CS 526
Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526
ABSTRACT INTERPRETATION

The slides adapted from Martin Vechev
So Far...

Dataflow Analysis

Lattice Theory & Abstract Domains

Abstraction, MOP, MFP Solutions

• What is the relationship between concrete program state and the analysis?
• How to build analyses in a rigorous manner?
MEANING OF DATAFLOW ANALYSIS (REVIEW)
Meaning of Dataflow Results

Concept of program state \( s \) for control-flow graphs

- Program point \( n \) where execution located
  (\( n \) is node that will execute next)
- Values of variables in program

Each execution generates a trajectory of states:

- \( s_0; s_1; \ldots; s_k \), where each \( s_i \in ST \)
- \( s_{i+1} \) generated from \( s_i \) by executing basic block to
  - Update variable values
  - Obtain new program point \( n \)
Relating States to Analysis Result

• Meaning of analysis results is given by an abstraction function \( AF : ST \rightarrow P \)

• Correctness condition: require that for all states \( s \)
  \[ AF(s) \leq \text{in}_n \]
  where \( n \) is the next statement to execute in state \( s \)
Sign Analysis Example

Sign analysis - compute sign of each variable $v$

Base Lattice: $P = \text{flat lattice on\{-,0,+\}}$

Actual lattice records a value for each variable $v$

• Example element: $[a\rightarrow+, b\rightarrow0, c\rightarrow-]$
Interpretation of Lattice Values

If value of $v$ in lattice is:

- **BOT**: no information about sign of $v$
- **-**: variable $v$ is negative
- **0**: variable $v$ is 0
- **+**: variable $v$ is positive
- **TOP**: $v$ may be positive or negative

What is **abstraction function** $AF$?

- $AF([x_1, \ldots, x_n]) = [\text{sign}(x_1), \ldots, \text{sign}(x_n)]$
- Where $\text{sign}(x) = 0$ if $x = 0$, $+$ if $x > 0$, $-$ if $x < 0$
### Operation $\otimes$ on Lattice

<table>
<thead>
<tr>
<th>$\otimes$</th>
<th>BOT</th>
<th>-</th>
<th>0</th>
<th>+</th>
<th>TOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOT</td>
<td>BOT</td>
<td>BOT</td>
<td>0</td>
<td>BOT</td>
<td>BOT</td>
</tr>
<tr>
<td>-</td>
<td>BOT</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>TOP</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>BOT</td>
<td>-</td>
<td>0</td>
<td>+</td>
<td>TOP</td>
</tr>
<tr>
<td>TOP</td>
<td>BOT</td>
<td>TOP</td>
<td>0</td>
<td>TOP</td>
<td>TOP</td>
</tr>
</tbody>
</table>
Transfer Functions

If $n$ of the form $v = c$

- $f_n(x) = x[v \rightarrow +]$ if $c$ is positive
- $f_n(x) = x[v \rightarrow 0]$ if $c$ is 0
- $f_n(x) = x[v \rightarrow -]$ if $c$ is negative

If $n$ of the form $v_1 = v_2 \cdot v_3$

- $f_n(x) = x[v_1 \rightarrow x[v_2] \otimes x[v_3]]$

$I = \text{TOP}$

(unfinished variables may have any sign)
Example

\[ a = 1 \]

\[ b = -1 \quad b = 1 \]

\[ c = a \times b \]
Imprecision In Example

Abstraction Imprecision:

\[ [a \rightarrow 1] \text{ abstracted as } [a \rightarrow +] \]

\[ \text{a = 1} \]

\[ \text{b = -1} \quad \text{b = 1} \]

Control Flow Imprecision:

\[ c = a * b \]

\[ [b \rightarrow \text{TOP}] \text{ summarizes results of all executions. In any execution state } s, \text{ AF}(s)[b] \neq \text{TOP} \]
General Sources of Imprecision

**Abstraction Imprecision**
- Concrete values (integers) abstracted as lattice values (-, 0, and +)
- Lattice values less precise than execution values
- Abstraction function throws away information

**Control Flow Imprecision**
- One lattice value for all possible control flow paths
- Analysis result has a single lattice value to summarize results of multiple concrete executions
- Join operation $\lor$ moves up in lattice to combine values from different execution paths
- Typically if $x \leq y$, then $x$ is more precise than $y$
Why To Allow Imprecision?

Make analysis tractable

Unbounded sets of values in execution
  • Typically abstracted by finite set of lattice values

Execution may visit unbounded set of states
  • Abstracted by computing joins of different paths
Intuition Behind Abstract Interpretation

Patrick Cousot’s Description:

• [http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html](http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html)

Also, Abstract interpretation vs Dataflow Analysis:

Abstract Interpretation

1. Define abstract domains (that represent important parts of program execution)

2. Define abstraction and concretization functions to relate the abstract domain with the program execution

3. Iterate updating the abstract state until convergence
ABSTRACT DOMAINS
Sign Domain

TOP
- 0 +
BOT
Interval Domain

Each variable takes a value from the following domain (a complete lattice):

Infinite height...
Relational Abstractions

The Interval domain is an example of a non-relational domain. It does not explicitly keep the relationship between variables.

Sometimes, it may be necessary to keep this relationship to be more precise. Octagon and Polyhedra domains keep the relationship. These domains are called relational domains.
Octagon Domain

Constraints are of the following form:

\[ \pm x \pm y \leq c \]

The slope is fixed.

An abstract state is a map from labels to conjunction of constraints:

\[
\begin{align*}
  x - y &\leq 3 \\
  y &\leq 8 \\
  y &\geq 2 \\
  x + y &\leq 15 \\
  x + y &\geq 5 \\
  x &\geq 1 \\
  x - y &\geq -20 \\
  x &\leq 7 
\end{align*}
\]
Polyhedra Domain

Constraints are of the following form:

\[ c_1 x_1 + c_2 x_2 \ldots + c_n x_n \leq c \]

The slope can vary.

An abstract state is again a map from labels to conjunction of constraints:

\[ x - y \geq -20 \land \]
\[ x - 3 \times y \leq 2 \land \]
\[ x + y \geq 5 \]

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In POPL '78
Some Uses of Numerical Domains

- Out of bounds checks
- Division by zero
- Aliasing (A. Venet, SAS’02)
- Predicate abstraction (P. Cousot, Verification by abstract interpretation, 2003)
- Resource usage (J. Navas et al., ICLP’07).
Previous domains were numerical...

How do we represent program states?
Domain of Program States

Our starting point is a domain where each element of the domain is a set of states. The domain of states is a complete lattice:

\[(\emptyset (\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma)\]

\[\Sigma = \text{Label} \times \text{Store}\]
Domain of Program States

Size of Set:

\[ n > 0 \]

\[ \sum \]

Each element is a finite set of states, e.g., \([P]\)
Program States for Points-To Analysis

// initially x = z = p = q = null
for (i = 0; i < 2; i++) {
    // allocate O₁, O₂
    A: x := newObject T₁;
    if (i == 0)
        p := x;
    else
        z := x;
}
// allocate O₃
B: x := newObject T₁;
    z.f := x;
// allocate O₄
C: q := newObject T₁;
x := null;

Abstraction:
{ pointer → {Allocation Sites}, … }
e.g. { p → {A}, x → {A}, z → {A} }

Concretization:
{ pointer → {Objects allocated} } …
e.g., { p → {O₁, O₂},
        x → {O₁, O₂},
        z → {O₁, O₂} }

Details: http://www.srl.inf.ethz.ch/sae2014/Pointer.pdf
Program States for Points-To Analysis

// initially x = z = p = q = null
for (i = 0; i < 2; i++) {
  // allocate O₁, O₂
  A: x := newObjObject T1;
  if (i == 0)
    p := x;
  else
    z := x;
}
// allocate O₃
B: x := newObjObject T1;
  z.f := x;
// allocate O₄
C: q := newObjObject T1;
  x := null;

The result of pointer analysis
at the fixed point:

- p \mapsto \emptyset, q \mapsto \emptyset, x \mapsto \emptyset, z \mapsto \emptyset
- p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{A\}, z \mapsto \{A\}
- p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{A\}, z \mapsto \{A\}
- p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{A\}, z \mapsto \{A\}
- p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{A\}, z \mapsto \{A\}
- p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{B\}, z \mapsto \{A\}
- p \mapsto \{A\}, q \mapsto \emptyset, x \mapsto \{B\}, z \mapsto \{A\}, A.f \mapsto \{B\}
- p \mapsto \{A\}, q \mapsto \{C\}, x \mapsto \{\}, z \mapsto \{A\}, A.f \mapsto \{B\}

The Art of (Sound) Approximation: Static Program Analysis

- Define a function $F^\#$ such that $F^\#$ approximates $F$. This is typically done manually and can be tricky but is done once and for a programming language.

- Then, use existing theorems which state that the least fixed point of $F^\#$, e.g. some $V$, approximates the least fixed point of $F$, e.g. $[P]$.

- Finally, automatically compute a fixed point of $F^\#$, that is a $V$ where $F^\#(V) = V$. 


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
FUNCTION APPROXIMATION
Approximating a Function

Given functions:

\[ F: C \rightarrow C \]
\[ F\#: C \rightarrow C \]

what does it mean for \( F\# \) to approximate \( F \)?

\[ \forall x \in C : F(x) \subseteq_c F\#(x) \]
Approximating a Function

What about when:

\[ F: C \to C \]
\[ F^\#: A \to A \]

We need to connect the concrete \( C \) and the abstract \( A \)

We will connect them via two functions \( \alpha \) and \( \gamma \)

\[ \alpha : C \to A \] is called the \textit{abstraction} function

\[ \gamma : A \to C \] is called the \textit{concretization} function
Connecting Concrete with Abstract

\[(C, \sqsubseteq_c) \rightarrow (A, \sqsubseteq_A)\]

\[\alpha\]
Approximating Function: Definition 1

So we have the 2 functions:

\[ F: \mathbb{C} \rightarrow \mathbb{C} \]
\[ F^\#: \mathbb{A} \rightarrow \mathbb{A} \]

If we know that \( \alpha \) and \( \gamma \) form a **Galois Connection**, then we can use the following definition of approximation:

\[ \forall z \in \mathbb{A} : \alpha(F(\gamma(z))) \subseteq_{A} F^\#(z) \]
Galois Connection

For the course, it is not important to know what Galois Connections are.

The only point to keep in mind that is that they place some restrictions on $\alpha$ and $\gamma$.

- For instance, they require $\alpha$ to be monotone.
Visualizing Definition 1

(C, \subseteq_c) \rightarrow (A, \subseteq_A)

F(x) \rightarrow F#(z)

\alpha \quad \gamma
Approximating a Function

This equation

\[ \forall z \in A : \alpha( F( \gamma(z)) ) \supseteq_A F^\#(z) \]

says that

- if we have some function in the abstract that we think **should approximate** the concrete function,
- then to check that this is indeed true, we need to prove
- that for any abstract element, (1) concretizing it, (2) applying the concrete function and (3) abstracting back again is **less than** applying the function in the abstract directly.
Least precise approximation

To approximate $F$, we can always define $F^\#(z) = T$

This solution is always sound as: $\forall z \in A : \alpha(F(\gamma(z))) \subseteq_A T$

However, it is not practically useful as it is too imprecise
Most precise approximation

\[ F^\#(z) = \alpha(F(\gamma(z))) \] is the best abstract function.

But, we often **cannot implement** best \( F^\#(z) \) algorithmically.

However, we can come up with a \( F^\#(z) \) that has the same behavior as \( \alpha(F(\gamma(z))) \) but a different implementation.

Any such \( F^\#(z) \) is referred to as the best transformer.
Key Theorem 1: Least Fixed Point Approximation

If we have:

1. **monotone** functions $F: C \rightarrow C$ and $F^\#: A \rightarrow A$
2. $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ forming a Galois Connection
3. $\forall z \in A : \alpha(F(\gamma(z))) \subseteq_A F^#(z)$ (that is, $F^#$ approximates $F$)

then:

$$\alpha(\text{lfp}(F)) \subseteq_A \text{lfp}(F^#)$$

This is important as it goes from **local** function approximation to **global** approximation. This is a key theorem in program analysis.
Least Fixed Point Approximation

The 3 premises to the theorem are usually proved manually.

Once proved, we can now automatically compute a least fixed point in the abstract and be sure that our result is sound!
So what is F\# then?

F\# is to be defined for the particular abstract domain $A$ that we work with. The domain $A$ can be Sign, Parity, Interval, Octagon, Polyhedra, and so on.

In our setting and commonly, we simply keep a map from every label (program counter) in the program to an abstract element in $A$.

Then F\# simply updates the mapping from labels to abstract elements.
**Cheat Sheet: Connecting Math and Analysis**

<table>
<thead>
<tr>
<th>Mathematical Concept</th>
<th>Use in Static Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete Lattice</td>
<td>Defines Abstract Domain and ensure joins exist.</td>
</tr>
<tr>
<td>Joins (⊔)</td>
<td>Combines facts arriving at a program point</td>
</tr>
<tr>
<td>Bottom (⊥)</td>
<td>Used for initialization of all but initial elements</td>
</tr>
<tr>
<td>Top (T)</td>
<td>Used for initialization of initial elements</td>
</tr>
<tr>
<td>Widening (∨)</td>
<td>Used to guarantee analysis termination</td>
</tr>
<tr>
<td>Function Approximation</td>
<td>Critical to make sure abstract semantics approximate the concrete semantics</td>
</tr>
<tr>
<td>Fixed Points</td>
<td>This is what is computed by the analysis</td>
</tr>
<tr>
<td>Tarski’s Theorem</td>
<td>Ensures fixed points exist.</td>
</tr>
</tbody>
</table>
Checkpoint

So far, we have seen a bunch of mathematical concepts such as lattices, functions, fixed points, function approximation, etc.

Next, we will see how to put these together in order to build static analyzers.
Domain of Program States

Our starting point is a domain where each element of the domain is a set of states. The domain of states is a complete lattice:

\[(\emptyset (\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma)\]

\[\Sigma = \text{Label} \times \text{Store}\]
Starting Point: Domain of States

Size of Set:

\( n > 0 \)

Each element is a finite set of states, e.g., \([P]\)
Let $[[P]]$ be the set of reachable states of a program $P$.

**Def.** Let function $F$ be (where $I$ is an initial set of states):

$$F(S) = I \cup \{ c' \mid c \in S \land c \rightarrow c' \}$$

Then, $[[P]]$ is a **fixed point** of $F$: i.e., $F([[P]]) = [[P]]$

(in fact, $[[P]]$ is the least fixed point of $F$)
Starting Point: Domain of States

Size of Set:

\[ n > 0 \]

Each element is a finite set of states, e.g., \([P]\)

Static analysis computes overapproximation of \([P]\)

\[
\sum
\]

\[
\emptyset
\]

\[
\{1, \{x \mapsto \emptyset, y \mapsto \emptyset, z \mapsto \emptyset\}\} \cup \{1, \{x \mapsto 2, y \mapsto 4, z \mapsto 1\}\}
\]

\[
\{1, \{x \mapsto 2, y \mapsto 4, z \mapsto 1\}\}
\]
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
Abstract Interpretation: Step 1

1. select/define an abstract domain
   • selected based on the type of properties you want to prove
Interval Domain

If we are interested in properties that involve the range of values that a variable can take, we can abstract the set of states into a map which captures the range of values that a variable can take.
Interval Domain

Each variable takes a value from the following domain (a complete lattice):
Interval Domain: Let's Define it

Let the interval domain **on integers** be a lattice: \((L, \subseteq_i, \cup_i, \cap_i, \perp_i, [-\infty, \infty])\)

We denote \(Z^\infty = \mathbb{Z} \cup \{-\infty, \infty\}\)
The set \(L = \{[x,y] \mid x, y \in \mathbb{Z}^\infty, x \leq y\} \cup \{\perp_i\}\)

For a set \(S \subseteq \mathbb{Z}^\infty\), \(\min(S)\) returns the minimum number in \(S\), \(\max(S)\) returns the maximum number in \(S\).

Operations \((\subseteq_i, \cup_i, \cap_i)\):
- \([a,b] \subseteq_i [c,d]\) if \(c \leq a\) and \(b \leq d\)
- \([a,b] \cup_i [c,d] = [\min(a,c), \max(b,d)]\)
- \([a,b] \cap_i [c,d] = \text{meet}(\max(a,c), \min(b,d))\)
  where \(\text{meet}(x,y)\) returns \([x,y]\) if \(x \leq y\) and \(\perp_i\) otherwise
The $L^i$ domain simply defines intervals, but to apply it to programs we need to take into account program labels (program counters) and program variables. Therefore, for programs, we use the domain $\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)$

That is, at each label and for each variable, we will keep the range for that variable. This domain is also a complete lattice.

The operators of $L^i \subseteq_i, \cup_i, \cap_i$ are lifted directly to both domains:
• $\text{Var} \rightarrow L^i$
• $\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)$
Intervals: Applied to Programs

\[ \alpha^i: \emptyset(\Sigma) \rightarrow (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)) \]

\[ \gamma^i: (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)) \rightarrow \emptyset(\Sigma) \]

Using \( \alpha^i \), we abstract a set of states into a map from program labels to interval ranges for each variable.

Using \( \gamma^i \), we concretize the intervals maps to a set of states.
Example of Abstraction and Concretization

\[ \alpha_i \left( \{ \langle 1, \{x \mapsto 1, y \mapsto 9, q \mapsto -2 \rangle, \langle 1, \{x \mapsto 5, y \mapsto 9, q \mapsto -2 \rangle, \langle 1, \{x \mapsto 8, y \mapsto 9, q \mapsto -2 \rangle, \langle 1, \{x \mapsto 1, y \mapsto 9, q \mapsto 4 \rangle, \langle 1, \{x \mapsto 5, y \mapsto 9, q \mapsto 4 \rangle, \langle 1, \{x \mapsto 8, y \mapsto 9, q \mapsto 4 \rangle \} \right) \right) \]

\[ = 1 \rightarrow (x \mapsto [1,8], y \mapsto [9,9], q \mapsto [-2,4]) \]

\[ \gamma_i (1 \rightarrow (x \mapsto [1,8], y \mapsto [9,9], q \mapsto [-2,4])) \]

\[ = \{ \langle 1, \{x \mapsto 1, y \mapsto 9, q \mapsto -2 \rangle, \langle 1, \{x \mapsto 5, y \mapsto 9, q \mapsto -2 \rangle, \langle 1, \{x \mapsto 8, y \mapsto 9, q \mapsto -2 \rangle, \langle 1, \{x \mapsto 1, y \mapsto 9, q \mapsto 4 \rangle, \langle 1, \{x \mapsto 5, y \mapsto 9, q \mapsto 4 \rangle, \langle 1, \{x \mapsto 8, y \mapsto 9, q \mapsto 4 \rangle, \langle 1, \{x \mapsto 7, y \mapsto 9, q \mapsto 3 \rangle, \langle 1, \{x \mapsto 3, y \mapsto 9, q \mapsto 4 \rangle, \langle 1, \{x \mapsto 1, y \mapsto 9, q \mapsto -1 \rangle, \ldots, \ldots, \ldots \} \}

Concretization includes many more states (in red) than what was abstracted...
Abstract Interpretation: Step 2

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
we still need to actually compute $\alpha_i \{[P]\}$
(or an over-approximation of it)
We need to approximate $F$

We want a function $F^i$ where:

$$F^i : (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)) \rightarrow (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i))$$

such that:

$$\alpha^i (\text{Ifp } F) \subseteq \text{Ifp } F^i$$
Here is a definition of $F^i$ which approximates the best transformer but works only on the abstract domain:

$$F^i : (\text{Lab} \to (\text{Var} \to L^i)) \to (\text{Lab} \to (\text{Var} \to L^i))$$

$$F^i(m) \ell = \begin{cases} 
\lambda v. [-\infty, \infty] & \text{if } \ell \text{ is initial label} \\
\bigsqcup \llbracket \text{action} \rrbracket_i(m(\ell')) & \text{otherwise}
\end{cases}$$

$$\llbracket \text{action} \rrbracket_i : (\text{Var} \to L^i) \to (\text{Var} \to L^i)$$
What is \((\ell', \text{action, } \ell)\) ?

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

**Actions:**

- (1, \(x := 5\), 2)
- (2, \(y := 7\), 3)
- (3, \(0 ≤ i\), 4)
- (3, \(0 > i\), 7)
- (4, \(y = y + 1\), 5)
- (5, \(i := i - 1\), 6)
- (6, \(\text{goto 3}\), 3)

Multiple (two) actions reach label 3
What is \((\ell', \text{action}, \ell)\) ?

An action can be:

- \(b \in \text{BExp}\) boolean expression in a conditional
- \(x := a\) here, \(a \in \text{AExp}\)
- skip

Next, we will define the effect of some of these things formally, while with others we will proceed by example.

The key point is to make sure that \(\llbracket \text{action} \rrbracket_i\) produces sound and precise results.
foo (int i) {
    int x := 5;
    int y := 7;
    if (i ≥ 0) {
        y := y + 1;
        i := i - 1;
        goto 3;
    }
}
\[ [x := a]_i \]

\[ [x := a]_i (m) = m [x \leftrightarrow v], \text{ where } \langle a, m \rangle \Downarrow_i v \]

\[ \langle a, m \rangle \Downarrow_i v \] says that given a map \( m \), the expression \( a \) evaluates to a value \( v \in L^i \)

The operational semantics rules for expression evaluation:

- any constant \( Z \) is abstracted to an element in \( L^i \)
- operators +, -, and * are re-defined for the Interval domain
If we add $\bot_i$ to any other element, we get $\bot_i$.

If both operands are not $\bot_i$, we get:

$$\begin{pmatrix} x, y \end{pmatrix} + \begin{pmatrix} z, q \end{pmatrix} = \begin{pmatrix} x + z, y + q \end{pmatrix}$$

what about $\ast$?

is $\begin{pmatrix} x, y \end{pmatrix} \ast \begin{pmatrix} z, q \end{pmatrix} = \begin{pmatrix} x \ast z, y \ast q \end{pmatrix}$ sound?
Let us first look at the expression: $a_1 \ c \ a_2$

For a map $m$, we need to define: $\llbracket a_1 \ c \ a_2 \rrbracket_i(m)$

which produces another map as a result.

Here, $c$ is a condition such as: $\leq, =, <$
What is $[x \leq y]$?

Easy case: $x_{\text{max}} \leq y_{\text{min}}$
- We simply

Suppose we have the program:

```c
// Here, x is [0,4] and y is [3,5]
if (x \leq y){
    1: ... // x? y?
}
```

What are the intervals for x and y at label 1?
Evaluating $[b]_i$

$[b_1 \lor b_2]_i(m) = [b_1]_i(m) \cup [b_2]_i(m)$

$[b_1 \land b_2]_i(m) = [b_1]_i(m) \cap [b_2]_i(m)$
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:
}
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
foo (int i) {
    int x := 5;
    int y := 7;
    if (i ≥ 0) {
        y := y + 1;
        i := i - 1;
        goto 3;
    }
    goto 7;
}

Fi(m)1 = \lambda \ v. [-\infty, \infty]
Fi(m)2 = \llbracket x := 5 \rrbracket_i (m(1))
Fi(m)3 = \llbracket y:= 7 \rrbracket_i (m(2)) \sqcup \llbracket \text{goto 3} \rrbracket_i (m(6))
Fi(m)4 = \llbracket i ≥ 0 \rrbracket_i (m(3))
Fi(m)5 = \llbracket y := y + 1 \rrbracket_i (m(4))
Fi(m)6 = \llbracket i := i - 1 \rrbracket_i (m(5))
Fi(m)7 = \llbracket i < 0 \rrbracket_i (m(3))
Let us compute the least fixed point of $F^i$
The collection of these lines denote the current iterate. The iterate is a map

```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: }
```

1: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
2: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
3: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
4: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
5: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
6: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
7: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
foo (int i) {
    int x := 5;
    int y := 7;
    if (i ≥ 0) {
        y := y + 1;
        i := i - 1;
        goto 3;
    }
}

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → ⊥i, y → ⊥i, i → ⊥i
3: x → ⊥i, y → ⊥i, i → ⊥i
4: x → ⊥i, y → ⊥i, i → ⊥i
5: x → ⊥i, y → ⊥i, i → ⊥i
6: x → ⊥i, y → ⊥i, i → ⊥i
7: x → ⊥i, y → ⊥i, i → ⊥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;
3: }
7: }

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → ⊥_i, y → ⊥_i, i → ⊥_i
4: x → ⊥_i, y → ⊥_i, i → ⊥_i
5: x → ⊥_i, y → ⊥_i, i → ⊥_i
6: x → ⊥_i, y → ⊥_i, i → ⊥_i
7: x → ⊥_i, y → ⊥_i, i → ⊥_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: }

1: x → [−∞,∞], y → [−∞,∞], i → [−∞,∞]
2: x → [5,5], y → [−∞,∞], i → [−∞,∞]
3: x → [5,5], y → [7,7], i → [−∞,∞]
4: x → ⊥_i, y → ⊥_i, i → ⊥_i
5: x → ⊥_i, y → ⊥_i, i → ⊥_i
6: x → ⊥_i, y → ⊥_i, i → ⊥_i
7: x → ⊥_i, y → ⊥_i, i → ⊥_i
Iterate 4

Notice how we propagated to both labels 4 and 7 at the same time

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: }

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,7], i → [-∞,∞]
4: x → [5,5], y → [7,7], i → [0,∞]
5: x → ⊥_i , y → ⊥_i , i → ⊥_i
6: x → ⊥_i , y → ⊥_i , i → ⊥_i
7: x → [5,5], y → [7,7], i → [-∞, -1]
Iterate 5

```go
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: }
```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[-∞,∞]</td>
<td>[-∞,∞]</td>
<td>[-∞,∞]</td>
</tr>
<tr>
<td>2</td>
<td>[5,5]</td>
<td>[-∞,∞]</td>
<td>[-∞,∞]</td>
</tr>
<tr>
<td>3</td>
<td>[5,5]</td>
<td>[7,7]</td>
<td>[-∞,∞]</td>
</tr>
<tr>
<td>4</td>
<td>[5,5]</td>
<td>[7,7]</td>
<td>[0,∞]</td>
</tr>
<tr>
<td>5</td>
<td>[5,5]</td>
<td>[8,8]</td>
<td>[0,∞]</td>
</tr>
<tr>
<td>6</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>7</td>
<td>[5,5]</td>
<td>[7,7]</td>
<td>[−∞,−1]</td>
</tr>
</tbody>
</table>
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:
}
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: }

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,8], i → [-∞,∞]
4: x → [5,5], y → [7,7], i → [0,∞]
5: x → [5,5], y → [8,8], i → [0,∞]
6: x → [5,5], y → [8,8], i → [-1,∞]
7: x → [5,5], y → [7,7], i → [-∞, -1]
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: }

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,8], i → [-∞,∞]
4: x → [5,5], y → [7,8], i → [0,∞]
5: x → [5,5], y → [8,8], i → [0,∞]
6: x → [5,5], y → [8,8], i → [-1,∞]
7: x → [5,5], y → [7,8], i → [-∞, -1]
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: }
1: x → [-∞, 8], y → [-∞, 8], i → [-∞, 8]
2: x → [5, 5], y → [-∞, 8], i → [-∞, 8]
3: x → [5, 5], y → [8, 8], i → [-∞, 8]
4: x → [5, 5], y → [7, 8], i → [-∞, 8]
5: x → [5, 5], y → [8, 9], i → [-∞, 8]
6: x → [5, 5], y → [8, 8], i → [-∞, 8]
7: x → [5, 5], y → [7, 8], i → [-∞, -1]
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;
3: }
7: }

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,8], i → [-∞,∞]
4: x → [5,5], y → [7,8], i → [0,∞]
5: x → [5,5], y → [8,9], i → [0,∞]
6: x → [5,5], y → [8,9], i → [-1,∞]
7: x → [5,5], y → [7,8], i → [-∞, -1]
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: }

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,9], i → [-∞,∞]
4: x → [5,5], y → [7,8], i → [0,∞]
5: x → [5,5], y → [8,9], i → [0,∞]
6: x → [5,5], y → [8,9], i → [-1,∞]
7: x → [5,5], y → [7,8], i → [-∞, -1]
Iterate 12

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: }

1: x → [−∞,∞], y → [−∞,∞], i → [−∞,∞]
2: x → [5,5], y → [−∞,∞], i → [−∞,∞]
3: x → [5,5], y → [7,9], i → [−∞,∞]
4: x → [5,5], y → [7,9], i → [0,∞]
5: x → [5,5], y → [8,9], i → [0,∞]
6: x → [5,5], y → [8,9], i → [−1,∞]
7: x → [5,5], y → [7,9], i → [−∞,−1]
Iterate 13

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

1: $x \rightarrow [-\infty,\infty]$, $y \rightarrow [-\infty,\infty]$, $i \rightarrow [-\infty,\infty]$
2: $x \rightarrow [5,5]$, $y \rightarrow [-\infty,\infty]$, $i \rightarrow [-\infty,\infty]$
3: $x \rightarrow [5,5]$, $y \rightarrow [7,9]$, $i \rightarrow [-\infty,\infty]$
4: $x \rightarrow [5,5]$, $y \rightarrow [7,9]$, $i \rightarrow [0,\infty]$
5: $x \rightarrow [5,5]$, $y \rightarrow [8,10]$, $i \rightarrow [0,\infty]$
6: $x \rightarrow [5,5]$, $y \rightarrow [8,9]$, $i \rightarrow [-1,\infty]$
7: $x \rightarrow [5,5]$, $y \rightarrow [7,9]$, $i \rightarrow [-\infty, -1]$
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;
3: }
7: }

1: x → [−∞, ∞], y → [−∞, ∞], i → [−∞, ∞]
2: x → [5, 5], y → [−∞, ∞], i → [−∞, ∞]
3: x → [5, 5], y → [7, 9], i → [−∞, ∞]
4: x → [5, 5], y → [7, 9], i → [0, ∞]
5: x → [5, 5], y → [8, 10], i → [0, ∞]
6: x → [5, 5], y → [8, 10], i → [−1, ∞]
7: x → [5, 5], y → [7, 9], i → [−∞, −1]
Iterate 15

```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: }
```

1: \(x \rightarrow [-\infty, \infty], \ y \rightarrow [-\infty, \infty], \ i \rightarrow [-\infty, \infty]\)
2: \(x \rightarrow [5, 5], \ y \rightarrow [-\infty, \infty], \ i \rightarrow [-\infty, \infty]\)
3: \(x \rightarrow [5, 5], \ y \rightarrow [7, 10], \ i \rightarrow [-\infty, \infty]\)
4: \(x \rightarrow [5, 5], \ y \rightarrow [7, 9], \ i \rightarrow [0, \infty]\)
5: \(x \rightarrow [5, 5], \ y \rightarrow [8, 10], \ i \rightarrow [0, \infty]\)
6: \(x \rightarrow [5, 5], \ y \rightarrow [8, 10], \ i \rightarrow [-1, \infty]\)
7: \(x \rightarrow [5, 5], \ y \rightarrow [7, 9], \ i \rightarrow [-\infty, -1]\)
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: }

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,10], i → [-∞,∞]
4: x → [5,5], y → [7,10], i → [0,∞]
5: x → [5,5], y → [8,10], i → [0,∞]
6: x → [5,5], y → [8,10], i → [-1,∞]
7: x → [5,5], y → [7,10], i → [-∞, -1]
The issue is that the iterates:

\[ F_i^0 (\lambda \ell . \lambda v. \bot_i), F_i^1 (\lambda \ell . \lambda v. \bot_i), F_i^2 (\lambda \ell . \lambda v. \bot_i), \ldots \]

will keep going on forever as the value of variable \( y \) will keep increasing. Hence, we will not be able to compute all of the iterates that we need in order to apply the fixed point theorem.

what should we do in this case?
Generally, if we have a complete lattice \((L, \sqsubseteq, \sqcup, \sqcap)\) and a monotone function \(F\), if the height is infinite or the computation of the iterates of \(F\) takes too long, we need to find a way to **approximate** the least fixed point of \(F\).

The interval domain and its function \(F^i\) is an example of this case.

We need to find a way to compute an element \(A\) such that:

\[
\text{lfp} \sqsubseteq F \sqsubseteq A
\]
Widening Operator

The operator $\nabla : L \times L \rightarrow L$ is called a **widening** operator if:

1. $\forall a, b \in L : a \sqcup b \sqsubseteq a \nabla b$ (widening approximates the join)

2. if $x^0 \sqsubseteq x^1 \sqsubseteq x^2 \sqsubseteq \ldots \sqsubseteq x^n \sqsubseteq \ldots$ is an increasing sequence then $y^0 \sqsubseteq y^1 \sqsubseteq y^2 \sqsubseteq \ldots \sqsubseteq y^n$ stabilizes after a finite number of steps

   where $y^0 = x^0$ and $\forall i \geq 0 : y^{i+1} = y^i \nabla x^{i+1}$

Widening is completely **independent of the function** $F$.

Much like the join, it is an operator defined for the particular domain.
**Useful Theorem**

If $L$ is a complete lattice, $\nabla : L \times L \to L$, $F : L \to L$ is monotone, then the sequence:

$$
\begin{align*}
    y^0 &= \bot \\
    y^1 &= y^0 \nabla F(y^0) \\
    y^2 &= y^1 \nabla F(y^1) \\
    \ldots \\
    y^n &= y^{n-1} \nabla F(y^{n-1})
\end{align*}
$$

will stabilize after a finite number of steps with $y^n$ being a post-fixedpoint of $F$.

By Tarski’s theorem, we know that a post-fixedpoint is above the least fixed point. Hence, it follows that: $\text{lfp} \subseteq F \subseteq y^n$
Widening for Interval Domain

Let us define a widening operator for the intervals

\[ \nabla_i : L^i \times L^i \rightarrow L^i \]

\[ [a, b] \nabla_i [c, d] = [e, f] \] where:

- if \( c < a \), then \( e = -\infty \), else \( e = a \)
- if \( d > b \), then \( f = \infty \), else \( f = b \)

if one of the operands is \( \perp \) the result is the other operand.
The basic intuition is that if we see that an end point is unstable, we move its value to the extreme case.

Exercise: show this operator satisfies the conditions for widening.
Examples: Widening for Interval

\[[1, 2] \triangledown_i [0, 2] = \]

\[[0, 2] \triangledown_i [1, 2] = \]

\[[1, 5] \triangledown_i [1, 5] = \]

\[[2, 3] \triangledown_i [2, 4] = \]
Examples: Widening for Interval

\[ [1, 2] \triangledown_i [0, 2] = [-\infty, 2] \]

\[ [0, 2] \triangledown_i [1, 2] = [0, 2] \]

\[ [1, 5] \triangledown_i [1, 5] = [1, 5] \]

\[ [2, 3] \triangledown_i [2, 4] = [2, \infty] \]
Let us again consider our program with the Interval domain but this time we will apply the widening operator.
Iterate 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: }
```

1: $x \rightarrow \bot_i$, $y \rightarrow \bot_i$, $i \rightarrow \bot_i$
2: $x \rightarrow \bot_i$, $y \rightarrow \bot_i$, $i \rightarrow \bot_i$
3: $x \rightarrow \bot_i$, $y \rightarrow \bot_i$, $i \rightarrow \bot_i$
4: $x \rightarrow \bot_i$, $y \rightarrow \bot_i$, $i \rightarrow \bot_i$
5: $x \rightarrow \bot_i$, $y \rightarrow \bot_i$, $i \rightarrow \bot_i$
6: $x \rightarrow \bot_i$, $y \rightarrow \bot_i$, $i \rightarrow \bot_i$
7: $x \rightarrow \bot_i$, $y \rightarrow \bot_i$, $i \rightarrow \bot_i$

$\text{it}^\theta = \bot$
Iterate 1

\[ \text{it}^1 = \text{it}^0 \uplus \]
\[ \text{F(it}^0) \]
\[ = \bot \uplus \]
\[ \text{F(\bot)} \]
\[ = \text{F(\bot)} \]

```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: }
```

1: \( x \rightarrow [-\infty, \infty] \), \( y \rightarrow [-\infty, \infty] \)
2: \( x \rightarrow \bot_i \), \( y \rightarrow \bot_i \), \( i \rightarrow \bot_i \)
3: \( x \rightarrow \bot_i \), \( y \rightarrow \bot_i \), \( i \rightarrow \bot_i \)
4: \( x \rightarrow \bot_i \), \( y \rightarrow \bot_i \), \( i \rightarrow \bot_i \)
5: \( x \rightarrow \bot_i \), \( y \rightarrow \bot_i \), \( i \rightarrow \bot_i \)
6: \( x \rightarrow \bot_i \), \( y \rightarrow \bot_i \), \( i \rightarrow \bot_i \)
7: \( x \rightarrow \bot_i \), \( y \rightarrow \bot_i \), \( i \rightarrow \bot_i \)
Iterate 2

\[ \text{it}^2 = \text{it}^1 \land F(\text{it}^1) \]

**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: }

1: \( x \rightarrow [-\infty,\infty], \ y \rightarrow [-\infty,\infty], \ i \rightarrow [-\infty,\infty] \)
2: \( x \rightarrow [5,5], \ y \rightarrow [-\infty,\infty], \ i \rightarrow [-\infty,\infty] \)
3: \( x \rightarrow \perp_i, \ y \rightarrow \perp_i, \ i \rightarrow \perp_i \)
4: \( x \rightarrow \perp_i, \ y \rightarrow \perp_i, \ i \rightarrow \perp_i \)
5: \( x \rightarrow \perp_i, \ y \rightarrow \perp_i, \ i \rightarrow \perp_i \)
6: \( x \rightarrow \perp_i, \ y \rightarrow \perp_i, \ i \rightarrow \perp_i \)
7: \( x \rightarrow \perp_i, \ y \rightarrow \perp_i, \ i \rightarrow \perp_i \)
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: }

1: \(x \rightarrow [-\infty,\infty], \ y \rightarrow [-\infty,\infty], \ i \rightarrow [-\infty,\infty]\)
2: \(x \rightarrow [5,5], \ y \rightarrow [-\infty,\infty], \ i \rightarrow [-\infty,\infty]\)
3: \(x \rightarrow [5,5], \ y \rightarrow [7,7], \ i \rightarrow [-\infty,\infty]\)
4: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
5: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
6: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
7: \(x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i\)
Iterate 4

Notice how we propagated to both labels 4 and 7 at the same time

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: }
}

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,7], i → [-∞,∞]
4: x → [5,5], y → [7,7], i → [0,∞]
5: x → ⊥i , y → ⊥i , i → ⊥i
6: x → ⊥i , y → ⊥i , i → ⊥i
7: x → [5,5], y → [7,7], i → [-∞, -1]
Iterate 5

\[ \text{It}^5 = \text{It}^4 \sqcup F(\text{It}^4) \]

```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: }
```
Iterate 6

\[ \text{it}^6 = \text{it}^5 \nabla F(\text{it}^5) \]

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: }

1: x \rightarrow [-\infty, \infty], \ y \rightarrow [-\infty, \infty], \ i \rightarrow [-\infty, \infty]
2: x \rightarrow [5, 5], \ y \rightarrow [-\infty, \infty], \ i \rightarrow [-\infty, \infty]
3: x \rightarrow [5, 5], \ y \rightarrow [7, 7], \ i \rightarrow [-\infty, \infty]
4: x \rightarrow [5, 5], \ y \rightarrow [7, 7], \ i \rightarrow [0, \infty]
5: x \rightarrow [5, 5], \ y \rightarrow [8, 8], \ i \rightarrow [0, \infty]
6: x \rightarrow [5, 5], \ y \rightarrow [8, 8], \ i \rightarrow [-1, \infty]
7: x \rightarrow [5, 5], \ y \rightarrow [7, 7], \ i \rightarrow [-\infty, -1]
Iterate 7: first compute $F(it^6)$

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

1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
2: $x \rightarrow [5,5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
3: $x \rightarrow [5,5]$, $y \rightarrow [7,8]$, $i \rightarrow [-\infty, \infty]$
4: $x \rightarrow [5,5]$, $y \rightarrow [7,7]$, $i \rightarrow [0, \infty]$
5: $x \rightarrow [5,5]$, $y \rightarrow [8,8]$, $i \rightarrow [0, \infty]$
6: $x \rightarrow [5,5]$, $y \rightarrow [8,8]$, $i \rightarrow [-1, \infty]$
7: $x \rightarrow [5,5]$, $y \rightarrow [7,7]$, $i \rightarrow [-\infty, -1]$

\[
\text{it}^7 = \text{it}^6 \nabla F(it^6)
\]
Iterate 7: then $\text{it}^6 \nabla F(\text{it}^6)$

we have:

$$3: x \rightarrow [5, 5], \ y \rightarrow [7, 7], \ i \rightarrow [-\infty, \infty]$$

$\nabla$

$$3: x \rightarrow [5, 5], \ y \rightarrow [7, 8], \ i \rightarrow [-\infty, \infty]$$

$=$

$$3: x \rightarrow [5, 5], \ y \rightarrow [7, \infty], \ i \rightarrow [-\infty, \infty]$$
Iterate 7: final result

\[ \text{foo (int } i \text{)} \{ \]
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:
\}

1: \( x \rightarrow [-\infty, \infty], y \rightarrow [-\infty, \infty], i \rightarrow [-\infty, \infty] \)
2: \( x \rightarrow [5,5], y \rightarrow [-\infty, \infty], i \rightarrow [-\infty, \infty] \)
3: \( x \rightarrow [5,5], y \rightarrow [7, \infty], i \rightarrow [-\infty, \infty] \)
4: \( x \rightarrow [5,5], y \rightarrow [7,7], i \rightarrow [0, \infty] \)
5: \( x \rightarrow [5,5], y \rightarrow [8,8], i \rightarrow [0, \infty] \)
6: \( x \rightarrow [5,5], y \rightarrow [8,8], i \rightarrow [-1, \infty] \)
7: \( x \rightarrow [5,5], y \rightarrow [7,7], i \rightarrow [-\infty, -1] \)

\[ \text{it}^7 = \text{it}^6 \nabla F(\text{it}^6) \]
Iterate 8

```
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:
}
```

```
1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7, ∞], i → [-∞,∞]
4: x → [5,5], y → [7, ∞], i → [0,∞]
5: x → [5,5], y → [8,8], i → [0,∞]
6: x → [5,5], y → [8,8], i → [-1,∞]
7: x → [5,5], y → [7, ∞], i → [-∞, -1]
```

\[ \text{it}^8 = \text{it}^7 \nabla F(\text{it}^7) \]
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: }

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,∞], i → [-∞,∞]
4: x → [5,5], y → [7,∞], i → [0,∞]
5: x → [5,5], y → [8,∞], i → [0,∞]
6: x → [5,5], y → [8,8], i → [-1,∞]
7: x → [5,5], y → [7,∞], i → [-∞, -1]
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: 
}

1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,∞], i → [-∞,∞]
4: x → [5,5], y → [7,∞], i → [0,∞]
5: x → [5,5], y → [8,∞], i → [0,∞]
6: x → [5,5], y → [8,∞], i → [-1,∞]
7: x → [5,5], y → [7,∞], i → [-∞, -1]

\[ it^{10} = it^9 \bigtriangleup F(it^9) \]
Iterate 11: a post fixed point is reached

```
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: }
```

```
1: x → [-∞,∞], y → [-∞,∞], i → [-∞,∞]
2: x → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7, ∞], i → [-∞,∞]
4: x → [5,5], y → [7, ∞], i → [0,∞]
5: x → [5,5], y → [8, ∞], i → [0,∞]
6: x → [5,5], y → [8, ∞], i → [-1,∞]
7: x → [5,5], y → [7, ∞], i → [-∞, -1]
```

\[ \text{it}^{11} = \text{it}^{10} \triangledown F(\text{it}^{10}) \]
Chaotic (Asynchronous) Iteration

\[ x_1 := \perp; \; x_2 = \perp; \ldots; x_n = \perp; \]
\[ W := \{1, \ldots, n\}; \]

\begin{verbatim}
while (W ≠ {}) do { 
    \ell := removeLabel(W);
    prev_\ell := x_\ell;
    x_\ell := prev_\ell \lor f_\ell (x_1, \ldots, x_n);
    if (x_\ell ≠ prev_\ell) 
        W := W ∪ influence(\ell);
}
\end{verbatim}

- \( W \) is the worklist, a set of labels left to be processed
- influence(\ell) returns the set of labels where the value at those labels is influenced by the result at \( \ell \)
- Re-compute only when necessary, thanks to influence(\ell)
- Asynchronous computation can be parallelized
Additional Reading

Covers many aspects of static analysis:

- Full formalization of dataflow analysis
- Lattices
- Detailed introduction to abstract interpretation

Good news:

- Available from Springer website (from academic network)
Additional Materials

List of classical papers and abstract domains: http://www.di.ens.fr/~cousot/AI/

Tools and libraries for abstract interpretation

- Astree
- Fluctuat
- Frama-C

Libraries of abstract domains:

- Oct (octagon)
- NewPolka and Parma (polyhedral)
- Recent: Fast Polyhedra Abstract Domain (POPL’17)