A Gentle Introduction to Program Analysis

Işık Dillig
University of Texas, Austin

January 21, 2014
Programming Languages Mentoring Workshop
Very broad topic, but generally speaking, **automated** analysis of program behavior
What is Program Analysis?

- Very broad topic, but generally speaking, **automated** analysis of program behavior

- Program analysis is about developing algorithms and tools that can analyze **other programs**
What is Program Analysis?

- Very broad topic, but generally speaking, **automated** analysis of program behavior

- Program analysis is about developing algorithms and tools that can analyze **other programs**

```
prog.c
Query
Program Analysis Tool
```
Applications of Program Analysis

- **Bug finding.** e.g., expose as many assertion failures as possible
Applications of Program Analysis

- **Bug finding.** e.g., expose as many assertion failures as possible

- **Security.** e.g., does an app leak private user data?
Applications of Program Analysis

- **Bug finding.** e.g., expose as many assertion failures as possible

- **Security.** e.g., does an app leak private user data?

- **Verification.** e.g., does the program always behave according to its specification?
Applications of Program Analysis

- **Bug finding.** e.g., expose as many assertion failures as possible

- **Security.** e.g., does an app leak private user data?

- **Verification.** e.g., does the program always behave according to its specification?

- **Compiler optimizations.** e.g., which variables should be kept in registers for fastest memory access?
Applications of Program Analysis

- **Bug finding.** e.g., expose as many assertion failures as possible

- **Security.** e.g., does an app leak private user data?

- **Verification.** e.g., does the program always behave according to its specification?

- **Compiler optimizations.** e.g., which variables should be kept in registers for fastest memory access?

- **Automatic parallelization.** e.g., is it safe to execute different loop iterations on parallel?
Dynamic vs. Static Program Analysis

- Two flavors of program analysis:
Dynamic vs. Static Program Analysis

- Two flavors of program analysis:
  - **Dynamic analysis**: Analyzes program while it is running
Two flavors of program analysis:

- **Dynamic analysis**: Analyzes program while it is running
- **Static analysis**: Analyzes source code of the program
Two flavors of program analysis:

- **Dynamic analysis:** Analyzes program while it is running
- **Static analysis:** Analyzes source code of the program

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ reasons about all executions</td>
<td>+ more precise</td>
</tr>
<tr>
<td>- less precise</td>
<td>- results limited to observed executions</td>
</tr>
</tbody>
</table>
**Typical static analysis question:** "Given source code of program P and desired property Q, does P exhibit Q in all possible executions?"

But this question is **undecidable**!

This means static analyses are either:

- **Unsound:** May say program is safe even though it is unsafe
- **Sound, but incomplete:** May say program is unsafe even though it is safe
- **Non-terminating:** Always gives correct answer when it terminates, but may run forever

Many static analysis techniques are sound but incomplete.
Typical static analysis question: "Given source code of program P and desired property Q, does P exhibit Q in all possible executions?"

But this question is undecidable!
Typical static analysis question: "Given source code of program P and desired property Q, does P exhibit Q in all possible executions?"

But this question is undecidable!

This means static analyses are either:
Typical static analysis question: "Given source code of program P and desired property Q, does P exhibit Q in all possible executions?"

But this question is **undecidable**!

This means static analyses are either:

- **Unsound**: May say program is safe even though it is unsafe
Typical static analysis question: "Given source code of program P and desired property Q, does P exhibit Q in all possible executions?"

But this question is **undecidable**!

This means static analyses are either:

- **Unsound**: May say program is safe even though it is unsafe
- **Sound, but incomplete**: May say program is unsafe even though it is safe
**Static Analysis**

- **Typical static analysis question:** "Given source code of program P and desired property Q, does P exhibit Q in all possible executions?"

- But this question is **undecidable**!

- This means static analyses are either:
  - **Unsound:** May say program is safe even though it is unsafe
  - **Sound, but incomplete:** May say program is unsafe even though it is safe
  - **Non-terminating:** Always gives correct answer when it terminates, but may run forever
Typical static analysis question: "Given source code of program P and desired property Q, does P exhibit Q in all possible executions?"

But this question is **undecidable**!

This means static analyses are either:

- **Unsound**: May say program is safe even though it is unsafe
- **Sound, but incomplete**: May say program is unsafe even though it is safe
- **Non-terminating**: Always gives correct answer when it terminates, but may run forever

Many static analysis techniques are sound but incomplete.
How to design sound static analyses?

Key idea: Overapproximate (i.e., abstract) program behavior
**Key idea:** Overapproximate (i.e., abstract) program behavior
How to design sound static analyses?

**Key idea:** Overapproximate (i.e., abstract) program behavior
How to design sound static analyses?

**Key idea:** Overapproximate (i.e., abstract) program behavior

- Bad states outside over-approximation
  ⇒ Program safe
How to design sound static analyses?

Key idea: Overapproximate (i.e., abstract) program behavior

- Bad states outside over-approximation
  ⇒ Program safe

- Bad states inside over-approximation, but outside $P$
  ⇒ false alarm
How to design sound static analyses?

**Key idea:** Overapproximate (i.e., abstract) program behavior

- Bad states outside over-approximation
  ⇒ Program safe

- Bad states inside over-approximation, but outside $P$
  ⇒ false alarm

⇒ **Goal:** Construct abstractions that are precise enough (i.e., few false alarms) and that scale to real programs
Examples of Abstractions

There is no "one size fits all" abstraction. What information is useful depends on what you want to prove about the program!

- **Application**
  - Useful abstraction
    - No division-by-zero errors
    - zero vs. non-zero
    - Data structure verification
      - list, tree, graph, . . .
    - No out-of-bounds array accesses
      - ranges of integer variables
Examples of Abstractions

There is no “one size fits all” abstraction

- What information is useful depends on what you want to prove about the program!
There is no “one size fits all” abstraction

- What information is useful depends on what you want to prove about the program!

<table>
<thead>
<tr>
<th>Application</th>
<th>Useful abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>No division-by-zero errors</td>
<td>zero vs. non-zero</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples of Abstractions

There is no “one size fits all” abstraction

- What information is useful depends on what you want to prove about the program!

<table>
<thead>
<tr>
<th>Application</th>
<th>Useful abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>No division-by-zero errors</td>
<td>zero vs. non-zero</td>
</tr>
<tr>
<td>Data structure verification</td>
<td>list, tree, graph, ...</td>
</tr>
</tbody>
</table>
There is no “one size fits all” abstraction

- What information is useful depends on what you want to prove about the program!

<table>
<thead>
<tr>
<th>Application</th>
<th>Useful abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>No division-by-zero errors</td>
<td>zero vs. non-zero</td>
</tr>
<tr>
<td>Data structure verification</td>
<td>list, tree, graph, ...</td>
</tr>
<tr>
<td>No out-of-bounds array accesses</td>
<td>ranges of integer variables</td>
</tr>
</tbody>
</table>
Useful theory for understanding how to design sound static analyses is abstract interpretation. Seminal ‘77 paper by Patrick & Radhia Cousot. Not a specific analysis, but rather a framework for designing sound-by-construction static analyses. Let’s look at an example: A static analysis that tracks the sign of each integer variable (e.g., positive, non-negative, zero etc.).
Useful theory for understanding how to design sound static analyses is **abstract interpretation**.
Useful theory for understanding how to design sound static analyses is **abstract interpretation**

Seminal ’77 paper by Patrick & Radhia Cousot
How to Create Sound Abstractions?

- Useful theory for understanding how to design sound static analyses is **abstract interpretation**

- Seminal ’77 paper by Patrick & Radhia Cousot

- Not a specific analysis, but rather a framework for designing sound-by-construction static analyses
Useful theory for understanding how to design sound static analyses is **abstract interpretation**.

Seminal ’77 paper by Patrick & Radhia Cousot.

Not a specific analysis, but rather a framework for designing sound-by-construction static analyses.

Let’s look at an example: A static analysis that tracks the sign of each integer variable (e.g., positive, non-negative, zero etc.)
An abstract domain is just a set of abstract values we want to track in our analysis.
First Step: Design An Abstract Domain

- **An abstract domain** is just a set of abstract values we want to track in our analysis.

For our example, let’s fix the following abstract domain:

- **pos**: \( \{ x \mid x \in \mathbb{Z} \land x > 0 \} \)
- **zero**: \( \{ 0 \} \)
- **neg**: \( \{ x \mid x \in \mathbb{Z} \land x < 0 \} \)
- **non-neg**: \( \{ x \mid x \in \mathbb{Z} \land x \geq 0 \} \)

In addition, every abstract domain contains:

- **⊤** (top): “Don’t know”, represents any value
- **⊥** (bottom): Represents empty-set
An abstract domain is just a set of abstract values we want to track in our analysis.

For our example, let’s fix the following abstract domain:

- **pos:** \(\{x \mid x \in \mathbb{Z} \land x > 0\}\)
- **zero:** \(\{0\}\)
- **neg:** \(\{x \mid x \in \mathbb{Z} \land x < 0\}\)
- **non-neg:** \(\{x \mid x \in \mathbb{Z} \land x \geq 0\}\)

In addition, every abstract domain contains:

- \(\top\) (top): “Don’t know”, represents any value
- \(\bot\) (bottom): Represents empty-set
First Step: Design An Abstract Domain

- An **abstract domain** is just a set of **abstract values** we want to track in our analysis.

For our example, let’s fix the following abstract domain:

- pos: \( \{ x \mid x \in \mathbb{Z} \land x > 0 \} \)
- zero: \( \{ 0 \} \)
- neg: \( \{ x \mid x \in \mathbb{Z} \land x < 0 \} \)
- non-neg: \( \{ x \mid x \in \mathbb{Z} \land x \geq 0 \} \)

In addition, every abstract domain contains:

- \( \top \) (top): “Don’t know”, represents any value
First Step: Design An Abstract Domain

- An **abstract domain** is just a set of **abstract values** we want to track in our analysis.

- For our example, let’s fix the following abstract domain:
  - **pos**: \( \{ x \mid x \in \mathbb{Z} \land x > 0 \} \)
  - **zero**: \( \{ 0 \} \)
  - **neg**: \( \{ x \mid x \in \mathbb{Z} \land x < 0 \} \)
  - **non-neg**: \( \{ x \mid x \in \mathbb{Z} \land x \geq 0 \} \)

- In addition, every abstract domain contains:
  - \( \top \) (**top**): “Don’t know”, represents any value
  - \( \bot \) (**bottom**): Represents empty-set
Abstraction function ($\alpha$) maps sets of concrete elements to the most precise value in the abstract domain.
Abstraction function \((\alpha)\) maps sets of concrete elements to the most precise value in the abstract domain

- \(\alpha\{2, 10, 0\} = \text{non-neg}\)
- \(\alpha\{3, 99\} = \text{pos}\)
- \(\alpha\{-3, 2\} = \top\)
Abstraction function \( (\alpha) \) maps sets of concrete elements to the most precise value in the abstract domain

\[
\alpha(\{2, 10, 0\}) = \text{non-neg}
\]
Abstraction function \( \alpha \) maps sets of concrete elements to the most precise value in the abstract domain

- \( \alpha(\{2, 10, 0\}) = \text{non-neg} \)
- \( \alpha(\{3, 99\}) = \)
Abstraction function \((\alpha)\) maps sets of concrete elements to the most precise value in the abstract domain
\[
\begin{align*}
\alpha(\{2, 10, 0\}) &= \text{non-neg} \\
\alpha(\{3, 99\}) &= \text{pos}
\end{align*}
\]
Second step: Abstraction and concretization functions

- **Abstraction function** ($\alpha$) maps sets of concrete elements to the most precise value in the abstract domain
  - $\alpha(\{2, 10, 0\}) = \text{non-neg}$
  - $\alpha(\{3, 99\}) = \text{pos}$
  - $\alpha(\{-3, 2\}) = \text{}$
Abstraction function \( (\alpha) \) maps sets of concrete elements to the most precise value in the abstract domain

- \( \alpha(\{2, 10, 0\}) = \text{non-neg} \)
- \( \alpha(\{3, 99\}) = \text{pos} \)
- \( \alpha(\{-3, 2\}) = \top \)
Second step: Abstraction and concretization functions

- **Abstraction function** ($\alpha$) maps sets of concrete elements to the most precise value in the abstract domain
  - $\alpha(\{2, 10, 0\}) = \text{non-neg}$
  - $\alpha(\{3, 99\}) = \text{pos}$
  - $\alpha(\{-3, 2\}) = \top$

- **Concretization function** ($\gamma$) maps each abstract value to sets of concrete elements
Second step: Abstraction and concretization functions

- **Abstraction function** ($\alpha$) maps sets of concrete elements to the most precise value in the abstract domain
  
  - $\alpha(\{2, 10, 0\}) = \text{non-neg}$
  
  - $\alpha(\{3, 99\}) = \text{pos}$
  
  - $\alpha(\{-3, 2\}) = \top$

- **Concretization function** ($\gamma$) maps each abstract value to sets of concrete elements
  
  - $\gamma(\text{pos}) = \{x \mid x \in \mathbb{Z} \land x > 0\}$
Concretization function defines partial order on abstract values:

Furthermore, in an abstract domain, every pair of elements has a lub and glb ⇒ mathematical lattice.
Concretization function defines partial order on abstract values:

\[ A_1 \leq A_2 \text{ iff } \gamma(A_1) \subseteq \gamma(A_2) \]
Concretization function defines partial order on abstract values:

\[ A_1 \leq A_2 \text{ iff } \gamma(A_1) \subseteq \gamma(A_2) \]

Furthermore, in an abstract domain, every pair of elements has a lub and glb \( \Rightarrow \) mathematical lattice

```
  T
 /\    \/
 non-neg pos zero neg
 /     \
 zero     \
 /       \
 non-neg
 /       \
 pos
 /     \
 zero
 /\  \
 non-neg
 /   \
 neg
 \
↓      
⊥
```
Lattices and Abstract Domains

- Concretization function defines partial order on abstract values:
  \[ A_1 \leq A_2 \text{ iff } \gamma(A_1) \subseteq \gamma(A_2) \]

- Furthermore, in an abstract domain, every pair of elements has a lub and glb \( \Rightarrow \) mathematical lattice

- Least upper bound of two elements is called their join – useful for reasoning about control flow in programs
Important property of the abstraction and concretization function is that they are almost inverses:
Important property of the abstraction and concretization function is that they are **almost inverses**:

\[ \alpha(\gamma(A)) = A \]
Important property of the abstraction and concretization function is that they are **almost inverses**: \[
\alpha(\gamma(A)) = A
\]
\[
C \subseteq \gamma(\alpha(C))
\]
Important property of the abstraction and concretization function is that they are **almost inverses**:

\[
\alpha(\gamma(A)) = A
\]

\[
C \subseteq \gamma(\alpha(C))
\]

This is called a **Galois insertion** and captures the soundness of the abstraction
Step 3: Abstract Semantics

- Given abstract domain, $\alpha, \gamma$, need to define abstract transformers (i.e., semantics) for each statement.
Step 3: Abstract Semantics

- Given abstract domain, $\alpha, \gamma$, need to define abstract transformers (i.e., semantics) for each statement
  - Describes how statements affect our abstraction
Step 3: Abstract Semantics

- Given abstract domain, $\alpha, \gamma$, need to define abstract transformers (i.e., semantics) for each statement
  - Describes how statements affect our abstraction
  - Abstract counter-part of operational semantics rules
Step 3: Abstract Semantics

- Given abstract domain, $\alpha, \gamma$, need to define abstract transformers (i.e., semantics) for each statement
  - Describes how statements affect our abstraction
  - Abstract counter-part of operational semantics rules

**Operational Semantics**

- $x = y \text{ op } z$
- $S: \text{ Var} \rightarrow \text{Concrete value}$
- $S': \text{ Var} \rightarrow \text{Concrete value}$
Step 3: Abstract Semantics

- Given abstract domain, $\alpha, \gamma$, need to define abstract transformers (i.e., semantics) for each statement
  - Describes how statements affect our abstraction
- Abstract counter-part of operational semantics rules

**Operational Semantics**

- $x = y \text{ op } z$
- $S: \text{Var} \rightarrow \text{Concrete value}$
- $S': \text{Var} \rightarrow \text{Concrete value}$

**Abstract Semantics**

- $x = y \text{ op } z$
- $A: \text{Var} \rightarrow \text{Abstract value}$
- $A': \text{Var} \rightarrow \text{Abstract value}$
For our sign analysis, we can define abstract transformer for $x = y + z$ as follows:

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
& pos & neg & zero & non-neg & \top & \bot \\
\hline
pos & pos & \top & pos & pos & \top & \bot \\
\hline
neg & \top & neg & neg & \top & \top & \bot \\
\hline
zero & pos & neg & zero & non-neg & \top & \bot \\
\hline
non-neg & pos & \top & non-neg & non-neg & \top & \bot \\
\hline
\top & \top & \top & \top & \top & \top & \bot \\
\hline
\bot & \bot & \bot & \bot & \bot & \bot & \bot \\
\hline
\end{array}
\]
For our sign analysis, we can define abstract transformer for $x = y + z$ as follows:

<table>
<thead>
<tr>
<th></th>
<th>pos</th>
<th>neg</th>
<th>zero</th>
<th>non-neg</th>
<th>⊤</th>
<th>⊥</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>pos</td>
<td>⊤</td>
<td>pos</td>
<td>pos</td>
<td>⊤</td>
<td>⊥</td>
</tr>
<tr>
<td>neg</td>
<td>⊤</td>
<td>neg</td>
<td>neg</td>
<td>⊤</td>
<td>⊤</td>
<td>⊥</td>
</tr>
<tr>
<td>zero</td>
<td>pos</td>
<td>neg</td>
<td>zero</td>
<td>non-neg</td>
<td>⊤</td>
<td>⊥</td>
</tr>
<tr>
<td>non-neg</td>
<td>pos</td>
<td>⊤</td>
<td>non-neg</td>
<td>non-neg</td>
<td>⊤</td>
<td>⊥</td>
</tr>
<tr>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
<td>⊥</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>

To ensure soundness of static analysis, must prove that abstract semantics faithfully models concrete semantics.
Putting It All Together

Abstract domain

Abstract semantics

Fixed-point engine

\( \alpha, \gamma \)

\( P \)
Putting It All Together

Abstract domain
\[ \alpha, \gamma \]

Abstract semantics

Fixed-point engine

\[ P \]
Fixed-point computations: Repeated symbolic execution of the program using abstract semantics until our approximation of the program reaches an equilibrium.
**Fixed-point computation:** Repeated symbolic execution of the program using abstract semantics until our approximation of the program reaches an equilibrium

**Least fixed-point:** Start with underapproximation and grow the approximation until it stops growing
**Fixed-point computation**: Repeated symbolic execution of the program using abstract semantics until our approximation of the program reaches an equilibrium.

**Least fixed-point**: Start with underapproximation and grow the approximation until it stops growing.

Assuming correctness of your abstract semantics, the least fixed point is an overapproximation of the program!
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**

- Want to compute abstract values at every program point
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point
- Initialize all abstract states to \( \perp \)
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point
- Initialize all abstract states to \( \perp \)
- Repeat until no abstract state changes at any program point:
Performing Least Fixed Point Computation

- Represent program as a control-flow graph
- Want to compute abstract values at every program point
- Initialize all abstract states to ⊥
- Repeat until no abstract state changes at any program point:
  - Compute abstract state on entry to a basic block B by taking the join of B’s predecessors
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**

- Want to compute abstract values at every program point

- Initialize all abstract states to $\bot$

- Repeat until no abstract state changes at any program point:
  - Compute abstract state on entry to a basic block $B$ by taking the join of $B$'s predecessors
  - Symbolically execute each basic block using abstract semantics
\begin{verbatim}
x = 0;
y = 0;
while(y <= n) 
{ 
   if (z == 0) {
      x = x+1;
   }
   else {
      x = x + y;
   }
   y = y+1 
}
\end{verbatim}
An Example

Is \( x \) always non-negative inside the loop?

\[
x = 0; \\
y = 0; \\
\text{while}(y \leq n) \\
{ \\
    \text{if} (z == 0) \\
        \{ x = x+1; \} \\
    \text{else} \{ x = x + y; \} \\
    y = y+1 \\
}\]
Fixed-Point Computation

```
x = 0
y = 1
loop head
```
Fixed-Point Computation

\[ x = 0, y = 1 \]

\[ x = x + 1, y = x + y \]

\[ y = y + 1 \]
Fixed-Point Computation

\[ x = \bot, \quad y = \bot \]
\[ x = 0, \quad y = 1 \]
\[ x = \bot, \quad y = \bot \]

\[ x = \bot, \quad y = \bot \]
\[ x = Z, \quad y = \bot \]
\[ x = \bot, \quad y = \bot \]

\[ x = \bot, \quad y = \bot \]
\[ x = x + 1 \]
\[ x = \bot, \quad y = \bot \]
\[ x = \bot, \quad y = \bot \]

\[ x = \bot, \quad y = \bot \]
\[ x = x + y \]
\[ x = \bot, \quad y = \bot \]
\[ x = \bot, \quad y = \bot \]

\[ x = \bot, \quad y = \bot \]
\[ y \leq n \]

\[ x = \bot, \quad y = \bot \]
\[ z = 0 \]
\[ x = \bot, \quad y = \bot \]
\[ x = \bot, \quad y = \bot \]

\[ x = \bot, \quad y = \bot \]
\[ z \neq 0 \]
\[ x = \bot, \quad y = \bot \]
\[ x = \bot, \quad y = \bot \]

\[ x = \bot, \quad y = \bot \]
\[ y = y + 1 \]

\[ x = \bot, \quad y = \bot \]
\[ x = \bot, \quad y = \bot \]

\[ x = \bot, \quad y = \bot \]
\[ x = \bot, \quad y = \bot \]
Fixed-Point Computation

\[ x = 0 \]
\[ y = 1 \]

**Loop Head**

- If \( y \leq n \)
  - **Branch**
    - If \( z = 0 \)
      - **Loop End**
        - \( x = x + 1 \)
        - \( y = y + 1 \)
    - If \( z \neq 0 \)
      - **Exit Block**
        - \( x = x + y \)

**Exit Block**

- \( x = 0, y = 1 \)

- \( x = z, y = 1 \)

- \( x = z, y = P \)
Fixed-Point Computation

$x = \bot, y = \bot$
$x = Z, y = \bot$
$x = Z, y = P$

$x = 0$
$y = 1$

Branch

$y \leq n$

Loop head

Exit block

$z = 0$
$z \neq 0$

Loop end

$y = y + 1$

$x = Z, y = P$

$x = x + 1$

$x = x + y$

$x = \bot, y = \bot$

$x = \bot, y = \bot$

$x = \bot, y = \bot$

$x = \bot, y = \bot$

$x = \bot, y = \bot$

$x = \bot, y = \bot$

$x = \bot, y = \bot$

$x = \bot, y = \bot$
Fixed-Point Computation

\[ x = \bot, y = \bot \]
\[ x = Z, y = \bot \]
\[ x = Z, y = P \]

\[ x = 0, y = 1 \]

\[ x = x + 1, x = x + y \]

\[ x = Z, y = P \]

\[ y \leq n \]
\[ z = 0, z \neq 0 \]

\[ x = Z, y = P \]

\[ x = Z, y = P \]
\[ x = Z, y = P \]
\[ x = Z, y = P \]
\[ x = Z, y = P \]
\[ x = Z, y = P \]
\[ x = Z, y = P \]

\[ x = \bot, y = \bot \]

\[ x = \bot, y = \bot \]
\[ x = \bot, y = \bot \]

\[ x = \bot, y = \bot \]
Fixed-Point Computation

\[
\begin{align*}
&x = \perp, y = \perp \\
&x = z, y = \perp \\
&x = z, y = p \\
&x = 0, y = 1 \\
&\text{loop head} \\
&\text{exit block} \\
&\text{branch} \\
&\text{loop end} \\
&y \leq n \\
&z = 0 \\
&z \neq 0 \\
&x = x + 1 \\
&x = x + y \\
&y = y + 1
\end{align*}
\]
Fixed-Point Computation

\[ x = 0, y = 1 \]

**Loop Head**

- \( x = \bot, y = \bot \)
- \( x = Z, y = \bot \)
- \( x = Z, y = P \)

**Exit Block**

- \( x = Z, y = P \)

**Branch**

- \( y \leq n \)
- \( z = 0 \)
- \( z \neq 0 \)

**Loop End**

- \( x = Z, y = P \)
- \( y = y + 1 \)
- \( x = P, y = P \)
Fixed-Point Computation

\[
\begin{align*}
\text{x} &= 0, \text{y} = 1 \\
\text{loop head} \\
\text{exit block} \\
\text{branch} \\
\text{loop end} \\
y &\leq n \\
z &= 0, z \neq 0
\end{align*}
\]
Fixed-Point Computation

\[
x = 0, \ y = 1
\]

\[
x = Z, \ y = \bot
\]

\[
x = Z, \ y = P
\]

\[
x = Z, \ y = P
\]

\[
x = Z, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]

\[
x = P, \ y = P
\]
Fixed-Point Computation

\[
\begin{align*}
x &= \perp, y &= \perp \\
x &= Z, y &= \perp \\
x &= Z, y &= P
\end{align*}
\]
Fixed-Point Computation

\[
x = \perp, y = \perp
\]
\[
x = Z, y = \perp
\]
\[
x = Z, y = P
\]

\[
x = 0
\]
\[
y = 1
\]

\[
\text{loop head}
\]

\[
x = x + 1
\]
\[
x = x + y
\]

\[
\text{exit block}
\]

\[
y \leq n
\]

\[
\text{branch}
\]

\[
z = 0
\]
\[
z \neq 0
\]

\[
x = Z, y = P
\]

\[
\text{Fixed point!}
\]

\[
x = P, y = P
\]

\[
x = P, y = P
\]

\[
x = P, y = P
\]

\[
x = P, y = P
\]

\[
x = NN, y = P
\]

\[
x = P, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = 0
\]
\[
y = 1
\]

\[
\text{loop head}
\]

\[
x = x + 1
\]
\[
x = x + y
\]

\[
\text{exit block}
\]

\[
y \leq n
\]

\[
\text{branch}
\]

\[
z = 0
\]
\[
z \neq 0
\]

\[
x = Z, y = P
\]

\[
\text{Fixed point!}
\]

\[
x = P, y = P
\]

\[
x = P, y = P
\]

\[
x = P, y = P
\]

\[
x = P, y = P
\]

\[
x = NN, y = P
\]

\[
x = P, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]

\[
x = NN, y = P
\]
In this example, we quickly reached least fixed point – but does this computation always terminate?
In this example, we quickly reached least fixed point – but does this computation always terminate?

Yes, assuming abstract domain forms complete lattice.
In this example, we quickly reached least fixed point – but does this computation always terminate?

- Yes, assuming abstract domain forms **complete lattice**
- This means every subset of elements (including infinite subsets) have a LUB
In this example, we quickly reached least fixed point – but does this computation always terminate?

- Yes, assuming abstract domain forms complete lattice

- This means every subset of elements (including infinite subsets) have a LUB

- Unfortunately, many interesting domains do not have this property, so we need widening operators for convergence.
Lessons To Take Away

- Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:
Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:

- **Abstraction**: Only reason about certain properties of interest
Lessons To Take Away

Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:

- **Abstraction**: Only reason about certain properties of interest

- **Fixed-point computation**: Allows us to obtain sound overapproximation of the program
Lessons To Take Away

- Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:
  - **Abstraction**: Only reason about certain properties of interest
  - **Fixed-point computation**: Allows us to obtain sound overapproximation of the program

- But many static analyses also differ in several ways:
Lessons To Take Away

Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:

- **Abstraction**: Only reason about certain properties of interest
- **Fixed-point computation**: Allows us to obtain sound overapproximation of the program

But many static analyses also differ in several ways:

- **Flow (in)sensitivity**: Some analyses only compute facts for the whole program, not for every program point
Lessons To Take Away

- Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:
  - **Abstraction:** Only reason about certain properties of interest
  - **Fixed-point computation:** Allows us to obtain sound overapproximation of the program

- But many static analyses also differ in several ways:
  - **Flow (in)sensitivity:** Some analyses only compute facts for the whole program, not for every program point
  - **Path sensitivity:** More precise analyses compute different facts for different program paths
Lessons To Take Away

Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:

- **Abstraction**: Only reason about certain properties of interest
- **Fixed-point computation**: Allows us to obtain sound overapproximation of the program

But many static analyses also differ in several ways:

- **Flow (in)sensitivity**: Some analyses only compute facts for the whole program, not for every program point
- **Path sensitivity**: More precise analyses compute different facts for different program paths
- **Analysis direction**: Forwards vs. backwards
Many open problems in program analysis
Challenges and Open Problems

Many open problems in program analysis

- Precise and scalable heap reasoning
Many open problems in program analysis

- Precise and scalable heap reasoning
- Concurrency
Challenges and Open Problems

Many open problems in program analysis

- Precise and scalable heap reasoning
- Concurrency
- Dealing with open programs

....

Exciting area with lots of interesting topics to work on!
Many open problems in program analysis

- Precise and scalable heap reasoning
- Concurrency
- Dealing with open programs
- Modular program analysis
Many open problems in program analysis

- Precise and scalable heap reasoning
- Concurrency
- Dealing with open programs
- Modular program analysis
- ...
Many open problems in program analysis

- Precise and scalable heap reasoning
- Concurrency
- Dealing with open programs
- Modular program analysis
- …

Exciting area with lots of interesting topics to work on!
Considering PhD or Postdoc?

If you are interested in program analysis or verification, consider applying to UT Austin!