

# Static Program Analysis

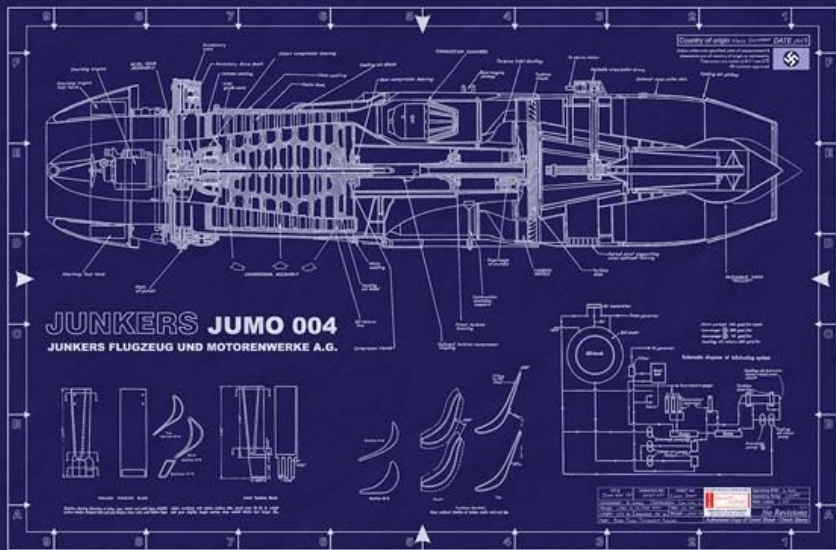
Kwangkeun Yi

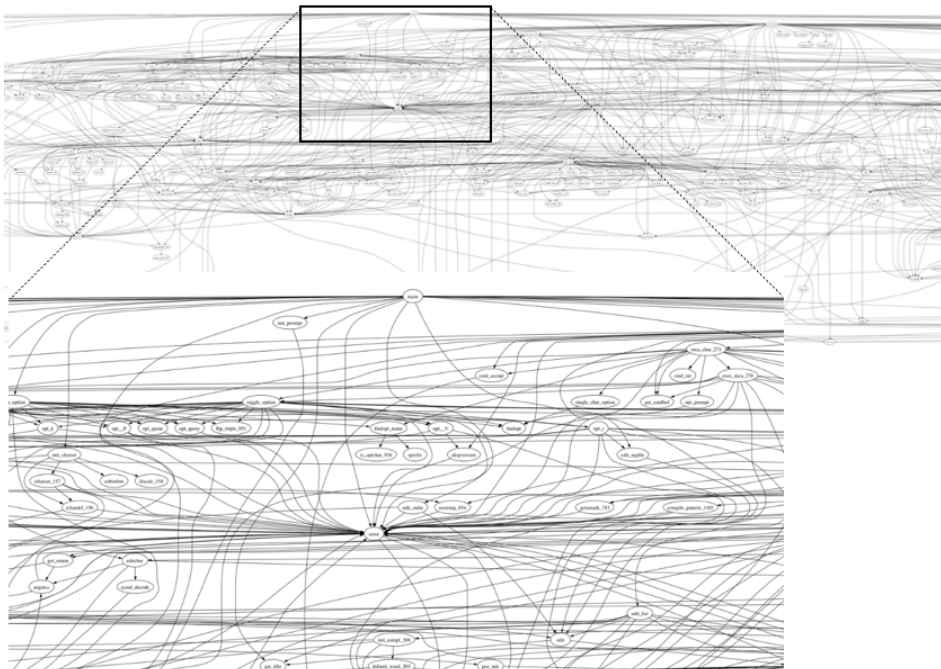
Seoul National University, Korea

- 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





# The Common Goal

	Computing area	Other engineering areas
Object	Software	Machine/building/circuit/chemical process design
Execution subject	Computer runs it	Nature runs it
Our question	Will it work as intended?	Will it work as intended?
Our knowledge	Program analysis	Newtonian mechanics, Maxwell equations, Navier-Stokes equations, the dynamic equations, and other principles

# 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  $\llbracket P \rrbracket \subseteq \mathcal{S}$  where

- **the semantics**  $\llbracket P \rrbracket$  = the set of all behaviors of  $P$
- **the specification**  $\mathcal{S}$  = the set of acceptable behaviors

# Semantics $\llbracket P \rrbracket$ and Semantic Properties $\mathcal{S}$

## Semantics $\llbracket P \rrbracket$ :

- compositional style (“denotational”)
  - ▶  $\llbracket AB \rrbracket = \dots \llbracket A \rrbracket \dots \llbracket B \rrbracket \dots$
- transitional style (“operational”)
  - ▶  $\llbracket AB \rrbracket = \{s_0 \hookrightarrow s_1 \hookrightarrow \dots, \dots\}$

## Semantic properties $\mathcal{S}$ :

- 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 *finite* time will never occur.

## Examples:

- no crashing error
  - ▶ no divide by zero, no bus error in C, no uncaught exceptions
- no invariant violation
  - ▶ some data structure should never get broken
- no value overrun
  - ▶ a variable's values 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

# Soundness and Completeness

“Analysis is sound.” “Analysis is complete.”

- **Soundness:**  $\text{analysis}(P) = \text{yes} \implies P$  satisfies the specification
- **Completeness:**  $\text{analysis}(P) = \text{yes} \iff P$  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

# Machine-Assisted Proving

## Approach

- ➊ Use a **specific language** to **formalize verification goals**
  - ➋ **Manually supply proof arguments**
  - ➌ 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

## Approach

- ① Focus on **finite state models** of programs
  - ② Perform **exhaustive exploration** of program states
- **Automatic**
  - **Sound** or **complete**, only with respect to the finite models
  - Software has  $\sim \infty$  states: the models need **approximation or non-termination (semi-algorithm)**

# Conservative Static Analysis

## Approach

- ➊ Perform **automatic verification**, yet which may fail
  - ➋ Compute a **conservative approximation of the program semantics**
- **Either sound or complete, not both**
  - **Sound & incomplete** static analysis is common:
    - ▶ optimizing compilers relies on it (supposed to)
    - ▶ Astrée, Sparrow, Facebook Infer, ML type systems, ...
  - **Automatic**
  - Incompleteness: **may reject safe programs (false alarms)**
  - Analysis algorithms **reason over program semantics**



# Bug Finding

## Approach

**Automatic**, **unsound** and **incomplete** algorithms

- commercial tools: 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

	automatic	sound	complete
testing	yes	no	yes
machine-assisted proving	no	yes	yes/no
finite-state model checking	yes	yes/no	yes/no
conservative static analysis	yes	yes	no
bug-finding	yes	no	no

# Focus of This Lecture: Conservative Static Analysis

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

- **checks**, for input program  $P$  in  $\mathbb{L}$ , if  $\llbracket P \rrbracket \subseteq \mathcal{S}$ ,
- **automatic** (software)
- **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 $\llbracket P \rrbracket$ Beforehand

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

For a Turing-complete language  $\mathbb{L}$ ,  
 $\nexists$  algorithm  $\text{Exact} : \text{Exact}(P) = \llbracket P \rrbracket$  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**  $\llbracket P \rrbracket^\sharp$  of  $\llbracket P \rrbracket$

- “finite”, hence
  - ▶ automatic (algorithm) &
  - ▶ static (without executing  $P$ )
- “sound”
  - ▶ over-approximation of  $\llbracket P \rrbracket$

Hence, ushers us to sound analysis:

$$(\text{analysis}(P) = \text{check } \llbracket P \rrbracket^\sharp \subseteq \mathcal{S}) \implies (P \text{ satisfies property } \mathcal{S})$$

# Need Formal Frameworks of Static Analysis (1/2)

Suppose that

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

You may, intuitively:

```

1:      x = readInt;
      while (x ≤ 99)
2:
          x++;
3:
      end
4:

```

Capture the dynamics by abstract equations; solve; reason.

$$\begin{aligned}
 x_1 &= [-\infty, +\infty] \text{ or } x_3 \\
 x_2 &= x_1 \text{ and } [-\infty, 99] \\
 x_3 &= x_2 + 1 \\
 x_4 &= x_1 \text{ and } [100, +\infty]
 \end{aligned}$$

# Need Formal Frameworks of Static Analysis (2/2)

*Abstract Interpretation* [CousotCousot]: a powerful design theory

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

```
x = readInt;
while (x ≤ 99)
  x++;
end
```

how?  
 $\implies$

```
x1 = [-∞, +∞] or x3
x2 = x1 and [-∞, 99]
x3 = x2 + 1
x4 = x1 and [100, +∞]
```

- Define an abstract semantics function  $\hat{F}$  s.t. ...
- How to solve the equations in a **finite** time?

```
x1 = [-∞, +∞] or x3
x2 = x1 and [-∞, 99]
x3 = x2 + 1
x4 = x1 and [100, ∞]
```

how?  
 $\implies$

```
x1 = [-∞, +∞]
x2 = [-∞, 99]
x3 = [-∞, 100]
x4 = [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

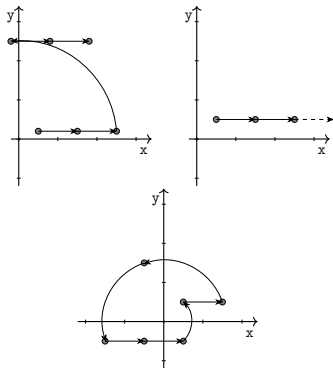
$p ::=$	$\text{init}(\mathfrak{R})$	initialization, with a state in $\mathfrak{R}$
	$\text{translation}(u, v)$	translation by vector $(u, v)$
	$\text{rotation}(u, v, \theta)$	rotation by center $(u, v)$ and angle $\theta$
	$p ; p$	sequence of operations
	$\{p\} \text{or} \{p\}$	non-deterministic choice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

## Example (Semantics)

```

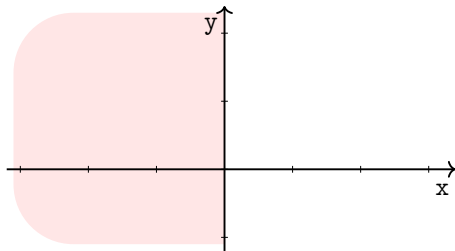
init([0,1] × [0,1]);
translation(1,0);
iter{
  {
    translation(1,0)
  }or{
    rotation(0,0,90°)
  }
}

```

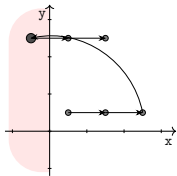


## Analysis Goal Is Safety Property: Reachability

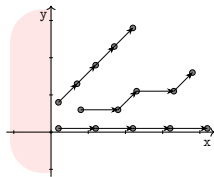
Analyze the set of reachable points, to check if the set intersects with a no-fly zone. Suppose that the no-fly zone is:



# Correct or Incorrect Executions



(a) An incorrect execution

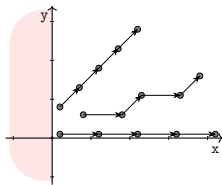


(b) Correct executions

# An Example Safe Program

## Example

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



# How to Finitely Over-Approximate the Set of Reachable Points?

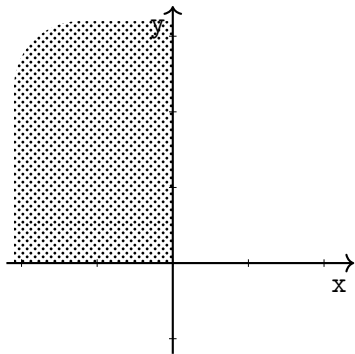
## Definition (Abstraction)

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*.

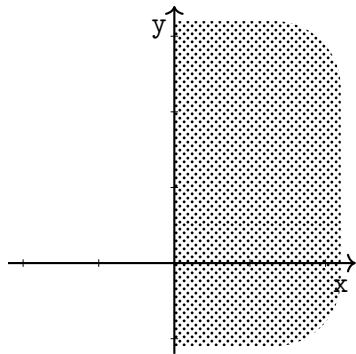
## Definition (Concretization)

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  $[x \leq 0, y \geq 0]$



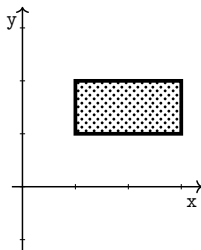
(d) Concretization of  $[x \geq 0]$

Figure: Signs abstraction

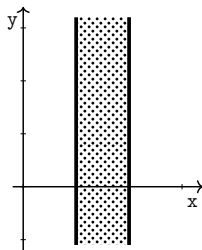
## Abstraction Example 2: Interval Abstraction

The abstract elements: conjunctions of non-relational inequality

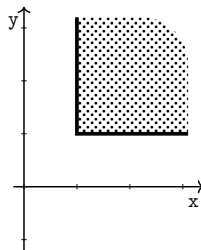
constraints:  $c_1 \leq x \leq c_2$ ,  $c'_1 \leq y \leq c'_2$



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



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



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

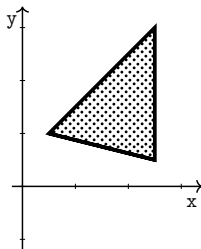
Figure: Intervals abstraction



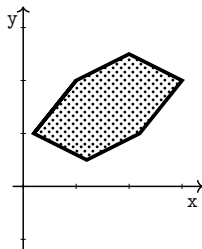
# Abstraction Example 3: Convex Polyhedra Abstraction

The abstract elements: conjunctions of linear inequality constraints:

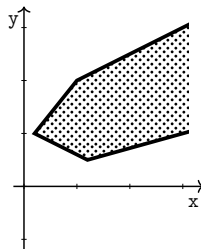
$$c_1x + c_2y \leq c_3$$



(a) Concretization of  $a_0$



(b) Concretization of  $a_1$



(c) Concretization of  $a_2$

Figure: Convex polyhedra abstraction

# An Example Program, Again

## Example

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

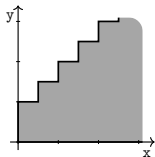
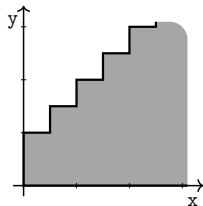
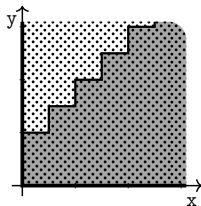


Figure: Reachable states

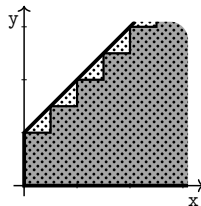
# Abstractions of the Semantics of the Example Program



(a) Reachable states



(b) Intervals abstraction



(c) Convex polyhedra abstraction

**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

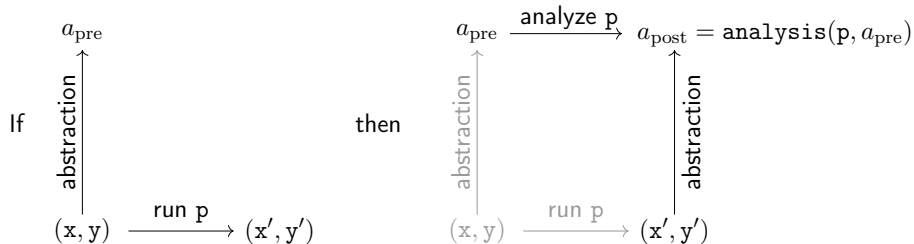


Figure: Sound analysis of a program  $p$

# Abstract Semantics Computation

Recall the example language

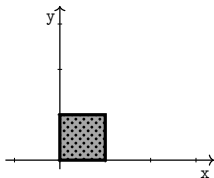
$p ::=$	$\text{init}(\mathfrak{R})$	initialization, with a state in $\mathfrak{R}$
	$\text{translation}(u, v)$	translation by vector $(u, v)$
	$\text{rotation}(u, v, \theta)$	rotation defined by center $(u, v)$ and angle $\theta$
	$p ; p$	sequence of operations
	$\{p\} \text{or} \{p\}$	non-deterministic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

## Approach

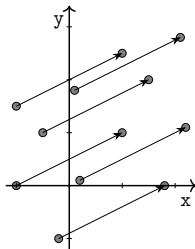
A sound analysis for a program is constructed by computing sound abstract semantics of the program's components.

# Abstract Semantics Computation: $\text{init}(\mathfrak{R})$

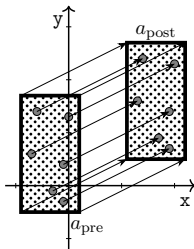
- Select, if any, the best abstraction of the region  $\mathfrak{R}$ .
- For the example program with the intervals or convex polyhedra abstract domains, analysis of  $\text{init}([0, 1] \times [0, 1])$  is



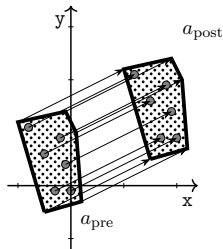
$\text{analysis}(\text{init}(\mathfrak{R}), a) = \text{best abstraction of the region } \mathfrak{R}$

Abstract Semantics Computation:  $\text{translation}(u, v)$ 

(a) Concrete semantics



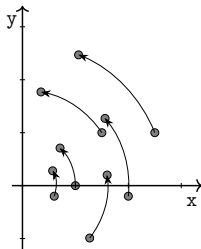
(b) Intervals



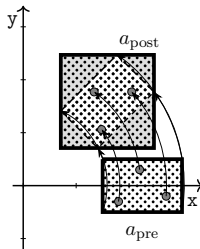
(c) Convex polyhedra

$\text{analysis}(\text{translation}(u, v), a) = \begin{cases} \text{return an abstract state that contains} \\ \text{the translation of } a \end{cases}$

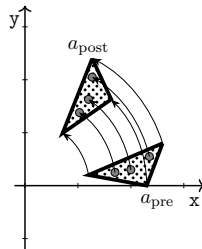


Abstract Semantics Computation:  $\text{rotation}(u, v, \theta)$ 

(d) Concrete semantics

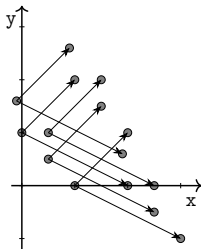


(e) Intervals

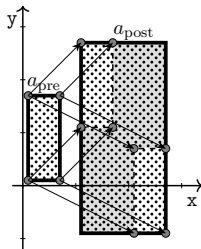


(f) Convex polyhedra

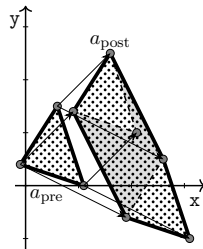
$\text{analysis}(\text{rotation}(u, v, \theta), a) = \begin{cases} \text{return an abstract state that contains} \\ \text{the rotation of } a \end{cases}$

Abstract Semantics Computation:  $\{p\}\text{or}\{p\}$ 

(g) Concrete semantics



(h) Intervals



(i) Convex polyhedra

$$\text{analysis}(\{p_0\}\text{or}\{p_1\}, a) = \text{union}(\text{analysis}(p_1, a), \text{analysis}(p_0, a))$$

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

$$\begin{aligned} & \{\} \\ & \text{or}\{p\} \\ & \text{or}\{p; p\} \\ & \text{or}\{p; p; p\} \\ & \text{or}\{p; p; p; p\} \\ & \vdots \end{aligned}$$

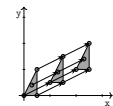
# Abstract Semantics Computation: $\text{iter}\{p\}$ (2/5)

## Example (Abstract iteration)

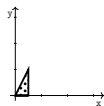
```

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

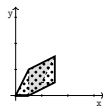
```



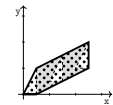
(j) Concrete semantics



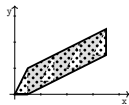
(k) Analysis of  $p_0$  (0 iteration)



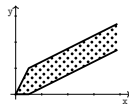
(l) Analysis of  $p_1$  (up to 1 iteration)



(m) Analysis of  $p_2$  (up to 2 iterations)



(n) Analysis of  $p_3$  (up to 3 iterations)



(o) Expected result

Figure: Abstract iteration

# Abstract Semantics Computation: $\text{iter}\{p\}$ (3/5)

Recall

$$\begin{aligned}\text{iter}\{p\} &= \{\} \text{ or } \{p\} \text{ or } \{p;p\} \text{ or } \cdots \\ &= \lim_i p_i\end{aligned}$$

where

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

Hence,

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

$$\text{operator widen} \quad \begin{cases} \text{over approximates unions} \\ \text{enforces finite convergence} \end{cases}$$

# Abstract Semantics Computation: $\text{iter}\{p\}$ (4/5)

## Example (Abstract iteration with widening)

```
init({(x,y) |  $0 \leq y \leq 2x$  and  $x \leq 0.5$ });
iter{
  translation(1,0.5)
}
```

- 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.

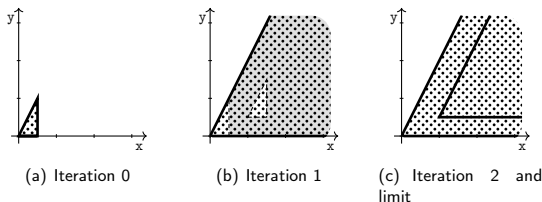
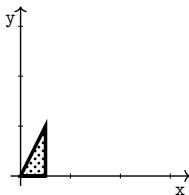


Figure: Abstract iteration with widening

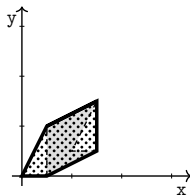
# Abstract Semantics Computation: $\text{iter}\{p\}$ (5/5)

## Example (Loop unrolling)

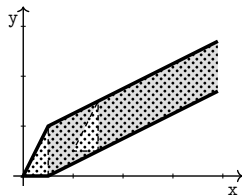
```
init({(x,y) |  $0 \leq y \leq 2x$  and  $x \leq 0.5$ });  
{ } or { translation(1, 0.5) };  
iter{ translation(1, 0.5) }
```



(a) Iteration 0



(b) Iteration 1, union



(c) Iteration 2, widen, limit

Figure: Abstract iteration with widening and unrolling



# Abstract Semantics Function `analysis` At a Glance

The `analysis`( $p, a$ ) is finitely computable and sound.

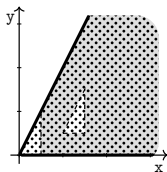
$$\begin{aligned}
 \text{analysis}(\text{init}(\mathfrak{R}), a) &= \text{best abstraction of the region } \mathfrak{R} \\
 \text{analysis}(\text{translation}(u, v), a) &= \begin{cases} \text{return an abstract state that contains} \\ \text{the translation of } a \end{cases} \\
 \text{analysis}(\text{rotation}(u,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

## Sound analysis

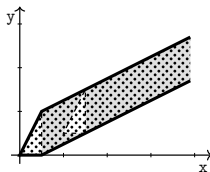
If an execution of  $p$  from a state  $(x, y)$  generates the state  $(x', y')$ ,  
 then for all abstract element  $a$  such that  $(x, y) \in \gamma(a)$ ,  
 $(x', y') \in \gamma(\text{analysis}(p, a))$

# 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 `analysis(p, -)` and collect all points  $\mathfrak{R}$  from every call to `analysis`.
- Since `analysis` is sound, the result is an over approx. of the reachable points.
- **If  $\mathfrak{R} \cap \mathcal{D} = \emptyset$ , then  $p$  is verified. Otherwise, we don't know.**



(a) An example  $\mathfrak{R}$



(b) A more precise  $\mathfrak{R}$

# Semantics Style: Compositional Versus Transitional

- Compositional semantics function analysis:
  - ▶ Semantics of  $p$  is defined by the semantics of the sub-parts of  $p$ .

$$\llbracket AB \rrbracket = \dots \llbracket A \rrbracket \dots \llbracket B \rrbracket \dots$$

- ▶ 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

$$\llbracket AB \rrbracket = \{s_0 \hookrightarrow s_1 \hookrightarrow \dots, \dots\}$$

# Example Language, Again

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

$$(l, p) \hookrightarrow (l', p')$$

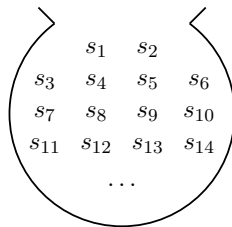
whenever the program statement at  $l$  moves the point  $p$  to  $p'$ .

$s_1 \hookrightarrow s_2 \hookrightarrow s_5 \hookrightarrow s_3 \hookrightarrow s_8 \hookrightarrow \dots$

$s_6 \hookrightarrow s_7 \hookrightarrow s_8 \hookrightarrow s_3 \hookrightarrow s_4$

$s_9 \hookrightarrow s_{10} \hookrightarrow s_8 \hookrightarrow s_{11} \hookrightarrow s_8 \hookrightarrow s_{11} \hookrightarrow s_{13}$

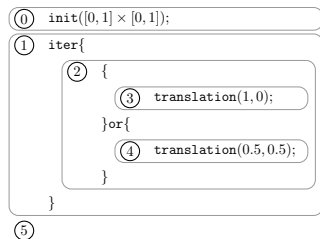
$s_{12} \hookrightarrow s_7 \hookrightarrow s_2 \hookrightarrow s_3 \hookrightarrow s_4 \hookrightarrow s_{14}$



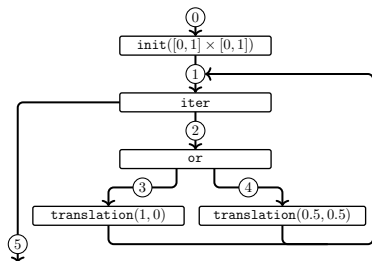
States  $s_1, s_6, s_9$ , and  $s_{12}$  are initial states.

Figure: Transition sequences and the set of occurring states

# Statement Labels



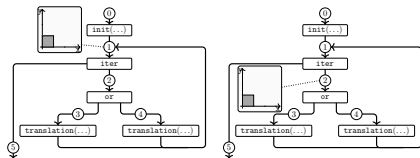
(a) Text view, with labels

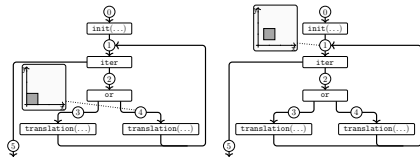


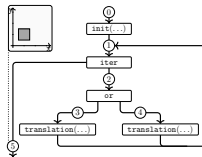
(b) Graph view, with labels

Figure: Example program with statement labels

# States in a Transition Sequence


 (a) State  $(1, p_1)$ 

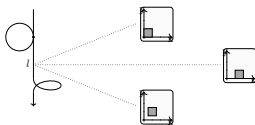
 (b) State  $(2, p_1)$ 

 (c) State  $(4, p_1)$ 

 (d) State  $(1, p_3)$ 

 (e) State  $(5, p_3)$

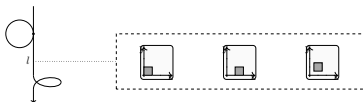
# Reachability Problem and Abstraction of States

- 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 pre conditions.

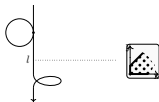
Collection of all states



Statement-wise collection:



Statement-wise abstraction:





# Abstract State Transition

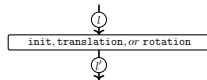
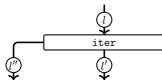
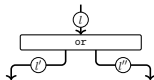
$Step^\#$  : a set of pairs of labels and abstract pre conditions  
 $\mapsto$   
 a set of pairs of labels and abstract post conditions

is

$$Step^\#(X) = \{x' \mid x \in X, x \hookrightarrow^\# x'\}$$

where

$$\begin{aligned} (or_l, a_{pre}) &\hookrightarrow^\# (next(l), a_{pre}) \\ (iter_l, a_{pre}) &\hookrightarrow^\# (next(l), a_{pre}) \\ (p_l, a_{pre}) &\hookrightarrow^\# (next(l), \text{analysis}(p_l, a_{pre})) \end{aligned}$$



# Analysis by Global Iterations

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

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

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

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

Thus the analysis algorithm should iterate the operation

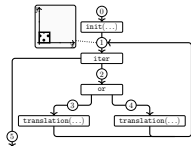
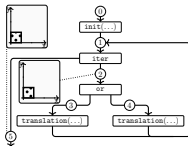
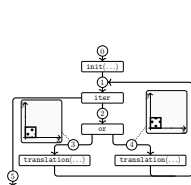
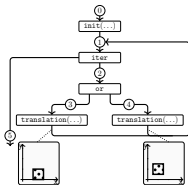
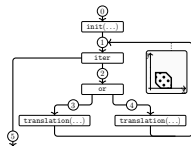
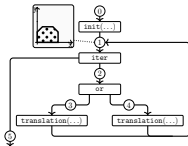
$$C \leftarrow C \cup Step^{\#}(C)$$

from  $I$  until stable:

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

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

$$s_0 \hookrightarrow s_1 \hookrightarrow s_2 \hookrightarrow \dots$$

where  $\hookrightarrow$  is a transition relation between states  $\mathbb{S}$

$$\hookrightarrow \subseteq \mathbb{S} \times \mathbb{S}$$

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

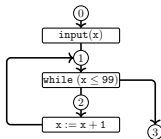
# Concrete Transition Sequence

## Example

Consider the following program

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

Let labels be “program points.”



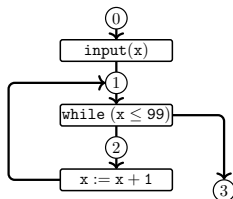
Let the initial state be  $\emptyset$ . Some transition sequences are:

For input 100:  $(0, \emptyset) \hookrightarrow (1, x \mapsto 100) \hookrightarrow (3, x \mapsto 100)$ .

For input 99:  $(0, \emptyset) \hookrightarrow (1, x \mapsto 99) \hookrightarrow (2, x \mapsto 99) \hookrightarrow (1, x \mapsto 100) \hookrightarrow (3, x \mapsto 100)$ .

For input 0:  $(0, \emptyset) \hookrightarrow (1, x \mapsto 0) \hookrightarrow (2, x \mapsto 0) \hookrightarrow (1, x \mapsto 1) \hookrightarrow \dots \hookrightarrow (3, x \mapsto 100)$ .

# Reachable States



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:

$$\begin{aligned}
 & \{(0, \emptyset), (1, x \mapsto 100), (3, x \mapsto 100)\} \\
 \cup & \{(0, \emptyset), (1, x \mapsto 99), (2, x \mapsto 99), (1, x \mapsto 100), (3, x \mapsto 100)\} \\
 \cup & \{(0, \emptyset), (1, x \mapsto 0), (2, x \mapsto 0), (1, x \mapsto 1), \dots, (2, x \mapsto 99), (1, x \mapsto 100), (3, x \mapsto 100)\} \\
 = & \{(0, \emptyset), (1, x \mapsto 0), \dots, (1, x \mapsto 100), (2, x \mapsto 0), \dots, (2, x \mapsto 99), (3, x \mapsto 100)\}
 \end{aligned}$$



# 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  $\hookrightarrow$ :

$$\begin{aligned} Step &: \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S}) \\ Step(X) &= \{s' \mid s \hookrightarrow s', s \in X\} \end{aligned}$$

The set of reachable states is

$$I \cup Step^1(I) \cup Step^2(I) \cup \dots$$

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

$$\begin{aligned} C_0 &= I \\ C_{i+1} &= I \cup Step(C_i) \end{aligned}$$

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  $F$

$$\begin{aligned} F &: \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S}) \\ F(X) &= I \cup \text{Step}(X) \end{aligned}$$

written as

$$\mathbf{lfp}F.$$

## Theorem (Least fixpoint)

*The least fixpoint  $\mathbf{lfp}F$  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  $\mathbb{S}$  be the set of states and  $\hookrightarrow$  be the one-step transition relation  $\subseteq \mathbb{S} \times \mathbb{S}$ . Let  $I$  be the set of its initial states and  $Step$  be the powerset-lifted version of  $\hookrightarrow$ :

$$\begin{aligned} Step : \wp(\mathbb{S}) &\rightarrow \wp(\mathbb{S}) \\ Step(X) &= \{s' \mid s \hookrightarrow s', s \in X\}. \end{aligned}$$

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

$$F(X) = I \cup Step(X).$$

# 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”:

$$F : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S})$$

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

$$F^\# : \mathbb{S}^\# \rightarrow \mathbb{S}^\#$$

$$F^\#(X^\#) = I^\# \cup^\# \text{Step}^\#(X^\#)$$

The forthcoming framework will guide us

- conditions for  $\mathbb{S}^\#$  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 $\wp(\mathbb{S})$ (1/2)

$$\wp(\mathbb{S}) \quad \text{where} \quad \mathbb{S} = \mathbb{L} \times \mathbb{M}$$

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

$$\begin{array}{ccccc} \text{set of states} & & \text{to} & & \text{label-wise collect} & & \text{to} & & \text{label-wise abstraction} \\ \wp(\mathbb{L} \times \mathbb{M}) & \xrightarrow{\text{abstraction}} & & \mathbb{L} \rightarrow \wp(\mathbb{M}) & \xrightarrow{\text{abstraction}} & & \mathbb{L} \rightarrow \mathbb{M}^\sharp. \end{array}$$

# Abstraction of the Semantic Domain $\wp(\mathbb{S})$ (2/2)

$$\wp(\mathbb{L} \times \mathbb{M}) \ni \begin{array}{l} \text{collection of} \\ \text{all states} \end{array} \left\{ \begin{array}{ll} (0, m_0), (0, m'_0), \dots, & \text{at } 0 \\ (1, m_1), (1, m'_1), \dots, & \text{at } 1 \\ \vdots & \\ (n, m_n), (n, m'_n), \dots, & \text{at } n \end{array} \right.$$

$$\mathbb{L} \rightarrow \wp(\mathbb{M}) \ni \begin{array}{l} \text{label-wise} \\ \text{collection} \end{array} \left\{ \begin{array}{l} (0, \{m_0, m'_0, \dots\}) \\ (1, \{m_1, m'_1, \dots\}) \\ \vdots \\ (n, \{m_n, m'_n, \dots\}) \end{array} \right.$$

$$\mathbb{L} \rightarrow \mathbb{M}^\# \ni \begin{array}{l} \text{label-wise} \\ \text{abstraction} \end{array} \left\{ \begin{array}{l} (0, M_0^\#) \\ (1, M_1^\#) \\ \vdots \\ (n, M_n^\#) \end{array} \right.$$

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

# Preliminary for Abstract Domains (1/3)

- Define an abstract domain as a *CPO*
  - ▶ a partial order set
  - ▶ has a least element  $\perp$
  - ▶ 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:

$$\forall c \in \mathbb{C}, \forall a \in \mathbb{A}, \quad \alpha(c) \sqsubseteq a \quad \Longleftrightarrow \quad c \subseteq \gamma(a)$$

We write such a pair as follows:

$$(\mathbb{C}, \subseteq) \xrightleftharpoons[\alpha]{\gamma} (\mathbb{A}, \sqsubseteq)$$

# Preliminary for Abstract Doamins (3/3)

Galois-connection properties we rely on:

For

$$(\mathbb{C}, \subseteq) \xrightleftharpoons[\alpha]{\gamma} (\mathbb{A}, \sqsubseteq)$$

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

(Proofs are in the book.)

# Abstract Domains (1/2)

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

$$(\wp(\mathbb{L} \times \mathbb{M}), \subseteq) \xleftrightarrow[\alpha]{\gamma} (\mathbb{L} \rightarrow \mathbb{M}^\#, \sqsubseteq).$$

- 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  $\sqsubseteq$  is the label-wise order:

$$a^\# \sqsubseteq b^\# \quad \text{iff} \quad \forall l \in \mathbb{L} : a^\#(l) \sqsubseteq_M b^\#(l)$$

where  $\sqsubseteq_M$  is the partial order of  $\mathbb{M}^\#$ .

# Abstract Domains (2/2)

The above Galois connection (abstraction)

$$(\wp(\mathbb{L} \times \mathbb{M}), \subseteq) \xleftrightarrow[\alpha]{\gamma} (\mathbb{L} \rightarrow \mathbb{M}^\#, \sqsubseteq).$$

composes two Galois connections:

$$\begin{aligned} & (\wp(\mathbb{L} \times \mathbb{M}), \subseteq) \\ & \xleftrightarrow[\alpha_0]{\gamma_0} (\mathbb{L} \rightarrow \wp(\mathbb{M}), \sqsubseteq) \quad (\sqsubseteq \text{ is the label-wise } \subseteq) \\ & \quad \xleftrightarrow[\alpha_1]{\gamma_1} (\mathbb{L} \rightarrow \mathbb{M}^\#, \sqsubseteq) \quad (\sqsubseteq \text{ is the label-wise } \sqsubseteq_M) \end{aligned}$$

$$\alpha_0 \left\{ \begin{array}{l} (0, m_0), (0, m'_0), \dots, \\ \vdots \\ (n, m_n), (n, m'_n), \dots \end{array} \right\} = \left\{ \begin{array}{l} (0, \{m_0, m'_0, \dots\}), \\ \vdots \\ (n, \{m_n, m'_n, \dots\}) \end{array} \right\}$$

$$\alpha_1 \left\{ \begin{array}{l} (0, \{m_0, m'_0, \dots\}), \\ \vdots \\ (n, \{m_n, m'_n, \dots\}) \end{array} \right\} = \left\{ \begin{array}{l} (0, M_0^\#), \\ \vdots \\ (n, M_n^\#) \end{array} \right\}$$

Thus, boils down to

$$(\wp(\mathbb{M}), \subseteq) \xleftrightarrow[\alpha_M]{\gamma_M} (\mathbb{M}^\#, \sqsubseteq_M).$$

# Abstract Semantic Functions

Let

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

A concrete semantic function  $F$

An abstract semantic function  $F^\#$

$$\mathbb{S} = \mathbb{L} \times \mathbb{M}$$

$$F : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S})$$

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

$$\text{Step} = \wp(\hookrightarrow)$$

$$\hookrightarrow \subseteq (\mathbb{L} \times \mathbb{M}) \times (\mathbb{L} \times \mathbb{M})$$

$$\mathbb{S}^\# = \mathbb{L} \rightarrow \mathbb{M}^\#$$

$$F^\# : \mathbb{S}^\# \rightarrow \mathbb{S}^\#$$

$$F^\#(X^\#) = \alpha(I) \cup^\# \text{Step}^\#(X^\#)$$

$$\text{Step}^\# = \wp(\text{id}, \cup_M^\#) \circ \pi \circ \wp(\hookrightarrow^\#)$$

$$\hookrightarrow^\# \subseteq (\mathbb{L} \times \mathbb{M}^\#) \times (\mathbb{L} \times \mathbb{M}^\#)$$

with relations  $\hookrightarrow$  and  $\hookrightarrow^\#$  being functions

As of  $Step^\sharp = \wp(\text{id}, \cup_M^\sharp) \circ \pi \circ \wp(\hookrightarrow^\sharp)$

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

- Abstract transition  $\wp(\hookrightarrow^\sharp)$ :
  - ▶ a set  $\subseteq \mathbb{L} \times \mathbb{M}^\sharp \mapsto$  a set  $\subseteq \mathbb{L} \times \mathbb{M}^\sharp$
- Partitioning  $\pi$ :
  - ▶ a set  $\subseteq \mathbb{L} \times \mathbb{M}^\sharp \mapsto$  a set  $\subseteq \mathbb{L} \times \wp(\mathbb{M}^\sharp)$
- Joining  $\wp(\text{id}, \cup_M^\sharp)$ :
  - ▶ a set  $\subseteq \mathbb{L} \times \wp(\mathbb{M}^\sharp) \mapsto$  an abstract state  $\in \mathbb{L} \rightarrow \mathbb{M}^\sharp$

## Example

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

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

## Example

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

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

Suppose the result is

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



## Example

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

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

Suppose the result is

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

By the subsequent partitioning operator  $\pi$ , the result becomes

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

## Example

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

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

Suppose the result is

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

By the subsequent partitioning operator  $\pi$ , the result becomes

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

The final organization operation  $\wp(\text{id}, \cup_M^\sharp)$  returns the post abstract state  $\in \mathbb{L} \rightarrow \mathbb{M}^\sharp$ :

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

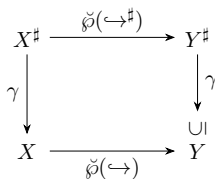
# Conditions for Sound $\hookrightarrow^\#$ and $\cup_-^\#$

- sound condition for  $\hookrightarrow^\#$ :

$$\check{\rho}(\hookrightarrow) \circ \gamma \subseteq \gamma \circ \check{\rho}(\hookrightarrow^\#)$$

- sound condition for  $\cup_-^\#$ :

$$\cup \circ (\gamma, \gamma) \subseteq \gamma \circ \cup_-^\#$$



Pattern for the sound condition for each semantic operator

$$f^\# : A^\# \rightarrow B^\#$$

$$f \circ \gamma_A \sqsubseteq_B \gamma_B \circ f^\#.$$

# Then, Follows Sound Static Analysis

- In case  $\mathbb{S}^\sharp$  is of finite-height and  $F^\sharp$  is monotone or extensive, then

$$\bigsqcup_{i \geq 0} F^{\sharp^i}(\perp)$$

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

- Otherwise, find a widening operator  $\nabla$ , then the following chain  
 $X_0 \sqsubseteq X_1 \sqsubseteq \dots$

$$X_0 = \perp \quad X_{i+1} = X_i \nabla F^\sharp(X_i)$$

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

# Underlying Theorems (1/2)

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

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

$$\bigsqcup_{i \geq 0} F^{\sharp i}(\perp)$$

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

$$\text{lfp}F \subseteq \gamma\left(\bigsqcup_{i \geq 0} F^{\sharp i}(\perp)\right) \quad \text{or equivalently} \quad \alpha(\text{lfp}F) \sqsubseteq \bigsqcup_{i \geq 0} F^{\sharp i}(\perp).$$

(Proof is in the book.)

## Underlying Theorems (2/2)

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

*Given a program, let  $F$  and  $F^\sharp$  be defined as in the framework. Let  $\nabla$  be a widening operator. Then the following chain  $Y_0 \sqsubseteq Y_1 \sqsubseteq \dots$*

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

*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 book.)

## Definition (Widening operator)

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

- 1 For all abstract elements  $a_0, a_1$ , we have

$$\gamma(a_0) \cup \gamma(a_1) \subseteq \gamma(a_0 \nabla a_1)$$

- 2 For all sequence  $(a_n)_{n \in \mathbb{N}}$  of abstract elements, the sequence  $(a'_n)_{n \in \mathbb{N}}$  defined below is finitely stationary:

$$\begin{cases} a'_0 &= a_0 \\ a'_{n+1} &= a'_n \nabla a_n \end{cases}$$

# Analysis Algorithm Based on Global Iterations: Basic Version (1/2)

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

$$\perp \sqsubseteq (F^\sharp)^1(\perp) \sqsubseteq (F^\sharp)^2(\perp) \sqsubseteq \dots$$

is finite and its biggest element is equal to

$$\bigsqcup_{i \geq 0} F^{\sharp^i}(\perp).$$

```

C ← ⊥
repeat
  R ← C
  C ← F♯(C)
until C ⊆ R
return R

```



# Analysis Algorithm Based on Global Iterations: Basic Version (2/2)

- Case:  $S^\sharp$  is of infinite-height or  $F^\sharp$  is neither monotonic nor extensive
- Use a widening operator  $\nabla$

```
C ← ⊥  
repeat  
  R ← C  
  C ← C ∇ F♯(C)  
until C ⊆ R  
return R
```

# Inefficiency of the Basic Algorithms

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

```

C ← ⊥
repeat
  R ← C
  C ← C ∇  $\underbrace{(\wp(\text{id}, \cup_M^\sharp) \circ \pi \circ \wp(\hookrightarrow^\sharp))}_{F^\sharp}(C)$ 
until C ⊆ R
return R

```

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

$$\wp(\hookrightarrow^\sharp)(C)$$

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

WorkList  $\leftarrow \mathbb{L}$ 
C  $\leftarrow \perp$ 
repeat
    R  $\leftarrow$  C
    C  $\leftarrow C \nabla F^\#(C|_{\text{WorkList}})$ 
    WorkList  $\leftarrow \{l \mid C(l) \not\sqsubseteq R(l), l \in \mathbb{L}\}$ 
until WorkList =  $\emptyset$ 
return R

```

# Improvement of the Worklist Algorithm

- Inefficient:  $\text{WorkList} \leftarrow \{l \mid C(l) \not\subseteq R(l), l \in \mathbb{L}\}$  re-scans all the labels.
  - ▶ Better: At application  $\hookrightarrow^\#$  to  $(l, C(l))$ , if its result  $(l', M^\#)$  is changed ( $M^\# \not\subseteq C(l')$ ), add  $l'$  to the worklist.
- Inefficient:  $C \nabla F^\#(C|_{\text{WorkList}})$  widens at all the labels.
  - ▶ Better: Apply  $\nabla$  only at the target of a loop. Use  $\cup^\#$  at other labels.

# Summary: Recipe for Defining Sound Static Analysis(1/4)

- 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 **lfp** $F$  where

concrete domain	$\wp(\mathbb{S}) = \wp(\mathbb{L} \times \mathbb{M})$
concrete semantic function	$F : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S})$
	$F(X) = I \cup \text{Step}(X)$
	$\text{Step} = \wp(\hookrightarrow)$
	$\hookrightarrow \subseteq (\mathbb{L} \times \mathbb{M}) \times (\mathbb{L} \times \mathbb{M})$

The  $\hookrightarrow$  is the one-step transition relation over  $\mathbb{L} \times \mathbb{M}$ .

## Summary: Recipe for Defining Sound Static Analysis(2/4)

- 3 Define its abstract domain and abstract semantic function as

$$\begin{array}{ll}
 \text{abstract domain} & \mathbb{S}^\# = \mathbb{L} \rightarrow \mathbb{M}^\# \\
 \text{abstract semantic function} & F^\# : \mathbb{S}^\# \rightarrow \mathbb{S}^\# \\
 & F^\#(X^\#) = \alpha(I) \cup^\# \text{Step}^\#(X^\#) \\
 & \text{Step}^\# = \wp(\text{id}, \cup_M^\#) \circ \pi \circ \wp(\hookrightarrow^\#) \\
 & \hookrightarrow^\# \subseteq (\mathbb{L} \times \mathbb{M}^\#) \times (\mathbb{L} \times \mathbb{M}^\#)
 \end{array}$$

The  $\hookrightarrow^\#$  is the one-step abstract transition relation over  $\mathbb{L} \times \mathbb{M}^\#$ .  
 Function  $\pi$  partitions a set  $\subseteq \mathbb{L} \times \mathbb{M}^\#$  by the labels in  $\mathbb{L}$  returning an element in  $\mathbb{L} \rightarrow \wp(\mathbb{M}^\#)$  represented as a set  $\subseteq \mathbb{L} \times \wp(\mathbb{M}^\#)$ .

# Summary: Recipe for Defining Sound Static Analysis(3/4)

- 4 Check the abstract domains  $\mathbb{S}^\sharp$  and  $\mathbb{M}^\sharp$  are CPOs, and forms a Galois-connection respectively with  $\wp(\mathbb{S})$  and  $\wp(\mathbb{M})$ :

$$(\wp(\mathbb{S}), \subseteq) \xleftrightarrow[\alpha]{\gamma} (\mathbb{S}^\sharp, \sqsubseteq) \quad \text{and} \quad (\wp(\mathbb{M}), \subseteq) \xleftrightarrow[\alpha_M]{\gamma_M} (\mathbb{M}^\sharp, \sqsubseteq_M)$$

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

$$a^\sharp \sqsubseteq b^\sharp \quad \text{iff} \quad \forall l \in \mathbb{L} : a^\sharp(l) \sqsubseteq_M b^\sharp(l).$$

- 5 Check the abstract one-step transition  $\hookrightarrow^\sharp$  and abstract union  $\cup_-^\sharp$  satisfy:

$$\begin{aligned} \wp(\hookrightarrow) \circ \gamma &\subseteq \gamma \circ \wp(\hookrightarrow^\sharp) \\ \cup \circ (\gamma, \gamma) &\subseteq \gamma \circ \cup_-^\sharp \end{aligned}$$

# Summary: Recipe for Defining Sound Static Analysis(4/4)

⑥ Then, sound static analysis is defined as follows:

- ▶ In case  $\mathbb{S}^\sharp$  is of finite-height (every its chain is finite) and  $F^\sharp$  is monotone or extensive, then

$$\bigsqcup_{i \geq 0} F^{\sharp^i}(\perp)$$

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

- ▶ Otherwise, find a widening operator  $\nabla$ , then the following chain  $X_0 \sqsubseteq X_1 \sqsubseteq \dots$

$$X_0 = \perp \quad X_{i+1} = X_i \nabla F^\sharp(X_i)$$

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



# Use Example: Target Language

$x \in \mathbb{X}$	program variables
$C ::=$	statements
skip	nop statement
$C ; C$	sequence of statements
$x := E$	assignment
input( $x$ )	read an integer input
if( $B$ ){ $C$ }else{ $C$ }	condition statement
while( $B$ ){ $C$ }	loop statement
goto $E$	goto with dynamically computed label
$E ::=$	expression
$n$	integer
$x$	variable
$E + E$	addition
$B ::=$	boolean expression
true   false	
$E < E$	comparison
$E = E$	equality
$P ::= C$	program

Figure: Syntax of a simple imperative language

# Use Example: Concrete State Transition Semantics

$$\text{lfp} F$$

of the continuous function

$$\begin{aligned} F &: \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S}) \\ F(X) &= I \cup \text{Step}(X) \\ \text{Step}(X) &= \wp(\hookrightarrow) \end{aligned}$$

where

$$\mathbb{S} = \mathbb{L} \times \mathbb{M}$$

and

$$\begin{aligned} \text{memories } \mathbb{M} &= \mathbb{X} \rightarrow \mathbb{V} \\ \text{values } \mathbb{V} &= \mathbb{Z} \cup \mathbb{L} \end{aligned}$$

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

$$\begin{aligned} \text{skip} &: (l, m) \hookrightarrow (\text{next}(l), m) \\ \text{input}(x) &: (l, m) \hookrightarrow (\text{next}(l), \text{update}_x(m, z)) \quad \text{for an input integer } z \\ x := E &: (l, m) \hookrightarrow (\text{next}(l), \text{update}_x(m, \text{eval}_E(m))) \\ \mathcal{C}_1; \mathcal{C}_2 &: (l, m) \hookrightarrow (\text{next}(l), m) \\ \text{if}(B)\{\mathcal{C}_1\}\text{else}\{\mathcal{C}_2\} &: (l, m) \hookrightarrow (\text{nextTrue}(l), \text{filter}_B(m)) \\ &: (l, m) \hookrightarrow (\text{nextFalse}(l), \text{filter}_{\neg B}(m)) \\ \text{while}(B)\{\mathcal{C}\} &: (l, m) \hookrightarrow (\text{nextTrue}(l), \text{filter}_B(m)) \\ &: (l, m) \hookrightarrow (\text{nextFalse}(l), \text{filter}_{\neg B}(m)) \\ \text{goto } E &: (l, m) \hookrightarrow (\text{eval}_E(m), m) \end{aligned}$$

## Use Example: Abstract State

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

$$(\wp(\mathbb{M}), \subseteq) \xleftrightarrow[\alpha_M]{\gamma_M} (\mathbb{M}^\sharp, \sqsubseteq_M)$$

defined as

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

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

$$(\wp(\mathbb{V}), \subseteq) \xleftrightarrow[\alpha_V]{\gamma_V} (\mathbb{V}^\sharp, \sqsubseteq_V).$$

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

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

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

# Use Example: Abstract State Transition Semantics

Case the  $l$ -labeled statement of

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

# Use Example: Abstract State Transition Semantics

Case the  $l$ -labeled statement of

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

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

$$\begin{aligned}
 F^\sharp & : \mathbb{S}^\sharp \rightarrow \mathbb{S}^\sharp \\
 F^\sharp(S^\sharp) & = \alpha(I) \cup^\sharp \text{Step}^\sharp(S^\sharp) \\
 \text{Step}^\sharp & = \wp(\text{id}, \cup_M^\sharp) \circ \pi \circ \wp(\hookrightarrow^\sharp).
 \end{aligned}$$

# Use Example: Abstract State Transition Semantics

Case the  $l$ -labeled statement of

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

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

$$\begin{aligned}
 F^\sharp : \mathbb{S}^\sharp & \rightarrow \mathbb{S}^\sharp \\
 F^\sharp(S^\sharp) & = \alpha(I) \cup^\sharp \text{Step}^\sharp(S^\sharp) \\
 \text{Step}^\sharp & = \wp(\text{id}, \cup_M^\sharp) \circ \pi \circ \wp(\hookrightarrow^\sharp).
 \end{aligned}$$

If the  $\text{Step}^\sharp$  and  $\cup_-^\sharp$  are sound abstractions of, respectively,  $\text{Step}$  and  $\cup_-$ :

$$\begin{aligned}
 \wp(\hookrightarrow) \circ \gamma & \subseteq \gamma \circ \wp(\hookrightarrow^\sharp) \\
 \cup \circ (\gamma, \gamma) & \subseteq \gamma \circ \cup_-^\sharp
 \end{aligned}$$

# Use Example: Abstract State Transition Semantics

Case the  $l$ -labeled statement of

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

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

$$\begin{aligned}
 F^\sharp : \mathbb{S}^\sharp & \rightarrow \mathbb{S}^\sharp \\
 F^\sharp(S^\sharp) & = \alpha(I) \cup^\sharp \text{Step}^\sharp(S^\sharp) \\
 \text{Step}^\sharp & = \wp(\text{id}, \cup_M^\sharp) \circ \pi \circ \wp(\hookrightarrow^\sharp).
 \end{aligned}$$

If the  $\text{Step}^\sharp$  and  $\cup_-^\sharp$  are sound abstractions of, respectively,  $\text{Step}$  and  $\cup_-$ :

$$\begin{aligned}
 \wp(\hookrightarrow) \circ \gamma & \subseteq \gamma \circ \wp(\hookrightarrow^\sharp) \\
 \cup \circ (\gamma, \gamma) & \subseteq \gamma \circ \cup_-^\sharp
 \end{aligned}$$

then we can use  $F^\sharp$  to soundly approximate the concrete semantics  
 $\text{lfp}^F$

Use Example: Defining Sound  $\hookrightarrow^\sharp$ Theorem (Soundness of  $\hookrightarrow^\sharp$ )

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

$$\begin{aligned} \wp(\text{eval}_E) \circ \gamma_M &\subseteq \gamma_V \circ \text{eval}_E^\sharp \\ \wp(\text{update}_x) \circ \times \circ (\gamma_M, \gamma_V) &\subseteq \gamma_M \circ \text{update}_x^\sharp \\ \wp(\text{filter}_B) \circ \gamma_M &\subseteq \gamma_M \circ \text{filter}_B^\sharp \\ \wp(\text{filter}_{\neg B}) \circ \gamma_M &\subseteq \gamma_M \circ \text{filter}_{\neg B}^\sharp \end{aligned}$$

*then  $\wp(\hookrightarrow) \circ \gamma \sqsubseteq \gamma \circ \wp(\hookrightarrow^\sharp)$ . (The  $\times$  is the Cartesian product operator of two sets.)*



## Use Example: Defining Sound $\sqcup^\#$

As of sound  $\sqcup^\#$ , one candidate is the least upper bound operator  $\sqcup$  if  $\mathbb{S}^\#$  and  $\mathbb{M}^\#$  are closed by  $\sqcup$  (e.g. lattices), since

$$\begin{aligned}
 (\gamma \circ \sqcup)(a^\#, b^\#) &= \gamma(a^\# \sqcup b^\#) \quad \sqsupseteq \quad \gamma(a^\#) \cup \gamma(b^\#) && \text{by monotone } \gamma \\
 &= (\sqcup \circ (\gamma, \gamma))(a^\#, b^\#).
 \end{aligned}$$

# 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

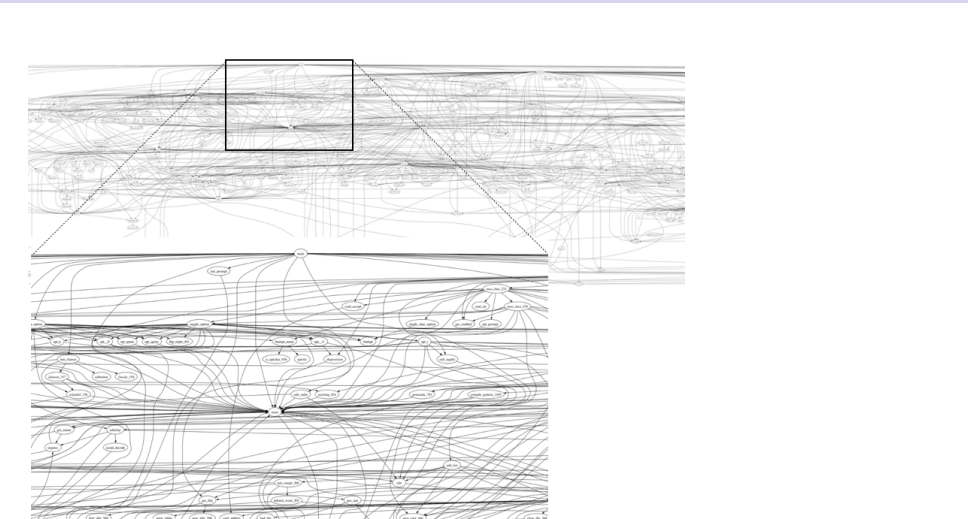


Figure: Call graph of less-382 (23,822 lines of code)

# 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

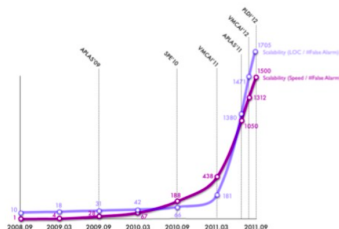
- Sparrow: a *sound*, global C analyzer for the memory safety property (no overrun, no null-pointer dereference, etc.)

<http://github.com/ropas/sparrow>

- ~ 10 hours in analyzing million lines of C [PLDI'12, TOPLAS'14]



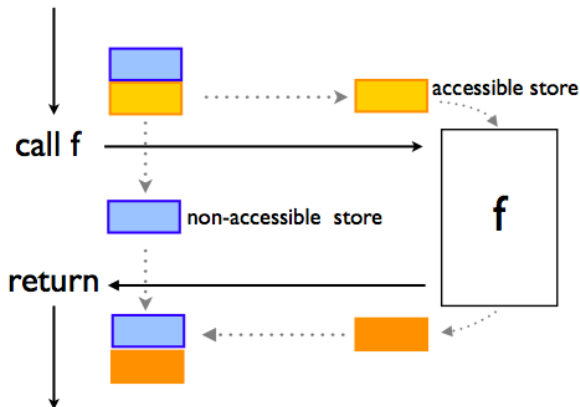
sound-&-global version



- < 1.4M in 10hrs with intervals
- < 0.14M in 20hrs with octagons

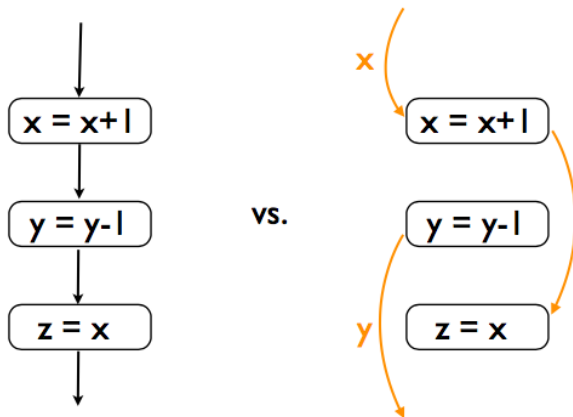
# Spatial Sparsity

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



# Temporal Sparsity

After the def of a memory, its use is far.



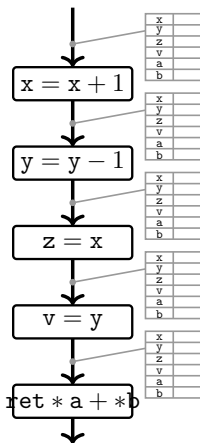
## Example (Code fragment)

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

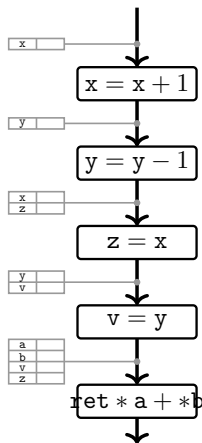
Assume that a points to v and b to z.



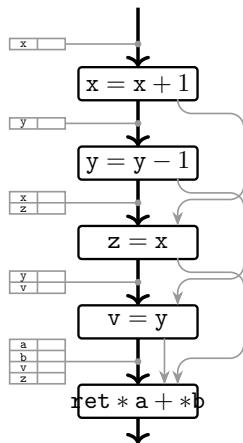
# Spatial and Temporal Sparsity of the Example Code



(a) Without exploiting the sparsities



(b) Spatial sparsity



(c) Spatial & temporal sparsity

Exploiting Spatial Sparsity: Need  $Access^\sharp(l)$ 

“abstract garbage collection”, “frame rule”

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

becomes

$$F_{sparse}^\sharp : (\mathbb{L} \rightarrow \mathbb{M}_{sparse}^\sharp) \rightarrow (\mathbb{L} \rightarrow \mathbb{M}_{sparse}^\sharp)$$

where

$$\mathbb{M}_{sparse}^\sharp = \{M^\sharp \in \mathbb{M}^\sharp \mid \text{dom}(M^\sharp) = \text{Access}^\sharp(l), l \in \mathbb{L}\} \cup \{\perp\}.$$

# Exploiting Temporal Sparsity: Need Def-Use Chain

Need the def-use chain information as follows.

- we streamline the abstract one-step relation

$$(l, M^\#) \hookrightarrow^\# (l', M'^\#) \quad \text{for } l' \in \text{next}^\#(l, M^\#).$$

so that the link  $\hookrightarrow^\#$  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

# Precision Preserving Sparse Analysis Framework

## Goal

$$F^\sharp : D^\sharp \rightarrow D^\sharp \xRightarrow{\text{sparsify}} F_{sparse}^\sharp : D^\sharp \rightarrow D^\sharp$$

$$\text{lfp} F^\sharp \stackrel{\text{still}}{=} \text{lfp} F_{sparse}^\sharp$$

# Precision Preserving Sparse Analysis: for Spatial Sparsity (1/3)

Need to safely estimate

$$Access^\sharp(l).$$

Use yet another sound static analysis, a further abstraction:

$$(\mathbb{L} \rightarrow \mathbb{M}^\sharp, \sqsubseteq) \xrightleftharpoons[\alpha]{\gamma} (\mathbb{M}^\sharp, \sqsubseteq_M)$$

(a “flow-insensitive” version of the “flow-sensitive” analysis design)

# Precision Preserving Sparse Analysis: for Temporal Sparsity (2/3)

- Let

$$D^\sharp : \mathbb{L} \rightarrow \wp(\mathbb{X}) \text{ and } U^\sharp : \mathbb{L} \rightarrow \wp(\mathbb{X})$$

be the def and use sets from the original analysis.

- Need to safely estimate  $D^\sharp$  and  $U^\sharp$ .
- Use yet another sound static analysis to compute

$$D_{pre}^\sharp \text{ and } U_{pre}^\sharp$$

such that

- ▶  $\forall l \in \mathbb{L} : D_{pre}^\sharp(l) \supseteq D^\sharp(l) \quad \text{and} \quad U_{pre}^\sharp(l) \supseteq U^\sharp(l).$
- ▶  $\forall l \in \mathbb{L} : U_{pre}^\sharp(l) \supseteq D_{pre}^\sharp(l) \setminus D^\sharp(l).$

## Precision Preserving Sparse Analysis: for Temporal Sparsity (3/3)

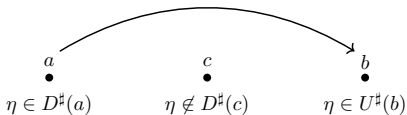
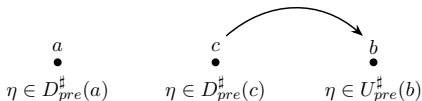
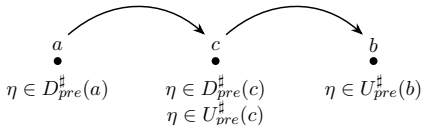
Let  $D_{pre}^\#$  and  $U_{pre}^\#$  be, respectively, safe def and use sets from a pre-analysis as defined before.

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



# 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,
- $\exists$  frameworks that are simple yet powerful enough
- review of their limitations

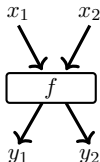
Three specialized frameworks:

- 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



we set up equations

$$y_1 = f(x_1 \sqcup x_2)$$

$$y_2 = f(x_1 \sqcup x_2)$$

# Example: Data-Flow Analysis for Integer Intervals

## Example (Data-flow analysis)

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

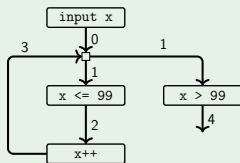


Figure: Control-flow graph

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

Figure: A set of equations for the program

# 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

# 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  $Facts$  be the set of all possible facts

Then, the analysis result is

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

where

$$\begin{aligned} Y_0 &= X_0, \\ Y_{i+1} &= Y \text{ such that } Y_i \vdash_R Y. \end{aligned}$$

Or, equivalently, the analysis result is the least fixpoint

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

of monotonic function  $\phi : \wp(Facts) \rightarrow \wp(Facts)$  :

$$\phi(X) = X_0 \cup (Y \text{ such that } X \vdash_R Y).$$

# Example: Pointer Analysis (1/3)

$P$	$::=$	$C$	program
$C$	$::=$		statement
		$L := R$	assignment
		$C ; C$	sequence
		<b>while</b> $B$ $C$	while-loop
$L$	$::=$	$x \mid *x$	target to assign to
$R$	$::=$	$n \mid x \mid *x \mid \&x$	value to assign
$B$			Boolean expression

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



## 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)

```
x := &a ; y := &x ;
while B
    *y := &b ;
*x := *y
```

- Initial facts are from the first two assignments:

$$x \rightarrow a, y \rightarrow x$$

- From  $y \rightarrow x$  and the while-loop body, add

$$x \rightarrow b$$

- From the last assignment:

- from  $x \rightarrow a$  and  $y \rightarrow x$ , add  $a \rightarrow a$
- from  $x \rightarrow b$  and  $y \rightarrow x$ , add  $b \rightarrow b$
- from  $x \rightarrow a$ ,  $y \rightarrow x$ , and  $x \rightarrow b$ , add  $a \rightarrow b$
- from  $x \rightarrow b$ ,  $y \rightarrow x$ , and  $x \rightarrow a$ , 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)

$P$	$::=$	$E$	program
$E$	$::=$		expression
		$n$	integer
		$x$	variable
		$\lambda x. E$	function
		$E E$	function application

- 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*

$$\tau ::= int \mid \tau \rightarrow \tau$$

$$\frac{}{\Gamma \vdash n : int} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

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

Figure: Proof rules of simple types

### 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 *int* because we can prove

$$\emptyset \vdash (\lambda x.x\ 1)(\lambda y.y) : int$$

as follows:

$$\begin{array}{c}
 \frac{x : int \rightarrow int \in \{x : int \rightarrow int\}}{\{x : int \rightarrow int\} \vdash x : int \rightarrow int} \quad \frac{}{\{x : int \rightarrow int\} \vdash 1 : int} \quad \frac{y : int \in \{y : int\}}{\{y : int\} \vdash y : int} \\
 \hline
 \frac{\{x : int \rightarrow int\} \vdash x\ 1 : int}{\emptyset \vdash \lambda x.x\ 1 : (int \rightarrow int) \rightarrow int} \quad \frac{\{y : int\} \vdash y : int}{\emptyset \vdash \lambda y.y : int \rightarrow int} \\
 \hline
 \emptyset \vdash (\lambda x.x\ 1)(\lambda y.y) : int
 \end{array}$$

# Example: Type Inference (4/4)

## Algorithm

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

$$V(\Gamma, n, \tau) = \{\tau \doteq \text{int}\}$$

$$V(\Gamma, \mathbf{x}, \tau) = \{\tau \doteq \Gamma(\mathbf{x})\}$$

$$V(\Gamma, \lambda \mathbf{x}. E, \tau) = \{\tau \doteq \alpha_1 \rightarrow \alpha_2\} \cup V(\Gamma + \mathbf{x} : \alpha_1, E, \alpha_2) \quad (\text{new } \alpha_i)$$

$$V(\Gamma, E_1 E_2, \tau) = V(\Gamma, E_1, \alpha \rightarrow \tau) \cup V(\Gamma, E_2, \alpha) \quad (\text{new } \alpha)$$

- 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.

# Static Analysis: an Abstract Interpretation Perspective

- 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

Thank you!