Software Architecture and Engineering: Part II

ETH Zurich, Spring 2017

Prof. Martin Vechev

http://www.srl.inf.ethz.ch/

SOFTWARE RELIABILITY LAB

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

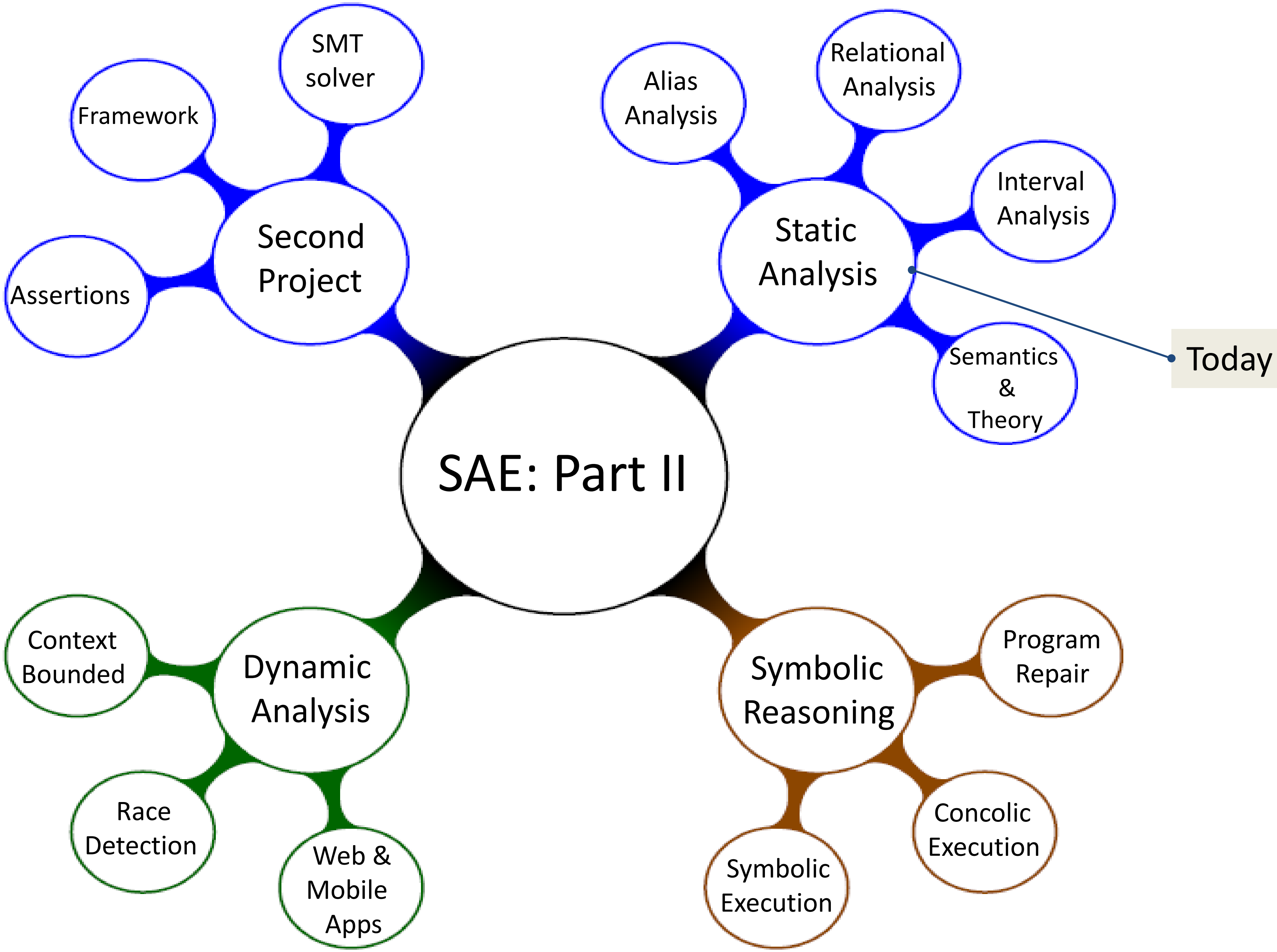
Motivation for Material

- Programs are everywhere
  - e.g., cars, smart phones, mobile apps, software-defined networks, spreadsheets, probabilistic programs, etc.

- Hard to manually reason about programs and get right
  - e.g.,: see Heartbleed bug, drivers, concurrency, etc.

- Defects have a massive economic effect
  - ~60 billion USD annually and growing

Wanted: Automated techniques that find bugs, ensure correctness and performance, and even repair code


SAE: Part II
Second Project
Framework
SMT solver
Assertions
Static Analysis
Alias Analysis
Relational Analysis
Interval Analysis
Semantics & Theory
Today
Dynamic Analysis
Context Bounded
Race Detection
Web & Mobile Apps
Symbolic Reasoning
Program Repair
Concolic Execution
Symbolic Execution

Question

Can you build an automatic analyzer which takes as input an arbitrary program and an arbitrary property such that if the analyzer answers:

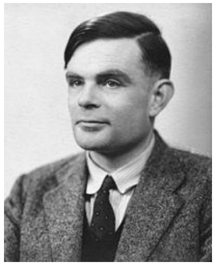
- "Yes" , then it is certain that the property holds
- "No" , then it is certain that the property does not hold

?

Question

Can you build an automatic analyzer which takes as input an arbitrary program and an arbitrary property such that if the analyzer answers:

- "Yes" , then it is certain that the property holds
- "No" , then it is certain that the property does not hold



Alan Turing

Answer:

No. The problem is undecidable

What now?

Change the question

New Question

Can you build an automatic analyzer which takes as input an arbitrary program and an arbitrary property such that if the analyzer answers:

- "Yes" , then it is certain that the property holds
- "No" , then it is ~~certain that the property does not hold~~ unknown if the property holds or not

?

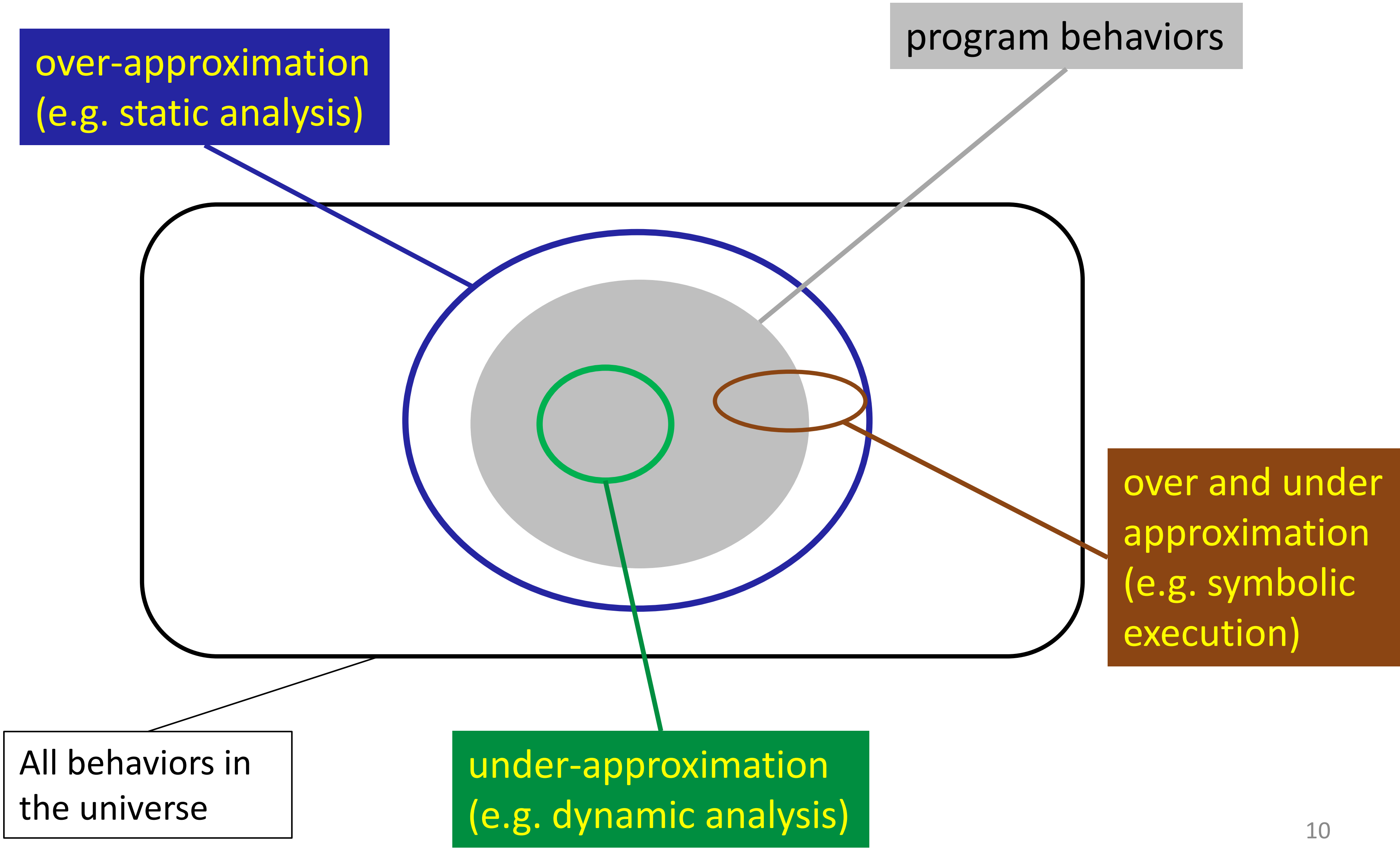
Trivial Solution

```
StaticAnalyzer(Program, Property)
{
    return "No";
}
```

Static Program Analysis: Challenge

The challenge is to build a static analyzer that is able to answer "Yes" for as many programs which satisfy the property.

Approaches to Program Analysis



Plan for Today

- Informal introduction to static program analysis
- Goal: get an intuition
- Understand why we need the math later

Static Program Analysis: cool facts

- Can automatically prove interesting properties such as
  - absence of null pointer dereferences, assertions at a program point, termination, absence of data races, information flow,…
- Can automatically find bugs in large scale programs, or detect bad patterns
  - for instance: the program fails to call the API “close” on a File object
  - or detect stylistic patterns..
  - what else?
- Nice combination of math and system building
  - combines program semantics, data structures, discrete math, logic, algorithms, decision procedures, …

Static Program Analysis: cool facts

- Can run the program without giving a concrete input
  - abstractly execute a piece of code from any point

- No need for manual annotations such as loop invariants
  - they are automatically inferred
  - what is a loop invariant ?

Lets look at a couple of examples what static analysis can do for us…

What is the result of this program ?

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
 if (n>100) then
  r = n-10;
 else
  t1 = n + 11;
  t2 = MC(t1);
  r  = MC(t2);
 endif;
end

var a:int, b:int;
begin
b = MC(a);
end
```

The McCarthy 91 function:

if (n ⩾ 101) then n - 10 else 91

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
 if (n>100) then
  r = n-10;
 else
  t1 = n + 11;
  t2 = MC(t1);
  r  = MC(t2);
 endif;
end

var a:int, b:int;
begin
b = MC(a);
end
```

Invariants per program point
(automatically computed):

```
top
n-101 ≥ 0
-n+r+10 = 0; n-101 ≥ 0
-n+100 ≥ 0
-n+t1-11 = 0; -n+100 ≥ 0
-n+t1-11 =0; -n+100 ≥ 0;
-n+t2-1 ≥ 0; t2-91  ≥ 0
-n+t1-11=0;-n+100 ≥ 0;-n+t2-1 ≥ 0;
t2-91 ≥ 0; r-t2+10 ≥ 0; r-91 ≥ 0
-n+r+10 ≥ 0; r-91 ≥ 0
top
-a+b+10 ≥ 0; b-91 ≥ 0
```

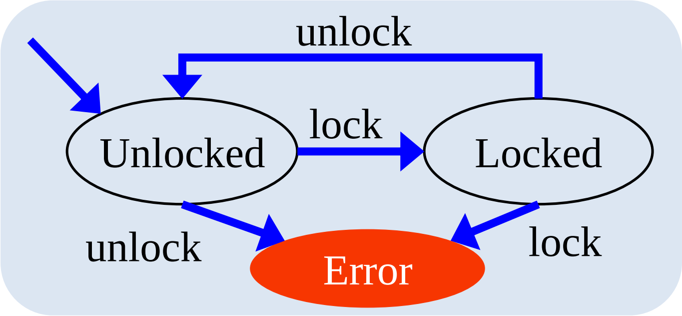
<u>Automatically</u> inferred using **numerical abstract domains**

Network driver: does it do what it is supposed to?

```
driver(int req) {

1: initlock();
2: lock();
3: old = packets;

4: if (req) {
5:   req = req->next;
6:   unlock();
7:   packets++;
   }

8: if (packets != old)
     goto 2;

9: unlock();
}
```

```
enum {Locked,Unlocked}
initlock() {s = Unlocked;}
lock() {
  if  (s == Locked) abort;
  else s = Locked; }
unlock() {
  if  (s == Unlocked) abort;
  else s = Unlocked; }
```

Property we want to check:

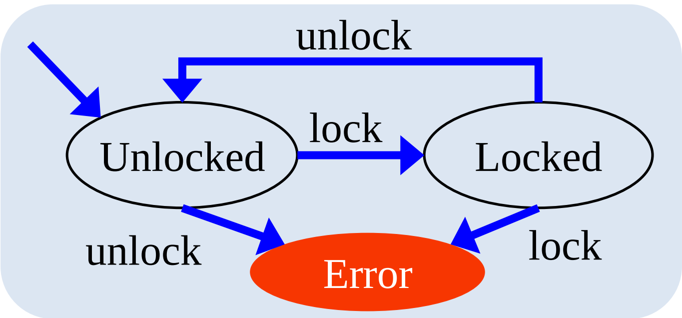


Network driver: does it do what it is supposed to?

```
driver(int req) {

1: initlock();
2: lock();
3: old = packets;

4: if (req) {
5:   req = req->next;
6:   unlock();
7:   packets++;
   }

8: if (packets != old)
      goto 2;

9: unlock();
}
```

```
enum {Locked,Unlocked}
initlock() {s = Unlocked;}
lock() {
  if  (s == Locked) abort;
  else s = Locked; }
unlock() {
  if  (s == Unlocked) abort;
  else s = Unlocked; }
```

Property we want to check:



<u>**Automatically**</u> verified using SLAM based on predicate abstraction

Real-World Program Analyzers (small sample)

A significantly increased interest in the last few years

coverity®

GRAMMATECH

IBM

Microsoft® Research

Klocwork

PANAYA Making ERP Easy

TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS | ACM.org | Join ACM | About Communications | ACM Resources | Alerts & Feeds

SIGN IN

COMMUNICATIONS OF THE ACM

Search

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRACTICE | CAREERS | MAGAZINE ARCHIVE

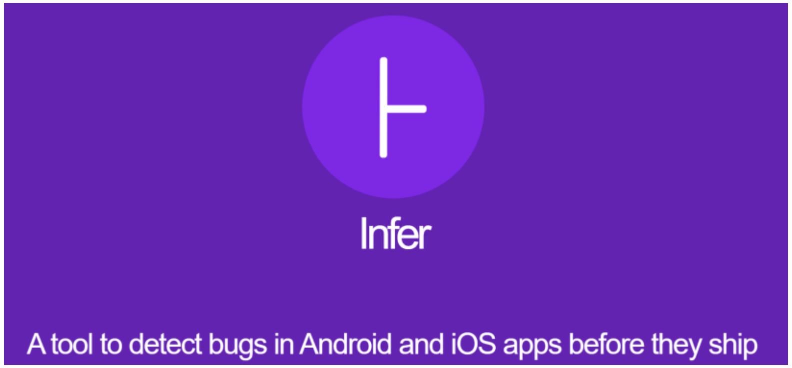
Home / Magazine Archive / February 2010 (Vol. 53, No. 2) / A Few Billion Lines of Code Later: Using Static Analysis... / Full Text

CONTRIBUTED ARTICLES

A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World

Facebook: last 2 years...

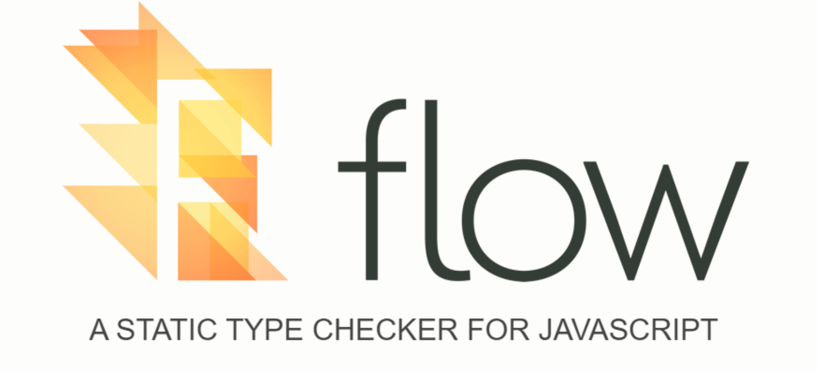
INFER: http://fbinfer.com/



Infer is an automated code review tool based on deep program analysis.

It is fully integrated and is a key part of FB's software development process. Fast and precise, it fits FB's culture on move fast and break things ☺

Flow: http://flowtype.org/



Flow is a static type checker and can automatically infer types of JavaScript programs, both user code and libraries.

Both tools are open sourced and are used by companies beyond Facebook.

Questions

- What are the core principles behind these tools? Is there a general theory?

- What kind of properties/defects/issues can these tools detect?

- How do you go about building one these tools?

- What concerns should be addressed to have such tools be practically adopted by software engineers?

Lets begin…

Static Analysis via Abstract Interpretation

- We will learn a style called abstract interpretation
  - a general theory of how to do approximation systematically
- Abstract interpretation is a very useful thinking framework
  - relate the concrete with the abstract, the infinite with the finite
- Many existing analyses can be seen as abstract interpreters
  - type systems, data-flow analysis, model checking, etc…

Abstract Interpretation: step-by-step

1. select/define an abstract domain
   - selected based on the type of **properties** you want to prove

2. define abstract semantics **for the language** w.r.t. to the domain
   - prove **sound** w.r.t **concrete semantics**
   - involves defining abstract transformers
     - that is, effect of statement / expression on the abstract domain

3. iterate abstract transformers over the abstract domain
   - until we reach a **fixed point**

The **fixed point** is the **over-approximation** of the program

Lets prove an assertion…

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

There are infinitely many executions here. We cannot just enumerate them all.

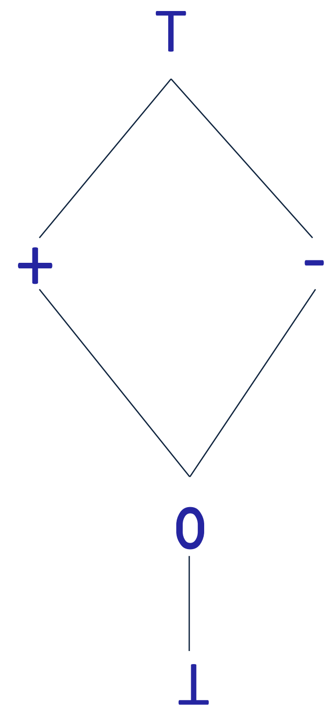
And even if they were finite, it would still take us a long time to enumerate them all…

Instead, let us do some over-approximation, so that we can reduce the space of what we need to enumerate…

Step 1: Select abstraction

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

Lets pick the sign abstraction



Why this abstract domain ?

Question: what does + represent in the sign abstraction?

Question: what does + represent in the sign abstraction?

Answer: + represents all positive numbers and 0.

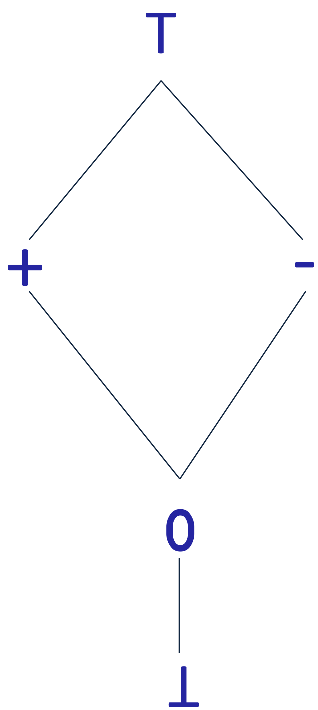
What about -, what does it mean ?

Step 1: Select abstraction

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
2	+	⊥	T

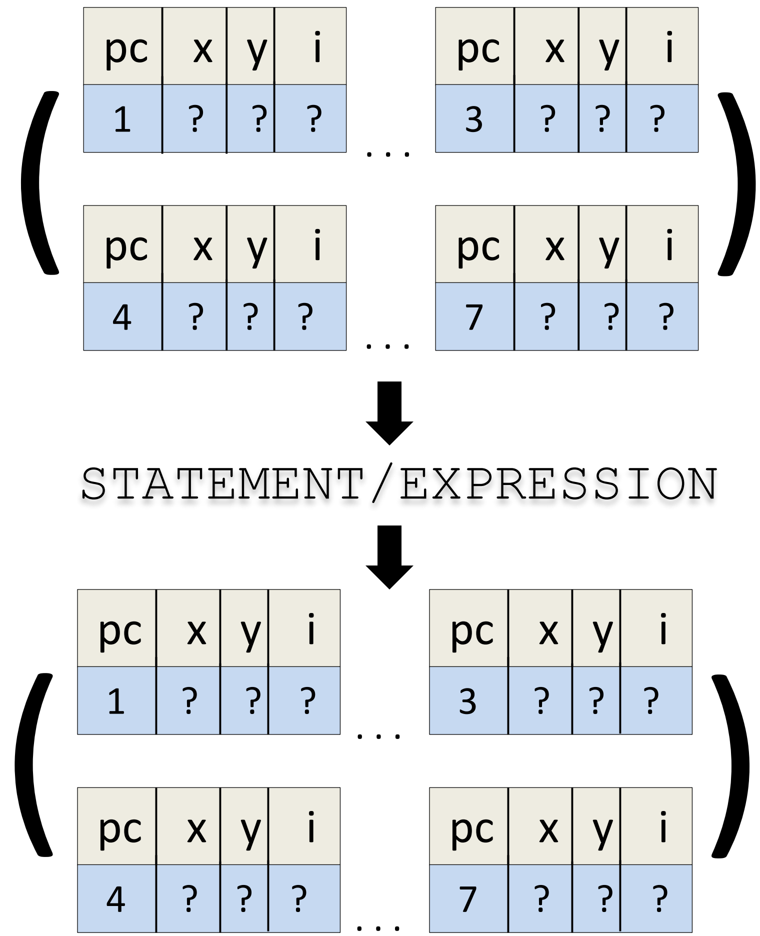
Example Abstract State



An abstract program state is a map from variables to elements in the domain

Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



An abstract transformer describes the effect of statement and expression evaluation on an abstract state

Important Point

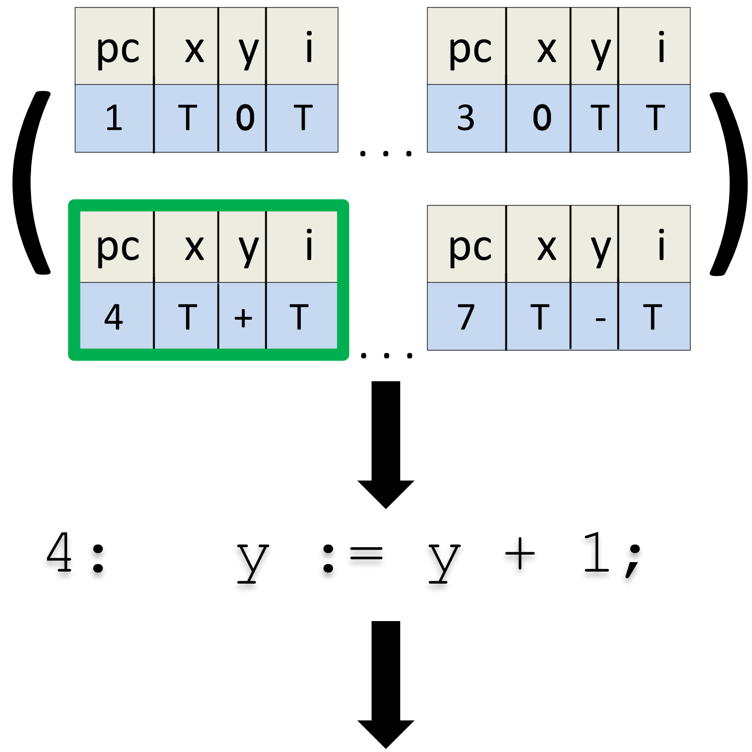
It is important to remember that abstract transformers are defined per programming language once and for all, and not per-program !

That is, they essentially define the new (abstract) semantics of the language (we will see them formally defined later)

This means that any program in the programming language can use the same transformers.

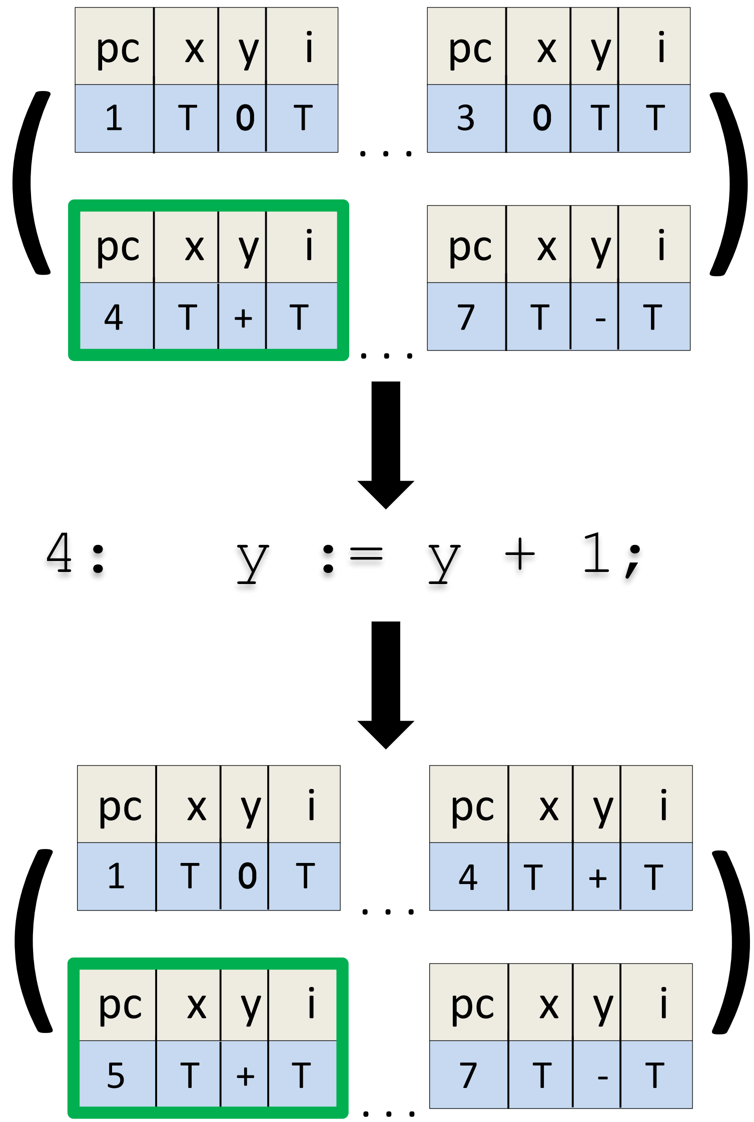
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



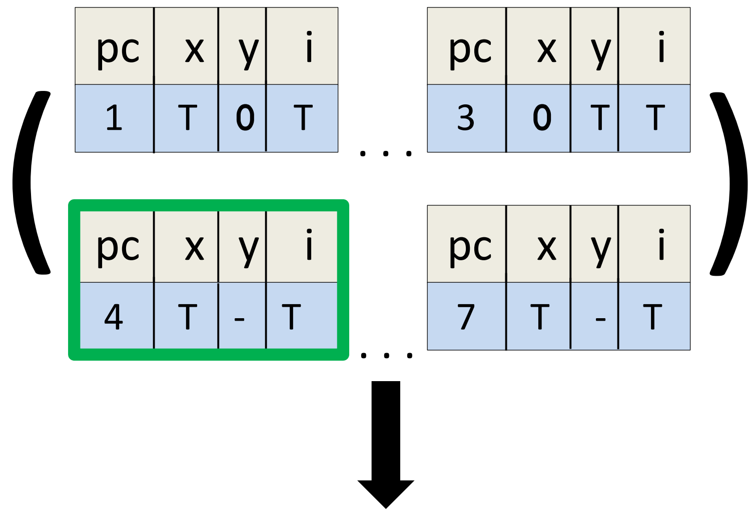
Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



Step 2: Define Transformers

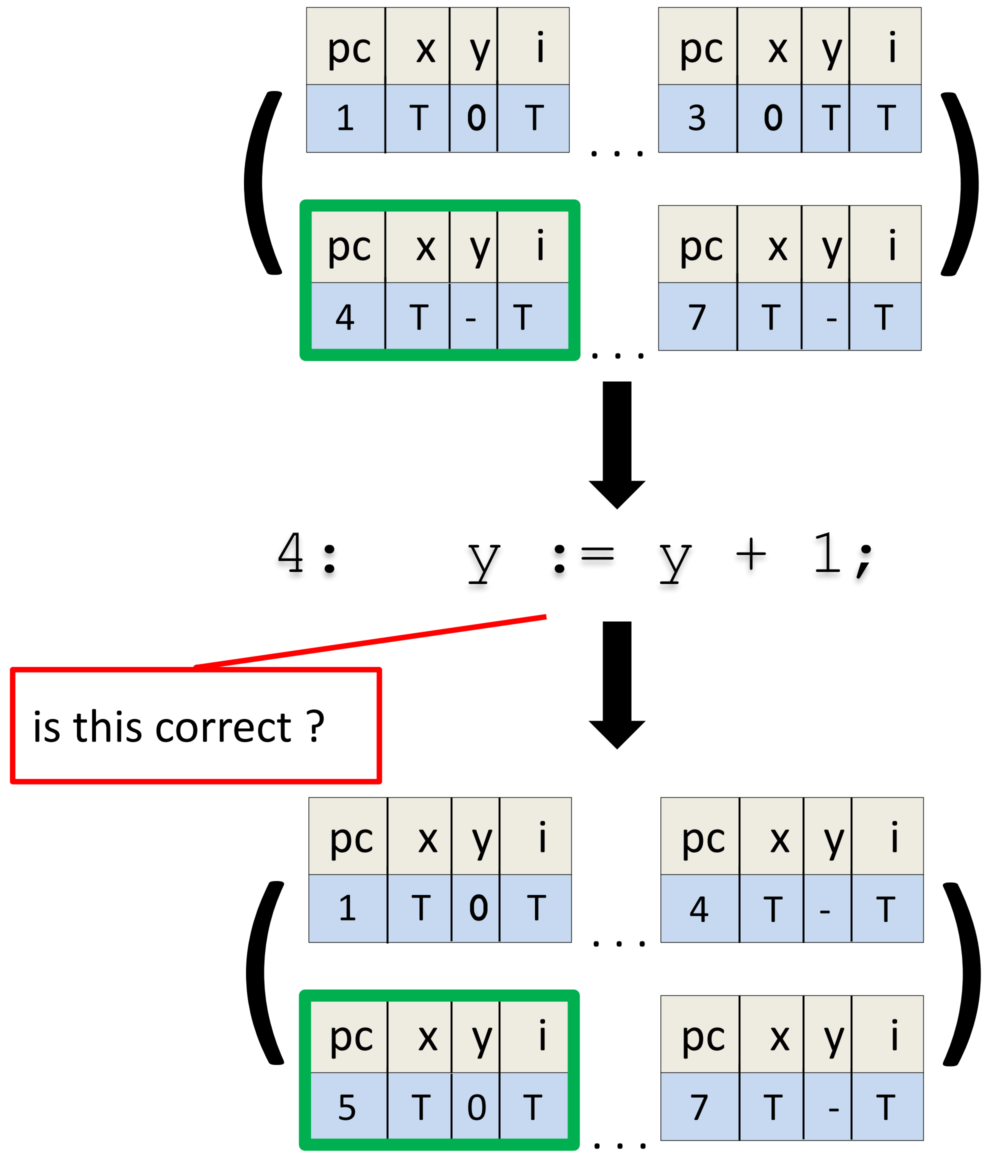
```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```




```
4:    y := y + 1;
```

Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```



Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	T	0	T

...

pc	x	y	i
3	0	T	T

pc	x	y	i
4	T	-	T

...

pc	x	y	i
7	T	-	T

```
4:    y := y + 1;
```

What does correct mean ?

Transformer Correctness

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }

7: assert 0 ≤ x + y
}
```

A correct abstract transformer should always produce results that are a superset of what a concrete transformer would produce

Unsound Transformer

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

This abstract state:

pc	x	y	i
4	T	-	T

represents infinitely many concrete states including:

pc	x	y	i
4	1	-3	2

If we perform `y := y + 1` on this concrete state, we get:

pc	x	y	i
5	1	-2	2

However, the abstract transformer produced an abstract state:

pc	x	y	i
5	T	0	T

This abstract state does not represent any state where y = - 2

The abstract transformer is unsound ! ☹

How about this ?

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	T	0	T
3	0	T	T
4	T	-	T
7	T	-	T

...

```
4:    y := y + 1;
```

This is correct, why?

pc	x	y	i
1	T	0	T
4	T	-	T
5	T	T	T
7	T	-	T

...

How about this ?

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	T	0	T

...

pc	x	y	i
3	0	T	T

pc	x	y	i
4	T	0	T

...

pc	x	y	i
7	T	-	T

```
4:    y := y + 1;
```

Is this sound ? Yes
Is it precise ? No

pc	x	y	i
1	T	0	T

...

pc	x	y	i
4	T	0	T

pc	x	y	i
5	T	T	T

...

pc	x	y	i
7	T	-	T

Imprecise Transformer

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

This abstract state:

pc	x	y	i
4	T	0	T

represents infinitely many concrete states where `y` is always **0** , including:

pc	x	y	i
4	1	0	2

If we perform `y := y + 1` on any of these concrete states, we will always get states where `y` is always positive, such as:

pc	x	y	i
5	1	1	2

However, the abstract transformer produces an abstract state where `y` can be any value, such as:

pc	x	y	i
5	T	T	T

The abstract transformer is imprecise ! ☹

How about this ?

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	T	0	T

...

pc	x	y	i
3	0	T	T

pc	x	y	i
4	T	0	T

...

pc	x	y	i
7	T	-	T

```
4:    y := y + 1;
```

Is this sound ?
Is it precise ?

pc	x	y	i
1	T	0	T

...

pc	x	y	i
4	T	0	T

pc	x	y	i
5	T	+	T

...

pc	x	y	i
7	T	-	T

How about this ?

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	T	0	T

...

pc	x	y	i
3	0	T	T

pc	x	y	i
4	T	0	T

...

pc	x	y	i
7	T	-	T

```
4:    y := y + 1;
```

Is this sound ? Yes
Is it precise ? Yes

pc	x	y	i
1	T	0	T

...

pc	x	y	i
4	T	0	T

pc	x	y	i
5	T	+	T

...

pc	x	y	i
7	T	-	T

Abstract Transformers

It is easy to be sound and imprecise: simply output $\top$

It is desirable to be both sound and precise. If we lose precision, it needs to be clear why and where:

- sometimes, computing the most precise transformer (also called the best transformer) is impossible
- for efficiency reasons, we may sacrifice some precision

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i – 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

Start with the least abstract element

pc	x	y	i
1	⊥	⊥	⊥

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

Lets do some iterations…

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	⊥

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

`int i`

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	⊥
2	⊥	⊥	⊥
3	⊥	⊥	⊥
4	⊥	⊥	⊥
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	⊥	⊥	⊥

`int i`

pc	x	y	i
1	⊥	⊥	T
2	⊥	⊥	⊥
3	⊥	⊥	⊥
4	⊥	⊥	⊥
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	⊥	⊥	⊥

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	⊤

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

```
1: int x :=5;
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

```
1: int x :=5;
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	⊤

pc	x	y	i
2	+	⊥	⊤

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

```
2:   int y :=7;
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:     y := y + 1;
5:     i := i - 1;
6:     goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

```
2:   int y :=7;
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

```
3: if (i ≥ 0)
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	⊥	⊥	⊥
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	⊥	⊥	⊥

3: if (i ≥ 0)

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	+	+	+
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	+	+	-

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	+	+	-

```
4: y := y + 1;
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	+	+	+
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	+	+	-

```
4: y := y + 1;
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	+	+	+
5	+	+	+
6	⊥	⊥	⊥
7	+	+	-

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	+	+	-

```
5: i := i - 1;
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	+	+	+
5	+	+	+
6	⊥	⊥	⊥
7	+	+	-

`5: i := i - 1;`

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	+	+	+
5	+	+	+
6	+	+	T
7	+	+	-

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

```
6: goto 3;
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

```
6: goto 3;
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	⊤

pc	x	y	i
2	+	⊥	⊤

pc	x	y	i
3	+	+	⊤

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	⊤

pc	x	y	i
7	+	+	-

ANY STATEMENT

No matter what statement we execute from this state, we reach that same state

What is the loop invariant ?

Step 4: Check property

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x + y
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

$P \models (0 \leq x + y)$ ✓

$P_{sign} \models (0 \leq x + y)$ ✓

sign domain precise enough
to prove property

Lets change the property

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ x - y
}
```

pc	x	y	i
1	⊥	⊥	⊤

pc	x	y	i
2	+	⊥	⊤

pc	x	y	i
3	+	+	⊤

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	⊤

pc	x	y	i
7	+	+	-

$$P \not\models (0 \leq x - y)$$

$$P_{sign} \not\models (0 \leq x - y)$$

sign domain is sound: property does not hold and it confirms it

Lets change the property again

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

pc	x	y	i
1	⊥	⊥	T

pc	x	y	i
2	+	⊥	T

pc	x	y	i
3	+	+	T

pc	x	y	i
4	+	+	+

pc	x	y	i
5	+	+	+

pc	x	y	i
6	+	+	T

pc	x	y	i
7	+	+	-

$$P \models (0 \leq y - x)$$ ✓

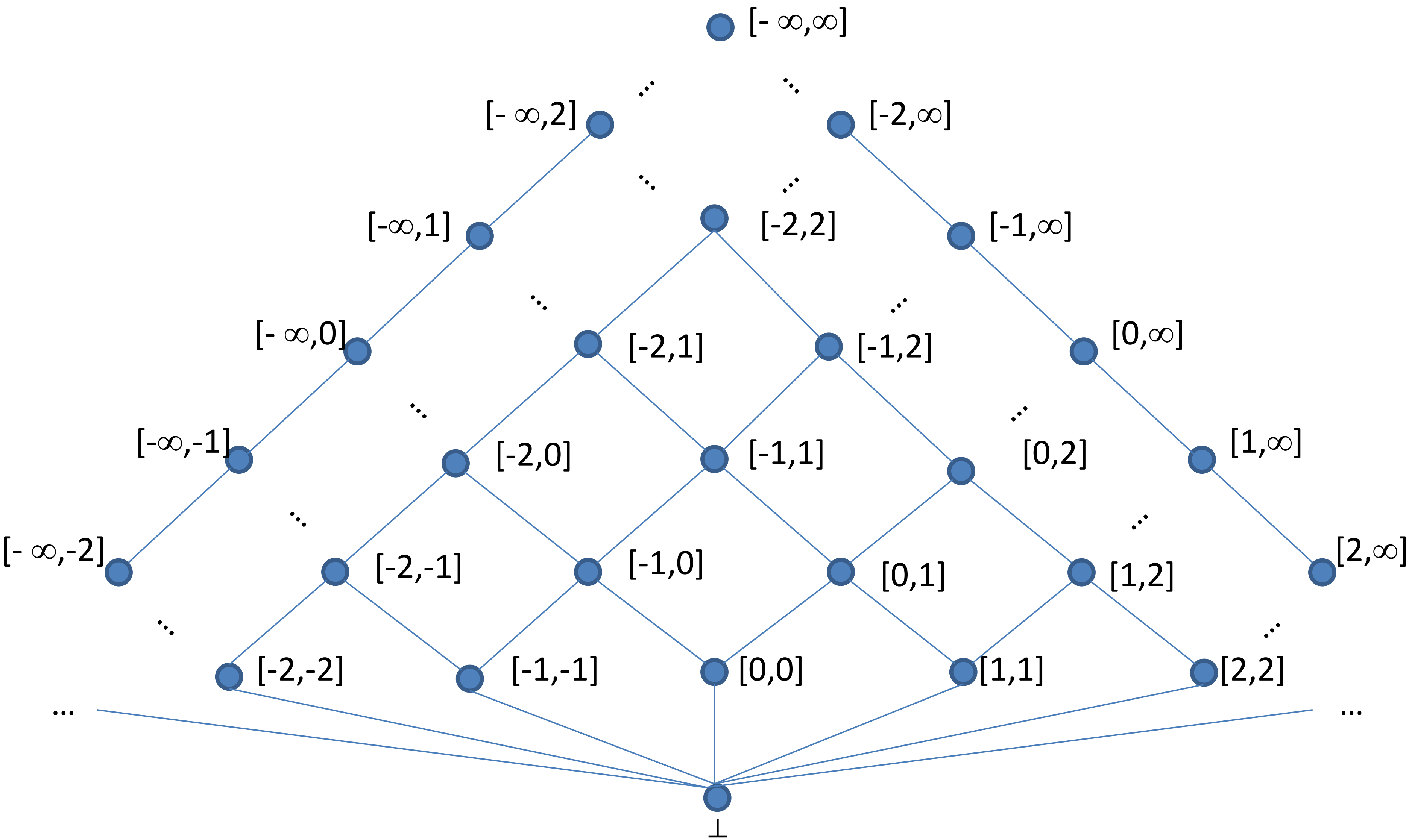
$$P_{sign} \not\models (0 \leq y - x)$$ ✗

sign domain too imprecise to prove property

Lets try another abstraction

This time, instead of abstracting variable values using the sign of the variable, we will abstract the values using an interval

Step 1: Select interval domain



Step 1: Select abstract domain

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:   y := y + 1;
5:   i := i - 1;
6:   goto 3;
   }

7: assert 0 ≤ y - x
}
```

An abstract program state now looks like:

pc	x	y	i
2	[−2, ∞]	[1,7]	[1,2]

Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
4	[−2, ∞]	[1,7]	[1,2]

…)

```
4:    y := y + 1;
```

?

Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
4	[-2, ∞]	[1,7]	[1,2]

…)

`4:    y := y + 1;`

why is this correct?

(…

pc	x	y	i
5	[-2, ∞]	[**2**,**8**]	[1,2]

…)

Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
5	[−2, ∞]	[1,7]	[1,2]

…)

↓

`5:    i := i - 1;`

↓

?

Step 2: Define Transformers

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(...

pc	x	y	i
5	[−2, ∞]	[1,7]	[1,2]

...)

↓

`5:    i := i - 1;`

↓

(...

pc	x	y	i
6	[−2, ∞]	[1,7]	[**0**,**1**]

...)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

Again, we start with
the least abstract element

pc	x	y	i
1	⊥	⊥	⊥

pc	x	y	i
2	⊥	⊥	⊥

pc	x	y	i
3	⊥	⊥	⊥

pc	x	y	i
4	⊥	⊥	⊥

pc	x	y	i
5	⊥	⊥	⊥

pc	x	y	i
6	⊥	⊥	⊥

pc	x	y	i
7	⊥	⊥	⊥

Lets the iterations begin !

Note:
we only show the change of 1 component

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
1	⊥	⊥	⊥

…)

↓

`int i`

↓

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0)  {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
1	$\perp$	$\perp$	$\perp$

…)

int i

(…

pc	x	y	i
1	$\perp$	$\perp$	$[-\infty, \infty]$

…)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
1	$\perp$	$\perp$	$[-\infty, \infty]$

…)

```
1: int x :=5;
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
1	⊥	⊥	[−∞, ∞]

…)

↓

`1: int x :=5;`

↓

(…

pc	x	y	i
2	[5, 5]	⊥	[−∞, ∞]

…)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
2	[5, 5]	⊥	[−∞, ∞]

…)

⬇

`2:    int y :=7;`

⬇

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

pc	x	y	i
2	[5, 5]	⊥	[−∞, ∞]

```
2:    int y :=7;
```

pc	x	y	i
3	[5, 5]	[7,7]	[−∞, ∞]

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
3	[5, 5]	[7,7]	[−∞, ∞]

…)

↓

`3: if (i ≥ 0)`

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
3	[5, 5]	[7,7]	[-∞, ∞]

…)

↓

`3: if (i ≥ 0)`

↓

(…

pc	x	y	i
4	[5, 5]	[7,7]	[0, ∞]

pc	x	y	i
7	[5, 5]	[7,7]	[-∞, -1]

…)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
4	[5, 5]	[7,7]	[0, ∞]

…)

```
4: y := y + 1;
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
4	[5, 5]	[7,7]	[0, ∞]

…)

`4: y := y + 1;`

(…

pc	x	y	i
5	[5, 5]	[8,8]	[0, ∞]

…)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
5	[5, 5]	[8,8]	[0, ∞]

 …)

```
5: i := i - 1;
```

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
5	[5, 5]	[8,8]	[0, ∞]

…)

`5: i := i - 1;`

(…

pc	x	y	i
6	[5, 5]	[8,8]	[-1, ∞]

…)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
6	[5, 5]	[8,8]	[-1, ∞]

…)

↓

`6: goto 3;`

↓

(…

pc	x	y	i
3	[5, 5]	[8,8]	[-1, ∞]

…)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

pc	x	y	i
3	[5, 5]	[8,8]	[-1, ∞]

pc	x	y	i
3	[5, 5]	[7,7]	[-∞, ∞]

join

What is going on here ?

(…

pc	x	y	i
3	[5, 5]	[7,8]	[-∞, ∞]

…)

Joins

When we have two abstract elements A and B, we can join them to produce their (least) upper bound

denoted by: $A \sqcup B$

we have that $A \sqsubseteq A \sqcup B$ and $B \sqsubseteq A \sqcup B$

$D \sqsubseteq E$ means that E is more abstract than D

In our example, we join the abstract states that occur at the same program label

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
3	[5, 5]	[7,8]	[-∞, ∞]

…)

`3: if (i ≥ 0)`

(…

pc	x	y	i
4	[5, 5]	[7,8]	[0, ∞]

pc	x	y	i
7	[5, 5]	[7,8]	[-∞, -1]

…)

Step 3: Iterate to a fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

(…

pc	x	y	i
4	[5, 5]	[7,8]	[0, ∞]

…)

`4: y := y + 1`

This will never terminate !

`y` will keep increasing forever !

But states reaching label `7` will always satisfy the assertion !

Cannot reach a Fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

With the interval abstraction we could not reach a fixed point.

The domain has infinite height.

What should we do ?

Introduce a special operator called the widening operator !

It ensures termination at the expense of precision

Widening instead of Join

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

pc	x	y	i
3	[5, 5]	[7,7]	[−∞, ∞]

pc	x	y	i
3	[5, 5]	[8,8]	[0, ∞]

widen

y is increasing, go directly to ∞

(… …)

pc	x	y	i
3	[5, 5]	[7,∞]	[−∞, ∞]

Fixed Point after Widening

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }

7: assert 0 ≤ y - x
}
```

With widening, our iteration now reaches a fixed point, where at label 7 we have:

(…

pc	x	y	i
7	[5, 5]	[7, ∞]	[−∞, −1]

…)

$$P \models (0 \leq y - x)$$ ✓

$$P_{\text{Interval}} \models (0 \leq y - x)$$ ✓

interval domain precise enough

to prove property !

Questions that should bother you

- What are we abstracting exactly?
- What are abstract domains mathematically?
- How do we discover best/sound abstract transformers? What does best mean?
- What is the function that we iterate to a fixed point?
- How do we ensure termination of the analysis?