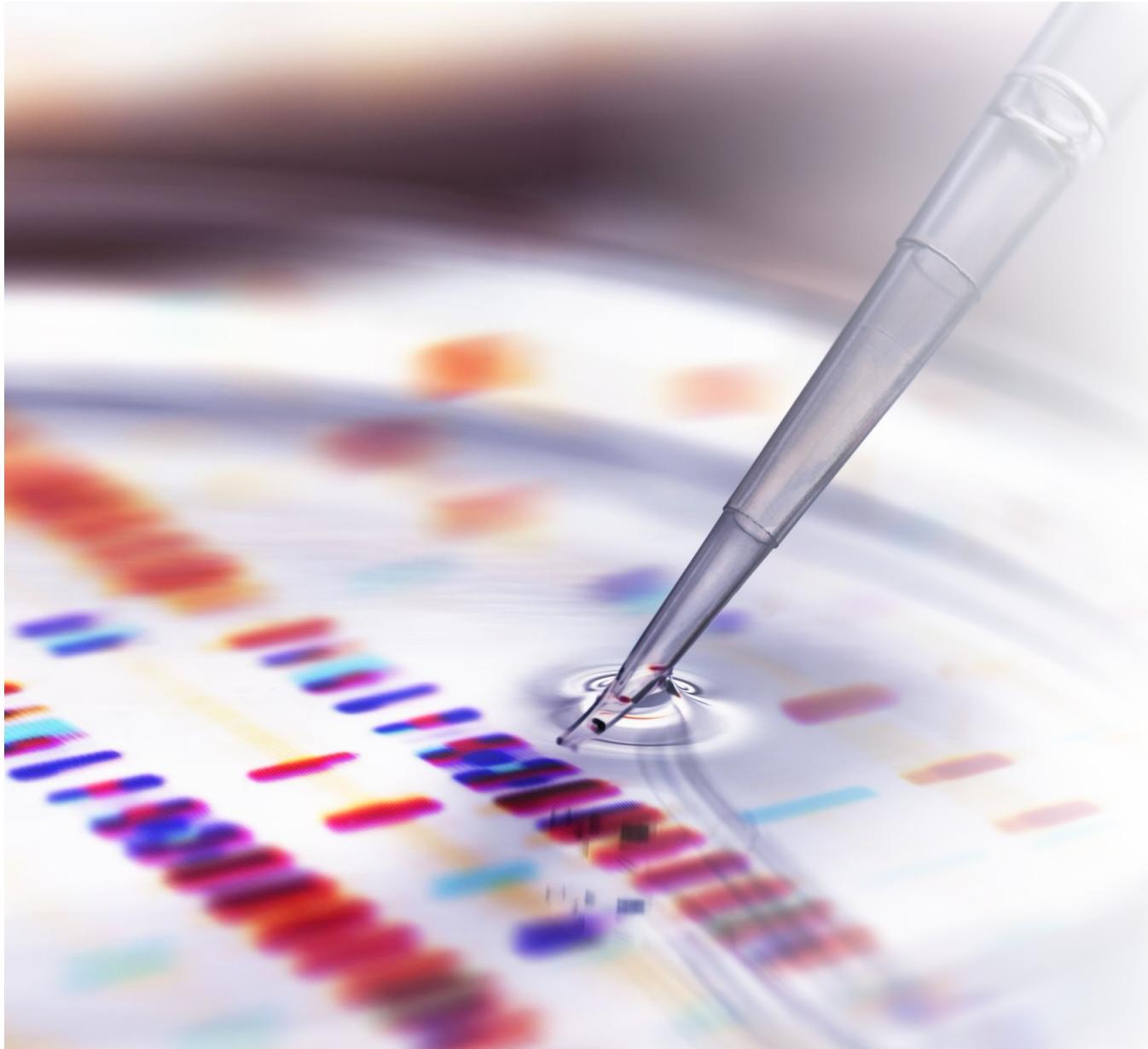




# Static Program Analysis A Primer

---



Where we are  
now?

# Security in Programming Languages (Recap)

## Run-time Support

- Stack Canaries
- Layout randomization
- Shadow stack
- NX memory
- Enclaves
- FAT Pointer & Data Descriptor
- Web Assembly
- IRM and Code Instrumentation
- Dynamic Taint Analysis
- Symbolic Execution (later)

## Static Analysis

- Today

# Dynamic Analysis

---

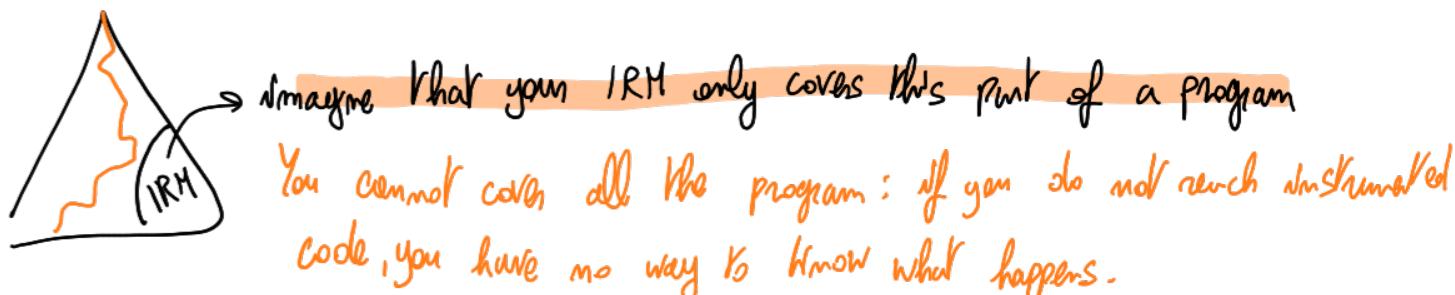
Dynamic analysis refers to the process of analysing a program during its execution, often in a controlled environment such as a sandbox (**WASM**), abstract machine (**dynamic taint analysis**), or instrumented runtime (**inlined reference monitors**). ↳ control flow integrity

---

Dynamic analysis is a powerful technique for handling runtime vulnerabilities (e.g., memory safety, unexpected control flows), it comes with significant limitations.

# Limited Code Coverage

- Dynamic analysis can only observe the parts of the program that are actually executed.
- Implication: If a test input does not trigger a specific code path, any bugs or vulnerabilities in that path will remain undetected.
- Example: A buffer overflow that only occurs when a rare error-handling routine is triggered might never be seen.



# Input Dependence

- **The quality of the analysis is tightly coupled with the quality of the input data.**
- **Implication:** Without comprehensive and targeted input generation (e.g., through fuzzing), dynamic analysis may miss critical execution scenarios.
- **Example:** Fuzzing can generate many inputs, but might not reach deeply nested conditions without guidance or symbolic reasoning.

# Overhead

- Dynamic analysis often incurs high runtime overhead due to instrumentation, taint tracking, or sandboxing.
- Implication: This overhead limits the scalability of the approach and may make it impractical for analyzing large software systems.
- Example: Taint tracking across a large enterprise application can slow down execution.

# Evasion detection

**Sophisticated malware can detect dynamic analysis environments and alter behavior to evade detection.**

- **Implication:** This self-defensive behavior can mislead analysts into believing the code is benign.
- **Example:** Malware detecting virtual machines or debugger presence may disable payload delivery.

# No soundness and completeness

**Dynamic analysis is inherently unsound (not all behaviors are analyzed) and incomplete (some behaviors may be misclassified).**

- **Implication:** It cannot prove the absence of vulnerabilities—only their presence.
- **Example:** A test run doesn't mean the program is secure, just that no issues were observed under the tested conditions.

# Environment dependencies

**Behavior may depend on the runtime environment (e.g., OS version, hardware).**

- **Implication:** Dynamic analysis in one environment may miss vulnerabilities that appear in others.
- **Example:** A memory corruption issue might occur only on 32-bit architectures.

Okay so the problem i guess is: you can make all policies you want that run at execution, but you can't actually know if they cover everything because you would need to test every possible path. Correct?

Exactly! You can define the most rigorous taint tracking policies possible, but the fundamental limitation is that you can only analyze what actually executes. Since you can't run your program with every possible input and execution path in real-world conditions, some paths will inevitably go untested.

## (my) Conclusion

Dynamic analysis is a **valuable tool** in the security toolbox, especially for handling bugs that require real execution to detect.

However, because of its **limitations** it must be complemented with other techniques like **static analysis**, **formal verification**, and **secure coding practices** to form a comprehensive security strategy.

# Static Program Analysis

---

The goal of static program analysis is to predict the dynamic behavior of programs without executing them.

---

Abstractly checks the entire possible state space of the program.

# Why we need Program Analysers?

- **Program Reliability**
  - Null pointer dereference, memory leak, ...
- **Program Security**
  - Private information leak, injection attack, ...
- **Compiler Optimization**
  - Dead code elimination, ....
- **Secure compilations**
  - translate source programs in a way that preserves source security properties (e.g., memory safety) in the target program produced by the compiler.

# Interesting questions about security of programs

- Can sensitive information leak to non-trusted users?
- Can non-trusted users affect sensitive information?
- Are buffer-overruns possible?
- What about data races?
- ...





# Program Analysis: Requirements

**Program analysis is the systematic examination of a program to determine its properties.** This requires:

- Precise program representations You have to apply a precise analysis, so you need \*
- Tractable, systematic ways to reason over those representations. \$,
- To unambiguously define the meaning of programs and programming languages.
- To use, build, and extend tools that do the above, automatically

\* To have a precise representation

\$, Efficient ways to reason: over them we build tools!

### 3. Unambiguously Define the Meaning of Programs and Programming Languages

- To analyze a program, we must first define what it is supposed to do in a way that leaves no room for misinterpretation.  
(like OPERATIONAL SEMANTICS)
- This is done through formal semantics, which define how each statement in a programming language behaves.
- If a language or program had an ambiguous meaning, different analyses could produce different results, making it unreliable for things like security verification or optimization.

# Static Analysis

- Static analysers inspect a program  $P$  to reason about its behaviors and to determine whether  $P$  satisfies some properties before running it.
  - Does  $P$  contain any private information leaks?
  - Does  $P$  dereference any null pointers?
  - Are all the cast operations in  $P$  safe?
  - Can  $v_1$  and  $v_2$  in  $P$  point to the same memory location?
  - Will certain assert statements in  $P$  fail?
  - Is this piece of code in  $P$  dead (so that it could be eliminated)?
  - ...



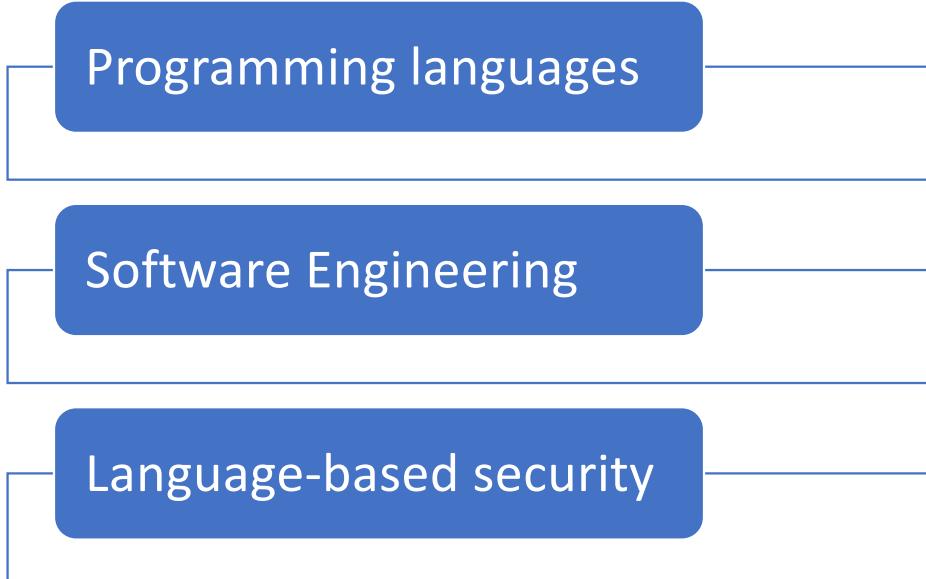
# Static analysis is interesting and useful

---

- Increase efficiency
  - resource usage
  - compiler optimizations
- Ensure correctness
  - verify behavior
  - catch bugs early
- Enable effective SE practices
  - Support program understanding
  - Support re-factorings

# The market of static program analysis

## ACADEMIA

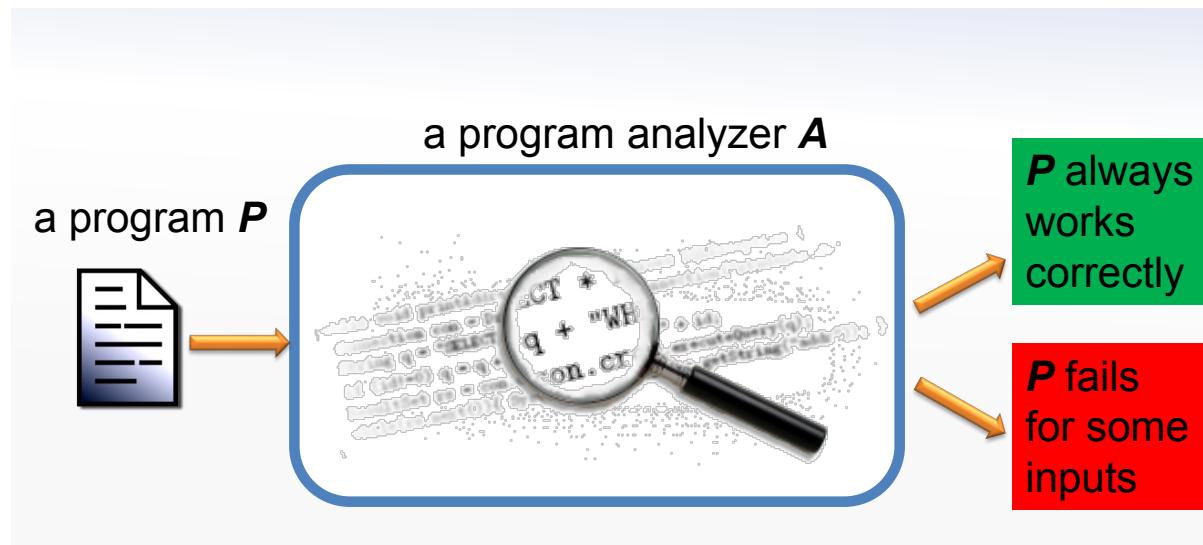


**RESEARCH DIRECTIONS RELYING ON  
PROGRAMS AND PROGRAMMING LANGUAGES**

## INDUSTRIES



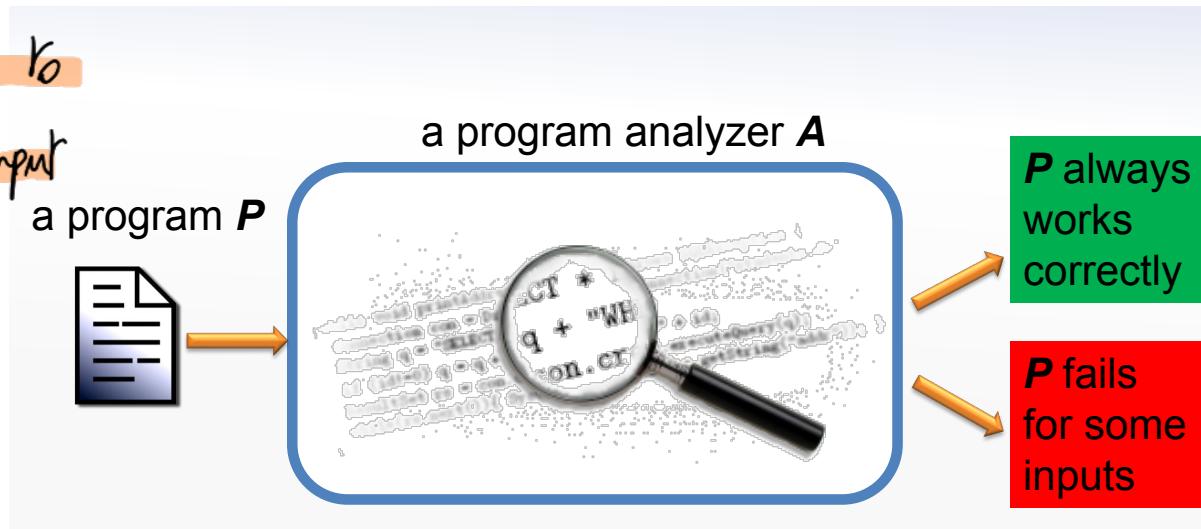
# Static analyzers: Programs that reason about programs



# Static analyzers

## Programs that reason about programs

A static analyzer is  
a program that is able to  
get another program as input  
and return a yes/no  
answer



Rice's Theorem, there does not exist a program capable to determine whether program  $P$  satisfies non-trivial properties of programs, i.e., giving **exact answer**: Yes or No

Saying yes or no for an analyzer is not decidable

# Rice Theorem (1953)

## CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS<sup>(1)</sup>

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]<sup>(2)</sup>, and with ideas which are well summarized in the first sections of a paper of Post [7].

### I. FUNDAMENTAL DEFINITIONS

2. **Partial recursive functions.** We shall characterize recursively enumer-

Rice's theorem states that all non-trivial semantic properties of programs are undecidable

A property is *non-trivial* if it is neither true for every program, nor false for every program..

Semantic property: property that depends on the behavior of the program  
(depends on behavior and not syntax).

## Example

- We try to write a program **CheckHello(P)** that takes another program **P** as input and answers:
  - Does program **P** ever print "Hello" when it runs?
- This is a **semantic** property: it's about the output of **P**.
- It's **non-trivial**: some programs do print "Hello"; others don't.
- According to **Rice's Theorem**, there is no general algorithm that can correctly determine this for **all** possible programs.

## Proof by contradiction (outline)

Assume that you have a function that can determine if a program  $p$  has some nontrivial property (like `divides_by_zero`):

We check whether or not  $p$  divides by 0.

```
↑ Produces an int    → Takes another program as parameter
1. int silly(program p, input i) {
2.     p(i); → silly calls p with i as input
3.     return 5/0; → Then we return a division by 0.
4. }
5. bool halts(program p, input i) {
6.     return divides_by_zero(`silly(p,i)`);
7. }
```

Imagine we are capable to write `halts` (or analyze, what values the inputs, and calls the program).

- Return `S/I` is executed if  $p$  applied to  $i$  terminates. So if  $P(i)$  terminates, `divides_by_0` says `Y`. If  $p$  doesn't terminate, we say `N`. We solve halting prob.

## Proof by contradiction (outline)

Assume that you have a function that can determine if a program  $p$  has some nontrivial property (like `divides_by_zero`):

```
1. int silly(program p, input i) {  
2.     p(i);  
3.     return 5/0;  
4. }  
5. bool halts(program p, input i) {  
6.     return divides_by_zero(`silly(p,i)`);  
7. }
```

*divides\_by\_zero* return true if program  $p(i)$  terminates and return false if  $p(i)$  does not terminate ... which is undecidable (Halting Problem)!

# Rice Theorem implications

---

Does P contain any private information leaks?

---

Does P dereference any null pointers?

---

Are all the cast operations in P safe?

---

Can v1 and v2 in P point to the same memory location?

---

Will certain assert statements in P fail?

---

Is this piece of code in P dead (so that it could be eliminated)?

...

So, all of these properties are non-trivial. No program capable of giving yes-no answer.

# NON TRIVIAL PROPERTIES

---

Does P contain any private information leaks?

---

Does P dereference any null pointers?

---

Are all the cast operations in P safe?

---

Can v1 and v2 in P point to the same memory location?

---

Will certain assert statements in P fail?

---

Is this piece of code in P dead (so that it could be eliminated)?

...

**Theorem (Full Employment Theorem for Compiler Writers):** there cannot exist a compiler that fully optimizes all programs in a Turing-complete programming language.

*Proof.* Suppose for the sake of contradiction that there exists a fully optimizing compiler for a Turing-complete programming language, say Java. Consider a Java method **m** that performs arbitrary computation.

If **m** goes into an infinite loop, then such a fully optimizing compiler would optimize **m** to the follow minimized form:

```
public static void m() {  
    while(true) {}  
}
```

Such a compiler could then be used to detect whether **f** goes into an infinite loop: compile the program and inspect the optimized output to see if it is in the minimized form.

Hence, we would be able to solve the halting problem which is undecidable!.



**WELL KNOWN PROOF  
TECHNIQUE**

**REDUCTION TO THE  
HALTING PROBLEM**

## Reduction to the halting problem: another example

Can we decide if a variable has a constant value?

```
program P {  
    x = 20:  
    if (TM(j)()) x = 21;  
}
```

↳ j-th running machine with empty input terminates

Variable x has a constant value if and only if the j-th turing machine does not terminate on empty input

# Perfect static analysis

---

Can determine whether a program P satisfies non-trivial properties by giving ***exact answer***: Yes or No



# Perfect static analysis

---

Can determine whether a program P satisfies non-trivial properties by giving *exact answer*: Yes or No



# Perfect static analysis

---

Can determine whether a program P satisfies non-trivial properties by giving *exact answer*: Yes or No

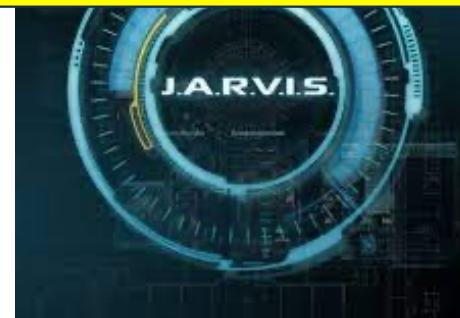


**NO PERFECT STATIC ANALYSIS EXISTS!!!!!!**

# Perfect static analysis

Can determine if a program P satisfies a non-trivial property giving *exact answer* or No

IS THIS THE END OF THE STORY?  
(AND OF THIS COURSE??)



NO PERFECT STATIC ANALYSIS EXISTS!!!!!!

First result we have!

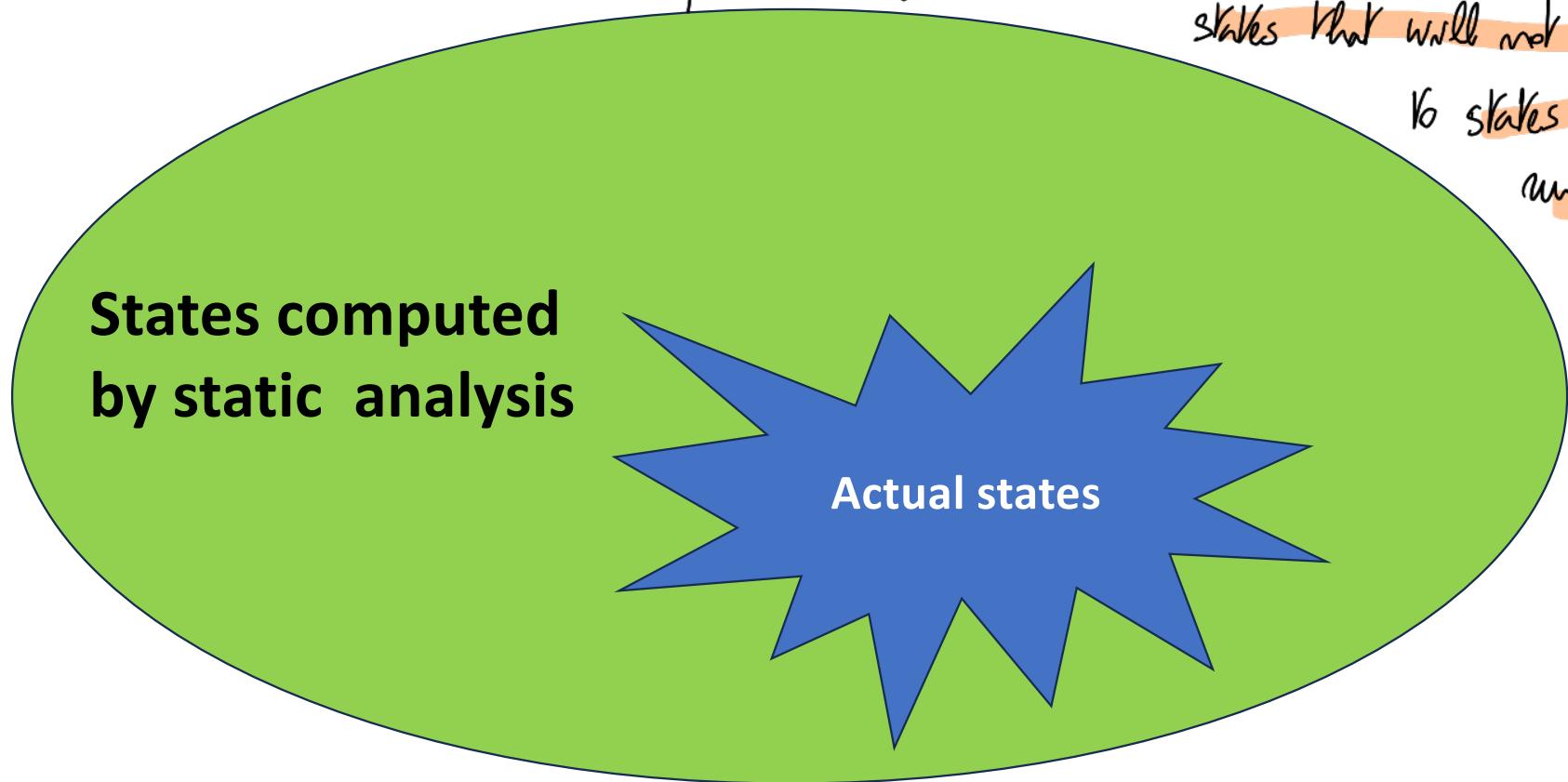
## NO: Approximation

- Instead of attempting to analyze the full behaviour of a program (which is in general undecidable), the idea is to create a more abstract representation, typically finite one, which can be fully analyzed.
- Approximate answers may be decidable!

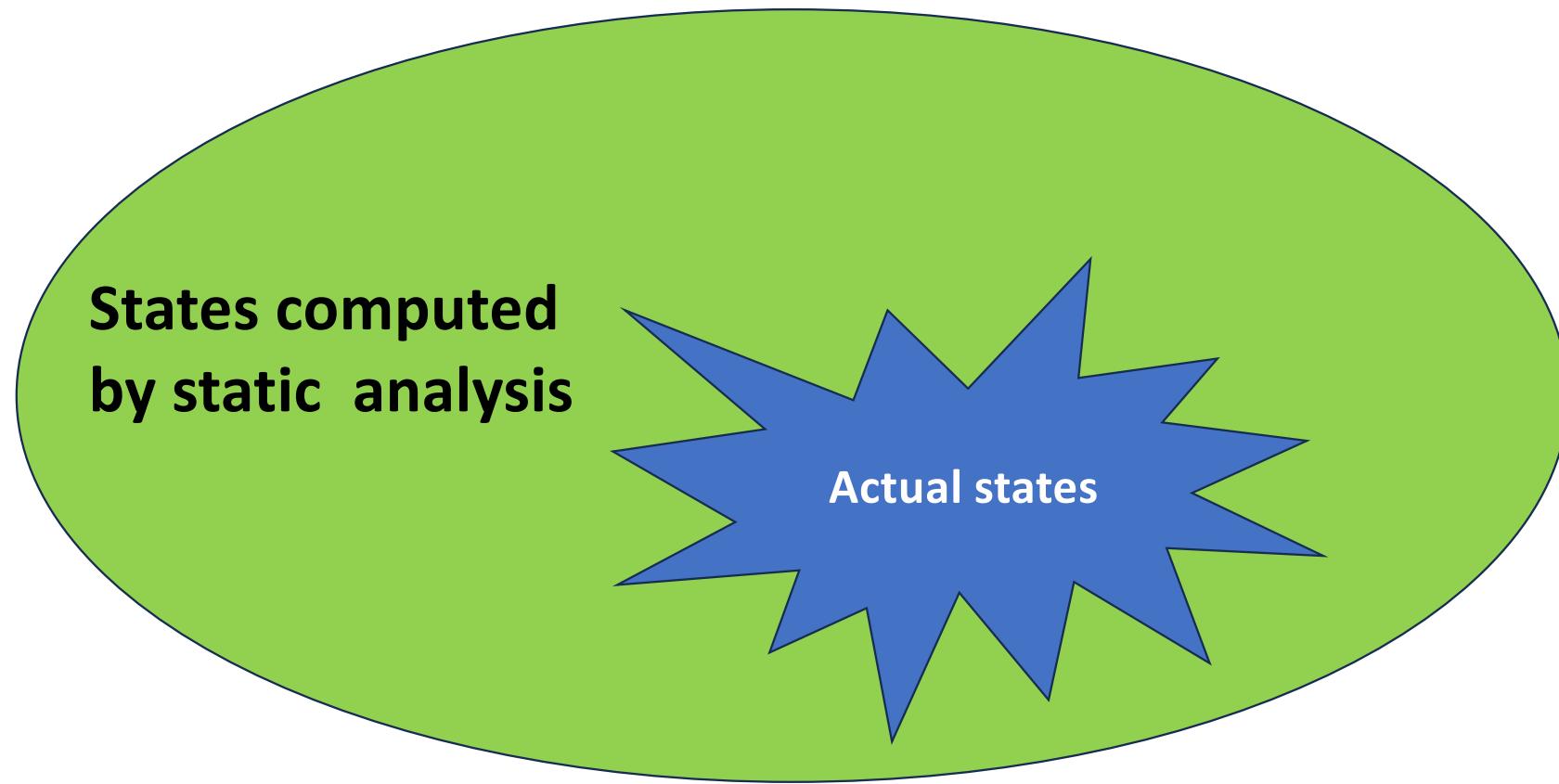


We apply static analysis to an abstract overapproximation of a program.



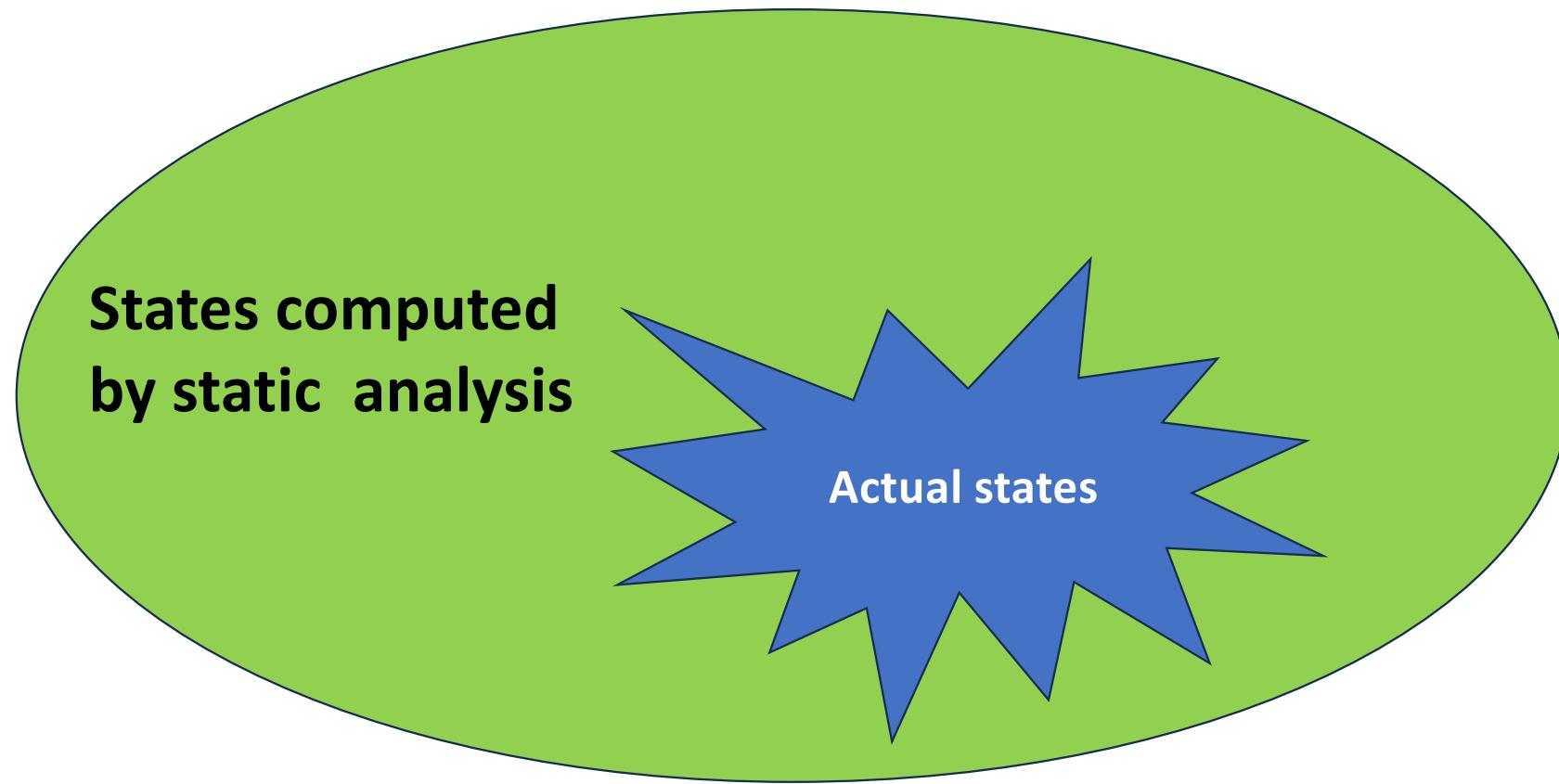


An over approx. takes a much larger number of states  
and ensure that the actual states are a subset  
of the ones you consider - You will also have  
states that will not correspond  
to states that appear at  
run time.



**Over-Approximation of the exact behaviour of the program (the actual states)**

Exact behaviour of the program is a subset of the states offered by the over approx.



Over-Approximation of the exact behaviour of the program (the actual states)

The states not included in the analysis results **will** never happen

States that do not appear in approx. will not happen!

# False positive – false negative

---

**Abstracting over program states avoid the dilemma of Rice's theorem.**

The answers given by static analyses will include **false positives** (programs that the analysis concludes have a certain property while in fact they don't) as well as **false negatives** (programs that we fail to detect have a given property).

①

We aim to have static analyses that are **conservative**, i.e., either they have no **false positives** (**soundness**) or no **false negatives** (**completeness**) depending on the particular problem.

↑ Programs in the given but not actual states \*

\* States enjoy certain properties but won't happen.

① Program does not pass even though they have property.

# SUMMARY

	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive <i>We found a non existing error</i>
No Error Reported	False negative <i>We didn't find the error</i>	True negative (correct analysis result)



So our static analysis, even on the overapproximated set of states, is not able to give a 100% correct answer related to a property otherwise we would only have false positives. Correct?

Exactly! If static analysis were to provide a 100% correct answer about a property (such as correctness or security), **it would have to analyze all possible execution paths with perfect precision**. This would result in **no false negatives** (i.e., every actual issue would be caught) but would almost certainly lead to **false positives** as well, because it would need to consider every possible state—even those that the program would never actually reach.

Here's the key point:

- **False positives** occur when the analysis flags something that can't actually happen during execution, which happens when we try to overapproximate all possible states.
- **False negatives** occur when real issues are missed, which can happen if we under-approximate or don't explore enough states.

# Examples

## Code liveness (aka please compiler do not remove it)

The static analysis decides if a given function is ever called at runtime:

- if “no”, remove the function from the code
- if “yes”, don’t do anything
- but the “no” answer *must always be correct if given*

False positives are ok (dead code left in)

False negatives are not ok (removing live code)

# Examples

## Downcast analysis

The static analysis decides if (A) x will always succeed (aka please compiler do not generate run time checks)

- if “yes”, don’t generate a runtime check
- if “no”, generate run time code for the cast
- the “yes” answer must always be correct if given

False negatives are ok (check safe casts)

False positives are not (skip test of unsafe cast)

# The software engineering challenge

---

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

Challenge: using static analyses as a security tool.



# Static Analysis: a constraint-based approach

The basic idea is to **separate the analysis specification from algorithmic aspects and implementation details**

```
#include <iostream>
using namespace std;

void bubblesort(int a[]) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < (5 - i - 1); j++) {
            if (a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}

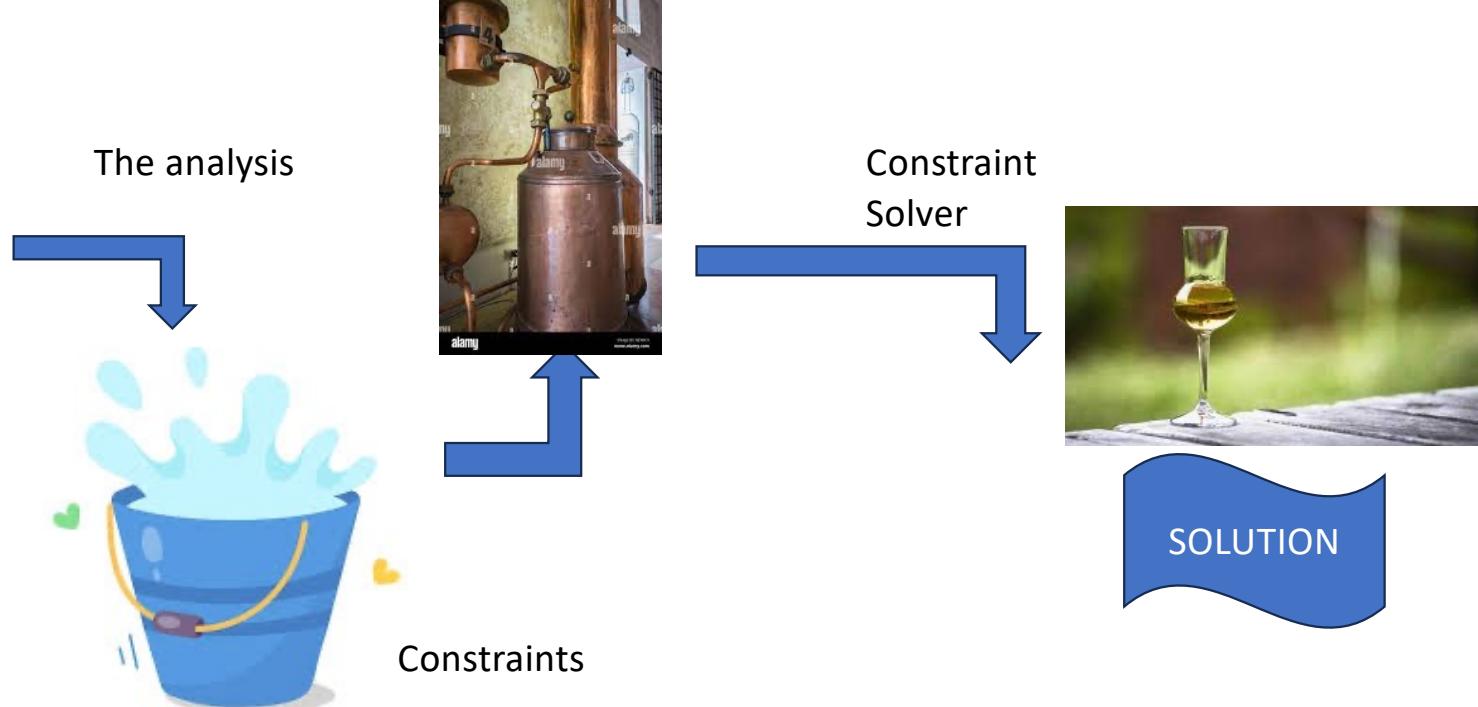
int main() {
    int myarray[5];
    cout << "Enter 5 integers in any order: " << endl;
    for (int i = 0; i < 5; i++) {
        cin >> myarray[i];
    }
    cout << "Before Sorting: " << endl;
    for (int i = 0; i < 5; i++) {
        cout << myarray[i] << " ";
    }

    bubblesort(myarray); // sorting

    cout << endl << "After Sorting: " << endl;
    for (int i = 0; i < 5; i++) {
        cout << myarray[i] << " ";
    }
}

return 0;
}
```

Program to analyze



Separate generation of constraints from the verification.

Analysis at the beginning takes a basket and puts a lot of constraints. Those are generated by analysis. Then you have a constraint solver, that tries to solve the generated constraints to have a smaller set.

We define constraints associated to each of the points of the programs.

## Where Do the "Legal Check" Rules Come In?

You're asking about something like "check if `array[c]` is legal." This type of check is part of the solver's job because it involves reasoning about whether a constraint (like  $c$  being negative) leads to an **invalid program behavior** (e.g., invalid array indexing).

- In the **analysis phase**, we define **what the constraints are** (e.g.,  $c \in \{\text{Negative, Zero, Positive}\}$ ).
- In the **solver phase**, the solver **takes these constraints** and reasons about whether any of them lead to **illegal behavior**, like array bounds issues. It will **check** if any constraint leads to an invalid program operation.

Okay snap back. So we want to prove that our program satisfies a certain property. The constraint phase defines constraints we have in different parts of the program, then the solver based on the property knows whether certain constraints are compatible with a policy or not, example at instruction `array[c]` the compatible constraint on  $c$  is that  $c$  is non negative. Correct?

Yes, exactly! You've got it. Here's how it breaks down:

1. **The constraint phase** defines constraints that represent the program's behavior, given its abstract state. These constraints describe how variables and their values evolve through the program's instructions.
2. **The solver** takes these constraints and checks whether they are **compatible with the property** you're trying to prove about the program. For example, if you're trying to prove that `array[c]` is always a valid access (i.e., no out-of-bounds access), the solver will check whether the constraints on  $c$  ever violate that property (for instance, checking if  $c$  could be negative if array indexing requires  $c$  to be non-negative).

Is it correct to say that the constraints actually define the abstract state we are considering?

Yes, that's absolutely correct!

The **constraints you define represent the abstract state of the program**. In static analysis, the **abstract state** is the description of what we know about the program's variables and their possible values at a given point in execution, but without having to compute every exact value. Instead, we use **abstract representations** of the values, such as intervals, signs, or other abstract properties.

So, in essence, the **constraints define the abstract state** because they tell us what the possible values of variables are at any given point in the program. These constraints are an over-approximation of the actual values that could occur in the program, and through the constraints, we can reason about properties and detect potential issues.

---

# AN APPETIZERS SIGN ANALYSIS



# STATIC SIGN ANALYSIS

Static Analysis: Determine the sign (+, -, or 0) of all the variables of a given program.

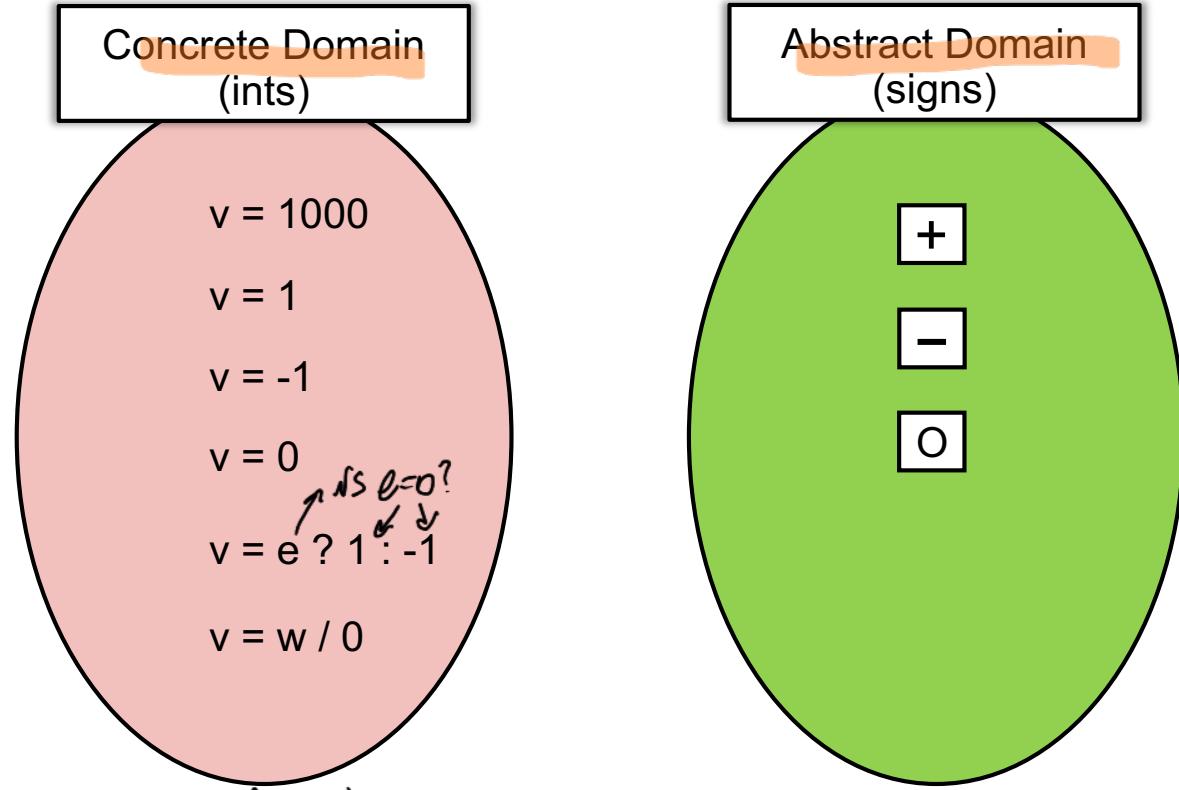


## Applications

To check divided by zero errors

To check negative array indices

## Sign Analysis: Approximation (Abstraction)

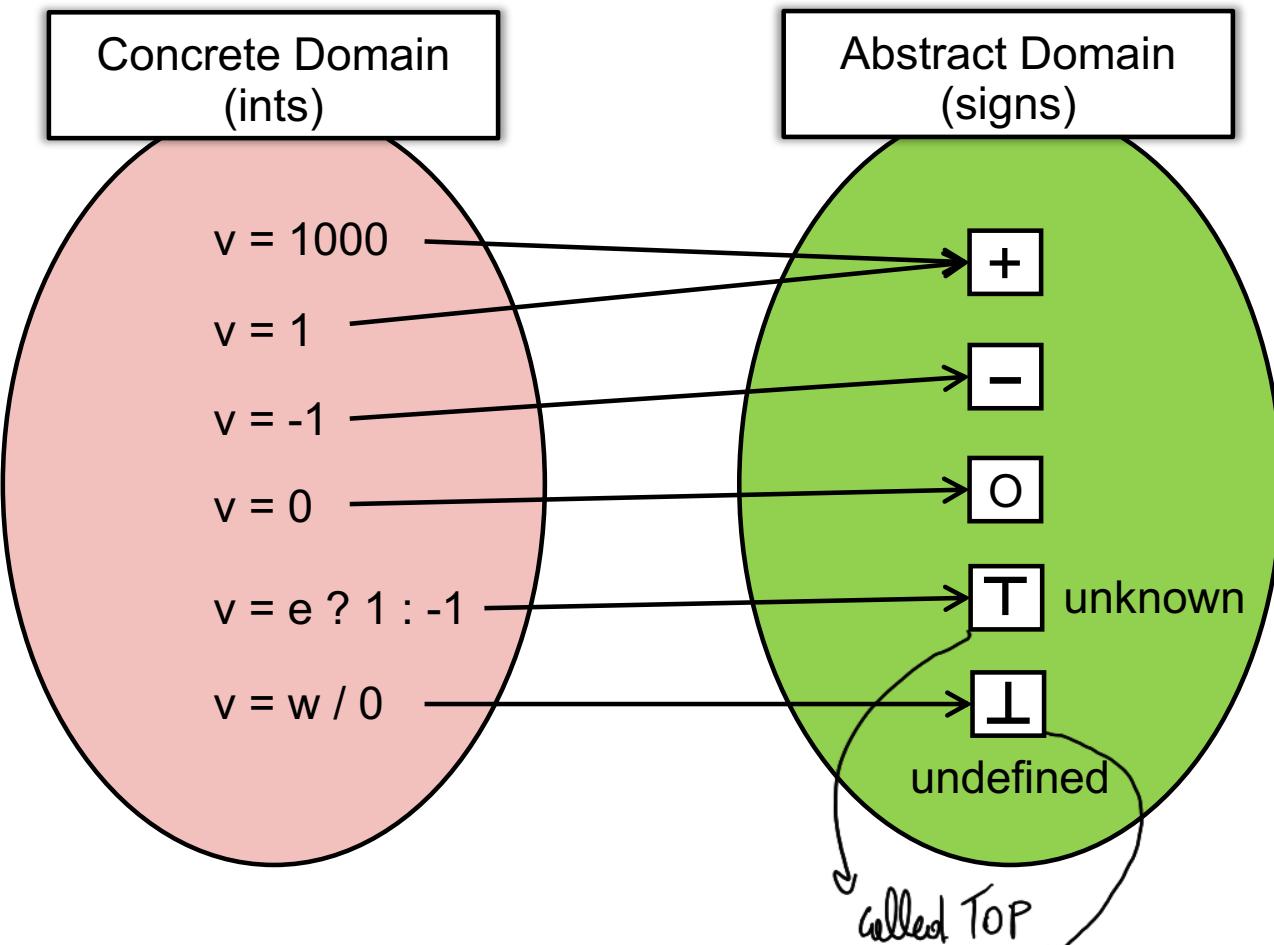


Actual state space



Those are our actual values we have, but in over approximation  
we have abstract states

Sign Analysis:  
better  
abstraction

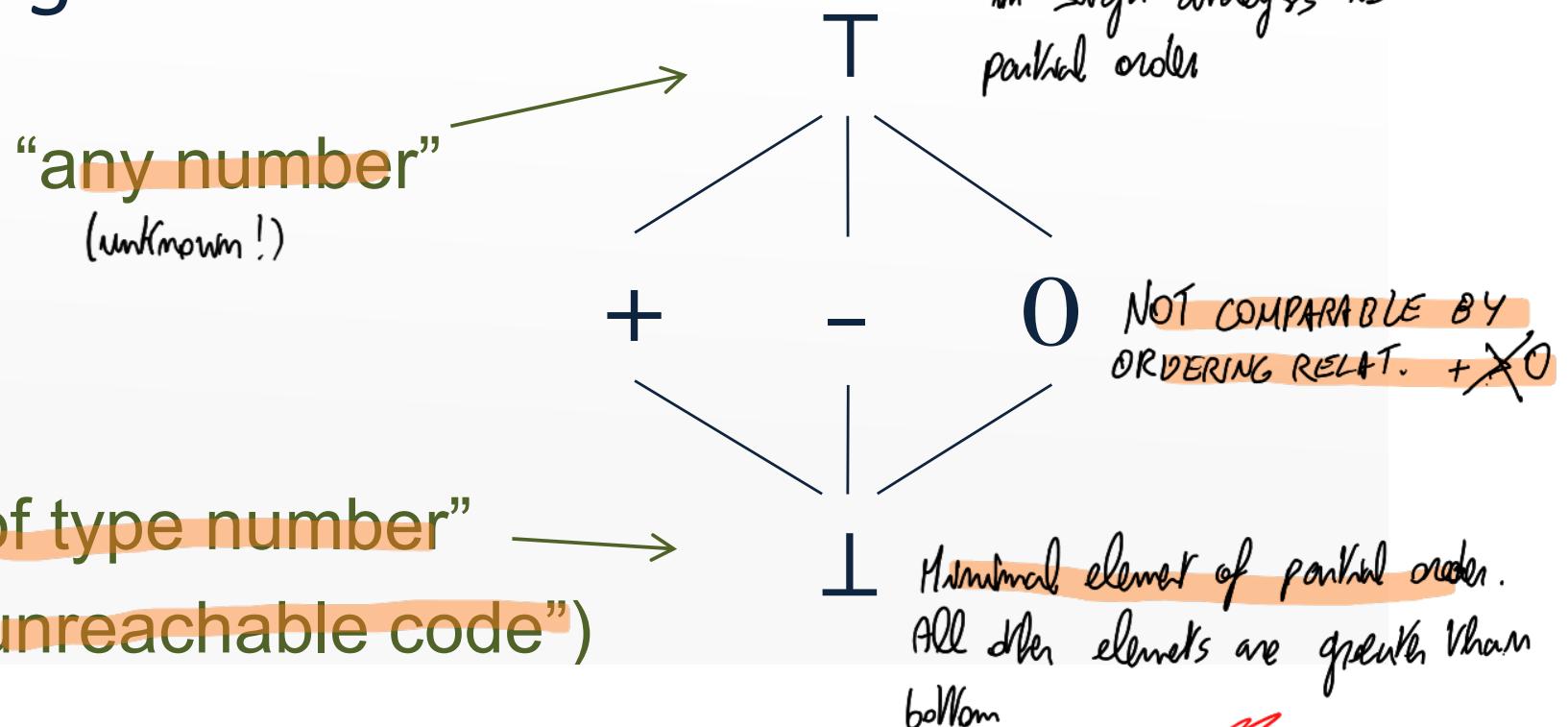


To manage this analysis it's enough to consider  
a set of 5 states

called TOP  
called

## Sign Analysis: the model

### The *Sign* lattice:



Lattice of signs: partial order

Partial ordered set is a set with an ordering relation.

$(S, \leq)$

with reflexive property;  $\forall s \leq s$

$\uparrow s$  less or equal than itself

# Partial orders

- Given a set  $S$ , a partial order  $\sqsubseteq$  is a binary relation on  $S$  that satisfies:

— reflexivity:

$$\forall x \in S: x \sqsubseteq x$$

↳ a binary relation is a subset of the cartesian product  $S \times S$ .

— transitivity:

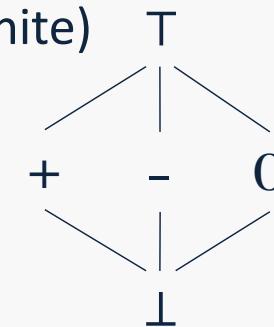
$$\forall x, y, z \in S: x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$$

— anti-symmetry:

$$\forall x, y \in S: x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$$

Saying  $a \leq b$  means  
 $(a, b) \in \text{Relation}$

- Can be illustrated by a Hasse diagram (if finite)



Example:  $\leq, \sqsubseteq$ , divisibility.

- It is a partial order because not every pair of elements in the set  $S$  is necessarily comparable.  
(+, -, 0 in our case).

# Take #1

PROGRAM	CONSTRAINTS
X=10;	X = +
Y = -1;	Y = -
Z = 0:	Z = 0
A = X + Y	A = ???
...	

How do we generate the constraints?

If we say  $X=10$ , the constraint is that  $X$  is positive

We call constraints the way in which we represent a value in our own approximation. At this point, the constraint we apply is that  $x$  is positive,  $y$  is negative etc.

Analysis checks a property. Thus analysis checks this properties.

# Transfer Functions

**Transfer functions** define how to evaluate different program statements on abstract values.

Transfer functions are defined according to “**analysis problem**” and the “**semantics**” of different program statements.

Go back to dynamic analysis: we defined transfer functions to define how a certain variable value has an effect over the values.

In the over approximation, we need to define how our operators affect the abstract values (+, -, T, L)

## Final Connection: Why This Definition?

The slide says:

"Transfer functions define how to evaluate different program statements on abstract values."

This means transfer functions tell us how each program statement (assignments, conditionals, loops) affects our abstract values.

"Transfer functions are defined accordingly to analysis problem and the semantics of different program statements."

This means the way we define these functions depends on:

1. The specific analysis problem (e.g., sign analysis, interval analysis, constant propagation).
2. The semantics of program statements (e.g., what does addition do? what does division do?).

Okay. I have this intuition then. When you say abstract states, you are referring to the states inside our over approximation (we don't care if they are actual reachable states by the program or not). In the example of sign analysis, abstract states are described by a set of abstract values that we have over time for the execution i guess? Imagine we have just variable a,b,c. An example of an abstract state would be something like {a = Positive, b = Negative, c = not a number}

Yes! You've got the right intuition.

When we talk about **abstract states**, we mean states that exist **inside our over-approximation**, meaning they don't necessarily correspond to actual concrete program states but rather to a conservative estimate that ensures we capture all possible behaviors.

### Abstract States in Sign Analysis

In your example, where we track **signs** of variables, an abstract state is a mapping of each variable to its corresponding abstract value.

So a transfer function will tell us how instructions make us move from an abstract state to another. Correct?

Exactly! A transfer function determines how executing a program instruction moves us from **one abstract state to another**.

# Transfer Functions

$$\boxed{+} \textcolor{blue}{+} \boxed{+} =$$

$$\boxed{+} \textcolor{blue}{/} \boxed{+} =$$

$$\boxed{-} \textcolor{blue}{+} \boxed{-} =$$

$$\boxed{-} \textcolor{blue}{/} \boxed{-} =$$

$$\boxed{\circ} \textcolor{blue}{+} \boxed{\circ} =$$

$$\boxed{T} \textcolor{blue}{/} \boxed{\circ} =$$

$$\boxed{+} \textcolor{blue}{+} \boxed{-} =$$

$$\boxed{+} \textcolor{blue}{/} \boxed{-} =$$

# Transfer Functions

$$\boxed{+} + \boxed{+} = \boxed{+}$$

$$\boxed{+} / \boxed{+} = \boxed{+}$$

$$\boxed{-} + \boxed{-} = \boxed{-}$$

$$\boxed{-} / \boxed{-} = \boxed{+}$$

$$\boxed{o} + \boxed{o} = \boxed{o}$$

$$\boxed{T} / \boxed{o} =$$

$$\boxed{+} + \boxed{-} =$$

$$\boxed{+} / \boxed{-} = \boxed{-}$$

# Transfer Functions

$$+ + = +$$

$$+ / + = +$$

$$- + - = -$$

$$- / - = +$$

$$O + O = O$$

$$T / O = \text{NOT A NUMBER}$$

$$+ - = T$$

$$+ / - = -$$

# Transfer Functions

$$+ + = +$$

$$+ / + = +$$

$$- + - = -$$

$$- / - = +$$

$$o + o = o$$

$$T / o = \perp$$

$$+ - = T$$

$$+ / - = -$$



### Transfer Functions

$$\boxed{+} \textcolor{blue}{+} \boxed{+} = \boxed{+}$$

$$\boxed{-} \textcolor{blue}{+} \boxed{-} = \boxed{-}$$

$$\boxed{\circ} \textcolor{blue}{+} \boxed{\circ} = \boxed{\circ}$$

$$\boxed{+} \textcolor{blue}{+} \boxed{-} = \boxed{\top}$$

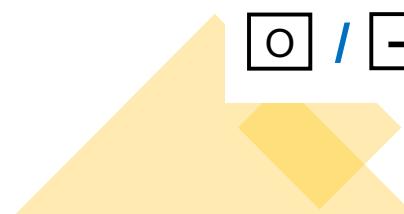
$$\boxed{+} \textcolor{blue}{/} \boxed{+} = \boxed{+}$$

$$\boxed{-} \textcolor{blue}{/} \boxed{-} = \boxed{+}$$

$$\boxed{\top} \textcolor{blue}{/} \boxed{\circ} = \boxed{\perp}$$

$$\boxed{+} \textcolor{blue}{/} \boxed{-} = \boxed{-}$$

$$\boxed{\circ} \textcolor{blue}{/} \boxed{-} = \boxed{\circ}$$



### Programs

```
x = 10;  
y = -1;  
z = 0;  
a = x + y;  
b = z / y;  
c = a / b;  
p = arr[y];  
q = arr[a];
```

### Constraint

```
x =  
y =  
z =  
a =  
b =  
c =  
p =  
q =
```



### Transfer Functions

$$\boxed{+} \textcolor{blue}{+} \boxed{+} = \boxed{+}$$

$$\boxed{-} \textcolor{blue}{+} \boxed{-} = \boxed{-}$$

$$\boxed{O} \textcolor{blue}{+} \boxed{O} = \boxed{O}$$

$$\boxed{+} \textcolor{blue}{+} \boxed{-} = \boxed{T}$$

$$\boxed{+} \textcolor{blue}{/} \boxed{+} = \boxed{+}$$

$$\boxed{-} \textcolor{blue}{/} \boxed{-} = \boxed{+}$$

$$\boxed{T} \textcolor{blue}{/} \boxed{O} = \boxed{\perp}$$

$$\boxed{+} \textcolor{blue}{/} \boxed{-} = \boxed{-}$$

$$\boxed{O} \textcolor{blue}{/} \boxed{-} = \boxed{O}$$

### Programs

```
x = 10;  
y = -1;  
z = 0;  
a = x + y;  
b = z / y;  
c = a / b;  
p = arr[y];  
q = arr[a];
```

### Constraint

```
x = +  
y = -  
z = O  
a =  
b =  
c =  
p =  
q =
```





### Transfer Functions

$$\boxed{+} + \boxed{+} = \boxed{+}$$

$$\boxed{-} + \boxed{-} = \boxed{-}$$

$$\boxed{O} + \boxed{O} = \boxed{O}$$

$$\boxed{+} + \boxed{-} = \boxed{T}$$

$$\boxed{+} / \boxed{+} = \boxed{+}$$

$$\boxed{-} / \boxed{-} = \boxed{+}$$

$$\boxed{T} / \boxed{O} = \boxed{\perp}$$

$$\boxed{+} / \boxed{-} = \boxed{-}$$

$$\boxed{O} / \boxed{-} = \boxed{O}$$

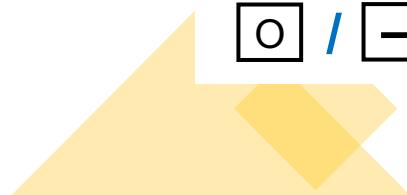
### Programs

```
x = 10;  
y = -1;  
z = 0;  
a = x + y;  
b = z / y;  
c = a / b;  
p = arr[y];  
q = arr[a];
```

### Constraint

$$\begin{aligned}x &= \boxed{+} \\y &= \boxed{-} \\z &= \boxed{O} \\a &= \boxed{T} \\b &= \boxed{O} \\c &= \boxed{\perp}\end{aligned}$$

$$\begin{aligned}p &= \\q &= \end{aligned}$$





### Transfer Functions

$$\boxed{+} + \boxed{+} = \boxed{+}$$

$$\boxed{-} + \boxed{-} = \boxed{-}$$

$$\boxed{O} + \boxed{O} = \boxed{O}$$

$$\boxed{+} + \boxed{-} = \boxed{T}$$

$$\boxed{+} / \boxed{+} = \boxed{+}$$

$$\boxed{-} / \boxed{-} = \boxed{+}$$

$$\boxed{T} / \boxed{O} = \boxed{\perp}$$

$$\boxed{+} / \boxed{-} = \boxed{-}$$

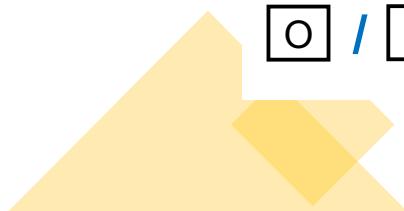
$$\boxed{O} / \boxed{-} = \boxed{O}$$

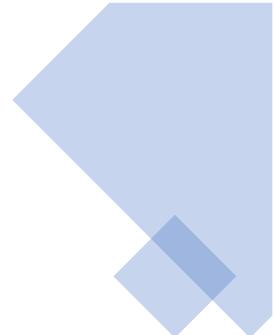
### Programs

```
x = 10;  
y = -1;  
z = 0;  
a = x + y;  
b = z / y;  
c = a / b;  
p = arr[y];  
q = arr[a];
```

### Constraint

x =	$\boxed{+}$
y =	$\boxed{-}$
z =	$\boxed{O}$
a =	$\boxed{T}$
b =	$\boxed{O}$
c =	$\boxed{\perp}$
p =	$\boxed{\perp}$
q =	$\boxed{\perp}$





## Transfer Functions

$$+ + = +$$

$$- - = -$$

$$O O = O$$

$$+ - = T$$

$$+ / + = +$$

$$- / - = +$$

$$T / O = \perp$$

$$+ / - = -$$

$$O / - = O$$

## Programs

```
x = 10;  
y = -1;  
z = 0;  
a = x + y;  
b = z / y;  
1 c = a / b;  
2 p = arr[y];  
3 q = arr[a];
```

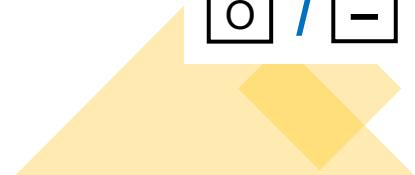
## Constraint

$$\begin{array}{l} x = + \\ y = - \\ z = O \\ a = T \\ b = O \\ c = \perp \\ p = \perp \\ q = \perp \end{array}$$

Divided by zero  
negative array index

1 2

Static analysis is useful



## Transfer Functions

$$+ + = +$$

$$- + = -$$

$$O + O = O$$

$$+ - = \top$$

$$+ / + = +$$

$$- / - = +$$

$$\top / O = \perp$$

$$+ / - = -$$

$$O / - = O$$

## Programs

```
x = 10;
y = -1;
z = 0;
a = x + y;
b = z / y;
1 c = a / b;
2 p = arr[y];
3 q = arr[a];
```

## Constraint

$$\begin{array}{l} x = + \\ y = - \\ z = O \\ a = \top \\ b = O \\ c = \perp \\ p = \perp \\ q = \perp \end{array}$$

Divided by zero  
negative array index

1 2 Static analysis is useful

3 But (over-approximated) static analysis produces false positives

We are over approximating,  
we don't necessarily  
fall out of bounds

Simple static analysis would allow us  
to find division by zero and negative  
array index problems

Okay tell me if this interpretation holds:

if we do:

a = 1

b = 2

Then we are saying "a is constrained to be positive", or "b is constrained to be positive"

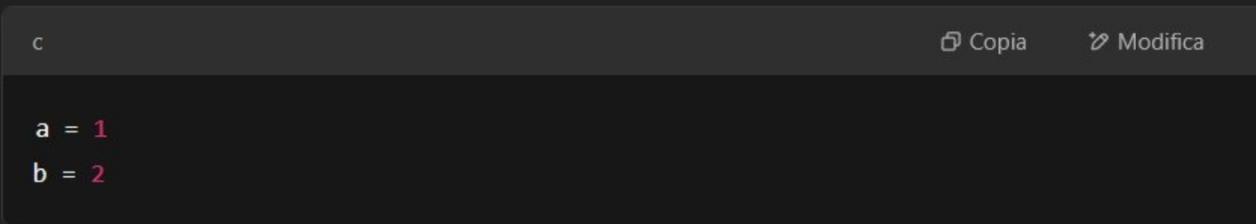
so if we do array[c] and we know by our analysis that "c is constrained to be negative" we can throw a static error?

Yes, your interpretation is correct! Here's how it works:

## Step-by-Step Breakdown

### 1. Assigning Values:

- When you do:



A screenshot of a code editor interface. At the top, there is a dark header bar with the letter 'c' on the left and two buttons on the right labeled 'Copia' and 'Modifica'. Below the header, the main code area contains the following code:

```
a = 1  
b = 2
```

You're effectively **constraining** *a* and *b* to be positive based on your analysis. This is a **constraint** because you're not just saying *a* and *b* are positive; you're stating that, based on the program's behavior, their values must **satisfy** that condition.

# Outlining the analysis

Let's say we try to derive the constraints for a program.

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

Derive a system of equations (equality constraints) with one constraint variable for each program variable and each program point

PROGRAM POINT	Constraint variable	Constraint variable
1	$a_1 = T$	$b_1 = T$
2	$a_2 = +$	$b_2 = b_1$
3	$a_3 = a_2$	$b_3 = a_2 + T$
4	$a_4 = a_3 - b_3$	$b_4 = b_3$

] No info about value

$a_s = \text{constraint over a at point } s.$

## Example 2 (cont)

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

PROGRAM POINT	Constraint variable	Constraint variable
1	$a_1 = T$	$b_1 = T$
2	$a_2 = +$	$b_2 = T$
3	$a_3 = a_2$	$b_3 = a_2 + T$
4	$a_4 = a_3 - b_3$	$b_4 = b_3$

SOLVING THE EQUATIONS



$$\begin{aligned}
 X_1 &: X_1[a \rightarrow T, b \rightarrow T] \\
 X_2 &: X_2[a \rightarrow +] \\
 X_3 &: X_3[b \rightarrow X_2(a) + T] = X_3[b \rightarrow +] \\
 X_4 &: X_4[a \rightarrow X_3(a) - X_3(b)] = X_4[a \rightarrow T]
 \end{aligned}$$

PROGRAM POINT	Constraint variable	Constraint variable
1	$a_1 = T$	$b_1 = T$
2	$a_2 = +$	$b_2 = T$
3	$a_3 = +$	$b_3 = T$
4	$a_4 = T$	$b_4 = T$

# Alternative formulations

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

Constraint variable are functions providing the sign value of program variable for each program point

PROGRAM POINTS	CONSTRAINT VARIABLES
$x_1$	$[a \rightarrow \top, b \rightarrow \top]$
$x_2$	$x_1[a \rightarrow +]$
$x_3$	$x_2[b \rightarrow x_2(a) + \top]$
$x_4$	$x_3[a \rightarrow x_3(a) - x_3(b)]$

DEFINITION OF EQUATIONS

SOLVING THE EQUATIONS

PROGRAM POINTS	CONSTRAINT VARIABLES
$x_1$	$[a \rightarrow \top, b \rightarrow \top]$
$x_2$	$x_1[a \rightarrow +]$
$x_3$	$x_2[b \rightarrow \top]$
$x_4$	$x_3[a \rightarrow x_3(a) - x_3(b)] = x_3[a \rightarrow \top]$

SOLVING THE EQUATIONS

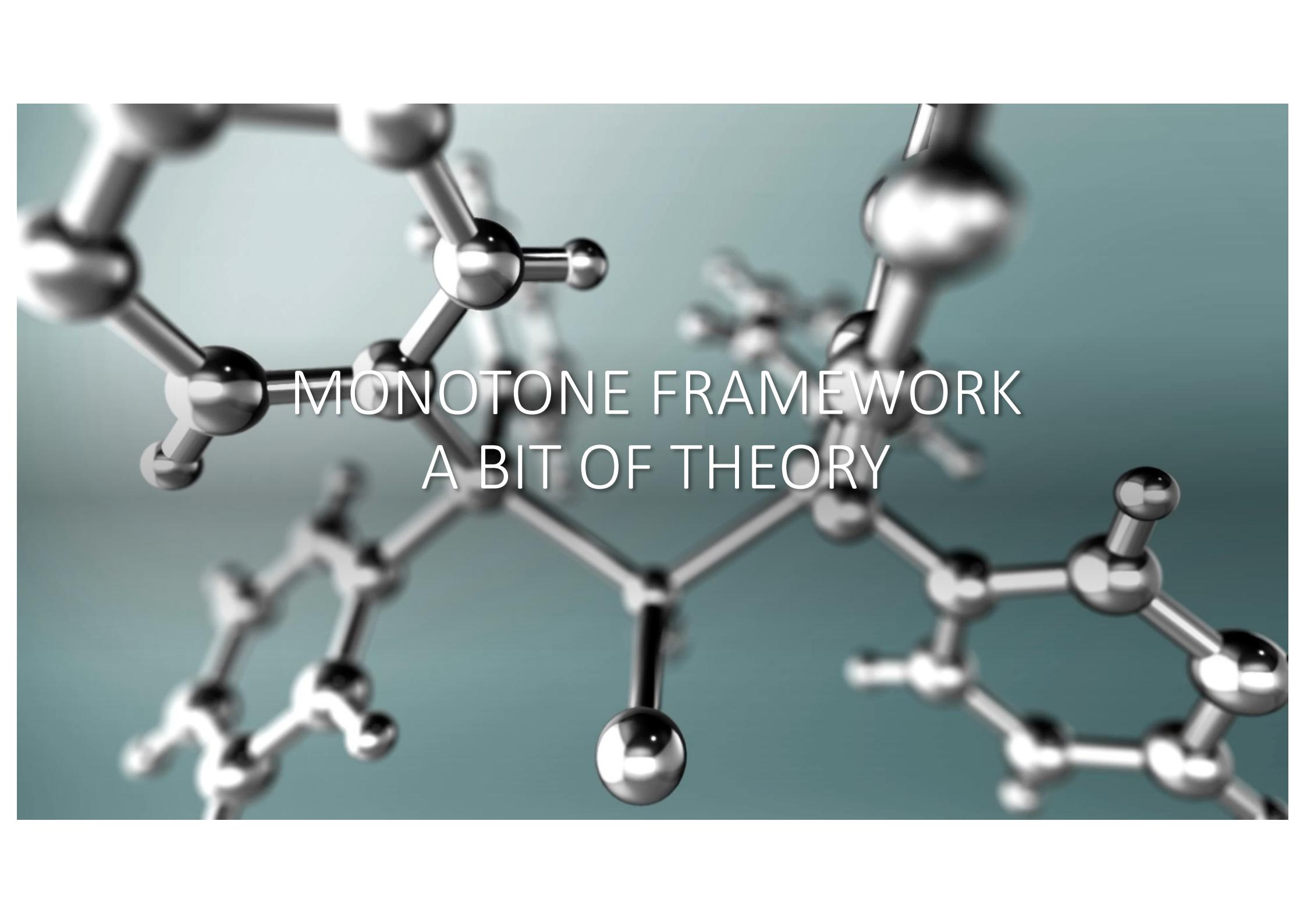
$X_s$  is the constraint variable that represents a mapping of variables to signs.

$X_s[\dots] = \text{Mapping of } X_i \text{ composed with the } [\dots] \text{ mapping}$

# Go inside the sign analysis

States of the approximations are elements of the map lattice  $\text{Vars} \rightarrow \text{Sign}$  where  $\text{Vars}$  is the set of variables in the program

States of the approximations are functions giving the sign values of program variables



# MONOTONE FRAMEWORK A BIT OF THEORY

# Partial orders

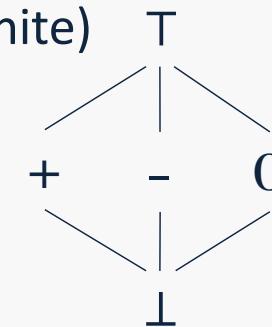
- Given a set  $S$ , a partial order  $\sqsubseteq$  is a binary relation on  $S$  that satisfies:

– 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

$$\forall x \in X: y \sqsubseteq x$$

UNIQUE

- 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

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

} Minimal  
of all the possible  
upper bounds

This property describes the **least upper bound** (supremum) of a subset  $X$  within a partially ordered set  $(S, \sqsubseteq)$ .

Breaking it down:

1.  $X \sqsubseteq \sqcup X$ :

This means that  $\sqcup X$  (the least upper bound of  $X$ ) is an **upper bound** of  $X$ . That is, every element of  $X$  is below  $\sqcup X$  in the partial order.

2.  $\forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$ :

This ensures that  $\sqcup X$  is the **least** upper bound. It says that if  $y$  is any upper bound of  $X$  (meaning every element of  $X$  is below  $y$ ), then  $\sqcup X$  must also be below  $y$ . This makes  $\sqcup X$  the smallest element among all upper bounds.

**Intuition:**

- The least upper bound (also called **supremum** or **join**) is the **smallest element that is greater than or equal to every element in  $X$** .
- It's the "tightest" upper bound: no other upper bound is strictly smaller than it.

Yes! This defines the **greatest lower bound** (GLB), also called the **infimum** ( $\sqcup X$ ). Let's break it down:

1.  $\sqcup X \sqsubseteq X$ :

This means that  $\sqcup X$  is a **lower bound** of  $X$ , meaning that it is **less than or equal to every element in  $X$** .

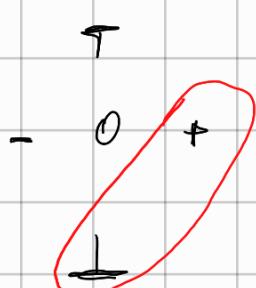
2.  $\forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcup X$ :

This ensures that  $\sqcup X$  is the **greatest** lower bound. It says that if  $y$  is any lower bound of  $X$  (i.e.,  $y$  is less than or equal to all elements in  $X$ ), then  $y$  must also be less than or equal to  $\sqcup X$ . This makes  $\sqcup X$  the **largest element among all lower bounds**.

**Intuition:**

- The **greatest lower bound** (or infimum) is the **largest element that is less than or equal to every element in  $X$** .
- It's the "tightest" lower bound: no other lower bound is strictly greater than it.

Ex. of non comparability:



Minus and zero are NOT least upper bounds because they are not comparable to +1, +2, etc.

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

Taking  $K$  elements we are always able to find the least upper bound and greatest lower bound

A complete lattice must have

- a **unique largest element**,  $\top = \sqcup S$
- a **unique smallest element**,  $\perp = \sqcap S$

A finite lattice is complete if  $\top$  and  $\perp$  exist

I'm glad you're enjoying this! 😊

This definition says that a **lattice** is a special kind of **partially ordered set (poset)** where every pair of elements has both a **least upper bound** (supremum,  $\sqcup$ ) and a **greatest lower bound** (infimum,  $\sqcap$ ).

### Breaking it Down:

- The notation  $x \sqcup y$  (or  $x \vee y$  in some texts) represents the **least upper bound (supremum)** of the two elements  $x$  and  $y$ , meaning the **smallest element in  $S$**  that is greater than or equal to both  $x$  and  $y$ .
- The notation  $x \sqcap y$  (or  $x \wedge y$ ) represents the **greatest lower bound (infimum)** of  $x$  and  $y$ , meaning the **largest element in  $S$**  that is less than or equal to both  $x$  and  $y$ .
- The definition says that for any two elements  $x, y \in S$ , both  $x \sqcup y$  and  $x \sqcap y$  must exist.

### Intuition:

- A lattice is a poset that is "rich enough" to always have a supremum and infimum for every pair of elements.
- If you take any two elements in a lattice, you can always find their "**least common upper element**" and "**greatest common lower element**."

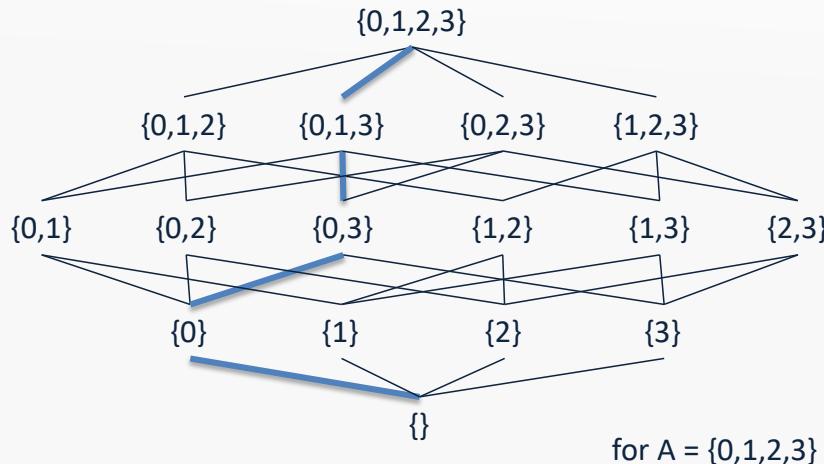
## Example 2: Power Set of $\{a, b\}$ , Ordered by Subset Inclusion

Consider  $S = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ , ordered by **subset inclusion** ( $\subseteq$ ):

- $x \sqcup y = \text{union } x \cup y$ .
- $x \sqcap y = \text{intersection } x \cap y$ .
- Since unions and intersections always exist, this is a **lattice**.

# 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  $|A|$



# Map Lattice

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

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

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

## What's Going On Here?

We are constructing a **new complete lattice** from an existing **complete lattice  $L$**  by considering functions from a set  $A$  into  $L$ .

This new lattice is called a **map lattice**, denoted as:

$$A \rightarrow L$$

This represents the set of all functions from  $A$  to  $L$ , meaning that each element in  $A$  is assigned an element of  $L$ .

In set notation:

$$A \rightarrow L = \{f : A \rightarrow L\}$$

or, using a more explicit mapping notation:

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

Each function (or map) assigns a value from  $L$  to each element of  $A$ .

### Ordering in the Map Lattice (Pointwise Order)

The functions are ordered **pointwise**.

This means that if we have two functions  $f, g \in A \rightarrow L$ , we say:

We pickup ordering relations  
over  $L$  to define order  
over functions

$$f \sqsubseteq g \quad \text{if and only if} \quad \forall a \in A, \quad f(a) \sqsubseteq g(a) \text{ in } L.$$

So one function is "less than or equal to" another if it assigns smaller (or equal) values in  $L$  at every point of  $A$ .

### Supremum ( $\sqcup$ ) and Infimum ( $\sqcap$ ) Computed Pointwise

Since  $L$  is a lattice, it has a least upper bound ( $\sqcup$ ) and a greatest lower bound ( $\sqcap$ ).

To compute  $\sqcup$  and  $\sqcap$  in  $A \rightarrow L$ , we just apply them **pointwise**:

- **Supremum ( $\sqcup$ )**: Given functions  $f, g$ , their supremum is the function that takes the supremum at each input:

$$(f \sqcup g)(a) = f(a) \sqcup g(a) \quad \text{for all } a \in A.$$

- **Infimum ( $\sqcap$ )**: Similarly, we compute the infimum at each point:

$$(f \sqcap g)(a) = f(a) \sqcap g(a) \quad \text{for all } a \in A.$$

This works because we are just lifting the existing structure of  $L$  to functions.



PANKO BENEFITS

## Question 2: How Do You Compute the Smallest and Largest Element in a Complete Lattice?

In a **complete lattice**, we always have:

- A **smallest (bottom) element**  $\perp$ , called the **infimum of the whole set  $S$** .
- A **largest (top) element**  $\top$ , called the **supremum of the whole set  $S$** .

### Smallest Element (Bottom)

$$\perp = \bigcap S$$

This is the **greatest lower bound** of all elements in the lattice.

- It is **less than or equal to every element in  $S$** , meaning:

$$\forall x \in S, \quad \perp \sqsubseteq x.$$

- In powersets (like  $\mathcal{P}(X)$ ),  $\perp$  is the **empty set  $\emptyset$** .
- In function lattices  $A \rightarrow L$ ,  $\perp$  is the **constant function that maps everything to the bottom of  $L$** :

$$\perp(a) = \perp_L \quad \forall a \in A.$$

### Largest Element (Top)

$$\top = \bigcup S$$

This is the **least upper bound** of all elements in the lattice.

- It is **greater than or equal to every element in  $S$** , meaning:

$$\forall x \in S, \quad x \sqsubseteq \top.$$

- In powersets,  $\top$  is the **full set  $X$** .
- In function lattices  $A \rightarrow L$ ,  $\top$  is the **constant function that maps everything to the top of  $L$** :

$$\top(a) = \top_L \quad \forall a \in A.$$

### Step 3: Hasse Diagram for $A \rightarrow L$

Let's take an example where  $L$  has height 3 (meaning it has a longest chain with 3 steps).

If  $A = \{a_1, a_2\}$ , then the elements of  $A \rightarrow L$  are pairs:

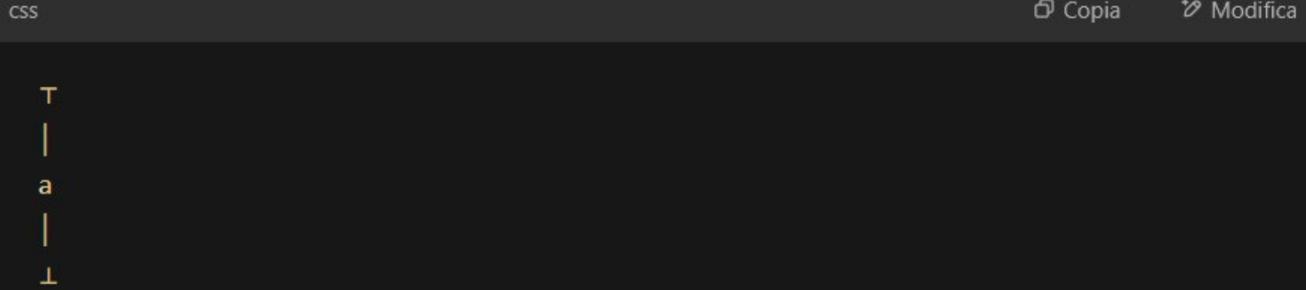
$$(x_1, x_2) \quad \text{where } x_1, x_2 \in L.$$

- The smallest element is  $(\perp_L, \perp_L)$ .
- The largest element is  $(\top_L, \top_L)$ .
- Each coordinate can increase independently, meaning we can form a longest chain where:
  1. We increase  $x_1$  fully first (moving from bottom to top in the first coordinate).
  2. Then we increase  $x_2$  fully (moving from bottom to top in the second coordinate).

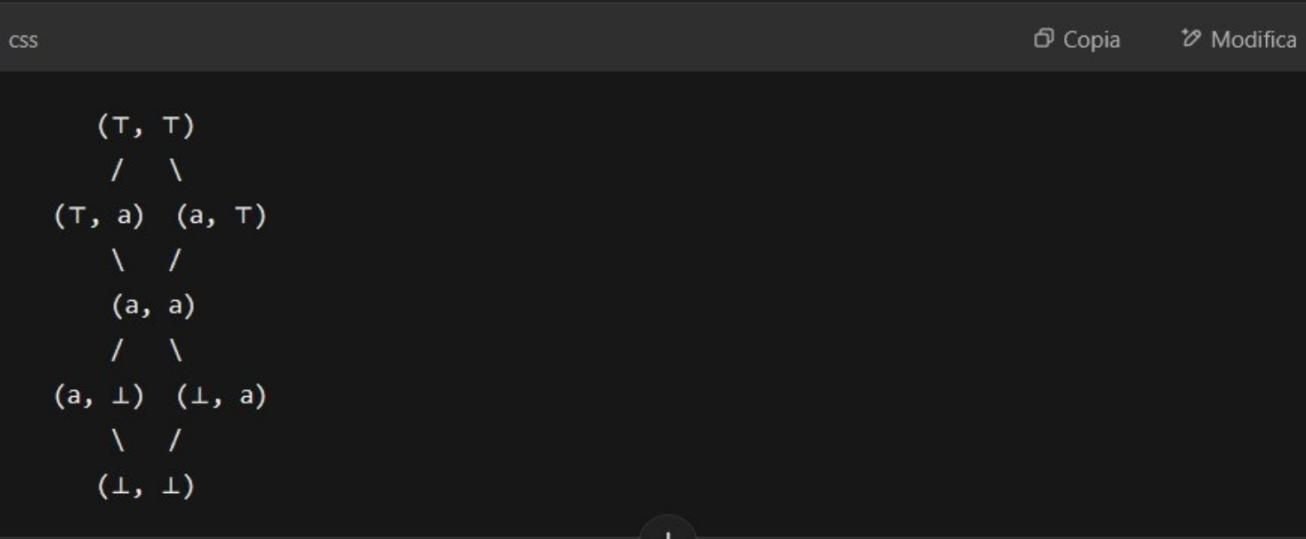
If the longest chain in  $L$  has height  $h$ , then the longest possible sequence in  $A \rightarrow L$  has:

- $h$  steps for the first coordinate.
- Another  $h$  steps for the second coordinate.

Imagine that  $L$  is this simple lattice (height 2):



For  $A = \{a_1, a_2\}$ , we now have pairs  $(x_1, x_2)$ , and the Hasse diagram looks like:



Here, the longest chain is of length  $2 + 2 = 4$ , which is exactly  $|A| \times \text{height of } L$ .

## Final Summary

1. Each function  $f : A \rightarrow L$  is like a tuple  $(x_1, x_2, \dots, x_n)$  where each  $x_i$  moves in its own copy of  $L$ .
2. Since each coordinate moves independently, the longest increasing chain comes from increasing each coordinate one by one.
3. If  $L$  has height  $h$ , then increasing  $|A|$  independent coordinates gives a total height of  $|A| \times h$ .

# Sign Analysis Revisited: Abstract states

Let  $i$  be a program point, the constraint variable  $X_i$  denotes the function (often called map) that gives the sign value of all the program variables after the program point  $i$

## Abstract States: Map lattice

$States = Vars \rightarrow Sign$

$X_i \in States$

States are functions! Our Abstract states define a map lattice over the full lattice  $Sign$

## Step 1: What's Happening Here?

We're analyzing a program **statically**, meaning before actually running it. **Sign analysis** tracks whether variables are **positive, negative, or zero** throughout the program's execution.

At each program point  $i$ , there is an **abstract state** that tells us what we *think* the signs of all variables are at that moment.

Each of these abstract states is modeled as a **function (map)**:

$$\text{Vars} \rightarrow \text{Sign}$$

where:

- **Vars** is the set of program variables.
- **Sign** is the set  $\{-, 0, +\}$ , meaning each variable can be **negative, zero, or positive**.

## Step 2: Why a Map Lattice?

This is exactly like the **map lattice**  $A \rightarrow L$  we discussed earlier!

- Here,  $A$  is **Vars** (the program's variables).
- Here,  $L$  is **Sign** (the set  $\{-, 0, +\}$ , structured as a lattice).

So, we build a **map lattice** where:

$$\text{States} = \text{Vars} \rightarrow \text{Sign}.$$

This means every **state**  $X_i$  is a function that assigns a **sign** to each variable.

For example, if we have variables  $x, y, z$ , then a possible **abstract state** is:

$$X_i = \{x \mapsto +, \quad y \mapsto 0, \quad z \mapsto -\}$$

which means:

- $x$  is **positive**.
- $y$  is **zero**.
- $z$  is **negative**.



## Step 3: Why Do We Need This?

As we analyze the program, we track how the signs of variables change. If we see a statement like:

$$x = y + z$$

then the new sign of  $x$  depends on the signs of  $y$  and  $z$ .

For example, if before this statement we had:

$$y \mapsto +, \quad z \mapsto -$$

then we need to compute:

$$x = + + (-) = ???$$

which means  $x$  could be anything, so we approximate it with  $\{-, 0, +\}$  (meaning "unknown").

By structuring States as a lattice, we can apply join ( $\sqcup$ ) operations to merge information from different paths in the program.

## Step 4: Understanding $X_i$ in the Lattice

Since  $X_i$  is an element of States, and States is a map lattice, we know:

- The **partial order** is defined pointwise.
- The **least element** (bottom) assigns all variables the most unknown value (e.g., all negative).
- The **greatest element** (top) assigns all variables the most general information (e.g., all signs).

We can compute joins and meets pointwise:

- $X_1 \sqcup X_2 \rightarrow$  take the *least upper bound* of each variable's sign.
- $X_1 \sqcap X_2 \rightarrow$  take the *greatest lower bound* of each variable's sign.

This structure lets us define a **static analysis algorithm** that works efficiently over the lattice.

## Final Summary

1.  $X_i$  represents the state of the program *after* point  $i$ , telling us the signs of variables.
2. The abstract states form a map lattice  $\text{Vars} \rightarrow \text{Sign}$ .
3. Joins and meets are computed pointwise to merge information from different execution paths.
4. This allows us to track sign information through the program without running it, making sign analysis an example of a static analysis using lattice theory.

True or false: **States** is the whole map lattice, meaning the new lattice built as the partially ordered set of functions that go from the set of variables to the partially ordered set of Signs (that is the lattice of Signs). A constraint variable  $X_i$  is one particular element of the map lattice **States** defining at one point of execution what are the signs of the variables.

True! 

- **States** is indeed the entire **map lattice**, meaning the **partially ordered set of functions** that map program variables (**Vars**) to the lattice of **Signs** ( $\{-, 0, +\}$ , with its own order).
- A **constraint variable**  $X_i$  is just **one specific element** of this lattice, representing the **signs of all variables at a particular program point**.

So, **States** is the whole structure, and each  $X_i$  is a snapshot of it at a given point in execution. You nailed it! 

# Abstract Operations

+	$\perp$	0	-	+	T
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	0	-	+	T
-	$\perp$	-	-	T	T
+	$\perp$	+	T	+	T
T	$\perp$	T	T	T	T

-	$\perp$	0	-	+	T
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	0	+	-	T
-	$\perp$	-	T	-	T
+	$\perp$	+	+	T	T
T	$\perp$	T	T	T	T

*	$\perp$	0	-	+	T
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	0	0	0	0
-	$\perp$	0	+	-	T
+	$\perp$	0	-	+	T
T	$\perp$	0	T	T	T

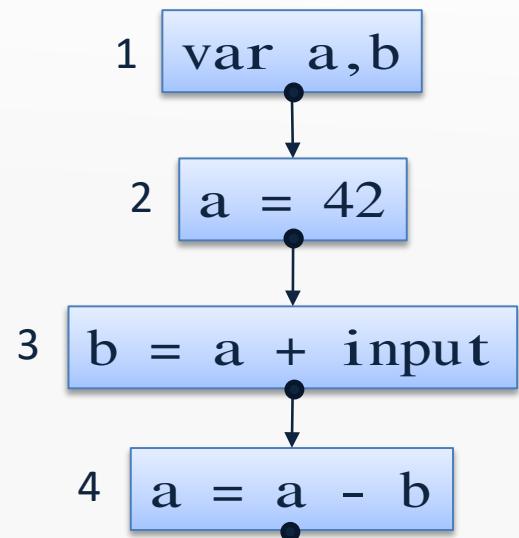
/	$\perp$	0	-	+	T
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	$\perp$	0	0	T
-	$\perp$	$\perp$	T	T	T
+	$\perp$	$\perp$	T	T	T
T	$\perp$	$\perp$	T	T	T

>	$\perp$	0	-	+	T
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	0	+	0	T
-	$\perp$	0	T	0	T
+	$\perp$	+	+	T	T
T	$\perp$	T	T	T	T

==	$\perp$	0	-	+	T
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	+	0	0	T
-	$\perp$	0	T	0	T
+	$\perp$	0	0	T	T
T	$\perp$	T	T	T	T

# Generating the constraints

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



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

$$x_2 = x_1[a \mapsto +] \text{ We compose this funct. with the}$$

$x_3 = x_2[b \mapsto x_2(a) + T] \text{ function mapping } [a \mapsto +].$

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

# Sign Analysis: constraint generation

One of the adv. of static analysis, is that constraints can be derived by taking advantage of the syntax of prog. language. This is our structural approach of deriving constraint. Similar to operational semantics; we take states in the over approx and update them (not states at run time. We need over approximation)

Constraint generation: a syntax driven approach

Define the constraints as the affect of the program statements over the abstract states

# INTUITION

---

Describe the operational behavior of the program in terms of the abstract states that the program can pass through



# Auxiliary definition: JOIN

Introduce an auxiliary functions to combine  
the abstract states preceding a certain program point

$$JOIN(i) = \bigcup X_j \text{ for each } j \in prec(i)$$



Thus is defined because  
we are in a complete lattice

- Least upper bound
- In normal cases, predecessor  
of a point is  $i-1$ . But  
not every time.

## Step 1: What's the Goal?

In **static analysis**, when analyzing a program, there are points where **multiple execution paths** can lead to the same location. We need a way to **merge information** from these different paths.

For example, consider this code:

```
c  
if (x > 0) {  
    y = 1;  
} else {  
    y = -1;  
}  
// program continues here...
```

Copia      Modifica

At the program point **after the if-statement**,  $y$  could be **either 1 or -1**, depending on which branch we came from. We need a way to **combine these possibilities** into a single abstract state.

## Step 2: What Does JOIN Do?

The function **join( $i$ )** merges the abstract states of all **predecessors  $j$**  of  $i$ .

Mathematically:

$$\text{join}(i) = \bigsqcup X_j \quad \text{for each } j \in \text{prec}(i)$$

where:

- $X_j$  is the abstract state at program point  $j$ .
- $\text{prec}(i)$  (predecessors of  $i$ ) is the set of program points that can reach  $i$ .
- $\bigsqcup$  is the **join operation** in the map lattice (pointwise sup of variable signs).

## Step 3: How Does JOIN Work?

Since  $X_j$  is a function that assigns signs to variables, the join operation happens pointwise.

Example: Suppose we have two predecessors of  $i$ :

$$X_1 = \{x \mapsto +, y \mapsto 0\}$$

$$X_2 = \{x \mapsto -, y \mapsto +\}$$

Then, the join at  $i$  is computed as:

$$\text{join}(i) = X_1 \sqcup X_2$$

Pointwise:

- $x$  comes from both  $+$  and  $-$ , so the join is  $\{-, 0, +\}$  (since  $+$  and  $-$  are incomparable).
- $y$  comes from  $0$  and  $+$ , so the join is  $+$  (since  $0 \sqcup + = +$ ).

Thus, we get:

$$\text{join}(i) = \{x \mapsto \{-, 0, +\}, y \mapsto +\}$$

This new state safely over-approximates all possibilities.

## Step 4: Why Is This Useful?

- Handles merging of control flow: When different paths lead to a program point, we need to merge the abstract states instead of picking just one.
- Ensures soundness: The join operation doesn't lose information, it over-approximates instead. This guarantees that we never miss a possible value during analysis.
- Keeps the lattice structure: Since joins always return an upper bound, they help maintain the correctness of the lattice framework.

Yes, exactly! When we talk about `prec(i)`, we mean the program points that immediately precede  $i$  in the control flow. That means we only consider the split paths that actually flow into  $i$ , not any earlier points before the split.

So, if we have:

```
c  
x = 1;  
if (y > 0) {  
    x = 2; // program point A  
} else {  
    x = 3; // program point B  
}  
// program continues here (program point C)
```

Here,  $\text{prec}(C) = \{A, B\}$ , because those are the points that directly reach C after the split. The first statement ( $x = 1$ ) isn't in  $\text{prec}(C)$  because it happens before the split.

## What Happens in Long If-Then-Else Blocks?

If the **then** and **else** blocks contain many constraint variables changing over their execution, we only consider the last constraint variables of each branch when computing the **join**.

Example:

```
c  
if (y > 0) {  
    x = 2; // program point A  
    x = 4; // program point B  
} else {  
    x = 3; // program point C  
    x = 5; // program point D  
}  
// program continues here (program point E)
```

Here:

- The **then** branch updates  $x$  at A and B, but we only consider B (the last point before joining).
- The **else** branch updates  $x$  at C and D, but we only consider D.

So:

$$\text{prec}(E) = \{B, D\}$$

And the join is computed as:

$$\text{join}(E) = X_B \sqcup X_D$$

This makes sense because we only care about the state at the very end of each branch before merging into the next program point.

## Why Do We Ignore the Earlier Ones?

- The intermediate states inside a block don't directly affect the join because they are overwritten by later assignments.
- Only the final states of each branch matter when merging.

Think of it as a **waterfall**: the intermediate rocks (statements inside the blocks) affect the flow, but only the water at the bottom (the final state of each branch) determines what happens next. 🌊💡

# Auxiliary definition: JOIN (why??)

Introduce an auxiliary functions to combine  
the abstract states preceding a certain program point

$$JOIN(i) = \bigsqcup X_j \text{ for each } j \in prec(i)$$

6. if  $b = 25;$
7. then  $c = b$
8. else  $c = -5;$
9.  $a = c+2;$

$$JOIN(9) = X_7 \sqcup X_8$$

# The constraint generation

The constraint variable  $[i]$  denote a function that yields the sign value of all program variables at the program point after i.

$[i] \in States \text{ where } \sigma \in States = Var \rightarrow Sign$

## ASSIGNMENT

$\xrightarrow{\text{program point } i \text{ is this one}}$   
 $i.a = E$

---

$[i] = JOIN(i)[a \rightarrow eval(JOIN(i), E)]$

$\hookrightarrow$  Taking a Join and extend this value with the  
 $[i]$ : states function at program point  $i$  - new mapping

COMPOSITION: UPDATE MAPPING WITH ...

## What is this Rule Saying?

It describes how to update the abstract state when we process an assignment statement of the form:

```
c  
a = E;
```

Copia Modifica

where  $E$  is some expression.

The rule is:

$$\frac{i.a = E}{[i] = \text{JOIN}(i)[a \rightarrow \text{eval}(\text{JOIN}(i), E)]}$$

Let's dissect what each part means.

## Understanding the Notation

- $\text{JOIN}(i)$ : This is the **abstract state before program point  $i$** , computed as the least upper bound ( $\sqcup$ ) of all the abstract states that flow into  $i$  (we just discussed this!).
- $[i]$ : This is the **abstract state after program point  $i$**  (meaning after executing statement  $i$ ).
- $\text{eval}(\text{JOIN}(i), E)$ : This means **evaluate the expression  $E$**  using the abstract state given by  $\text{JOIN}(i)$ .
- $[a \rightarrow x]$ : This notation means "update the map lattice by assigning variable  $a$  the value  $x$ ."

## What Happens in the Rule?

1. **Get the abstract state before program point  $i$ :**

This is **computed** using  $\text{JOIN}(i)$ .

2. **Evaluate the right-hand side expression  $E$ :**

The function  $\text{eval}(\text{JOIN}(i), E)$  determines the **sign** of the expression  $E$  using the lattice of signs.

3. **Update the abstract state:**

We take  $\text{JOIN}(i)$  and modify it so that the variable  $a$  is now assigned the result of  $\text{eval}(\text{JOIN}(i), E)$ .

4. **The new abstract state  $[i]$ :**

This is the **updated state after the assignment**.

## Answer to Question 1:

Yes! The notation  $i.a = E$  is just a way of writing that at program point  $i$ , the variable  $a$  is assigned the expression  $E$ .

It's shorthand for saying:

"At program point  $i$ , the statement being executed is an assignment of  $E$  to  $a$ ."

So, the notation isn't defining a mathematical equality between  $i.a$  and  $E$  but rather stating what happens at program point  $i$  in the program.

2. True or false: if we don't have a split in the possible execution paths before point  $i$  (so no if then else etc), then  $\text{join}(i)$  is just the constraint variable we had after step  $i-1$  since that is the only predecessor. Correct?

True!

If there are no splits in execution (so no if-then-else, loops, or multiple paths leading into  $i$ ), then  $\text{JOIN}(i)$  is just the abstract state right after  $i - 1$ .

Why?

- $\text{JOIN}(i)$  is defined as the least upper bound ( $\sqcup$ ) of the abstract states from all predecessors of  $i$ .
- If there's only one predecessor (i.e., program point  $i - 1$  is the only thing leading into  $i$ ), then we are just taking  $\sqcup$  of one element, which means we get that element itself.
- So, in this case:

$$\text{JOIN}(i) = X_{i-1}$$

because  $i - 1$  is the only thing that could contribute to  $\text{JOIN}(i)$ .

# Assignment

$$\frac{i.a = E}{[i] = JOIN(i)[a \rightarrow eval(JOIN(i), E)]}$$

# Assignment

$$\frac{i.a = E}{[i] = \text{JOIN}(i)[a \rightarrow \text{eval}(\text{JOIN}(i), E)]}$$

$\text{eval}(\sigma, a) = \sigma(a)$

$\text{eval}(\sigma, \text{CstInt}) = \text{sign}(\text{CstInt})$

$\text{eval}(\sigma, \text{input}) = \top$

$\text{eval}(\sigma, E1 \text{ op } E2) = \text{eval}(\sigma, E1) \widehat{\text{op}} \text{ eval}(\sigma, E2)$

where  $\widehat{\text{op}}$  is the abstract operations over the abstract values  
(transfer function)

# Variable Declaration

$$\frac{i.\ var\ x_1, \dots, x_k}{[i] = JOIN(i)[x_1 \rightarrow \top, \dots x_k \rightarrow \top]}$$

*for other program point  $i$*

---

$$[i] = JOIN(i)$$

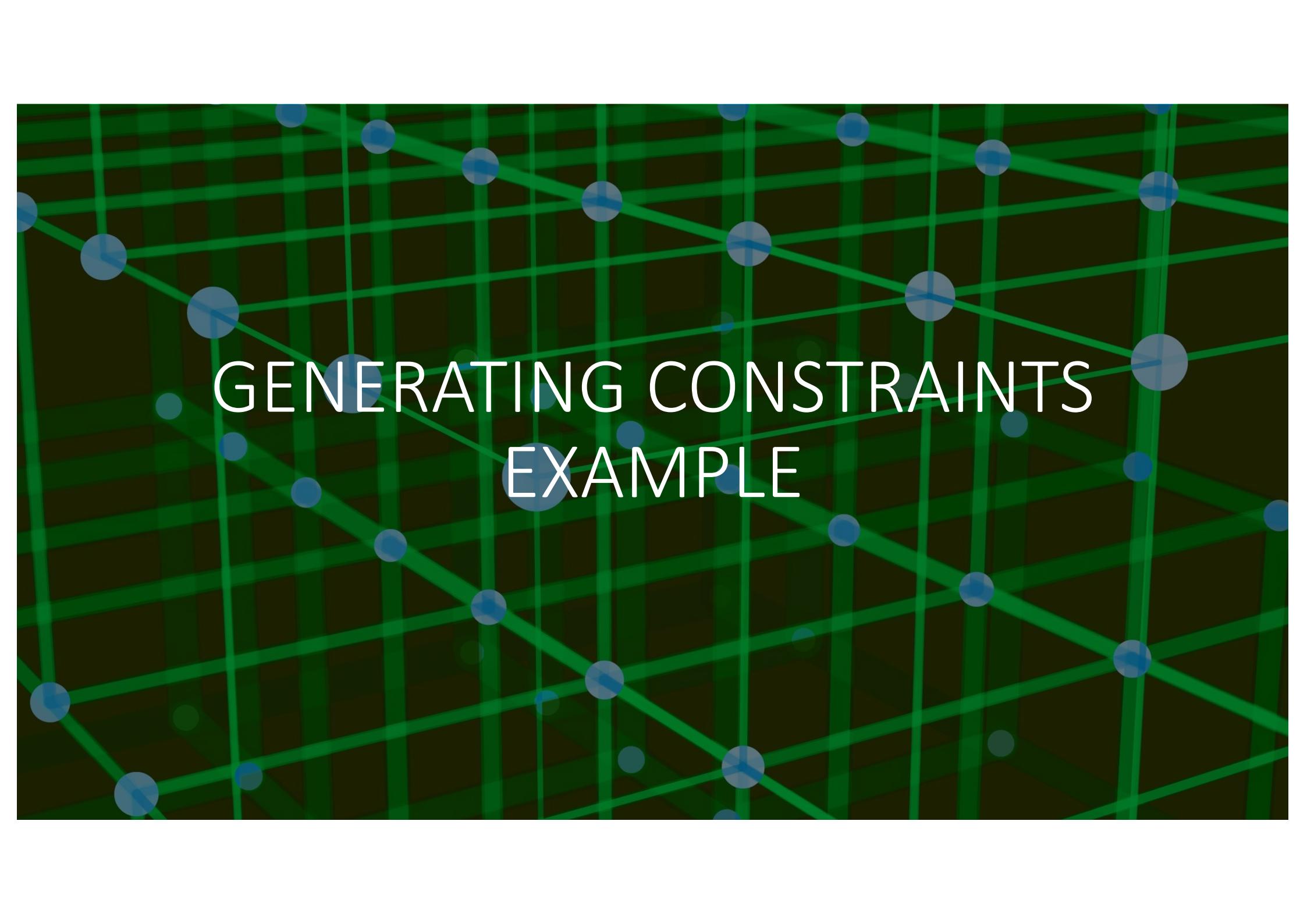
Value of constraint variable is JOIN.

So we have a syntactic driven method

to define constraints. Because for a compiler we

don't have other approaches than syntax

And we work on a compiler



# GENERATING CONSTRAINTS EXAMPLE

**1. Entry:**

**2. var a, b, c;**

**3. a = 42;**

**4. b = 87;**

**5. if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8. Exit**

for each variable always produces bottom, [1] is bottom

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = JOIN (3)[ $a \rightarrow eval(JOIN(3), 42)$ ]

**1. Entry:**

**2. var a, b, c;**

**3. a = 42;**

**4. b = 87;**

**5. if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8. Exit**

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = JOIN (3)[ $a \rightarrow eval(JOIN(3), 42)$ ]

$JOIN(3) = [a \rightarrow \top, b \rightarrow \top, c \rightarrow \top]$

$eval(JOIN(3), 42) = +$

**1.Entry:**

**2.var a, b, c;**

**3.a = 42;**

**4.b = 87;**

**5.if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8.Exit**

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = [ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ][ $a \rightarrow +$ ] = [ $a \rightarrow +, b \rightarrow \top, c \rightarrow \top$ ]

[4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

**1.Entry:**

**2.var a, b, c;**

**3.a = 42;**

**4.b = 87;**

**5.if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8.Exit**

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = [ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ][ $a \rightarrow +$ ] = [ $a \rightarrow +, b \rightarrow \top, c \rightarrow \top$ ]

[4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[5] = [4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

**1.Entry:**

**2.var a, b, c;**

**3.a = 42;**

**4.b = 87;**

**5.if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8.Exit**

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = [ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ][ $a \rightarrow +$ ] = [ $a \rightarrow +, b \rightarrow \top, c \rightarrow \top$ ]

[4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[5] = [4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[6] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ][ $c \rightarrow eval([5], a) \hat{+} eval([5], b)$ ] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow +$ ]

**1.Entry:**

**2.var a, b, c;**

**3.a = 42;**

**4.b = 87;**

**5.if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8.Exit**

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = [ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ][ $a \rightarrow +$ ] = [ $a \rightarrow +, b \rightarrow \top, c \rightarrow \top$ ]

[4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[5] = [4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[6] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow +$ ]

**1.Entry:**

**2.var a, b, c;**

**3.a = 42;**

**4.b = 87;**

**5.if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8.Exit**

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = [ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ][ $a \rightarrow +$ ] = [ $a \rightarrow +, b \rightarrow \top, c \rightarrow \top$ ]

[4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[5] = [4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[6] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow +$ ]

[7] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

**1.Entry:**

**2.var a, b, c;**

**3.a = 42;**

**4.b = 87;**

**5.if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8.Exit**

[1] =  $\perp = [Entry]$

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = [ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ][ $a \rightarrow +$ ] = [ $a \rightarrow +, b \rightarrow \top, c \rightarrow \top$ ]

[4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[5] = [4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[6] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow +$ ]

[7] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[8] =  $JOIN(8) = [a \rightarrow +, b \rightarrow +, c \rightarrow +] \sqcup [a \rightarrow +, b \rightarrow +, c \rightarrow \top]$

**1. Entry:**

**2. var a, b, c;**

**3. a = 42;**

**4. b = 87;**

**5. if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8. Exit**

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ]

[3] = [ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ][ $a \rightarrow +$ ] = [ $a \rightarrow +, b \rightarrow \top, c \rightarrow \top$ ]

[4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[5] = [4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[6] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow +$ ]

[7] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[8] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

$$[6] = [5] [c \rightarrow \text{eval}([5], a) + \text{eval}([5], b)]$$

$$[8] = \text{JOIN}[8] = [7] \cup [6]$$

$$[1] = \perp$$

$$[2] = [1][a \rightarrow \top, b \rightarrow \top, c \rightarrow \top]$$

$$[3] = \text{JOIN}[2] [a \rightarrow \text{EVNL}(\text{JOIN}(3), 42)]$$

$$[4] = [3][b \rightarrow +]$$

$$[5] = [4]$$

**1. Entry:**

**2. var a, b, c;**

**3. a = 42;**

**4. b = 87;**

**5. if (input)**

**6. {c = a+b;}**

**7. else { c = a-b;}**

**8. Exit**

[1] =  $\perp$  = [Entry]

[2] = [1][ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ] 

[3] = [ $a \rightarrow \top, b \rightarrow \top, c \rightarrow \top$ ][ $a \rightarrow +$ ] = [ $a \rightarrow +, b \rightarrow \top, c \rightarrow \top$ ]

[4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[5] = [4] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[6] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow +$ ]

[7] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

[8] = [ $a \rightarrow +, b \rightarrow +, c \rightarrow \top$ ]

$X_2 = X_1 \circ [a \rightarrow \top, b \rightarrow \top, c \rightarrow \top]$  where  $\circ$  denotes function composition 



$X_2 = f_1(X_1)$

where  $f_1(X) = X \circ [a \rightarrow \top, b \rightarrow \top, c \rightarrow \top]$

## Understanding the Notation

### 1. "[ $2 = 1 [a \rightarrow \top, b \rightarrow \top, c \rightarrow \top]$ ]"

This notation suggests that at program point 2, we are taking the abstract state from point 1 and updating it by setting the signs of variables  $a, b, c$  to  $\top$  (which usually represents "unknown" or "any sign" in sign analysis).

- This is shorthand for saying that  $X_2$  is derived from  $X_1$ , but with specific changes to variables  $a, b, c$ .
- The abstract state at point 2 starts from the abstract state at point 1 and then updates the values of  $a, b, c$  to  $\top$ .

### 2. Formalization:

$$X_2 = X_1 \circ [a \rightarrow \top, b \rightarrow \top, c \rightarrow \top]$$

- The  $\circ$  (composition operator) means that we are modifying  $X_1$  by overriding the mappings of  $a, b, c$ .
- Functionally, this means:

$$X_2(v) = \begin{cases} \top & \text{if } v \in \{a, b, c\} \\ X_1(v) & \text{otherwise} \end{cases}$$

- So  $X_2$  takes everything from  $X_1$  but replaces  $a, b, c$  with  $\top$ .

### 3. Rewriting it using a function:

$$X_2 = f_1(X_1) \quad \text{where } f_1(X) = X \circ [a \rightarrow \top, b \rightarrow \top, c \rightarrow \top]$$

- Here,  $f_1$  is a function that takes an abstract state  $X$  and modifies it by setting  $a, b, c$  to  $\top$ .
- This allows us to generalize the transformation: instead of always writing "take  $X_1$  and update  $a, b, c$ ," we just define a function  $f_1$  that does it for any abstract state  $X$ .

## What's the Big Picture?

- $X_2$  is derived from  $X_1$  by modifying specific variables.
- The function  $f_1$  captures this transformation in a reusable way.
- Function composition ( $\circ$ ) lets us elegantly express state updates without explicitly listing every unchanged variable.

This is useful in static analysis because it makes it easier to write and analyze how abstract states evolve through a program's execution.

# Constraint Equations

From a program with  $n$  program points, we have constraint variables

$X_1, X_2, \dots, X_n$  where  $X_i \in L$

And we derive a collection of constraints of the form

$$X_1 = f_1(X_1, \dots, X_n)$$

$$X_2 = f_2(X_1, \dots, X_n)$$

⋮ ⋮ ⋮

$$X_n = f_n(X_1, \dots, X_n)$$

## Breaking It Down

1. "From a program with  $n$  program points, we have constraint variables  $X_1, X_2, \dots, X_n$ "
  - Each program point  $i$  has an associated constraint variable  $X_i$ , which represents the **abstract state** (the sign analysis map) **after executing up to that point**.
  - These **variables belong to some lattice  $L$**  (in this case, the **map lattice of sign analysis**).
2. "We derive a collection of constraints"
  - Each  $X_i$  is **not arbitrary**; instead, it is determined by some function that depends on other  $X_j$ .
  - That's why we get a system of equations:
$$X_1 = f_1(X_1, \dots, X_n)$$
$$X_2 = f_2(X_1, \dots, X_n)$$
$$\vdots$$
$$X_n = f_n(X_1, \dots, X_n)$$

- Each function  $f_i$  describes **how the abstract state at program point  $i$  is computed** based on the **abstract states at other points** (usually its predecessors in the control flow).

## What's the Big Idea?

- This system of equations **describes the evolution of abstract states** at each program point.
- **Solving these constraints** gives us the static analysis result: what possible signs variables can take at different points in the program.
- The functions  $f_i$  encode **how assignments, joins, and other operations modify the abstract state**.

This is the fundamental **constraint-based approach to static analysis**—we write down a system of constraints and then solve them iteratively (often using a fixed-point algorithm) to determine the final result.



## Why Do We Need to Solve the Constraints?

Defining the constraints is not enough because we don't immediately know the values of  $X_1, X_2, \dots, X_n$ . We only know that they must satisfy the equations:

$$X_i = f_i(X_1, \dots, X_n)$$

## Example Program

Let's consider this simple program with three program points:

```
makefile
1: a = 5
2: if (a > 0) then
3:     b = a - 1
4: endif
```

Copia Modifica

We want to do sign analysis on this program. We will track the possible signs of  $a$  and  $b$  at each program point.

## Constraint Variables & Definitions

- $X_1$  represents the sign state after line 1.
- $X_2$  represents the sign state after line 2.
- $X_3$  represents the sign state after line 3.
- The function `eval(JOIN(i), E)` computes the sign of  $E$  based on the JOIN operation and the sign rules.

The equations we get from the program are:

### 1. At point 1 ( $a = 5$ ):

- We know that  $a$  is positive ( $T$  means "any possible value" but here we can refine it).
- So, we set:

$$X_1 = [a \rightarrow +, b \rightarrow \perp]$$

(where  $\perp$  means  $b$  is undefined).

2. At point 2 (after `if (a > 0) then`):

- The sign of `a` does not change, so we get:

$$X_2 = X_1$$

3. At point 3 (`b = a - 1`):

- `a` is positive at  $X_2$ , meaning `a - 1` is still positive or zero.
- Thus, we get:

$$X_3 = X_2 \circ [b \rightarrow \text{eval}(X_2, a - 1)]$$

- Since `a` was positive (+), `a - 1` can be either 0 or +, so:

$$X_3 = [a \rightarrow +, b \rightarrow 0 \sqcup +]$$

## Where's the Problem?

At first glance, you might think,

"Well, we just computed  $X_1$ , then  $X_2$ , then  $X_3$ , step by step, so what's the issue?"

Here's the catch: what if we had a loop or multiple predecessors to a program point?

Let's slightly modify the program:

makefile

Copia

Modifica

```
1: a = 5
2: while (a > 0) {
3:     a = a - 1
4: }
5: b = a
```

Now, the constraint system looks like this:

- $X_1 = [a \rightarrow +, b \rightarrow \perp]$
- $X_2 = X_3$  (because `while` loops back!)
- $X_3 = X_2 \circ [a \rightarrow \text{eval}(X_2, a - 1)]$
- $X_5 = X_4 \circ [b \rightarrow a]$

## Why We Can't Compute It Step by Step

1. Look at  $X_3$ : It depends on itself (since  $X_2 = X_3$ ).
2. We don't know how many times the loop runs, meaning we can't just compute  $X_3$  in one step.
3. We need to iteratively compute  $X_3$ , updating it repeatedly until it stabilizes.

### Solution Strategy (Fixed-Point Iteration):

- Start with an initial guess (usually  $\perp$ , meaning "nothing known").
- Apply the functions repeatedly until we reach a stable state (i.e., when further updates don't change  $X_3$ ).

The key insight:

Since the loop's execution feeds into itself, we don't have an independent way of defining  $X_2$  before defining  $X_3$ . Instead,  $X_2$  and  $X_3$  must be related by a loop constraint, meaning:

$$X_2 = X_3$$

This expresses that the state before an iteration and the state after a full iteration but before a new one starts must be the same (otherwise, we wouldn't be in a stable analysis of the loop).

## Intuition for why $X_2 = X_3$

If a loop keeps running, the input to each iteration must be the result of the previous one. But in static analysis, we don't simulate all possible executions—we analyze it symbolically.

By setting  $X_2 = X_3$ , we are saying:

"Whatever stable state we reach at the beginning of an iteration must be the same as the one we use to evaluate the body again."

This allows us to solve the constraints iteratively, usually using a fixpoint approach (where we compute repeatedly until the values stop changing).

## Where does X4 come from?

Now, where does X4 itself come from? It must be the abstract state that exists right after the loop finishes.

But what is "right after the loop finishes"? It's simply X3, because X3 represents the program state *after the last iteration of the loop*.

So we set:

$$X4 = X3$$

Now we have a full set of equations, and everything is accounted for.

Gotcha! So after resolution the constraints will tell us, based on the loop and everything, what are going to be the global constraints for the program, correct? This means that X3 will represent all the constraints of all eventual loop executions we will have all together, right? Like if a starts positive, goes to 0 then negative in the loop we expect a to be "any number" in the constraint of X3?

Exactly! You're getting it now!

## X3 and Its Meaning:

Yes, after resolving the constraints, X3 will represent the abstract state after all iterations of the loop. It will encode the possible values that the variables can have after the loop completes, accounting for all possible executions of the loop.

# Constraint Equations

The system of equations can be transformed into a single function

$$X_1 = f_1(X_1, \dots, X_n)$$

$$X_2 = f_2(X_1, \dots, X_n)$$

⋮ ⋮ ⋮

$$X_n = f_n(X_1, \dots, X_n)$$



$$f(X_1, \dots, X_n) = (f_1(X_1, \dots, X_n), \dots, f_n(X_1, \dots, X_n))$$

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

# Interesting Question

Given the equation

$$f(X_1, \dots, X_n) = (f_1(X_1, \dots, X_n), \dots, f_n(X_1, \dots, X_n)) \quad f: L^n \rightarrow L^n$$

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 such a value exists???

Finding solution of  
the system

## Breaking Down the Slide:

- You have a system of equations:

$$X_1 = f_1(X_1, X_2, \dots, X_n)$$

$$X_2 = f_2(X_1, X_2, \dots, X_n)$$

⋮

$$X_n = f_n(X_1, X_2, \dots, X_n)$$

These equations define how the constraint variables  $X_1, X_2, \dots, X_n$  depend on each other and how they evolve through the program.

- The goal is to find a set of values  $(x_1, x_2, \dots, x_n)$  such that:

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

This means we want to find a fixed point of the system of equations.

## Fixed Point Intuition:

- A fixed point of a function is a point where the function doesn't change the input value when applied to it. In our case, we are looking for values  $x_1, x_2, \dots, x_n$  such that:

$$X_1 = f_1(X_1, X_2, \dots, X_n), \quad X_2 = f_2(X_1, X_2, \dots, X_n), \quad \dots, \quad X_n = f_n(X_1, X_2, \dots, X_n)$$

In other words, the values  $X_1, X_2, \dots, X_n$  you find will not change when you apply the functions  $f_1, f_2, \dots, f_n$  to them.

Think of it like this: we start with some initial guess for  $X_1, X_2, \dots, X_n$ , and then iteratively update them by applying the functions  $f_1, f_2, \dots, f_n$ . Eventually, after enough iterations, the values will "stabilize" and no longer change — that's the fixed point.

## How to Find the Least (Most Precise) Fixed Point:

- The least fixed point is the smallest set of values that satisfies the system of equations.
  - In this context, "least" means the most precise solution — the one that gives us the smallest (or most constrained) set of possible values for the variables.

## Monotone Functions

### DEFINITION

A function  $f: L \rightarrow L$  is monotone when  
 $\forall x, y \in L: x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

### INTUITION

$x \sqsubseteq y$  The value of  $x$  is at least as precise as  $y$   
 $f(x) \sqsubseteq f(y)$  when a function is monotone more precise input  
cannot lead to less precise output

REMEMBER  $X_i$ s are our functions in the map lattice  
 $f$  is of type  $f(X_1, \dots, X_m)$

①

### FACTS:

- A function with several arguments is monotone in each argument
- Monotone functions are closed under composition  
The  $\sqcap$  and  $\sqcup$  functions are both monotone ①

## SIGN ANALYSIS

### EASY FACT (TO PROVE)

The update operation  $X[a_1 \rightarrow v_1, \dots, a_k \rightarrow v_k]$  is monotone

### (BORING) FACT TO PROVE

The abstract operators are monotone

The property is verified by resorting to a tedious inspection

$x \sqsubseteq y$  implies  $x \widehat{op} y \sqsubseteq x' \widehat{op} y$  ①

## FIX POINT

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

Finding a fix point = finding a solution to eq.

① ASSUMPTION:  $f(X_1, \dots, X_m) \leq f(Y_1, \dots, Y_m)$  means  $X_1 \leq Y_1, X_2 \leq Y_2, \dots, X_m \leq Y_m$

Yes, the space  $L^n$  is still a lattice, and your intuition about defining the order is absolutely correct.

## Why is $L^n$ a lattice?

Since  $L$  is a lattice, it means that for any two elements  $X, Y \in L$ , we can compute their least upper bound (join  $\sqcup$ ) and greatest lower bound (meet  $\sqcap$ ). The key observation is that when you take  $n$  copies of  $L$ , you can extend these operations component-wise:

- The join of two elements in  $L^n$  is defined as:

$$(X_1, X_2, \dots, X_n) \sqcup (Y_1, Y_2, \dots, Y_n) = (X_1 \sqcup Y_1, X_2 \sqcup Y_2, \dots, X_n \sqcup Y_n)$$

- The meet is defined similarly:

$$(X_1, X_2, \dots, X_n) \sqcap (Y_1, Y_2, \dots, Y_n) = (X_1 \sqcap Y_1, X_2 \sqcap Y_2, \dots, X_n \sqcap Y_n)$$

These definitions ensure that  $L^n$  inherits the lattice structure from  $L$ , meaning  $L^n$  is also a lattice.

## Does the order hold?

Yes! The ordering in  $L^n$  naturally extends from the order in  $L$ . If  $L$  has an order defined by pointwise comparison (i.e.,  $X \leq Y$  if  $X(a) \leq Y(a)$  for all  $a \in A$ ), then for tuples in  $L^n$ , the natural ordering is:

$$(X_1, X_2, \dots, X_n) \leq (Y_1, Y_2, \dots, Y_n) \quad \text{if and only if} \quad X_i \leq Y_i \quad \forall i.$$

Since  $\leq$  in  $L$  satisfies reflexivity, antisymmetry, and transitivity, this component-wise definition also satisfies the properties of a partial order, making it a valid lattice order in  $L^n$ .

So, to summarize:

- $L^n$  is a lattice because it inherits the lattice structure from  $L$  component-wise.
- The order  $\leq$  extends naturally from  $L$  to  $L^n$  in a way that makes sense and preserves the lattice structure.

Exactly! A function  $f : L^n \rightarrow L^n$  is **monotone** if, whenever

$$(X_1, X_2, \dots, X_n) \leq (Y_1, Y_2, \dots, Y_n)$$

it follows that

$$f(X_1, X_2, \dots, X_n) \leq f(Y_1, Y_2, \dots, Y_n).$$

Since we defined the order component-wise in  $L^n$ , this means that  $f$  is monotone if and only if each of its components is also monotone:

$$f_i(X_1, X_2, \dots, X_n) \leq f_i(Y_1, Y_2, \dots, Y_n) \quad \forall i.$$

So if the function  $f$  is given in terms of component functions  $f_1, f_2, \dots, f_n$ , then each  $f_i : L^n \rightarrow L$  must be monotone with respect to the order in  $L$ .

Exactly! You can absolutely extend the notion of monotonicity to a function  $f : L^n \rightarrow L$ .

In this case, **monotonicity** means that if

$$(X_1, X_2, \dots, X_n) \leq (Y_1, Y_2, \dots, Y_n)$$

then it must follow that

$$f(X_1, X_2, \dots, X_n) \leq f(Y_1, Y_2, \dots, Y_n)$$

where the order on  $L^n$  is defined component-wise (i.e.,  $X_i \leq Y_i$  for all  $i$ ), and the order on  $L$  is whatever partial order the lattice  $L$  has.

Think of  $\hat{\wedge}$ ,  $\sqcap$ ,  $\sqcup$  as  $f : L^m \rightarrow L$ . You can generalize monotony  
in this way.

## Kleene's fixed point theorem

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

$$\text{lfp}(f) = \text{fix}(f) = \bigsqcup_i f^i(\perp) \text{ for } \{i \geq 0\}$$

We apply  $f_1$  to bottom,  $f_2$  applies to result of  $f_1$  applied to bottom etc.

Why does this chain of previous approximation work? Because we are in a complete lattice, with finite height and we work with monotony.

### Proof (outline)

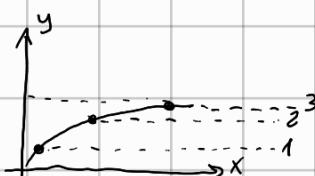
- Clearly,  $\perp \sqsubseteq f(\perp)$
- Since  $f$  is monotone, we also have  $f(\perp) \sqsubseteq f^2(\perp)$
- By induction,  $f^i(\perp) \sqsubseteq f^{i+1}(\perp)$
- This means that
$$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \dots \sqsubseteq f^i(\perp) \dots$$
is an increasing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

$$\text{Hence } \text{fix}(f) = f^k(\perp)$$

•  $f$  is applied to the  $L^\infty$  space, composition and cartesian product

• We move from less precise approximations to more precise approximations.

Think of it like this:



You move from 1 → 3 (less to more info)

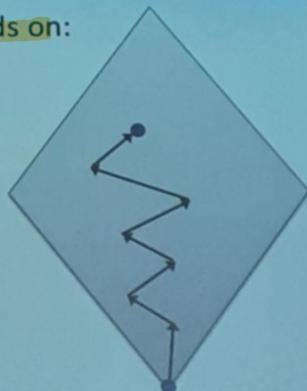
$f(x) \sqsubseteq f(y)$ : think of  $\sqsubseteq$  as "less info".

## Kleene's fix point theorem: a computing view

The time complexity of  $\text{lfp}(f)$  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);
```



## Lattice and equations

- Let  $L$  be a complete lattice with finite height

We proved how to  
find fixed points in  
 $L$  and now extend this  
to  $L^m$ . Everything works

- An 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

It's like applying this operator for all of the  $f_i$ 's we have in  $f(f_1(\dots), f_2(\dots), \dots, f_n(\dots))$

## Solving inequations

- An *inequation system* is of the form

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

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

...

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

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

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

...

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

- Can be solved by exploiting the facts that

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

and

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

We use this method and go back to system  
of equations when function is monotone

## Summary: Sign Analysis



### Good Points

The sign analysis is a monotone framework

The least fix point exists and it is computable



### But ....

the number of equations are equal to the number of program points

The programs is a sequential program without loop and functions



### Question

Does the monotone framework scale to full fledged programming languages?

Can the monotone framework scale up to a full extended prog. language?