Software Architecture and Engineering: Part II

ETH Zurich, Spring 2017
Prof. Martin Vechev

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

Static Analysis
- SMT solver
- Alias Analysis
- Relational Analysis
- Interval Analysis
- Semantics & Theory

Dynamic Analysis
- Framework
- Assertions
- Second Project
- Assertions
- Context Bounded
- Race Detection
- Web & Mobile Apps

Symbolic Reasoning
- Symbolic Execution
- Concolic Execution
- Symbolic Execution
- Program Repair

Today
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

Answer:
No. The problem is undecidable

Alan Turing
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
  unknown if the property holds or not
• “No”, then it is certain that the property does not hold

?
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

- over-approximation (e.g. static analysis)
- under-approximation (e.g. dynamic analysis)
- over and under approximation (e.g. symbolic execution)

All behaviors in the universe
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 \geq 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 \geq 0`
  - `-n+r+10 = 0; n-101 \geq 0`
  - `-n+100 \geq 0`
  - `-n+t1-11 = 0; -n+100 \geq 0`
  - `-n+t2-1 \geq 0; t2-91 \geq 0`
  - `-n+t1-11=0; -n+100 \geq 0; -n+t2-1 \geq 0; t2-91 \geq 0; r-t2+10 \geq 0; r-91 \geq 0`
  - `-n+r+10 \geq 0; r-91 \geq 0`
  - `-a+b+10 \geq 0; b-91 \geq 0`

*Automatically* 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:

- Unlocked
- Locked
- Error

Diagram:
```
Unlocked \(\xrightarrow{\text{lock}}\) Locked
\(\xleftarrow{\text{unlock}}\) Error
\(\xrightarrow{\text{unlock}}\) Unlocked
```

Property we want to check:
```
Network driver: does it do what it is supposed to?```
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();
}
```

```java
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:

- Automatically verified using SLAM based on predicate abstraction
Real-World Program Analyzers (small sample)

A significantly increased interest in the last few years
Facebook: last 2 years...

**INFER:** [http://fbinfer.com/](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/](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...

```plaintext
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
}
```

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.
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
}
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

```plaintext
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
}
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>-</td>
<td>T</td>
</tr>
</tbody>
</table>
```

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
}
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
}
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

This abstract state:

\[
\text{foo (int i) } \{
\begin{array}{l}
1: \text{int } x := 5; \\
2: \text{int } y := 7; \\
3: \text{if } (i \geq 0) \{ \\
4: \quad y := y + 1; \\
5: \quad i := i - 1; \\
6: \quad \text{goto 3;}
\end{array}
\}
\]

\[
\text{assert } 0 \leq x + y
\]

represents infinitely many concrete states including:

\[
\begin{array}{cccc}
\text{pc} & x & y & i \\
4 & T & - & T
\end{array}
\]

If we perform \( y := y + 1 \) on this concrete state, we get:

\[
\begin{array}{cccc}
\text{pc} & x & y & i \\
4 & 1 & -3 & 2
\end{array}
\]

However, the abstract transformer produced an abstract state:

\[
\begin{array}{cccc}
\text{pc} & x & y & i \\
5 & 1 & -2 & 2
\end{array}
\]

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

The abstract transformer is unsound!

\[\frownie\]
How about this?

```c
foo (int i) {
    int x := 5;
    int y := 7;
    if (i >= 0) {
        y := y + 1;
        i := i - 1;
        goto 3;
    }
    assert 0 <= x + y
}
```

This is correct, why?
How about this?

```plaintext
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
}
```

Is this sound? Yes
Is it precise? No

```
1 T 0 T
3 0 T T
4 T 0 T
7 T - T
```
Imprecise Transformer

This abstract state:

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>T</td>
<td>0</td>
<td>T</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

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
}
How about this?

```
foo (int i) {
    1: int x := 5;
    2: int y := 7;
    3: if (i \geq 0) {
        4:   y := y + 1;
        5:   i := i - 1;
        6:   goto 3;
    }
    7: assert 0 \leq x + y
}
```

Is this sound? Yes
Is it precise? Yes
It is easy to be sound and imprecise: simply output $T$.

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

Start with the least abstract element

foo (int i) {
    int x := 5;
    int y := 7;
    if (i ≥ 0) {
        y := y + 1;
        i := i - 1;
        goto 3;
    }
    assert 0 ≤ x + y
}

Lets do some iterations...
Step 3: Iterate to a fixed point

```c
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 3: Iterate to a fixed point

```c
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 3: Iterate to a fixed point

```plaintext
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 3: Iterate to a fixed point

```plaintext
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 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
}
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
}
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
}
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
}
```
Step 3: Iterate to a fixed point

```c
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
}
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>
```

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
}
Step 3: Iterate to a fixed point

```c
foo (int i) {
    int x := 5;
    int y := 7;
    if (i ≥ 0) {
        y := y + 1;
        i := i - 1;
        goto 3;
    }
    assert 0 ≤ x + y
}
```
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
}
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 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
}
Step 3: Iterate to a fixed point

```plaintext
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
}
```

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
}
Let's change the property

```c
foo (int i) {
    1: int x := 5;
    2: int y := 7;
    3: if (i \geq 0) {
        4: y := y + 1;
        5: i := i - 1;
        6: goto 3;
    }
    7: assert 0 \leq x - y
}
```

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>+</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

\[ P \neq (0 \leq x - y) \]

\[ P_{\text{sign}} \neq (0 \leq x - y) \]

\textit{sign domain is sound}: property does not hold and it confirms it
```
foo (int i) {
    int x := 5;
    int y := 7;
    if (i ≥ 0) {
        y := y + 1;
        i := i - 1;
        goto 3;
    }
    assert 0 ≤ y - x
}
```

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>⊥</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>+</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>+</td>
<td>+ T</td>
</tr>
<tr>
<td>7</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

\[ P \equiv (0 \leq y - x) \]

\[ P_{\text{sign}} \not\equiv (0 \leq y - x) \]

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:

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>[-2, ∞]</td>
<td>[1,7]</td>
<td>[1,2]</td>
</tr>
</tbody>
</table>
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
}
```

```markdown
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[-2, ∞]</td>
<td>[1,7]</td>
<td>[1,2]</td>
</tr>
</tbody>
</table>
```

4: \[ y := y + 1; \]
Step 2: Define Transformers

```plaintext
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
}
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[-2, ∞]</td>
<td>[1,7]</td>
<td>[1,2]</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>[-2, ∞]</td>
<td>[2,8]</td>
<td>[1,2]</td>
</tr>
</tbody>
</table>
```
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
}
Step 2: Define Transformers

```c
foo (int i) {
    int x := 5;
    int y := 7;
    if (i >= 0) {
      y := y + 1;
      i := i - 1;
      goto 3;
    }
    assert 0 <= y - x
}
```

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>[-2, ∞]</td>
<td>[1,7]</td>
<td>[1,2]</td>
</tr>
</tbody>
</table>

5: i := i - 1;

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>[-2, ∞]</td>
<td>[1,7]</td>
<td>[0,1]</td>
</tr>
</tbody>
</table>
Step 3: Iterate to a fixed point

Again, we start with the least abstract element

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
}
Step 3: Iterate to a fixed point

```c
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
}
```

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

int i
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
}
Step 3: Iterate to a fixed point

```plaintext
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
}
```
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
}
```
Step 3: Iterate to a fixed point

```plaintext
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
}
```

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>[5, 5]</td>
<td>⊥</td>
<td>[−∞, ∞]</td>
</tr>
</tbody>
</table>

2: int y := 7;
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
}
Step 3: Iterate to a fixed point

```c
foo (int i) {
    int x := 5;
    int y := 7;

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

    assert 0 ≤ y - x
}
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5,5]</td>
<td>[7,7]</td>
<td>[−∞, ∞]</td>
</tr>
</tbody>
</table>
```

3: if (i ≥ 0)
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
}
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
}
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
}
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
}
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
}
```

```
\[
\begin{array}{|c|c|c|c|}
\hline
pc & x & y & i \\
\hline
5 & [5, 5] & [8, 8] & [0, ∞] \\
\hline
\hline
\end{array}
\]
```
Step 3: Iterate to a fixed point

```c
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
}
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5, 5]</td>
<td>[8, 8]</td>
<td>[-1, ∞]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 5]</td>
<td>[8, 8]</td>
<td>[-1, ∞]</td>
</tr>
</tbody>
</table>
```

6: goto 3;

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5, 5]</td>
<td>[8, 8]</td>
<td>[-1, ∞]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 5]</td>
<td>[8, 8]</td>
<td>[-1, ∞]</td>
</tr>
</tbody>
</table>
```

...
Step 3: Iterate to a fixed point

```
foo (int i) {
    int x := 5;
    int y := 7;
    if (i ≥ 0) {
        y := y + 1;
        i := i - 1;
        goto 3;
    }
    assert 0 ≤ y - x
}
```

What is going on here?

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5, 5]</td>
<td>[8, 8]</td>
<td>[-1, ∞]</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5, 5]</td>
<td>[7, 7]</td>
<td>[-∞, ∞]</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5, 5]</td>
<td>[7, 8]</td>
<td>[-∞, ∞]</td>
</tr>
</tbody>
</table>
```
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
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
}
Step 3: Iterate to a fixed point

```c
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
}
```

This will never terminate!

\[ y \text{ will keep increasing forever!} \]

But states reaching label 7 will always satisfy the assertion!
Cannot reach a Fixed point

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

```c
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
}
```
Widening instead of Join

```plaintext
foo (int i) {
  1: int x := 5;
  2: int y := 7;
  3: if (i ≥ 0) {
        y := y + 1;
        i := i - 1;
  6:   goto 3;
  7:   assert 0 ≤ y - x
}

y is increasing, go directly to ∞
```

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5,5]</td>
<td>[7,7]</td>
<td>(−∞, ∞)</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5,5]</td>
<td>[8,8]</td>
<td>[0, ∞]</td>
</tr>
</tbody>
</table>

widen

(···

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[5,5]</td>
<td>[7,∞]</td>
<td>(−∞, ∞)</td>
</tr>
</tbody>
</table>
Fixed Point after Widening

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

\[
\begin{array}{|c|c|c|c|}
\hline
\text{pc} & \text{x} & \text{y} & \text{i} \\
\hline
7 & [5, 5] & [7, \infty] & [-\infty, -1] \\
\hline
\end{array}
\]

\[P = (0 \leq y - x) \quad \checkmark \]

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?