Static Program Analysis

Anders Møller

amoeller@cs.au.dk

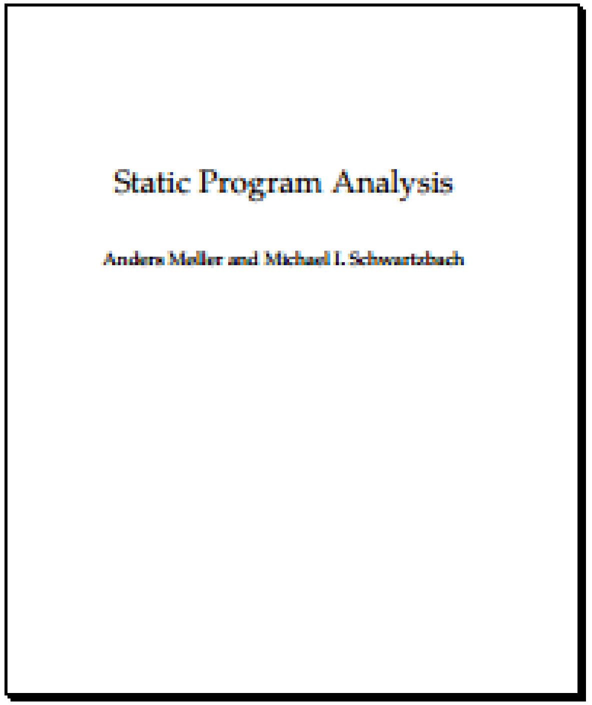
Aarhus University

About the lectures

Based on material from https://cs.au.dk/~amoeller/spa/

- lecture notes
- slides
- implementation (in Scala)
- exercises (theoretical & practical)

solid mathematical foundation $\leftrightarrow$ practical implementations



Static Program Analysis

Anders Møller and Michael I. Schwartzbach

Agenda

- Introduction to Static Program Analysis
- A simple flow-sensitive analysis: sign analysis
- Lattices, monotone functions, and fixed points
- Interprocedural analysis and context-sensitivity
- Flow-insensitive points-to analysis and control-flow analysis
- Abstract interpretation theory

Questions about programs

- Are buffer-overruns possible? Use-after-free?
- Can sensitive information leak to non-trusted users?
- Can non-trusted users affect sensitive information?
- How large can the heap become during execution?
- Does the program terminate on all inputs?
- Data races?
- SQL injections?
- XSS?
- …

Why are the answers interesting?

- Increase efficiency
  - resource usage
  - compiler optimizations
- Ensure correctness
  - verify behavior
  - catch bugs early
- Support program understanding
- Enable refactorings

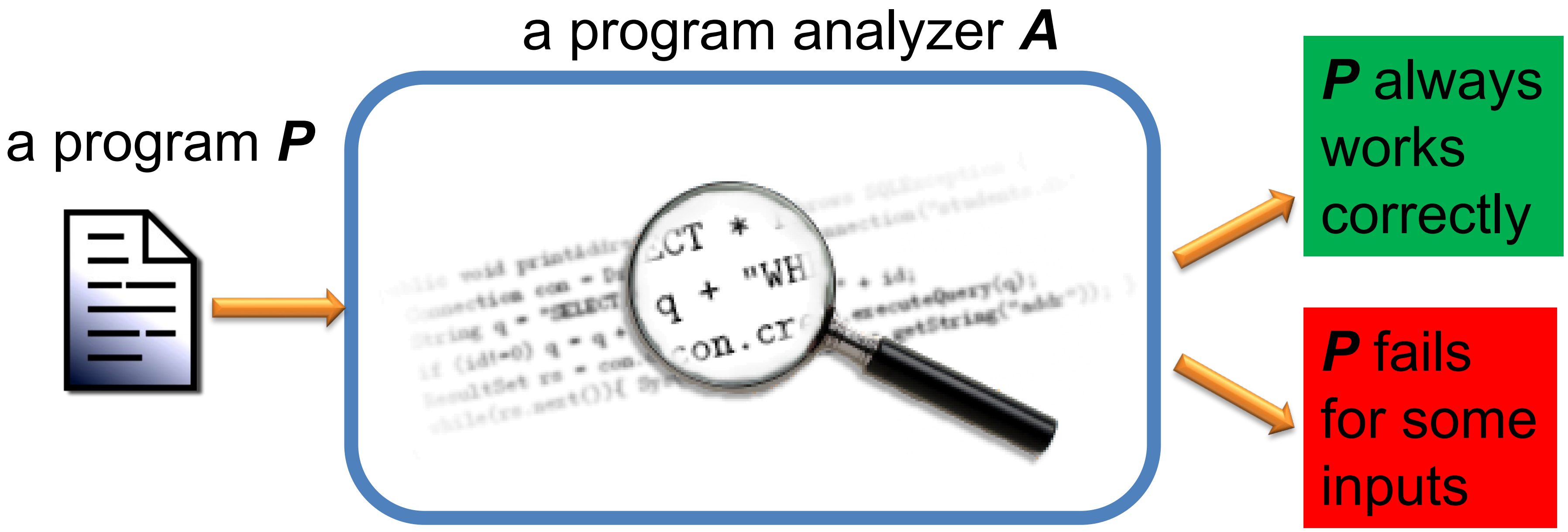
Testing?

"Program testing can be used to show the presence of bugs, but never to show their absence."

[Dijkstra, 1972]

Nevertheless, testing often takes 50% of the development cost

Programs that reason about programs


a program analyzer ***A***
a program ***P***
P always works correctly
P fails for some inputs

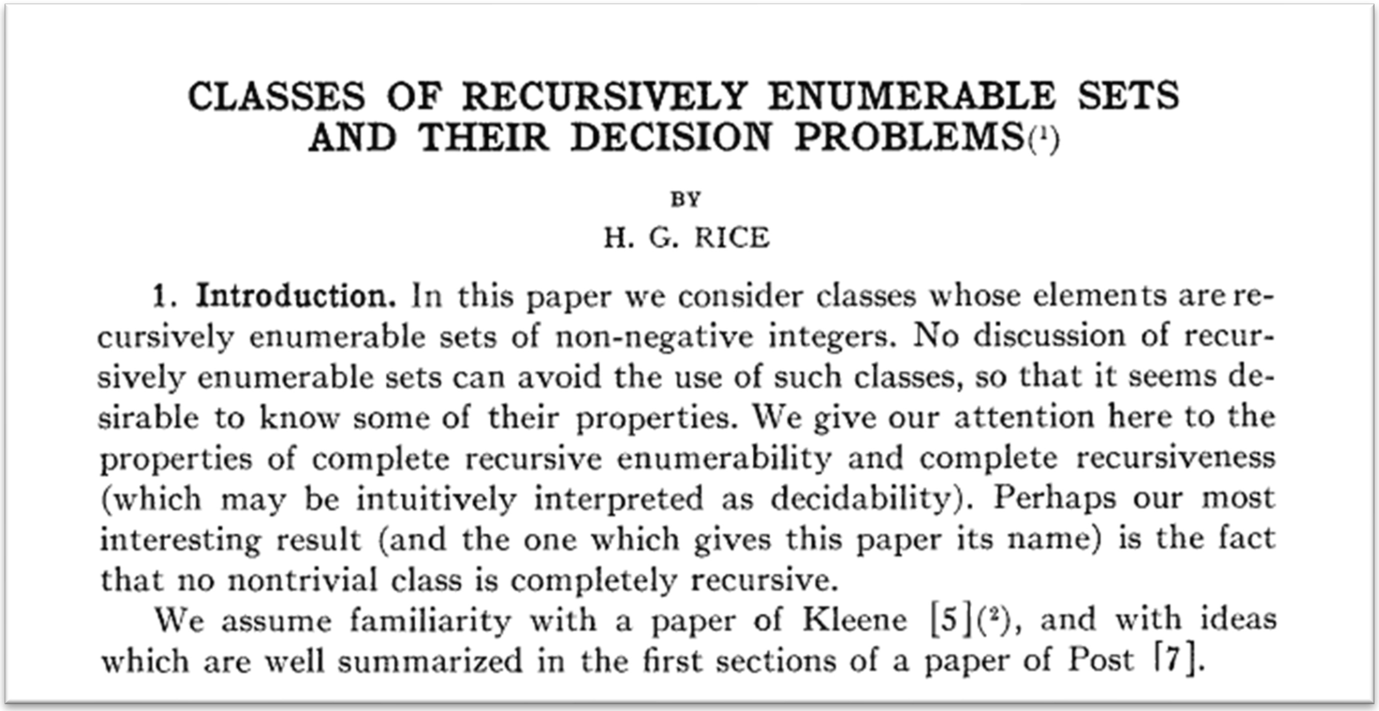
Requirements to the perfect program analyzer

SOUNDNESS (don’t miss any errors)

COMPLETENESS (don’t raise false alarms)

TERMINATION (always give an answer)

Rice's theorem, 1953



CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS(1)

BY

H. G. RICE

1. Introduction. In this paper we consider classes whose elements are recursively enumerable sets of non-negative integers. No discussion of recursively enumerable sets can avoid the use of such classes, so that it seems desirable to know some of their properties. We give our attention here to the properties of complete recursive enumerability and complete recursiveness (which may be intuitively interpreted as decidability). Perhaps our most interesting result (and the one which gives this paper its name) is the fact that no nontrivial class is completely recursive.

We assume familiarity with a paper of Kleene [5](2), and with ideas which are well summarized in the first sections of a paper of Post [7].

Any non-trivial property of the behavior of programs in a Turing-complete language is undecidable!

Approximation

- *Approximate* answers may be decidable!

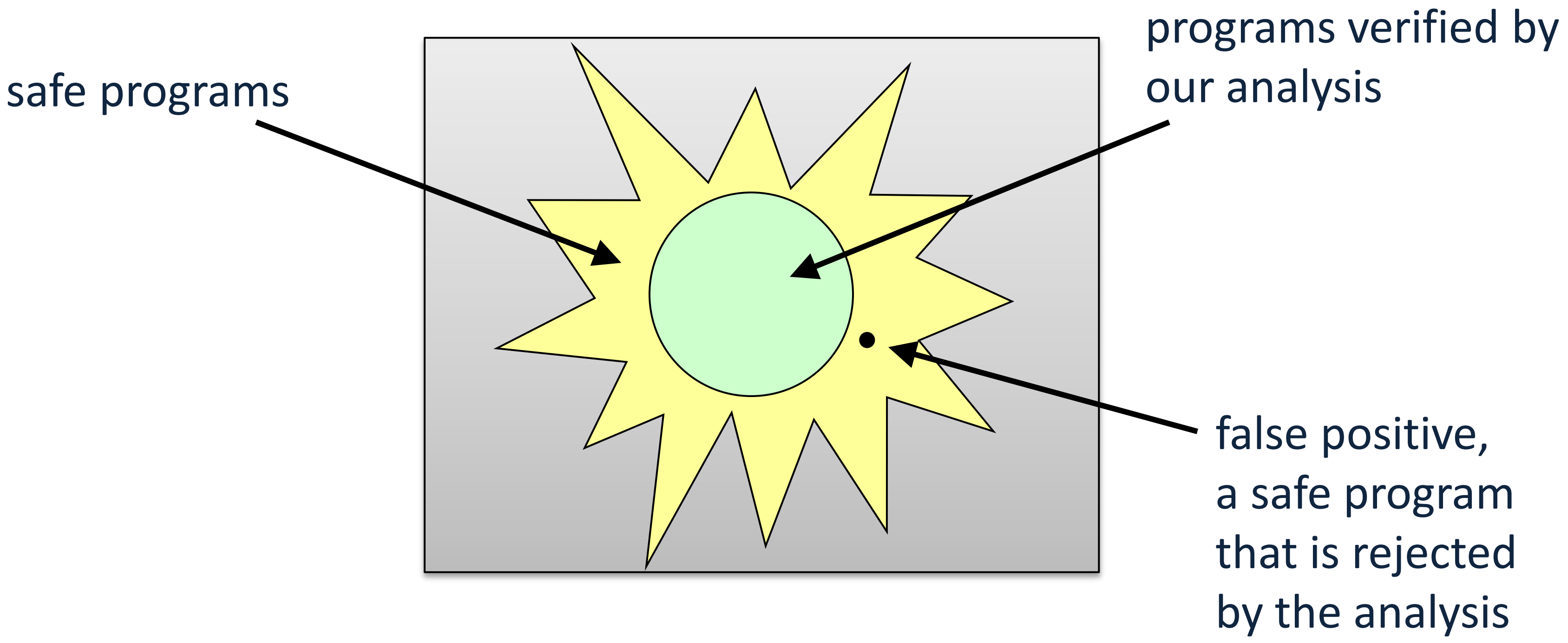
- False positives & false negatives

Approximation

- The approximation should be *conservative*
  - i.e. only err on “the safe side”

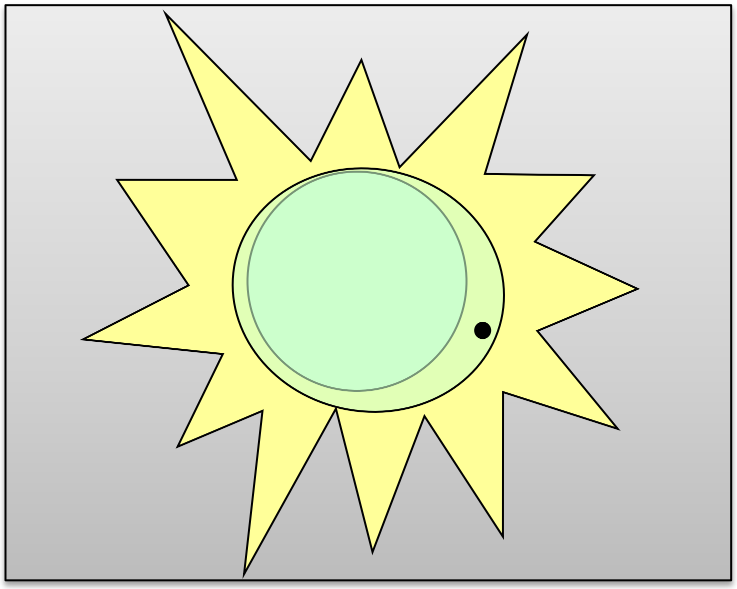
- A sound but incomplete **verification** tool:
  - “there are definitely no null pointer errors in this program!”
  - “there may be null pointer errors in this program”
- A sound but incomplete **bug detection** tool:
  - “there is definitely a null pointer error in this program!”
  - “there may be null pointer errors in this program”

Conservative approximations


safe programs
programs verified by our analysis
false positive, a safe program that is rejected by the analysis

Analysis precision

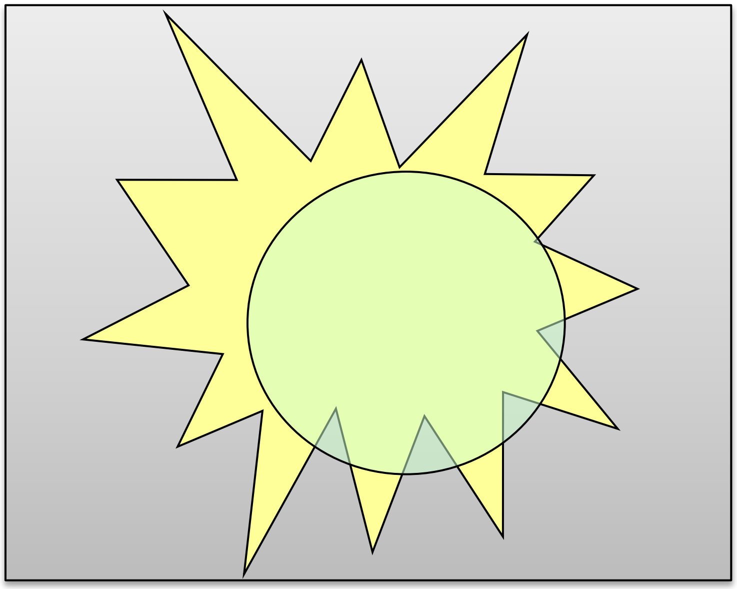
- Make the analysis a bit more clever:



- An eternal struggle
- And a great source of publications

Unsoundness

- The analysis may be unsound:



- Often a deliberate pragmatic choice

The engineering challenge

- A correct but trivial approximation algorithm may just give the useless answer every time

- The *engineering challenge* is to give the useful answer often enough to fuel the client application

- ... and to do so within reasonable time and space

- This is the hard (and fun) part of static analysis!

Does anyone use static program analysis?

For optimization:

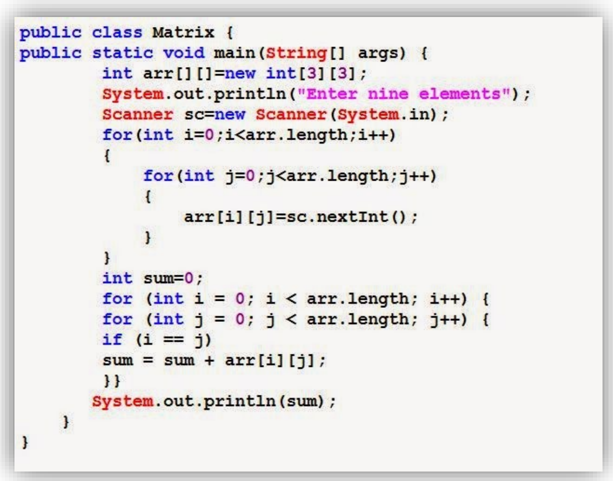
- every optimizing compiler and modern JIT

For verification or error detection:

- Astrée
- Infer
- COVERITY
- PVS-Studio
- FORTIFY
- Semmle™
- klocwork
- GrammaTech CodeSonar
- IBM Security AppScan
- CHECKMARX

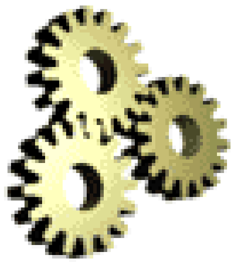
A constraint-based approach

Conceptually separates the analysis specification from algorithmic aspects and implementation details



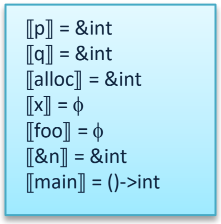
```
public class Matrix {
public static void main(String[] args) {
        int arr[][]=new int[3][3];
        System.out.println("Enter nine elements");
        Scanner sc=new Scanner(System.in);
        for(int i=0;i<arr.length;i++)
        {
            for(int j=0;j<arr.length;j++)
            {
                arr[i][j]=sc.nextInt();
            }
        }
        int sum=0;
        for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
        if (i == j)
        sum = sum + arr[i][j];
        }}
       System.out.println(sum);
    }
}
```

program to analyze



constraint solver

mathematical constraints



⟦p⟧ = &int
⟦q⟧ = &int
⟦alloc⟧ = &int
⟦x⟧ = ϕ
⟦foo⟧ = ϕ
⟦&n⟧ = &int
⟦main⟧ = ()->int

solution

Challenging features in modern programming language

- Higher-order functions
- Mutable records or objects, arrays
- Integer or floating-point computations
- Inheritance and dynamic dispatching
- Exceptions
- Reflection
- Standard libraries
- …

The TIP language

- ***T***iny ***I***mperative ***P***rogramming language
- Example language used in this course:
  - minimal C-style syntax
  - cut down as much as possible
  - enough features to make static analysis challenging and fun
- Scala implementation available

An iterative factorial function

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

A recursive factorial function

```
rec(n) {
  var f;
  if (n==0) {
    f=1;
  } else {
    f=n*rec(n-1);
  }
  return f;
}
```

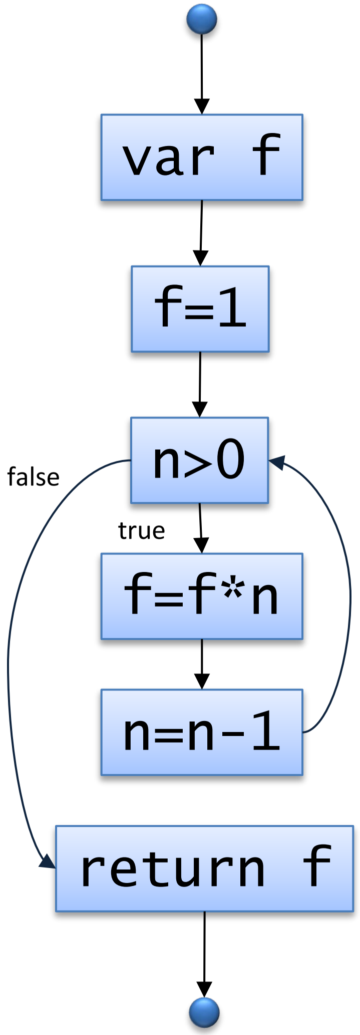
An unnecessarily complicated factorial function

```
foo(p,x) {
  var f,q;
  if (*p==0) {
    f=1;
  } else {
    q = alloc;
    *q = (*p)-1;
    f=(*p)*((x)(q,x));
  }
  return f;
}
```

```
main() {
  var n;
  n = input;
  return foo(&n,foo);
}
```

Control flow graphs for flow-sensitive analysis

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



Normalization

- Sometimes convenient to ensure that each CFG node performs only one operation

- *Normalization*: flatten nested expressions, using fresh variables

```
x = f(y+3)*5;
```

$\Rightarrow$

```
t1 = y+3;
t2 = f(t1);
x = t2*5;
```

Example: sign analysis

Sign analysis

- Determine the sign (+,−,0) of all expressions
- The *Sign* lattice:

"any number" $\rightarrow$

$$\begin{tikzcd}
& \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
+ \arrow[dr, dash] & - \arrow[d, dash] & 0 \arrow[dl, dash] \\
& \bot &
\end{tikzcd}$$

"not of type number" $\rightarrow$
(or, "unreachable code")

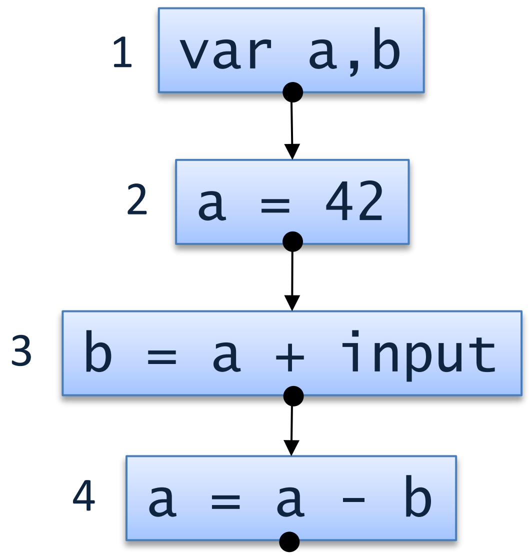
The terminology will be defined later – this is just an appetizer…

- States are modeled by the map lattice *Var* → *Sign*

  where *Var* is the set of variables in the program

Generating constraints

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

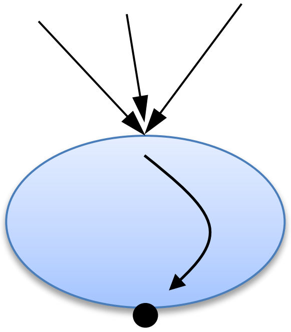
  $[\![\, x = E \,]\!] = \mathit{JOIN}(v)[x \mapsto \mathit{eval}(\mathit{JOIN}(v),E)]$

- For variable declarations:

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

- For all other nodes:

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



where $\mathit{JOIN}(v) = \bigsqcup_{w \in \mathit{pred}(v)} [\![w]\!]$ $\leftarrow$ 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	⊤
-	⊥	⊥	⊤	⊤	⊤
+	⊥	⊥	⊤	⊤	⊤
⊤	⊥	⊥	⊤	⊤	⊤

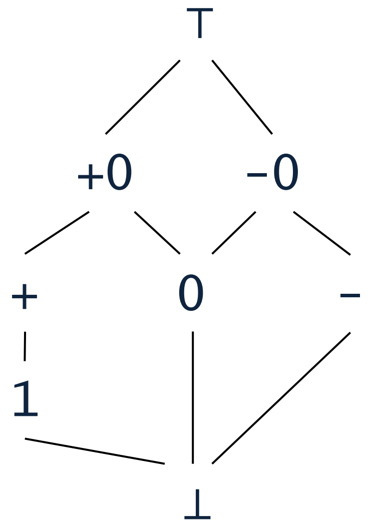
>	⊥	0	-	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	0	⊤
-	⊥	0	⊤	0	⊤
+	⊥	+	+	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊤

==	⊥	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 ⊤
  - `+/+` is analyzed as ⊤, since e.g. ½ is rounded down
- Use a richer lattice for better precision:


$$\begin{tikzcd}
 & & \top \arrow[dl, dash] \arrow[dr, dash] & & \\
 & +0 \arrow[dl, dash] \arrow[dr, dash] & & -0 \arrow[dl, dash] \arrow[dr, dash] & \\
+ \arrow[d, dash] & & 0 \arrow[dd, dash] & & - \arrow[ddll, dash] \\
1 \arrow[drr, dash] & & & & \\
 & & \perp & &
\end{tikzcd}$$

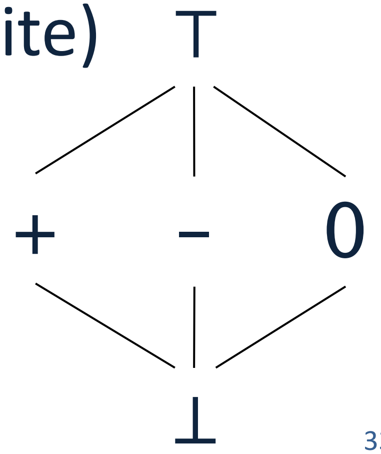
- Abstract operators are now 8×8 tables

Lattices, monotone functions, and fixed points

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 $\bigsqcup X$ 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

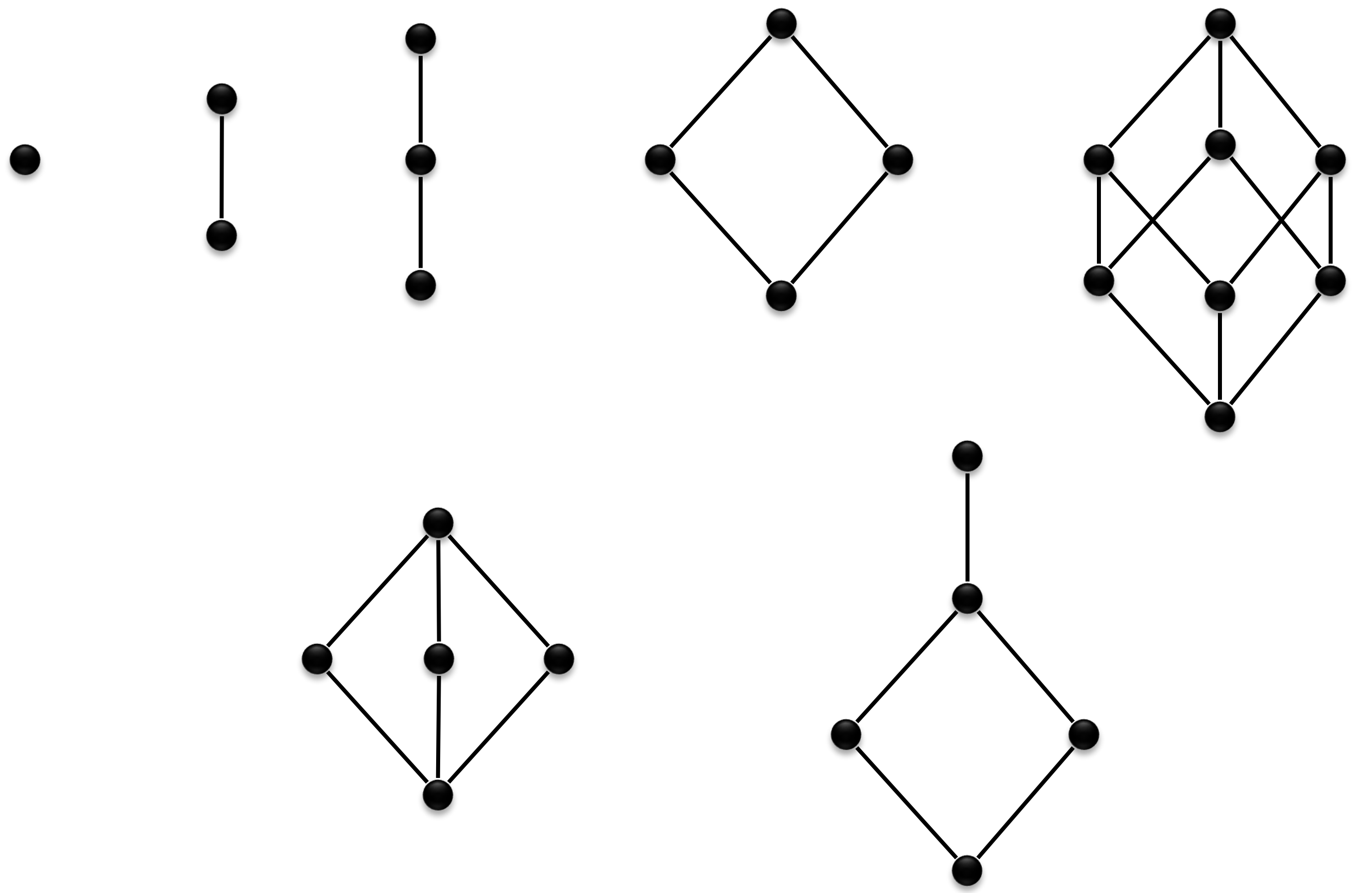
- A *lattice* is a partial order where $x \sqcup y$ and $x \sqcap y$ exist for all $x,y \in S$ ($x \sqcup y$ is notation for $\bigsqcup\{x,y\}$)
- A *complete lattice* is a partial order where $\bigsqcup X$ and $\bigsqcap X$ exist for all $X \subseteq S$

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

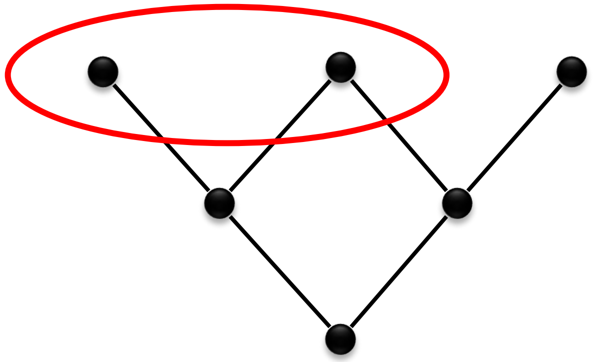
  (exercise)

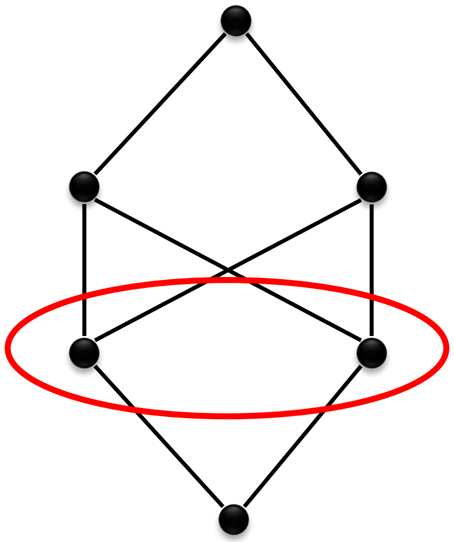
- A finite lattice is complete if $\top$ and $\bot$ exist

These partial orders are lattices



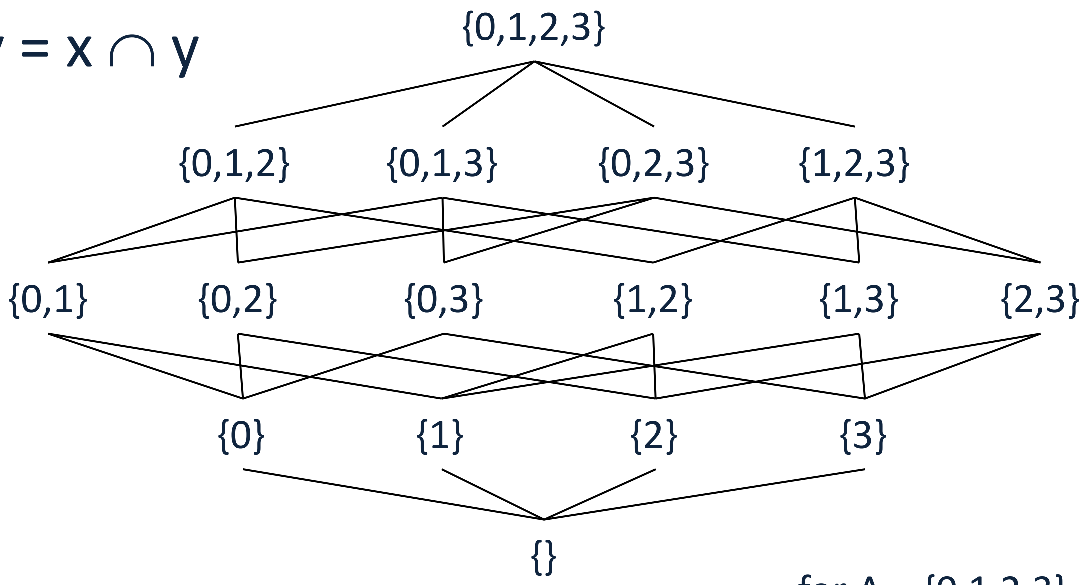
These partial orders are *not* lattices





The powerset lattice

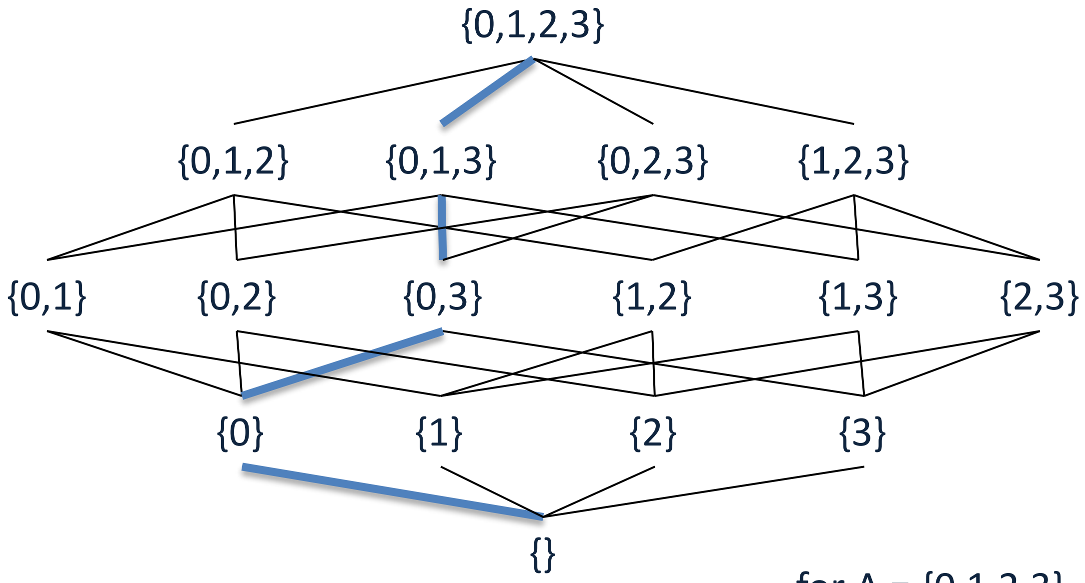
- Every finite set A defines a complete lattice ($\mathcal{P}$(A),$\subseteq$) where
  - $\bot = \emptyset$
  - $\top$ = A
  - x $\sqcup$ y = x $\cup$ y
  - x $\sqcap$ y = x $\cap$ y



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

Lattice height

- The *height* of a lattice is the length of the longest path from ⊥ to ⊤
- 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:

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

ordered pointwise

Example: $A \to L$ where
- A is the set of program variables
- L is the *Sign* lattice

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

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
- $\mathit{height}(L_1 \times L_2 \times \ldots \times L_n) = \mathit{height}(L_1) + \ldots + \mathit{height}(L_n)$

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

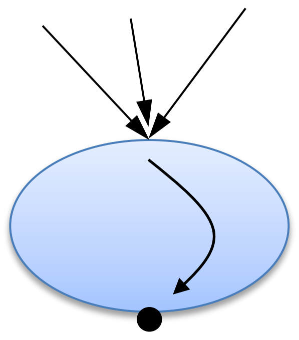
Sign analysis constraints, revisited

- The variable $[\![v]\!]$ denotes a map that gives the sign value for all variables at the program point *after* CFG node v
- $[\![v]\!] \in \mathit{State}$ where $\mathit{State} = \mathit{Var} \to \mathit{Sign}$
- For assignments:

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

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

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



where $\mathit{JOIN}(v) = \bigsqcup\limits_{w \in \mathit{pred}(v)} [\![w]\!]$ $\leftarrow$ combines information from predecessors

Generating constraints

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

$$[\![\mathit{entry}]\!] = \bot$$
$$[\![\texttt{var a,b,c}]\!] = [\![\mathit{entry}]\!][a \mapsto \top, b \mapsto \top, c \mapsto \top]$$
$$[\![\texttt{a = 42}]\!] = [\![\texttt{var a,b,c}]\!][a \mapsto +]$$
$$[\![\texttt{b = 87}]\!] = [\![\texttt{a = 42}]\!][b \mapsto +]$$
$$[\![\texttt{input}]\!] = [\![\texttt{b = 87}]\!]$$
$$[\![\texttt{c = a + b}]\!] = [\![\texttt{input}]\!][c \mapsto [\![\texttt{input}]\!](a) + [\![\texttt{input}]\!](b)]$$
$$[\![\texttt{c = a - b}]\!] = [\![\texttt{input}]\!][c \mapsto [\![\texttt{input}]\!](a) - [\![\texttt{input}]\!](b)]$$
using l.u.b. $\longrightarrow$ $$[\![\mathit{exit}]\!] = [\![\texttt{c = a + b}]\!] \sqcup [\![\texttt{c = a - b}]\!]$$

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 $\swarrow$

- 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 $f: L_1 \rightarrow L_2$ 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
- $x \sqsubseteq y$ 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
- 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*:

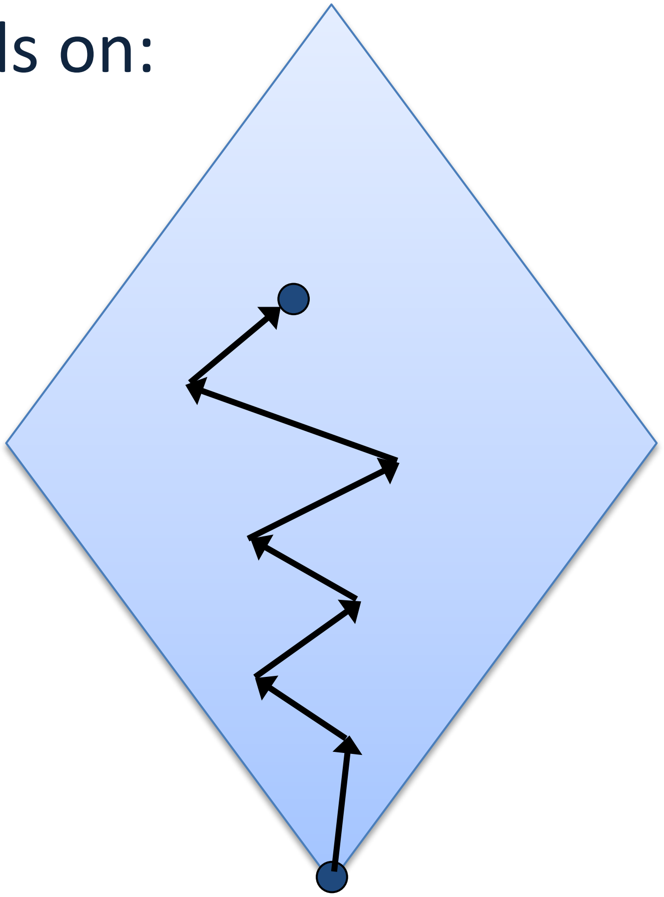
$$lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

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);
```



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

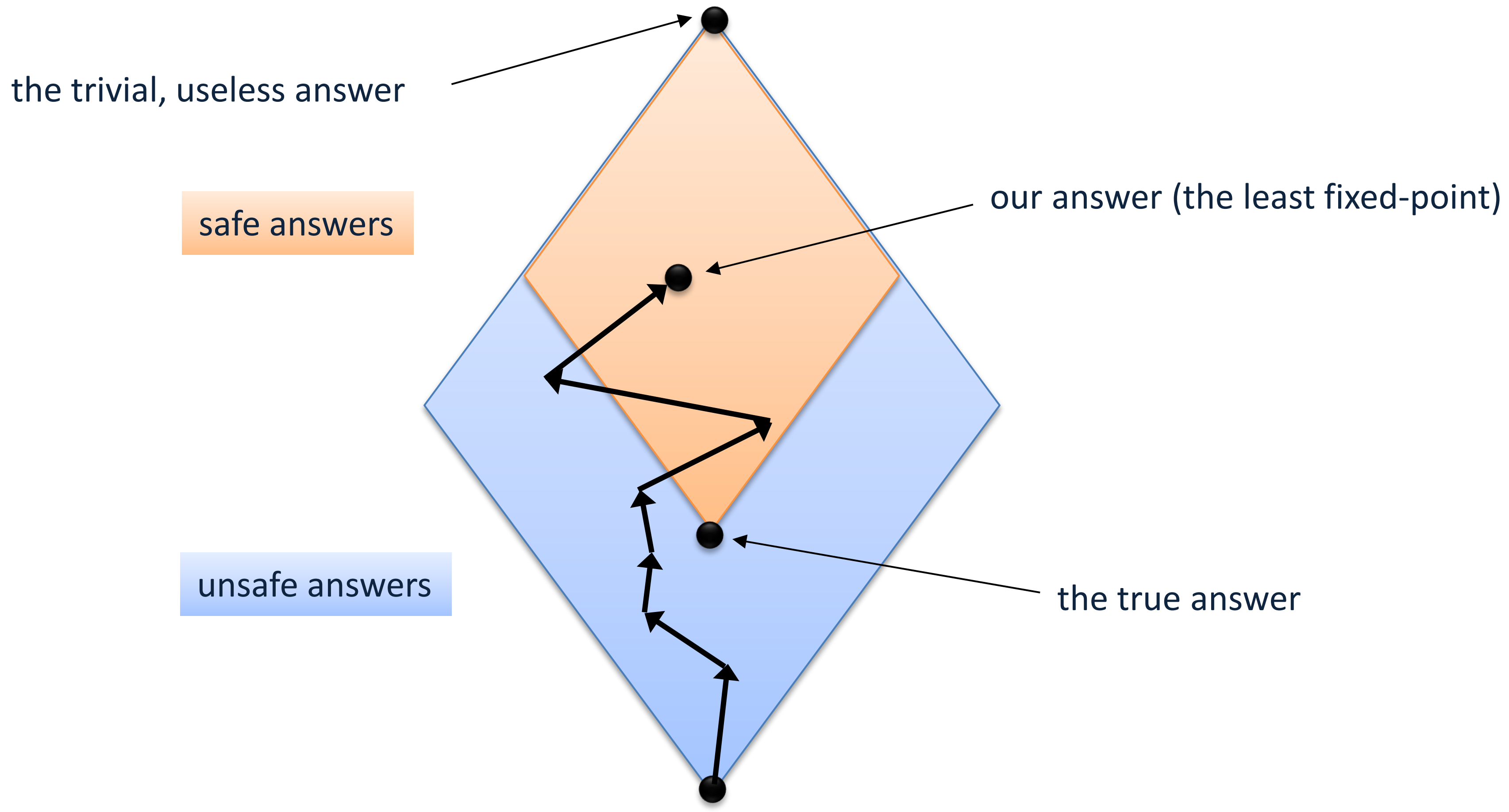
Solving equations

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

Lattice points as answers

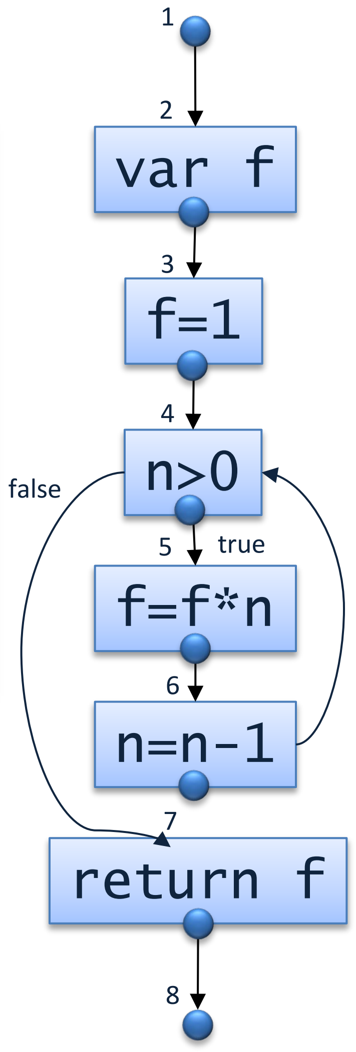


the trivial, useless answer
safe answers
our answer (the least fixed-point)
unsafe answers
the true answer

Conservative approximation…

Example: sign analysis

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



[n → I, f → ⊥]

[n → I, f → I]

[n → I, f → ±]

[n → I, f → 王]

[n → I, f → I]

[n → I, f → I]

[n → I, f → 王]

[n → I, f → 王]

Interprocedural analysis

CFG for whole programs

The idea:

- construct a CFG for each function
- then glue them together to reflect function calls and returns

We just need to take care of:

- parameter passing
- return values
- values of local variables across calls
- …assuming no global variables, no heap, and no higher-order functions

A simplifying assumption

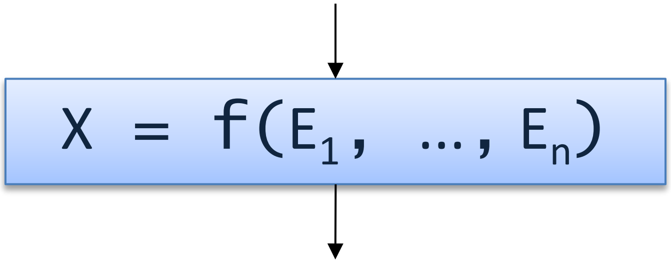
- Assume that all function calls are of the form

$$\texttt{X = f(E}_1\texttt{, …, E}_n\texttt{);}$$

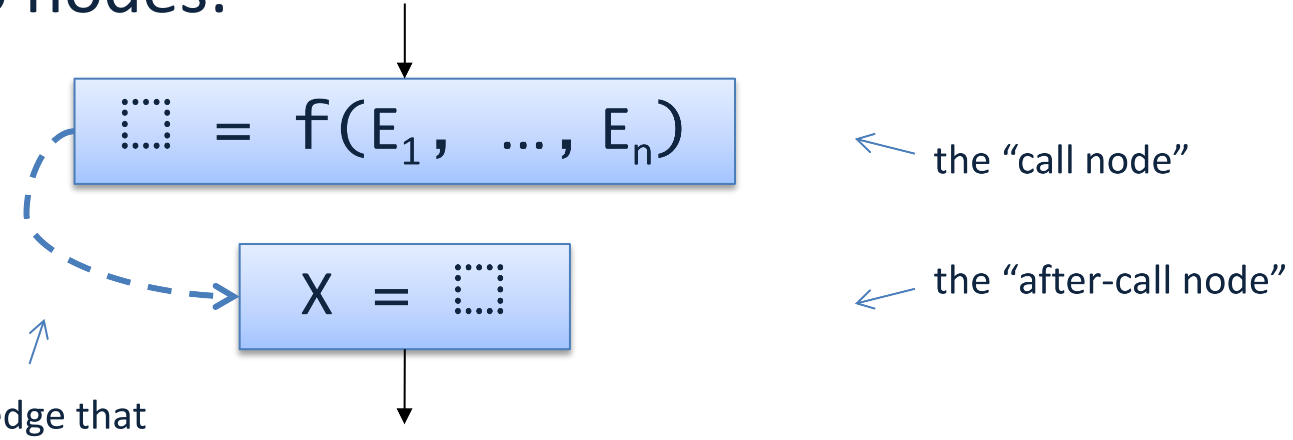
- This can always be obtained by normalization

Interprocedural CFGs (1/3)

Split each original call node



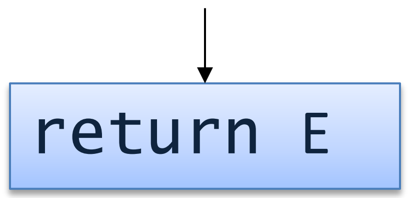
into two nodes:



a special edge that connects the call node with its after-call node

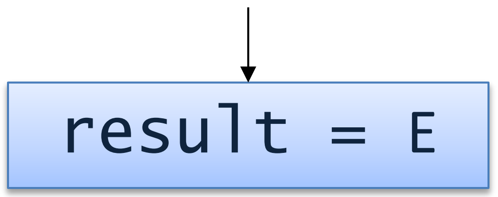
Interprocedural CFGs (2/3)

Change each return node



```
return E
```

into an assignment:

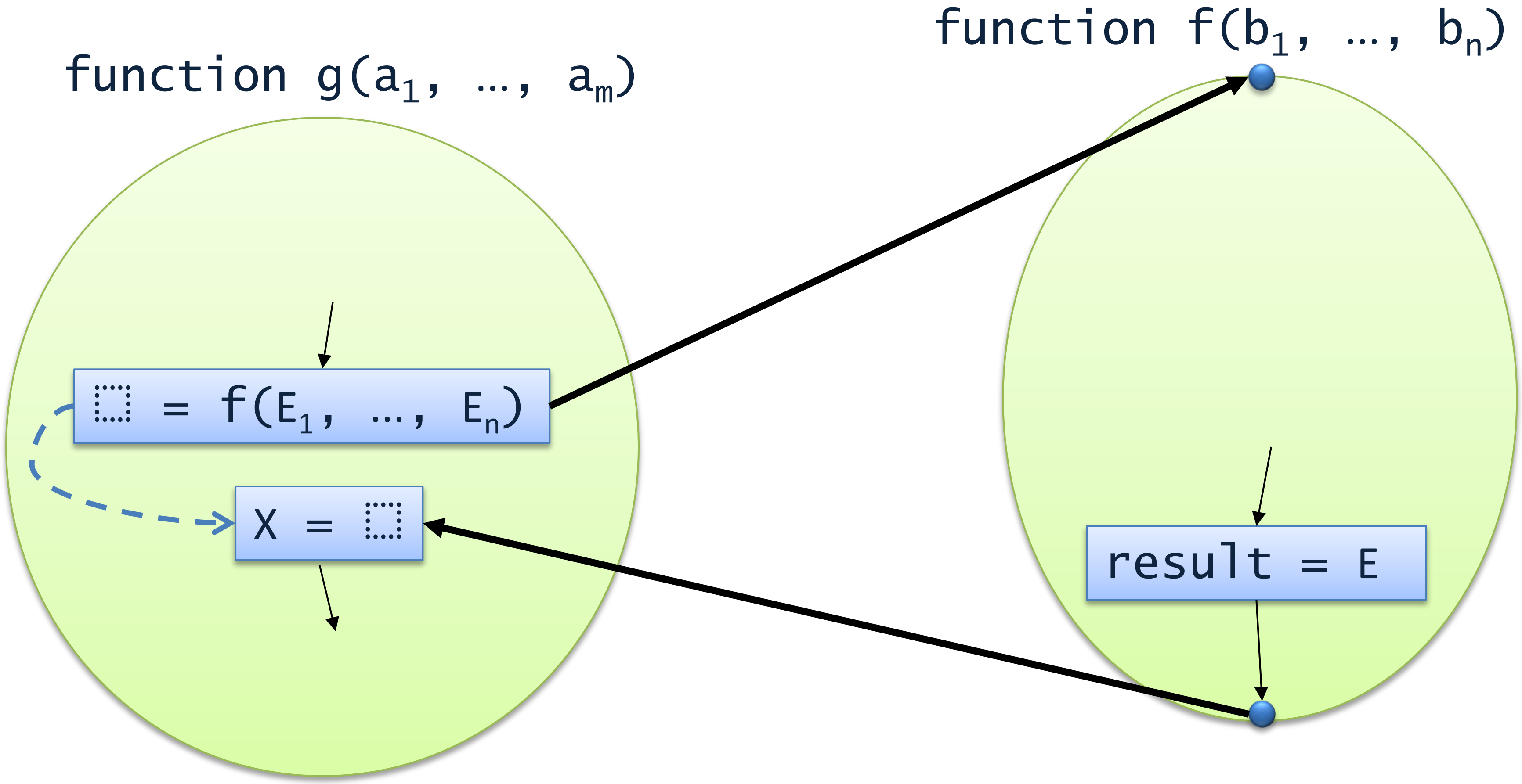


```
result = E
```

(where `result` is a fresh variable)

Interprocedural CFGs (3/3)

Add call edges and return edges:



Example: interprocedural sign analysis

- Recall the intraprocedural sign analysis…
- Lattice for abstract values:

$$\begin{tikzcd}
 & & \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\mathit{Sign} = & + \arrow[dr, dash] & - \arrow[d, dash] & 0 \arrow[dl, dash] \\
 & & \bot &
\end{tikzcd}$$

- Lattice for abstract states:

  $\mathit{Var} \rightarrow \mathit{Sign}$

Example: interprocedural sign analysis

- Constraint for entry node v of function f($b_1, \ldots, b_n$):

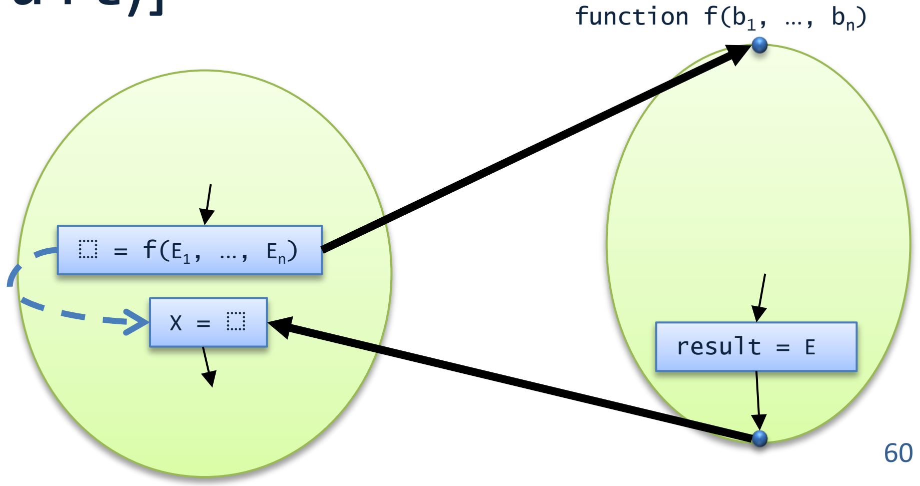
$$[\![v]\!] = \bigsqcup_{w \in pred(v)} \bot[b_1 \rightarrow eval([\![w]\!], E_1^w), \ldots, b_n \rightarrow eval([\![w]\!], E_n^w)]$$

where E_i^w is i'th argument at w

- Constraint for after-call node v labeled $X = ⬚$, with call node v':

$$[\![v]\!] = [\![v']\!][X \rightarrow [\![w]\!](\texttt{result})]$$

where $w \in pred(v)$



Context insensitive analysis

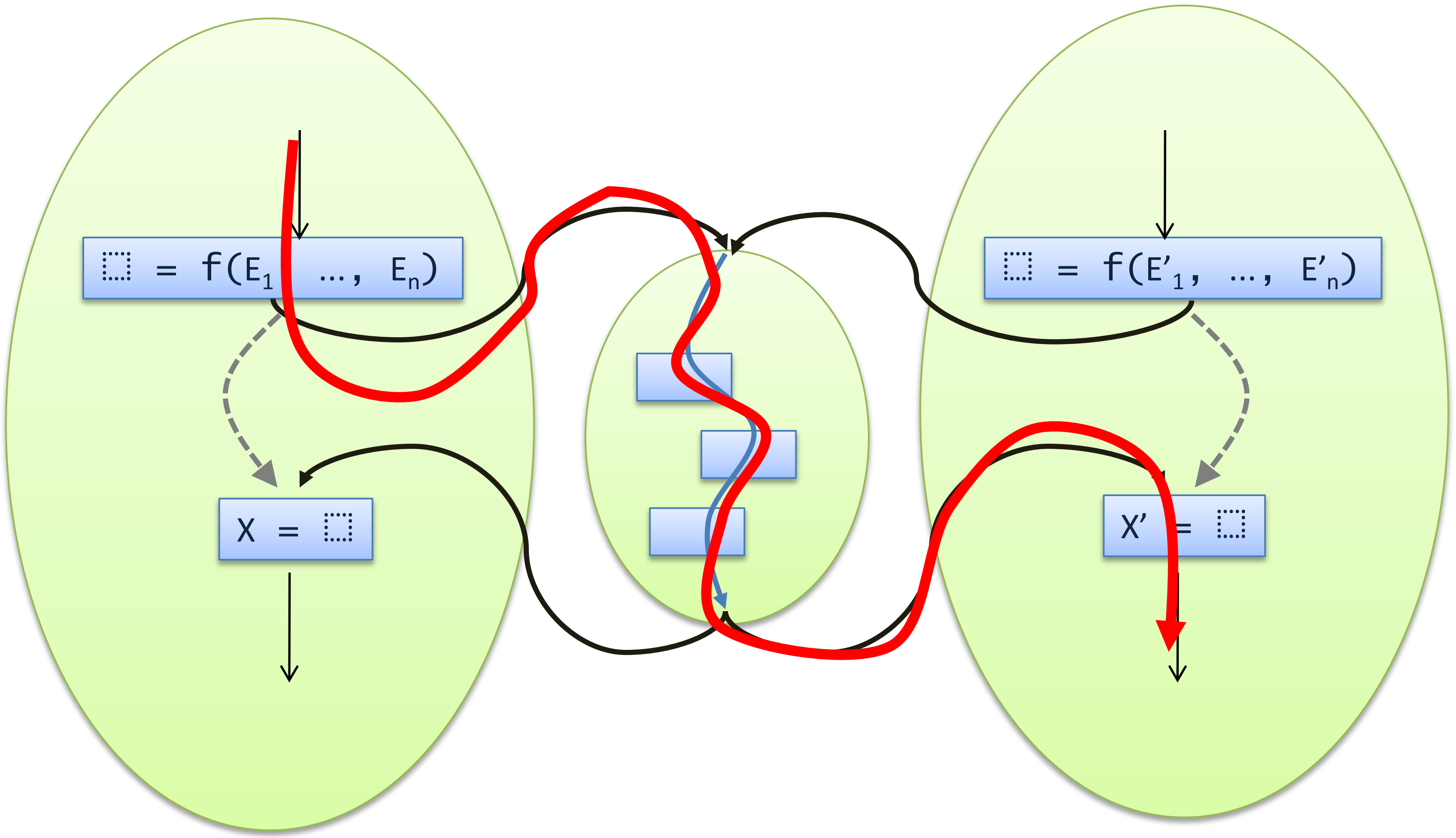
What is the sign of the return value of g?

```
f(z) {
  return z*42;
}

g() {
  var x,y;
  x = f(0);
  y = f(87);
  return x + y;
}
```

Our current analysis says "T"

Interprocedurally invalid paths



Function cloning (alternatively, function inlining)

- Clone functions such that each function has only one callee

- Can avoid interprocedurally invalid paths ☺
- For high nesting depths, gives exponential blow-up 😐
- Doesn’t work on (mutually) recursive functions ☹

- Use heuristics to determine when to apply (trade-off between CFG size and precision)

Example, with cloning

What is the sign of the return value of g?

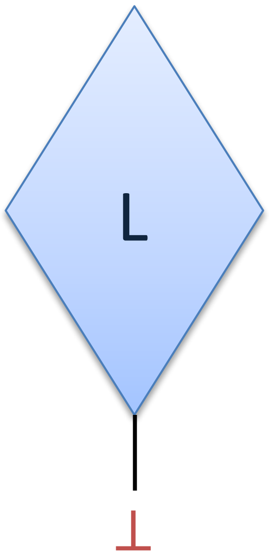
```
f1(z1) {
  return z1*42;
}

f2(z2) {
  return z2*42;
}

g() {
  var x,y;
  x = f1(0);
  y = f2(87);
  return x + y;
}
```

Lift lattice

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



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

Context sensitive analysis

- Function cloning provides a kind of context sensitivity

- Instead of physically copying the function CFGs, do it *logically*
- Replace the lattice for abstract states, *State*, by

$$\boxed{\quad Context \rightarrow lift(State) \quad}$$

  where *Context* is a set of ***call contexts***
  - the contexts are abstractions of the state at function entry
  - *Context* must be finite to ensure finite height of the lattice
  - the bottom element of *lift*(*State*) represents "unreachable" contexts
- Different strategies for choosing the set *Context*…

Constraints for CFG nodes that do not involve function calls and returns

Easily adjusted to the new lattice $\mathit{Context} \to \mathit{lift}(\mathit{State})$

Example if v is an assignment node $x = E$ in sign analysis:

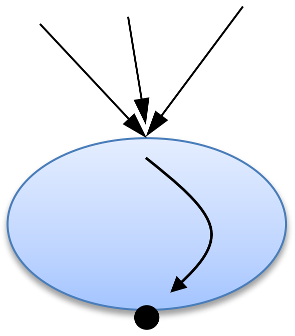
$[\![\, \mathrm{v} \,]\!] = \mathit{JOIN}(\mathrm{v})[x \mapsto \mathit{eval}(\mathit{JOIN}(\mathrm{v}),E)]$

becomes

$$[\![\, \mathrm{v} \,]\!](\mathrm{c}) = \begin{cases} s[x \mapsto \mathit{eval}(s,E)] & \text{if } s = \mathit{JOIN}(\mathrm{v},\mathrm{c}) \in \mathit{State} \\ \text{unreachable} & \text{if } \mathit{JOIN}(\mathrm{v},\mathrm{c}) = \text{unreachable} \end{cases}$$

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

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



One-level cloning

- Let $c_1,\ldots,c_n$ be the call nodes in the program
- Define *Context*$=\{c_1,\ldots,c_n\} \cup \{\varepsilon\}$
  - each call node now defines its own “call context” (using ε to represent the call context at the main function)
  - the context is then like the return address of the top-most stack frame in the call stack
- Same effect as one-level cloning, but without actually copying the function CFGs
- Usually straightforward to generalize the constraints for a context insensitive analysis to this lattice
- (Example: context-sensitive sign analysis – later…)

The call string approach

- Let $c_1,\ldots,c_n$ be the call nodes in the program
- Define *Context* as the set of strings over $\{c_1,\ldots,c_n\}$ of length $\leq k$
  - such a string represents the top-most k call locations on the call stack
  - the empty string ε again represents the initial call context at the main function
- For k=1 this amounts to one-level cloning

Example:
interprocedural sign analysis with call strings (k=1)

Lattice for abstract states: *Context* → *lift*(*Var* → *Sign*)
where *Context*={ε, c_1, c_2}

```
f(z) {
  var t1,t2;
  t1 = z*6;
  t2 = t1*7;
  return t2;
}
...
x = f(0);  // c1
y = f(87); // c2
...
```

[ε ↦ unreachable,
 c1 ↦ ⊥[z↦0, t1↦0, t2↦0],
 c2 ↦ ⊥[z↦+, t1↦+, t2↦+]]

What is an example program that requires **k=2** to avoid loss of precision?

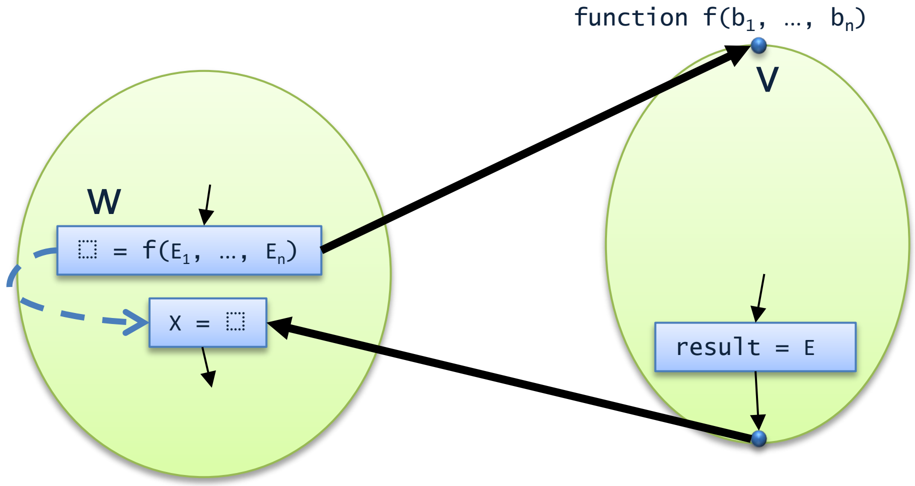
Context sensitivity with call strings

function entry nodes, for k=1

Constraint for entry node v of function f(b_1, …, b_n):
(if not ‘main’)

$$[\![v]\!](c) = \bigsqcup_{\substack{w \in pred(v)\ \wedge \\ c = w\ \wedge \\ c' \in Context}} s_w^{c'}$$

only consider the call node w that matches the context c



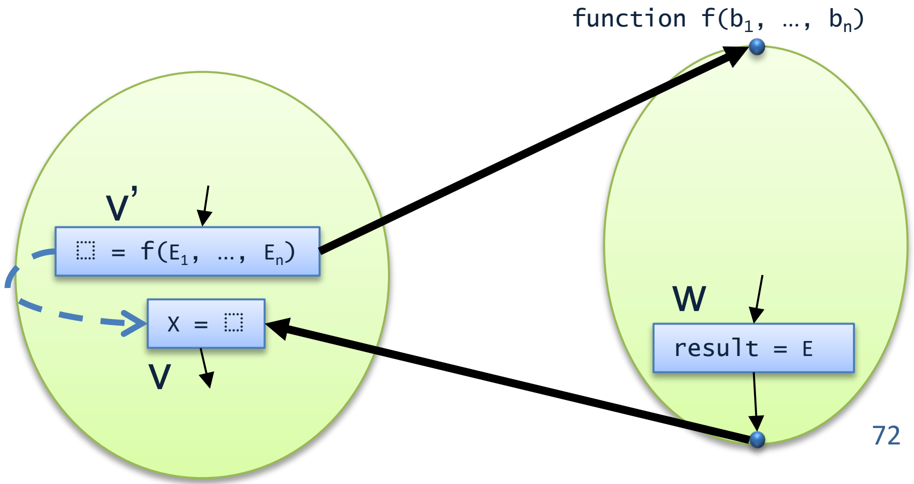
$$s_w^{c'} = \begin{cases} \text{unreachable} & \text{if } [\![w]\!](c') = \text{unreachable} \\ \perp[b_1 \rightarrow eval([\![w]\!](c'), E_1^w), \ldots, b_n \rightarrow eval([\![w]\!](c'), E_n^w)] & \text{otherwise} \end{cases}$$

Context sensitivity with call strings

after-call nodes, for k=1

Constraint for after-call node v labeled $X = ⬚$,
with call node v' and exit node $w \in \text{pred}(v)$:

$$[\![v]\!](c) = \begin{cases} \text{unreachable} & \text{if } [\![v']\!](c) = \text{unreachable} \lor [\![w]\!](v') = \text{unreachable} \\ [\![v']\!](c)[X \rightarrow [\![w]\!](v')(\texttt{result})] & \text{otherwise} \end{cases}$$



The functional approach

- The call string approach considers *control flow*
  - but why distinguish between two different call sites if their abstract states are the same?
- The functional approach instead considers *data*
- In the most general form, choose

  $$\mathit{Context} = \mathit{State}$$

  (requires *State* to be finite)
- Each element of the lattice $\mathit{State} \rightarrow \mathit{lift}(\mathit{State})$
  is now a map m that provides an element m(x) from *State* (or "unreachable") for each possible x
  where x describes the state at function entry

Example:
interprocedural sign analysis with the functional approach

Lattice for abstract states: $\mathit{Context} \rightarrow \mathit{lift}(\mathit{Var} \rightarrow \mathit{Sign})$
where $\mathit{Context} = \mathit{Var} \rightarrow \mathit{Sign}$

```
f(z) {
  var t1,t2;
  t1 = z*6;
  t2 = t1*7;
  return t2;
}
...
x = f(0);
y = f(87);
...
```

$[\bot[\mathtt{z}\mapsto 0] \mapsto \bot[\mathtt{z}\mapsto 0, \mathtt{t1}\mapsto 0, \mathtt{t2}\mapsto 0],$
$\bot[\mathtt{z}\mapsto +] \mapsto \bot[\mathtt{z}\mapsto +, \mathtt{t1}\mapsto +, \mathtt{t2}\mapsto +],$
all other contexts $\mapsto$ unreachable $]$

Another example:

interprocedural sign analysis with the functional approach

Lattice for abstract states: $Context \rightarrow lift(Var \rightarrow Sign)$

where $Context = Var \rightarrow Sign$

```
f(z) {
  var t1,t2;
  t1 = z*6;
  t2 = t1*7;
  return t2;
}
g(a) {
  return f(a);
}
...
x = g(0);
y = g(87);
```

$[\bot[z\mapsto 0] \mapsto \bot[z\mapsto 0,\ t1\mapsto 0,\ t2\mapsto 0],$
$\bot[z\mapsto +] \mapsto \bot[z\mapsto +,\ t1\mapsto +,\ t2\mapsto +],$
all other contexts $\mapsto$ unreachable $]$

The functional approach

- The lattice element for a function exit node is thus a ***function summary*** that maps abstract function input to abstract function output
- This can be exploited at call nodes!
- When entering a function with abstract state x:
  - consider the function summary s for that function
  - if s(x) already has been computed, use that to model the entire function body, then proceed directly to the after-call node
- Avoids the problem with interprocedurally invalid paths!
- …but may be expensive if *State* is large

Example:

interprocedural sign analysis with the functional approach

Lattice for abstract states: $Context \rightarrow lift(Var \rightarrow Sign)$

where $Context = Var \rightarrow Sign$

```
f(z) {
  var t1,t2;
  t1 = z*6;
  t2 = t1*7;
  return t2;
}
...
x = f(0);
y = f(87);
z = f(42);
...
```

The abstract state at the exit of `f` can be used as a function summary

[⊥[z↦0] ↦ ⊥[z↦0, `t1`↦0, `t2`↦0, `result`↦0],
⊥[z↦+] ↦ ⊥[z↦+, `t1`↦+, `t2`↦+, `result`↦+],
all other contexts ↦ unreachable]

At this call, we can reuse the already computed exit abstract state of `f` for the context ⊥[z↦+]

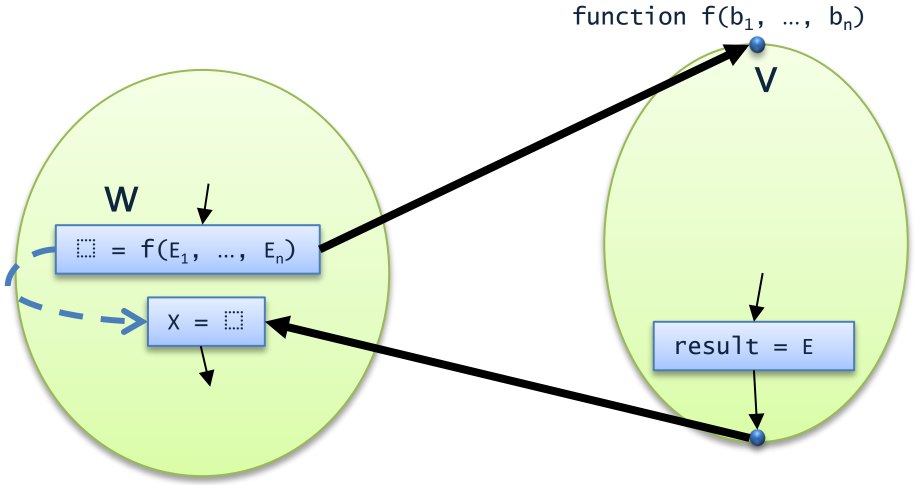
Context sensitivity with the functional approach

function entry nodes

Constraint for entry node v of function f($b_1, \ldots, b_n$):
(if not 'main')

$$[\![v]\!](c) = \bigsqcup_{\substack{w \in pred(v) \land \\ c = s_w^{c'} \land \\ c' \in Context}} s_w^{c'}$$

only considerthe call node w if the abstract state from that node matches the context c



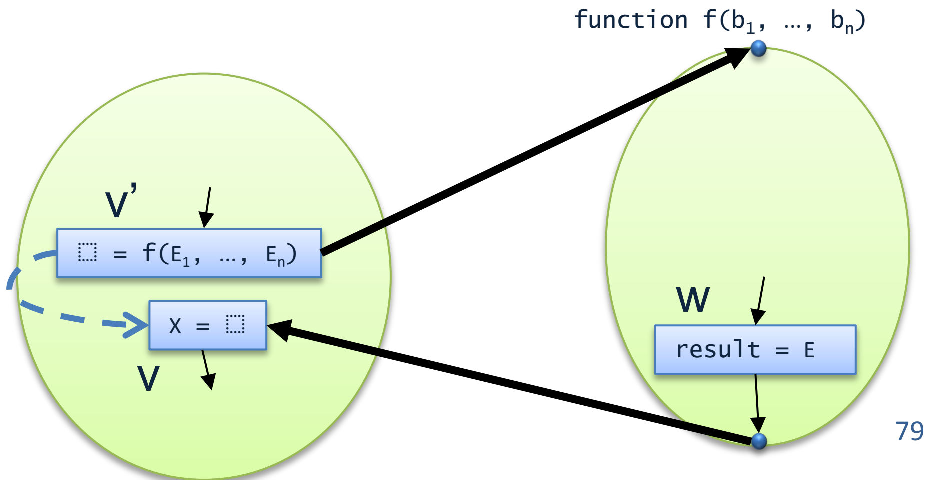
where $s_w^{c'}$ is defined as before

Context sensitivity with the functional approach

after-call nodes

Constraint for after-call node v labeled $X = ⬚$, with call node v' and exit node $w \in pred(v)$:

$$[\![v]\!](c) = \begin{cases} \text{unreachable} & \text{if } [\![v']\!](c) = \text{unreachable} \lor [\![w]\!](s^c_{v'}) = \text{unreachable} \\ [\![v']\!](c)[X \rightarrow [\![w]\!](s^c_{v'})(\texttt{result})] & \text{otherwise} \end{cases}$$



Choosing the right context sensitivity strategy

- The call string approach is expensive for k>1
  - solution: choose k adaptively for each call site
- The functional approach is expensive if *State* is large
  - solution: only consider selected parts of the abstract state as context, for example abstract information about the function parameter values (called *parameter sensitivity*), or, in object-oriented languages, abstract information about the receiver object 'this' (called *object sensitivity* or *type sensitivity*)

Control flow analysis

TIP with first-class functions

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = f(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

Control flow complications

- First-class functions in TIP complicate CFG construction:
  - several functions may be invoked at a call site
  - this depends on the dataflow
  - but dataflow analysis first requires a CFG

- Same situation for other features:
  - function values with free variables (closures)
  - a class hierarchy with objects and methods
  - prototype objects with dynamic properties

Control flow analysis

- A control flow analysis approximates the call graph
  - conservatively computes possible functions at call sites
- Control flow analysis is usually flow-*insensitive*:
  - based on the AST
  - the call graph can be used for an interprocedural CFG
  - a subsequent dataflow analysis may use the CFG
- Alternative: use flow-sensitive analysis
  - potentially on-the-fly, during dataflow analysis

CFA constraints (1/2)

- Tokens are all functions $\{f_1, f_2, \ldots, f_k\}$
- For every AST node, v, we introduce the variable $[\![v]\!]$ denoting the set of functions to which v may evaluate

- For function definitions $f(\ldots)\{\ldots\}$:

  $f \in [\![f]\!]$

- For assignments $x = E$:

  $[\![E]\!] \subseteq [\![x]\!]$

CFA constraints (2/2)

- For **direct** function calls $f(E_1, \ldots, E_n)$:

  $[\![E_i]\!] \subseteq [\![a_i]\!]$ for i=1,...,n $\wedge$ $[\![E']\!] \subseteq [\![f(E_1, \ldots, E_n)]\!]$

  where f is a function with arguments $a_1, \ldots, a_n$ and return expression E'

- For **computed** function calls $E(E_1, \ldots, E_n)$:

  $f \in [\![E]\!] \Rightarrow ([\![E_i]\!] \subseteq [\![a_i]\!]$ for i=1,...,n $\wedge$ $[\![E']\!] \subseteq [\![(E)(E_1, \ldots, E_n)]\!])$

  for every function f with arguments $a_1, \ldots, a_n$

  and return expression E'

  – If we consider typable programs only:

    only generate constraints for those functions f
    for which the call would be type correct

Generated constraints

$\mathtt{inc} \in \llbracket \mathtt{inc} \rrbracket$

$\mathtt{dec} \in \llbracket \mathtt{dec} \rrbracket$

$\mathtt{ide} \in \llbracket \mathtt{ide} \rrbracket$

$\llbracket \mathtt{ide} \rrbracket \subseteq \llbracket \mathtt{f} \rrbracket$

$\llbracket \mathtt{f(n)} \rrbracket \subseteq \llbracket \mathtt{r} \rrbracket$

$\mathtt{inc} \in \llbracket \mathtt{f} \rrbracket \Rightarrow \llbracket \mathtt{n} \rrbracket \subseteq \llbracket \mathtt{i} \rrbracket \wedge \llbracket \mathtt{i+1} \rrbracket \subseteq \llbracket \mathtt{f(n)} \rrbracket$

$\mathtt{dec} \in \llbracket \mathtt{f} \rrbracket \Rightarrow \llbracket \mathtt{n} \rrbracket \subseteq \llbracket \mathtt{j} \rrbracket \wedge \llbracket \mathtt{j-1} \rrbracket \subseteq \llbracket \mathtt{f(n)} \rrbracket$

$\mathtt{ide} \in \llbracket \mathtt{f} \rrbracket \Rightarrow \llbracket \mathtt{n} \rrbracket \subseteq \llbracket \mathtt{k} \rrbracket \wedge \llbracket \mathtt{k} \rrbracket \subseteq \llbracket \mathtt{f(n)} \rrbracket$

$\llbracket \mathtt{input} \rrbracket \subseteq \llbracket \mathtt{x} \rrbracket$

$\llbracket \mathtt{foo(x,inc)} \rrbracket \subseteq \llbracket \mathtt{y} \rrbracket$

$\llbracket \mathtt{foo(x,dec)} \rrbracket \subseteq \llbracket \mathtt{y} \rrbracket$

$\mathtt{foo} \in \llbracket \mathtt{foo} \rrbracket$

$$\left.\begin{array}{l} \mathtt{foo} \in \llbracket \mathtt{foo} \rrbracket \Rightarrow \llbracket \mathtt{x} \rrbracket \subseteq \llbracket \mathtt{n} \rrbracket \wedge \llbracket \mathtt{inc} \rrbracket \subseteq \llbracket \mathtt{f} \rrbracket \wedge \llbracket \mathtt{r} \rrbracket \subseteq \llbracket \mathtt{foo(x,inc)} \rrbracket \\ \mathtt{foo} \in \llbracket \mathtt{foo} \rrbracket \Rightarrow \llbracket \mathtt{x} \rrbracket \subseteq \llbracket \mathtt{n} \rrbracket \wedge \llbracket \mathtt{dec} \rrbracket \subseteq \llbracket \mathtt{f} \rrbracket \wedge \llbracket \mathtt{r} \rrbracket \subseteq \llbracket \mathtt{foo(x,dec)} \rrbracket \end{array}\right\} \text{assuming we do not use the special rule for direct calls}$$

$\mathtt{main} \in \llbracket \mathtt{main} \rrbracket$

(At each call we only consider functions with matching number of parameters)

Least solution

$[\![\texttt{inc}]\!] = \{\texttt{inc}\}$

$[\![\texttt{dec}]\!] = \{\texttt{dec}\}$

$[\![\texttt{ide}]\!] = \{\texttt{ide}\}$

$[\![\texttt{f}]\!] = \{\texttt{inc}, \texttt{dec}, \texttt{ide}\}$

$[\![\texttt{foo}]\!] = \{\texttt{foo}\}$

$[\![\texttt{main}]\!] = \{\texttt{main}\}$

(the solution is the empty set for the remaining constraint variables)

With this information, we can construct the call edges and return edges in the interprocedural CFG

The cubic framework

- We have a set of tokens $T=\{t_1, t_2, \ldots, t_k\}$
- We have a collection of constraint variables $V=\{x_1, \ldots, x_n\}$ ranging over subsets of tokens
- A collection of constraints of these forms:
  - $t \in x$
  - $x \subseteq y$
  - $t \in x \Rightarrow y \subseteq z$
- Compute the unique minimal solution
  - this exists since solutions are closed under intersection
- A cubic time algorithm exists!

The solver data structure

- Each variable is mapped to a node in a directed graph
- Each node has a bitvector in $\{0,1\}^k$
  - initially set to all 0’s
- Each bit has a list of pairs of variables
  - used to model conditional constraints
- The edges model inclusion constraints
- The bitvectors will at all times directly represent the minimal solution to the constraints seen so far

The solver data structure

- $x.sol \subseteq T$: the set of tokens for x (the bitvectors)
- $x.succ \subseteq V$: the successors of x (the edges)
- $x.cond(t) \subseteq V\times V$: the conditional constraints for x and t
- $W \subseteq T\times V$: a worklist (initially empty)

Adding constraints

- $t \in x$

  ```
  addToken(t, x)
  propagate()
  ```

- $x \subseteq y$

  ```
  addEdge(x, y)
  propagate()
  ```

- $t \in x \Rightarrow y \subseteq z$

  ```
  if t ∈ x.sol
    addEdge(y, z)
    propagate()
  else
    add (y, z) to x.cond(t)
  ```

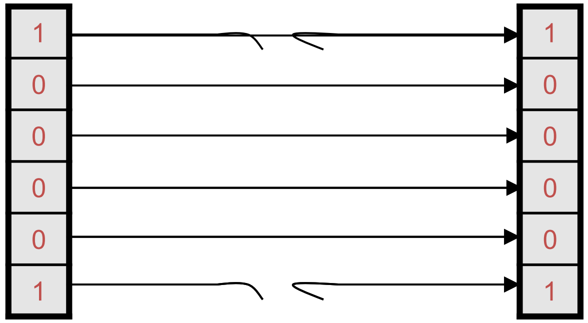
```
addToken(t, x):
  if t ∉ x.sol
    add t to x.sol
    add (t, x) to W

addEdge(x, y):
  if x ≠ y ∧ y ∉ x.succ
    add y to x.succ
    for each t in x.sol
      addToken(t, y)

propagate():
  while W ≠ ∅
    pick and remove (t, x) from W
    for each (y, z) in x.cond(t)
      addEdge(y, z)
    for each y in x.succ
      addToken(t, y)
```

Time complexity

- O(*n*) functions and O(*n*) applications, with program size *n*
- O(n) singleton constraints, O(n) subset constraints, O(n^2) conditional constraints
- O(n) nodes, O(n^2) edges, O(n) bits per node
- addToken takes time O(1)
- addEdge takes amortized time O(n)
- Each pair (t, x) is processed at most once by propagate
- O(n^2) calls to addEdge (either immediately or via propagate)
- O(n^3) calls to addToken



Time complexity

- Adding it all up, the upper bound is O(n^3)
- This is known as the *cubic time bottleneck*:
  - occurs in many different scenarios
  - but O(n^3/log n) is possible…

Implementation tricks

- Cycle elimination (collapse nodes if there is a cycle of inclusion constraints)
- Process worklist in topological order
- Interleaving solution propagation and constraint processing
- Shared bit vector representation
- Type filtering
- On-demand processing
- Difference propagation
- Subsumed node compaction
- …

Points-to analysis

Analyzing programs with pointers

How do we perform e.g.
constant propagation analysis
when the programming language
has pointers?
(or object references?)

```
...
*x = 42;
*y = -87;
z = *x;
// is z 42 or -87?
```

Points-to analysis

- Determine for each pointer variable X the set $pt(X)$ of the cells X may point to
- A *conservative* ("may points-to") analysis:
  - the set may be too large
  - can show absence of aliasing: $pt(X) \cap pt(Y) = \emptyset$
- We'll focus on *flow-insensitive* analyses:
  - take place on the AST
  - before or together with the control-flow analysis

Pointer normalization

- Assume that all pointer usage is normalized:
  - $X =$ `alloc` P where P is `null` or an integer constant
  - $X = \&Y$
  - $X = Y$
  - $X = {*Y}$
  - ${*X} = Y$
  - $X =$ `null`
- Simply introduce lots of temporary variables…
- All sub-expressions are now named
- We choose to ignore the fact that the cells created at variable declarations are uninitialized (otherwise it is impossible to get useful results from a flow-insensitive analysis)

Andersen's analysis (1/2)

- For every cell *c*, introduce a constraint variable $[\![c]\!]$ ranging over sets of cells, i.e. $[\![\cdot]\!]: \mathit{Cell} \rightarrow \mathcal{P}(\mathit{Cell})$

- Generate constraints:
  - $X = \texttt{alloc}\ P$: $\texttt{alloc-}i \in [\![X]\!]$
  - $X = \&Y$: $Y \in [\![X]\!]$
  - $X = Y$: $[\![Y]\!] \subseteq [\![X]\!]$
  - $X = {*}Y$: $\mathrm{c} \in [\![Y]\!] \Rightarrow [\![\mathrm{c}]\!] \subseteq [\![X]\!]$ for each $\mathrm{c} \in \mathit{Cell}$
  - ${*}X = Y$: $\mathrm{c} \in [\![X]\!] \Rightarrow [\![Y]\!] \subseteq [\![\mathrm{c}]\!]$ for each $\mathrm{c} \in \mathit{Cell}$
  - $X = \texttt{null}$: (no constraints)

(For the conditional constraints, there's no need to add a constraint for the cell x if &x does not occur in the program)

Andersen’s analysis (2/2)

- The points-to map is defined as:

  $pt(X) = [\![X]\!]$

- The constraints fit into the cubic framework ☺
- Unique minimal solution in time $O(n^3)$
- In practice, for Java: $O(n^2)$

- The analysis is flow-insensitive but *directional*
  - models the direction of the flow of values in assignments

Interprocedural pointer analysis

- In TIP, function values and pointers may be mixed together:

```
(***x)(1,2,3)
```

- In this case the CFA and the points-to analysis must happen *simultaneously*!

- The idea: Treat function values as a kind of pointers

Function call normalization

- Assume that all function calls are of the form

$$X = X_0(X_1, \ldots, X_n)$$

- Assume that all return statements are of the form

`return` X'`;`

- As usual, simply introduce lots of temporary variables…

- Include all function names in the set *Cell*

CFA with Andersen

Andersen's analysis is already closely connected to control-flow analysis!

- For the function call

  $$X = X_0(X_1, \ldots, X_n)$$

  and every occurrence of

  $$f(X'_1, \ldots, X'_n)\ \{ \ldots \texttt{return}\ X'; \}$$

  add these constraints:

  $$f \in [\![f]\!]$$

  $$f \in [\![X_0]\!] \Rightarrow ([\![X_i]\!] \subseteq [\![X'_i]\!] \text{ for i=1,...,}n \wedge [\![X']\!] \subseteq [\![X]\!])$$

- (Similarly for simple function calls)
- Fits directly into the cubic framework!

Context-sensitive pointer analysis

```
foo(a) {
  return *a;
}

bar() {
  ...
  x = alloc null; // alloc-1
  y = alloc null; // alloc-2
  *x = alloc null; // alloc-3
  *y = alloc null; // alloc-4
  ...
  q = foo(x);
  w = foo(y);
  ...
}
```

Are q and w aliases?

Context-sensitive pointer analysis

- Generalize the abstract domain $\mathit{Cell} \rightarrow \mathcal{P}(\mathit{Cell})$ to
  $$\mathit{Context} \rightarrow \mathit{Cell} \rightarrow \mathcal{P}(\mathit{Cell})$$
  (or equivalently: $\mathit{Cell} \times \mathit{Context} \rightarrow \mathcal{P}(\mathit{Cell})$)
  where *Context* is a (finite) set of call contexts
- As usual, many possible choices of the set *Context*
  – recall the call string approach and the functional approach
- We can also track the set of reachable contexts
  (like the use of lifted lattices earlier):
  $$\mathit{Context} \rightarrow \mathrm{lift}(\mathit{Cell} \rightarrow \mathcal{P}(\mathit{Cell}))$$

- Does this still fit into the cubic solver?

Context-sensitive pointer analysis

```
mk() {
  return alloc null; // alloc-1
}

baz() {
  var x,y;
  x = mk();
  y = mk();
  ...
}
```

Are x and y aliases?

$[\![x]\!] = \{\texttt{alloc-1}\}$
$[\![y]\!] = \{\texttt{alloc-1}\}$

Context-sensitive pointer analysis

- We can go one step further and introduce *context-sensitive heap* (a.k.a. *heap cloning*)
- Let each abstract cell be a pair of
  - `alloc-`*i* (the `alloc` with index *i*) or *X* (a program variable)
  - a heap context from a (finite) set *HeapContext*
- This allows abstract cells to be named by the source code allocation site ***and (information from) the current context***
- One choice:
  - set *HeapContext* = *Context*
  - at `alloc`, use the entire current call context as heap context

Context-sensitive pointer analysis with heap cloning

Assuming we use the call string approach with k=1, so *Context* = {ε, c1, c2}, and *HeapContext* = *Context*

```
mk() {
  return alloc null; // alloc-1
}

baz() {
  var x,y;
  x = mk(); // c1
  y = mk(); // c2
  ...
}
```

Are x and y aliases?

$[\![x]\!] = \{ (\texttt{alloc-1}, c1) \}$

$[\![y]\!] = \{ (\texttt{alloc-1}, c2) \}$

Abstract interpretation

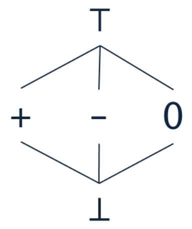
Abstract interpretation provides a solid mathematical foundation for reasoning about static program analyses

- Is my analysis **sound**? (Does it safely approximate the actual program behavior?)
- Is it as **precise** as possible for the currently used analysis lattice? If not, where can precision losses arise? Which precision losses can be avoided (without sacrificing soundness)?

Answering such questions requires a precise definition of the semantics of the programming language, and precise definitions of the analysis abstractions in terms of the semantics

Sign analysis, recap

$$Sign \;=$$



$$State = Var \rightarrow Sign$$

$$State^n$$

$$\begin{aligned}
[\![v_1]\!] &= af_{v_1}([\![v_1]\!], \ldots, [\![v_n]\!]) \\
[\![v_2]\!] &= af_{v_2}([\![v_1]\!], \ldots, [\![v_n]\!]) \\
&\vdots \\
[\![v_n]\!] &= af_{v_n}([\![v_1]\!], \ldots, [\![v_n]\!])
\end{aligned}$$

$$af([\![v_1]\!], \ldots, [\![v_n]\!]) = \big(af_{v_1}([\![v_1]\!], \ldots, [\![v_n]\!]), \ldots, af_{v_n}([\![v_1]\!], \ldots, [\![v_n]\!])\big)$$

$af : State^n \rightarrow State^n$ is the analysis representation of the given program

$$[\![P]\!] = lfp(af)$$

Program semantics as constraint systems

$$ConcreteState = Var \hookrightarrow \mathbb{Z}$$

$$\llbracket v \rrbracket \subseteq ConcreteState$$

This is called a *reachable states collecting semantics*

The semantics of expressions

$$ceval : ConcreteState \times Exp \rightarrow \mathcal{P}(\mathbb{Z})$$

$$\begin{aligned}
&ceval(\rho, X) = \{\rho(X)\}\\
&ceval(\rho, I) = \{I\}\\
&ceval(\rho, \texttt{input}) = \mathbb{Z}\\
&ceval(\rho, E_1 \, \texttt{op} \, E_2) = \{v_1 \, \texttt{op} \, v_2 \mid v_1 \in ceval(\rho, E_1) \ \wedge \ v_2 \in ceval(\rho, E_2)\}
\end{aligned}$$

$$ceval(R, E) = \bigcup_{\rho \in R} ceval(\rho, E)$$

Successors and joins

$$csucc\colon ConcreteState \times Node \to \mathcal{P}(Node)$$

$$csucc(R, v) = \bigcup_{\rho \in R} csucc(\rho, v)$$

$$CJOIN(v) =$$
$$\{\rho \in ConcreteState \mid \exists w \in Node\colon \rho \in [\![w]\!] \wedge v \in csucc(\rho, w)\}$$

Semantics of statements

$$\{\!\!\{X{=}E\}\!\!\} = \{\rho[X \mapsto z] \mid \rho \in CJOIN(v) \;\wedge\; z \in ceval(\rho, E)\}$$

$$\{\!\!\{\mathtt{var}\ X_1\mathtt{,}\ \ldots\ \mathtt{,}X_n\}\!\!\} = \\ \{\rho[X_1 \mapsto z_1, \ldots, X_n \mapsto z_n] \mid \rho \in CJOIN(v) \wedge z_1 \in \mathbb{Z} \wedge \cdots \wedge z_n \in \mathbb{Z}\}$$

$$\{\!\!\{entry\}\!\!\} = \{[]\}$$

$$\{\!\!\{v\}\!\!\} = CJOIN(v)$$

The resulting constraint system

$$\begin{aligned}
\{\![v_1]\!\} &= cf_{v_1}(\{\![v_1]\!\}, \ldots, \{\![v_n]\!\}) \\
\{\![v_2]\!\} &= cf_{v_2}(\{\![v_1]\!\}, \ldots, \{\![v_n]\!\}) \\
&\vdots \\
\{\![v_n]\!\} &= cf_{v_n}(\{\![v_1]\!\}, \ldots, \{\![v_n]\!\})
\end{aligned}$$

$$cf_v : \left(\mathcal{P}(ConcreteState)\right)^n \to \mathcal{P}(ConcreteState)$$

$$cf(x_1, \ldots, x_n) = \left(cf_{v_1}(x_1, \ldots, x_n), \ldots, cf_{v_n}(x_1, \ldots, x_n)\right)$$

is the semantic representation of the given program

$$x = cf(x)$$

$$\{\![P]\!\} = lfp(cf)$$

Example

```
var x;
x = 0;
while (input) {
  x = x + 2;
}
```

	solution 1	solution 2
$[\![entry]\!]$	$\{[]\}$	$\{[]\}$
$[\![$`var x`$]\!]$	$\{[\mathtt{x} \mapsto z] \mid z \in \mathbb{Z}\}$	$\{[\mathtt{x} \mapsto z] \mid z \in \mathbb{Z}\}$
$[\![$`x = 0`$]\!]$	$\{[\mathtt{x} \mapsto 0]\}$	$\{[\mathtt{x} \mapsto 0]\}$
$[\![$`input`$]\!]$	$\{[\mathtt{x} \mapsto z] \mid z \in \{0, 2, 4, \dots\}\}$	$\{[\mathtt{x} \mapsto z] \mid z \in \mathbb{Z}\}$
$[\![$`x = x + 2`$]\!]$	$\{[\mathtt{x} \mapsto z] \mid z \in \{2, 4, \dots\}\}$	$\{[\mathtt{x} \mapsto z] \mid z \in \mathbb{Z}\}$
$[\![exit]\!]$	$\{[\mathtt{x} \mapsto z] \mid z \in \{0, 2, 4, \dots\}\}$	$\{[\mathtt{x} \mapsto z] \mid z \in \mathbb{Z}\}$

the least solution

Kleene's fixed point theorem for complete join morphisms

$f : L \to L$ is a *complete join morphism* if $f(\bigsqcup A) = \bigsqcup_{a \in A} f(a)$ for every $A \subseteq L$

If *f* is a complete join morphism:

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

(even when *L* has infinite height!)

$(\mathcal{P}(\mathit{ConcreteState}))^n$ is a complete lattice (a product of a powerset lattice)

cf is a complete join morphism

Tarski's fixed-point theorem

In a complete lattice L, every monotone function $f : L \to L$ has a unique least fixed point given by $\bigsqcap\{x \in L \mid f(x) \sqsubseteq x\}$.

cf is monotone

Semantics vs. analysis

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

$$\{\![\texttt{b = 87}]\!\} = \{[\mathsf{a} \mapsto 42, \mathsf{b} \mapsto 87, \mathsf{c} \mapsto z] \mid z \in \mathbb{Z}\}$$
$$\{\![\texttt{c = a - b}]\!\} = \{[\mathsf{a} \mapsto 42, \mathsf{b} \mapsto 87, \mathsf{c} \mapsto -45]\}$$
$$\{\![\mathit{exit}]\!\} = \{[\mathsf{a} \mapsto 42, \mathsf{b} \mapsto 87, \mathsf{c} \mapsto 129], [\mathsf{a} \mapsto 42, \mathsf{b} \mapsto 87, \mathsf{c} \mapsto -45]\}$$

$$[\![\texttt{b = 87}]\!] = [\mathsf{a} \mapsto +, \mathsf{b} \mapsto +, \mathsf{c} \mapsto \top]$$
$$[\![\texttt{c = a - b}]\!] = [\mathsf{a} \mapsto +, \mathsf{b} \mapsto +, \mathsf{c} \mapsto \top]$$
$$[\![\mathit{exit}]\!] = [\mathsf{a} \mapsto +, \mathsf{b} \mapsto +, \mathsf{c} \mapsto \top]$$

Abstraction functions for sign analysis

$$\alpha_{\mathrm{a}} : \mathcal{P}(\mathbb{Z}) \to \mathit{Sign}$$
$$\alpha_{\mathrm{b}} : \mathcal{P}(\mathit{ConcreteState}) \to \mathit{State}$$
$$\alpha_{\mathrm{c}} : \big(\mathcal{P}(\mathit{ConcreteState})\big)^n \to \mathit{State}^n$$

$$\alpha_{\mathrm{a}}(D) = \begin{cases} \bot & \text{if } D \text{ is empty} \\ + & \text{if } D \text{ is nonempty and contains only positive integers} \\ - & \text{if } D \text{ is nonempty and contains only negative integers} \\ \mathbf{0} & \text{if } D \text{ is nonempty and contains only the integer } 0 \\ \top & \text{otherwise} \end{cases}$$
for any $D \in \mathcal{P}(\mathbb{Z})$

$$\alpha_{\mathrm{b}}(R) = \sigma \text{ where } \sigma(X) = \alpha_{\mathrm{a}}(\{\rho(X) \mid \rho \in R\})$$
for any $R \subseteq \mathit{ConcreteState}$ and $X \in \mathit{Var}$

$$\alpha_{\mathrm{c}}(R_1, \ldots, R_n) = (\alpha_b(R_1), \ldots, \alpha_b(R_n))$$
for any $R_1, \ldots, R_n \subseteq \mathit{ConcreteState}$

Concretization functions for sign analysis

$$\gamma_{\mathrm{a}}\colon \mathit{Sign} \rightarrow \mathcal{P}(\mathbb{Z})$$
$$\gamma_{\mathrm{b}}\colon \mathit{State} \rightarrow \mathcal{P}(\mathit{ConcreteState})$$
$$\gamma_{\mathrm{c}}\colon \mathit{State}^n \rightarrow \big(\mathcal{P}(\mathit{ConcreteState})\big)^n$$

$$\gamma_{\mathrm{a}}(s) = \begin{cases} \emptyset & \text{if } s = \bot \\ \{1, 2, 3, \ldots\} & \text{if } s = \texttt{+} \\ \{-1, -2, -3, \ldots\} & \text{if } s = \texttt{-} \\ \{0\} & \text{if } s = \mathbf{0} \\ \mathbb{Z} & \text{if } s = \top \end{cases}$$
for any $s \in \mathit{Sign}$

$$\gamma_{\mathrm{b}}(\sigma) = \{\rho \in \mathit{ConcreteState} \mid \rho(X) \in \gamma_{\mathrm{a}}(\sigma(X)) \text{ for all } X \in \mathit{Var}\}$$
for any $\sigma \in \mathit{State}$

$$\gamma_{\mathrm{c}}(\sigma_1, \ldots, \sigma_n) = (\gamma_{\mathrm{b}}(\sigma_1), \ldots, \gamma_{\mathrm{b}}(\sigma_n))$$
for any $(\sigma_1, \ldots, \sigma_n) \in \mathit{State}^n$

Monotonicity of abstraction and concretization functions

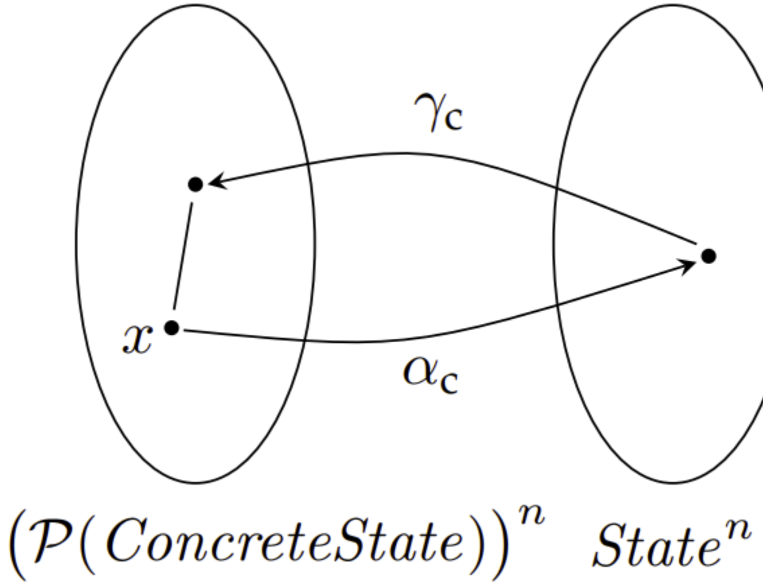
Concretatization functions are, like abstraction functions, naturally monotone.

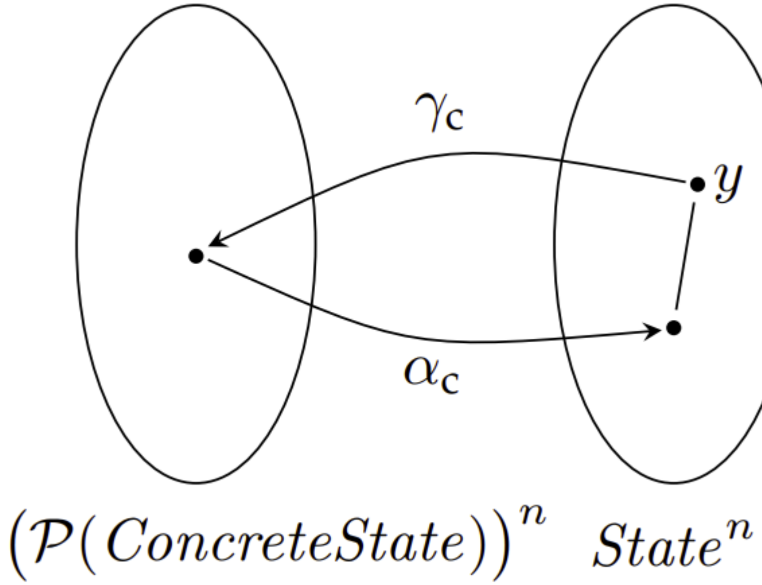
(A larger set of concrete values should correspond to a larger abstract state, and conversely)

Galois connections

The pair of monotone functions, α and γ, is called a *Galois connection* if

$$\gamma \circ \alpha \text{ is extensive}$$
$$\alpha \circ \gamma \text{ is reductive}$$





all three pairs of abstraction and concretization functions (α_a, γ_a), (α_b, γ_b), and (α_c, γ_c) from the sign analysis example are Galois connections

Galois connections

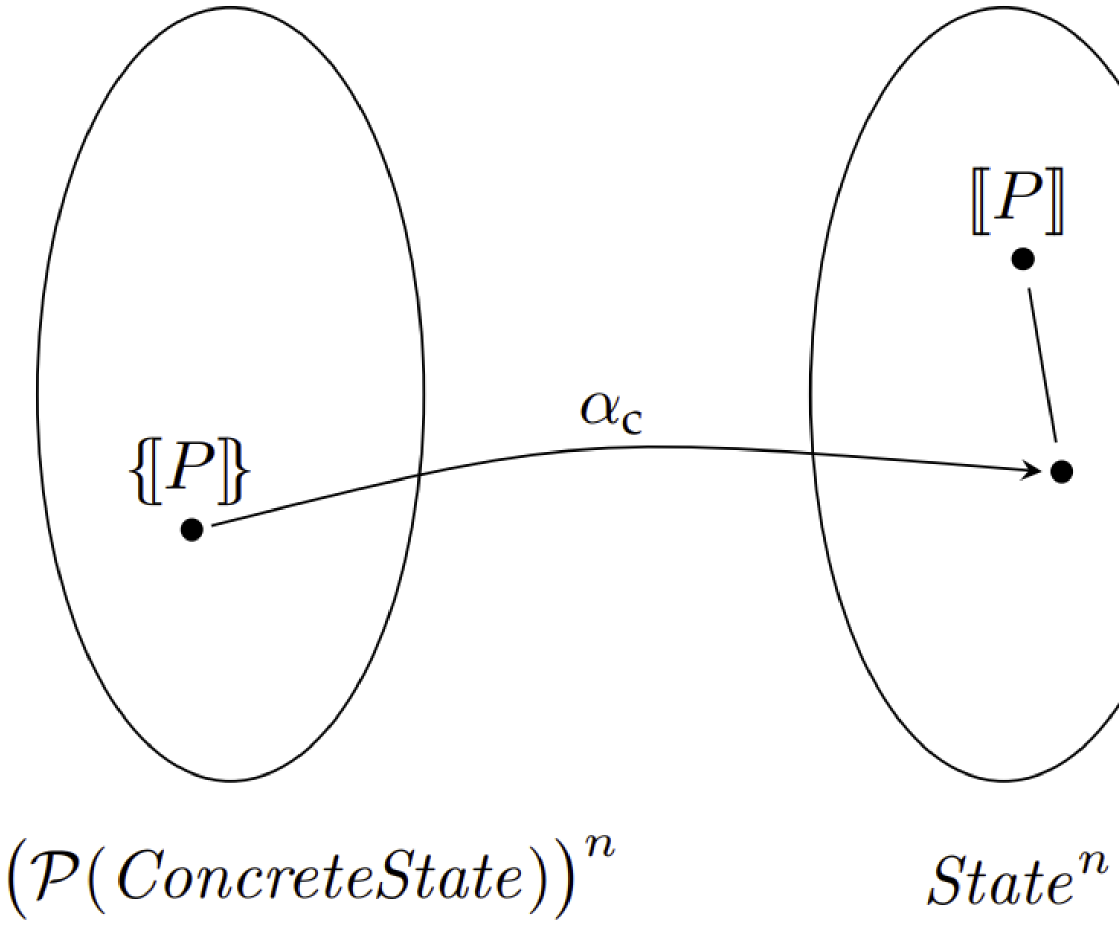
For Galois connections, the concretization function uniquely determines the abstraction function and vice versa:

$$\gamma(y) = \bigsqcup\{x \in L_1 \mid \alpha(x) \sqsubseteq y\}$$

$$\alpha(x) = \bigsqcap\{y \in L_2 \mid x \sqsubseteq \gamma(y)\}$$

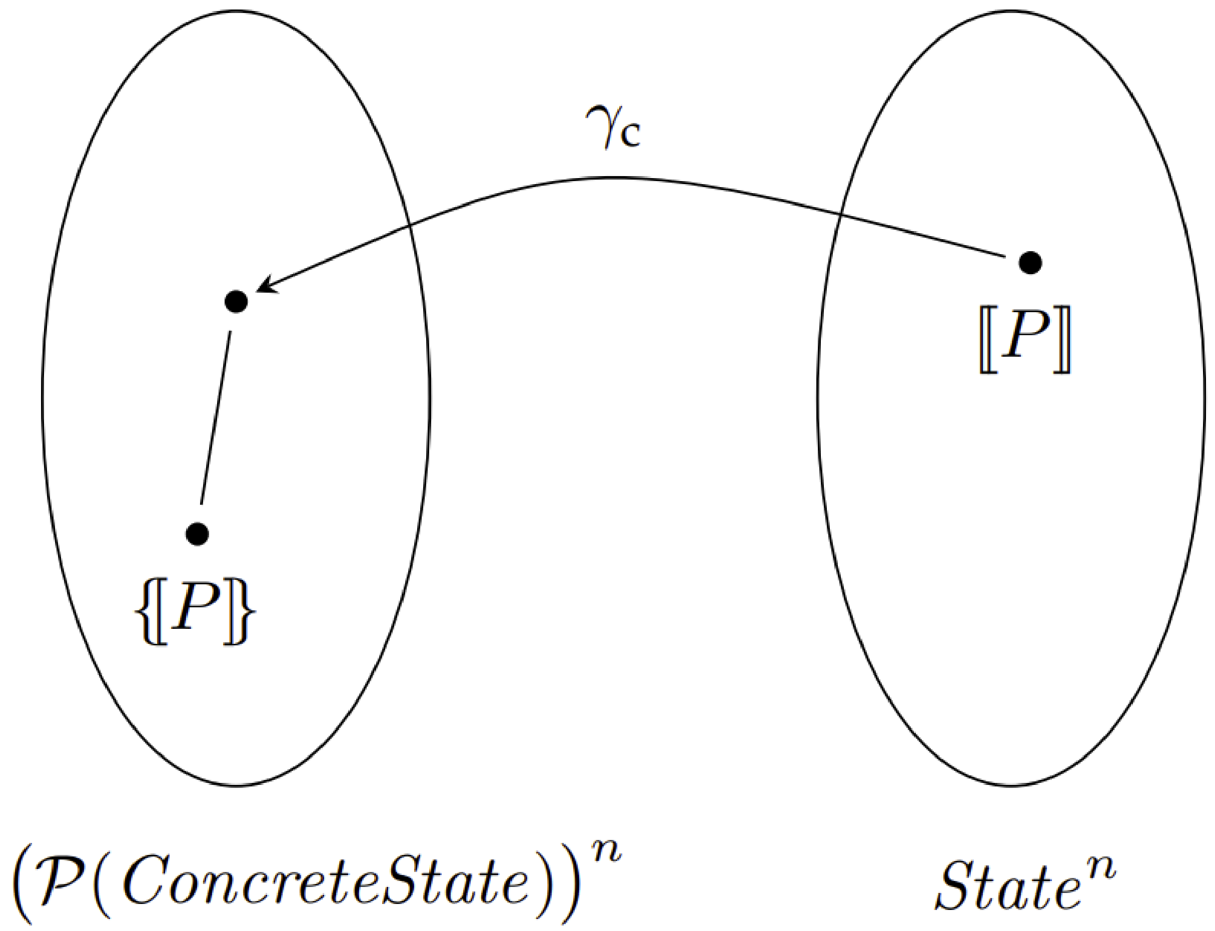
Soundness

$$\alpha(\{\![P]\!\}) \sqsubseteq [\![P]\!]$$



Soundness

$$\{\!|P|\!\} \sqsubseteq \gamma([\![P]\!])$$



Sound abstractions

$$\alpha_{\mathsf{a}}(ceval(R, E)) \sqsubseteq eval(\alpha_{\mathsf{b}}(R), E)$$

$$csucc(R, v) \subseteq succ(v) \text{ for any } R \subseteq ConcreteState$$

$$\alpha_{\mathsf{b}}(CJOIN(v)) \sqsubseteq JOIN(v)$$

if $\alpha_{\mathsf{b}}(\{[\![w]\!]\}) \sqsubseteq [\![w]\!]$ for all $w \in Node$

Sound abstractions

if v represents an assignment statement $X = E$:

$$cf_v(\{\![v_1]\!\}, \ldots, \{\![v_n]\!\}) = \{\rho[X \mapsto z] \mid \rho \in CJOIN(v) \;\wedge\; z \in ceval(\rho, E)\}$$
$$af_v([\![v_1]\!], \ldots, [\![v_n]\!]) = \sigma[X \mapsto eval(\sigma, E)] \text{ where } \sigma = JOIN(v)$$

$$\alpha_{\mathsf{b}}(cf_v(R_1, \ldots, R_n)) \sqsubseteq af_v(\alpha_{\mathsf{b}}(R_1), \ldots, \alpha_{\mathsf{b}}(R_n))$$

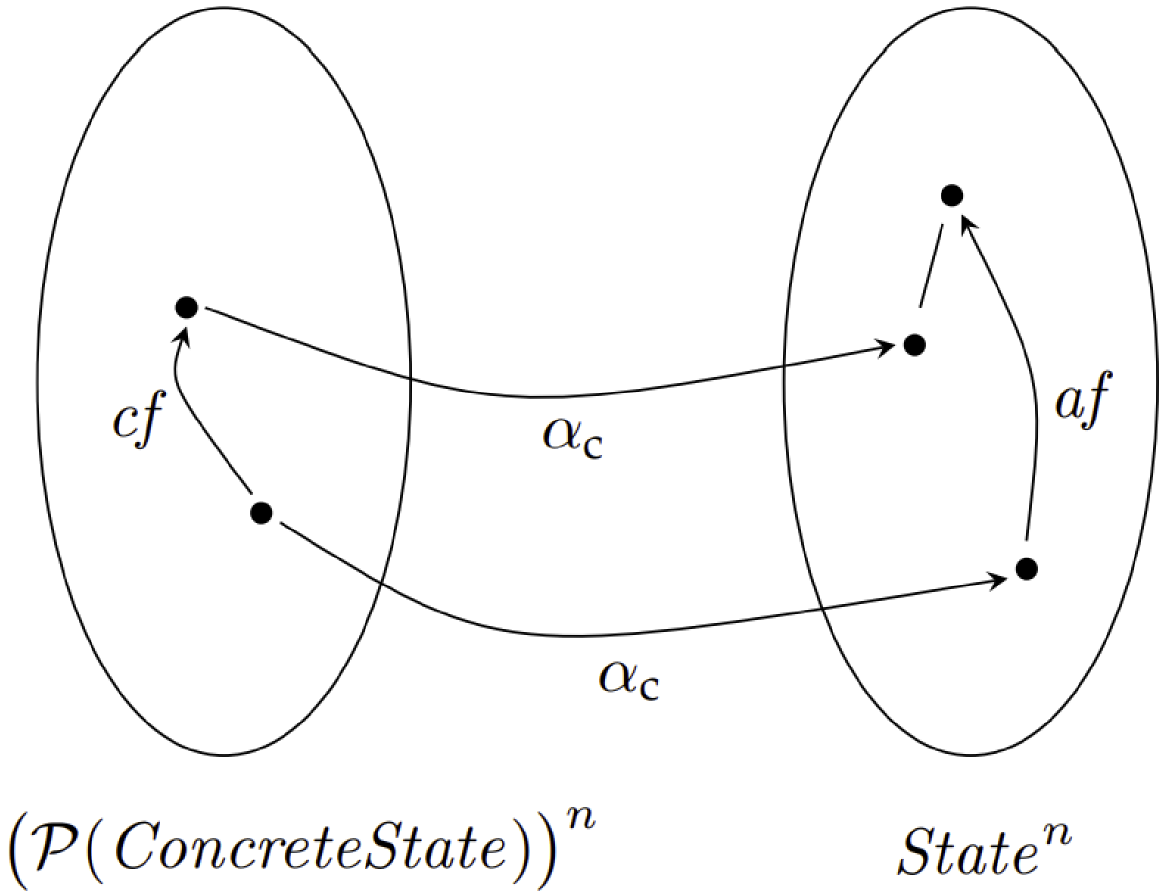
The two constraint systems

$$cf(\{\![v_1]\!\}, \ldots, \{\![v_n]\!\}) = \left(cf_{v_1}(\{\![v_1]\!\}, \ldots, \{\![v_n]\!\}), \ldots, cf_{v_n}(\{\![v_1]\!\}, \ldots, \{\![v_n]\!\})\right)$$

$$af([\![v_1]\!], \ldots, [\![v_n]\!]) = \left(af_{v_1}([\![v_1]\!], \ldots, [\![v_n]\!]), \ldots, af_{v_n}([\![v_1]\!], \ldots, [\![v_n]\!])\right)$$

Sound abstractions

$$\alpha_c(cf(R_1, \ldots, R_n)) \sqsubseteq af(\alpha_c(R_1, \ldots, R_n))$$



Sound abstractions

$$\alpha \circ cf \sqsubseteq af \circ \alpha$$

$$cf \circ \gamma \sqsubseteq \gamma \circ af$$

Equivalent, if α and γ form a Galois connection

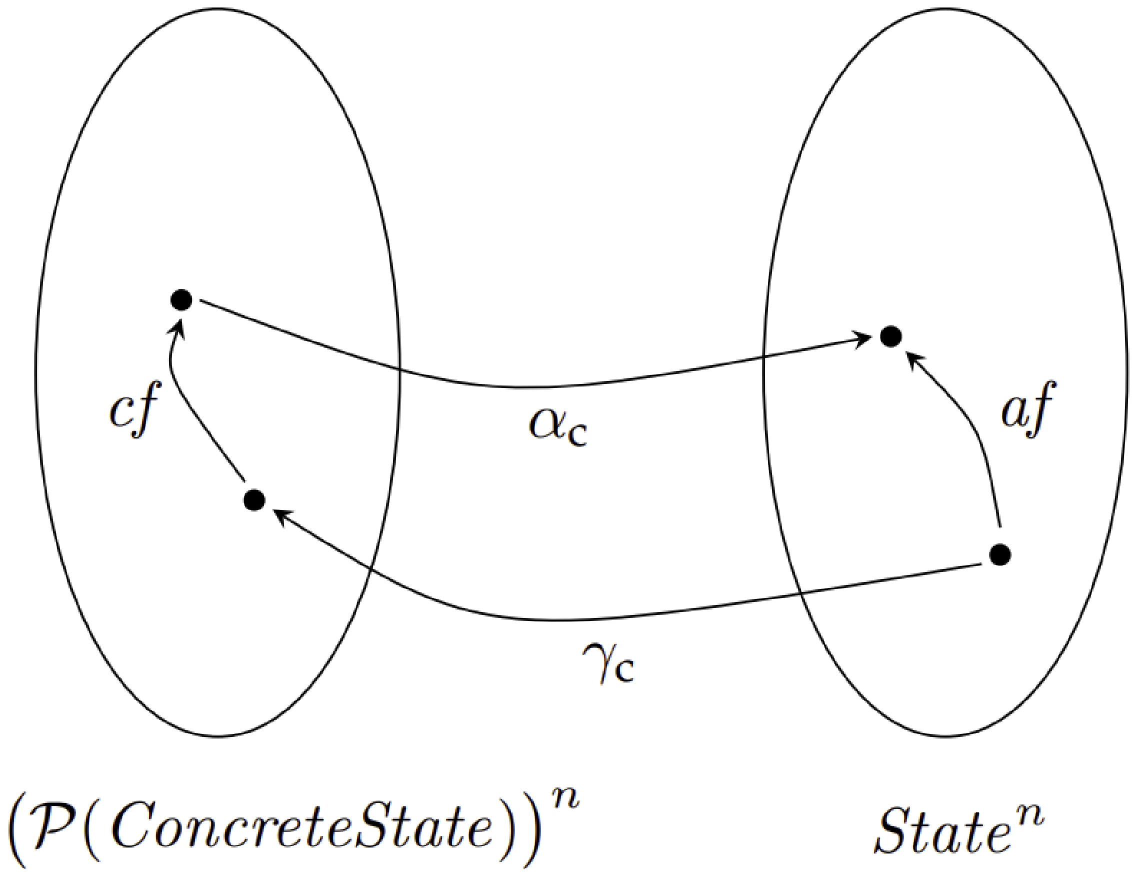
The soundness theorem

If L_1 and L_2 are complete lattices with a concretization function $\gamma : L_2 \to L_1$, $\mathit{cf} : L_1 \to L_1$ and $\mathit{af} : L_2 \to L_2$ are monotone, and af is a sound abstraction of cf with respect to γ, i.e., $\mathit{cf} \circ \gamma \sqsubseteq \gamma \circ \mathit{af}$, then $\mathit{lfp}(\mathit{cf}) \sqsubseteq \gamma(\mathit{lfp}(\mathit{af}))$.

Optimal abstractions

af is an optimal abstraction of *cf* if

$$af = \alpha \circ cf \circ \gamma$$



Optimal abstractions in sign analysis?

$\widehat{*}$ is optimal:

$$s_1 \widehat{*} s_2 = \alpha_a(\gamma_a(s_1) \cdot \gamma_a(s_2))$$

eval is *not* optimal:

$$\sigma(\mathtt{x}) = \top$$

$$eval(\sigma, \mathtt{x}\text{-}\mathtt{x}) = \top$$

$$\alpha_b(ceval(\gamma_b(\sigma), \mathtt{x}\text{-}\mathtt{x})) = \mathbf{0}$$

Even if we could make *eval* optimal, the analysis result is not always optimal:

```
x = input;
y = x;
z = x - y;
```

Completeness

$$\alpha(\{\!\![P]\!\!\}) \sqsupseteq [\![P]\!]$$

Sound *and* complete: $\alpha(\{\!\![P]\!\!\}) = [\![P]\!]$

(Intuitively, the analysis result is the most precise possible for the currently used lattice)

Not the same as $\{\!\![P]\!\!\} = \gamma([\![P]\!])$ (called “exact”)

(Intuitively, the analysis result exactly captures the semantics of the program)

Completeness in sign analysis?

$\widehat{*}$ is complete:

$$\alpha_a(D_1) \widehat{*} \alpha_a(D_2) \sqsubseteq \alpha_a(D_1 \cdot D_2)$$

$\widehat{+}$ is *not* complete

$$\alpha_a(D_1) \widehat{+} \alpha_a(D_2) \not\sqsubseteq \alpha_a(D_1 + D_2)$$

Sign analysis is sound and complete for *some* programs,
but not for all programs