Static Analysis: an Abstract Interpretation Perspective

Kwangkeun Yi

Seoul National University

5/2019@The 9th SSFT
This lecture is based on the following forthcoming book

*Static Analysis: an Abstract Interpretation Perspective*, Yi and Rival, MIT Press
1 Introduction

2 Static Analysis: a Gentle Introduction

3 A General Framework in Transitional Style

4 A Technique for Scalability: Sparse Analysis

5 Specialized Frameworks
Outline

1 Introduction

2 Static Analysis: a Gentle Introduction

3 A General Framework in Transitional Style

4 A Technique for Scalability: Sparse Analysis

5 Specialized Frameworks
Our Interest

How to verify specific properties about program executions before execution:

- absence of run-time errors i.e., no crashes
- preservation of invariants

Verification

Make sure that $[P] \subseteq S$ where

- the semantics $[P]$ = the set of all behaviors of $P$
- the specification $S$ = the set of acceptable behaviors
Semantics $[P]$ and Semantic Properties $S$

Semantics $[P]$:
- compositional style (“denotational”)
  $[AB] = \cdots [A] \cdots [B] \cdots$
- transitional style (“operational”)
  $[AB] = \{s_0 \leftrightarrow s_1 \leftrightarrow \cdots, \cdots\}$

Semantic properties $S$:
- safety
  - some behavior observable in finite time will never occur.
- liveness
  - some behavior observable after infinite time will never occur.
Safety Properties

Some behavior observable in \textit{finite} time will never occur.

**Examples:**

- absence of crashing error  
  e.g., no uncaught exceptions in ML, no memory errors in C  

- preservation of a general invariant  
  e.g., some data structure should never get broken  

- assertion on variable values  
  e.g., the values of a variable always in a given range
Liveness Properties

Some behavior observable after *infinite* time will never occur.

**Examples:**

- no unbounded repetition of a given behavior
- no starvation
- no non-termination
“Analysis is sound.” “Analysis is complete.”

- **Soundness**: \( \text{analysis}(P) = \text{yes} \implies P \text{ satisfies the specification} \)
- **Completeness**: \( \text{analysis}(P) = \text{yes} \iff P \text{ satisfies the specification} \)
Spectrum of Program Analysis Techniques

- testing
- machine-assisted proving
- finite-state model checking
- conservative static analysis
- bug-finding
Testing

Approach

1. Consider finitely many, finite executions
2. For each of them, check whether it violates the specification

- If the finite executions find no bug, then accept.
- **Unsound**: can accept programs that violate the specification
- **Complete**: does not reject programs that satisfy the specification
Introduction

Machine-Assisted Proving

Approach

1. Use a specific language to formalize verification goals
2. Manually supply proof arguments
3. Let the proofs be automatically verified

- tools: Coq, Isabelle/HOL, PVS, ...
- **Applications**: CompCert (certified compiler), seL4 (secure micro-kernel), ...
- **Not automatic**: key proof arguments need to be found by users
- **Sound**: if the formalization is correct
- **Quasi-complete** (only limited by the expressiveness of the logics)
Finite-State Model Checking

**Introduction**

Focus on **finite state models** of programs

Perform **exhaustive exploration** of program states

- **Automatic**
- **Sound or complete**, only with respect to the finite models
- But, software has $\sim \infty$ states: need finite approximation or non-termination
Conservative Static Analysis

Principle

1. Perform automatic verification, yet which may fail
2. Compute a conservative approximation of the program semantics

- Either sound or complete, not both
- Sound & incomplete static analysis is common:
  - ML type systems, Astrée, Sparrow, Facebook Infer, ...
  - optimizing compilers relies on it
- Automatic
- Incompleteness: may reject safe programs
  or may raise false alarms
- Analysis algorithms reason over program semantics
Bug Finding

Approach

Automatic, unsound and incomplete algorithms

- Coverity, CodeSonar, SparrowFasoo, ...
- **Automatic and generally fast**
- **No mathematical guarantee about the results**
  - may reject a correct program, and accept an incorrect one
  - may raise false alarm and fail to report true violations
- Used to increase software quality without any guarantee
## High-level Comparison

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>sound</th>
<th>complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>testing</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>machine-assisted proving</td>
<td>no</td>
<td>yes</td>
<td>yes/no</td>
</tr>
<tr>
<td>finite-state model checking</td>
<td>yes</td>
<td>yes</td>
<td>yes/no</td>
</tr>
<tr>
<td><strong>conservative static analysis</strong></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>bug-finding</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Focus of This Lecture: Conservative Static Analysis

A general technique, for any programming language $\mathbb{L}$ and safety property $S$, that

- **checks**, for input program $P$ in $\mathbb{L}$, if $[P] \subseteq S$,
- **automatic** (algorithm)
- **finite** (terminating)
- **sound** (guarantee)
- **malleable** for arbitrary precision

A forthcoming framework
Will guide us how to design such static analysis.
Problem: How to Finitely Compute $[P]$ Beforehand

- Finite & exact computation $\text{Exact}(P)$ of $[P]$ is impossible, in general.

  For a Turing-complete language $\mathbb{L}$,
  
  $\not\exists$ algorithm $\text{Exact} : \text{Exact}(P) = [P]$ for all $P$ in $\mathbb{L}$.

- Otherwise, we can solve the Halting Problem.
  - Given $P$, see if $\text{Exact}(P; 1/0)$ has divide-by-zero.
Answers: Conservative Static Analysis

Technique for **finite sound estimation** $[P]^\#$ of $[P]$

- “finite”, hence
  - automatic (algorithm) &
  - static (without executing $P$)

- “sound”
  - over-approximation of $[P]$

Hence, ushers us to sound analysis:

$$(\text{analysis}(P) = \text{check } [P]^\# \subseteq S) \implies (P \text{ satisfies property } S)$$
Introduction

Need Formal Frameworks of Static Analysis (1/2)

Suppose that

- We are interested in the value ranges of variables.
- How to finitely estimate $[P]$ for the property?

You may, intuitively:

```c
x = readInt;
1:
   while (x <= 99)
2:
      x++;
3:
   end
4:

Capture the dynamics by abstract equations; solve; reason.

$x_1 = [-\infty, +\infty]$ or $x_3$
$x_2 = x_1 \text{ and } [-\infty, 99]$
$x_3 = x_2 + 1$
$x_4 = x_1 \text{ and } [100, +\infty]$
```
Abstract Interpretation [CousotCousot]: a powerful design theory

- How to derive correct yet arbitrarily precise equations?
  - Non-obvious: ptrs, heap, exns, high-order ftns, etc.

```plaintext
x = readInt;
while (x ≤ 99) {
  x++;
}
```

- Define an abstract semantics function $\hat{F}$ s.t. 

- How to solve the equations in a finite time?

```plaintext
x_1 = [-∞, +∞] or x_3
x_2 = x_1 and [-∞, 99]
x_3 = x_2 + 1
x_4 = x_1 and [100, +∞]
```

- Fixpoint iterations for an upperbound of $\text{fix}\hat{F}$
Outline

1. Introduction

2. Static Analysis: a Gentle Introduction

3. A General Framework in Transitional Style

4. A Technique for Scalability: Sparse Analysis

5. Specialized Frameworks
Example Language

\[
p ::= \init(\mathcal{R}) \quad \text{initialization, with a state in } \mathcal{R} \\
| \ \text{translation}(u, v) \quad \text{translation by vector } (u, v) \\
| \ \text{rotation}(u, v, \theta) \quad \text{rotation by center } (u, v) \text{ and angle } \theta \\
| \ p ; p \quad \text{sequence of operations} \\
| \ \{p\} \text{or}\{p\} \quad \text{non-deterministic choice} \\
| \ \text{iter}\{p\} \quad \text{non-deterministic iterations}
\]
Example (Semantics)

\[
\begin{align*}
\text{init}([0, 1] \times [0, 1]); \\
\text{translation}(1, 0); \\
\text{iter} \{ \\
\quad \text{translation}(1, 0) \\
\} \text{ or } \{ \\
\quad \text{rotation}(0, 0, 90^\circ) \\
\}
\end{align*}
\]
Analysis Goal Is Safety Property: Reachability

Analyze the set of reachable points, to check if the set intersects with a no-fly zone. Suppose that the no-fly zone is:
Correct or Incorrect Executions

(a) An incorrect execution

(b) Correct executions
An Example Safe Program

Example

```plaintext
init([0, 1] × [0, 1]);
iter{
    translation(1, 0);
} or {
    translation(0.5, 0.5);
}
```

Graph showing the progression of the translation operations.
How to Finitely Over-Approximate the Set of Reachable Points?

Definition (Abstraction)

We call abstraction a set $\mathcal{A}$ of logical properties of program states, which are called abstract properties or abstract elements. A set of abstract properties is called an abstract domain.

Definition (Concretization)

Given an abstract element $a$ of $\mathcal{A}$, we call concretization the set of program states that satisfy it. We denote it by $\gamma(a)$. 
Abstraction Example 1: Signs Abstraction

(c) Concretization of $[x \leq 0, y \geq 0]$  
(d) Concretization of $[x \geq 0]$

Figure: Signs abstraction
Abstraction Example 2: Interval Abstraction

The abstract elements: conjunctions of non-relational inequality constraints: $c_1 \leq x \leq c_2, c'_1 \leq y \leq c'_2$

(a) Concretization of $[1 \leq x \leq 3, 1 \leq y \leq 2]$

(b) Concretization of $[1 \leq x \leq 2]$

(c) Concretization of $[1 \leq x, 1 \leq y]$

Figure: Intervals abstraction
Abstraction Example 3: Convex Polyhedra Abstraction

The abstract elements: conjunctions of linear inequality constraints:
\[ c_1x + c_2y \leq c_3 \]

Figure: Convex polyhedra abstraction
An Example Program, Again

Example

```plaintext
init([0, 1] \times [0, 1]);
iter{
    {translation(1, 0);
    } or {
        translation(0.5, 0.5);
    }
}
```

Figure: Reachable states
Abstractions of the Semantics of the Example Program

Figure: Program’s reachable states and abstraction
Sound Analysis Function for the Example Language

- Input: a program \( p \) and an abstract area \( a \) (pre-state)
- Output: an abstract area \( a' \) (post-state)

**Definition (Sound analysis)**

An analysis is sound if and only if it captures the real executions of the input program.

If an execution of \( p \) moves a point \((x, y)\) to point \((x', y')\), then for all abstract element \( a \) such that \((x, y) \in \gamma(a), (x', y') \in \gamma(\text{analysis}(p, a))\)
Sound Analysis Function as a Diagram

\[
\text{If } \quad a_{\text{pre}} \quad \text{abstraction} \quad (x, y) \xrightarrow{\text{run } p} (x', y') \quad \text{then} \quad \text{abstraction} \quad a_{\text{post}} = \text{analysis}(p, a_{\text{pre}}) \\
\]

**Figure:** Sound analysis of a program $p$
Abstract Semantics Computation

Recall the example language

\[
p ::= \text{init}(R) \quad \text{initialization, with a state in } R \\
    | \text{translation}(u, v) \quad \text{translation by vector } (u, v) \\
    | \text{rotation}(u, v, \theta) \quad \text{rotation defined by center } (u, v) \text{ and angle } \theta \\
    | p; p \quad \text{sequence of operations} \\
    | \{p\}\text{or}\{p\} \quad \text{non-deterministic choice} \\
    | \text{iter}\{p\} \quad \text{non-deterministic iterations}
\]

**Approach**

A sound analysis for a program is constructed by computing sound abstract semantics of the program’s components.
Select, if any, the best abstraction of the region $R$.

For the example program with the intervals or convex polyhedra abstract domains, analysis of $\text{init}([0, 1] \times [0, 1])$ is

$$\text{analysis}(\text{init}(R), a) = \text{best abstraction of the region } R$$
Abstract Semantics Computation: \( \text{translation}(u, v) \)

(a) Concrete semantics
(b) Intervals
(c) Convex polyhedra

\[
\text{analysis}(\text{translation}(u, v), a) = \begin{cases} 
\text{return an abstract state that contains} \\
\text{the translation of } a 
\end{cases}
\]
Abstract Semantics Computation: rotation\((u, v, \theta)\)

\[
\text{analysis}(\text{rotation}(u, v, \theta), a) = \begin{cases} 
\text{return an abstract state that contains} \\
\text{the rotation of } a
\end{cases}
\]
Abstract Semantics Computation: \{p\} or \{p\}

\begin{align*}
\text{analysis(}\{p_0\} \text{ or } \{p_1\}, a) &= \text{union}\left(\text{analysis}(p_1, a), \text{analysis}(p_0, a)\right)
\end{align*}
Abstract Semantics Computation: \( p_0 ; p_1 \)

\[
\text{analysis}(p_0; p_1, a) = \text{analysis}(p_1, \text{analysis}(p_0, a))
\]
Abstract Semantics Computation: $\text{iter}\{p\}$ (1/5)

$\text{iter}\{p\}$ is equivalent to

\[
\begin{align*}
&\{\} \\
\text{or}\{p\} \\
\text{or}\{p; p\} \\
\text{or}\{p; p; p\} \\
\text{or}\{p; p; p; p\} \\
\vdots
\end{align*}
\]
Abstract Semantics Computation: \texttt{iter}\{p\} (2/5)

Example (Abstract iteration)

\begin{verbatim}
init({(x,y) | 0 \leq y \leq 2x and x \leq 0.5});
iter{
    translation(1,0.5)
}
\end{verbatim}

Figure: Abstract iteration
Abstract Semantics Computation: \( \text{iter}\{p\} \) (3/5)

Recall

\[
\text{iter}\{p\} = \{\} \text{ or } \{p\} \text{ or } \{p; p\} \text{ or } \cdots = \lim_i p_i
\]

where

\[
p_0 = \{\} \quad p_{k+1} = p_k \text{ or } \{p_k; p\}
\]

Hence,

\[
\text{analysis}(\text{iter}\{p\}, a) = \begin{cases}
R \leftarrow a; \\
\text{repeat} \\
\quad T \leftarrow R; \\
\quad R \leftarrow \text{widen}(R, \text{analysis}(p, R)); \\
\text{until inclusion}(R, T) \\
\text{return } T;
\end{cases}
\]

operator \text{widen} \quad \begin{cases}
\text{over approximates unions} \\
\text{enforces finite convergence}
\end{cases}
Example (Abstract iteration with widening)

\[
\begin{align*}
\text{init}(\{(x, y) \mid 0 \leq y \leq 2x \text{ and } x \leq 0.5\}); \\
\text{iter}\{ \\
\quad \text{translation}(1, 0.5) \\
\}
\end{align*}
\]

- The constraints \(0 \leq y\) and \(y \leq 2x\) are stable after iteration 1; thus, they are preserved.
- The constraint \(x \leq 0.5\) is not preserved; thus, it is discarded.

Figure: Abstract iteration with widening
Example (Loop unrolling)

\[
\text{init}(\{(x, y) \mid 0 \leq y \leq 2x \text{ and } x \leq 0.5\}); \\
\{\} \text{ or } \{ \text{translation}(1, 0.5) \}; \\
\text{iter}\{ \text{translation}(1, 0.5) \}
\]

**Figure**: Abstract iteration with widening and unrolling

(a) Iteration 0  
(b) Iteration 1, union  
(c) Iteration 2, widen, limit
Abstract Semantics Function $\text{analysis}$ at a Glance

The $\text{analysis}(p, a)$ is finitely computable and sound.

\[
\begin{align*}
\text{analysis}(\text{init}(R), a) & = \text{best abstraction of the region } R \\
\text{analysis}(\text{translation}(u, v), a) & = \begin{cases} \\
\text{return an abstract state that contains} & \\
\text{the translation of } a & \\
\text{analysis}(\text{rotation}(u, v, \theta), a) & = \begin{cases} \\
\text{return an abstract state that contains} & \\
\text{the rotation of } a & \\
\text{analysis}\{p_0\} \cup \{p_1\}, a) & = \text{union(analysis}(p_1, a), \text{analysis}(p_0, a)) \\
\text{analysis}(p_0; p_1, a) & = \text{analysis}(p_1, \text{analysis}(p_0, a)) \\
\text{analysis}(\text{iter}\{p\}, a) & = \begin{cases} \\
R \leftarrow a; & \\
\text{repeat} & \\
T \leftarrow R; & \\
R \leftarrow \text{widen}(R, \text{analysis}(p, R)); & \\
\text{until inclusion}(R, T) & \\
\text{return } T; & \\
\end{cases}
\end{cases}
\end{align*}
\]

**Sound analysis**

If an execution of $p$ from a state $(x, y)$ generates the state $(x', y')$, then for all abstract element $a$ such that $(x, y) \in \gamma(a)$,

\[(x', y') \in \gamma(\text{analysis}(p, a))\]
Verification of the Property of Interest

- Does program compute a point inside no-fly zone $\mathcal{D}$?
- Need to collect the set of reachable points.
- Run $\text{analysis}(p, -)$ and collect all points $\mathcal{R}$ from every call to $\text{analysis}$.
- Since $\text{analysis}$ is sound, the result is an overapproximation of the reachable points.
- If $\mathcal{R} \cap \mathcal{D} = \emptyset$, then $p$ is verified. Otherwise, we don't know.

(a) A $\mathcal{R}$  
(b) A more precise $\mathcal{R}$
Semantics Style: Compositional Versus Transitional

- Compositional semantics function analysis:
  - Semantics of $p$ is defined by the semantics of the sub-parts of $p$.
  \[
  [AB] = \cdots [A] \cdots [B] \cdots
  \]
  - Proving its soundness is thus by structural induction on $p$.

- For some realistic programming languages, even defining their compositional ("denotational") semantics is a hurdle.
  - gotos, exceptions, function calls

Transitional-style ("operational") semantics avoids the hurdle

\[
[AB] = \{ s_0 \leftrightarrow s_1 \leftrightarrow \cdots , \cdots \}
\]
Example Language, Again

\[
p ::= \text{init}(\mathcal{R}) \quad \text{initialization, with a state in } \mathcal{R}
\]
\[
| \quad \text{translation}(u,v) \quad \text{translation by vector } (u,v)
\]
\[
| \quad \text{rotation}(u,v,\theta) \quad \text{rotation by center } (u,v) \text{ and angle } \theta
\]
\[
| \quad p ; p \quad \text{sequence of operations}
\]
\[
| \quad \{p\} \text{or}\{p\} \quad \text{non-deterministic choice}
\]
\[
| \quad \text{iter}\{p\} \quad \text{non-deterministic iterations}
\]
Semantics as State Transitions

**Definition (Transitional semantics)**

An execution of a program is a sequence of transitions between states.

- a state is a pair \((l, p)\) of statement label \(l\) and an \((x, y)\) point \(p\).
- a single transition
  \[
  (l, p) \leftrightarrow (l', p')
  \]
  whenever the program statement at \(l\) moves the point \(p\) to \(p'\).

States \(s_1, s_6, s_9, \text{ and } s_{12}\) are initial states.

**Figure:** Transition sequences and the set of occurring states
Statement Labels

(a) Text view, with labels

(b) Graph view, with labels

Figure: Example program with statement labels
States in a Transition Sequence

(a) State \((1, p_1)\)

(b) State \((2, p_1)\)

(c) State \((4, p_1)\)

(d) State \((1, p_3)\)

(e) State \((5, p_3)\)
Reachability Problem and Abstraction of States

- Reachability problem: compute the set of all states that can occur during all transition sequences of the input program.
- An abstract state is a set of pairs of statement labels and abstract preconditions.

Collection of all states:

Statement-wise collection:

Statement-wise abstraction:
Abstract State Transition

\[ \text{Step}^{\#}: \text{ a set of pairs of labels and abstract pre conditions } \]
\[ \mapsto \]
\[ \text{a set of pairs of labels and abstract post conditions} \]

is

\[ \text{Step}^{\#}(X) = \{ x' \mid x \in X, x \rightarrow^{\#} x' \} \]

where

\[
\begin{align*}
(\text{or}_l, a_{\text{pre}}) & \rightarrow^{\#} (\text{next}(l), a_{\text{pre}}) \\
(\text{iter}_l, a_{\text{pre}}) & \rightarrow^{\#} (\text{next}(l), a_{\text{pre}}) \\
(p_l, a_{\text{pre}}) & \rightarrow^{\#} (\text{next}(l), \text{analysis}(p_l, a_{\text{pre}}))
\end{align*}
\]
Analysis by Global Iterations

The analysis goal is to accumulate from the initial abstract state $I$:

$$\text{Step}^{\#0}(I) \cup \text{Step}^{\#1}(I) \cup \text{Step}^{\#2}(I) \cup \ldots$$

which is the limit $C_\infty$ of $C_i = \text{Step}^{\#0}(I) \cup \text{Step}^{\#1}(I) \cup \cdots \cup \text{Step}^{\#i}(I)$ where

$$C_{k+1} = C_k \cup \text{Step}^{\#}(C_k).$$

Thus the analysis algorithm should iterate the operation

$$C \leftarrow C \cup \text{Step}^{\#}(C)$$

from $I$ until stable:

$$\text{analysis}_T(p, I) = \begin{cases} 
    C \leftarrow I \\
    \text{repeat} \\
    \quad R \leftarrow C \\
    \quad C \leftarrow \text{widen}_T(C, \text{Step}^{\#}(C)) \\
    \text{until } \text{inclusion}_T(C, R) \\
    \text{return } R 
\end{cases}$$

where $\text{widen}_T$ over-approximates unions and enforces finite convergence.
Analysis in Action

(f) State $(1, a_1)$

(g) States $(2, a_1)$ and $(5, a_1)$

(h) States $(3, a_1)$ and $(4, a_1)$

(i) States $(1, a_2)$ and $(1, a_3)$

(j) State $(1, \text{union} \{(a_2, a_3)\})$

(k) State $(1, \text{union} \{(a_1, a_2, a_3)\})$
Principles of a Static Analysis, Sketchy

- **Selection of the semantics and properties of interest:**
  - define the behaviors of programs
  - define the properties that need to be verified
  - formal definitions

- **Choice of the abstraction:**
  - define the space of abstract elements over which the abstract semantics is defined
  - define what the abstract elements mean
  - define abstract semantics and prove its soundness

- **Derivation of the analysis algorithms from the semantics and from the abstraction:**
  - algorithm follows the semantic formalism in use
  - e.g., compositional algorithm in the style of program interpreter
  - e.g., transitional algorithm by a monolithic, global iterations
Outline

1. Introduction
2. Static Analysis: a Gentle Introduction
3. A General Framework in Transitional Style
4. A Technique for Scalability: Sparse Analysis
5. Specialized Frameworks
Transitional Semantics

State transition sequence

\[ s_0 \xrightarrow{\cdot} s_1 \xrightarrow{\cdot} s_2 \xrightarrow{\cdot} \cdots \]

where $\xrightarrow{\cdot}$ is a transition relation between states $\mathcal{S}$

\[ \xrightarrow{\cdot} \subseteq \mathcal{S} \times \mathcal{S} \]

A state $s \in \mathcal{S}$ of the program is a pair $(l, m)$ of a program label $l$ and the machine state $m$ at that program label during execution.
Concrete Transition Sequence

Example
Consider the following program

```
input(x);
while (x ≤ 99)
{x := x + 1}
```

Let labels be “program points”. Such labeled representations of this program in graph is

Let the initial state be the empty memory $\emptyset$. Some transition sequences are:

For input 100: $(0, \emptyset) \rightarrow (1, x \mapsto 100) \rightarrow (3, x \mapsto 100)$.
For input 99: $(0, \emptyset) \rightarrow (1, x \mapsto 99) \rightarrow (2, x \mapsto 99) \rightarrow (1, x \mapsto 100) \rightarrow (3, x \mapsto 100)$.
For input 0: $(0, \emptyset) \rightarrow (1, x \mapsto 0) \rightarrow (2, x \mapsto 0) \rightarrow (1, x \mapsto 1) \rightarrow \cdots \rightarrow (3, x \mapsto 100)$. 
A General Framework in Transitional Style

Reachable States

\[
\begin{align*}
0 & \quad \text{input(x)} \\
1 & \quad \text{while } (x \leq 99) \\
2 & \quad x := x + 1
\end{align*}
\]

Assume that the possible inputs are 0, 99, and 100. Then, the set of all reachable states are the set of states occurring in the three transition sequences:

\[
\begin{align*}
&\{(0, \emptyset), (1, x \mapsto 100), (3, x \mapsto 100)\} \\
\cup &\{(0, \emptyset), (1, x \mapsto 99), (2, x \mapsto 99), (1, x \mapsto 100), (3, x \mapsto 100)\} \\
\cup &\{(0, \emptyset), (1, x \mapsto 0), (2, x \mapsto 0), (1, x \mapsto 1), \ldots, (2, x \mapsto 99), (1, x \mapsto 100), (3, x \mapsto 100)\} \\
= &\{(0, \emptyset), (1, x \mapsto 0), \ldots, (1, x \mapsto 100), (2, x \mapsto 0), \ldots, (2, x \mapsto 99), (3, x \mapsto 100)\}
\]
Concrete Semantics: the Set of Reachable States (1/3)

Given a program, let $I$ be the set of its initial states and $Step$ be the powerset-lifted version of $\rightarrow$:

$$Step : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$$

$$Step(X) = \{ s' \mid s \hookrightarrow s', s \in X \}$$

The set of reachable states is

$$I \cup Step^1(I) \cup Step^2(I) \cup \cdots .$$

which is, equivalently, the limit of $C_i$s

$$C_0 = I$$

$$C_{i+1} = I \cup Step(C_i)$$

which is, the least solution of

$$X = I \cup Step(X).$$
Concrete Semantics: the Set of Reachable States (2/3)

The least solution of

\[ X = I \cup \text{Step}(X) \]

is also called the least fixpoint of \( F \)

\[ F : \wp(S) \to \wp(S) \]
\[ F(X) = I \cup \text{Step}(X) \]

written as

\[ \text{lfp}F. \]

**Theorem (Least fixpoint)**

The least fixpoint \( \text{lfp}F \) of \( F(X) = I \cup \text{Step}(X) \) is

\[ \bigcup_{i \geq 0} F^i(\emptyset) \]

where \( F^0(X) = X \) and \( F^{n+1}(X) = F(F^n(X)) \).
Concrete Semantics: the Set of Reachable States (3/3)

Definition (Concrete semantics, the set of reachable states)

Given a program, let $S$ be the set of states and $\rightarrow$ be the one-step transition relation $\subseteq S \times S$. Let $I$ be the set of its initial states and $\text{Step}$ be the powerset-lifted version of $\rightarrow$:

$$\text{Step} : \wp(S) \to \wp(S)$$

$$\text{Step}(X) = \{ s' \mid s \xrightarrow{} s', s \in X \}.$$ 

Then the concrete semantics of the program, the set of all reachable states from $I$, is defined as the least fixpoint $\text{lfp}F$ of $F$

$$F(X) = I \cup \text{Step}(X).$$
Analysis Goal

Program-label-wise reachability

For each program label we want to know the set of memories that can occur at that label during executions of the input program.

- labels: “partitioning indices”
- e.g., statement labels as in programs, statement labels after loop unrolling, statement labels after function inlining
Abstract Semantics

Define the abstract semantics “homomorphically”:

\[ F : \wp(S) \rightarrow \wp(S) \]
\[ F(X) = I \cup \text{Step}(X) \]
\[ F^\# : S^\# \rightarrow S^\# \]
\[ F^\#(X^\#) = I^\# \cup ^\# \text{Step}^\#(X^\#) \]

The forthcoming framework will guide us

- conditions for \( S^\# \) and \( F^\# \)
- so that the abstract semantics is finitely computable and is an upper-approximation of concrete semantics \( \text{lfp}F \).
Abstraction of the Semantic Domain $\phi(S)$ (1/2)

$\phi(S)$ where $S = L \times M$

Label-wise (two-step) abstraction of states:

$$
\phi(L \times M) \xrightarrow{\text{abstraction}} L \rightarrow \phi(M) \xrightarrow{\text{abstraction}} L \rightarrow M^\sharp.
$$
### Abstraction of the Semantic Domain $\varphi(\mathbb{S})$ (2/2)

**A General Framework in Transitional Style**

| $\varphi(\mathbb{L} \times \mathbb{M}) \ni$ | collection of all states | \[
(0, m_0), (0, m'_0), \ldots, \text{ at 0} \\
(1, m_1), (1, m'_1), \ldots, \text{ at 1} \\
\vdots \\
(n, m_n), (n, m'_n), \ldots \text{ at } n
\]
| $\mathbb{L} \rightarrow \varphi(\mathbb{M}) \ni$ | label-wise collection | \[
(0, \{m_0, m'_0, \ldots\}) \\
(1, \{m_1, m'_1, \ldots\}) \\
\vdots \\
(n, \{m_n, m'_n, \ldots\})
\]
| $\mathbb{L} \rightarrow \mathbb{M}^\# \ni$ | label-wise abstraction | \[
(0, M_0^\#) \\
(1, M_1^\#) \\
\vdots \\
(n, M_n^\#)
\]

Each $M_l^\#$ over-approximates the set $\{m_l, m'_l, \ldots\}$ collected at label $l$. 

Kwangkeun Yi (Seoul National University) Static Analysis 5/2019@The 9th SSFT 69 / 130
Define an abstract domain as a *CPO*
  ▶ a partial order set
  ▶ has a least element \( \bot \)
  ▶ has a least-upper bound for every *chain*

An abstract domain as \( \sqcup \)-semilattices also work.
Preliminary for Abstract Domains (2/3)

Abstract and concrete domains are structured “consistently”.

Definition (Galois connection)

A *Galois connection* is a pair made of a concretization function $\gamma$ and an abstraction function $\alpha$ such that:

$$\forall c \in C, \forall a \in A, \quad \alpha(c) \sqsubseteq a \iff c \subseteq \gamma(a)$$

We write such a pair as follows:

$$(C, \subseteq) \xymatrix{\approx\ar @{-} @<0.5em> [r]^\gamma && (A, \sqsubseteq) \ar @{-} @<-0.5em> [l]_\alpha}$$
Galois-connection properties we rely on:
For

\[(\mathcal{C}, \subseteq) \xleftrightarrow{\gamma}{\alpha} (\mathcal{A}, \sqsubseteq)\]

- \(\alpha\) and \(\gamma\) are monotone functions
- \(\forall c \in \mathcal{C}, \ c \subseteq \gamma(\alpha(c))\)
- \(\forall a \in \mathcal{A}, \ \alpha(\gamma(a)) \sqsubseteq a\)
- If both \(\mathcal{C}\) and \(\mathcal{A}\) are CPOs, then \(\alpha\) is continuous.

(Proofs are in the supplementary note.)
Abstract Domains (1/2)

Design an abstract domain as a CPO that is Galois-connected with the concrete domain:

\[(\wp(L \times M), \subseteq) \xrightarrow{\alpha} (L \to M^\#, \subseteq).\]

- Abstraction \(\alpha\) defines how each concrete elmt (set of concrete states) is abstracted into an abstract elmt.
- Concretization \(\gamma\) defines the set of concrete states implied by each abstract state.
- Partial order \(\subseteq\) is the label-wise order:

\[a^\# \subseteq b^\# \iff \forall l \in L : a^\#(l) \subseteq_M b^\#(l)\]

where \(\subseteq_M\) is the partial order of \(M^\#\).
Abstract Domains (2/2)

The above Galois connection (abstraction)

\[(\varnothing(\mathbb{L} \times \mathbb{M}), \subseteq) \overset{\gamma}{\leftarrow} \overset{\alpha}{\rightarrow} (\mathbb{L} \rightarrow \mathbb{M}^\sharp, \subseteq).\]

composes two Galois connections:

\[
\begin{align*}
(\varnothing(\mathbb{L} \times \mathbb{M}), \subseteq) & \overset{\gamma_0}{\leftarrow} \overset{\alpha_0}{\rightarrow} (\mathbb{L} \rightarrow \varnothing(\mathbb{M}), \subseteq) \quad (\subseteq \text{ is the label-wise } \subseteq) \\
& \overset{\gamma_1}{\leftarrow} \overset{\alpha_1}{\rightarrow} (\mathbb{L} \rightarrow \mathbb{M}^\sharp, \subseteq) \quad (\subseteq \text{ is the label-wise } \subseteq_M)
\end{align*}
\]

Thus, boils down to

\[
(\varnothing(\mathbb{M}), \subseteq) \overset{\gamma_M}{\leftarrow} \overset{\alpha_M}{\rightarrow} (\mathbb{M}^\sharp, \subseteq_M).
\]
Abstract Semantic Functions

Let

\[(\wp(\mathbb{L} \times \mathbb{M}), \subseteq) \xleftarrow{\gamma} (\mathbb{L} \rightarrow \mathbb{M}^\#, \subseteq).\]

A concrete semantic function \(F\)

\[\begin{align*}
S &= \mathbb{L} \times \mathbb{M} \\
F &: \wp(S) \rightarrow \wp(S) \\
F(X) &= I \cup \text{Step}(X) \\
\text{Step} &= \wp(\rightarrow) \\
\rightarrow \subseteq (\mathbb{L} \times \mathbb{M}) \times (\mathbb{L} \times \mathbb{M})
\end{align*}\]

An abstract semantic function \(F^\#\)

\[\begin{align*}
S^\# &= \mathbb{L} \rightarrow \mathbb{M}^\# \\
F^\# &: S^\# \rightarrow S^\# \\
F^\#(X^\#) &= \alpha(I) \cup^\# \text{Step}^\#(X^\#) \\
\text{Step}^\# &= \wp(\text{id}, \sqcup_M) \circ \pi \circ \wp(\rightarrow^\#) \\
\rightarrow^\# \subseteq (\mathbb{L} \times \mathbb{M}^\#) \times (\mathbb{L} \times \mathbb{M}^\#)
\end{align*}\]

with relations \(\rightarrow\) and \(\rightarrow^\#\) being functions
As of $\text{Step}^\# = \varnothing(\text{id}, \sqcup_M) \circ \pi \circ \tilde{\varnothing}(\hookrightarrow^\#)$

$\text{Step}^\# : (\mathbb{L} \rightarrow \mathbb{M}^\#) \rightarrow (\mathbb{L} \rightarrow \mathbb{M}^\#)$

- Abstract transition $\tilde{\varnothing}(\hookrightarrow^\#)$:
  - a set $\subseteq \mathbb{L} \times \mathbb{M}^\# \mapsto$ a set $\subseteq \mathbb{L} \times \mathbb{M}^\#$

- Paritioning $\pi$:
  - a set $\subseteq \mathbb{L} \times \mathbb{M}^\# \mapsto$ a set $\subseteq \mathbb{L} \times \varnothing(\mathbb{M}^\#)$

- Joining $\varnothing(\text{id}, \sqcup_M)$:
  - a set $\subseteq \mathbb{L} \times \varnothing(\mathbb{M}^\#) \mapsto$ an abstract state $\in \mathbb{L} \rightarrow \mathbb{M}^\#$
Example

Suppose the program has two labels $l_1$ and $l_2$. That is, $\mathbb{L} = \{l_1, l_2\}$. Given an abstract state $\{(l_1, M_1^\#), (l_2, M_2^\#)\}$, Step$^\#$ first applies $\varnothing(\hookrightarrow^\#)$ to it:

$$\hookrightarrow^\#(l_1, M_1^\#) \cup \hookrightarrow^\#(l_2, M_2^\#).$$

Suppose $\hookrightarrow^\#(l_1, M_1^\#)$ returns $\{(l_1, M_1'^\#), (l_2, M_2''^\#)\}$ and $\hookrightarrow^\#(l_2, M_2^\#)$ returns $\{(l_1, M_2'^\#)\}$. Then the result is

$$\{(l_1, M_1'^\#), (l_2, M_2''^\#), (l_1, M_2'^\#)\}.$$

The subsequent application of the operator $\pi$ partitions the result by labels into

$$\{(l_1, \{M_1'^\#, M_2'^\#\}), (l_2, \{M_2''^\#\})\}.$$

The final organization operation $\varnothing(id, \sqcup_M)$ returns the post abstract state $\in \mathbb{L} \rightarrow M^\#$:

$$\{(l_1, M_1'^\# \sqcup_M M_2'^\#), (l_2, M_2''^\#)\}.$$

Kwangkeun Yi (Seoul National U)  Static Analysis  5/2019@The 9th SSFT  77 / 130
Conditions for Sound $\hookrightarrow^\#$ and $\cup^\#$

- Sound condition for $\hookrightarrow^\#$:
  \[ \circ (\hookrightarrow) \circ \gamma \subseteq \gamma \circ \circ (\hookrightarrow^\#) \]

- Sound condition for $\cup^\#$:
  \[ \cup \circ (\gamma, \gamma) \subseteq \gamma \circ \cup^\# \]

Pattern for the sound condition for each semantic operator $f^\# : A^\# \rightarrow B^\#$

\[ f \circ \gamma_A \subseteq_B \gamma_B \circ f^\# . \]
Then, Follows Sound Static Analysis

- In case $S^\#$ is of finite-height and $F^\#$ is monotone or extensive, then
  \[
  \bigsqcup_{i \geq 0} F^\#_i(\bot)
  \]
  is finitely computable and over-approximates the concrete semantics $\text{lfp} F$.

- Otherwise, find a widening operator $\triangledown$, then the following chain
  $X_0 \sqsubseteq X_1 \sqsubseteq \cdots$
  \[
  X_0 = \bot, \quad X_{i+1} = X_i \triangledown F^\#(X_i)
  \]
  is finite and its last element over-approximates the concrete semantics $\text{lfp} F$. 

Kwangkeun Yi (Seoul National U)  
Static Analysis  
5/2019®The 9th SSFT  79 / 130
Underlying Theorems (1/2)

Theorem (Sound static analysis by $F^\#$)

Given a program, let $F$ and $F^\#$ be defined as in the framework. If $S^\#$ is of finite-height (every chain $S^\#$ is finite) and $F^\#$ is monotone or extensive, then

$$\bigsqcup_{i \geq 0} F^{\#i}(\bot)$$

is finitely computable and over-approximates $\text{lfp}F$:

$$\text{lfp}F \subseteq \gamma(\bigsqcup_{i \geq 0} F^{\#i}(\bot)) \quad \text{or equivalently} \quad \alpha(\text{lfp}F) \subseteq \bigsqcup_{i \geq 0} F^{\#i}(\bot).$$

(Proof is in the supplementary note.)
Underlying Theorems (2/2)

Theorem (Sound static analysis by $F^\#$ and widening operator $\triangledown$)

Given a program, let $F$ and $F^\#$ be defined as in the framework. Let $\triangledown$ be a widening operator. Then the following chain $Y_0 \sqsubseteq Y_1 \sqsubseteq \cdots$

$$Y_0 = \perp \quad Y_{i+1} = Y_i \triangledown F^\#(Y_i)$$

is finite and its last element $Y_{\text{lim}}$ over-approximates $\text{lfp}F$:

$$\text{lfp}F \subseteq \gamma(Y_{\text{lim}}) \quad \text{or equivalently} \quad \alpha(\text{lfp}F) \sqsubseteq Y_{\text{lim}}.$$ 

(Proof is in the supplementary note.)
Definition (Widening operator)

A \textit{widening} operator over an abstract domain $\mathbb{A}$ is a binary operator $\triangledown$, such that:

1. For all abstract elements $a_0, a_1$, we have
   \[ \gamma(a_0) \cup \gamma(a_1) \subseteq \gamma(a_0 \triangledown a_1) \]

2. For all sequence $(a_n)_{n \in \mathbb{N}}$ of abstract elements, the sequence $(a'_n)_{n \in \mathbb{N}}$ defined below is ultimately stationary:
   \[
   \begin{align*}
   a'_0 &= a_0 \\
   a'_{n+1} &= a'_n \triangledown a_n
   \end{align*}
   \]
Analysis Algorithm Based on Global Iterations: Basic Version (1/2)

- Case: $S^\#$ is of finite-height and $F^\#$ is monotone or extensive
- Note the increasing chain

\[
\bot \sqsubseteq (F^\#)^1(\bot) \sqsubseteq (F^\#)^2(\bot) \sqsubseteq \cdots
\]

is finite and its biggest element is equal to

\[
\bigcup_{i \geq 0} F^\#^i(\bot).
\]

```plaintext
C ← ⊥
repeat
    R ← C
    C ← F^\#(C)
until C ⊑ R
return R
```
Case: $S^#$ is of infinite-height or $F^#$ is neither monotonic nor extensive

Use a widening operator $\nabla$

```plaintext
C ← ⊥
repeat
   R ← C
   C ← C $\nabla$ $F^#(C)$
until C ⊑ R
return R
```
Inefficiency of the Basic Algorithms

Recall the algorithm with $F^\#(C)$ being inlined:

\[
\begin{align*}
C & \leftarrow \bot \\
\text{repeat} & \\
R & \leftarrow C \\
C & \leftarrow C \triangledown (\varphi(id, \sqcup) \circ \pi \circ \tilde{\varphi}(\rightarrow^\#))(C) \\
\text{until } C \sqsubseteq R & \\
\text{return } R
\end{align*}
\]

- $|C| \sim$ the number of labels in the input program!
- Better apply

\[
\tilde{\varphi}(\rightarrow^#)(C)
\]

only to necessary labels
Analysis Algorithm Based on Global Iterations: Worklist Version

- worklist: the set of labels whose input memories are changed in the previous iteration

\[
C : \mathbb{L} \to \mathbb{M}^\#
\]

\[
F^# : (\mathbb{L} \to \mathbb{M}^#) \to (\mathbb{L} \to \mathbb{M}^#)
\]

\[
\text{WorkList}: \emptyset(\mathbb{L})
\]

\[
\text{WorkList} \leftarrow \mathbb{L}
\]

\[
C \leftarrow \bot
\]

repeat

\[
R \leftarrow C
\]

\[
C \leftarrow C \triangledown F^#(C|\text{WorkList})
\]

\[
\text{WorkList} \leftarrow \{l \mid C(l) \not\subseteq R(l), l \in \mathbb{L}\}
\]

until \(\text{WorkList} = \emptyset\)

return \(R\)
Improvement of the Worklist Algorithm

- Inefficient: \( \text{WorkList} \leftarrow \{ l \mid C(l) \not< R(l), l \in L \} \) re-scans all the labels.
  - Better: At application \( \mapsto \# \) to \((l, C(l))\), if its result \((l', M\#)\) is changed \((M\# \not< C(l'))\), add \(l'\) to the worklist.

- Inefficient: \( C \bigtriangleup F\#(C|_{\text{WorkList}}) \) widens at all the labels.
  - Better: Apply \( \bigtriangleup \) only at the target of a loop. Use \( \cup\# \) at other labels.
1. Define $\mathbb{M}$ to be the set of memory states that can occur during program executions. Let $\mathbb{L}$ be the finite and fixed set of labels of a given program.

2. Define a concrete semantics as the $\text{lfp} F$ where

\begin{align*}
\text{concrete domain} & \quad \wp(\mathbb{S}) = \wp(\mathbb{L} \times \mathbb{M}) \\
\text{concrete semantic function} & \quad F : \wp(\mathbb{S}) \to \wp(\mathbb{S}) \\
F(X) & = I \cup \text{Step}(X) \\
\text{Step} & = \wp(\hookrightarrow) \\
\hookrightarrow & \subseteq (\mathbb{L} \times \mathbb{M}) \times (\mathbb{L} \times \mathbb{M})
\end{align*}

The $\hookrightarrow$ is the one-step transition relation over $\mathbb{L} \times \mathbb{M}$.
Define its abstract domain and abstract semantic function as:

\[
\begin{align*}
\text{abstract domain} & \quad S^# = L \rightarrow M^# \\
\text{abstract semantic function} & \quad F^# : S^# \rightarrow S^# \\
F^#(X^#) & \quad = \quad \alpha(I) \cup^# \text{Step}^#(X^#) \\
\text{Step}^# & \quad = \quad \varnothing(id, \sqcup_M) \circ \pi \circ \varnothing(\leftarrow^#) \\
\leftarrow^# & \quad \subseteq \quad (L \times M^#) \times (L \times M^#)
\end{align*}
\]

The \(\leftarrow^#\) is the one-step abstract transition relation over \(L \times M^#\).

Function \(\pi\) partitions a set \(\subseteq L \times M^#\) by the labels in \(L\) returning an element in \(L \rightarrow \varnothing(M^#)\) represented as a set \(\subseteq L \times \varnothing(M^#)\).
Check the abstract domains $\mathcal{S}^\#$ and $\mathcal{M}^\#$ are CPOs, and forms a Galois-connection respectively with $\wp(\mathcal{S})$ and $\wp(\mathcal{M})$:

\[
(\wp(\mathcal{S}), \subseteq) \xleftrightarrow{\gamma} (\mathcal{S}^\#, \sqsubseteq) \quad \text{and} \quad (\wp(\mathcal{M}), \subseteq) \xleftrightarrow{\gamma_M} (\mathcal{M}^\#, \sqsubseteq_M)
\]

where the partial order $\sqsubseteq$ of $\mathcal{S}^\#$ is label-wise $\sqsubseteq_M$:

\[
a^\# \sqsubseteq b^\# \iff \forall l \in \mathcal{L} : a^\#(l) \sqsubseteq_M b^\#(l).
\]

Check the abstract one-step transition $\hookrightarrow^\#$ and abstract union $\bigcup^\#$ satisfy:

\[
\wp(\hookrightarrow) \circ \gamma \sqsubseteq \gamma \circ \wp(\hookrightarrow^\#)
\]

\[
\bigcup \circ (\gamma, \gamma) \sqsubseteq \gamma \circ \bigcup^\#
\]
Then, sound static analysis is defined as follows:

- In case $S^\#$ is of finite-height (every its chain is finite) and $F^\#$ is monotone or extensive, then

$$\bigcup_{i \geq 0} F^\#^i (\bot)$$

is finitely computable and over-approximates the concrete semantics $\text{lfp} F$.

- Otherwise, find a widening operator $\triangledown$, then the following chain

$$X_0 \sqsubseteq X_1 \sqsubseteq \cdots$$

$$X_0 = \bot \quad X_{i+1} = X_i \triangledown F^\#(X_i)$$

is finite and its last element over-approximates the concrete semantics $\text{lfp} F$. 
Use Example: Target Language

\[ x \in X \]  \quad \text{program variables}

\[
C ::= \\
| \text{skip} \quad \text{nop statement} \\
| C; C \quad \text{sequence of statements} \\
| x := E \quad \text{assignment} \\
| \text{input}(x) \quad \text{read an integer input} \\
| \text{if}(B)\{C\}\text{else}\{C\} \quad \text{condition statement} \\
| \text{while}(B)\{C\} \quad \text{loop statement} \\
| \text{goto } E \quad \text{goto with dynamically computed label}
\]

\[
E ::= \\
| n \quad \text{integer} \\
| x \quad \text{variable} \\
| E + E \quad \text{addition}
\]

\[
B ::= \\
| \text{true} | \text{false} \\
| E < E \quad \text{comparison} \\
| E = E \quad \text{equality}
\]

\[
P ::= C \quad \text{program}
\]

Figure: Syntax of a simple imperative language
Use Example: Concrete State Transition Semantics

\[ \text{lf} p F \]

of the continuous function

\[
F : \wp(S) \to \wp(S) \\
F(X) = I \cup \text{Step}(X) \\
\text{Step}(X) = \wp(\rightarrow).
\]

where

\[
S = L \times M
\]

and

memories \quad \mathbb{M} = X \to V \\
values \quad \mathbb{V} = Z \cup L.

The state transition relation \((l, m) \rightarrow (l', m')\) is defined as follows.

- **skip** : \((l, m) \rightarrow (\text{next}(l), m)\)
- **input(x)** : \((l, m) \rightarrow (\text{next}(l), \text{update}_x(m, z))\) for an input integer \(z\)
- **x := E** : \((l, m) \rightarrow (\text{next}(l), \text{update}_x(m, \text{eval}_E(m)))\)
- **\(C_1; C_2\)** : \((l, m) \rightarrow (\text{next}(l), m)\)
- **if(B){C_1} else{C_2}** : \((l, m) \rightarrow (\text{next}\text{True}(l), \text{filter}_B(m))\)
  - : \((l, m) \rightarrow (\text{next}\text{False}(l), \text{filter}_{\neg B}(m))\)
- **while(B){C}** : \((l, m) \rightarrow (\text{next}\text{True}(l), \text{filter}_B(m))\)
  - : \((l, m) \rightarrow (\text{next}\text{False}(l), \text{filter}_{\neg B}(m))\)
- **goto E** : \((l, m) \rightarrow (\text{eval}_E(m), m)\)
Use Example: Abstract State

An abstract domain $\mathbb{M}^\#$ is a CPO such that

$$
\left(\wp(\mathbb{M}), \subseteq\right) \xleftarrow{\gamma_M} \xrightarrow{\alpha_M} \left(\mathbb{M}^\#, \subseteq_M\right)
$$

defined as

$$
\mathbb{M}^\# \in \mathbb{M}^\# = X \rightarrow \mathbb{V}^#
$$

where $\mathbb{V}^#$ is an abstract domain that is a CPO such that

$$
\left(\wp(\mathbb{V}), \subseteq\right) \xleftarrow{\gamma_V} \xrightarrow{\alpha_V} \left(\mathbb{V}^#, \subseteq_V\right).
$$

We design $\mathbb{V}^#$ as

$$
\mathbb{V}^# = \mathbb{Z}^# \times \mathbb{L}^#
$$

where $\mathbb{Z}^#$ is a CPO that is Galois connected with $\wp(\mathbb{Z})$, and $\mathbb{L}^#$ is the powerset $\wp(\mathbb{L})$ of labels.

All abstract domains are Galois-connected CPOs, homomorphic to their concrete correspondents.
Use Example: Abstract State Transition Semantics

Case the $l$-labeled statement of

- **skip**: $(l, M^\#) \hookrightarrow^\# (\text{next}(l), M^\#)$
- **input($x$)**: $(l, M^\#) \hookrightarrow^\# (\text{next}(l), \text{update}_x^\#(M^\#, \alpha(Z)))$
- **$x := E$**: $(l, M^\#) \hookrightarrow^\# (\text{next}(l), \text{update}_x^\#(M^\#, \text{eval}_E^\#(M^\#)))$
- **$C_1; C_2$**: $(l, M^\#) \hookrightarrow^\# (\text{next}(l), M^\#)$
- **if($B$){ $C_1$} else{ $C_2$}**: $(l, M^\#) \hookrightarrow^\# (\text{nextTrue}(l), \text{filter}_B^\#(M^\#))$
- **while($B$){ $C$}**: $(l, M^\#) \hookrightarrow^\# (\text{nextFalse}(l), \text{filter}_{\neg B}^\#(M^\#))$
- **goto $E$**: $(l, M^\#) \hookrightarrow^\# (l', M^\#)$ for $l' \in L$ of $(z^\#, L) = \text{eval}_E^\#(M^\#)$

Let $F^\#$ be defined as the framework:

$$F^\#: \mathbb{S}^\# \rightarrow \mathbb{S}^\#$$
$$F^\#(S^\#) = \alpha(I) \cup^\# \text{Step}^\#(S^\#)$$
$$\text{Step}^\# = \varphi(\text{id}, \cup_M) \circ \pi \circ \tilde{\gamma}(\hookrightarrow^\#).$$

If the $\text{Step}^\#$ and $\cup^\#$ are sound abstractions of, respectively, $\text{Step}$ and $\cup$, as required by the framework:

$$\tilde{\gamma}(\hookrightarrow) \circ \gamma \subseteq \gamma \circ \tilde{\gamma}(\hookrightarrow^\#)$$
$$\cup \circ (\gamma, \gamma) \subseteq \gamma \circ \cup^\#$$

then we can use $F^\#$ to soundly approximates the concrete semantics $\text{lfp}\, F^\#$.
Use Example: Defining Sound $\leftarrow \#$

If each of the abstract semantic operators is a sound abstraction of its concrete correspondent, then $\leftarrow \#$ is a sound abstraction of $\leftarrow$:

**Theorem (Soundness of $\leftarrow \#$)**

*If the semantic operators satisfy the following soundness properties:*

\[
\varnothing (\text{eval}_E) \circ \gamma_M \subseteq \gamma_V \circ \text{eval}_E^\# \\
\varnothing (\text{update}_x) \circ \times \circ (\gamma_M, \gamma_V) \subseteq \gamma_M \circ \text{update}_x^\# \\
\varnothing (\text{filter}_B) \circ \gamma_M \subseteq \gamma_M \circ \text{filter}_B^\# \\
\varnothing (\text{filter}_{\neg B}) \circ \gamma_M \subseteq \gamma_M \circ \text{filter}_{\neg B}^\#
\]

*then* $\varnothing (\leftarrow) \circ \gamma \subseteq \gamma \circ \varnothing (\leftarrow \#)$. (*The* $\times$ *is the Cartesian product operator of two sets.*)
Use Example: Defining Sound $\cup^\#$

As of a sound $\cup^\#$, one candidate is the least upper bound operator $\sqcup$ if $\mathcal{S}^\#$ is closed by $\sqcup$, because

$$(\gamma \circ \sqcup)(a^\#, b^\#) = \gamma(a^\# \sqcup b^\#) \sqsubseteq \gamma(a^\#) \cup \gamma(b^\#) \quad \text{by the monotonicity}$$

$$= (\sqcup \circ (\gamma, \gamma))(a^\#, b^\#).$$
Outline

1 Introduction
2 Static Analysis: a Gentle Introduction
3 A General Framework in Transitional Style
4 A Technique for Scalability: Sparse Analysis
5 Specialized Frameworks
Scalability Challenge

**Figure:** Call graph of `less-382` (23,822 lines of code)
Sparse Analysis

- Exploit the semantic sparsity of the input program to analyze
- Spatial sparsity & temporal sparsity

Right part at right moment
Example Performance Gain by Sparse Analysis

- Sparrow: a “sound”, global C analyzer for the memory safety property (no overrun, no null-pointer dereference, etc.)

  http://github.com/ropas/sparrow

- ~10 hours in analyzing million lines of C
Spatial Sparcity

Each program portion accesses only a small part of the memory.
Temporal Sparcity

After the def of a memory, its use is far.
Example (Code fragment)

```c
x = x + 1;
y = y - 1;
z = x;
v = y;
ret *a + *b
```

Assume that \( a \) points to \( v \) and \( b \) to \( z \).
Spatial and Temporal Sparsity of the Example Code

(a) Without exploiting the sparsities

(b) Spatial sparsity

(c) Spatial & temporal sparsity
Exploiting Spatial Sparsity: Need $Access^\#(l)$

“abstract garbage collection”, “frame rule”

$$F^\# : (\mathbb{L} \rightarrow \mathbb{M}^\#) \rightarrow (\mathbb{L} \rightarrow \mathbb{M}^\#)$$

becomes

$$F^\#_{sparse} : (\mathbb{L} \rightarrow \mathbb{M}^\#_{sparse}) \rightarrow (\mathbb{L} \rightarrow \mathbb{M}^\#_{sparse})$$

where

$$\mathbb{M}^\#_{sparse} = \{ M^\# \in \mathbb{M}^\# | \text{dom}(M^\#) = Access^\#(l), l \in \mathbb{L} \} \cup \{ \perp \}.$$
Exploiting Temporal Sparsity: Need Def-Use Chain

Need the def-use chain information as follows.

- we streamline the abstract one-step relation

\[(l, M^\#) \xrightarrow{\#} (l', M'^\#) \quad \text{for } l' \in \text{next}^\#(l, M^\#).\]

so that the link \( \xrightarrow{\#} \) should follow the **def-use chain**:

- from (def) a label where a location is defined
- to (use) a label where the defined location is read
A Technique for Scalability: Sparse Analysis

Precision Preserving Sparse Analysis Framework

Goal

\[ F^\# : D^\# \rightarrow D^\# \xrightarrow{\text{sparsify}} F^\#_{\text{sparse}} : D^\# \rightarrow D^\# \]

\[ \text{lfp} F^\# = \text{lfp} F^\#_{\text{sparse}} \]
Precision Preserving Sparse Analysis: for Spatial Sparsity (1/3)

Need to safely estimate

\[ Access^\#(l). \]

Use yet another sound static analysis, a further abstraction:

\[
(\mathbb{L} \rightarrow M^\#, \subseteq) \xleftrightarrow{\gamma} (M^\#, \subseteq_M)
\]

(a “flow-insensitive” version of the “flow-sensitive” analysis design)
Precision Preserving Sparse Analysis: for Temporal Sparsity (2/3)

- Let
  \[ D^\# : \mathbb{L} \rightarrow \wp(\mathbb{X}) \text{ and } U^\# : \mathbb{L} \rightarrow \wp(\mathbb{X}) \]
  be the def and use sets from the original analysis.
- Need to safely estimate \( D^\# \) and \( U^\# \).
- Use yet another sound static analysis to compute
  \[ D^\#_{pre} \text{ and } U^\#_{pre} \]
  such that
  - \( \forall l \in \mathbb{L} : D^\#_{pre}(l) \supseteq D^\#(l) \) and \( U^\#_{pre}(l) \supseteq U^\#(l) \).
  - \( \forall l \in \mathbb{L} : U^\#_{pre}(l) \supseteq D^\#_{pre}(l) \setminus D^\#(l) \).
Precision Preserving Sparse Analysis: for Temporal Sparsity (3/3)

Let $D^\#_{pre}$ and $U^\#_{pre}$ be, respectively, safe def and use sets from a pre-analysis as defined before.

**Definition (Precision preserving def-use chain)**

Label $a$ to label $b$ is a def-use chain for an abstract location $\eta$ whenever $\eta \in D^\#_{pre}(a)$, $\eta \in U^\#_{pre}(b)$, and $\eta$ may not be re-defined inbetween the two labels.

**Precision preservation**

Then, the resulting sparse analysis version has the same precision as the original non-sparse analysis.
Need for the Second Condition for $D_{pre}^\#$ and $U_{pre}^\#$

(d) Original analysis def-use edge for $\eta$

\[ \eta \in D^\#(a) \quad \eta \notin D^\#(c) \quad \eta \in U^\#(b) \]

(e) Missing def-use edge (a to b) for $\eta$ because of over-approximate $D_{pre}^\#(c)$

\[ \eta \in D_{pre}^\#(a) \quad \eta \in D_{pre}^\#(c) \quad \eta \in U_{pre}^\#(b) \]

(f) Recovered def-use edge (a to b via c) for $\eta$ by safe $U_{pre}^\#(c)$

\[ \eta \in D_{pre}^\#(a) \quad \eta \in D_{pre}^\#(c) \quad \eta \in U_{pre}^\#(c) \quad \eta \in U_{pre}^\#(b) \]
Outline

1. Introduction
2. Static Analysis: a Gentle Introduction
3. A General Framework in Transitional Style
4. A Technique for Scalability: Sparse Analysis
5. Specialized Frameworks
Specialized Frameworks

Practical alternatives to the aforementioned general, abstract interpretation framework

- for simple languages and properties,
- frameworks that are simple yet powerful enough
- review of their limitations

Three specialized frameworks:

- static analysis by equations
- static analysis by monotonic closure
- static analysis by proof construction
Static Analysis by Equations

- Static analysis = equation setup and resolution
  - equations capture all the executions of the program
  - a solution of the equations is the analysis result
- Represent programs by control-flow graphs
  - nodes for semantic functions (statements)
  - edges for control flow
- Straightforward to set up sound equations

For each node

\[
\begin{align*}
y_1 &= f(x_1 \sqcup x_2) \\
y_2 &= f(x_1 \sqcup x_2)
\end{align*}
\]
Example: Data-Flow Analysis for Integer Intervals

Example (Data-flow analysis)

input (x);
while (x <= 99)
  x := x+1

Figure: Control-flow graph

Figure: A set of equations for the program

\[
\begin{align*}
x_0 &= [-\infty, +\infty] \\
x_1 &= x_0 \sqcup x_3 \\
x_2 &= x_1 \sqcap [-\infty, 99] \\
x_3 &= x_2 \oplus 1 \\
x_4 &= x_1 \sqcap [100, +\infty]
\end{align*}
\]
Limitations

Not powerful enough for arbitrary languages

- control-flow before analysis?
  - control is also computed in modern languages
  - no: the dichotomy of control being fixed and data being dynamic

- sound transformation function?
  - error prone for complicated features of modern languages
  - e.g. function call/return, function as a data, dynamic method dispatch, exception, pointer manipulation, dynamic memory allocation, ...

- lacks a systematic approach
  - to prove the correctness of the analysis
  - to vary the accuracy of the analysis
Static Analysis by Monotonic Closure (1/2)

- Static analysis = setting up initial facts then collecting new facts by a kind of chain reaction
  - has rules for collecting initial facts
  - has rules for generating new facts from existing facts
- the initial facts immediate from the program text
- the chain reaction steps simulate the program semantics
- the universe of facts are finite for each program
- analysis accumulates facts until no more possible
Specialized Frameworks

Static Analysis by Monotonic Closure (2/2)

- let $R$ be the set of the chain-reaction rules
- let $X_0$ be the initial fact set
- let $\text{Facts}$ be the set of all possible facts

Then, the analysis result is

$$\bigcup_{i \geq 0} Y_i,$$

where

$$Y_0 = X_0,$$

$$Y_{i+1} = Y \text{ such that } Y_i \vdash_R Y.$$

Or, equivalently, the analysis result is the least fixpoint

$$\bigcup_{i \geq 0} \phi^i(\emptyset)$$

of monotonic function $\phi : \wp(\text{Facts}) \to \wp(\text{Facts}) :$

$$\phi(X) = X_0 \cup (Y \text{ such that } X \vdash_R Y).$$
Example: Pointer Analysis (1/3)

- **Goal:** estimate all “points-to” relations between variables that can occur during executions
- **a → b:** variable a can point to (can have the address of) variable b

\[
P ::= C \quad \text{program} \\
C ::= L := R \quad \text{assignment} \\
| \quad C ; C \quad \text{sequence} \\
| \quad \text{while } B C \quad \text{while-loop} \\
L ::= x \mid *x \quad \text{target to assign to} \\
R ::= n \mid x \mid *x \mid &x \quad \text{value to assign} \\
B \quad \text{Boolean expression}
\]
Example: Pointer Analysis (2/3)

The initial facts that are obvious from the program text are collected by this rule:

\[
\frac{x := &y}{x \rightarrow y}
\]

The chain-reaction rules are as follows for other cases of assignments:

\[
\frac{x := y \quad y \rightarrow z}{x \rightarrow z}
\]

\[
\frac{x := \*y \quad y \rightarrow z \quad z \rightarrow w}{x \rightarrow w}
\]

\[
\frac{\*x := y \quad x \rightarrow w \quad y \rightarrow z}{w \rightarrow z}
\]

\[
\frac{\*x := \*y \quad x \rightarrow w \quad y \rightarrow z \quad z \rightarrow v}{w \rightarrow v}
\]

\[
\frac{\*x := &y \quad x \rightarrow w}{w \rightarrow y}
\]
Example: Pointer Analysis (3/3)

Example (Pointer analysis steps)

\[
\begin{align*}
\text{x} & := \&a; \quad \text{y} := \&x; \\
\text{while } B \\
\quad \text{*y} & := \&b; \\
\quad \text{*x} & := \text{*y}
\end{align*}
\]

• Initial facts are from the first two assignments:

\[
\text{x} \rightarrow a, \quad \text{y} \rightarrow x
\]

• From \( y \rightarrow x \) and the while-loop body, add

\[
\text{x} \rightarrow b
\]

• From the last assignment:
  
  ▶ from \( \text{x} \rightarrow a \) and \( \text{y} \rightarrow x \), add \( a \rightarrow a \)
  
  ▶ from \( \text{x} \rightarrow b \) and \( \text{y} \rightarrow x \), add \( b \rightarrow b \)
  
  ▶ from \( \text{x} \rightarrow a \), \( \text{y} \rightarrow x \), and \( \text{x} \rightarrow b \), add \( a \rightarrow b \)
  
  ▶ from \( \text{x} \rightarrow b \), \( \text{y} \rightarrow x \), and \( \text{x} \rightarrow a \), add \( b \rightarrow a \)
Limitations

Not powerful enough for arbitrary language
- sound rules?
  - error prone for complicated features of modern languages
  - e.g. function call/return, function as a data, dynamic method dispatch, exception, pointer manipulation, dynamic memory allocation, ...

- accuracy problem
  - consider program a set of statements, with no order between them
  - rules do not consider the control flow
  - the analysis blindly collects every possible facts when rules hold
  - accuracy improvement by more elaborate rules, but no systematic way for soundness proof
Static Analysis by Proof Construction

- Static analysis = proof construction in a finite proof system
- finite proof system = a finite set of inference rules for a predefined set of judgments
- The soundness corresponds to the soundness of the proof system.
  - the input program is provable $\Rightarrow$ the program satisfies the proven judgment.
Example: Type Inference (1/4)

\[ P ::= \begin{array}{l}
E & \text{program} \\
E ::= \begin{array}{l}
E \mid n & \text{integer} \\
x & \text{variable} \\
\lambda x.E & \text{function} \\
e E & \text{function application}
\end{array}
\]

- judgment that says expression \( E \) has type \( \tau \) is written as
  \[ \Gamma \vdash E : \tau \]
  - \( \Gamma \) is a set of type assumptions for the free variables in \( E \).
Example: Type Inference (2/4)

Consider *simple types*

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]

---

**Figure:** Proof rules of simple types

\[ \frac{\Gamma \vdash n : \text{int}}{\Gamma \vdash x : \tau} \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \]

\[ \frac{\Gamma + x : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \lambda x. E : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1 E_2 : \tau_2} \]

---

**Theorem (Soundness of the proof rules)**

Let \( E \) be a program, an expression without free variables. If \( \emptyset \vdash E : \tau \), then the program runs without a type error and returns a value of type \( \tau \) if it terminates.
Example: Type Inference (3/4)

Program

\((\lambda x. x \ 1)(\lambda y. y)\)

is typed \(\text{int}\) because we can prove

\[\emptyset \vdash (\lambda x. x \ 1)(\lambda y. y) : \text{int}\]

as follows:

\[
\begin{align*}
\{ x : \text{int} \rightarrow \text{int} \} & \vdash x : \text{int} \rightarrow \text{int} \\
\{ x : \text{int} \rightarrow \text{int} \} & \vdash 1 : \text{int} \\
\{ y : \text{int} \} & \vdash y : \text{int} \\
\emptyset & \vdash \lambda x. x \ 1 : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \\
\emptyset & \vdash \lambda y. y : \text{int} \rightarrow \text{int} \\
\emptyset & \vdash (\lambda x. x \ 1)(\lambda y. y) : \text{int}
\end{align*}
\]
Example: Type Inference (4/4)

Algorithm

- given a program \( E \), \( V(\emptyset, E, \alpha) \) returns type equations.

\[
\begin{align*}
V(\Gamma, n, \tau) &= \{\tau \doteq int\} \\
V(\Gamma, x, \tau) &= \{\tau \doteq \Gamma(x)\} \\
V(\Gamma, \lambda x. E, \tau) &= \{\tau \doteq \alpha_1 \rightarrow \alpha_2\} \cup V(\Gamma + x : \alpha_1, E, \alpha_2) \quad \text{(new } \alpha_i) \\
V(\Gamma, E_1 E_2, \tau) &= V(\Gamma, E_1, \alpha \rightarrow \tau) \cup V(\Gamma, E_2, \alpha) \quad \text{(new } \alpha) \\
\end{align*}
\]

- solving the equations is done by the unification procedure

**Theorem (Correctness of the algorithm)**

Solving the equations \(\equiv\) proving in the simple type system

More precise analysis?

- need new sound proof rules (e.g., polymorphic type systems)
Limitations

- For target languages that lack a sound static type system, we have to invent it.
  - design a finite proof system
  - prove the soundness of the proof system
  - design its algorithm that automates proving
  - prove the correctness of the algorithm
- What if the unification procedure is not enough?
  - for some properties, the algorithm can generate constraints that are unsolvable by the unification procedure
- For some conventional imperative languages, sound and precise-enough static type systems are elusive.
Introduction

2 Static Analysis: a Gentle Introduction

3 A General Framework in Transitional Style

4 A Technique for Scalability: Sparse Analysis

5 Specialized Frameworks

Thank you!