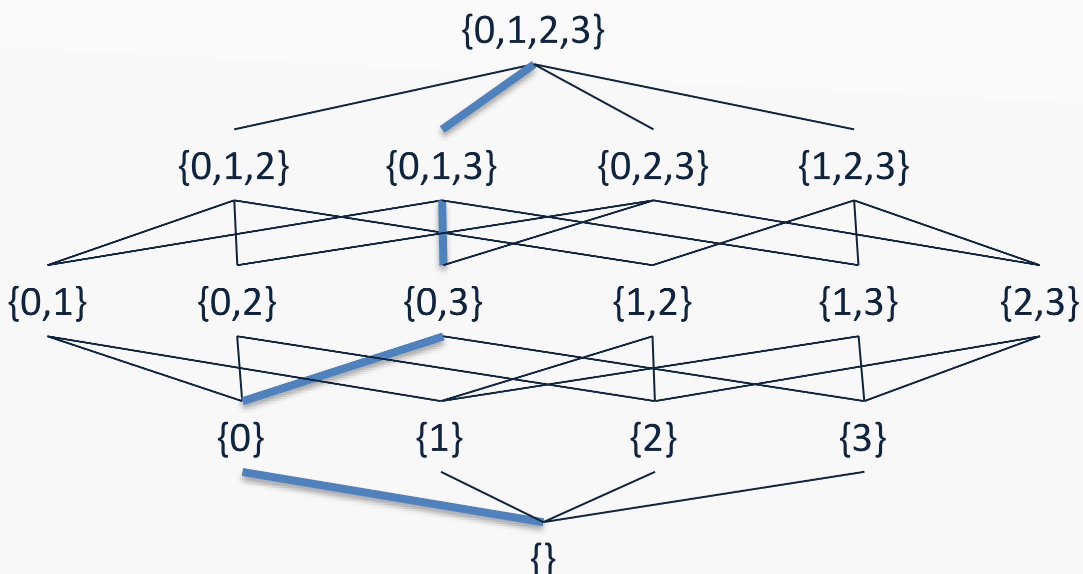
- Every finite set A defines a complete lattice ($\mathcal{P}$(A),$\subseteq$) where
  - $\bot = \varnothing$
  - $\top = A$
  - $x \sqcup y = x \cup y$
  - $x \sqcap y = x \cap y$



for A = {0,1,2,3}

Lattice height

- The *height* of a lattice is the length of the longest path from $\perp$ to $\top$
- The lattice $(\mathcal{P}(A),\subseteq)$ has height $|A|$



for A = {0,1,2,3}

Map lattice

- If A is a set and L is a complete lattice, then we obtain a complete lattice called a map lattice:

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

ordered pointwise

Example: $A \rightarrow L$ where

- A is the set of program variables
- L is the *Sign* lattice

- $\sqcup$ and $\sqcap$ can be computed pointwise
- $\mathit{height}(A \rightarrow L) = |A| \cdot \mathit{height}(L)$

Product lattice

- If $L_1, L_2, \ldots, L_n$ are complete lattices, then so is the *product*:

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

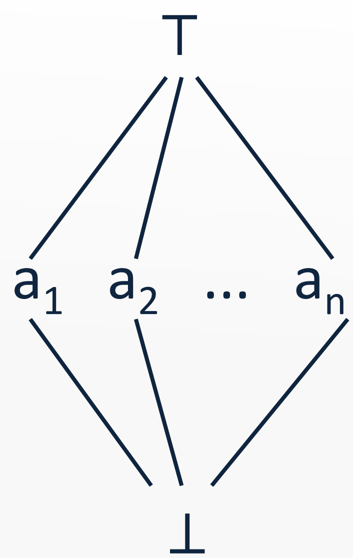
  where $\sqsubseteq$ is defined pointwise

- Note that $\sqcup$ and $\sqcap$ can be computed pointwise
- $height(L_1 \times L_2 \times \ldots \times L_n) = height(L_1) + \ldots + height(L_n)$

Example:
each L_i is the map lattice $A \to L$ from the previous slide, and n is the number of CFG nodes

Flat lattice

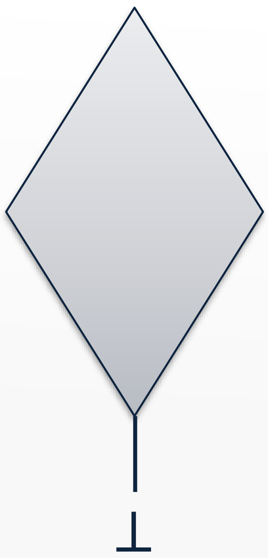
- If A is a set, then *flat*(A) is a complete lattice:



- *height*(*flat*(A)) = 2

Lift lattice

- If L is a complete lattice, then so is *lift*(L), which is:



- *height*(*lift*(L)) = *height*(L)+1

Sign analysis constraints, revisited

- The variable $[\![v]\!]$ denotes a map that gives the sign value for all variables at the program point *after* CFG node v

- $[\![v]\!] \in State$ where $State = Var \to Sign$

- For assignments:

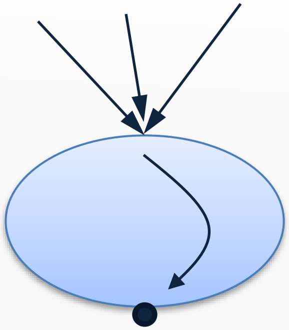
  $[\![\ x = E\ ]\!] = JOIN(v)[x \mapsto eval(JOIN(v),E)]$

- For variable declarations:

  $[\![\ \texttt{var}\ x_1, \ldots, x_n\ ]\!] = JOIN(v)[x_1 \mapsto \top, \ldots, x_n \mapsto \top]$

- For all other nodes:

  $[\![v]\!] = JOIN(v)$



where $JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$ ← combines information from predecessors

Generating constraints

```
var a,b,c;
a = 42;
b = 87;
if (input) {
  c = a + b;
} else {
  c = a - b;
}
```

$$[\![entry]\!] = \bot$$
$$[\![\texttt{var a,b,c}]\!] = [\![entry]\!][a \mapsto \top, b \mapsto \top, c \mapsto \top]$$
$$[\![\texttt{a = 42}]\!] = [\![\texttt{var a,b,c}]\!][a \mapsto +]$$
$$[\![\texttt{b = 87}]\!] = [\![\texttt{a = 42}]\!][b \mapsto +]$$
$$[\![\texttt{input}]\!] = [\![\texttt{b = 87}]\!]$$
$$[\![\texttt{c = a + b}]\!] = [\![\texttt{input}]\!][c \mapsto [\![\texttt{input}]\!](a) + [\![\texttt{input}]\!](b)]$$
$$[\![\texttt{c = a - b}]\!] = [\![\texttt{input}]\!][c \mapsto [\![\texttt{input}]\!](a) - [\![\texttt{input}]\!](b)]$$

using l.u.b. $\longrightarrow$ $$[\![exit]\!] = [\![\texttt{c = a + b}]\!] \sqcup [\![\texttt{c = a - b}]\!]$$

Constraints

- From the program being analyzed, we have constraint variables $x_1, \ldots, x_n \in L$ and a collection of constraints:

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

  $x_2 = f_2(x_1, \ldots, x_n)$

  …

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

Note that L^n is a product lattice

- These can be collected into a single function $f: L^n \to L^n$:

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

- 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 $f: L_1 \rightarrow L_2$ is *monotone* when

  $\forall x,y \in L_1: x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

- 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 (exercises)
- $x \sqsubseteq y$ can be interpreted as "x is at least as precise as y"
- When f is monotone:
  "more precise input cannot lead to less precise output"

Monotonicity for the sign analysis

Example, constraints for assignments:
$[\![x = E]\!] = JOIN(\mathrm{v})[x \mapsto eval(JOIN(\mathrm{v}),E)]$

- The $\sqcup$ operator and map updates are monotone
- Compositions preserve monotonicity

(exercises)

- Are the abstract operators monotone?
- Can be verified by a tedious inspection:
  - $\forall x,y,x' \in \mathrm{L}: x \sqsubseteq x' \Rightarrow x\ \overline{\mathrm{op}}\ y \sqsubseteq x'\ \overline{\mathrm{op}}\ y$
  - $\forall x,y,y' \in \mathrm{L}: y \sqsubseteq y' \Rightarrow x\ \overline{\mathrm{op}}\ y \sqsubseteq x\ \overline{\mathrm{op}}\ y'$

Kleene's fixed-point theorem

$x \in L$ is a *fixed point* of $f: L \rightarrow L$ iff $f(x)=x$

In a complete lattice with finite height, every monotone function f has a *unique least fixed-point*:

$$lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

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: $f^k(\bot) = f^{k+1}(\bot)$
- If $x \sqsubseteq y$ then $x \sqcup y = y$ (exercise)
- So $\mathit{lfp}(f) = f^k(\bot)$

Proof of unique least

- Assume that x is another fixed-point: $x = \mathrm{f}(x)$
- Clearly, $\perp \sqsubseteq x$
- By induction and monotonicity, $\mathrm{f}^i(\perp) \sqsubseteq \mathrm{f}^i(x) = x$
- In particular, $lfp(\mathrm{f}) = \mathrm{f}^k(\perp) \sqsubseteq x$, i.e. $lfp(\mathrm{f})$ is least

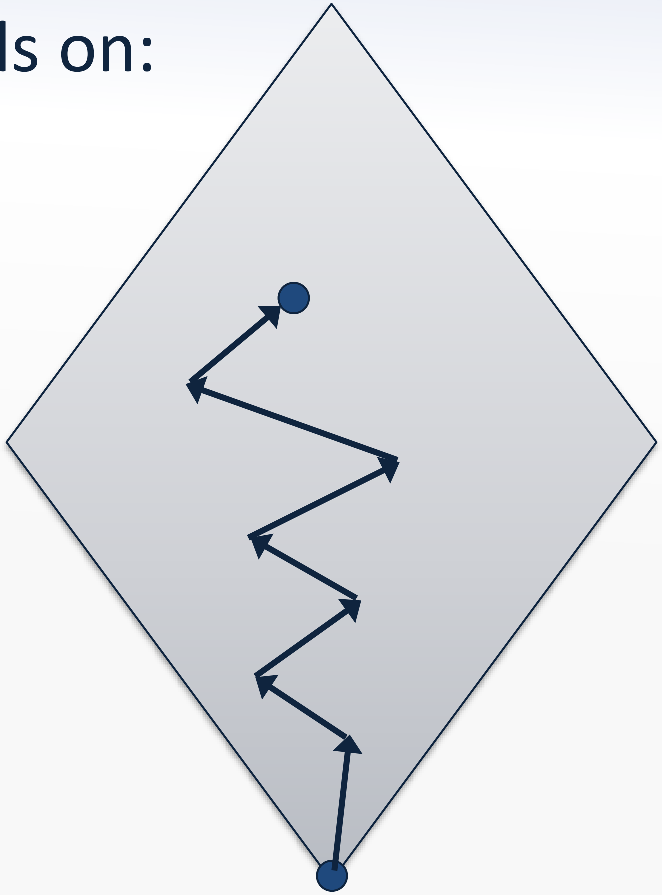
- Uniqueness then follows from anti-symmetry

Computing fixed-points

The time complexity of *lfp*(f) depends on:

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

```
x = ⊥;
do {
  t = x;
  x = f(x);
} while (x≠t);
```



Implementation: `TIP/src/tip/solvers/FixpointSolvers.scala`

Summary: lattice equations

- Let L be a complete lattice with finite height

- An *equation system* is of the form:

  $x_1 = f_1(x_1, \ldots, x_n)$
  $x_2 = f_2(x_1, \ldots, x_n)$
  …
  $x_n = f_n(x_1, \ldots, x_n)$

  where x_i are variables and each $f_i: L^n \to L$ 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 $f: L^n \to L^n$ defined by

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

- A solution is always a fixed-point (for any kind of equation)
- The least one is the most precise

Solving inequations

- An *inequation system* is of the form

$$\begin{aligned} x_1 &\sqsubseteq f_1(x_1, \ldots, x_n) \\ x_2 &\sqsubseteq f_2(x_1, \ldots, x_n) \\ &\ldots \\ x_n &\sqsubseteq f_n(x_1, \ldots, x_n) \end{aligned} \qquad \text{or} \qquad \begin{aligned} x_1 &\sqsupseteq f_1(x_1, \ldots, x_n) \\ x_2 &\sqsupseteq f_2(x_1, \ldots, x_n) \\ &\ldots \\ x_n &\sqsupseteq f_n(x_1, \ldots, x_n) \end{aligned}$$

- Can be solved by exploiting the facts that

$$x \sqsubseteq y \iff x = x \sqcap y$$

and

$$x \sqsupseteq y \iff x = x \sqcup y$$

Monotone frameworks

John B. Kam, Jeffrey D. Ullman: Monotone Data Flow Analysis Frameworks. Acta Inf. 7: 305-317 (1977)

- A CFG to be analyzed, nodes *Node* = $\{v_1, v_2, ..., v_n\}$
- A finite-height complete lattice L of possible answers
  - fixed or parametrized by the given program
- A constraint variable $[\![v]\!] \in L$ for every CFG node v
- A dataflow constraint for each syntactic construct
  - relates the value of $[\![v]\!]$ to the variables for other nodes
  - typically a node is related to its neighbors
  - the constraints must be monotone functions:

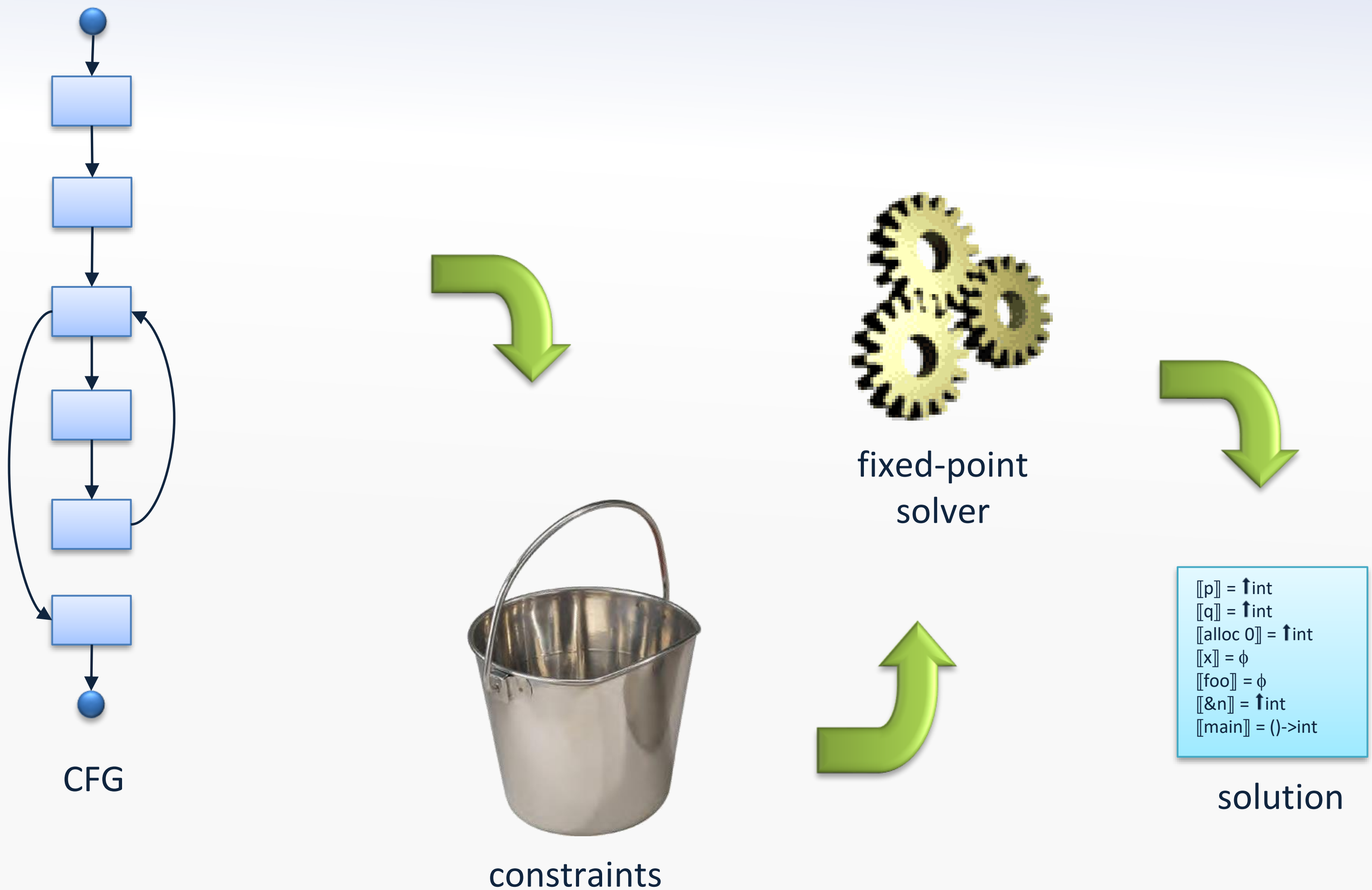
    $[\![v_i]\!] = f_i([\![v_1]\!], [\![v_2]\!], ..., [\![v_n]\!])$

Monotone frameworks

- 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:

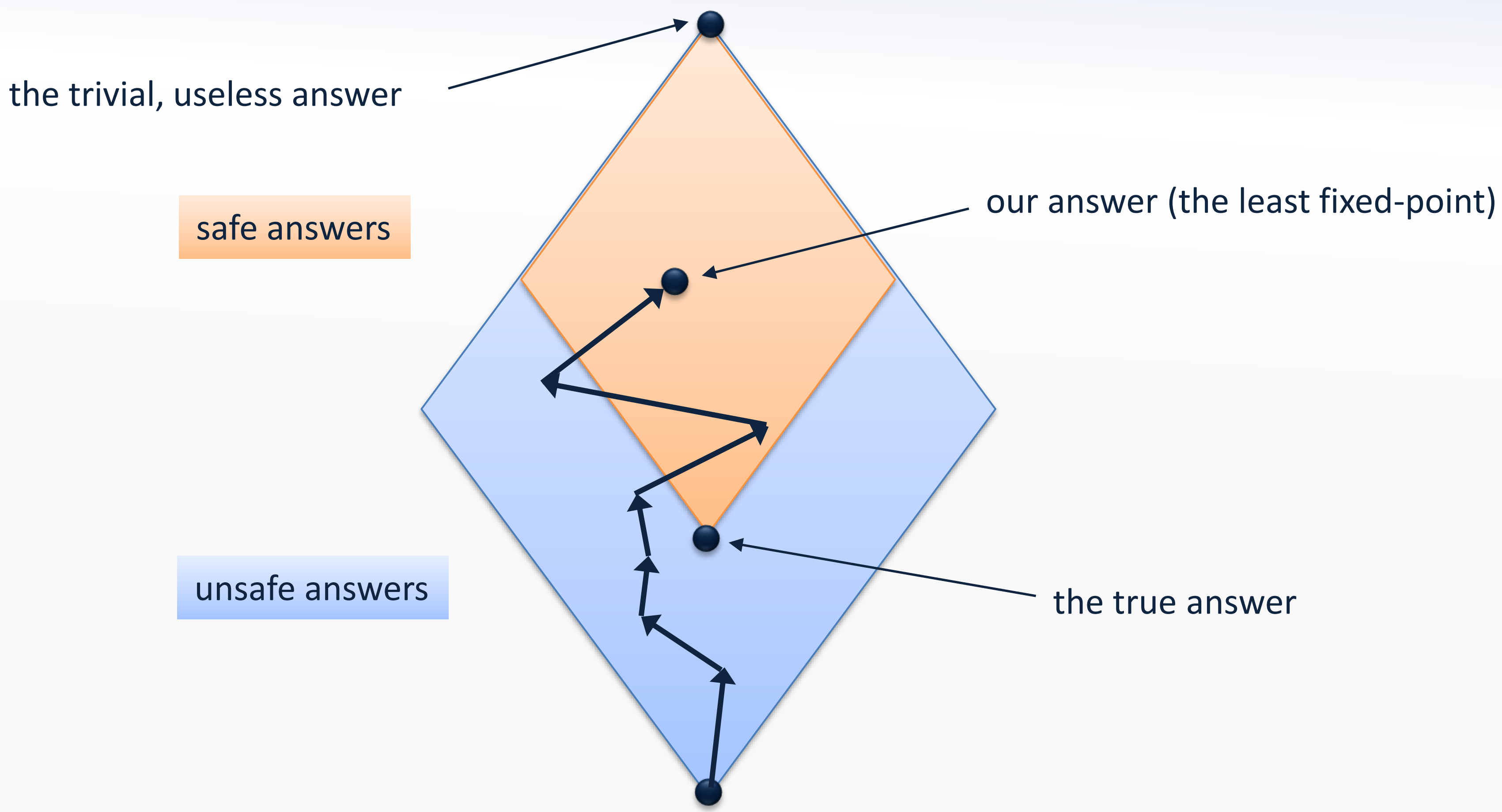
    $f(x_1,...,x_n) = (f_1(x_1,...,x_n), ..., f_n(x_1,...,x_n))$
- This solution gives an answer from L for each CFG node

Generating and solving constraints



Conceptually, we separate constraint generation from constraint solving, but in implementations, the two stages are typically interleaved

Lattice points as answers



Conservative approximation…

The naive algorithm

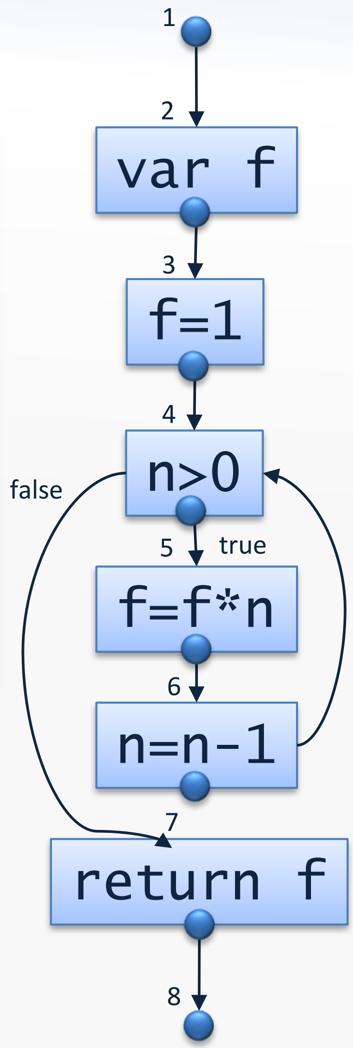
```
x = (⊥, ⊥, ..., ⊥);
do {
  t = x;
  x = f(x);
} while (x≠t);
```

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

Implementation: `SimpleFixpointSolver`

Example: sign analysis

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```



$[n \to \top, f \to \perp]$

$[n \to \top, f \to \top]$

$[n \to \top, f \to +]$

$[n \to \top, f \to \top]$

$[n \to \top, f \to \top]$

$[n \to \top, f \to \top]$

$[n \to \top, f \to \top]$

$[n \to \top, f \to \top]$

Note: some of the constraints are mutually recursive in this example

(We shall later see how to improve precision for the loop condition)

The naive algorithm

	$f^0(\perp, \perp, \dots, \perp)$	$f^1(\perp, \perp, \dots, \perp)$	…	$f^k(\perp, \perp, \dots, \perp)$
1	$\perp$	$f_1(\perp, \perp, \dots, \perp)$	…	…
2	$\perp$	$f_2(\perp, \perp, \dots, \perp)$	…	…
…	…	…	…	…
n	$\perp$	$f_n(\perp, \perp, \dots, \perp)$	…	…

Computing each new entry is done using the previous column

- Without using the entries in the current column that have already been computed!
- And many entries are likely unchanged from one column to the next!

Chaotic iteration

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

```
x₁ = ⊥; ... xₙ = ⊥;
while ((x₁,...,xₙ) ≠ f(x₁,..., xₙ)) {
      pick i nondeterministically such
        that xᵢ ≠ fᵢ(x₁, ..., xₙ)
      xᵢ = fᵢ(x₁, ..., xₙ);
}
```

We now exploit the special structure of L^n

– may require a higher number of iterations,

but less work in each iteration

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 $\forall j: \underline{x}^j \sqsubseteq x^j$
- 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:`… 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 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:

$$\mathit{dep}: \mathit{Node} \rightarrow 2^{\mathit{Node}}$$

that for v∈*Node* gives the set of nodes (i.e. constraint variables) w where v occurs on the right-hand side of the constraint for w

The worklist algorithm (2/2)

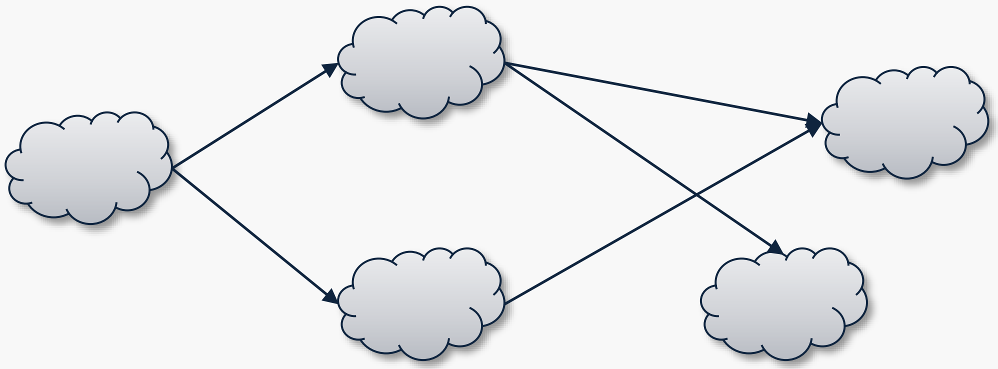
```
x₁ = ⊥; ... xₙ = ⊥;
W = {v₁, ..., vₙ};
while (W≠∅) {
  vᵢ = W.removeNext();
  y = fᵢ(x₁, ..., xₙ);
  if (y≠xᵢ) {
    for (vⱼ ∈ dep(vᵢ)) W.add(vⱼ);
    xᵢ = y;
  }
}
```

Implementation: `SimpleWorklistFixpointSolver`

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

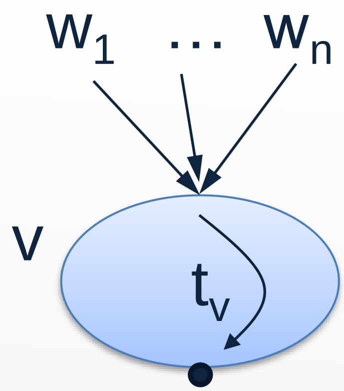


Transfer functions

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

$$[\![v]\!] = t_v(JOIN(v))$$

where t_v: *State* $\to$ *State* is called the ***transfer function*** for *v*



- Example:

$$[\![x = E]\!] = JOIN(v)[x \mapsto eval(JOIN(v),E)]$$
$$= t_v(JOIN(v))$$

where

$$t_v(s) = s[x \mapsto eval(s,E)]$$

Sign Analysis, continued...

- Another improvement of the worklist algorithm:
  - only add the entry node to the worklist initially
  - then let dataflow propagate through the program according to the constraints...

- Now, what if the constraint rule for variable declarations was:

  $[\![\texttt{var } x_1 \texttt{, ..., } x_n]\!] = JOIN(v)[x_1 \mapsto \bot, \ldots, x_n \mapsto \bot]$

  (would make sense if we treat "uninitialized" as "no value" instead of "any value")

- Problem: iteration would stop before the fixpoint!
- Solution: replace $Var \rightarrow Sign$ by $lift(Var \rightarrow Sign)$

  (allows us to distinguish between "unreachable" and "all variables are non-integers")
- This trick is also useful for context-sensitive analysis! (later...)

Implementation: `WorklistFixpointSolverWithReachability`, `MapLiftLatticeSolver`