

# Static Program Analysis

## Part 3 – lattices and fixpoints

<http://cs.au.dk/~amoeller/spa/>

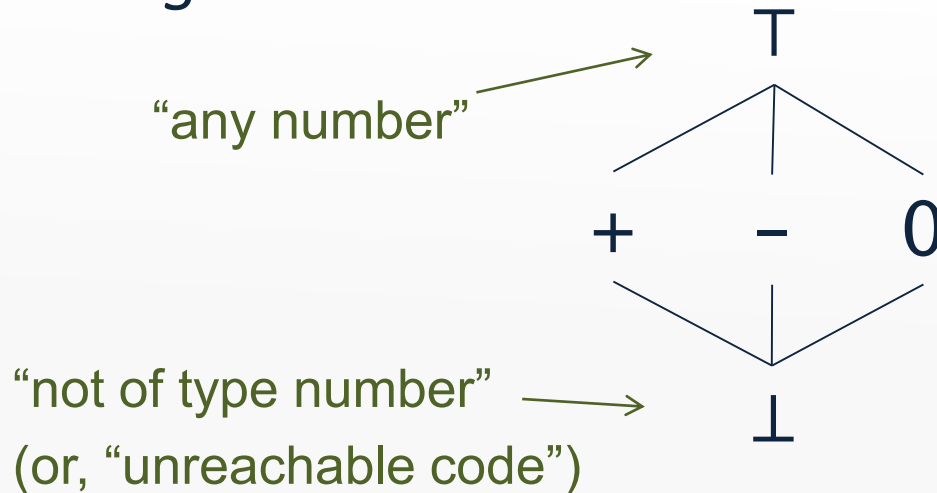
Anders Møller & Michael I. Schwartzbach  
Computer Science, Aarhus University

# 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:



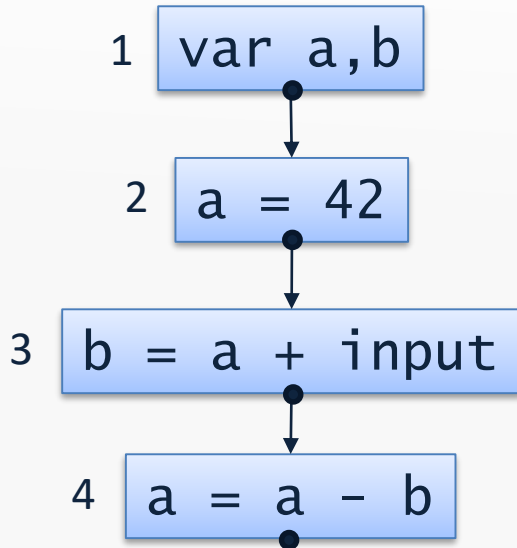
The terminology will be defined later – this is just an appetizer...

- States are modeled by the map lattice  $Vars \rightarrow Sign$  where  $Vars$  is the set of variables in the program

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

# Generating constraints

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



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

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

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

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

# Sign analysis constraints

- The variable  $\llbracket v \rrbracket$  denotes a map that gives the sign value for all variables at the program point *after* node  $v$

- For variable declarations:

$$\llbracket \text{var } x_1, \dots, x_n \rrbracket = \text{JOIN}(v)[x_1 \mapsto T, \dots, x_n \mapsto T]$$

- For assignments:

$$\llbracket x = E \rrbracket = \text{JOIN}(v)[x \mapsto \text{eval}(\text{JOIN}(v), E)]$$

- For all other nodes:

$$\llbracket v \rrbracket = \text{JOIN}(v)$$

where  $\text{JOIN}(v) = \bigsqcup_{w \in \text{pred}(v)} \llbracket w \rrbracket$

← combines information from predecessors (explained later...)

# Evaluating signs

- The *eval* function is an *abstract evaluation*:
  - $eval(\sigma, x) = \sigma(x)$
  - $eval(\sigma, intconst) = sign(intconst)$
  - $eval(\sigma, E_1 \text{ op } E_2) = \overline{op}(eval(\sigma, E_1), eval(\sigma, E_2))$
- $\sigma: Vars \rightarrow Sign$  is an abstract state
- The *sign* function gives the sign of an integer
- The  $\overline{op}$  function is an abstract evaluation of the given operator

# Abstract operators

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

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

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

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

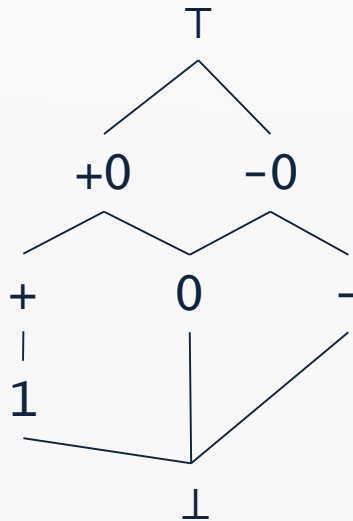
| > | ⊥ | 0 | - | + | ⊤ |
|---|---|---|---|---|---|
| ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| 0 | ⊥ | 0 | + | 0 | ⊤ |
| - | ⊥ | 0 | ⊤ | 0 | ⊤ |
| + | ⊥ | + | + | ⊤ | ⊤ |
| ⊤ | ⊥ | ⊤ | ⊤ | ⊤ | ⊤ |

| == | ⊥ | 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 T
  - $+ / +$  is analyzed as T, since e.g.  $\frac{1}{2}$  is rounded down
- Use a richer lattice for better precision:

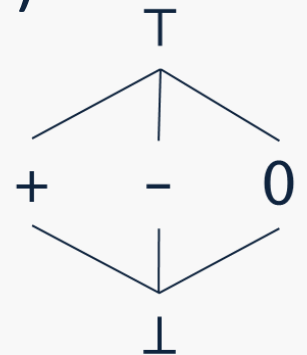


- Abstract operators are now  $8 \times 8$  tables



# 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:  $\forall x, y, z \in S: x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - anti-symmetry:  $\forall x, y \in S: x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
- Can be illustrated by a Hasse diagram (if finite)



# Upper and lower bounds

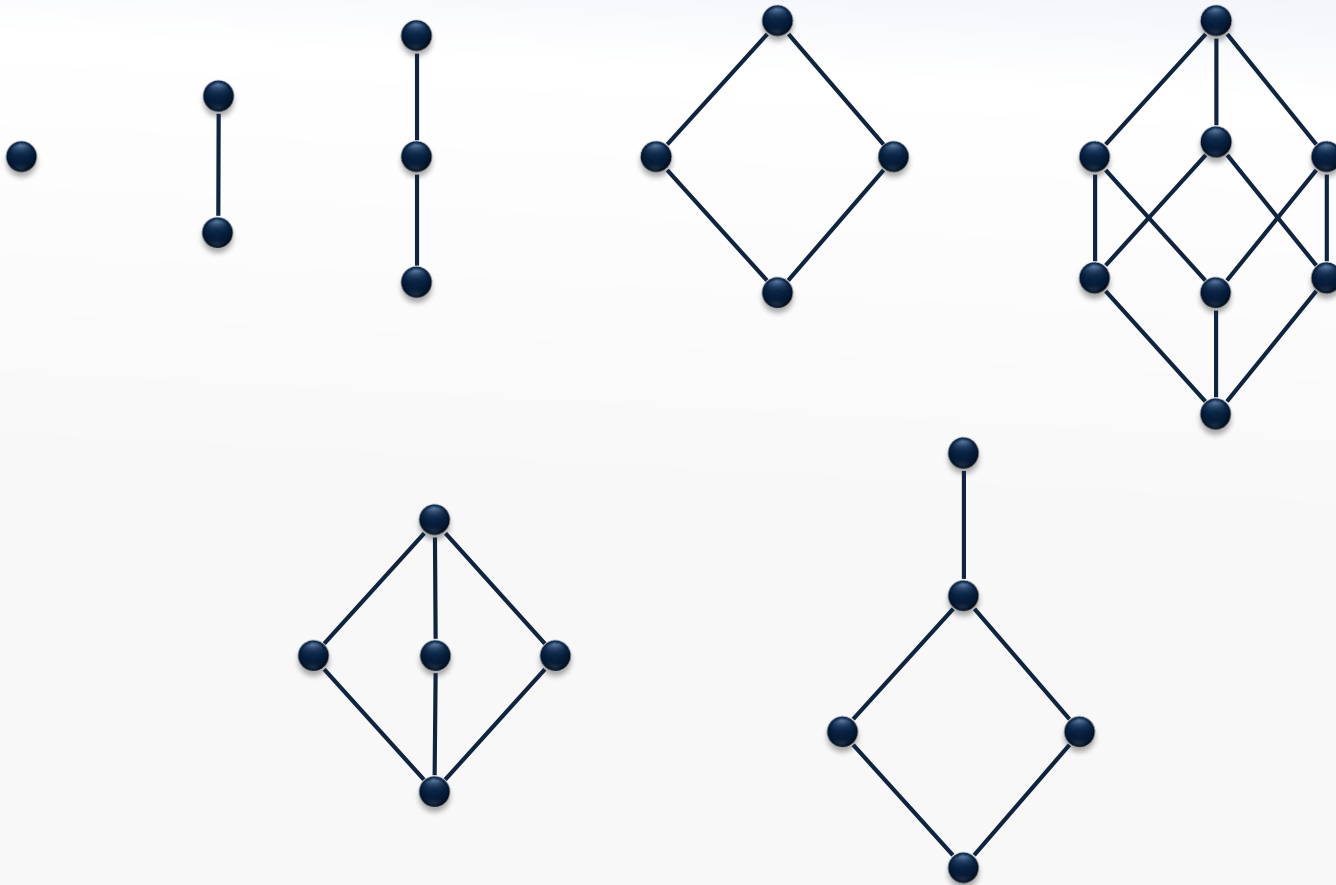
- Let  $X \subseteq S$  be a subset
- We say that  $y \in S$  is an *upper* bound ( $X \sqsubseteq y$ ) when
$$\forall x \in X: x \sqsubseteq y$$
- We say that  $y \in S$  is a *lower* bound ( $y \sqsubseteq X$ ) when
$$\forall x \in X: y \sqsubseteq x$$
- A *least* upper bound  $\sqcup X$  is defined by
$$X \sqsubseteq \sqcup X \wedge \forall y \in S: X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$
- A *greatest* lower bound  $\sqcap X$  is defined by
$$\sqcap X \sqsubseteq X \wedge \forall y \in S: y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

# Lattices

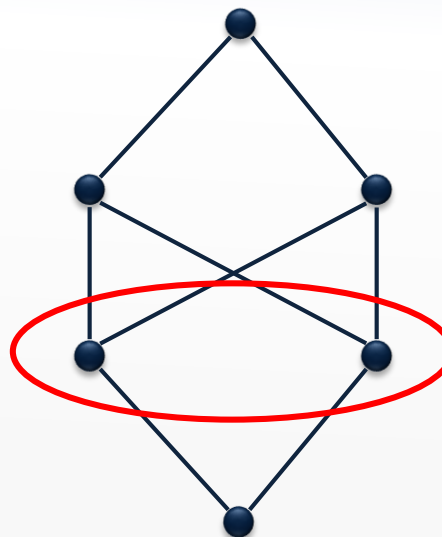
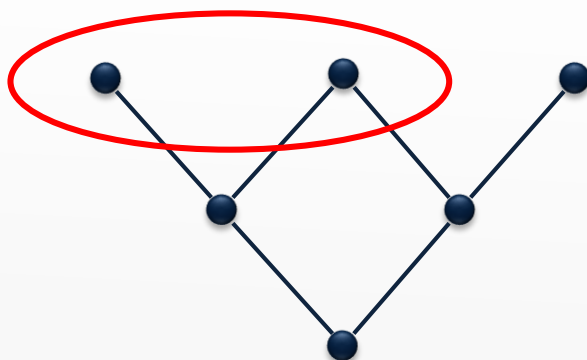
- A (complete) *lattice* is a partial order where  $\sqcup X$  and  $\sqcap X$  exist for all  $X \subseteq S$
- A lattice must have
  - a unique largest element,  $\top = \sqcup S$  (exercise)
  - a unique smallest element,  $\perp = \sqcap S$
- If  $S$  is a finite set, then it defines a lattice iff
  - $\top$  and  $\perp$  exist in  $S$
  - $x \sqcup y$  and  $x \sqcap y$  exist for all  $x, y \in S$  ( $x \sqcup y$  is notation for  $\sqcup\{x, y\}$ )

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

# These partial orders are lattices

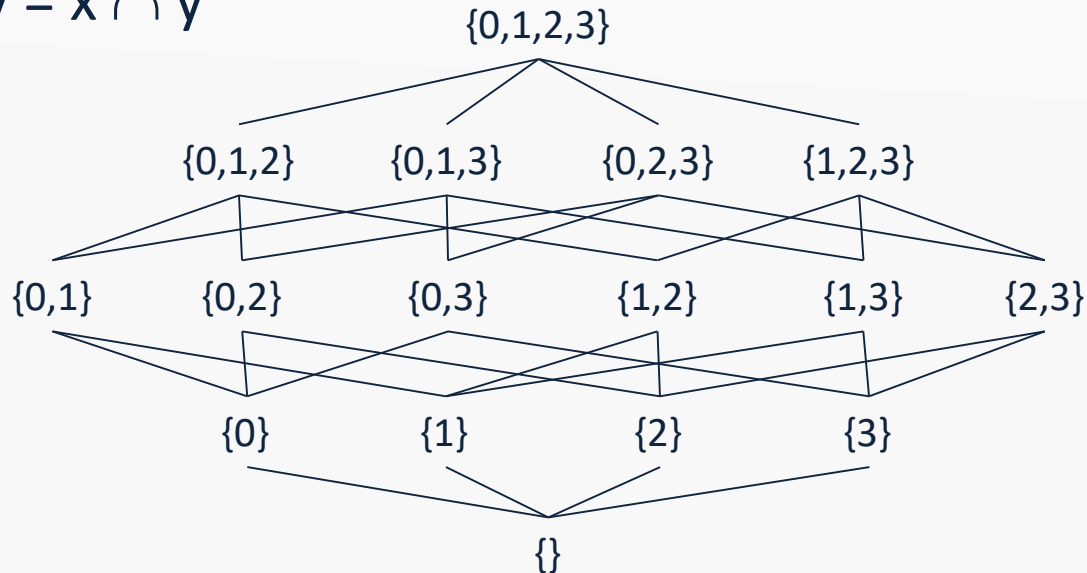


# These partial orders are *not* lattices



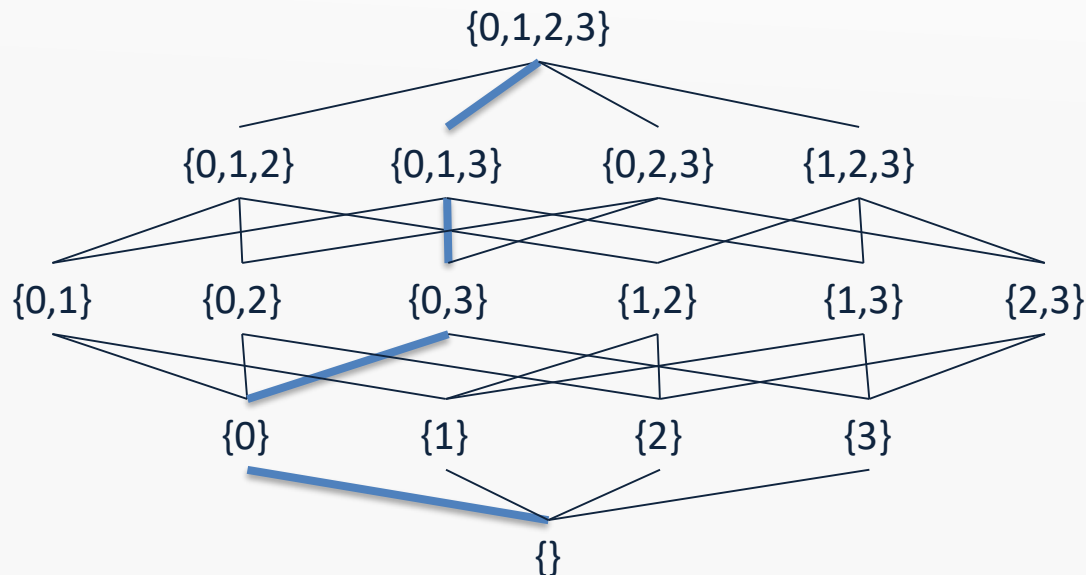
# The powerset lattice

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



# Lattice height

- The *height* of a lattice is the length of the longest path from  $\perp$  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:

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

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
- $height(A \rightarrow L) = |A| \cdot height(L)$



# Product lattice

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

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

where  $\sqsubseteq$  is defined pointwise

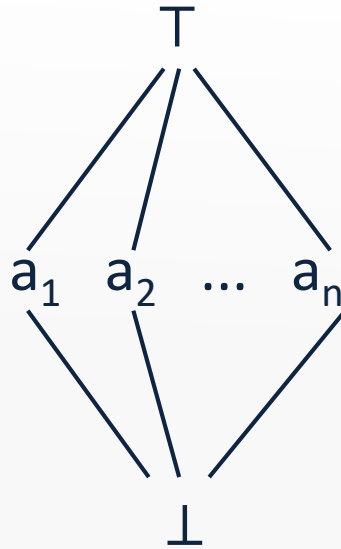
- Note that  $\sqcup$  and  $\sqcap$  can be computed pointwise
- $height(L_1 \times L_2 \times \dots \times L_n) = height(L_1) + \dots + height(L_n)$

Example:

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

# Flat lattice

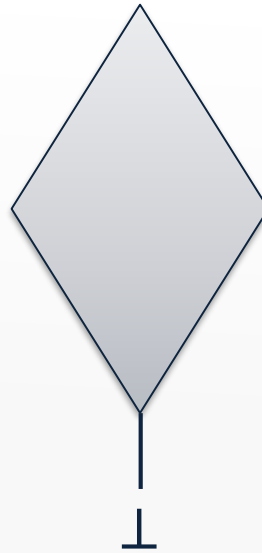
- If  $A$  is a set, then  $flat(A)$  is a lattice:



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

# Lift lattice

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



- $\text{height}(\text{lift}(L)) = \text{height}(L) + 1$

# Sign analysis constraints, revisited

- The variable  $\llbracket v \rrbracket$  denotes a map that gives the sign value for all variables at the program point *after* node  $v$

- $\llbracket v \rrbracket \in States$  where  $States = Vars \rightarrow Sign$

- For variable declarations:

$$\llbracket \text{var } x_1, \dots, x_n \rrbracket = JOIN(v)[x_1 \mapsto T, \dots, x_n \mapsto T]$$

- For assignments:

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

- For all other nodes:

$$\llbracket v \rrbracket = JOIN(v)$$

where  $JOIN(v) = \sqcup_{w \in pred(v)} \llbracket w \rrbracket$

← combines information from predecessors

```

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

```

# Generating constraints



$$\llbracket \text{entry} \rrbracket = \perp$$

$$\llbracket \text{var } a,b,c \rrbracket = \llbracket \text{entry} \rrbracket [a \mapsto T, b \mapsto T, c \mapsto T]$$

$$\llbracket a = 42 \rrbracket = \llbracket \text{var } a,b,c \rrbracket [a \mapsto +]$$

$$\llbracket b = 87 \rrbracket = \llbracket a = 42 \rrbracket [b \mapsto +]$$

$$\llbracket \text{input} \rrbracket = \llbracket b = 87 \rrbracket$$

$$\llbracket c = a + b \rrbracket = \llbracket \text{input} \rrbracket [c \mapsto \llbracket \text{input} \rrbracket (a) + \llbracket \text{input} \rrbracket (b)]$$

$$\llbracket c = a - b \rrbracket = \llbracket \text{input} \rrbracket [c \mapsto \llbracket \text{input} \rrbracket (a) - \llbracket \text{input} \rrbracket (b)]$$

using l.u.b.  $\longrightarrow$

$$\llbracket \text{exit} \rrbracket = \llbracket c = a + b \rrbracket \sqcup \llbracket c = a - b \rrbracket$$

# Constraints

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

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

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

...

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

Note that  $L^n$  is  
a product lattice



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

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

- How do we find the least (i.e. most precise) value of  $x_1, \dots, x_n$  such that  $x_1, \dots, x_n = f(x_1, \dots, x_n)$  (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 (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”

A *least upper bound*  $\sqcup X$  is defined by

$$x \sqsubseteq \sqcup X \wedge \forall y \in S: x \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

A *greatest lower bound*  $\sqcap X$  is defined by

$$\sqcap X \sqsubseteq x \wedge \forall y \in S: y \sqsubseteq x \Rightarrow y \sqsubseteq \sqcap X$$

$x \sqcup y$  and  $x \sqcap y$  exist for all  $x, y \in S$  ( $x \sqcup y$  is notation for  $\sqcup\{x, y\}$ )

# Monotonicity for the sign analysis

Example, constraints for assignments:  
 $\llbracket x = E \rrbracket = JOIN(v)[x \mapsto eval(JOIN(v), E)]$

- The  $\sqcup$  operator and map updates are monotone
- Compositions preserve monotonicity (exercises)
- Are the abstract operators monotone?
- Can be verified by a tedious inspection:
  - $\forall x, y, x' \in L: x \sqsubseteq x' \Rightarrow x \overline{op} y \sqsubseteq x' \overline{op} y$
  - $\forall x, y, y' \in L: y \sqsubseteq y' \Rightarrow x \overline{op} y \sqsubseteq x \overline{op} y'$



# Kleene's fixed-point theorem

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

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

$$\text{fix}(f) = \bigsqcup_{i \geq 0} f^i(\perp)$$

克莱尼不动点定理：序理论的不动点定理

$L$ ：完全格  $f$ 是单调函数

$f$ 的最小不动点是 $f$ 的升 Kleene链的最小上界，这个链是在 $L$ 的底元素上迭代 $f$ 而获得

# Proof of existence

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

$$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \dots \sqsubseteq f^i(\perp) \sqsubseteq \dots$$

is an increasing chain

- $L$  has finite height, so for some  $k$ :  $f^k(\perp) = f^{k+1}(\perp)$
- If  $x \sqsubseteq y$  then  $x \sqcup y = y$  (exercise)
- So  $\text{fix}(f) = f^k(\perp)$

# Proof of unique least

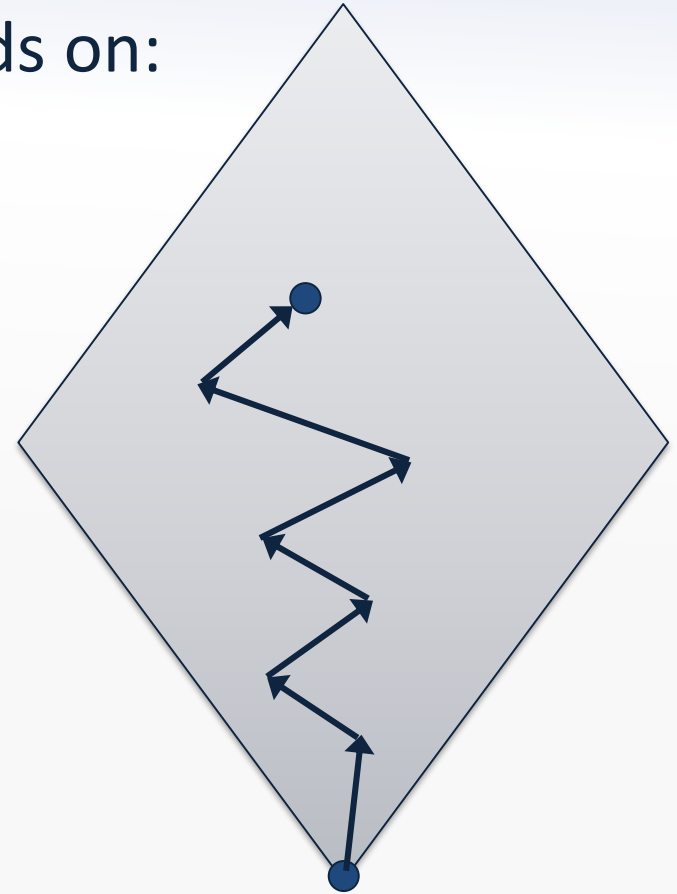
- Assume that  $x$  is another fixed-point:  $x = f(x)$
- Clearly,  $\perp \sqsubseteq x$
- By induction,  $f^i(\perp) \sqsubseteq f^i(x) = x$
- In particular,  $fix(f) = f^k(\perp) \sqsubseteq x$ , i.e.  $fix(f)$  is least
- Uniqueness then follows from anti-symmetry

# Computing fixed-points

The time complexity of  $fix(f)$  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 lattice with finite height

- A *equation system* is of the form:

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

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

...

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

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  $f: L^n \rightarrow L^n$  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

- A *inequation system* is of the form

$$x_1 \sqsubseteq f_1(x_1, \dots, x_n)$$

$$x_2 \sqsubseteq f_2(x_1, \dots, x_n)$$

...

$$x_n \sqsubseteq f_n(x_1, \dots, x_n)$$

or

$$x_1 \sqsupseteq f_1(x_1, \dots, x_n)$$

$$x_2 \sqsupseteq f_2(x_1, \dots, x_n)$$

...

$$x_n \sqsupseteq f_n(x_1, \dots, x_n)$$

- Can be solved by exploiting the facts that

$$x \sqsubseteq y \Leftrightarrow x = x \sqcap y$$

and

$$x \sqsupseteq y \Leftrightarrow x = x \sqcup y$$

# Monotone frameworks

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

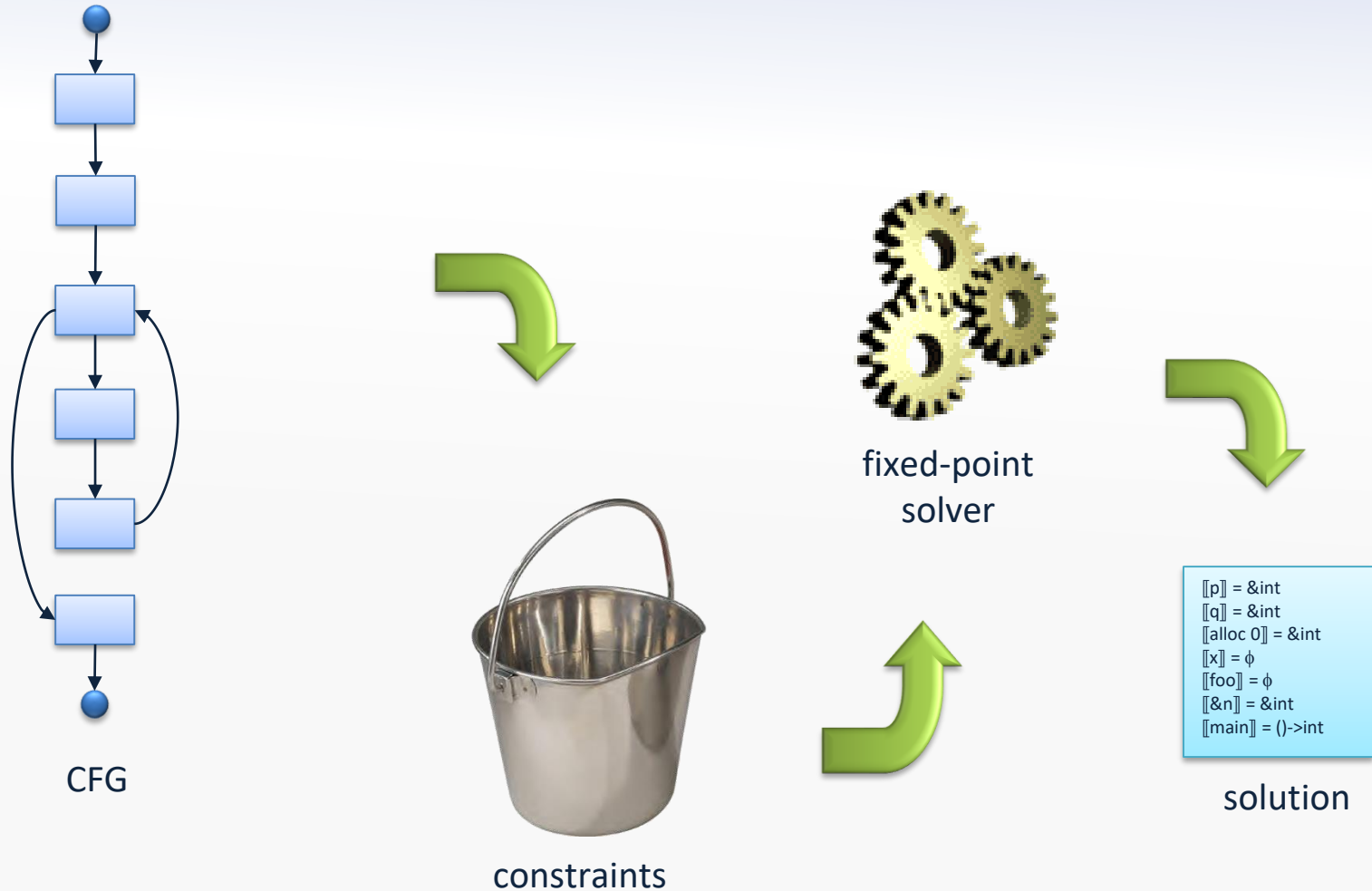
- A CFG to be analyzed, nodes  $\text{Nodes} = \{v_1, v_2, \dots, v_n\}$
- A finite-height lattice  $L$  of possible answers
  - fixed or parametrized by the given program
- A constraint variable  $\llbracket v \rrbracket \in L$  for every CFG node  $v$
- A dataflow constraint for each syntactic construct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



# Monotone frameworks

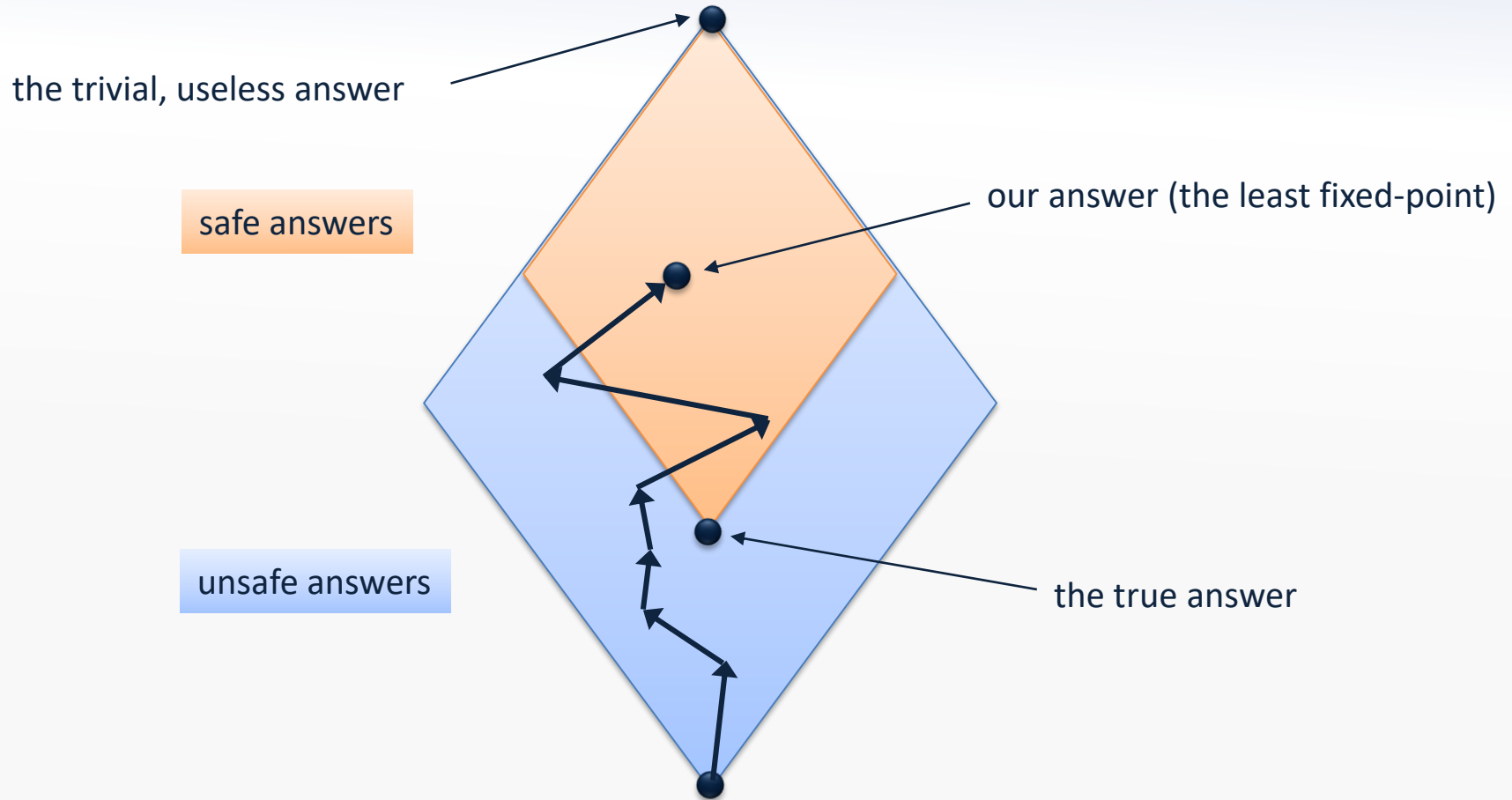
- Extract all constraints for the CFG
- Solve constraints using the fixed-point algorithm:
  - we work in the lattice  $L^n$  where  $L$  is a lattice describing abstract states
  - computing the least fixed-point of the combined function:  
$$f(x_1, \dots, x_n) = (f_1(x_1, \dots, x_n), \dots, f_n(x_1,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

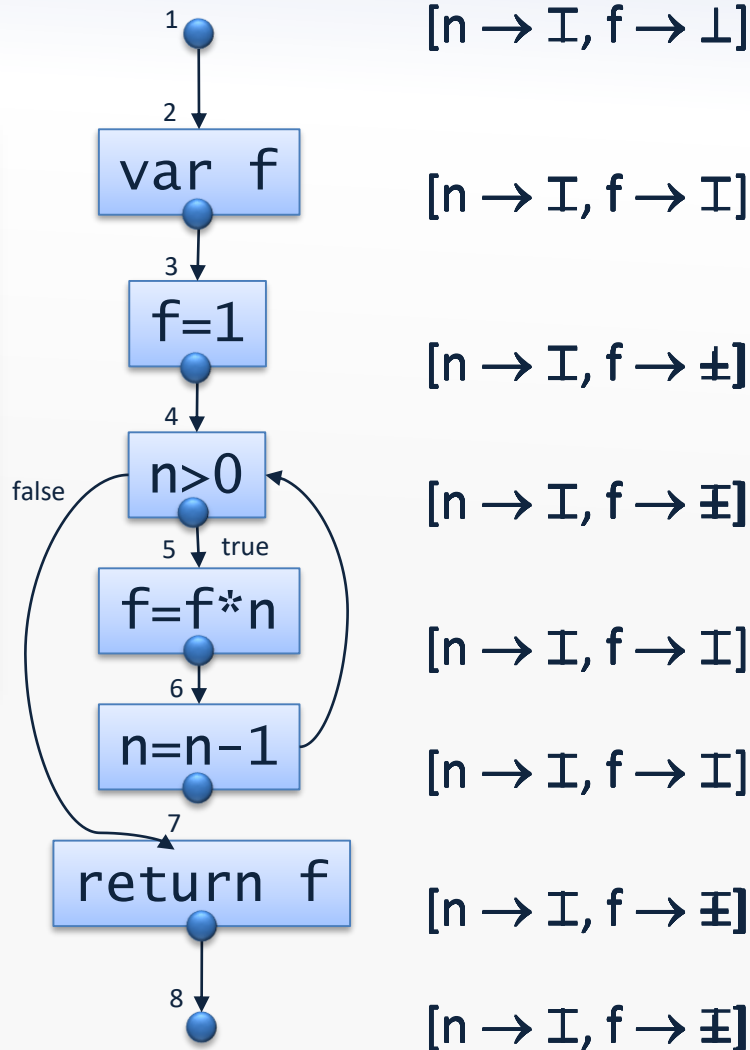
```
x = (⊥, ⊥, ..., ⊥);  
do {  
    t = x;  
    x = f(x);  
} while (x≠t);
```

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

Implementation: `SimpleFixedPointSolver`

# Example: sign analysis

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



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   | $\perp$                           | $f_1^1(\perp, \perp, \dots, \perp)$ | ... | $f_1^k(\perp, \perp, \dots, \perp)$ |
| 2   | $\perp$                           | $f_2^1(\perp, \perp, \dots, \perp)$ | ... | $f_2^k(\perp, \perp, \dots, \perp)$ |
| ... | ...                               | ...                                 | ... | ...                                 |
| $n$ | $\perp$                           | $f_n^1(\perp, \perp, \dots, \perp)$ | ... | $f_n^k(\perp, \perp, \dots, \perp)$ |

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!

怎么减少对不变的状态  
转换的重复计算

# Chaotic iteration

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

```
x1 = ⊥; ... xn = ⊥;  
while ((x1, ..., xn) ≠ f(x1, ..., xn)) {  
    pick i nondeterministically such 只对会改变状态的状态转换进行计算  
        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  $x=(x_1, \dots, x_n)$  in the  $j$ 'th iteration of the naive algorithm
- Let  $\underline{x}^j$  be the value of  $x=(x_1, \dots, x_n)$  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  $\exists \dot{i} : \dots$  in chaotic iteration is not practical
- Idea: predict  $\dot{i}$  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:

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

that for  $v \in \text{Nodes}$  gives the variables  $w$  where  $v$  occurs on the right-hand side of the constraint for  $w$

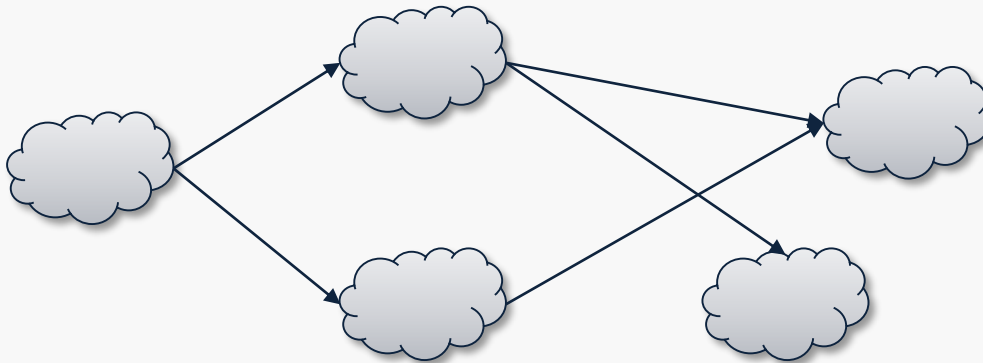
# The worklist algorithm (2/2)

```
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

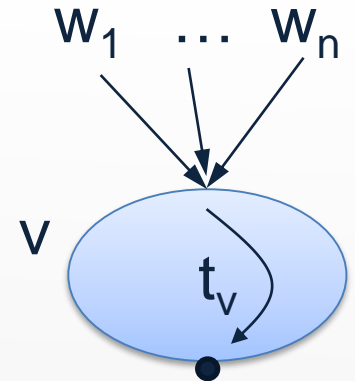


# Transfer functions

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

$$\llbracket v \rrbracket = t_v(\text{JOIN}(v))$$

where  $t_v: \text{States} \rightarrow \text{States}$  is called the **transfer function** for  $v$



- Example:

$$\begin{aligned} \llbracket x = E \rrbracket &= \text{JOIN}(v)[x \mapsto \text{eval}(\text{JOIN}(v), E)] \\ &= t_v(\text{JOIN}(v)) \end{aligned}$$

where

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

# 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:  
$$\llbracket \text{var } x_1, \dots, x_n \rrbracket = \text{JOIN}(v)[x_1 \mapsto \perp, \dots, x_n \mapsto \perp]$$

(would make sense if we treat “uninitialized” as “no value” instead of “any value”)
- Problem: iteration would stop before the fixpoint!
- Solution: replace  $\text{Vars} \rightarrow \text{Sign}$  by  $\text{lift}(\text{Vars} \rightarrow \text{Sign})$   
(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`

# Inheritance and Software Reuse

- Virtual Inheritance

- C++ 为解决Diamond 继承问题而提出的解决方案
- [Bjarne Stroustrup](#) 的 [Multiple Inheritance for C++](#)

- Mixin

- 类中的方法可以被其他类使用而不必是其他类的父类
- [Gilad Bracha](#) 等 *Mixin-based Inheritance* [[OOPSLA1990](#)]

- Traits

- 一组方法可以用于扩展一个类的功能
- 源于Self 语言, *Traits: An approach to multiple-inheritance subclassing* [[SIGOA 1982](#)]
- [Oscar Nierstrasz](#) 等的 *Traits: Composable Units of Behaviour* [[ECOOP 2003](#)]

- Prototypal Inheritance

- 通过原型来对现有的对象进行复用
- 源于Self 语言
- [Classifying Prototype-based Programming Languages](#) 和 [Prototypal Inheritance in JavaScript](#)

# Generic Programming

- STL(Standard Template Library)

- [Alexander Stepanov](#) 是 C++ STL的主要设计和实现者
- C++ STL于1994年向 ANSI/ISO C++ 标准委员会建议, 后经少量修改和扩展于1997年成为官方的 C++ 标准的一部分

- Concepts

- 是对一个类型中所支持的操作的描述, 包括语法和语义
- 最早于1998年在 STL 中使用 [Concepts: The Future of Generic Programming](#) , 该术语出自 [Alexander Stepanov](#)
- [a bit of background for concepts and C++17—Bjarne Stroustrup](#)



# FixpointSolvers

- [LatticeSolver](#)
  - [ lattice | [analyze\(\)](#) ]
- ▲ [SimpleFixpointSolver](#) : LatticeSolver
  - [ | [analyze\(\)](#), [fun\(\)](#) ]
- [MapLatticeSolver](#)[N] : LatticeSolver with [Dependencies](#)[N]
  - [ | [transfer\(\)](#), [funsub\(\)](#), [join\(\)](#) ]
- [SimpleMapLatticeFixpointSolver](#)[N]: SimpleFixpointSolver with MapLatticeSolver[N]
  - [ domain | [fun\(\)](#) ]
- [MapLiftLatticeSolver](#)[N]: MapLatticeSolver[N] with Dependencies[N]
- [Worklist](#)[N]     N: type of the elements in the worklist
  - [ | [process\(\)](#), [add\(\)](#), [run\(\)](#) ]
- [ListSetWorklist](#)[N] : Worklist[N]
  - [ worklist | [add\(\)](#), [run\(\)](#) ]
- [WorklistFixpointSolver](#)[N] : MapLatticeSolver[N] with ListSetWorklist[N] with Dependencies[N]
  - [ x | [process\(\)](#) ]
- ▲ [SimpleWorklistFixpointSolver](#)[N] : WorklistFixpointSolver[N]
- [WorklistFixpointSolverWithReachability](#)[N] extends WorklistFixpointSolver[N] with MapLiftLatticeSolver[N]

**Exercise 4.2:** Prove that if  $x \sqcup y$  exists then  $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$ , and conversely, if  $x \sqcap y$  exists then  $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$ .

**Exercise 4.6:** Prove that  $\bigsqcup S$  and  $\bigsqcap S$  are the unique largest element and the unique smallest element, respectively, in  $S$ . In other words, we have  $\top = \bigsqcup S$  and  $\perp = \bigsqcap S$ .

**Exercise 4.16:** A function  $f: L \rightarrow L$  where  $L$  is a lattice is *extensive* when  $\forall x \in L: x \sqsubseteq f(x)$ . Assume  $L$  is the powerset lattice  $2^{\{0,1,2,3,4\}}$ . Give examples of different functions  $L \rightarrow L$  that are, respectively,

- (a) extensive and monotone,
- (b) extensive but not monotone,
- (c) not extensive but monotone, and
- (d) not extensive and not monotone.

**Exercise 4.18:** A function  $f: L_1 \rightarrow L_2$  where  $L_1$  and  $L_2$  are lattices is *distributive* when  $\forall x, y \in L_1: f(x) \sqcup f(y) = f(x \sqcup y)$ .

- (a) Show that every distributive function is also monotone.
- (b) Show that not every monotone function is also distributive.

**Exercise 4.29:** Does the fixed-point theorem also hold without the assumption that  $f$  is monotone? If yes, give a proof; if no, give a counterexample.

**Exercise 4.30:** Does the fixed-point theorem also hold without the assumption that the lattice has finite height? If yes, give a proof; if no, give a counterexample.