CS 477: Dataflow Analysis and Abstract Interpretation

Sasa Misailovic

Based on previous slides by Martin Vechev

University of Illinois at Urbana-Champaign
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 for a particular programming language.

- Then, use existing theorems which state that the least fixed point of $F^\#$, e.g. denote it $V$, approximates the least fixed point of $F$, e.g. denote it $\lceil P \rceil$.

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

* For a reminder and discussion about soundness and precision, see these articles:
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 \( F \)
(for the purpose of “classical” program analysis) ?

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

What about when:

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

We need to connect the concrete C and the abstract A

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

\[ \alpha : C \rightarrow A \] is called the **abstraction** function

\[ \gamma : A \rightarrow C \] is called the **concretization** function
Connecting Concrete with Abstract

$\alpha$

$(C, \subseteq_c)$

$(A, \subseteq_A)$
Approximating Function: Definition 1

So we have the 2 functions:

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

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

\[ \forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z) \]
Galois Connection

For the course, **it is not important** to know precisely 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

\[ \forall z \in A : \alpha(F(\gamma(z))) \subseteq_A F^#(z) \]
Reminder: Why Abstract Domain?
Approximating a Function

This equation

$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^#(z)$$

says that

• if we have some function in the abstract domain 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.
Visualizing Definition 1

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

\[ \alpha(F(\gamma(Z)))) = [a \mapsto +, b \mapsto 0+] \]

\[ F(X) = \{ [a \mapsto 1, b \mapsto 0], [a \mapsto 1, b \mapsto 1], \ldots \} \]

\[ F(x) \rightarrow [[a=a+1]](x) \]

\[ X = \{ [a \mapsto 0, b \mapsto 0], [a \mapsto 0, b \mapsto 1], \ldots \} \]

\[ Z = [a \mapsto 0, b \mapsto 0+] \]

\[ F^#(z) = [a \mapsto +, b \mapsto 0+] \]

\[ F^#(z) \rightarrow [[a=a+]](z) \]
Least precise approximation

To approximate F, we can always define \( \overset{\#}{F}(z) = T \)

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

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 I: Least Fixed Point Approximation

If we have:

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

then:

\[
\alpha(\text{lfp}(F)) \sqsubseteq_A \text{lfp}(F^\#)
\]

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 $\mathbf{A}$ that we work with. The domain $\mathbf{A}$ 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 $\mathbf{A}$.

Then $\mathbf{F}$ simply updates the mapping from labels to abstract elements.
\[ F# : (\text{Lab} \rightarrow \text{A}) \rightarrow (\text{Lab} \rightarrow \text{A}) \]

\[
F#(m)_{\ell} = \begin{cases} 
T & \text{if } \ell \text{ is initial label} \\
\bigsqcup [\text{action}](m(\ell')) & \text{otherwise}
\end{cases}
\]

\[ [\text{action}] : \text{A} \rightarrow \text{A} \]

[\text{action}] is the key ingredient here. It captures the effect of a language statement on the abstract domain \( \text{A} \). Once we define it, we have \( F# \)

[\text{action}] is often referred to as the \textbf{abstract transformer} (cf. transfer function in dataflow analyses).
What is \((\ell',\text{action}, \ell)\) ?

```plaintext
type 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;
  }
  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 action?

An action can be:

- $b \in BExp$ boolean expression in a conditional
- $x := a$ here, $a \in AExp$
- skip

In performing an action, the assignment and the boolean expression of a conditional is **fully evaluated**

$x := y + x$

$\{x \mapsto 2, \ y \mapsto 0\} \rightarrow \{x \mapsto 4, \ y \mapsto 0\}$

$\{x \mapsto 2, \ y \mapsto 0\} \rightarrow \text{if (x > 5) ...}$
Defining \[\text{action}\]

\[\text{action}\] captures the abstract semantics of the language for a particular abstract domain.

We will see precise definitions for some actions in the Interval domain. Defining \[\text{action}\] for complex domains such as Octagon can be quite tricky.
### 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**: 

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

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

Size of Set:

\[ n > 0 \]

\[ \sum \]

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

\[
\{\langle 1, \{x\mapsto 42, y\mapsto 0, z\mapsto 0 \} \rangle, \\
\langle 1, \{x\mapsto 42, y\mapsto 43, z\mapsto 44 \} \rangle \}
\]

\[
\{\langle 1, \{x\mapsto 2, y\mapsto 4, z\mapsto 1 \} \rangle, \\
\langle 2, \{x\mapsto 3, y\mapsto 4, z\mapsto 1 \} \rangle \}
\]
Representing $\llbracket P \rrbracket$

Let $\llbracket P \rrbracket$ be the set of reachable states of a program $P$. (we discussed this in the Operational semantics lecture)

**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, $\llbracket P \rrbracket$ is a **fixed point** of $F$: i.e., $F(\llbracket P \rrbracket) = \llbracket P \rrbracket$

(in fact, $\llbracket P \rrbracket$ 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 \([P]\)

\[
\sum \{ \langle 1, \{x \mapsto 42, y \mapsto 0, z \mapsto 0 \} \rangle \}
\]

\[
\{ \langle 1, \{x \mapsto 2, y \mapsto 4, z \mapsto 1 \} \rangle \}
\]
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.
Reminder: Interval Arithmetics

Intervals: $X = [X_{\text{min}}, X_{\text{max}}]$ and $Y = [Y_{\text{min}}, Y_{\text{max}}]$

Operations:
$X + Y = [X_{\text{min}} + Y_{\text{min}}, X_{\text{max}} + Y_{\text{max}}]$

$X \times Y = [\min (X_{\text{min}} \times Y_{\text{min}}, X_{\text{min}} \times Y_{\text{max}}, X_{\text{max}} \times Y_{\text{min}}, X_{\text{max}} \times Y_{\text{max}}), \max (X_{\text{min}} \times Y_{\text{min}}, X_{\text{min}} \times Y_{\text{max}}, X_{\text{max}} \times Y_{\text{min}}, X_{\text{max}} \times Y_{\text{max}})]$

The definition of subtraction can then be $X + [-1, -1] \times Y$

The definition for division $X / Y$ is similar to multiplication, but defined properly only when 0 is not in the range $Y$. 
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: \((L^i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \bot_i, [-\infty, \infty])\)

We denote \(Z^\infty = Z \cup \{-\infty, \infty\}\)

The set \(L^i = \{[x, y] \mid x, y \in Z^\infty, x \leq y\} \cup \{\bot_i\}\)

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

Operations (\(\sqsubseteq_i, \sqcup_i, \sqcap_i\)):

- \([a, b] \sqsubseteq_i [c, d]\) if \(c \leq a\) and \(b \leq d\)
- \([a, b] \sqcup_i [c, d] = [\min(a, c), \max(b, d)]\)
- \([a, b] \sqcap_i [c, d] = \text{meet}(\max(a, c), \min(b, d))\)
  where \(\text{meet}(x, y)\) returns \([x, y]\) if \(x \leq y\) and \(\bot_i\) otherwise
Intervals: Applied to Programs

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 $L^i \subseteq_i, \cup_i, \sqcap_i$ are lifted directly to both domains:

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

$$\alpha^i (\{ \langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto -2\rangle, \langle 1,\{x\mapsto 5,y\mapsto 9,q\mapsto -2\rangle, \langle 1,\{x\mapsto 8,y\mapsto 9,q\mapsto -2\rangle, \\
\langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto 4\rangle, \langle 1,\{x\mapsto 5,y\mapsto 9,q\mapsto 4\rangle, \langle 1,\{x\mapsto 8,y\mapsto 9,q\mapsto 4\rangle \}) \\
= 1 \rightarrow (x \mapsto [1,8], y \mapsto [9,9], q \mapsto [-2,4])
$$

$$\gamma^i (1 \rightarrow (x \mapsto [1,8], y \mapsto [9,9], q \mapsto [-2,4])
= \{ \langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto -2\rangle, \langle 1,\{x\mapsto 5,y\mapsto 9,q\mapsto -2\rangle, \langle 1,\{x\mapsto 8,y\mapsto 9,q\mapsto -2\rangle, \\
\langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto 4\rangle, \langle 1,\{x\mapsto 5,y\mapsto 9,q\mapsto 4\rangle, \langle 1,\{x\mapsto 8,y\mapsto 9,q\mapsto 4\rangle, \\
\langle 1,\{x\mapsto 7,y\mapsto 9,q\mapsto 3\rangle, \langle 1,\{x\mapsto 3,y\mapsto 9,q\mapsto 4\rangle, \langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto -1\rangle, \\
..., ..., ...
\}$$

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:

\[
\alpha^i (\text{Ifp } F) \subseteq \text{Ifp } F_i
\]
Let's define $F^i$

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

Here is a definition of $F^i$ which **approximates** the best transformer but **works only on the abstract domain**:

\[
F^i(m) \ell = \begin{cases} 
\lambda v. \left[\infty, \infty\right] & \text{if } \ell \text{ is initial label} \\
\bigcup (\ell', \text{action}, \ell) \left[\text{action}\right]^i(m(\ell')) & \text{otherwise}
\end{cases}
\]

$[\text{action}]^i : \text{(Var} \to \text{L}^i) \to \text{(Var} \to \text{L}^i)$
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;
  }
  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, goto 3, 3)

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

- \((\ell', \text{action}, \ell)\) is an edge in the control-flow graph.

- More formally, if there exists a transition \(t = \langle \ell', \sigma' \rangle \rightarrow \langle \ell, \sigma \rangle\) in a program trace in \(P\), where \(t\) was performed by statement called \text{action}, then \((\ell', \text{action}, \ell)\) must exist. This says that we are sound: we never miss a flow.

- However, \((\ell', \text{action}, \ell)\) may exist even if no such transition \(t\) above occurs. In this case, the analysis would be imprecise as we would unnecessarily create more flows.
What is $(\ell', \text{action}, \ell)$?

An action can be:

- $b \in \text{BExp}$ boolean expression in a conditional
- $\text{x:= a}$ 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.
**F^i on an example**

\[ F^i : (\text{Label} \to (\text{Var} \to \mathbb{L}_i)) \to (\text{Label} \to (\text{Var} \to \mathbb{L}_i)) \]

```plaintext
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: }
```

\[ F^i(m) 1 = \lambda v. [-\infty, \infty] \]
\[ F^i(m) 2 = [x := 5]_i(m(1)) \]
\[ F^i(m) 3 = [y := 7]_i(m(2)) \sqcup [\text{goto 3}]_i(m(6)) \]
\[ F^i(m) 4 = [i \geq 0]_i(m(3)) \]
\[ F^i(m) 5 = [y := y + 1]_i(m(4)) \]
\[ F^i(m) 6 = [i := i - 1]_i(m(5)) \]
\[ F^i(m) 7 = [i < 0]_i(m(3)) \]
\[ [x := a] \circ (m) = m [x \mapsto v] \], where \langle a, m \rangle \Downarrow_i v

\langle a, m \rangle \Downarrow_i v \text{ says that given a map } m, \text{ the expression } a \text{ evaluates to a value } v \in L^i \text{ (using interval arithmetic)}

The operational semantics rules for expression evaluation:

- any constant \( Z \) is abstracted to an element in \( L^i \)
- operators +, -, and \( \ast \) are re-defined for the Interval domain
Arithmetic Expressions

If we add $\bot_i$ to any other element, we get $\bot_i$.

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

$$[x, y] + [z, q] = [x + z, y + q]$$

what about $\ast$?

is $[x, y] \ast [z, q] = [x \ast z, y \ast q]$ sound?

Look for all four combinations!
Let us first look at the expression: $a_1 \ c \ a_2$

Here, $c$ is a condition such as: $\leq, \ =, \ <$

For a memory map $m$, we need to define: $[a_1 \ c \ a_2]_i(m)$ which produces another map as the result.
What is $\lbrack x \leq y \rbrack$ ?

Easy case: $x_{\text{max}} \leq y_{\text{min}}$
• We simply keep the intervals of $x$ and $y$

But, 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 ?
Definition of $[l_1, u_1] \leq [l_2, u_2]$
Definition of \([l_1, u_1] \leq [l_2, u_2]\)

\([l_1, u_1] \leq [l_2, u_2] = ([l_1, u_1] \cap_i [-\infty, u_2], [l_1, \infty] \cap_i [l_2, u_2])\)

\([0, 4] \leq [3, 5] = (x=[0, 4] \cap_i [-\infty, 5], y=[0, \infty] \cap_i [3, 5])\)

\(= (x=[0, 4], y=[3, 5])\)

Exercise: define < and =
Evaluating $[b]_i$

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

$[b_1 \land b_2]_i (m) = [b_1]_i (m) \cap [b_2]_i (m)$
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
Chaotic (Asynchronous) Iteration

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

\textbf{while} \ (W \neq \{\}) \ \textbf{do} \ \{ \\
\quad \ell := \text{removeLabel}(W); \\
\quad \text{prev}_\ell := x_\ell; \\
\quad x_\ell := f_\ell(x_1, \ldots, x_n); \\
\quad \text{if} \ (x_\ell \neq \text{prev}_\ell) \\
\quad \quad W := W \cup \text{influence}(\ell); \\
\}

- \( W \) is the worklist, a set of labels left to be processed

- \( \text{influence}(\ell) \) 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 \( \text{influence}(\ell) \)

- Asynchronous computation can be parallelized
\( \mathcal{F}^i \) on an example

\( \mathcal{F}^i : (\text{Label} \to (\text{Var} \to L^i)) \to (\text{Label} \to (\text{Var} \to L^i)) \)

\begin{align*}
\text{foo (int } i \text{) } &\{ \\
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: \}
\}
\end{align*}

\begin{align*}
\mathcal{F}^i (m) 1 &= \lambda v. [-\infty, \infty] \\
\mathcal{F}^i (m) 2 &= \llbracket x := 5 \rrbracket_i (m(1)) \\
\mathcal{F}^i (m) 3 &= \llbracket y := 7 \rrbracket_i (m(2)) \sqcup \llbracket \text{goto 3} \rrbracket_i (m(6)) \\
\mathcal{F}^i (m) 4 &= \llbracket i \geq 0 \rrbracket_i (m(3)) \\
\mathcal{F}^i (m) 5 &= \llbracket y := y + 1 \rrbracket_i (m(4)) \\
\mathcal{F}^i (m) 6 &= \llbracket i := i - 1 \rrbracket_i (m(5)) \\
\mathcal{F}^i (m) 7 &= \llbracket i < 0 \rrbracket_i (m(3))
\end{align*}
Let us compute the least fixed point of $F^i$
Iterate 0

The collection of these lines denote the current iterate. The iterate is a map

```cpp
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 → ⊥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;
    }
    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

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: }

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

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

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 → [5,5], y → [8,8], i → [0,∞]
6: x → ⊥i, y → ⊥i, i → ⊥i
7: x → [5,5], y → [7,7], i → [-∞, -1]
Iterate 6

```cpp
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 7

```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: \( x \rightarrow [5, 5], \ y \rightarrow [7, 8], \ i \rightarrow [-\infty, \infty] \)
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: \( x \rightarrow [5, 5], \ y \rightarrow [7, 7], \ i \rightarrow [-\infty, -1] \)
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;
3: }
7: }

<table>
<thead>
<tr>
<th></th>
<th>x → [−∞, ∞], y → [−∞, ∞], i → [−∞, ∞]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x → [5, 5], y → [−∞, ∞], i → [−∞, ∞]</td>
</tr>
<tr>
<td>2</td>
<td>x → [5, 5], y → [7, 8], i → [−∞, ∞]</td>
</tr>
<tr>
<td>4</td>
<td>x → [5, 5], y → [7, 8], i → [0, ∞]</td>
</tr>
<tr>
<td>5</td>
<td>x → [5, 5], y → [8, 9], i → [0, ∞]</td>
</tr>
<tr>
<td>6</td>
<td>x → [5, 5], y → [8, 8], i → [−1, ∞]</td>
</tr>
<tr>
<td>7</td>
<td>x → [5, 5], y → [7, 8], i → [−∞, −1]</td>
</tr>
</tbody>
</table>
Iterate 10

```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:   }
```

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: \(x \rightarrow [5, 5], y \rightarrow [7, 8], i \rightarrow [0, \infty]\)
5: \(x \rightarrow [5, 5], y \rightarrow [8, 9], i \rightarrow [0, \infty]\)
6: \(x \rightarrow [5, 5], y \rightarrow [8, 9], i \rightarrow [-1, \infty]\)
7: \(x \rightarrow [5, 5], y \rightarrow [7, 8], i \rightarrow [-\infty, -1]\)
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;
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, 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]
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:
}
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 → [7,9], 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 16

```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:
}

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: \( F^1, F^2, F^3, \ldots \)

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 \(F\), then when 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:

\[
\text{Ifp (F)} \sqsubseteq A
\]
The operator $\nabla: L \times L \rightarrow L$ is called a **widening** operator if:

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

- 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, $\lor: L \times L \rightarrow L$, $F: L \rightarrow L$ is monotone

Then the sequence:

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

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: $\text{lfp}(F) \subseteq y^n$
Let us define a widening operator for the intervals \([a, b] \otimes_i [c, d] = [e, f]\) 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

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

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

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

\[
[2, 3] \triangledown_i [2, 4] =
\]

Let us define a widening operator for the intervals

\[
\triangledown_i : \mathbb{L} \times \mathbb{L} \rightarrow \mathbb{L}
\]

\[
[a, b] \triangledown_i [c, d] = [e, f]
\]

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.
Examples: Widening for Interval

Let us define a widening operator for the intervals

\[ \nabla_i : L^1 \times L^1 \rightarrow L^1 \]

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

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 \( \bot \) the result is the other operand.

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

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

\[ [1, 5] \nabla_i [1, 5] = [1, 5] \]

\[ [2, 3] \nabla_i [2, 4] = [2, \infty] \]
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 → ⊥_i, y → ⊥_i, i → ⊥_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
Iterate 1

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

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 \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

\[ it^2 = it^1 \uplus F(it^1) \]

```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: \( x \rightarrow \bot_i, \ y \rightarrow \bot_i, \ i \rightarrow \bot_i \)
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) {
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 

\[ \text{it}^3 = \text{it}^2 \nabla F(\text{it}^2) \]
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

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

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

\[ \text{it}^5 = \text{it}^4 \triangledown F(\text{it}^4) \]
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 $F(it^6)$

$$it^7 = it^6 \nabla F(it^6)$$

```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;
3: }
4: }
```

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: $x \rightarrow [5, 5], \ y \rightarrow [7, 7], \ i \rightarrow [-\infty, -1]$
Iterate 7: then $it^6 \nabla F(it^6)$

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

```
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 → [5,5], y → [-∞,∞], i → [-∞,∞]
3: x → [5,5], y → [7,∞], 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]
```

\[ it^7 = it^6 ∨ F(it^6) \]
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]
Iterate 9

\[ \text{it}^9 = \text{it}^8 \lor F(\text{it}^8) \]

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, ∞], 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]
Iterate 11: a post fixed point is reached

\[
\text{It}^{11} = \text{It}^{10} \triangledown F(\text{It}^{10})
\]

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]
Chaotic (Asynchronous) Iteration

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

\textbf{while} (\( W \neq \{\} \)) \textbf{do} \{ \\
\quad \ell := \text{removeLabel}(W); \\
\quad \text{prev}_\ell := x_\ell; \\
\quad x_\ell := f_\ell(x_1, \ldots, x_n); \\
\quad \text{if} (x_\ell \neq \text{prev}_\ell) \quad \text{then} \\
\qquad W := W \cup \text{influence}(\ell); \\
\}

- \( W \) is the worklist, a set of labels left to be processed.
- \( \text{influence}(\ell) \) 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 \( \text{influence}(\ell) \).
- Asynchronous computation can be parallelized.
Chaotic (Asynchronous) Iteration With Widening

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

\[
\text{while } (W \neq \{\}) \text{ do } \{
\quad \ell := \text{removeLabel}(W);
\quad \text{prev}_\ell := x_\ell;
\quad x_\ell := \text{prev}_\ell \lor f_\ell (x_1, \ldots, x_n);
\quad \text{if } (x_\ell \neq \text{prev}_\ell)
\quad \quad W := W \cup \text{influence}(\ell);
\}\]

- \( W \) is the worklist, a set of labels left to be processed
- \( \text{influence}(\ell) \) 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 \( \text{influence}(\ell) \)
- Asynchronous computation can be parallelized
HEAP ANALYSIS
Pointer Analysis

Pointer and Alias Analysis are fundamental to reasoning about heap manipulating programs (pretty much all programs today).

• **Pointer Analysis:**
  • What objects does each pointer points to?
  • Also called points-to analysis

• **Alias Analysis:**
  • Can two pointers point to the same location?
  • Client of pointer analysis
Example

\[ X = 1 \]
\[ P = \&X \]
\[ Q = P \]
\[ \ast P = 2 \]

Points-to Pair: pair \((r_1, r_2)\) denoting that one of the memory locations of \(r_1\) an ordered may hold the address of one of the memory locations of \(r_2\).

Points-to pairs

// \((P, X)\)
// \{(P, X), (Q, X)\}
**Example**

\[
\begin{align*}
X &= I \\
P &= &X \\
Q &= P \\
R &= Q
\end{align*}
\]

**Points-to Pair:** pair \((r_1, r_2)\) denoting that one of the memory locations of \(r_1\) An ordered may hold the address of one of the memory locations of \(r_2\).

**Points-to pairs**

// \((P, X)\)

// \{(P, X), (Q, X)\}

// \{(P, X), (Q, X), (R, X)\}

*“Short notation”: vs the long one that would list all the aliases.*
Challenges of Points-To Analysis

- **Pointers to pointers**, which can occur in many ways: take address of pointer; pointer to structure containing pointer; pass a pointer to a procedure by reference
- **Aggregate objects**: structures and arrays containing pointers
- **Recursive data structures** (lists, trees, graphs, etc.) closely related problem: anonymous heap locations
- **Control-flow**: analyzing different data paths
- **Interprocedural**: a location is often accessed from multiple functions; a common pattern (e.g., pass by reference)
- **Compile-time cost**
  - Number of variables, $|V|$, can be large
  - Number of alias pairs at a point can be $O(|V|^2)$
Naming Schemes for Heap Objects

The Naming Problem: Example 1

```c
int main() {
    // Two distinct objects
    T* p = create(n);
    T* q = create(m);
}

T* create(int num) {
    // Many objects allocated here
    return new T(num);
}
```

Q. Should we try to distinguish the objects created in `main()`?
Naming Schemes for Heap Objects

The Naming Problem: Example 2

```c
T* makelist(int len) {
    T* newObj = new T; // Many distinct objects
    // allocated here
    newObj->next = (--len == 0)? NULL :
        makelist(len);
}
```

Q. Can we distinguish the objects created in makelist()?
Possible Naming Abstractions

$H_0$: One name for the entire heap

$H_T$: One name per type $T$ (for type-safe languages)

$H_L$: One name per heap allocation site $L$ (line number)

$H_C$: One name per (acyclic) call path $C$ (“cloning”)

$H_F$: One name per immediate caller $F$ or call-site (“one-level cloning”)
Program States for Points-To Analysis

Abstraction:
\{ \text{pointer} \rightarrow \{\text{Allocation Sites}\}, \ldots \} \\
e.g., \{ p \rightarrow \{A\}, x \rightarrow \{A\}, z \rightarrow \{A\} \}

Concretization:
\{ \text{pointer} \rightarrow \{\text{Objects allocated}\} \ldots \} \\
e.g., \{ p \rightarrow \{O_1, O_2\}, x \rightarrow \{O_1, O_2\}, z \rightarrow \{O_1, O_2\} \}

```java
// initially x = z = p = q = null
for (i = 0; i < 2; i++) {
    // allocate O_1, O_2
    A: x := newObject T1;
    if (i == 0) 
        p := x;
    else
        z := x;
}
// allocate O_3
B: x := newObject T1;
    z.f := x;
// allocate O_4
C: q := newObject T1;
    x := null;
```
Program States for Points-To Analysis

```java
// 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:

- \( p \rightarrow \emptyset, q \rightarrow \emptyset, x \rightarrow \emptyset, z \rightarrow \emptyset \)
- \( p \rightarrow \{A\}, q \rightarrow \emptyset, x \rightarrow \{A\}, z \rightarrow \{A\} \)
- \( p \rightarrow \{A\}, q \rightarrow \emptyset, x \rightarrow \{A\}, z \rightarrow \{A\} \)
- \( p \rightarrow \{A\}, q \rightarrow \emptyset, x \rightarrow \{A\}, z \rightarrow \{A\} \)
- \( p \rightarrow \{A\}, q \rightarrow \emptyset, x \rightarrow \{A\}, z \rightarrow \{A\} \)
- \( p \rightarrow \{A\}, q \rightarrow \emptyset, x \rightarrow \{B\}, z \rightarrow \{A\} \)
- \( p \rightarrow \{A\}, q \rightarrow \emptyset, x \rightarrow \{B\}, z \rightarrow \{A\}, A.f \rightarrow \{B\} \)
- \( p \rightarrow \{A\}, q \rightarrow \{C\}, x \rightarrow \{\}, z \rightarrow \{A\}, A.f \rightarrow \{B\} \)
RELATIONAL ABSTRACT
DOMAINS
Sign Domain

\[
\begin{align*}
\text{TOP} & \quad 0 & \quad \text{BOT} \\
- & \quad 0 & \quad +
\end{align*}
\]
Constant Domain

\[
\begin{array}{c c c c c}
& -2 & -1 & 0 & +1 & +2 & \ldots \\
\ldots & & & & & & \\
& & & & & & \\
\end{array}
\]
Each variable takes a value from the following domain (a complete lattice):
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

The slope is fixed

constraints are of the following form:

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

\[ \pm x \leq c \]

\[ \pm y \leq c \]

an abstract state is a map from labels to conjunction of constraints

\[ x - y \leq 3 \wedge y \leq 8 \wedge y \geq 2 \wedge x + y \leq 15 \wedge x + y \geq 5 \wedge x \geq 1 \wedge x - y \geq -20 \wedge x \leq 7 \]
Polyhedra Domain

The 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:

\[
\begin{align*}
  x - y &\geq -20 \\
  x - 3y &\leq 2 \\
  x + y &\geq 5
\end{align*}
\]

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In POPL '78
Approximating a Function: Definition 2

We have the 2 functions:

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

But what if \( \alpha \) and \( \gamma \) do not form a Galois Connection?

- For instance, \( \alpha \) is not monotone.
- For instance, Polyhedral domain does not form GC.

Then, we can use the following definition of approximation:

\[ \forall z \in A : F(\gamma(z)) \subseteq_c \gamma(F^\#(z)) \]
Visualizing Definition 2

\[ \forall z \in A : F(\gamma(z)) \sqsubseteq_c \gamma(F^\#(z)) \]
Key Theorem 2: Least Fixed Point Approximation

If we have:

1. **monotone** functions \( F : C \to C \) and \( F^\#: A \to A \)
2. \( \gamma : A \to C \) is monotone
3. \( \forall z \in A : F(\gamma(z)) \sqsubseteq_c \gamma(F^#(z)) \) (that is, \( F^# \) approximates \( F \))

then:

\[
\text{lfp}(F) \sqsubseteq_c \gamma(\text{lfp}(F^#))
\]

This is important as it goes from **local** function approximation to **global** approximation. Another key theorem in program analysis.
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).
• Machine Learning: Certifying Neural Networks (Singh et al POPL ‘19)
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)