Practical Abstract Interpretation with LiSA

Luca Negrini

Department of Environmental Sciences, Informatics and Statistics, Ca’ Foscari University of Venice

luca.negrini@unive.it

Abstract Interpretation Overview

Given `P` we want to compute properties over its undecidable semantics

Abstract Interpretation Overview

Given `P` we want to compute properties over its undecidable semantics

We define `P`’s semantics as the fixpoint of a monotone function, and then we abstract…

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$'s semantics as the fixpoint of a monotone function, and then we abstract...

$$\mathtt{P}\ (\mathcal{D})$$

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$’s semantics as the fixpoint of a monotone function, and then we abstract…

$$\mathtt{P}\,(\mathcal{D}) \xrightarrow{\text{undecidable}} \mathcal{S}$$

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$'s semantics as the fixpoint of a monotone function, and then we abstract…

$$\begin{tikzcd}
\mathtt{P}\ (\mathcal{D}) \arrow[r, "\text{undecidable}"] \arrow[d, "\alpha"] & \mathcal{S} \\
\mathtt{P}\ (\mathcal{D}^{\sharp}) &
\end{tikzcd}$$

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$’s semantics as the fixpoint of a monotone function, and then we abstract...

$$\begin{tikzcd}
\mathtt{P}\,(\mathcal{D}) \arrow[r, "\text{undecidable}"] \arrow[d, "\alpha"] & \mathcal{S} \\
\mathtt{P}\,(\mathcal{D}^\sharp) \arrow[r, "\text{decidable}"] & \mathcal{S}^\sharp
\end{tikzcd}$$

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$'s semantics as the fixpoint of a monotone function, and then we abstract...

$$\begin{tikzcd}
\mathtt{P}\,(\mathcal{D}) \arrow[r, "\text{undecidable}"] \arrow[d, "\alpha"] & \mathcal{S} \\
\mathtt{P}\,(\mathcal{D}^\sharp) \arrow[r, "\text{decidable}"] & \mathcal{S}^\sharp \arrow[u, Rightarrow]
\end{tikzcd}$$

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$’s semantics as the fixpoint of a monotone function, and then we abstract...

$$\begin{tikzcd}
\mathtt{P}\,(\mathcal{D}) \arrow[r, "\text{undecidable}"] \arrow[d, "\alpha"] & \mathcal{S} \\
\mathtt{P}\,(\mathcal{D}^{\sharp}) \arrow[r, "\text{decidable}"] & \mathcal{S}^{\sharp} \arrow[u, Rightarrow]
\end{tikzcd}$$

Abstract Interpretation is a framework!

We just define:

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$'s semantics as the fixpoint of a monotone function, and then we abstract…

$$\begin{tikzcd}
\mathtt{P}\ (\mathcal{D}) \arrow[rr, "\text{undecidable}"] \arrow[d, "\alpha"] & & \mathcal{S} \\
\mathtt{P}\ (\mathcal{D}^\sharp) \arrow[rr, "\text{decidable}"] & & \mathcal{S}^\sharp \arrow[u, Rightarrow]
\end{tikzcd}$$

Abstract Interpretation is a framework!

We just define:

- ▷ a set of elements $\mathcal{D}^\sharp$

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$'s semantics as the fixpoint of a monotone function, and then we abstract...

$$\begin{tikzcd}
\mathtt{P}\ (\mathcal{D}) \arrow[rr, "\text{undecidable}"] \arrow[d, "\alpha"] & & \mathcal{S} \\
\mathtt{P}\ (\mathcal{D}^\sharp) \arrow[rr, "\text{decidable}"] & & \mathcal{S}^\sharp \arrow[u, Rightarrow]
\end{tikzcd}$$

Abstract Interpretation is a framework!

We just define:

- ▷ a set of elements $\mathcal{D}^\sharp$
- ▷ operators $\sqsubseteq$, $\sqcup$, $\sqcap$, ∇, Δ over $\mathcal{D}^\sharp$

Abstract Interpretation Overview

Given $\mathtt{P}$ we want to compute properties over its undecidable semantics

We define $\mathtt{P}$'s semantics as the fixpoint of a monotone function, and then we abstract...

$$\begin{tikzcd}
\mathtt{P}\ (\mathcal{D}) \arrow[r, "\text{undecidable}"] \arrow[d, "\alpha"] & \mathcal{S} \\
\mathtt{P}\ (\mathcal{D}^\sharp) \arrow[r, "\text{decidable}"] & \mathcal{S}^\sharp \arrow[u, Rightarrow]
\end{tikzcd}$$

Abstract Interpretation is a framework!

We just define:

▷ a set of elements $\mathcal{D}^\sharp$

▷ operators $\sqsubseteq, \sqcup, \sqcap, \nabla, \Delta$ over $\mathcal{D}^\sharp$

▷ transformers $\mathbb{S}^\sharp[\![\cdot]\!], \mathbb{E}^\sharp[\![\cdot]\!], \mathbb{C}^\sharp[\![\cdot]\!]$ over $\mathcal{D}^\sharp$

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = \textsc{Intv}$:

```
$\ell_0$N = int(input())
$\ell_1$x = 0
$\ell_2$while $\ell_3$x < N:
    $\ell_4$x += 1$\ell_5$
$\ell_6$print(x)$\ell_7$
```

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = \textsc{Intv}$:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
  ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$\ell_0 : \{\}$

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = \textsc{Intv}$:

```
$\ell_0$N = int(input())
$\ell_1$x = 0
$\ell_2$while $\ell_3$x < N:
  $\ell_4$x += 1$\ell_5$
$\ell_6$print(x)$\ell_7$
```

$$\ell_0 : \{\}$$

$$\ell_1 : \{\mathrm{N} \to [-\infty, +\infty]\}$$

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = $ Intv:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
  ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$$\ell_0 : \{\}$$

$$\ell_1 : \{\mathtt{N} \rightarrow [-\infty, +\infty]\}$$

$$\ell_2 : \{\mathtt{N} \rightarrow [-\infty, +\infty], \mathtt{x} \rightarrow [0, 0]\}$$

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = $ INTV:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
  ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$$\ell_0 : \{\}$$

$$\ell_1 : \{\mathrm{N} \to [-\infty, +\infty]\}$$

$$\ell_2 : \{\mathrm{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

$$\ell_3 : \{\mathrm{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = \textsc{Intv}$:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
  ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$$\ell_0 : \{\}$$

$$\ell_1 : \{\mathtt{N} \to [-\infty, +\infty]\}$$

$$\ell_2 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

$$\ell_3 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

$$\ell_4 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = \textsc{Intv}$:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
  ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$$\ell_0 : \{\}$$

$$\ell_1 : \{\mathtt{N} \to [-\infty, +\infty]\}$$

$$\ell_2 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

$$\ell_3 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

$$\ell_4 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

$$\ell_5 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [1, 1]\}$$

We must update ℓ_3!

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = $ INTV:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
  ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$$\ell_0 : \{\}$$

$$\ell_1 : \{\mathtt{N} \to [-\infty, +\infty]\}$$

$$\ell_2 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

$$\ell_3 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 1]\}$$

$$\ell_4 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 1]\}$$

$$\ell_5 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [1, 2]\}$$

We must keep updating ℓ_3!

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = $ INTV:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
    ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$$\ell_0 : \{\}$$

$$\ell_1 : \{\mathtt{N} \to [-\infty, +\infty]\}$$

$$\ell_2 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$$

$$\ell_3 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 2]\}$$

$$\ell_4 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 2]\}$$

$$\ell_5 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [1, 3]\}$$

We must keep updating ℓ_3 with ∇!

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = $ INTV:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
    ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$\ell_0 : \{\}$

$\ell_1 : \{\mathtt{N} \to [-\infty, +\infty]\}$

$\ell_2 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, 0]\}$

$\ell_3 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, +\infty]\}$

$\ell_4 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [0, +\infty]\}$

$\ell_5 : \{\mathtt{N} \to [-\infty, +\infty], \mathtt{x} \to [1, +\infty]\}$

We must keep updating ℓ_3 with ∇!

Applying Abstract Interpretation

You can apply $\mathcal{D}^\sharp$ to compute program invariants!

For instance, using $\mathcal{D}^\sharp = \textsc{Intv}$:

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
   ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$$\ell_0 : \{\}$$

$$\ell_1 : \{\mathtt{N} \rightarrow [-\infty, +\infty]\}$$

$$\ell_2 : \{\mathtt{N} \rightarrow [-\infty, +\infty], \mathtt{x} \rightarrow [0, 0]\}$$

$$\ell_3 : \{\mathtt{N} \rightarrow [-\infty, +\infty], \mathtt{x} \rightarrow [0, +\infty]\}$$

$$\ell_4 : \{\mathtt{N} \rightarrow [-\infty, +\infty], \mathtt{x} \rightarrow [0, +\infty]\}$$

$$\ell_5 : \{\mathtt{N} \rightarrow [-\infty, +\infty], \mathtt{x} \rightarrow [1, +\infty]\}$$

$$\ell_6 : \{\mathtt{N} \rightarrow [-\infty, +\infty], \mathtt{x} \rightarrow [0, +\infty]\}$$

$$\ell_7 : \{\mathtt{N} \rightarrow [-\infty, +\infty], \mathtt{x} \rightarrow [0, +\infty]\}$$

We must keep updating ℓ_3 with ∇!

How do we put this in practice?

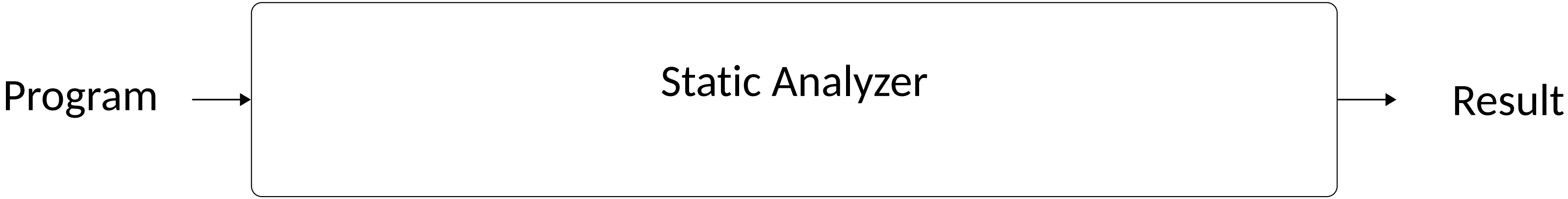
Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

From Theory to Practice

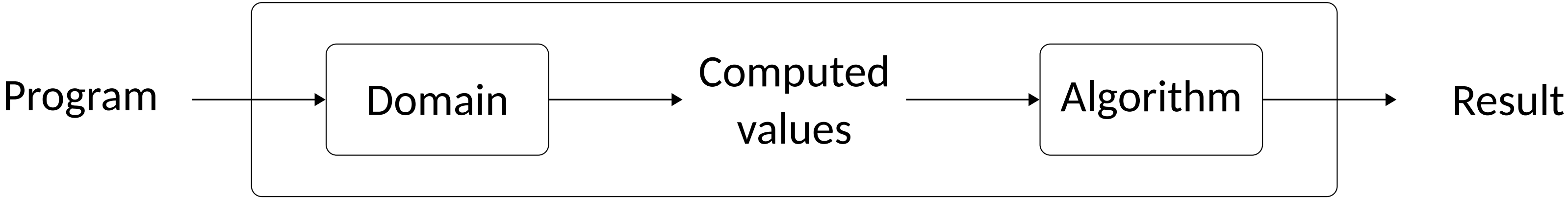


Program

Static Analyzer

Result

From Theory to Practice



Program

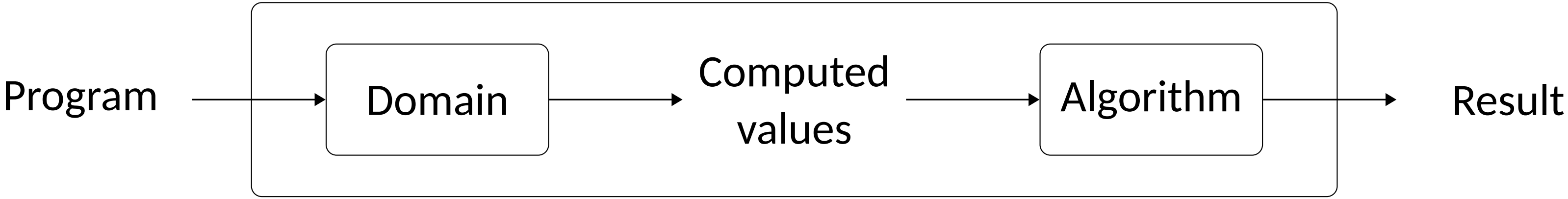
Domain

Computed values

Algorithm

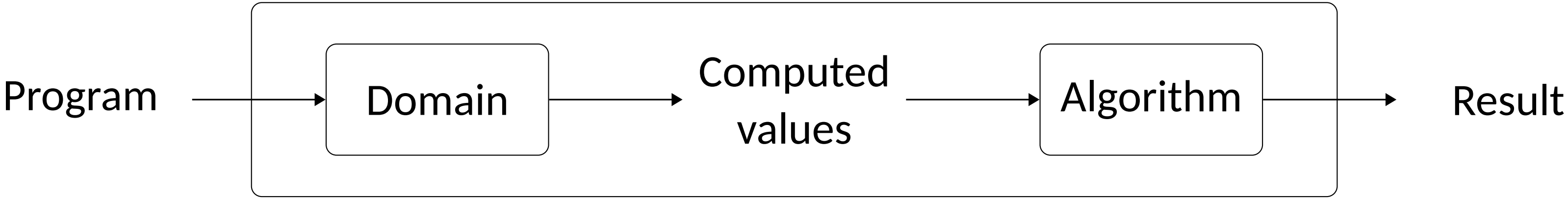
Result

From Theory to Practice



```
y = (2 * 2) - 4
    z = 1 / y
```

From Theory to Practice

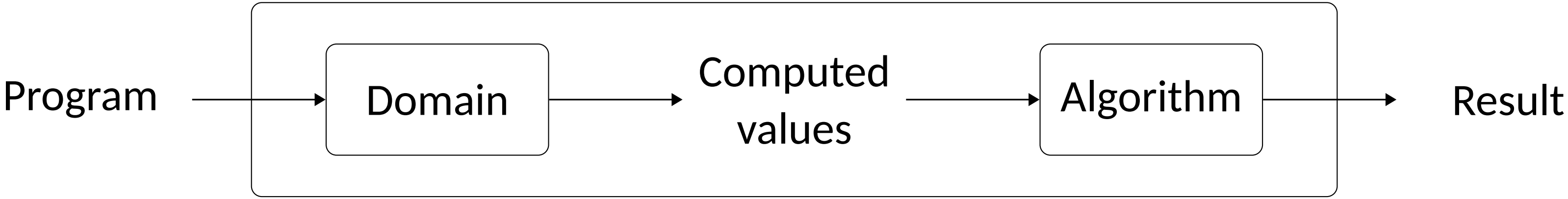


y = (2 * 2) - 4
z = 1 / y

constant propagation

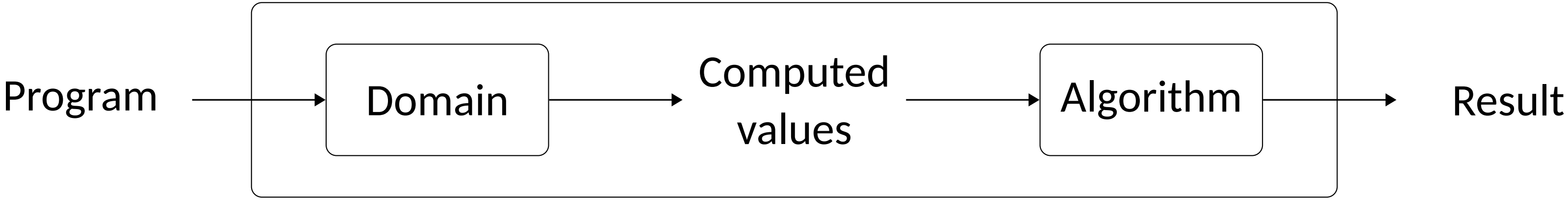
$y \mapsto 0$
$z \mapsto$ error

From Theory to Practice



y = (2 * 2) - 4
z = 1 / y
	constant propagation	$y \mapsto 0$
		$z \mapsto$ error
			check for div by zero	message to the user

From Theory to Practice



y = (2 * 2) - 4
z = 1 / y

constant propagation

$y \mapsto 0$
$z \mapsto$ error

check for div by zero

message to the user

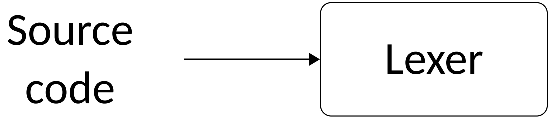
how do we parse the code?

A (Simplified) Compiler/Interpreter

Source
code

```
x = y + 2
```

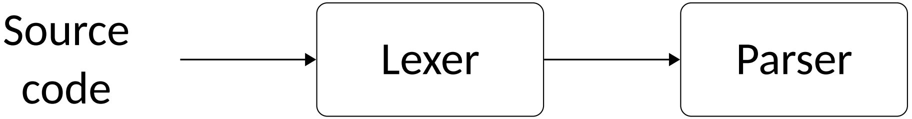
A (Simplified) Compiler/Interpreter



x = y + 2

ID EQ ID
PLUS LITERAL

A (Simplified) Compiler/Interpreter



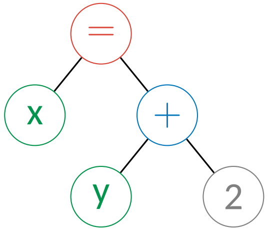
Source code

Lexer

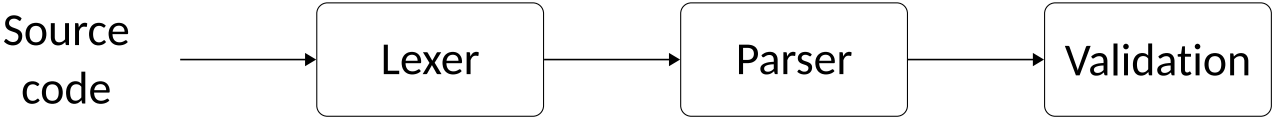
Parser

x = y + 2

ID EQ ID
PLUS LITERAL



A (Simplified) Compiler/Interpreter



Source code

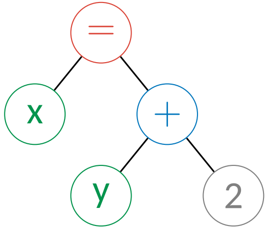
Lexer

Parser

Validation

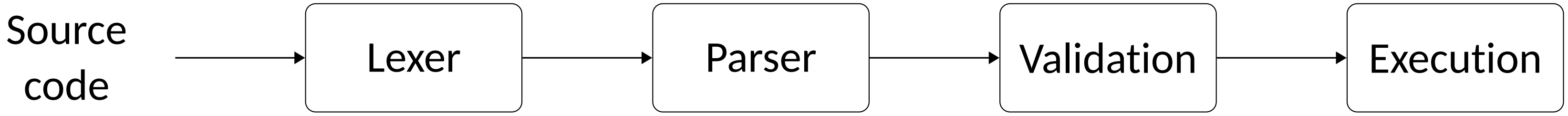
x = y + 2

ID EQ ID PLUS LITERAL



type check,
duplicate functions,
...

A (Simplified) Compiler/Interpreter



Source code

Lexer

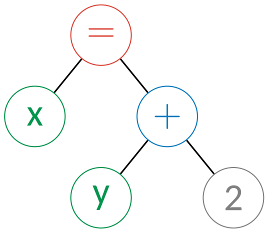
Parser

Validation

Execution

x = y + 2

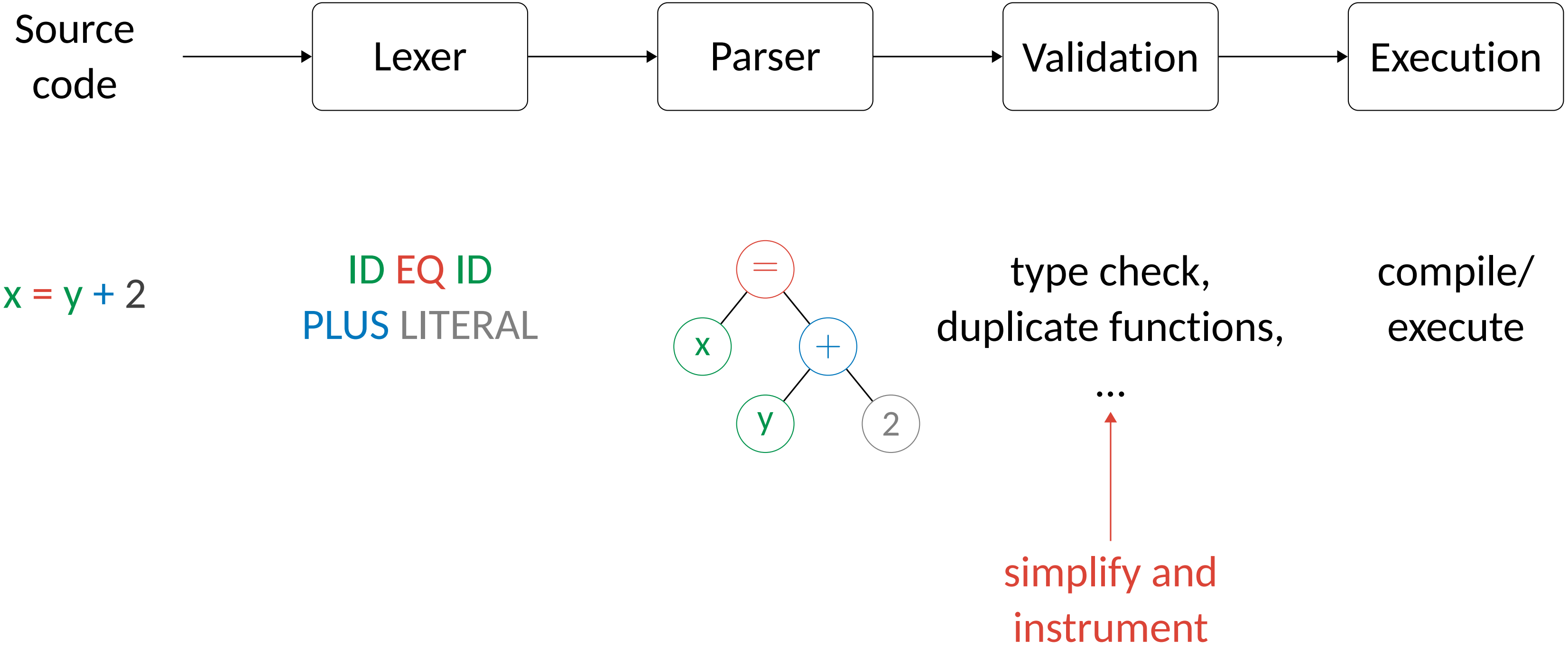
ID EQ ID PLUS LITERAL



type check, duplicate functions, …

compile/ execute

A (Simplified) Compiler/Interpreter



Source code

Lexer

Parser

Validation

Execution

x = y + 2

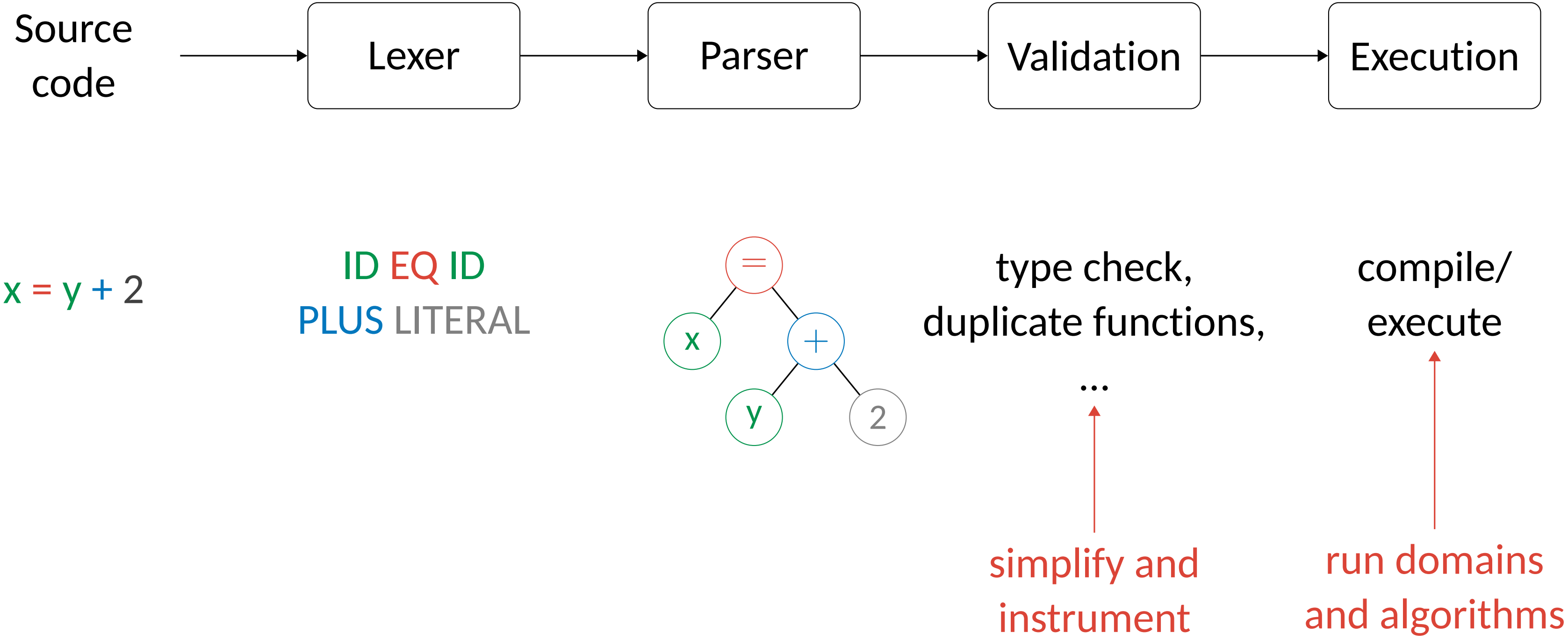
ID EQ ID PLUS LITERAL

type check, duplicate functions, …

compile/ execute

simplify and instrument

A (Simplified) Compiler/Interpreter



Source code

Lexer

Parser

Validation

Execution

x = y + 2

ID EQ ID PLUS LITERAL

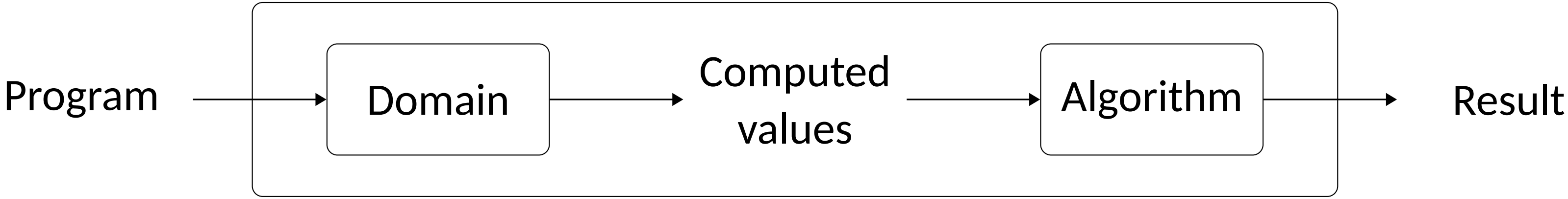
type check, duplicate functions, …

compile/ execute

simplify and instrument

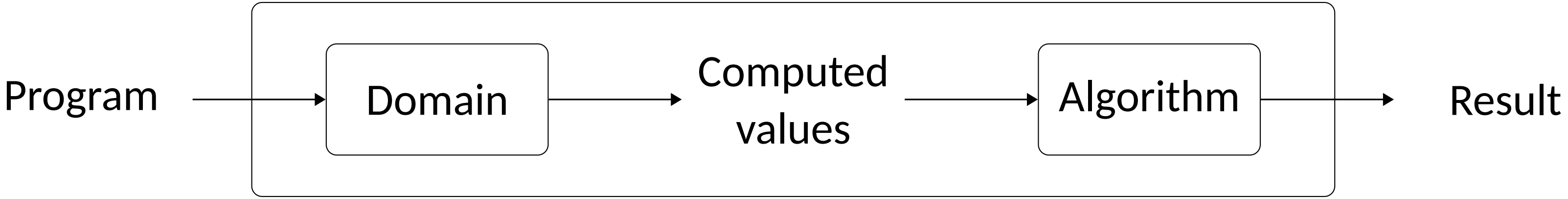
run domains and algorithms

From Theory to Practice



y = (2 * 2) - 4
z = 1 / y
	constant propagation	$y \mapsto 0$
		$z \mapsto$ error
			check for div by zero	message to the user

From Theory to Practice



Program

Domain

Computed values

Algorithm

Result

y = (2 * 2) - 4
z = 1 / y
	constant propagation	$y \mapsto 0$
		$z \mapsto$ error
			check for div by zero	message to the user

how do we run the domain?

Running Fixpoints

- **Solving a system of equations** [CC92]

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
  ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```

$\mathcal{X}_i$ is state reaching ℓ_i:

$$\begin{cases} \mathcal{X}_0 = \mathcal{I} & \mathcal{X}_1 = \mathbb{S}^\sharp[\![\texttt{N = int(input())}]\!]\mathcal{X}_0 \\ \mathcal{X}_2 = \mathbb{S}^\sharp[\![\texttt{x = 0}]\!]\mathcal{X}_1 & \mathcal{X}_3 = \mathcal{X}_2 \nabla^n \mathcal{X}_5 \\ \mathcal{X}_4 = \mathbb{S}^\sharp[\![\texttt{x < N}]\!]\mathcal{X}_3 & \mathcal{X}_5 = \mathbb{S}^\sharp[\![\texttt{x += 1}]\!]\mathcal{X}_4 \\ \mathcal{X}_6 = \mathbb{S}^\sharp[\![\texttt{not x < N}]\!]\mathcal{X}_2 & \mathcal{X}_7 = \mathbb{S}^\sharp[\![\texttt{print(x)}]\!]\mathcal{X}_6 \end{cases}$$

Running Fixpoints

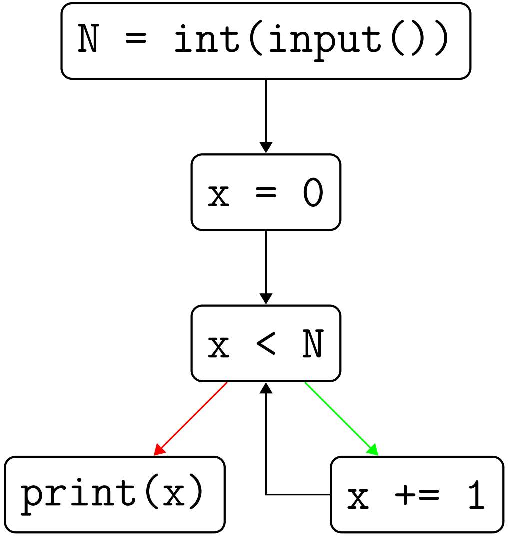
- ▷ **Using inductive abstract interpreter** [Cou21]

$$\begin{aligned}
\mathbb{S}^\sharp[\![P \triangleq st;\,]\!]\mathcal{X} &\triangleq \mathbb{S}^\sharp[\![st]\!]\mathcal{X} \\
\mathbb{S}^\sharp[\![x = e]\!]\mathcal{X} &\triangleq \mathtt{assign}[\![x = e]\!]\mathcal{X} \\
\mathbb{S}^\sharp[\![st_1; st_2]\!]\mathcal{X} &\triangleq \mathbb{S}^\sharp[\![st_2]\!](\mathbb{S}^\sharp[\![st_1]\!]\mathcal{X}) \\
\mathbb{S}^\sharp[\![\textbf{if } b \textbf{ then } st_1 \textbf{ else } st_2]\!]\mathcal{X} &\triangleq \mathbb{S}^\sharp[\![st_1]\!](\mathtt{assume}[\![b]\!]\mathcal{X}) \sqcup \mathbb{S}^\sharp[\![st_2]\!](\mathtt{assume}[\![\mathtt{not}\ \ b]\!]\mathcal{X}) \\
\mathbb{S}^\sharp[\![\textbf{while } b \textbf{ do } st]\!]\mathcal{X} &\triangleq \mathtt{assume}[\![\mathtt{not}\ \ b]\!]\mathit{lfp}_F \ \text{(using } \nabla^n\text{)} \\
\text{where } F(\mathcal{Y}) &\triangleq \mathcal{X} \sqcup \mathbb{S}^\sharp[\![st]\!](\mathtt{assume}[\![b]\!]\mathcal{Y}) \\
&\dots
\end{aligned}$$

Running Fixpoints

- **Worklist fixpoint over a CFG [CC77]**

```
ℓ0N = int(input())
ℓ1x = 0
ℓ2while ℓ3x < N:
    ℓ4x += 1ℓ5
ℓ6print(x)ℓ7
```



N = int(input())

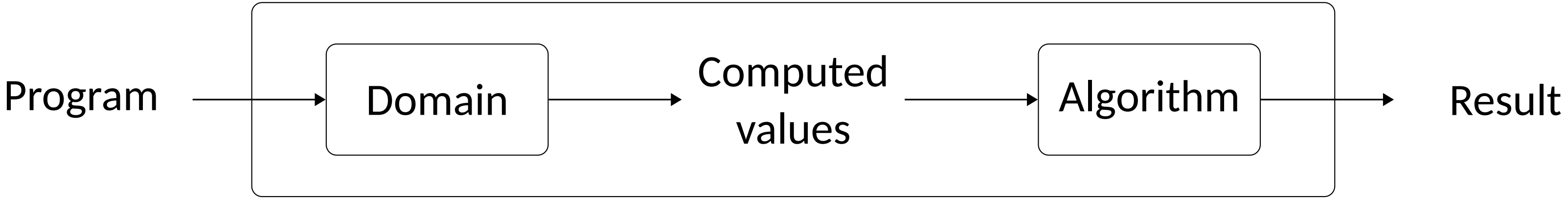
x = 0

x < N

print(x)

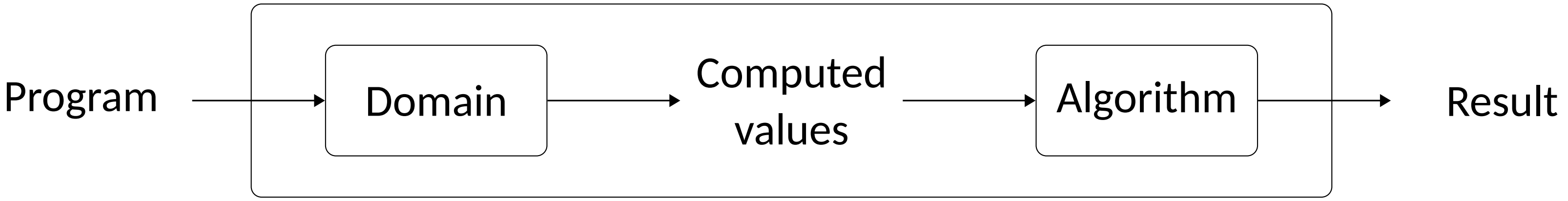
x += 1

From Theory to Practice



y = (2 * 2) - 4
z = 1 / y
	constant propagation	$y \mapsto 0$
		$z \mapsto$ error
			check for div by zero	message to the user

From Theory to Practice



y = (2 * 2) - 4
z = 1 / obj.f

constant propagation

$y \mapsto 0$
$z \mapsto$??

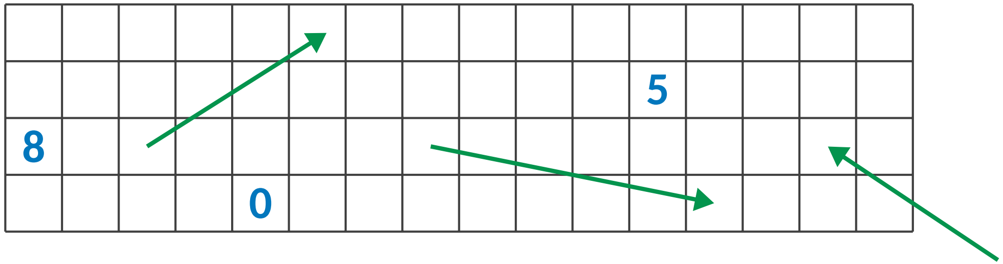
check for div by zero

message to the user

how do we track dynamic memory?

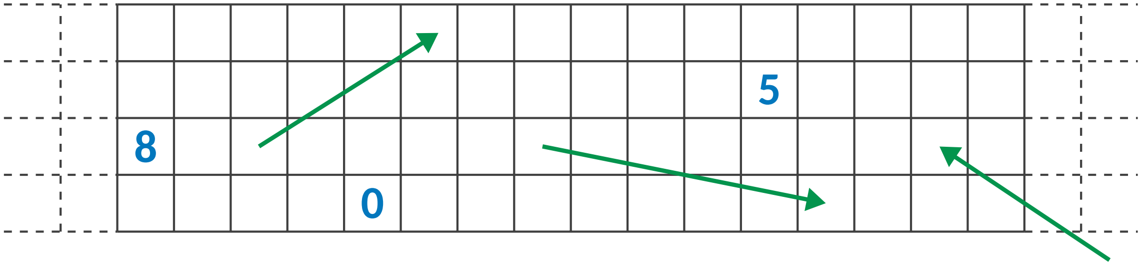
Abstracting Dynamic Memory

Analyzing memory means tracking which cells contain values and which ones refer to other cells



Abstracting Dynamic Memory

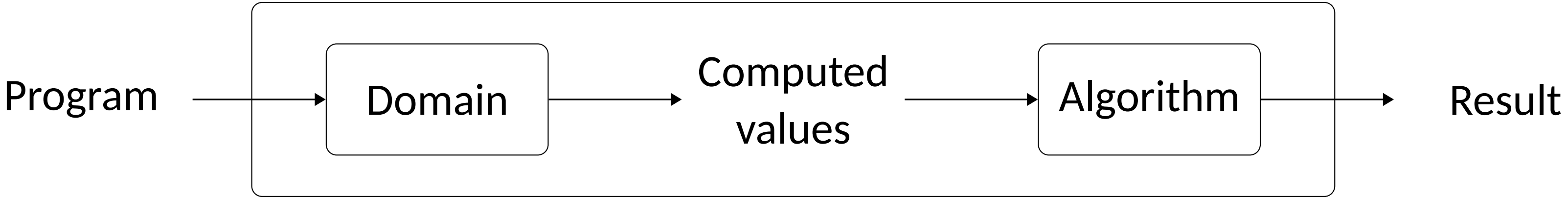
Analyzing memory means tracking which cells contain values and which ones refer to other cells



Variables are finite in number, (allocated) memory cells **might not be**

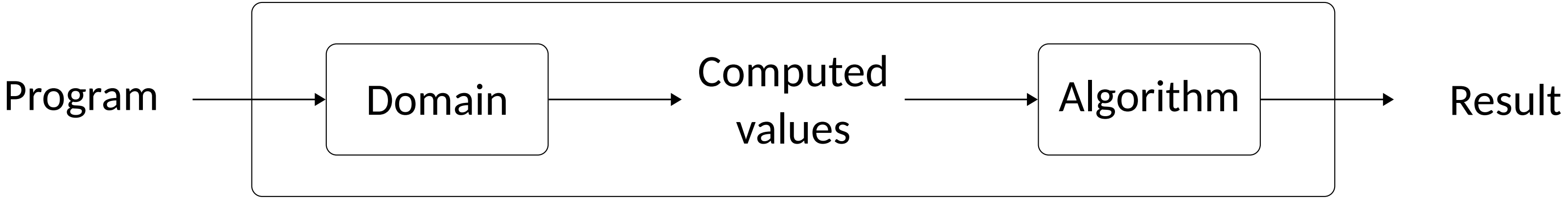
Usually, memory domains focus on shape approximations or specific properties [Hin01, KK16]

From Theory to Practice



y = (2 * 2) - 4
z = 1 / y
	constant propagation	$y \mapsto 0$
		$z \mapsto$ error
			check for div by zero	message to the user

From Theory to Practice



y = (2 * 2) - 4
z = 1 / f(y)

constant propagation

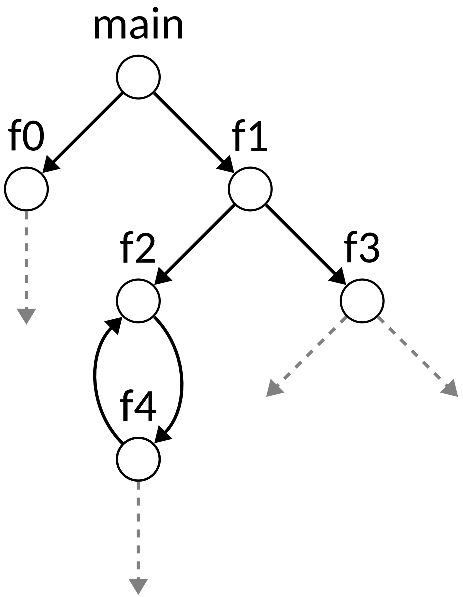
$y \mapsto 0$
$z \mapsto$??

check for div by zero

message to the user

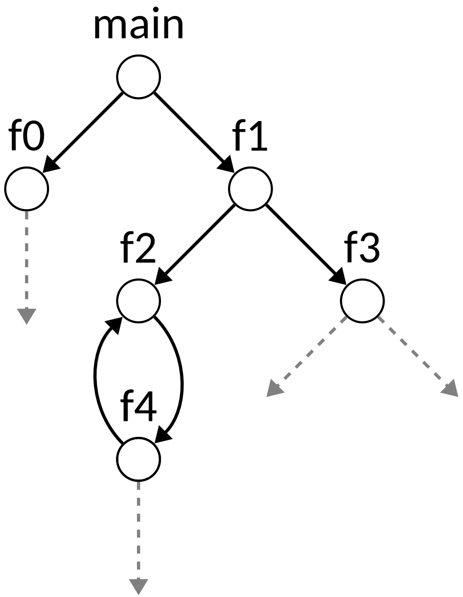
how do we handle calls and functions?

Abstracting Calls



Abstracting calls means modeling the call graph of the program

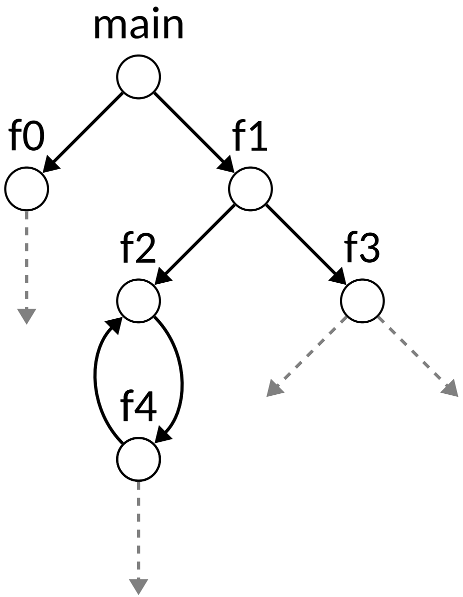
Abstracting Calls



Abstracting calls means modeling the call graph of the program

Solving calls is language-specific, but it generally requires typing information [DGC95, BS96, GDDC97]

Abstracting Calls



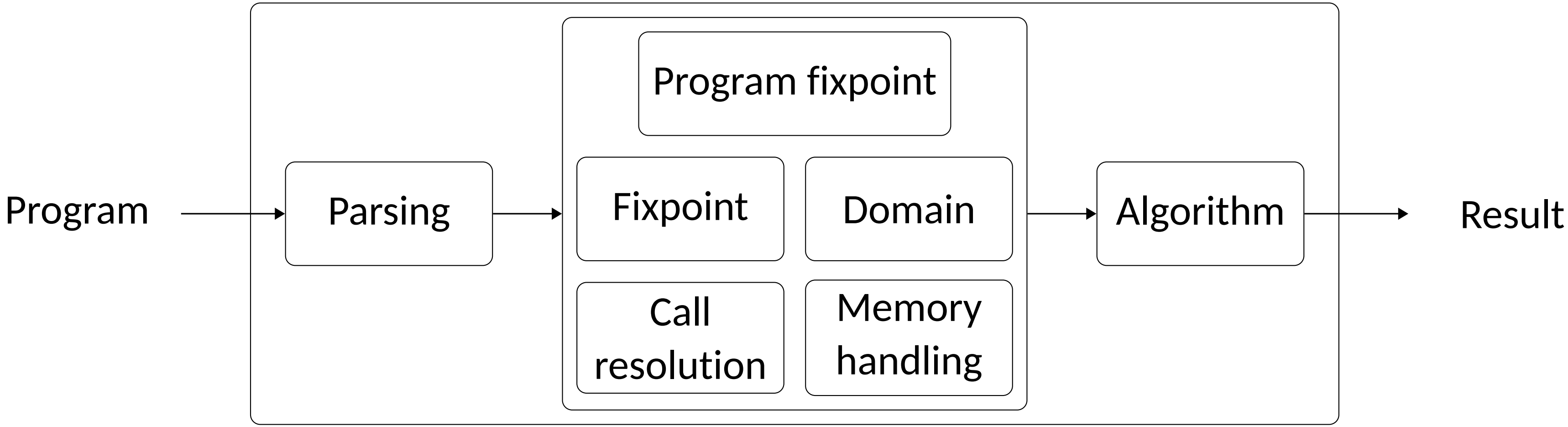
Abstracting calls means modeling the call graph of the program

Solving calls is language-specific, but it generally requires typing information [DGC95, BS96, GDDC97]

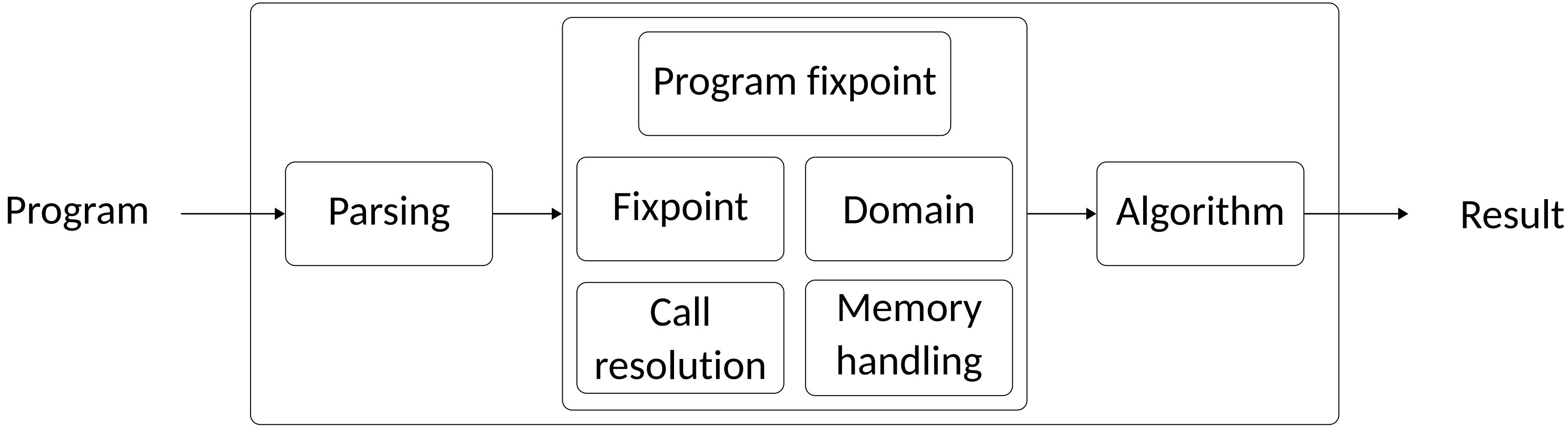
Call evaluation is highly dependent on how we run the overall analysis [PS81]:

- following call chains
- bottom-up
- …

A Realistic Analyzer's Structure



A Realistic Analyzer's Structure



Different ways to implement this!

Mainly a trade-off between efficiency and reusability

Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

LiSA, a Library for Static Analysis

Library for creating static analyzers based on abstract interpretation [NFAC23, Neg23]

- For multiple programming languages
- For a variety of different domains

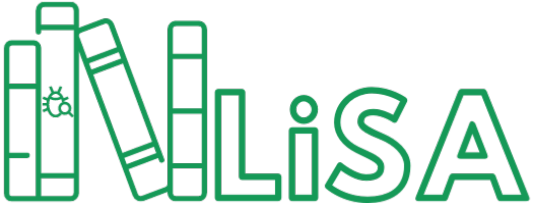
Purposes:

- Experiment with modular implementations
- Be easy to pick-up (simple, close to formalization)
- Fast prototyping of analyzers and domains

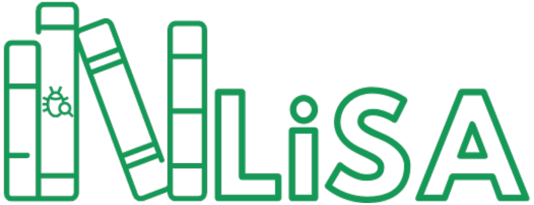
Open source Java library

Created and maintained by the SSV group @ Ca’ Foscari

LiSA is…

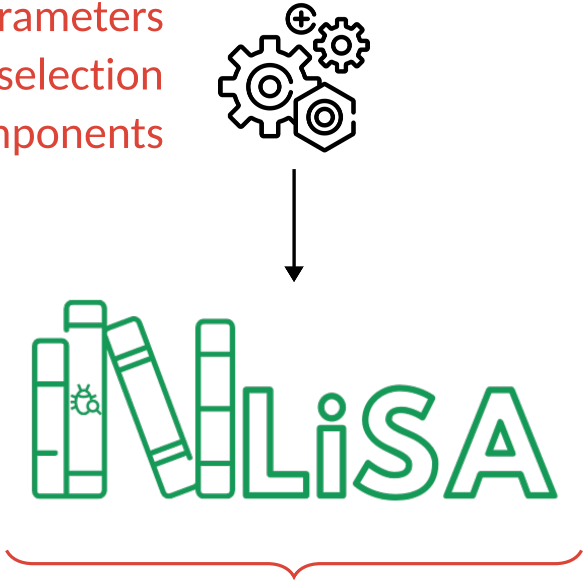


LiSA is…



generic analysis engine
pre-developed components

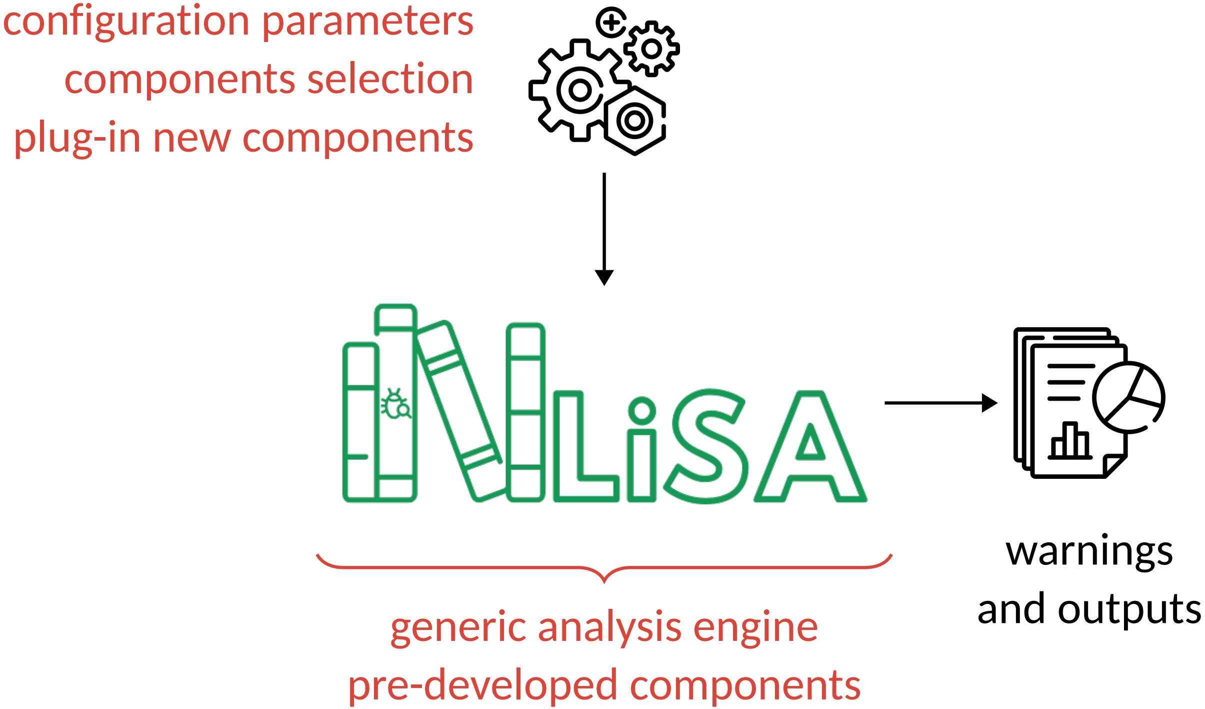
LiSA is...



configuration parameters
components selection
plug-in new components

generic analysis engine
pre-developed components

LiSA is…

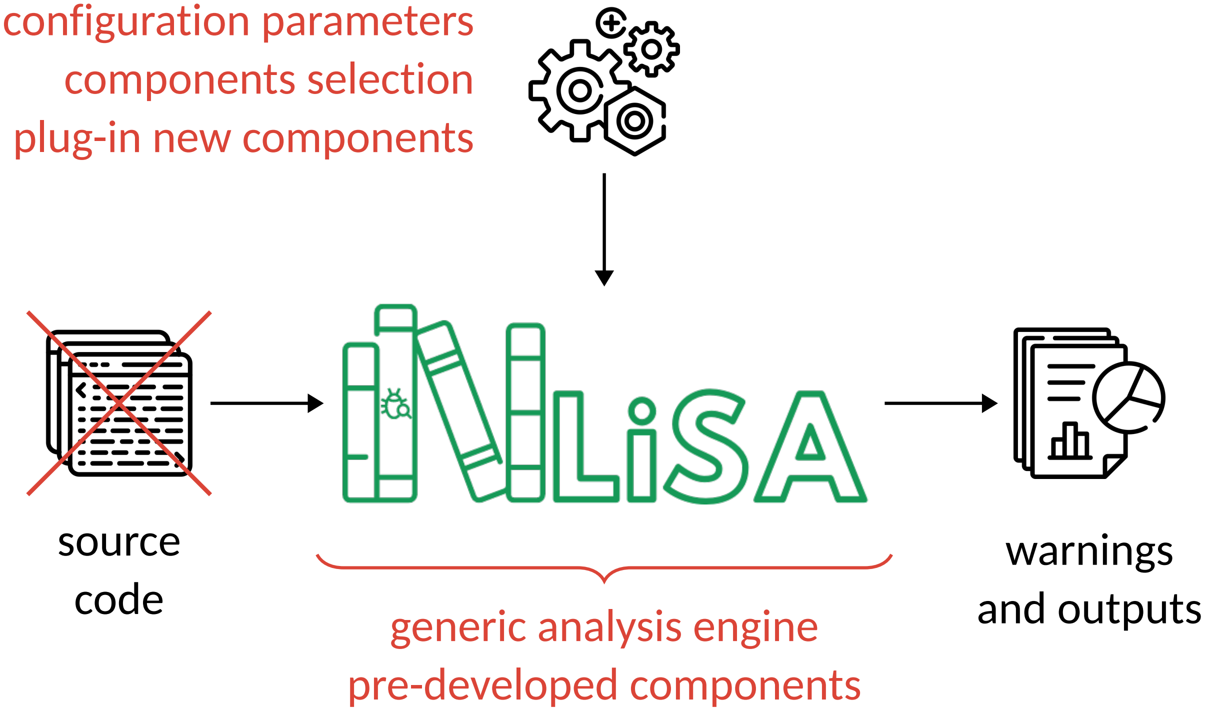


configuration parameters
components selection
plug-in new components

generic analysis engine
pre-developed components

warnings
and outputs

LiSA is… Not a Full Analyzer



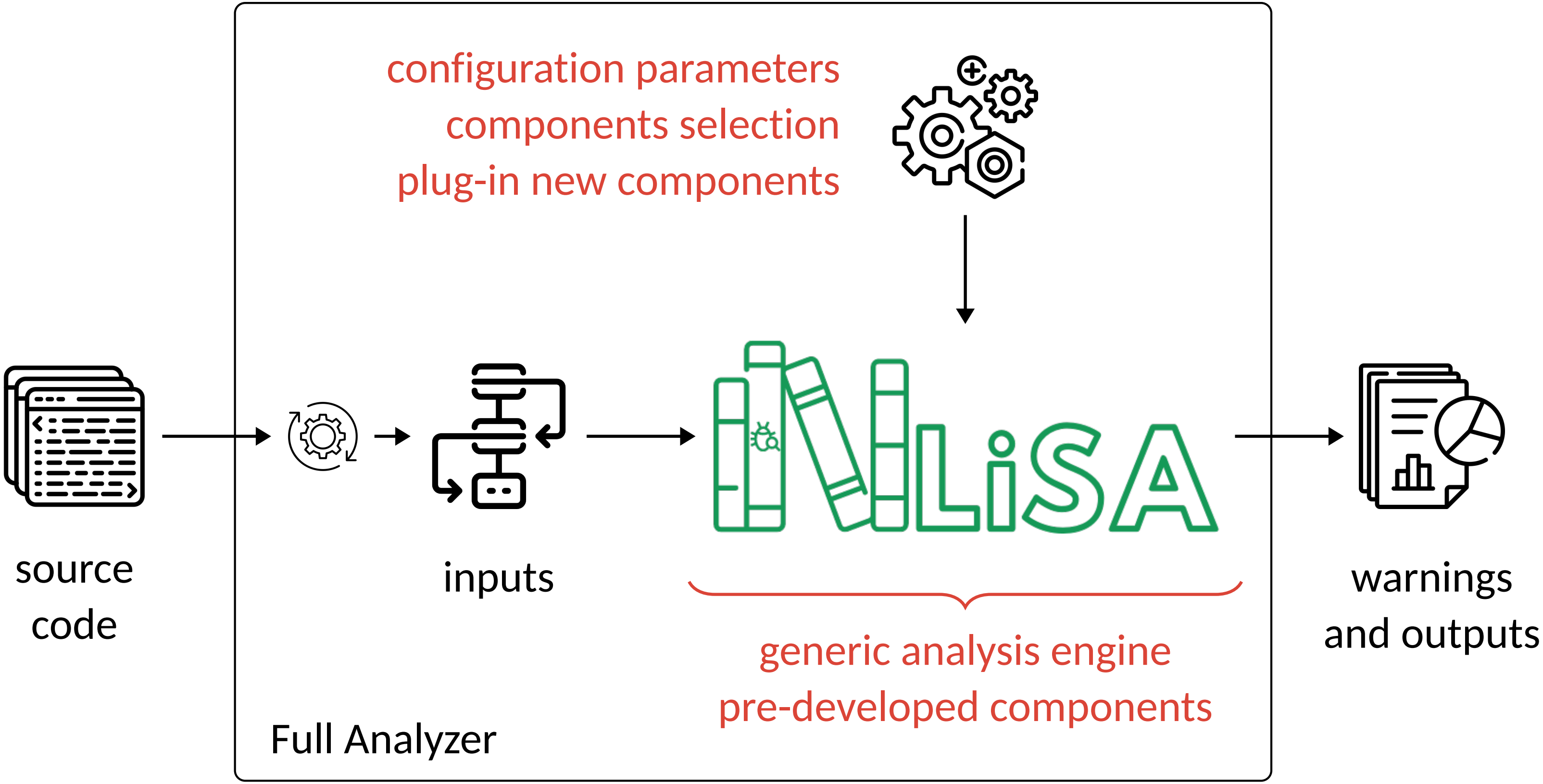
configuration parameters
components selection
plug-in new components

source code

generic analysis engine
pre-developed components

warnings and outputs

LiSA is... a Library



LiSA Today

Languages:

- Go
- Michelson
- EVM
- Python (wip)
- Rust (wip)
- LLVM (wip)

Topics:

- Blockchain
- Strings
- Dynamic languages
- Numeric trends
- Data Science (wip)
- Modular interactions (wip)

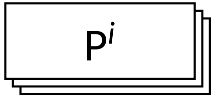
Teaching:

- SCSR @ Ca’ Foscari: used for 4 years!
- Seminars and tutorials all around

Today's Plan

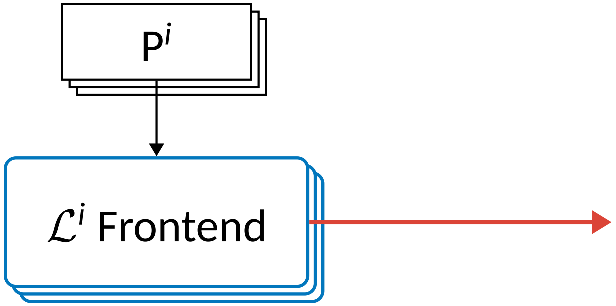
1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA's High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

LiSA Overview



P^i

LiSA Overview



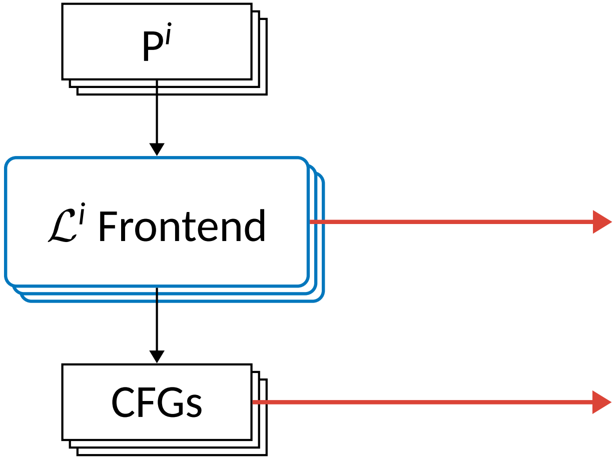
Compilation to LiSA program

Definition of the language **execution model**

Definition of **instruction semantics**

□ language dependent

LiSA Overview



Compilation to LiSA program

Definition of the language **execution model**

Definition of **instruction semantics**

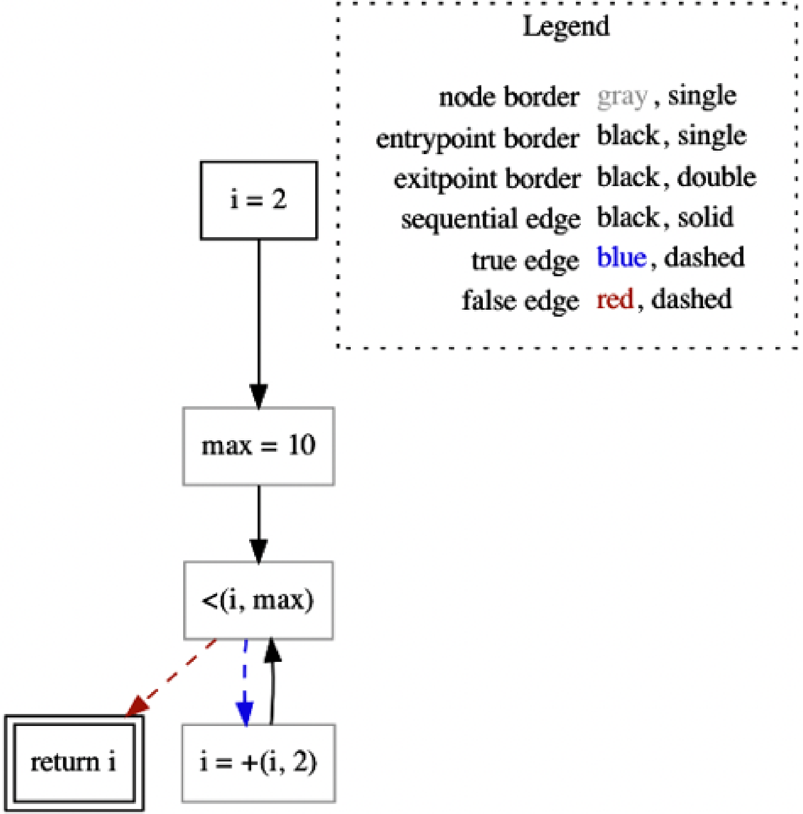
Embedded control flow

Standard fixpoints

□ language dependent

CFGs

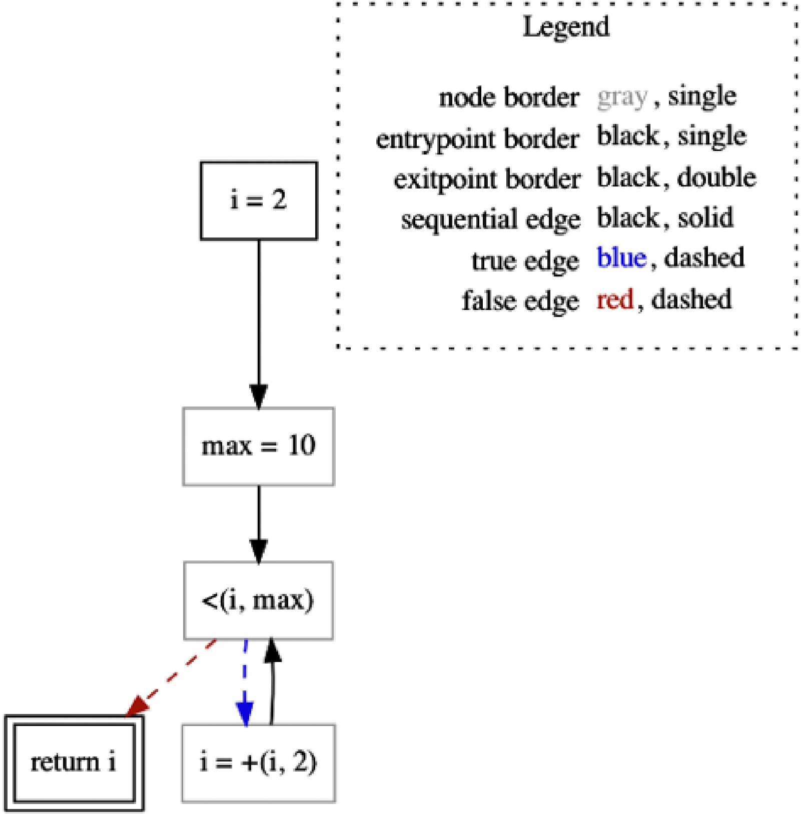
```
int i = 2;
int max = 10;
while (i < max)
  i = i + 1;
return i;
```



CFGs

```
int i = 2;
int max = 10;
while (i < max)
  i = i + 1;
return i;
```

Nodes are `Statement` instances

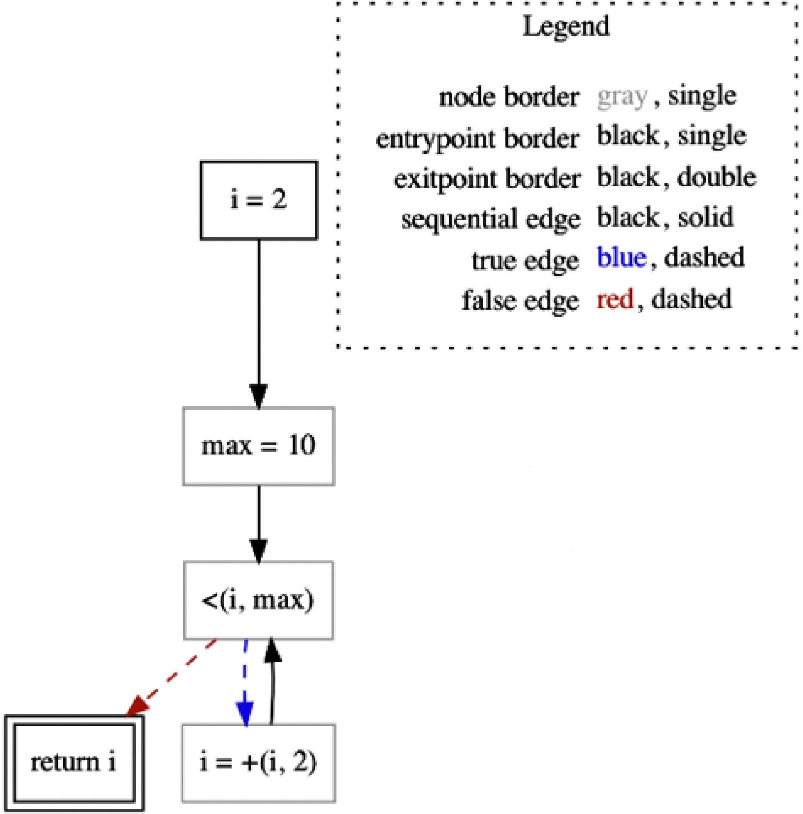

Legend
node border gray , single
entrypoint border black , single
exitpoint border black , double
sequential edge black , solid
true edge blue , dashed
false edge red , dashed
i = 2
max = 10
<(i, max)
return i
i = +(i, 2)

CFGs

```
int i = 2;
int max = 10;
while (i < max)
  i = i + 1;
return i;
```

Nodes are `Statement` instances

Edges are `Edge` instances



Legend

node border	gray, single
entrypoint border	black, single
exitpoint border	black, double
sequential edge	black, solid
true edge	blue, dashed
false edge	red, dashed

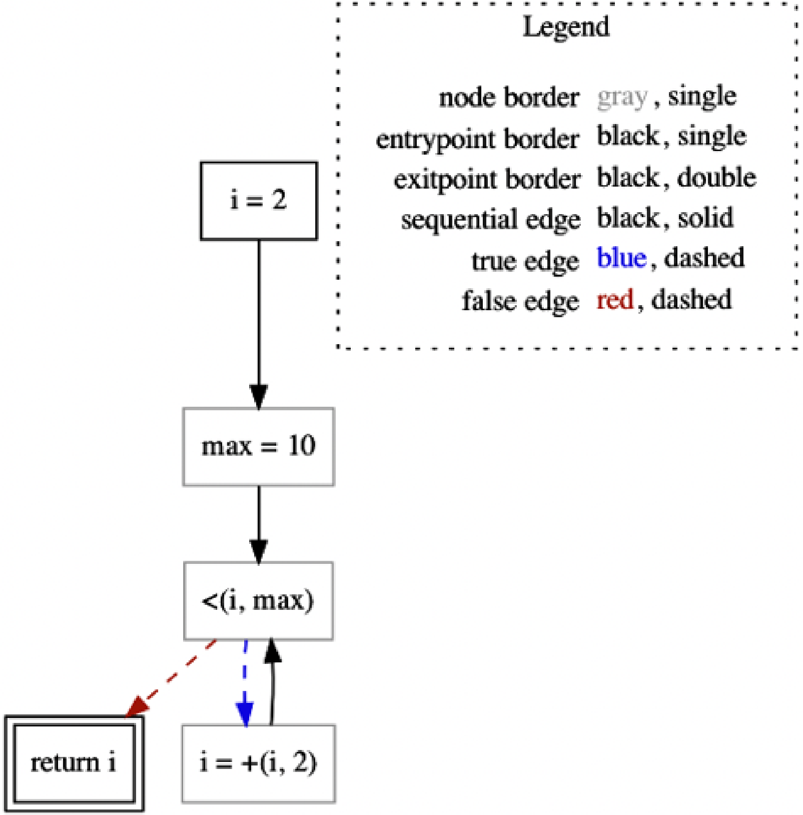
CFGs

```
int i = 2;
int max = 10;
while (i < max)
  i = i + 1;
return i;
```

Nodes are `Statement` instances

Edges are `Edge` instances

LiSA does not fix a semantics for any `Statement`/`Edge`, but lets users define them (more details later)



Computing Fixpoints on CFGs

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;
```

$\bot$ for not yet processed/unreachable

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;
$in_{n_0} \leftarrow \top$;
```

$\bot$ for not yet processed/unreachable

$\top$ for no information

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;                                        $\bot$ for not yet processed/unreachable
$in_{n_0} \leftarrow \top$;                                         $\top$ for no information
$out_{n_0} \leftarrow \texttt{semantics}_{n_0}(in_{n_0})$;          use Statement.semantics()
```

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;
$in_{n_0} \leftarrow \top$;
$out_{n_0} \leftarrow \mathtt{semantics}_{n_0}(in_{n_0})$;
$ws \leftarrow \mathtt{succ}(n_0)$;
```

$\bot$ for not yet processed/unreachable

$\top$ for no information

use `Statement.semantics()`

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;
$in_{n_0} \leftarrow \top$;
$out_{n_0} \leftarrow \texttt{semantics}_{n_0}(in_{n_0})$;
$ws \leftarrow \texttt{succ}(n_0)$;
while $ws \neq \emptyset$ do
    $n \leftarrow \texttt{pop}(ws)$;
```

$\bot$ for not yet processed/unreachable

$\top$ for no information

use `Statement.semantics()`

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;
$in_{n_0} \leftarrow \top$;
$out_{n_0} \leftarrow \mathtt{semantics}_{n_0}(in_{n_0})$;
$ws \leftarrow \mathtt{succ}(n_0)$;
while $ws \neq \emptyset$ do
    $n \leftarrow \mathtt{pop}(ws)$;
    $in_n \leftarrow \bigsqcup\{\mathtt{traverse}_{m \to n}(out_m) :$
                  $m \in \mathtt{preds}(n)\}$;
    $tmp_n \leftarrow \mathtt{semantics}_n(in_n)$;
```

$\bot$ for not yet processed/unreachable

$\top$ for no information

use `Statement.semantics()`

$\sqcup$ to over-approximate entry states

use `Edge.traverse()`

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;
$in_{n_0} \leftarrow \top$;
$out_{n_0} \leftarrow \texttt{semantics}_{n_0}(in_{n_0})$;
$ws \leftarrow \texttt{succ}(n_0)$;
while $ws \neq \emptyset$ do
    $n \leftarrow \texttt{pop}(ws)$;
    $in_n \leftarrow \bigsqcup\{\texttt{traverse}_{m \to n}(out_m) :$
                    $m \in \texttt{preds}(n)\}$;
    $tmp_n \leftarrow \texttt{semantics}_n(in_n)$;
    if $tmp_n \not\sqsubseteq out_n$ then
```

$\bot$ for not yet processed/unreachable

$\top$ for no information

use `Statement.semantics()`

$\sqcup$ to over-approximate entry states

use `Edge.traverse()`

$\sqsubseteq$ to keep “highest” result

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;
$in_{n_0} \leftarrow \top$;
$out_{n_0} \leftarrow \texttt{semantics}_{n_0}(in_{n_0})$;
$ws \leftarrow \texttt{succ}(n_0)$;
while $ws \neq \emptyset$ do
    $n \leftarrow \texttt{pop}(ws)$;
    $in_n \leftarrow \bigsqcup\{\texttt{traverse}_{m \to n}(out_m)$ :
                $m \in \texttt{preds}(n)\}$;
    $tmp_n \leftarrow \texttt{semantics}_n(in_n)$;
    if $tmp_n \not\sqsubseteq out_n$ then
        $out_n \leftarrow out_n \oplus tmp_n$;
        $ws \leftarrow ws \cup \texttt{succ}(n)$;
```

$\bot$ for not yet processed/unreachable

$\top$ for no information

use `Statement.semantics()`

$\sqcup$ to over-approximate entry states

use `Edge.traverse()`

$\sqsubseteq$ to keep “highest” result

$\sqcup$ to move “upwards”/∇ for convergence

Computing Fixpoints on CFGs

```
forall $n \in N$ do
    $out_n \leftarrow \bot$;
$in_{n_0} \leftarrow \top$;
$out_{n_0} \leftarrow \mathtt{semantics}_{n_0}(in_{n_0})$;
$ws \leftarrow \mathtt{succ}(n_0)$;
while $ws \neq \emptyset$ do
    $n \leftarrow \mathtt{pop}(ws)$;
    $in_n \leftarrow \bigsqcup\{\mathtt{traverse}_{m \to n}(out_m)$ :
                $m \in \mathtt{preds}(n)\}$;
    $tmp_n \leftarrow \mathtt{semantics}_n(in_n)$;
    if $tmp_n \not\sqsubseteq out_n$ then
        $out_n \leftarrow out_n \oplus tmp_n$;
        $ws \leftarrow ws \cup \mathtt{succ}(n)$;
return $out$;
```

$\bot$ for not yet processed/unreachable

$\top$ for no information

use `Statement.semantics()`

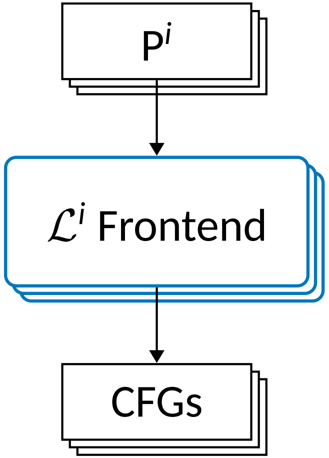
$\sqcup$ to over-approximate entry states

use `Edge.traverse()`

$\sqsubseteq$ to keep “highest” result

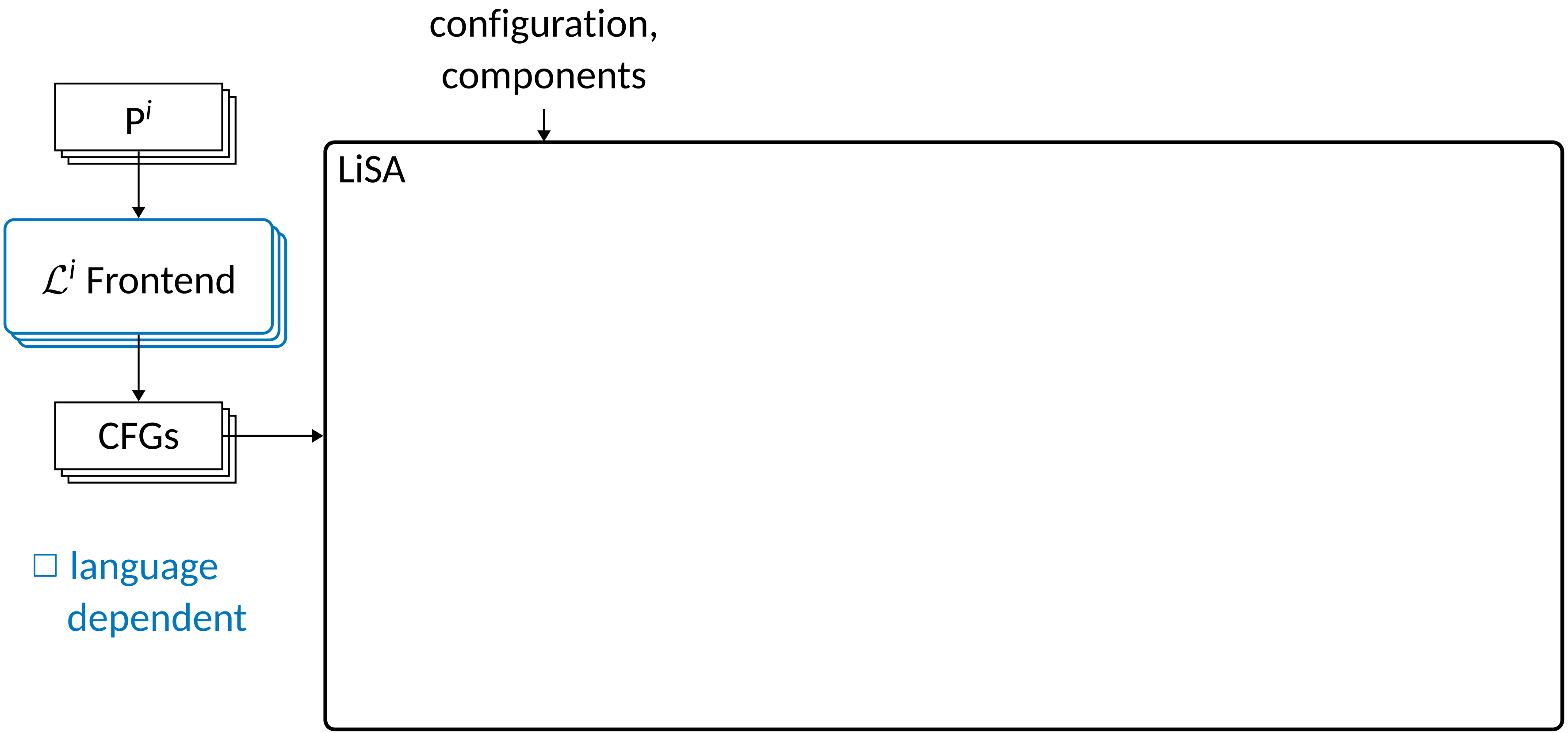
$\sqcup$ to move “upwards”/∇ for convergence

LiSA Overview



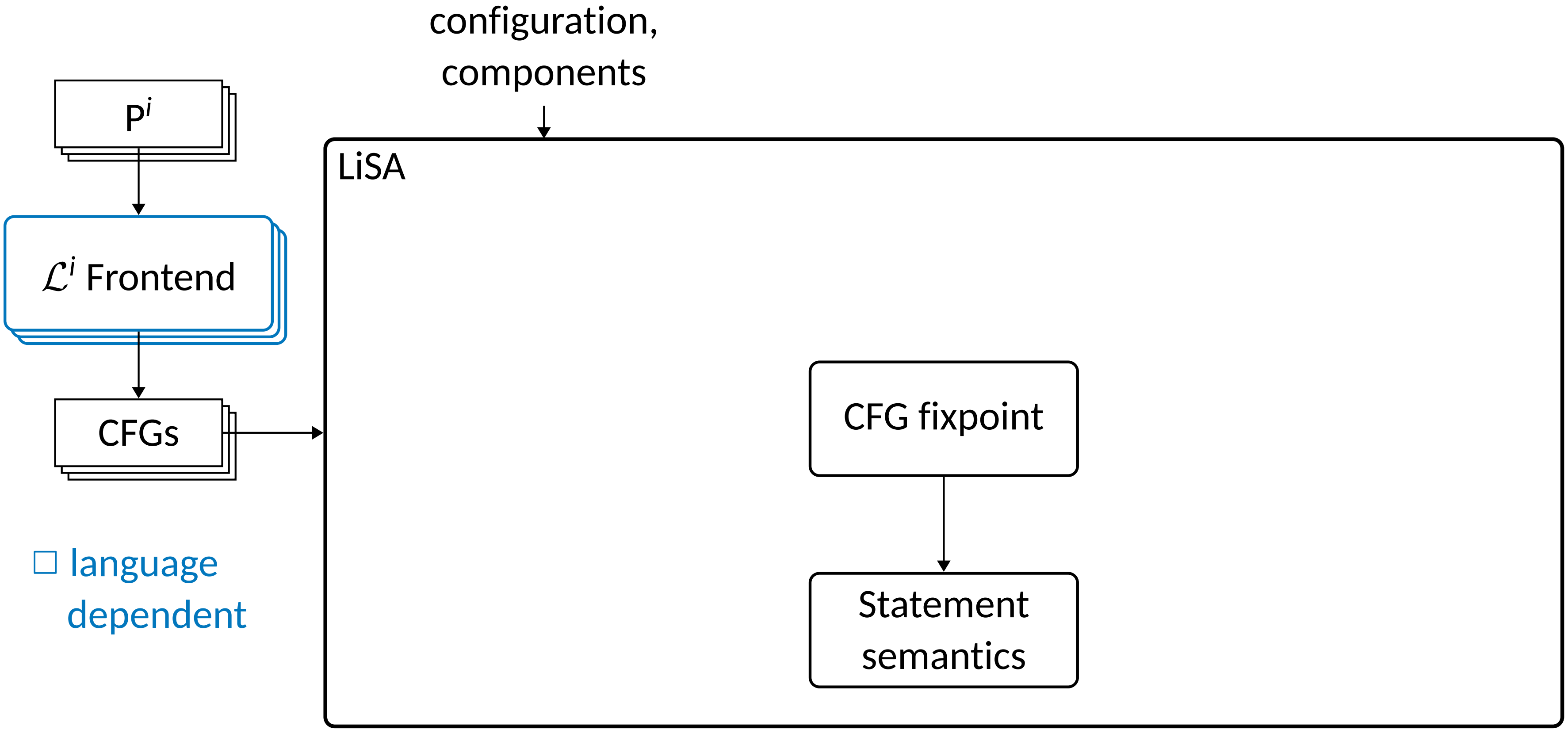
□ language dependent

LiSA Overview



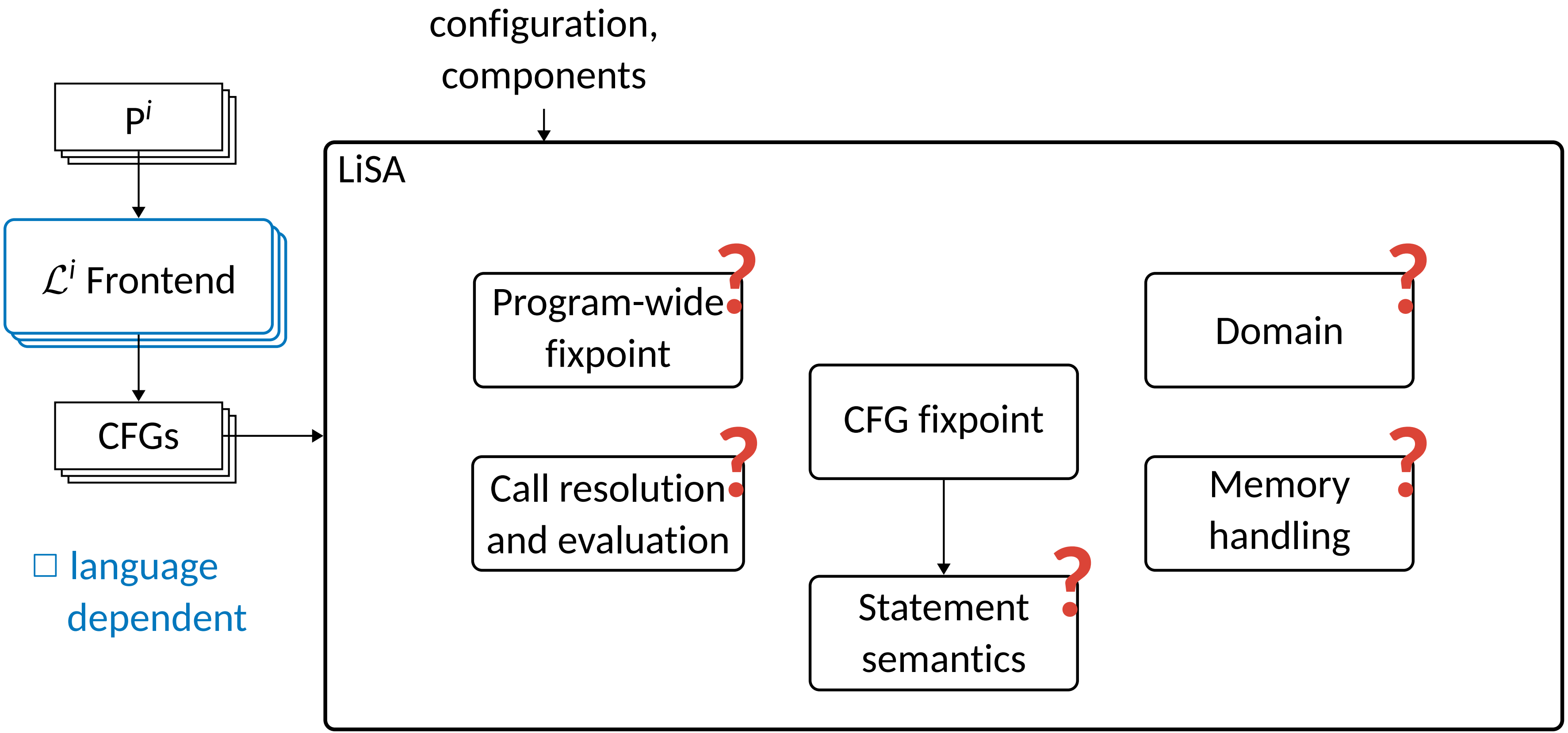
□ language dependent

LiSA Overview



□ language dependent

LiSA Overview



Today's Plan

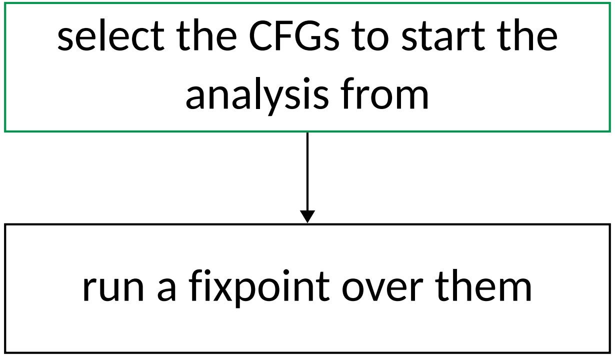
1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA's High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

The Program-wide Fixpoint

The Program-wide Fixpoint

select the CFGs to start the analysis from

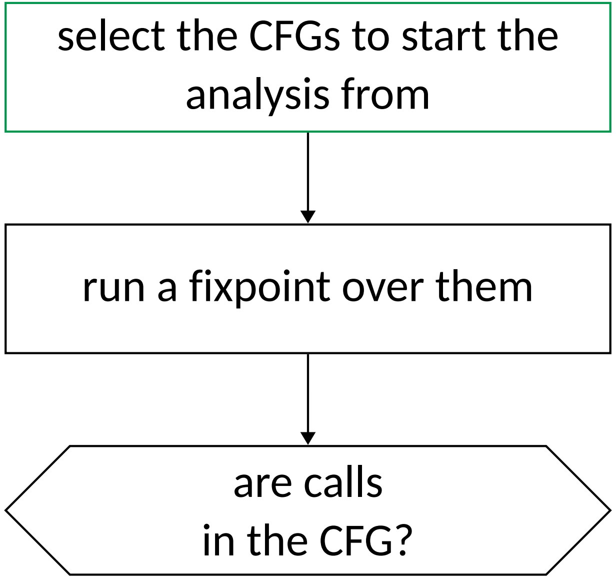
The Program-wide Fixpoint



select the CFGs to start the analysis from

run a fixpoint over them

The Program-wide Fixpoint

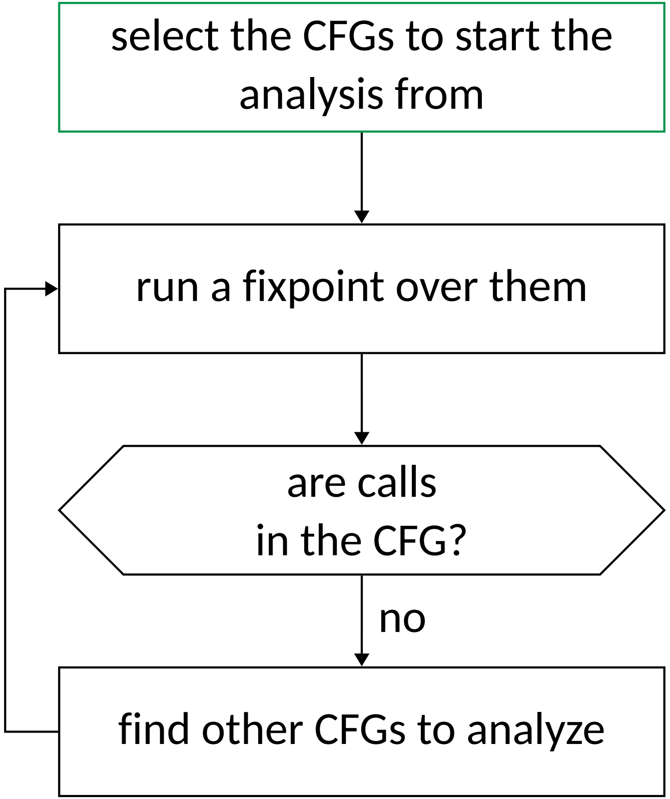


select the CFGs to start the analysis from

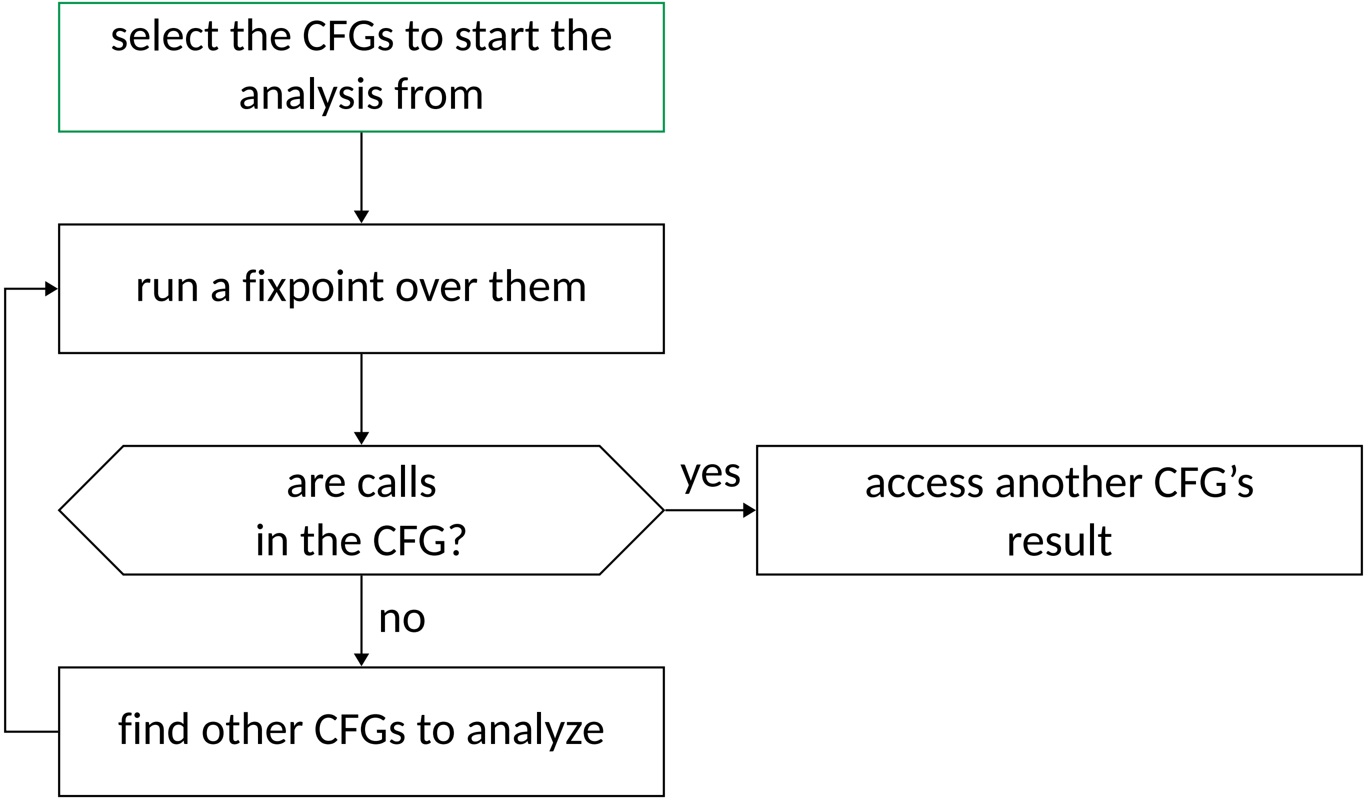
run a fixpoint over them

are calls in the CFG?

The Program-wide Fixpoint


select the CFGs to start the analysis from
run a fixpoint over them
are calls in the CFG?
no
find other CFGs to analyze

The Program-wide Fixpoint


select the CFGs to start the analysis from

run a fixpoint over them

are calls in the CFG?

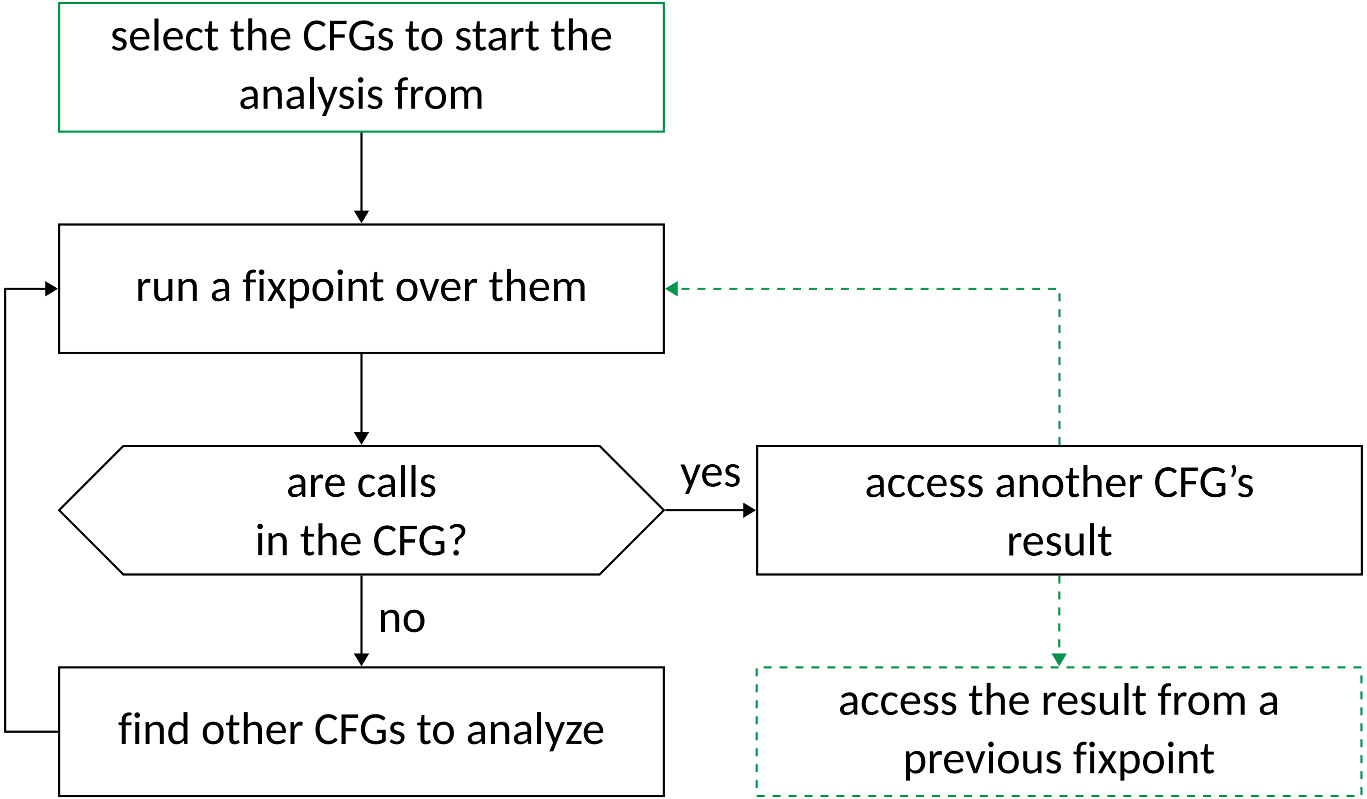
yes

access another CFG’s result

no

find other CFGs to analyze

The Program-wide Fixpoint



select the CFGs to start the analysis from

run a fixpoint over them

are calls in the CFG?

yes

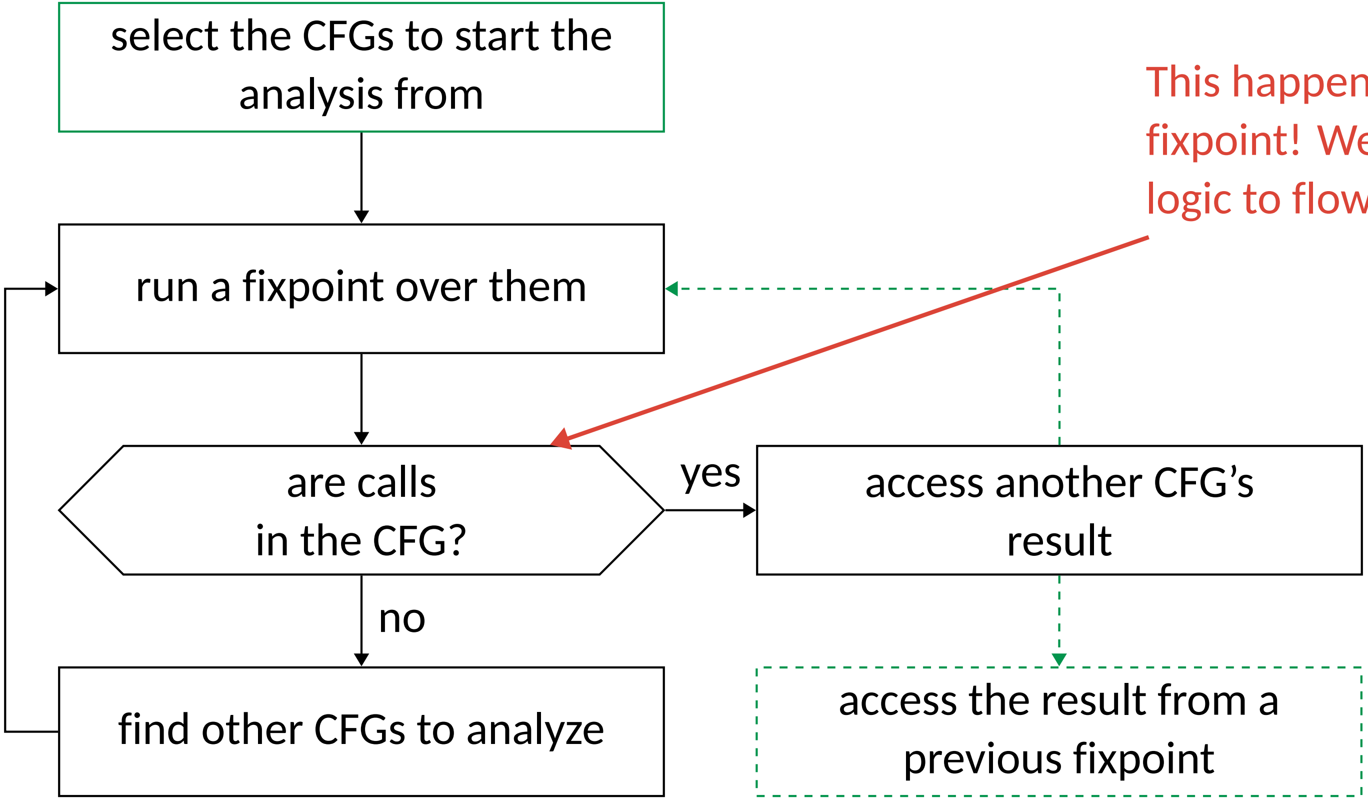
access another CFG’s result

no

find other CFGs to analyze

access the result from a previous fixpoint

The Program-wide Fixpoint



select the CFGs to start the analysis from

run a fixpoint over them

are calls in the CFG?

yes

access another CFG's result

no

find other CFGs to analyze

access the result from a previous fixpoint

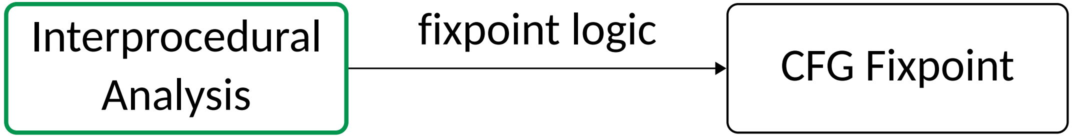
This happens inside the CFG fixpoint! We do not want this logic to flow over there

The Interprocedural Analysis

Interprocedural
Analysis

□ configurable

The Interprocedural Analysis

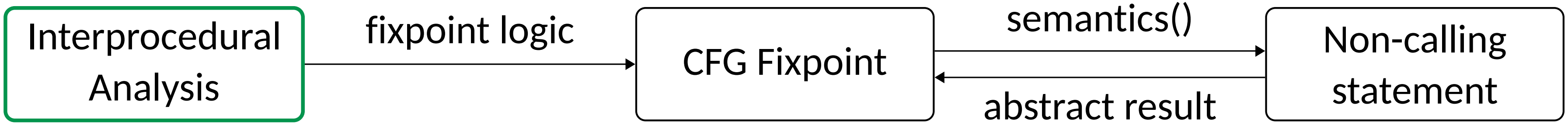


Logic can be:

- Consider CFGs in isolation
- Start at main and follow calls
- …

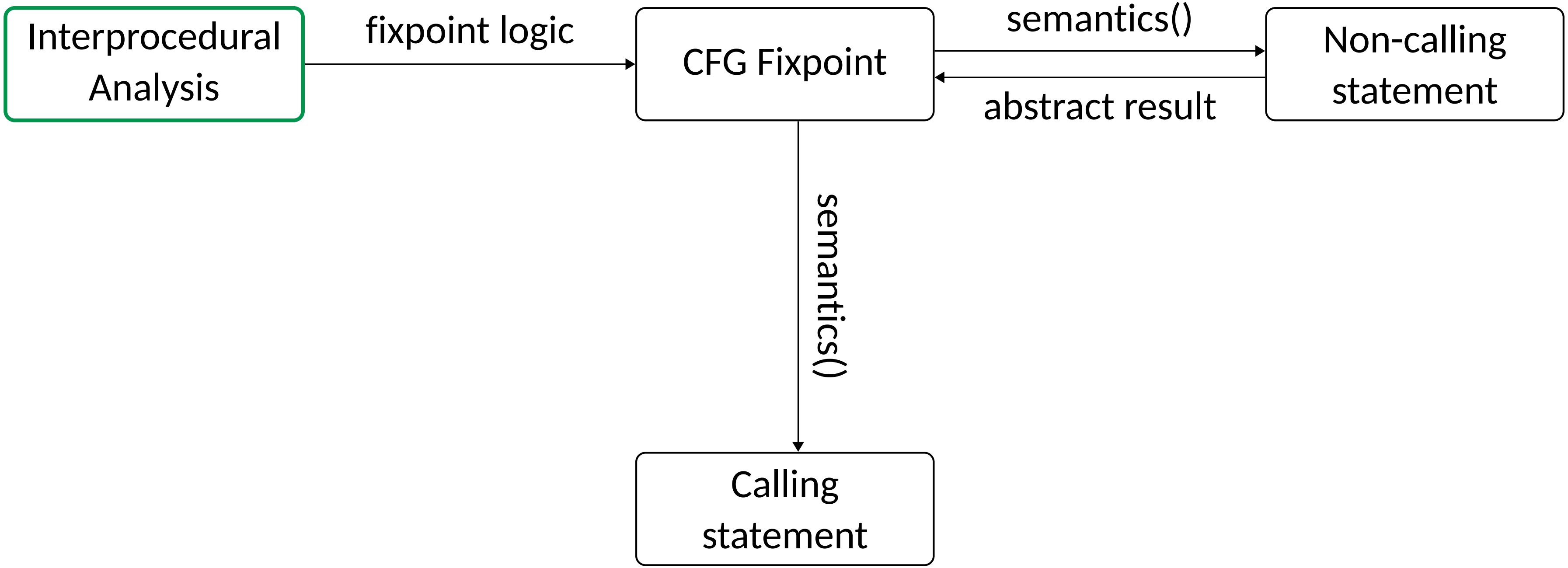
□ configurable

The Interprocedural Analysis



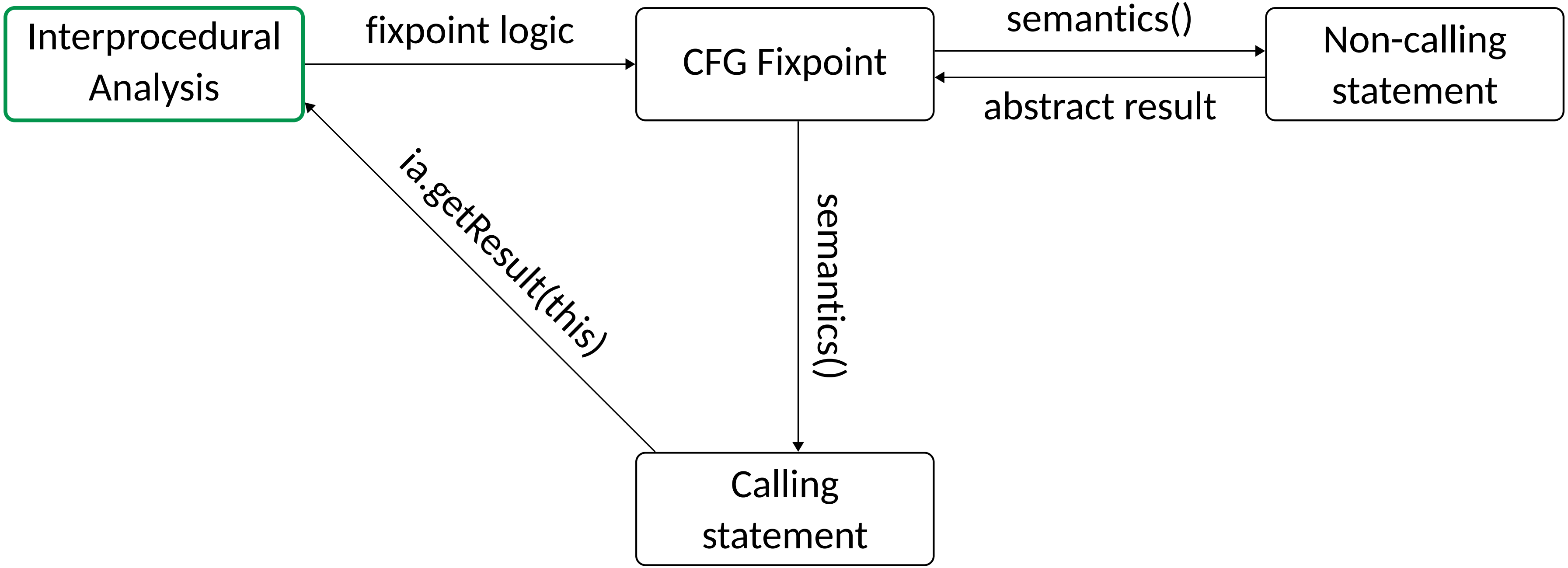
□ configurable

The Interprocedural Analysis



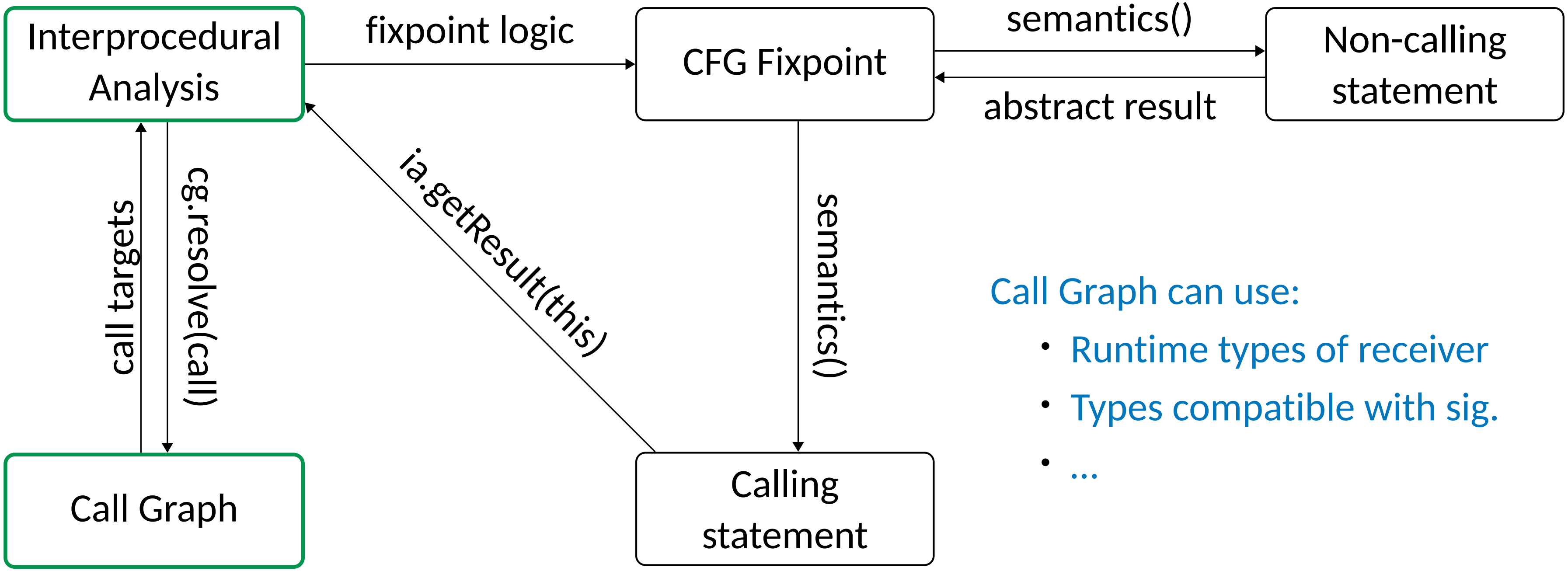
□ configurable

The Interprocedural Analysis



□ configurable

The Interprocedural Analysis

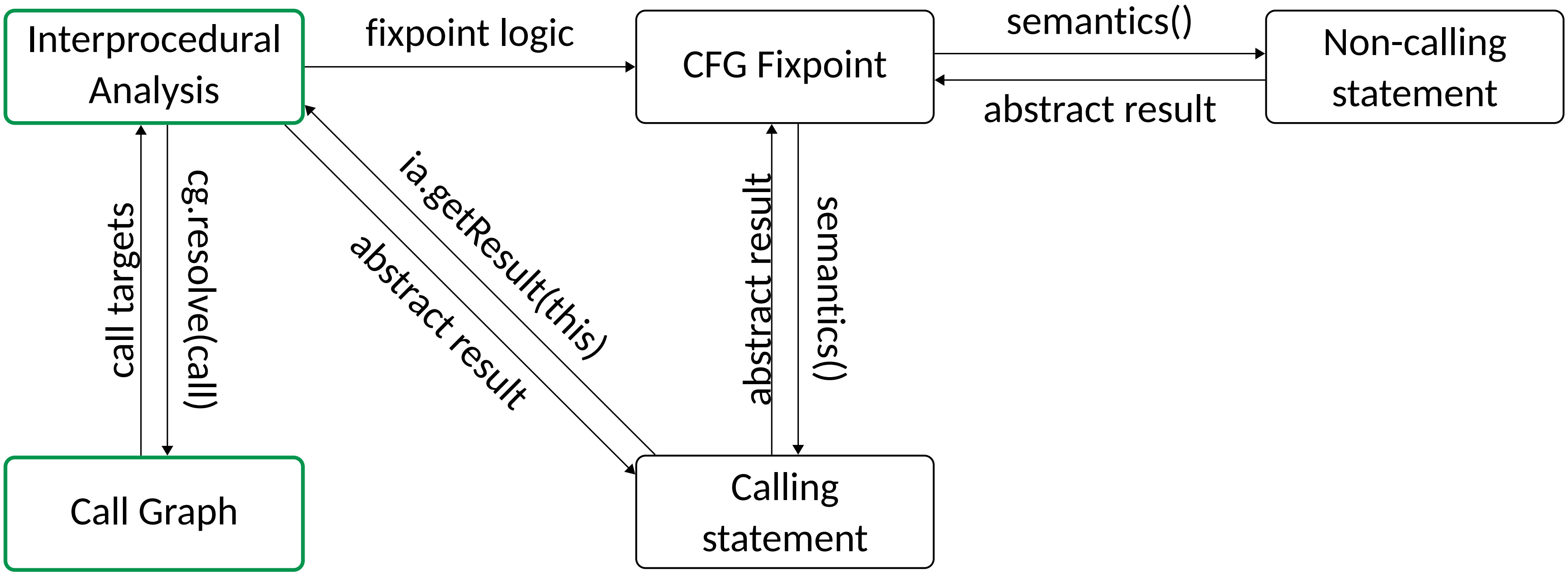


Call Graph can use:

- Runtime types of receiver
- Types compatible with sig.
- …

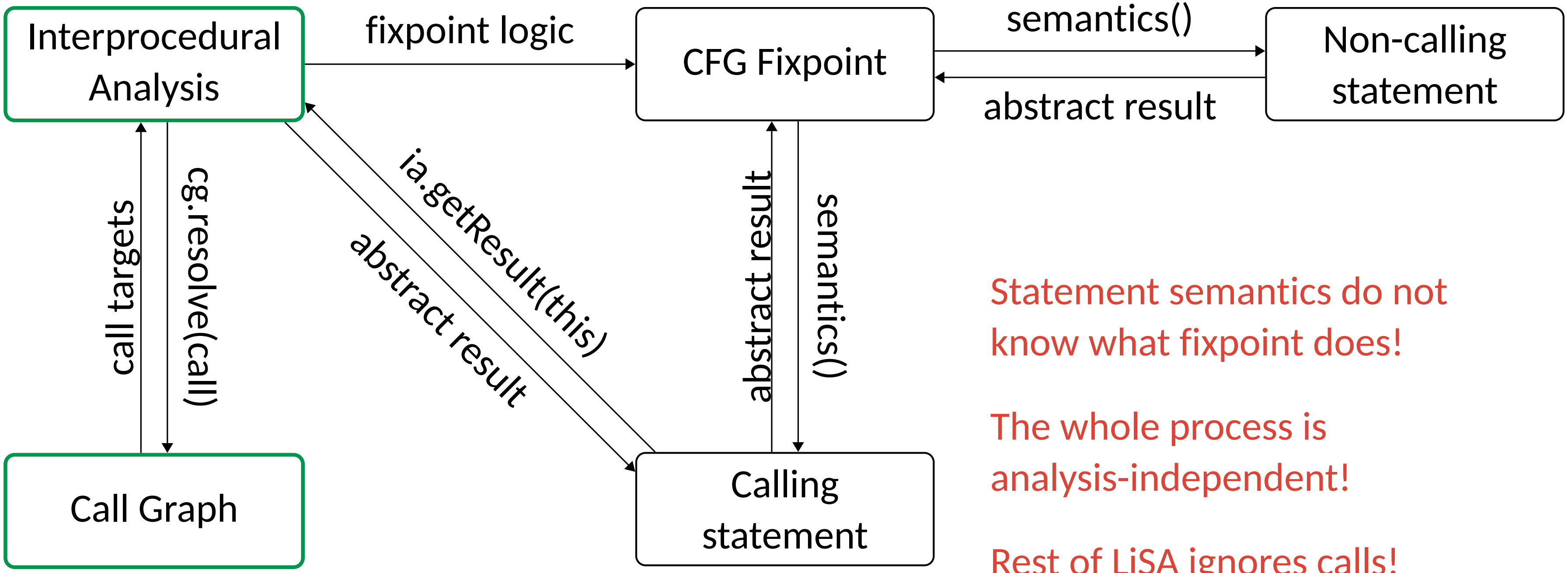
□ configurable

The Interprocedural Analysis



□ configurable

The Interprocedural Analysis



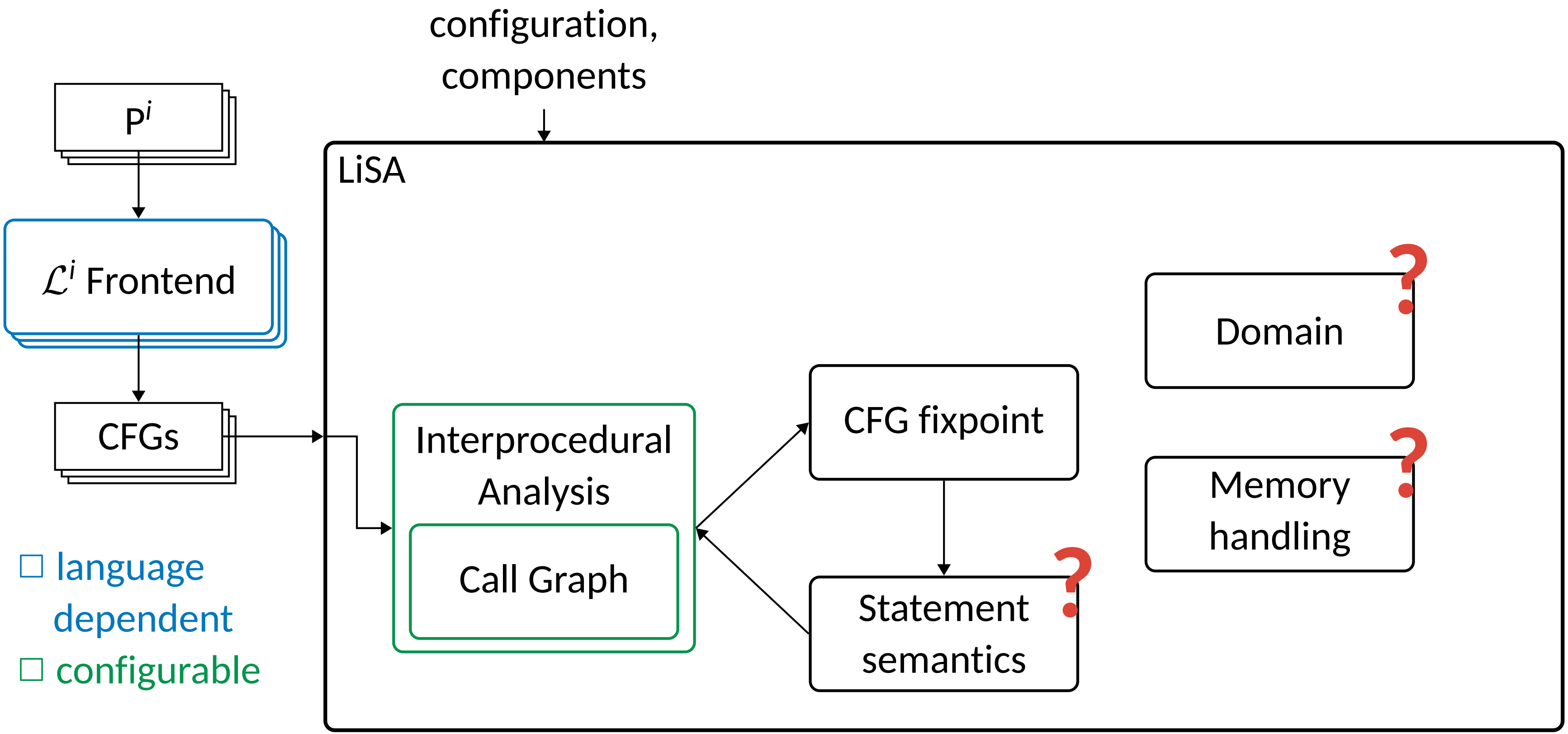
Statement semantics do not know what fixpoint does!

The whole process is analysis-independent!

Rest of LiSA ignores calls!

□ configurable

LiSA Overview



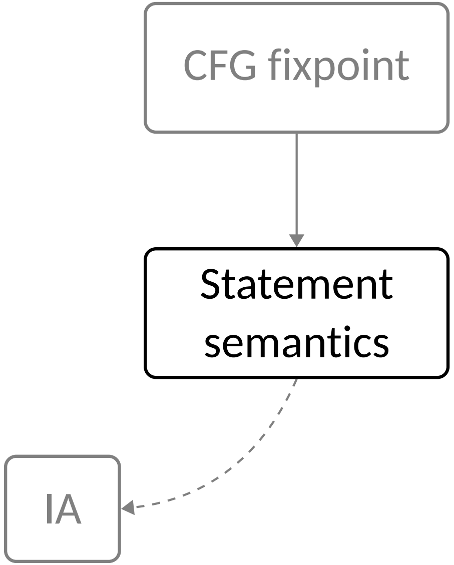
□ language dependent

□ configurable

Today’s Plan

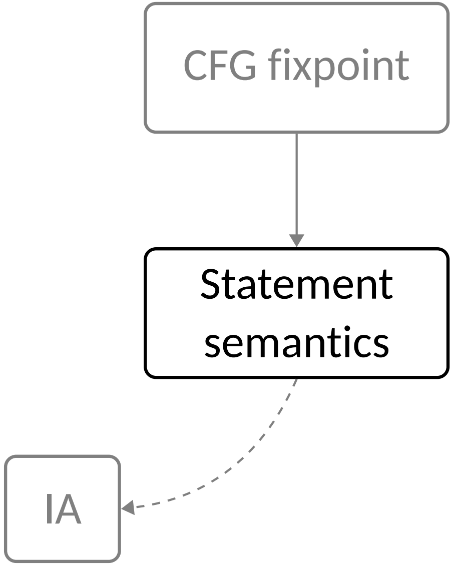
1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

Syntax vs Semantics



We want to specify semantics **generically**

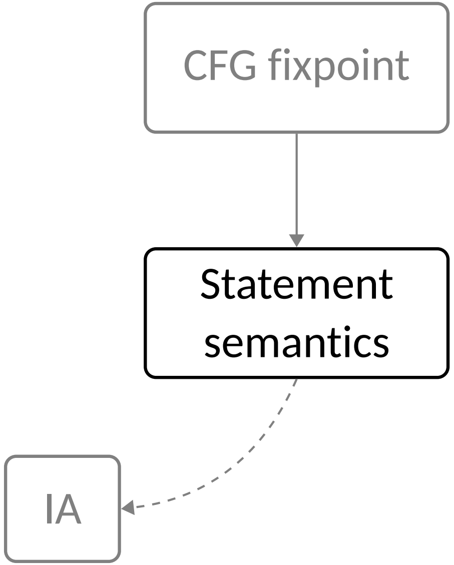
Syntax vs Semantics



We want to specify semantics **generically**

- It should apply to all analyses

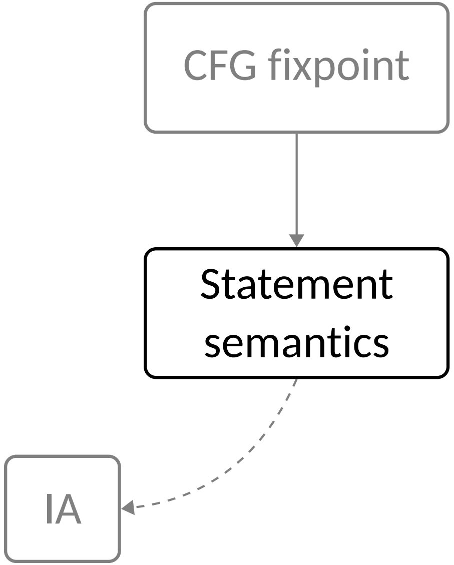
Syntax vs Semantics



We want to specify semantics **generically**

- It should apply to all analyses
- Analyses should not give meaning to the syntax

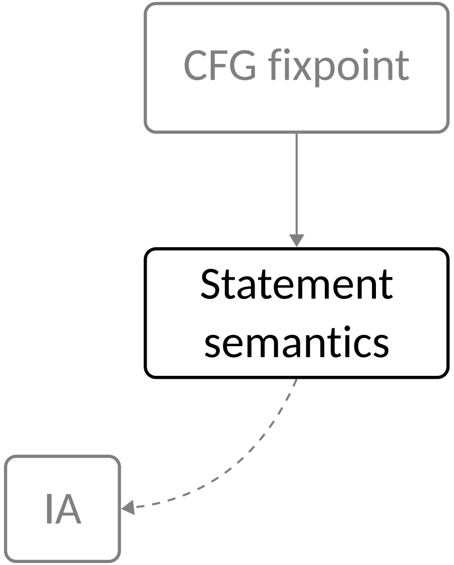
Syntax vs Semantics



We want to specify semantics **generically**

- It should apply to all analyses
- Analyses should not give meaning to the syntax
  - Java's `a + b` can be a sum or a concatenation

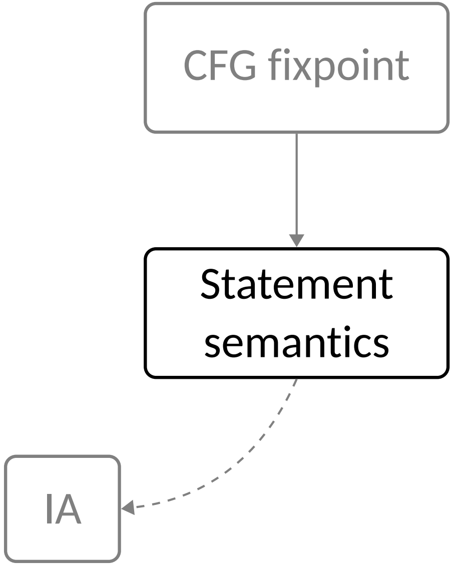
Syntax vs Semantics



We want to specify semantics **generically**

- It should apply to all analyses
- Analyses should not give meaning to the syntax
  - Java's `a + b` can be a sum or a concatenation
  - Python's `[f(x) for x in dict.values()]` expands to a loop

Syntax vs Semantics

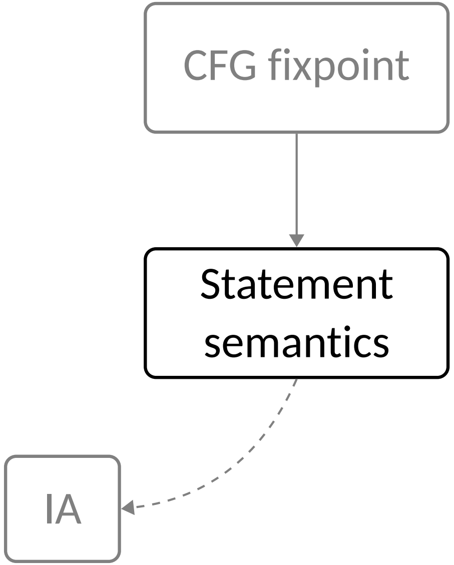


We want to specify semantics **generically**

- It should apply to all analyses
- Analyses should not give meaning to the syntax
  - Java’s `a + b` can be a sum or a concatenation
  - Python’s `[f(x) for x in dict.values()]` expands to a loop

Classic solution: rewrite to an IR

Syntax vs Semantics



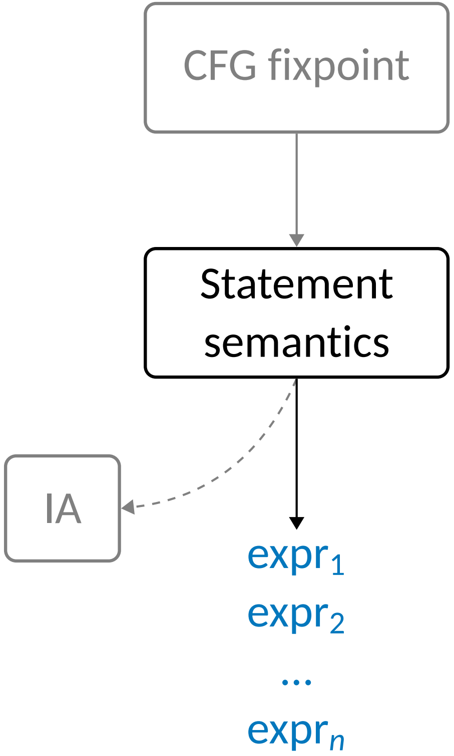
We want to specify semantics **generically**

- It should apply to all analyses
- Analyses should not give meaning to the syntax
  - Java's `a + b` can be a sum or a concatenation
  - Python's `[f(x) for x in dict.values()]` expands to a loop

Classic solution: rewrite to an IR

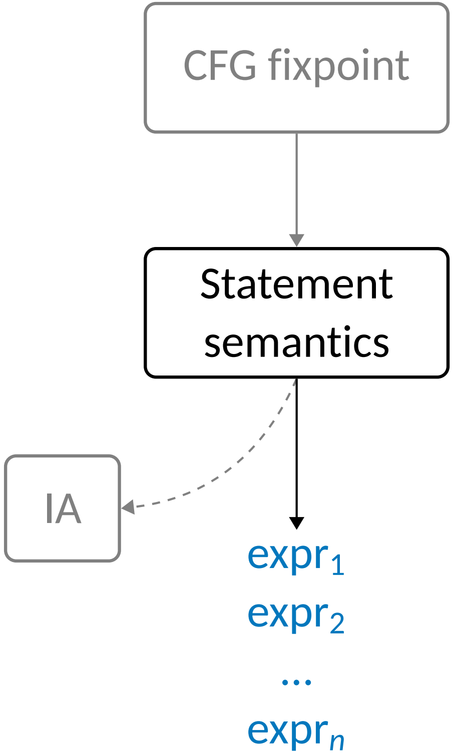
But we might want to exploit semantic information!

Symbolic Expressions



Our solution: break `Statements` into `SymbolicExpression`s

Symbolic Expressions

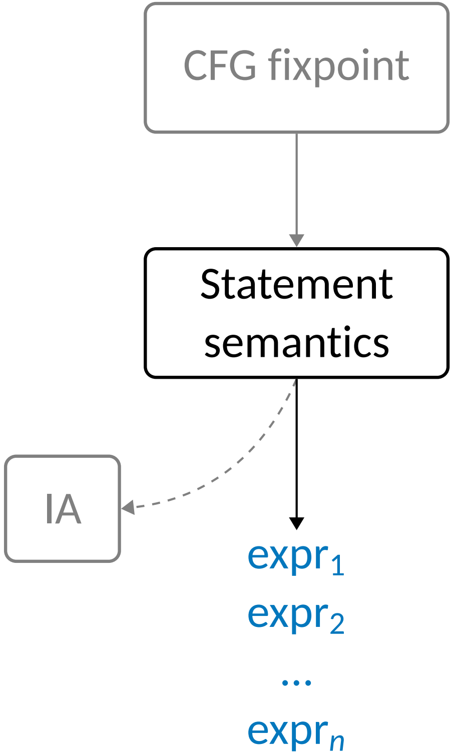


Our solution: break `Statements` into `SymbolicExpressions`

The rewriting happens dynamically, and can exploit the analysis

- Types, possible values, …

Symbolic Expressions



Our solution: break `Statements` into `SymbolicExpressions`

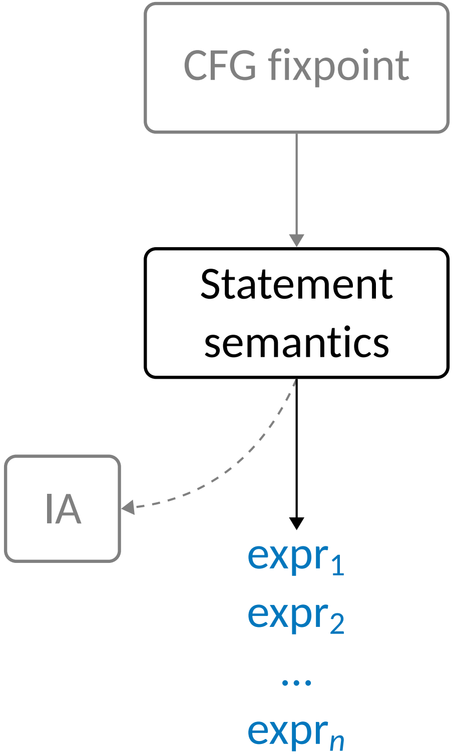
The rewriting happens dynamically, and can exploit the analysis

- Types, possible values, …

`SymbolicExpressions` are extensible

- We provide some, but users can define more

Symbolic Expressions



Our solution: break `Statements` into `SymbolicExpressions`

The rewriting happens dynamically, and can exploit the analysis

- Types, possible values, …

`SymbolicExpressions` are extensible

- We provide some, but users can define more

Analyses only have to model these expressions, ignoring the syntax behind them!

Examples (Pseudocode)

```
Plus.semantics(): // left + right
  if type(left) is string or type(right) is string:
    return domain.eval(new BinaryExpression("cat", left, right))
  else:
    return domain.eval(new BinaryExpression("+", left, right))
```

Examples (Pseudocode)

```
Plus.semantics(): // left + right
  if type(left) is string or type(right) is string:
    return domain.eval(new BinaryExpression("cat", left, right))
  else:
    return domain.eval(new BinaryExpression("+", left, right))
```

```
Conditional.semantics(): // condition ? ifTrue : ifFalse
  sat = domain.isSatisfied(condition)
  tt = ifTrue.semantics()
  ff = ifFalse.semantics()
  return sat.isTrue() ? tt : (sat.isFalse() ? ff : tt ⊔ ff)
```

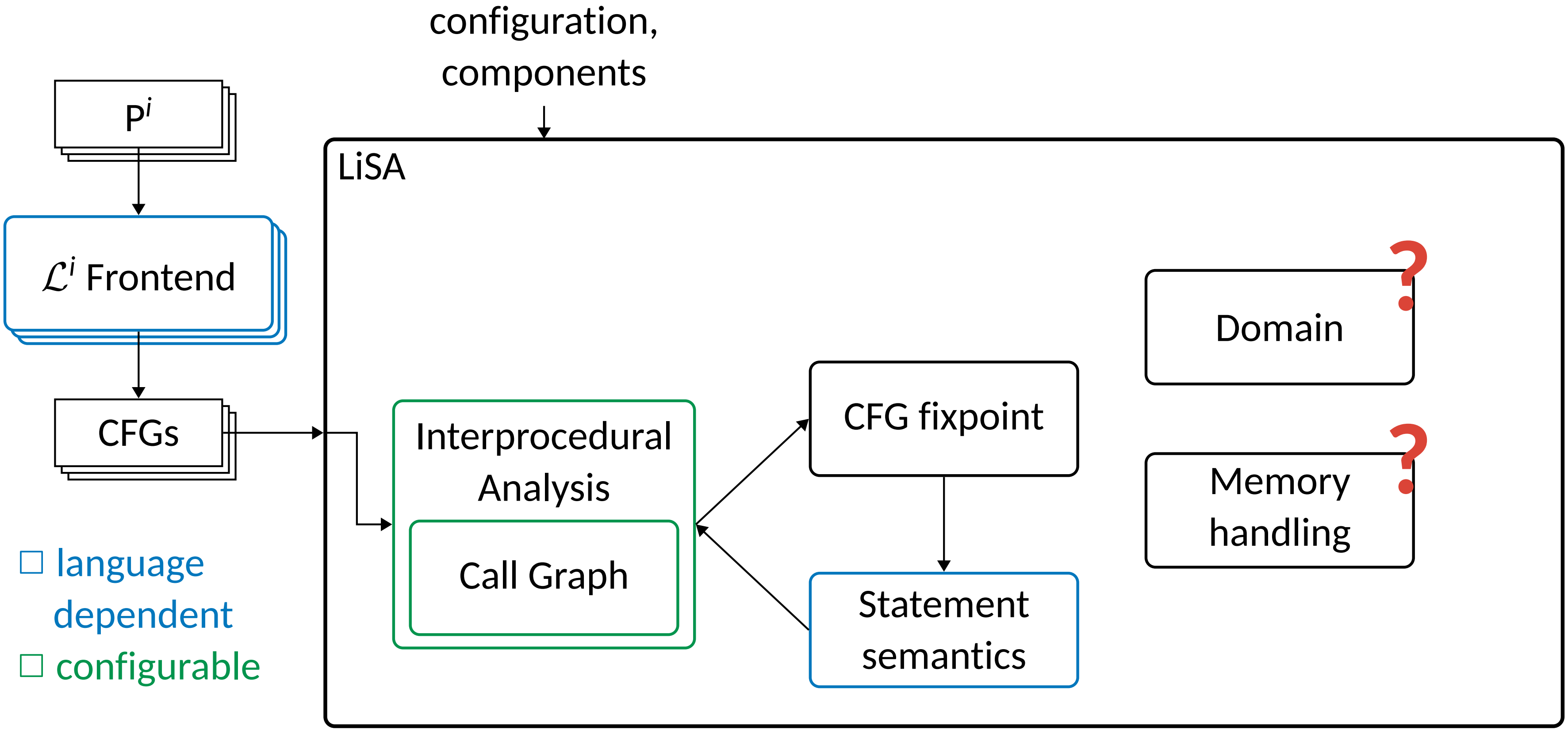
Examples (Pseudocode)

```
Plus.semantics(): // left + right
  if type(left) is string or type(right) is string:
    return domain.eval(new BinaryExpression("cat", left, right))
  else:
    return domain.eval(new BinaryExpression("+", left, right))
```

```
Conditional.semantics(): // condition ? ifTrue : ifFalse
  sat = domain.isSatisfied(condition)
  tt = ifTrue.semantics()
  ff = ifFalse.semantics()
  return sat.isTrue() ? tt : (sat.isFalse() ? ff : tt ⊔ ff)
```

```
PreIncrement.semantics(): // ++var
  add = new BinaryExpression("+", var, new Const(1))
  d1 = domain.store(var, add)
  return d1.eval(var)
```

LiSA Overview



Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

Combining Values and Memory

While calls are “easy” to abstract away, dynamic memory (stack or heap) and computed values are strongly coupled

- Values are moved from variables to dynamic memory continuously

Combining Values and Memory

While calls are “easy” to abstract away, dynamic memory (stack or heap) and computed values are strongly coupled

- Values are moved from variables to dynamic memory continuously

Despite this, we still want to keep management of the two **separate**

- For reusability
- To reduce the complexity of the implementations
- To allow newcomers to write only the analyses they need

What Happens Concretely?

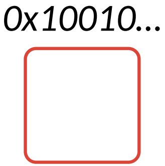
Every memory-dealing instruction “refers” to an address in memory

```
class Foo {
  int f, g;
  Foo() {
    f = 1;
    g = 2;
  }
}
```

What Happens Concretely?

Every memory-dealing instruction “refers” to an address in memory

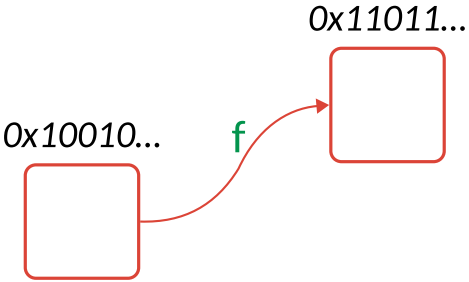
```java
class Foo {
  int f, g;
  Foo() {
    f = 1;
    g = 2;
  }
}
```



What Happens Concretely?

Every memory-dealing instruction “refers” to an address in memory

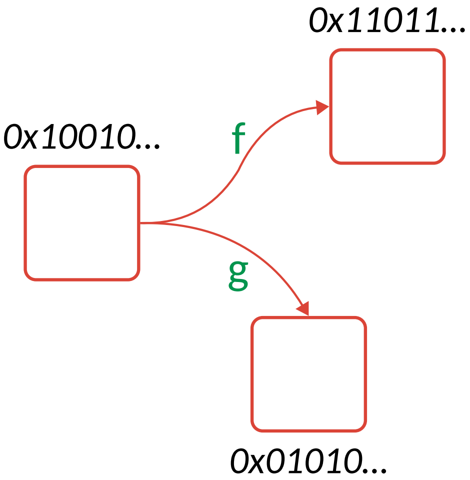
```
class Foo {
  int f, g;
  Foo() {
    f = 1;
    g = 2;
  }
}
```



What Happens Concretely?

Every memory-dealing instruction “refers” to an address in memory

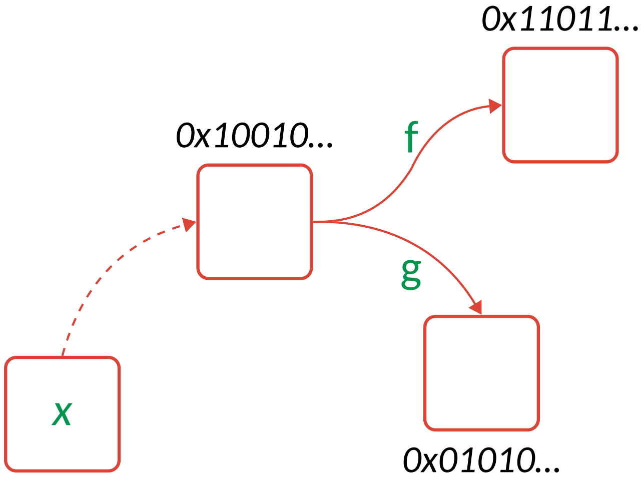
```
class Foo {
  int f, g;
  Foo() {
    f = 1;
    g = 2;
  }
}
```



What Happens Concretely?

Every memory-dealing instruction “refers” to an address in memory

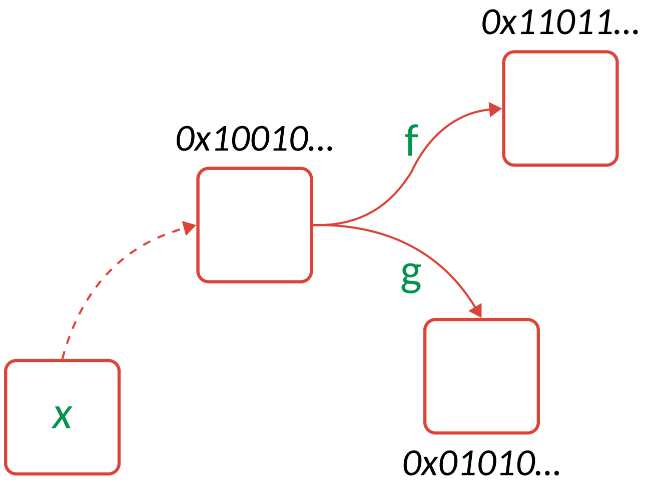
```
class Foo {
  int f, g;
  Foo() {
    f = 1;
    g = 2;
  }
}
```



What Happens Concretely?

Every memory-dealing instruction “refers” to an address in memory

```
class Foo {
  int f, g;
  Foo() {
    f = 1;
    g = 2;
  }
}
```

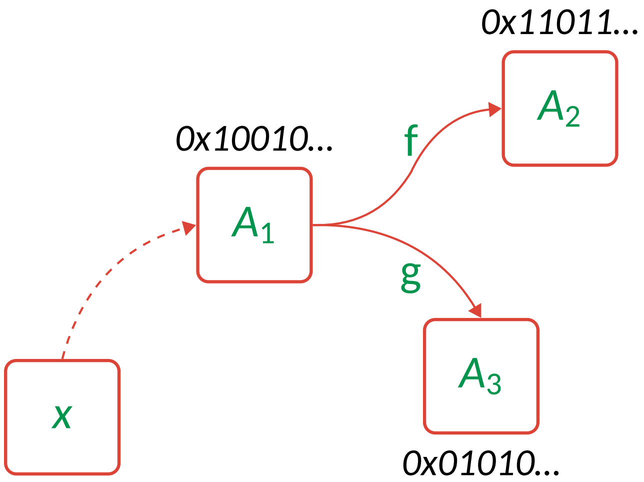


What if we treat addresses as variables?

What Happens Concretely?

Every memory-dealing instruction “refers” to an address in memory

```java
class Foo {
  int f, g;
  Foo() {
    f = 1;
    g = 2;
  }
}
```

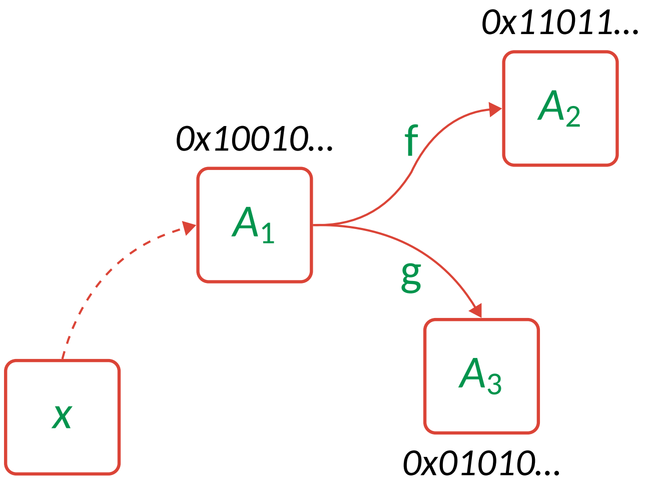


What if we treat addresses as variables?

What Happens Concretely?

Every memory-dealing instruction “refers” to an address in memory

```
class Foo {
  int f, g;
  Foo() {
    f = 1;
    g = 2;
  }
}
```



```
x = new Foo();
x.f = 5;
x.g = 7;
```

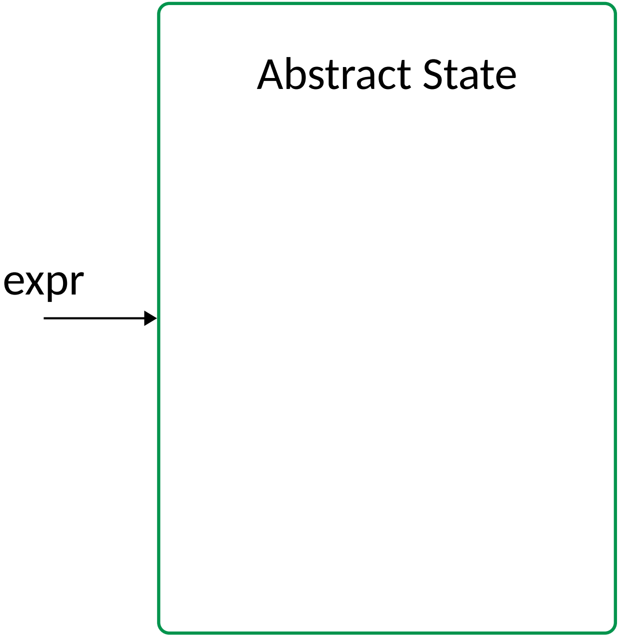
Becomes:

```
x = &A1;
A2 = 5;
A3 = 7;
```

What if we treat addresses as variables?

The Abstract State

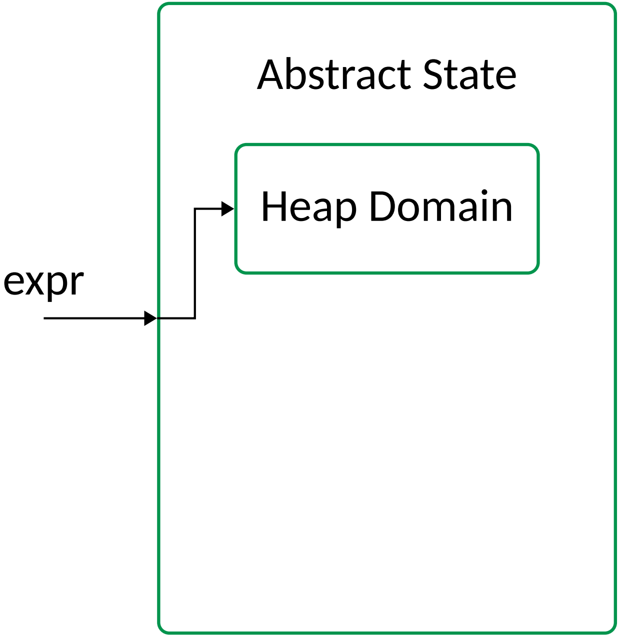
[Fer16]



□ configurable

The Abstract State

[Fer16]



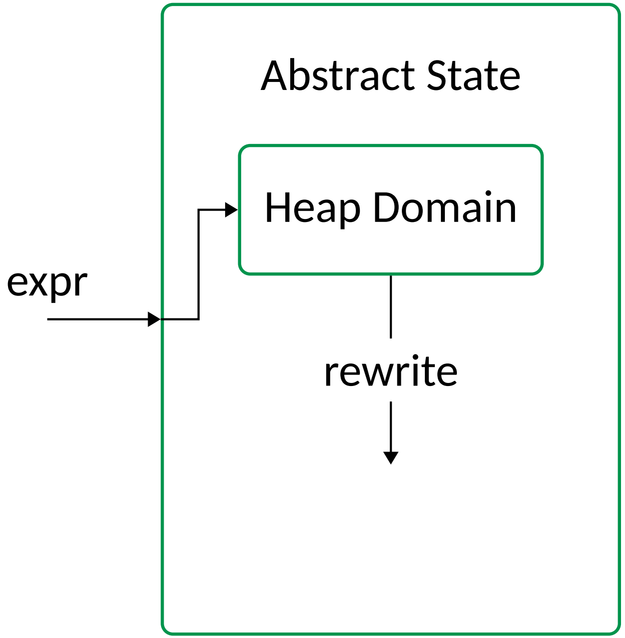
□ configurable

The Heap Domain tracks the **structure of the dynamic memory**

- new allocations
- pointer aliasing
- …

The Abstract State

[Fer16]



The Heap Domain tracks the **structure of the dynamic memory**

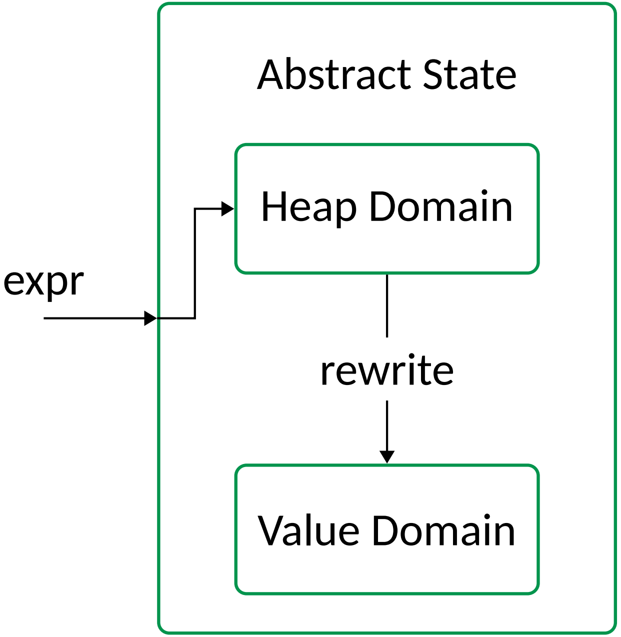
- new allocations
- pointer aliasing
- …

The Heap Domain **rewrites memory-related expressions** to their symbolic variables

□ configurable

The Abstract State

[Fer16]



□ configurable

The Heap Domain tracks the **structure of the dynamic memory**

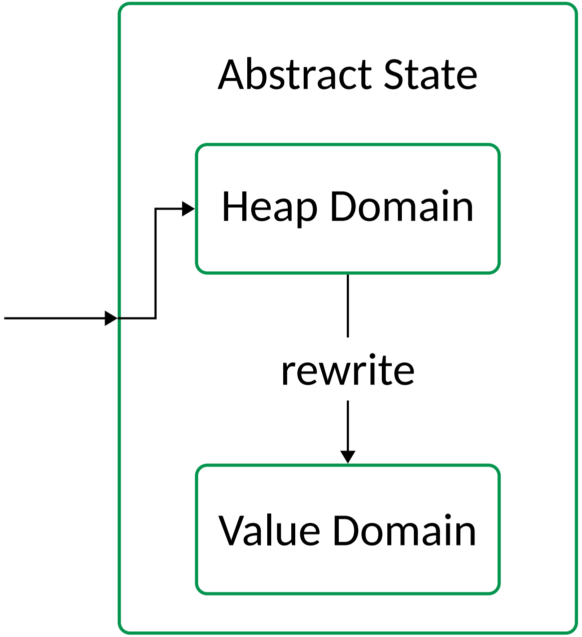
- new allocations
- pointer aliasing
- …

The Heap Domain **rewrites memory-related expressions** to their symbolic variables

The Value Domain only “sees” expressions with **variables and constants**!

The Abstract State

[Fer16]

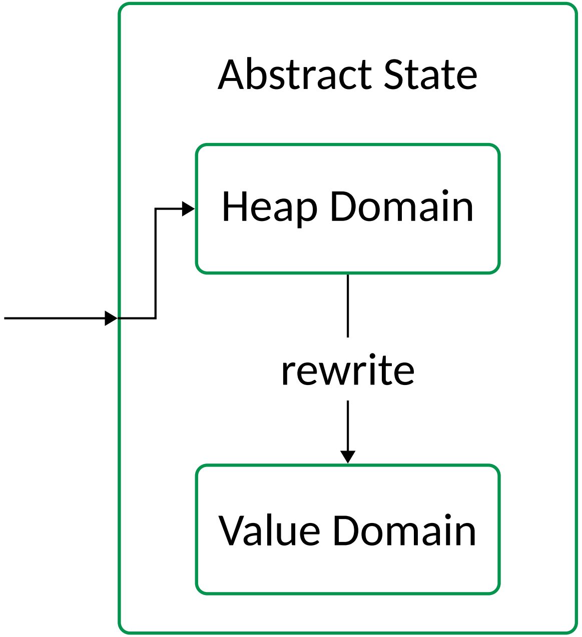


		precision		
expression	**variable**	**low**	**medium**	**high**
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

□ configurable

The Abstract State

[Fer16]

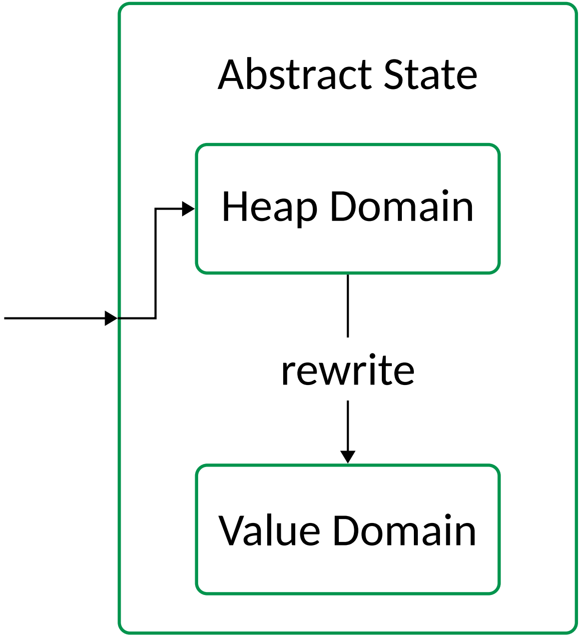


□ configurable

		precision		
expression	**variable**	**low**	**medium**	**high**
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

The Abstract State

[Fer16]

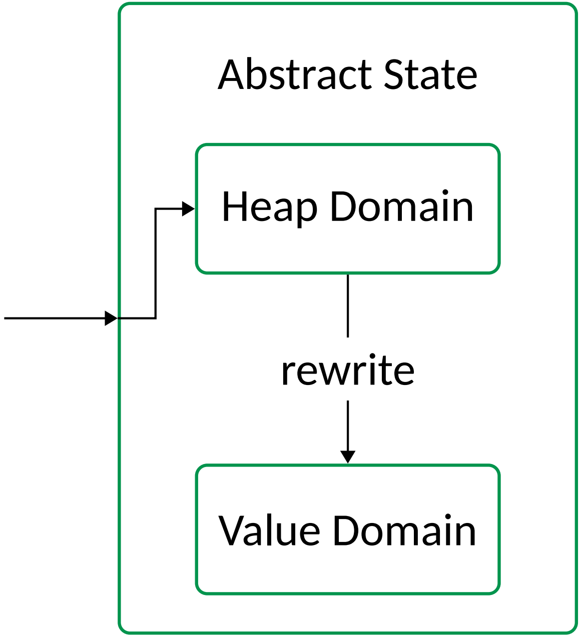


expression	variable	precision: low	precision: medium	precision: high
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

□ configurable

The Abstract State

[Fer16]

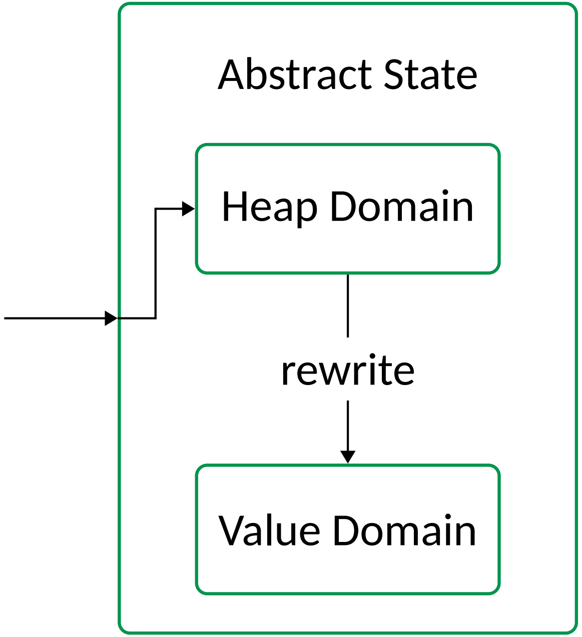


		precision		
expression	**variable**	**low**	**medium**	**high**
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

□ configurable

The Abstract State

[Fer16]



□ configurable

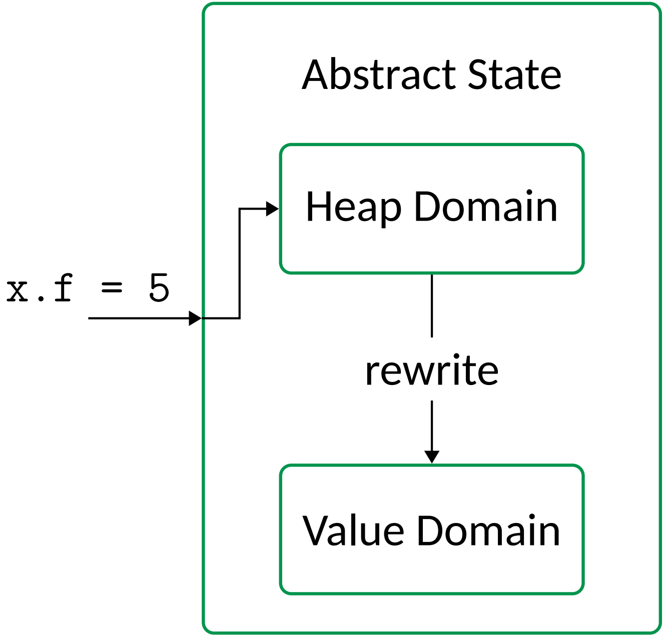
expression	variable	precision low	 medium	 high
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain:

The Abstract State

[Fer16]



		precision		
expression	**variable**	**low**	**medium**	**high**
`new Foo()`	A_1	L	L_1	L_1
→ `x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

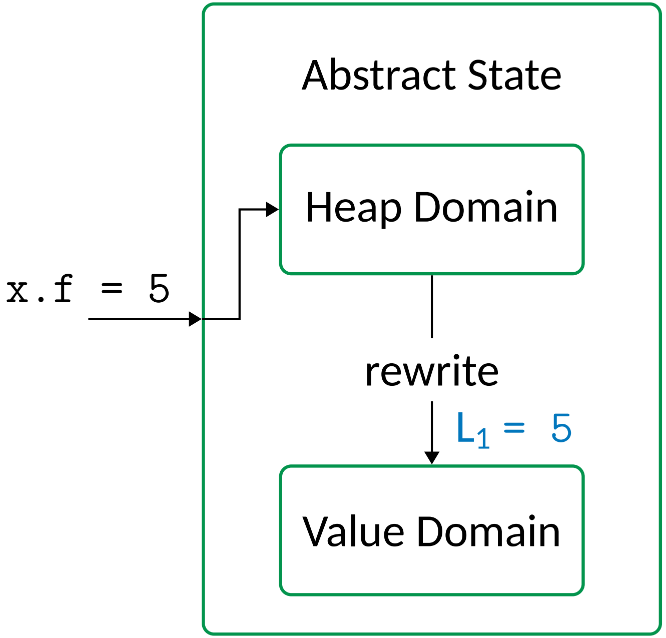
Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain:

□ configurable

The Abstract State

[Fer16]



		precision		
expression	**variable**	**low**	**medium**	**high**
`new Foo()`	A_1	L	L_1	L_1
→ `x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

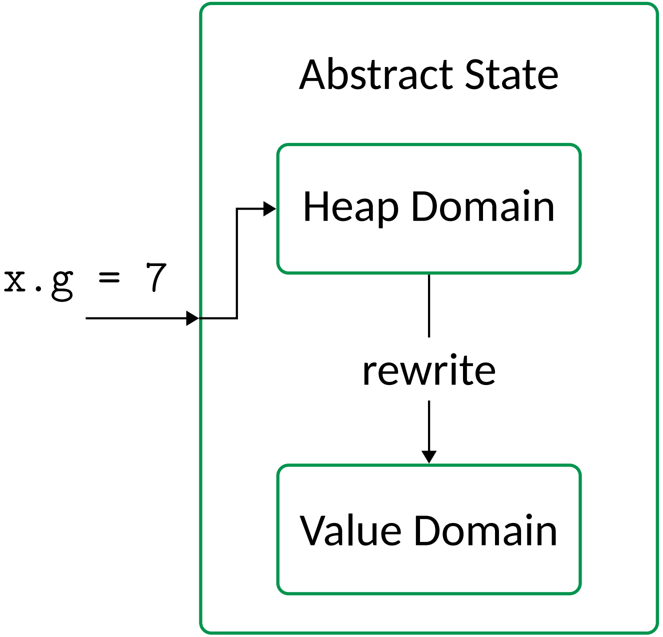
Heap Domain: $x \mapsto L_1$

Value Domain: $L_1 \mapsto \{5\}$

□ configurable

The Abstract State

[Fer16]



		precision		
expression	**variable**	**low**	**medium**	**high**
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
→ `x.g`	A_3	L	L_1	$L_{1.2}$

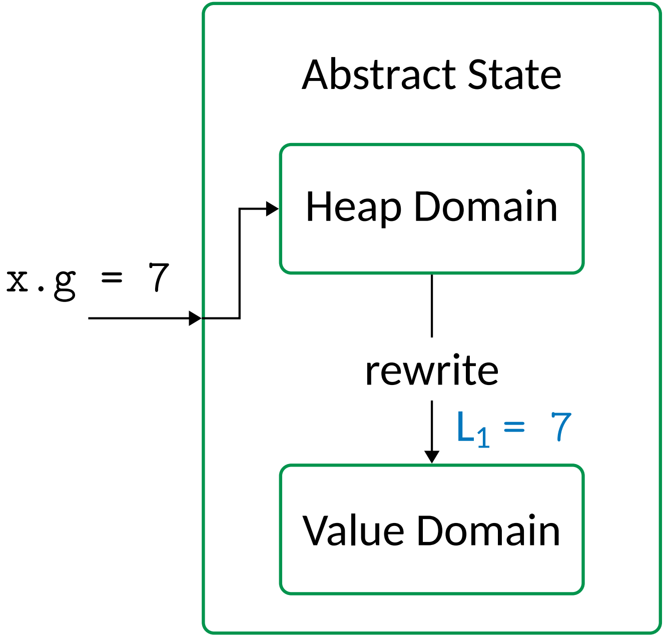
Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain: $L_1 \mapsto \{5\}$

□ configurable

The Abstract State

[Fer16]



x.g = 7

Abstract State

Heap Domain

rewrite

$L_1 = 7$

Value Domain

			precision	
expression	variable	low	medium	high
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
→ `x.g`	A_3	L	L_1	$L_{1.2}$

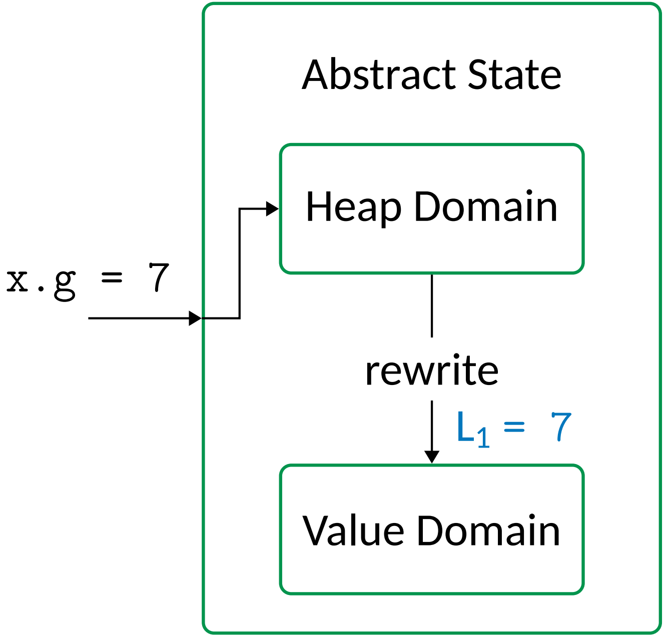
Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain: $L_1 \mapsto \{5, 7\}$

□ configurable

The Abstract State

[Fer16]



□ configurable

		precision		
expression	variable	low	medium	high
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
→ `x.g`	A_3	L	L_1	$L_{1.2}$

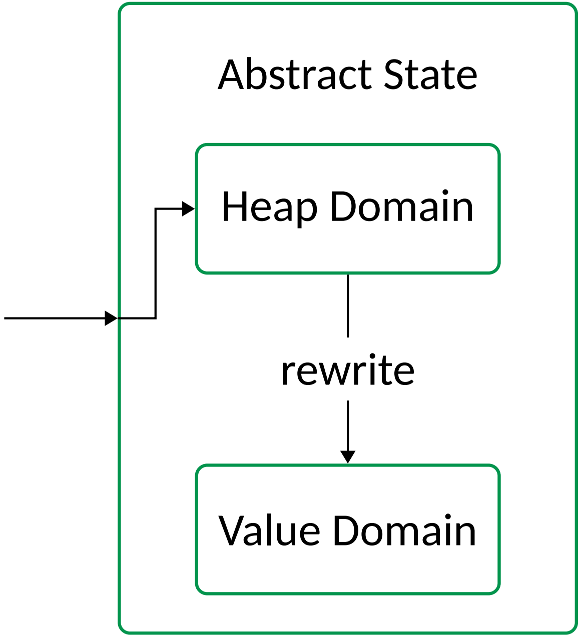
Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain: $L_1 \mapsto \{5, 7\}$

L_1 is assigned twice, so we lost precision!
Can we do better?

The Abstract State

[Fer16]



		precision		
expression	variable	low	medium	high
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

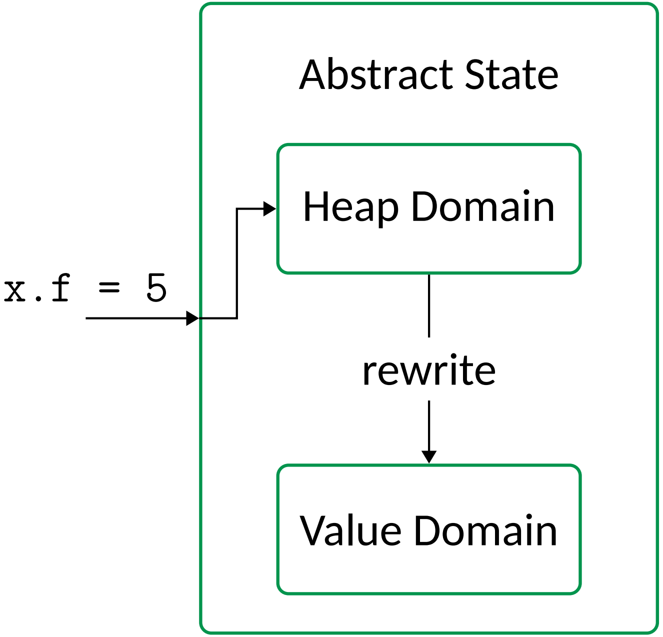
Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain:

☐ configurable

The Abstract State

[Fer16]



`x.f = 5`

Abstract State

Heap Domain

rewrite

Value Domain

		precision		
expression	variable	low	medium	high
`new Foo()`	A_1	L	L_1	L_1
→ `x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

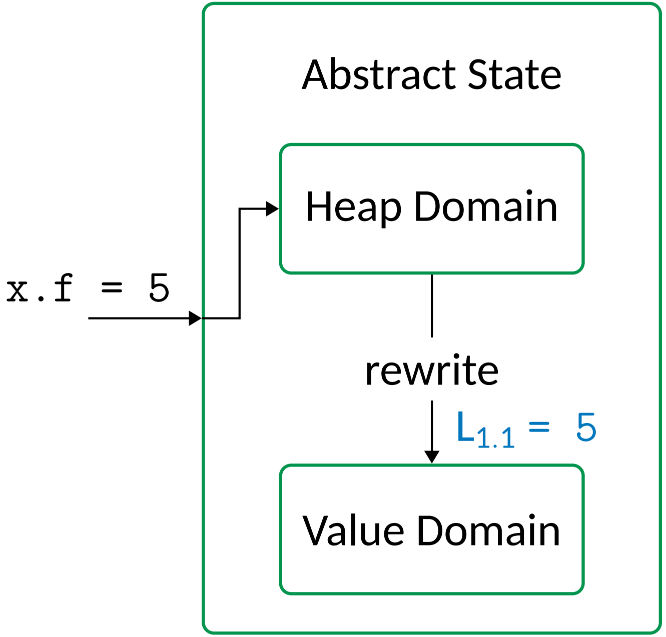
Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain:

□ configurable

The Abstract State

[Fer16]



		precision		
expression	variable	low	medium	high
`new Foo()`	A_1	L	L_1	L_1
→ `x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

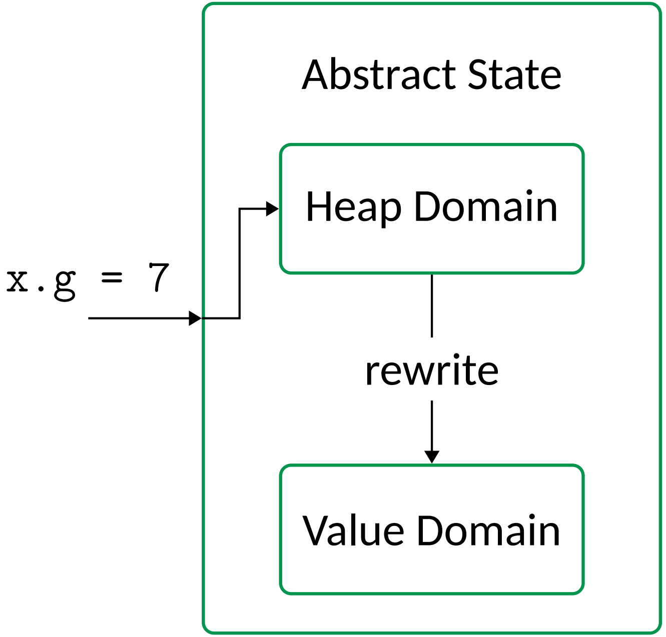
Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain: $L_{1.1} \mapsto \{5\}$

□ configurable

The Abstract State

[Fer16]



		precision		
expression	variable	low	medium	high
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
→ `x.g`	A_3	L	L_1	$L_{1.2}$

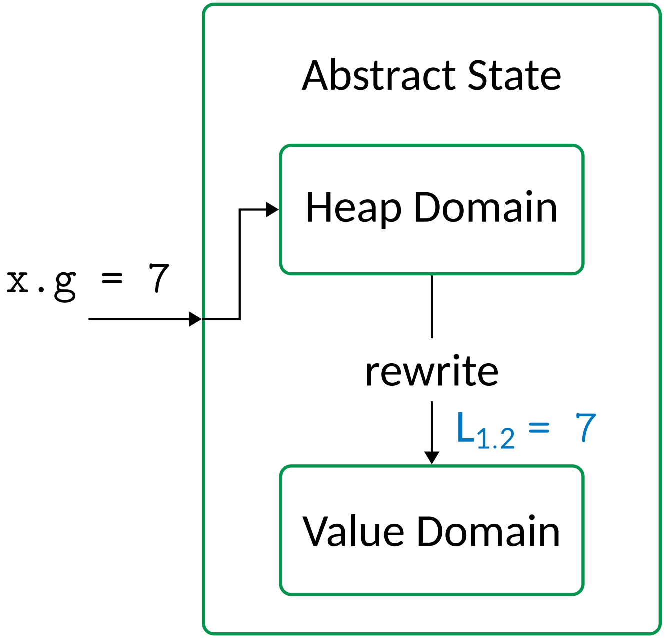
Heap Domain: $\mathtt{x} \mapsto L_1$

Value Domain: $L_{1.1} \mapsto \{5\}$

□ configurable

The Abstract State

[Fer16]



□ configurable

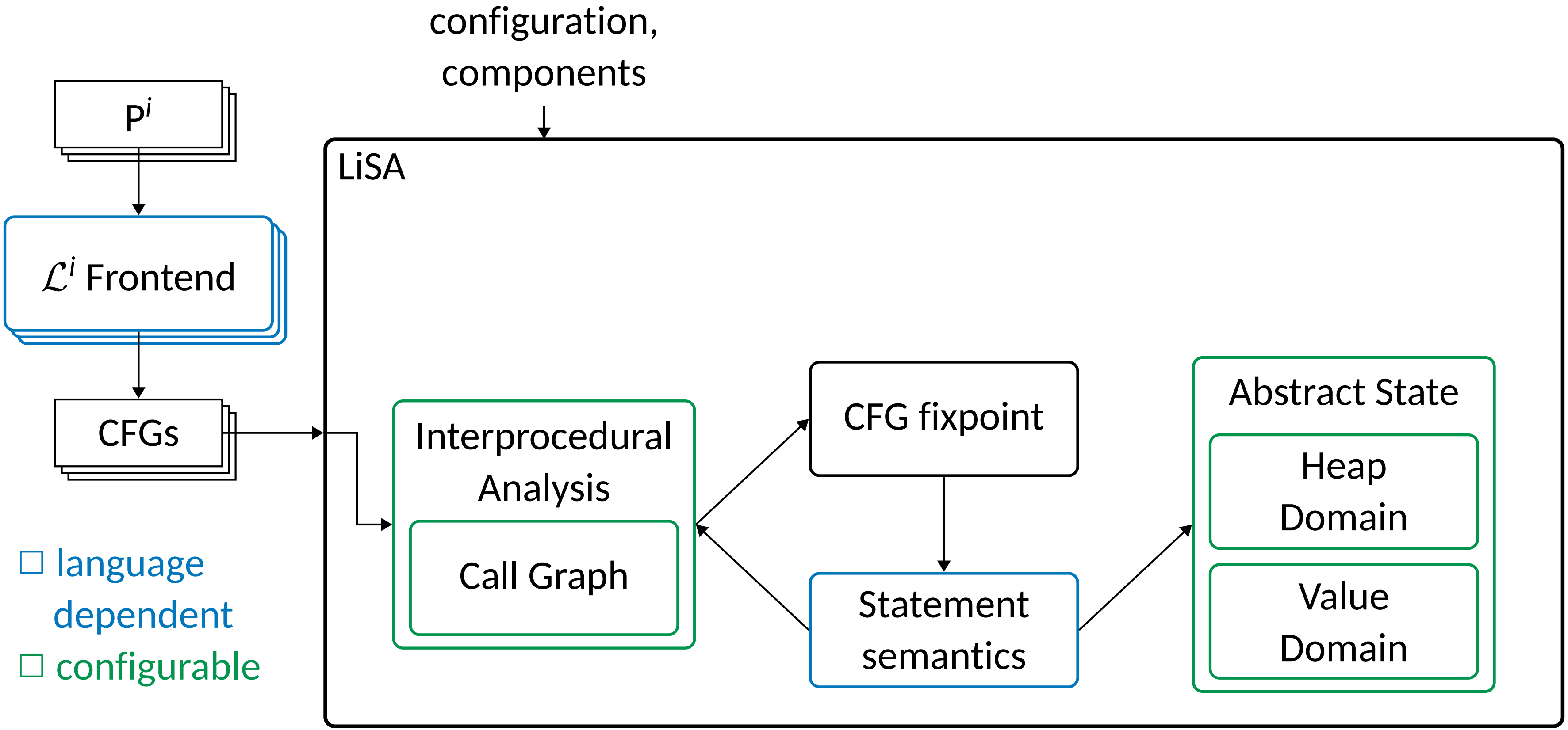
		precision		
expression	variable	low	medium	high
`new Foo()`	A_1	L	L_1	L_1
`x.f`	A_2	L	L_1	$L_{1.1}$
`x.g`	A_3	L	L_1	$L_{1.2}$

Heap Domain: $\mathtt{x} \mapsto L_1$

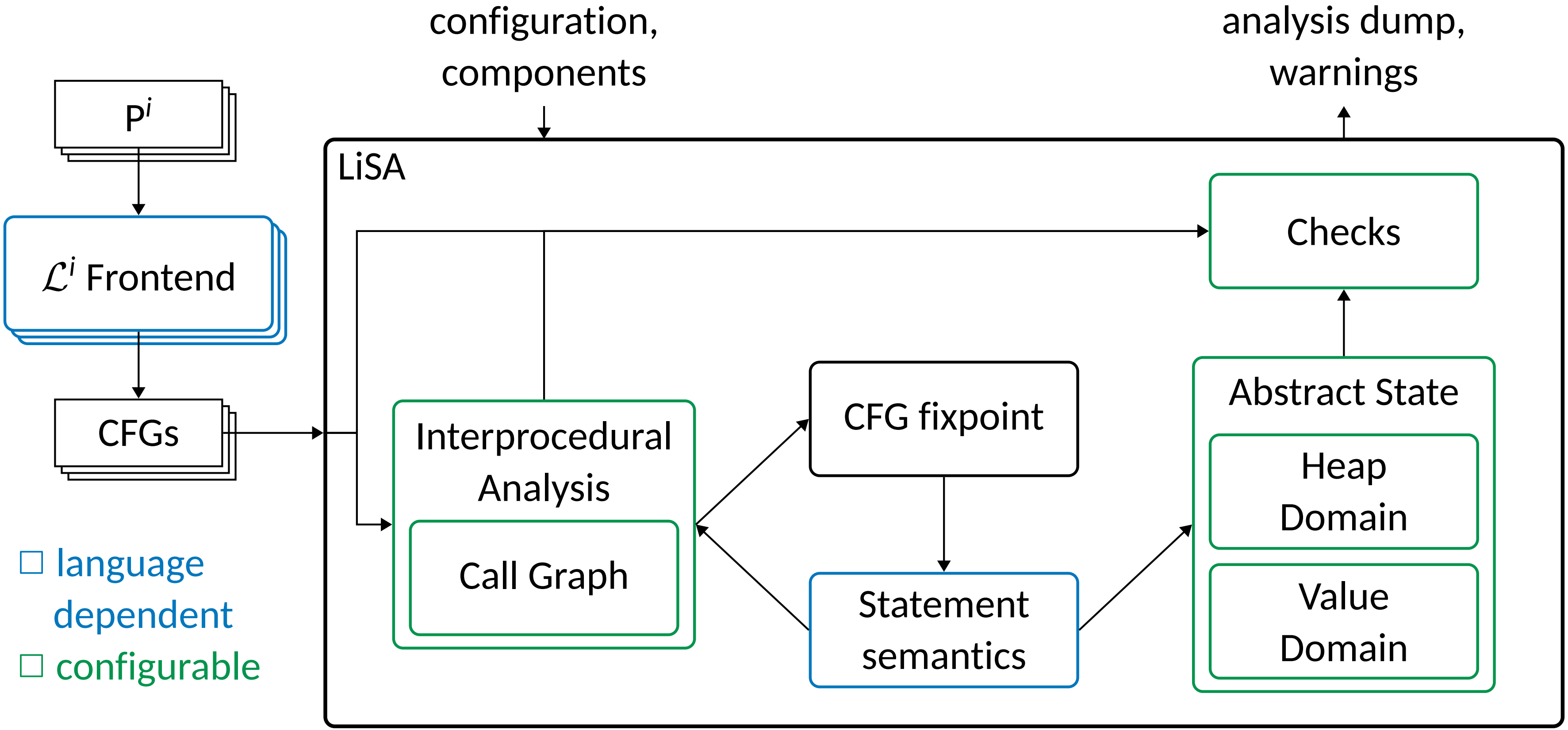
Value Domain: $L_{1.1} \mapsto \{5\}, L_{1.2} \mapsto \{7\}$

$L_{1.1}$ and $L_{1.2}$ are assigned once each, so we are more precise without changing the Value Domain!

LiSA Overview



LiSA Overview



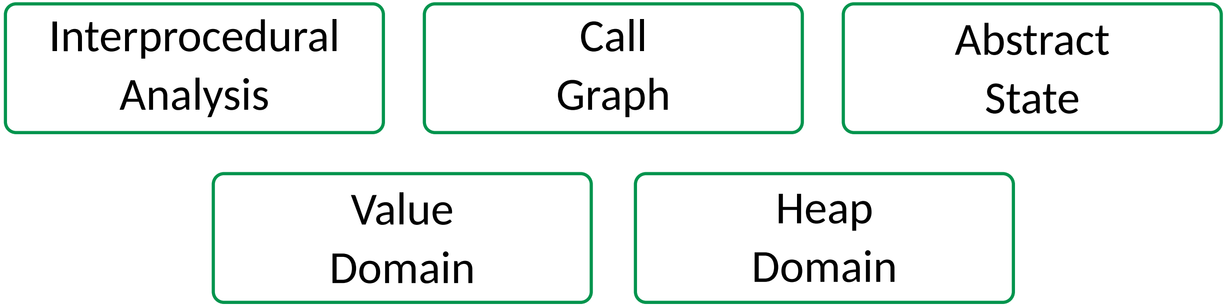
□ language dependent

□ configurable

Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

Configuring Analysis Components



Interprocedural Analysis

Call Graph

Abstract State

Value Domain

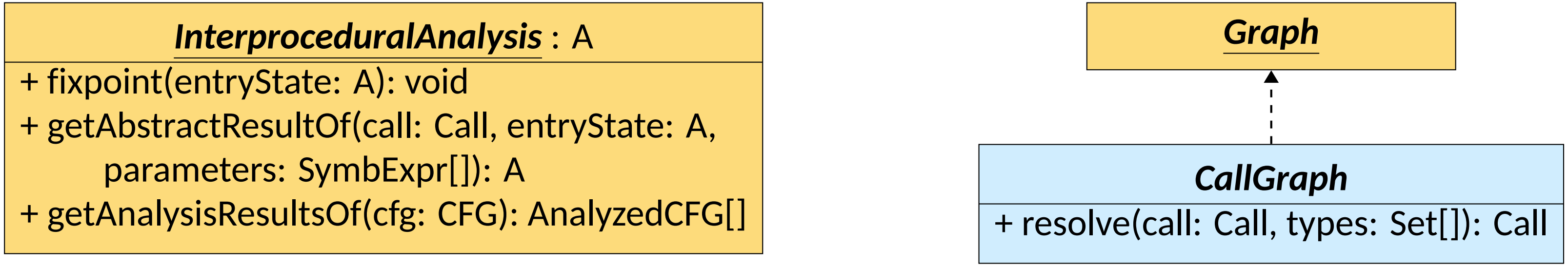
Heap Domain

Configurable components can be passed to the engine modularly

They are defined as **interfaces** 🟨 or **abstract classes** 🟦 whose instances provide specific logic

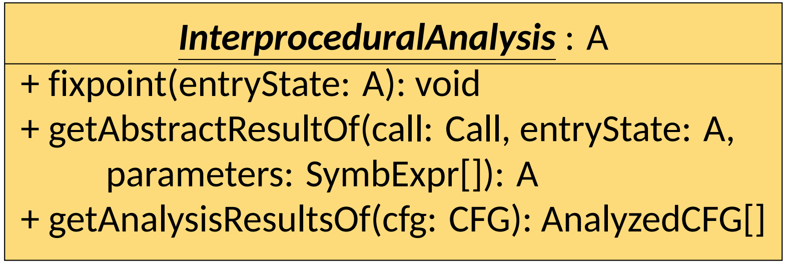
Calls and Program Fixpoint

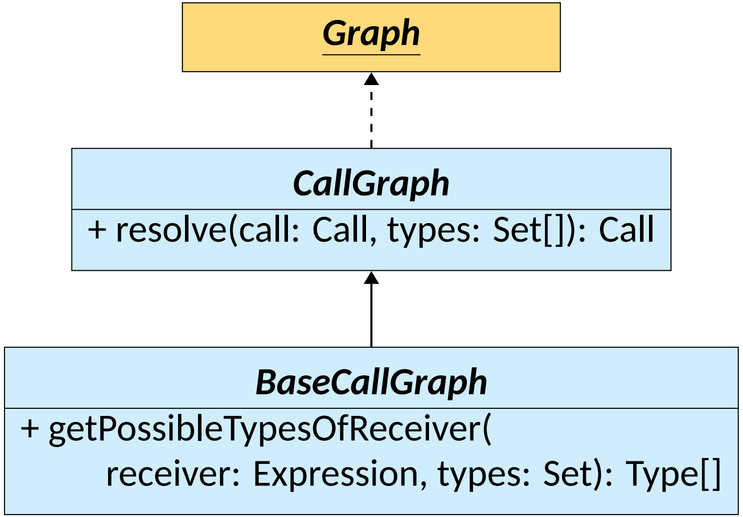
Both components are **analysis independent**


InterproceduralAnalysis : A
+ fixpoint(entryState: A): void
+ getAbstractResultOf(call: Call, entryState: A, parameters: SymbExpr[]): A
+ getAnalysisResultsOf(cfg: CFG): AnalyzedCFG[]
Graph
CallGraph
+ resolve(call: Call, types: Set[]): Call

Calls and Program Fixpoint

Both components are **analysis independent**


InterproceduralAnalysis : A
+ fixpoint(entryState: A): void
+ getAbstractResultOf(call: Call, entryState: A,
parameters: SymbExpr[]): A
+ getAnalysisResultsOf(cfg: CFG): AnalyzedCFG[]

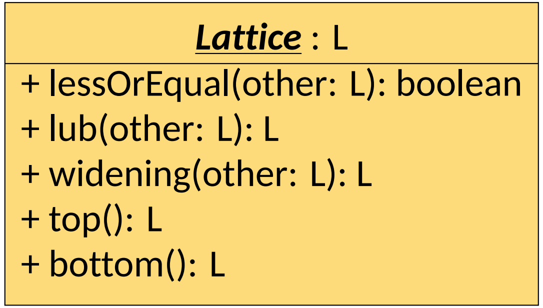

Graph
CallGraph
+ resolve(call: Call, types: Set[]): Call
BaseCallGraph
+ getPossibleTypesOfReceiver(
receiver: Expression, types: Set): Type[]

Abstract States

Abstract states must be built, merged, and compared during the fixpoint

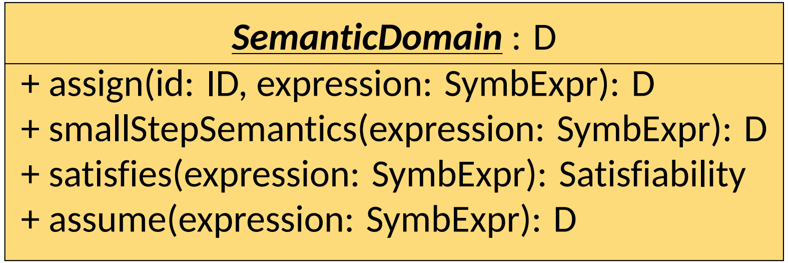
Abstract States

Abstract states must be built, merged, and compared during the fixpoint



Lattice : L

+ lessOrEqual(other: L): boolean
+ lub(other: L): L
+ widening(other: L): L
+ top(): L
+ bottom(): L

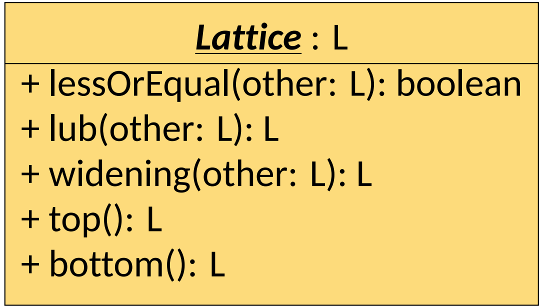


SemanticDomain : D

+ assign(id: ID, expression: SymbExpr): D
+ smallStepSemantics(expression: SymbExpr): D
+ satisfies(expression: SymbExpr): Satisfiability
+ assume(expression: SymbExpr): D

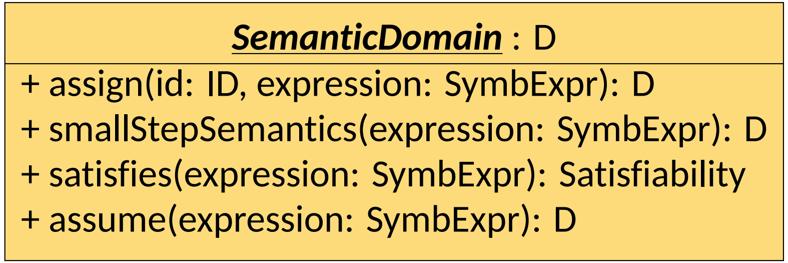
Abstract States

Abstract states must be built, merged, and compared during the fixpoint



Lattice : L

+ lessOrEqual(other: L): boolean
+ lub(other: L): L
+ widening(other: L): L
+ top(): L
+ bottom(): L



SemanticDomain : D

+ assign(id: ID, expression: SymbExpr): D
+ smallStepSemantics(expression: SymbExpr): D
+ satisfies(expression: SymbExpr): Satisfiability
+ assume(expression: SymbExpr): D

Abstract State, Value Domain, and Heap Domain all implement both!

Examples (Less Pseudocode)

```
Plus.semantics(): // left + right
  if type(left) is string or type(right) is string:
    return domain.smallStepSemantics(new BinaryExpression("cat", left, right))
  else:
    return domain.smallStepSemantics(new BinaryExpression("+", left, right))
```

Examples (Less Pseudocode)

```
Plus.semantics(): // left + right
  if type(left) is string or type(right) is string:
    return domain.smallStepSemantics(new BinaryExpression("cat", left, right))
  else:
    return domain.smallStepSemantics(new BinaryExpression("+", left, right))
```

```
Conditional.semantics(): // condition ? ifTrue : ifFalse
  sat = domain.satisfies(condition)
  tt = domain.assume(condition).smallStepSemantics(ifTrue)
  ff = domain.assume(condition.negate()).smallStepSemantics(ifFalse)
  return sat.holds() ? tt : (sat.notHolds() ? ff : tt.lub(ff))
```

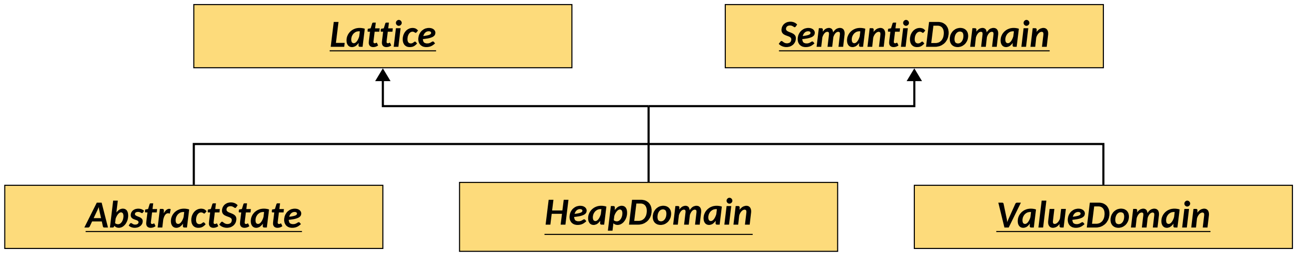
Examples (Less Pseudocode)

```
Plus.semantics(): // left + right
  if type(left) is string or type(right) is string:
    return domain.smallStepSemantics(new BinaryExpression("cat", left, right))
  else:
    return domain.smallStepSemantics(new BinaryExpression("+", left, right))
```

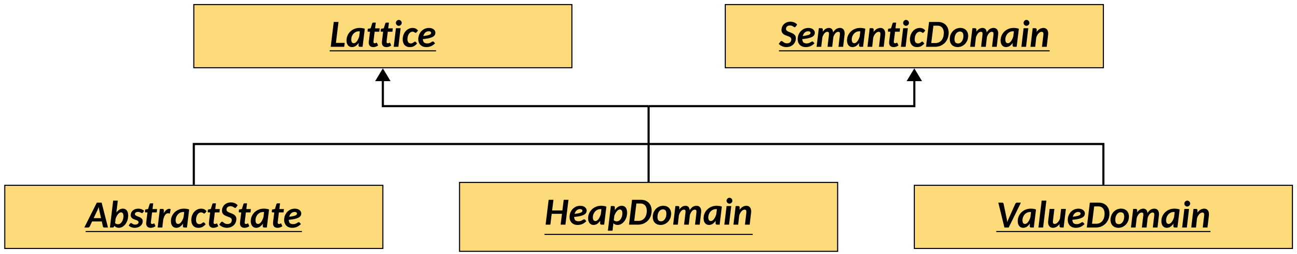
```
Conditional.semantics(): // condition ? ifTrue : ifFalse
  sat = domain.satisfies(condition)
  tt = domain.assume(condition).smallStepSemantics(ifTrue)
  ff = domain.assume(condition.negate()).smallStepSemantics(ifFalse)
  return sat.holds() ? tt : (sat.notHolds() ? ff : tt.lub(ff))
```

```
PreIncrement.semantics(): // ++var
  add = new BinaryExpression("+", var, new Const(1))
  d1 = domain.assign(var, add)
  return d1.smallStepSemantics(var)
```

Abstract State, Heap and Value Domain



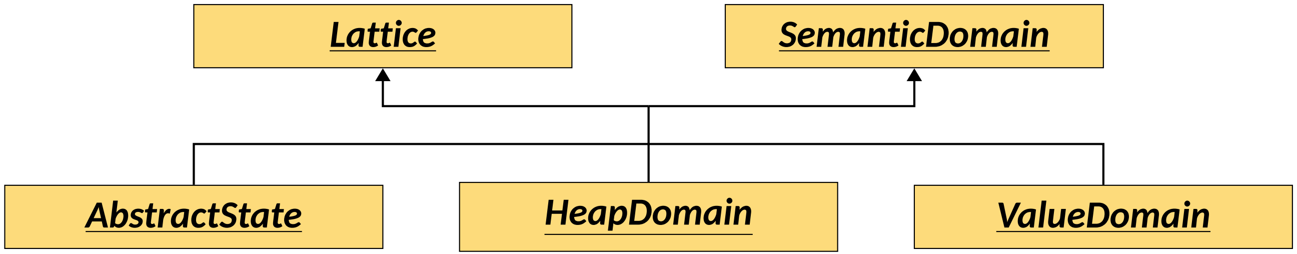
Abstract State, Heap and Value Domain



Each instance is a lattice element:

- Its state carries the abstract information
- Lattice operators compare/combine that information

Abstract State, Heap and Value Domain



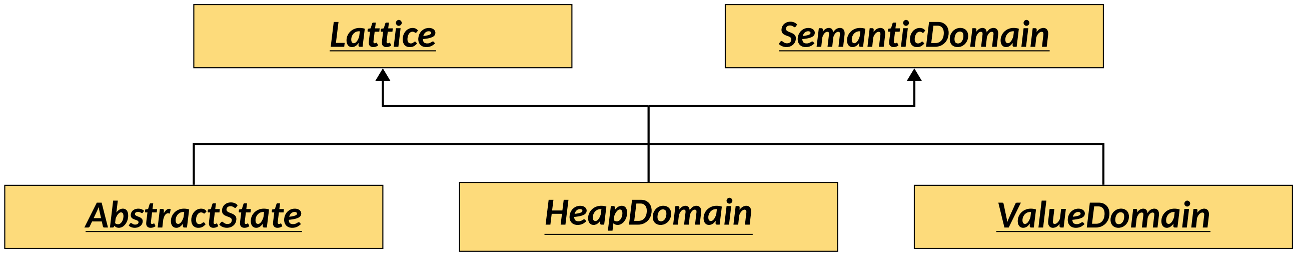
Each instance is a lattice element:

- Its state carries the abstract information
- Lattice operators compare/combine that information

Each instance can be used as a pre-state:

- `SemanticDomain` operators can transform it into a post-state

Abstract State, Heap and Value Domain



Each instance is a lattice element:

- Its state carries the abstract information
- Lattice operators compare/combine that information

Each instance can be used as a pre-state:

- `SemanticDomain` operators can transform it into a post-state

Can we further modularize?

Relational vs Non-Relational

Relational analyses have specific structures

- Hard to factor out general implementation patterns

Relational vs Non-Relational

Relational analyses have specific structures

- Hard to factor out general implementation patterns

Non-relational analyses instead share a common structure:

- Track values of single variables through a **function** (i.e., a map)
- The mapping changes **only with assignments** (usually)
- Evaluation logic is **analysis-specific**

Relational vs Non-Relational

Relational analyses have specific structures

- Hard to factor out general implementation patterns

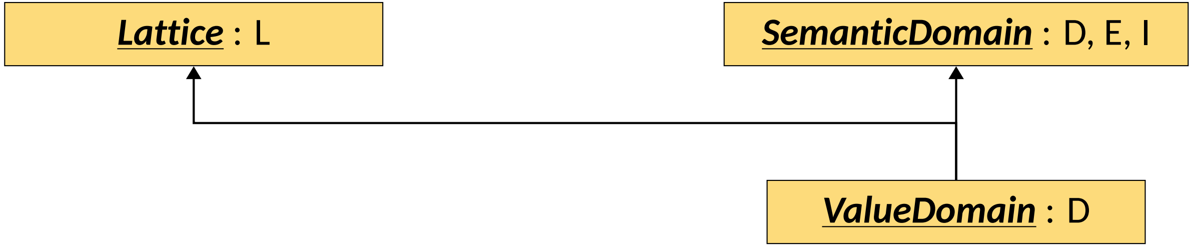
Non-relational analyses instead share a common structure:

- Track values of single variables through a **function** (i.e., a map)
- The mapping changes **only with assignments** (usually)
- Evaluation logic is **analysis-specific**

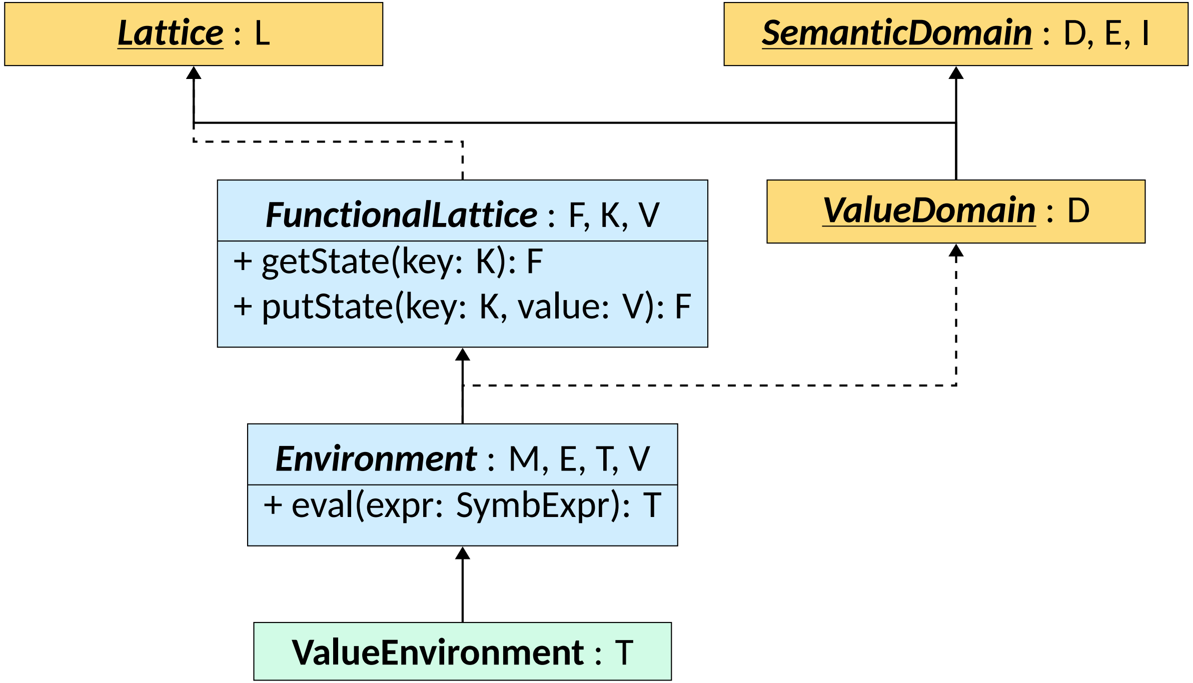
There is space for modularization!

- Abstract common logic (map and assignment)
- Build instances of atomic information with domain-specific logic

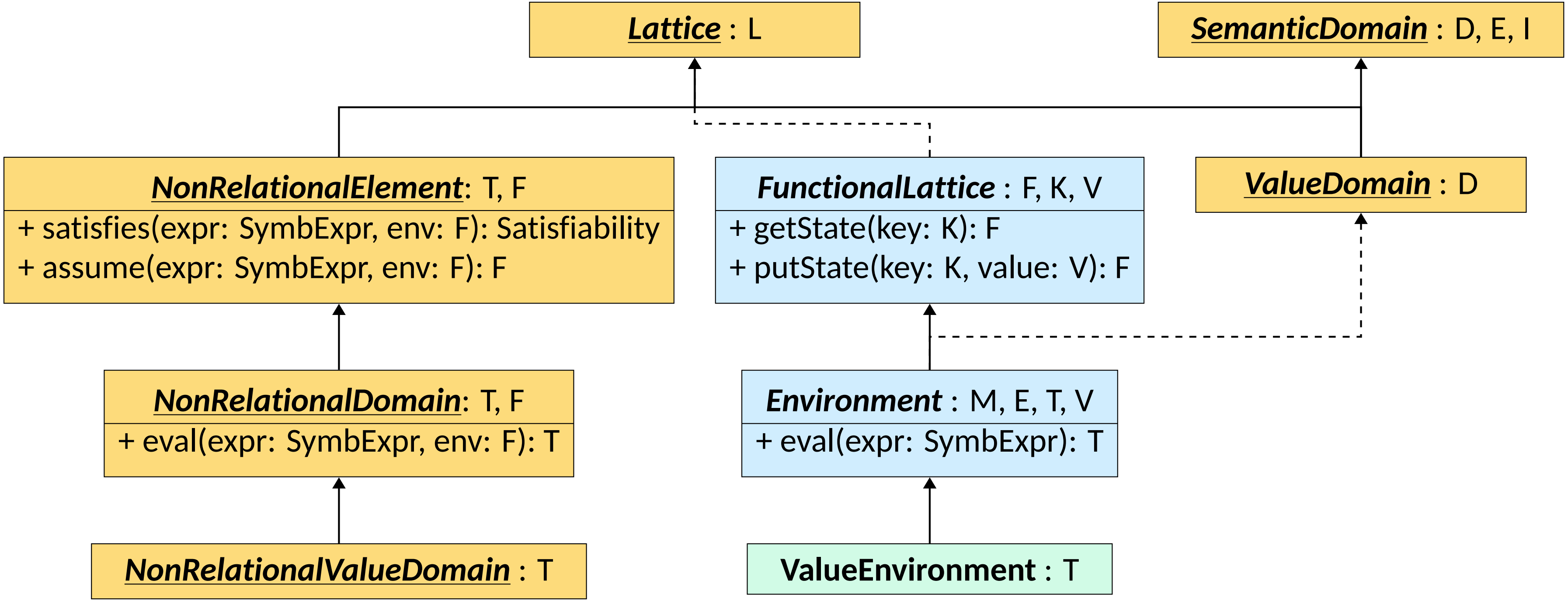
Main Classes for Value Analyses



Main Classes for Value Analyses


Lattice : L
SemanticDomain : D, E, I
FunctionalLattice : F, K, V
+ getState(key: K): F
+ putState(key: K, value: V): F
ValueDomain : D
Environment : M, E, T, V
+ eval(expr: SymbExpr): T
ValueEnvironment : T

Main Classes for Value Analyses



Lattice : L

SemanticDomain : D, E, I

NonRelationalElement: T, F
+ satisfies(expr: SymbExpr, env: F): Satisfiability
+ assume(expr: SymbExpr, env: F): F

FunctionalLattice : F, K, V
+ getState(key: K): F
+ putState(key: K, value: V): F

ValueDomain : D

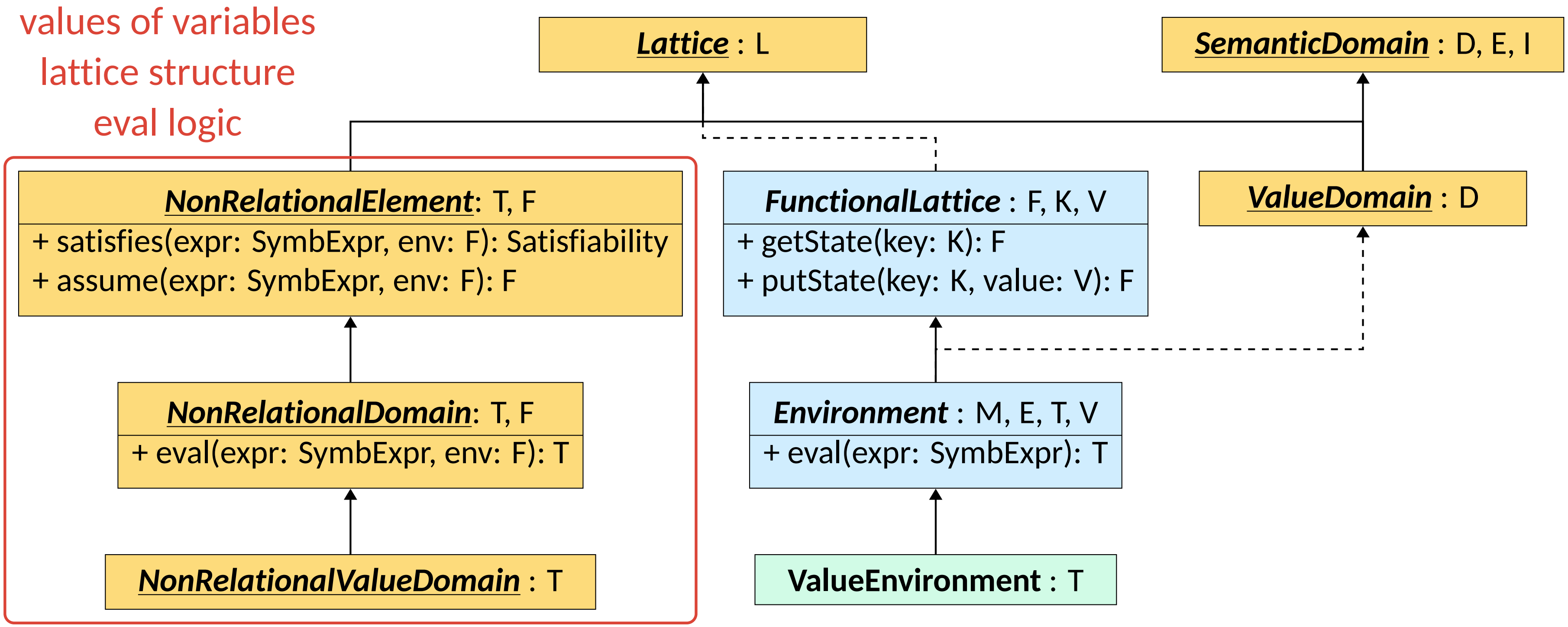
NonRelationalDomain: T, F
+ eval(expr: SymbExpr, env: F): T

Environment : M, E, T, V
+ eval(expr: SymbExpr): T

NonRelationalValueDomain : T

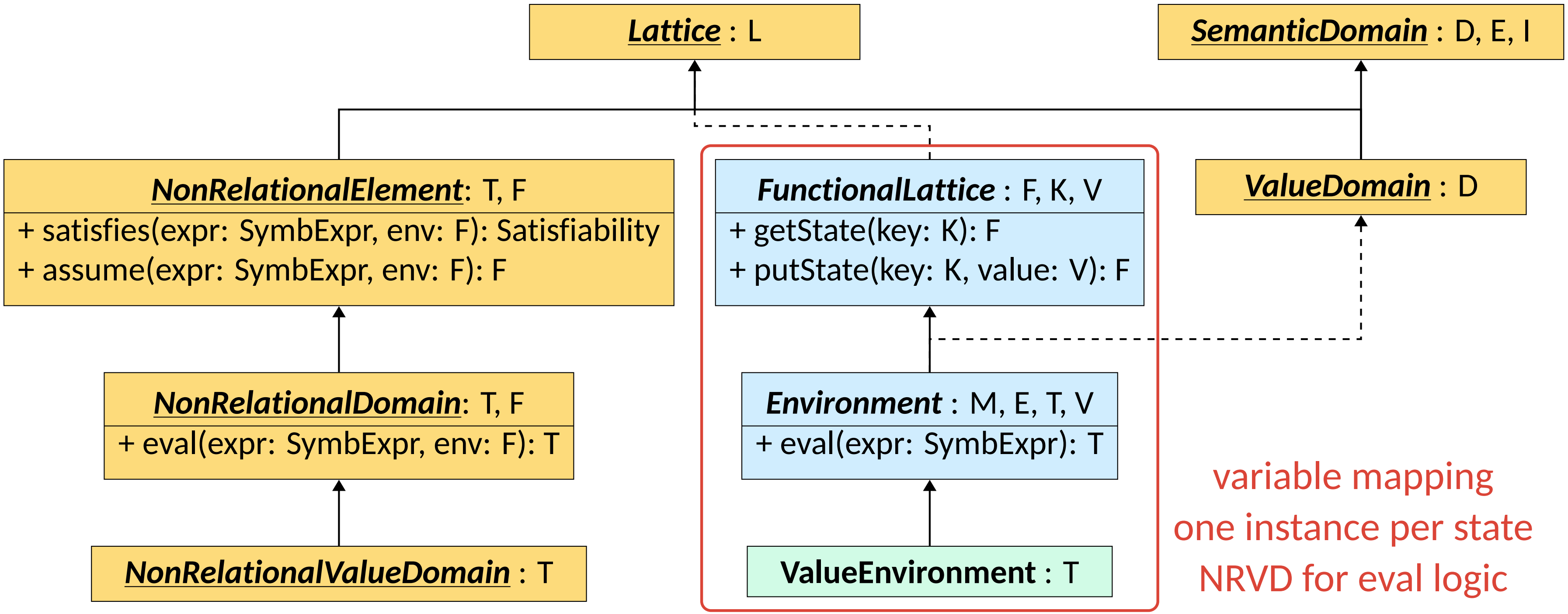
ValueEnvironment : T

Main Classes for Value Analyses



values of variables
lattice structure
eval logic
Lattice : L
SemanticDomain : D, E, I
NonRelationalElement: T, F
+ satisfies(expr: SymbExpr, env: F): Satisfiability
+ assume(expr: SymbExpr, env: F): F
FunctionalLattice : F, K, V
+ getState(key: K): F
+ putState(key: K, value: V): F
ValueDomain : D
NonRelationalDomain: T, F
+ eval(expr: SymbExpr, env: F): T
Environment : M, E, T, V
+ eval(expr: SymbExpr): T
NonRelationalValueDomain : T
ValueEnvironment : T

Main Classes for Value Analyses


Lattice : L
SemanticDomain : D, E, I
NonRelationalElement: T, F
+ satisfies(expr: SymbExpr, env: F): Satisfiability
+ assume(expr: SymbExpr, env: F): F
FunctionalLattice : F, K, V
+ getState(key: K): F
+ putState(key: K, value: V): F
ValueDomain : D
NonRelationalDomain: T, F
+ eval(expr: SymbExpr, env: F): T
Environment : M, E, T, V
+ eval(expr: SymbExpr): T
NonRelationalValueDomain : T
ValueEnvironment : T

variable mapping
one instance per state
NRVD for eval logic

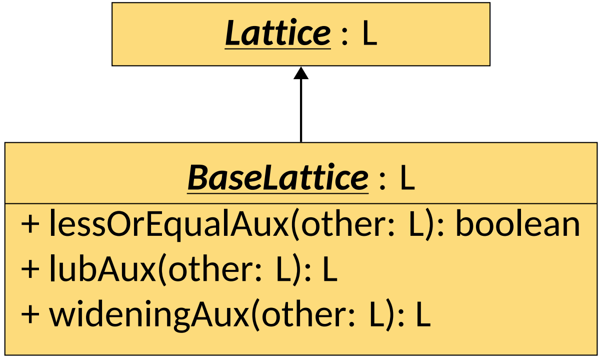
Easing Development: Lattice Functionalities

We can avoid coding common lattice logic and standard lattice structures

Lattice : L

Easing Development: Lattice Functionalities

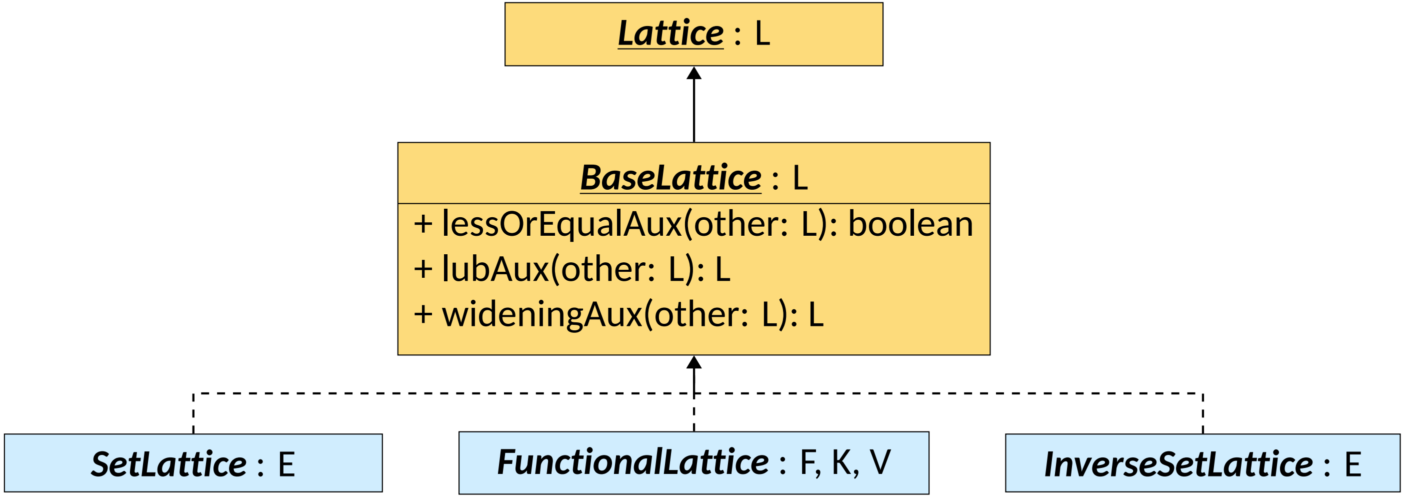
We can avoid coding common lattice logic and standard lattice structures



Lattice : L
BaseLattice : L
+ lessOrEqualAux(other: L): boolean
+ lubAux(other: L): L
+ wideningAux(other: L): L

Easing Development: Lattice Functionalities

We can avoid coding common lattice logic and standard lattice structures



Lattice : L
BaseLattice : L
+ lessOrEqualAux(other: L): boolean
+ lubAux(other: L): L
+ wideningAux(other: L): L
SetLattice : E
FunctionalLattice : F, K, V
InverseSetLattice : E

$\langle \wp(X), \subseteq, \cup, \cap, \emptyset, X \rangle$

$\langle K \rightarrow V, \dot{\sqsubseteq}_V, \dot{\sqcup}_V, \dot{\sqcap}_V, \bot, \top \rangle$

$\langle \wp(X), \supseteq, \cap, \cup, X, \emptyset \rangle$

Easing Development: Recursive Evaluation

Interpreters evaluate expressions recursively:

$$\dfrac{\dfrac{\dfrac{\llbracket\texttt{x}\rrbracket\{x \mapsto 7, y \mapsto 3\} = 7}{\llbracket\texttt{x * 2}\rrbracket\{x \mapsto 7, y \mapsto 3\} = 14} \qquad \llbracket\texttt{y}\rrbracket\{x \mapsto 7, y \mapsto 3\} = 3}{\llbracket\texttt{x * 2 + y}\rrbracket\{x \mapsto 7, y \mapsto 3\} = 17}}{\llbracket\texttt{x = x * 2 + y}\rrbracket\{x \mapsto 7, y \mapsto 3\} = \{x \mapsto 17, y \mapsto 3\}}$$

Easing Development: Recursive Evaluation

Interpreters evaluate expressions recursively:

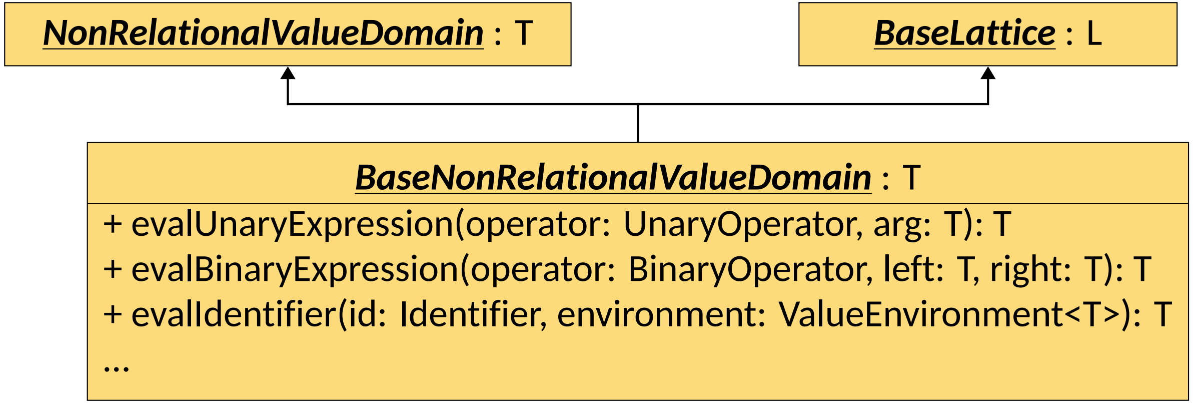
$$\dfrac{\dfrac{\dfrac{[\![\texttt{x}]\!]\{x \mapsto 7, y \mapsto 3\} = 7}{[\![\texttt{x * 2}]\!]\{x \mapsto 7, y \mapsto 3\} = 14} \qquad [\![\texttt{y}]\!]\{x \mapsto 7, y \mapsto 3\} = 3}{[\![\texttt{x * 2 + y}]\!]\{x \mapsto 7, y \mapsto 3\} = 17}}{[\![\texttt{x = x * 2 + y}]\!]\{x \mapsto 7, y \mapsto 3\} = \{x \mapsto 17, y \mapsto 3\}}$$

Symbolic Expressions implement the **visitor pattern** to ease recursive traversal:

```
public <T> T accept(ExpressionVisitor<T> visitor, Object... params) {
  T left = this.left.accept(visitor, params);
  T right = this.right.accept(visitor, params);
  return visitor.visit(this, left, right, params);
}
```

Easing Development: Recursive Evaluation

`BaseNonRelationalValueDomain` is an `ExpressionVisitor`:



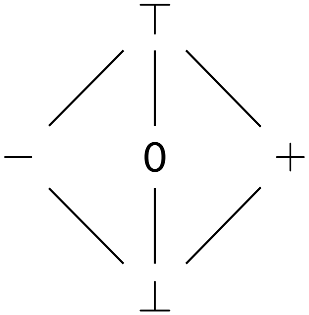
NonRelationalValueDomain : T
BaseLattice : L
BaseNonRelationalValueDomain : T
+ evalUnaryExpression(operator: UnaryOperator, arg: T): T
+ evalBinaryExpression(operator: BinaryOperator, left: T, right: T): T
+ evalIdentifier(id: Identifier, environment: ValueEnvironment<T>): T
...

Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

The Signs Domain

$\textsc{Sgn} \triangleq$

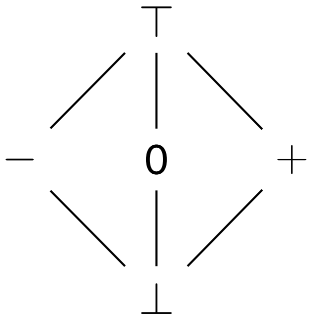


$$\textsc{Sign} \triangleq \langle \textsc{Id} \rightarrow \textsc{Sgn}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \bot, \top \rangle$$

The Signs Domain

$\textsc{Sign}$ is a value domain

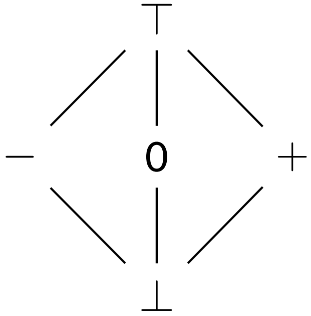
$\textsc{Sgn} \triangleq$



$$\textsc{Sign} \triangleq \langle \textsc{Id} \to \textsc{Sgn}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \bot, \top \rangle$$

The Signs Domain

$\textsc{Sgn} \triangleq$

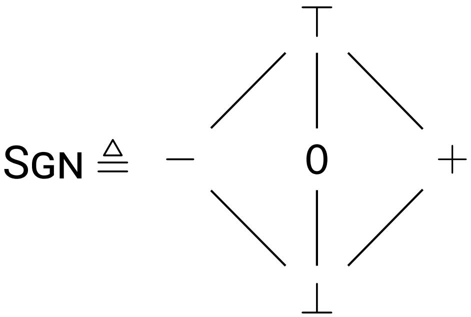


$$\textsc{Sign} \triangleq \langle \textsc{Id} \to \textsc{Sgn}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \bot, \top \rangle$$

Sign is a value domain

Sign is non relational

The Signs Domain



$\textsc{Sign} \triangleq \langle \textsc{Id} \to \textsc{Sgn}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \bot, \top \rangle$

SIGN is a value domain

SIGN is non relational

We implement it as a `NonRelationalValueDomain`

▷ A single class for SGN

- Lattice operators for SGN
- Evaluation logic combining SGN instances
- Can use `Base` interface to simplify evaluation

▷ `ValueEnvironment` manages lifting and assignments

Environment’s Assign (Simplified)

```java
public ValueEnvironment<T> assign(Identifier id, ValueExpression expression) {
  if (isBottom())
    // bottom values are preserved
    return (M) this;

  Map<Identifier, T> func = this.function.clone();
  T value = lattice.eval(expression, this);

  if (id.isWeak() && this.function.containsKey(id))
    // if we have a weak identifier for which we already have
    // information, we we perform a weak assignment
    value = value.lub(getState(id));

  func.put(id, value);
  return new ValueEnvironment<>(lattice, func);
}
```

Link to assign’s implementation

Signs' Implementation

Link to full Signs implementation (220 lines, 178 containing code)

Methods implemented:

- constructors, `equals()` and `hashCode()`
- `top()` and `bottom()` to retrieve the respective lattice elements (if these do not return a constant value, `isTop()` and `isBottom()` also have to be implemented)
- `lessOrEqualsAux()` and `lubAux()` for lattice operators
- `evalNonNullConstant()`, `evalUnaryExpression()`, and `evalBinaryExpression()` to evaluate expressions
- `representation()` to dump SIGN instances in various formats

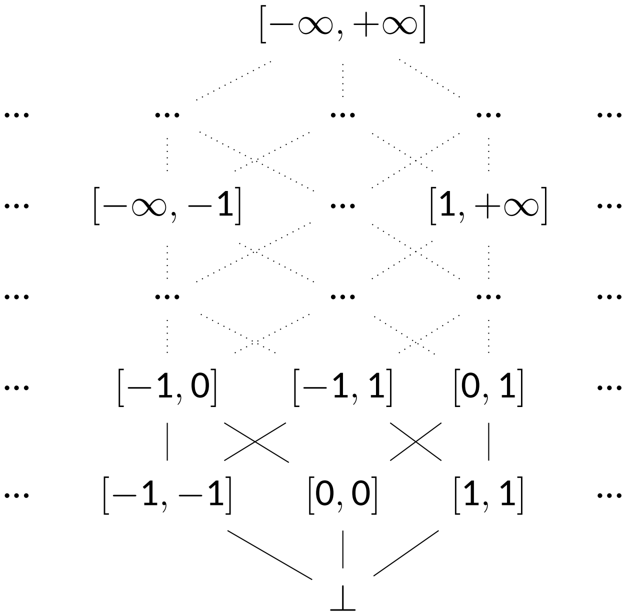
Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

The Intervals Domain

[CC77]

$\textsc{Intv} \triangleq$



$\textsc{Interval} \triangleq \langle \textsc{Id} \to \textsc{Intv}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \bot, \top \rangle$

$\textsc{Interval}$ is a value domain

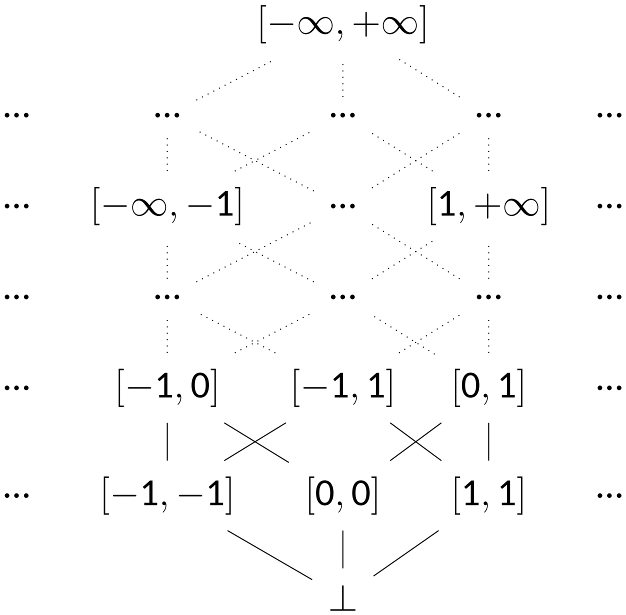
$\textsc{Interval}$ is non relational

We implement it as a `NonRelationalValueDomain`

The Intervals Domain

[CC77]

INTV ≜



$$\text{INTERVAL} \triangleq \langle \text{ID} \rightarrow \text{INTV}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \bot, \top \rangle$$

INTERVAL is a value domain

INTERVAL is non relational

We implement it as a `NonRelationalValueDomain`

▷ Same as SIGN and SGN, but

- We need the widening and glb as well
- We add `assume()` and `satisfies()`

Intervals' Implementation

Link to full Intervals implementation (357 lines, 294 containing code)

Methods implemented:

- constructors, `equals()` and `hashCode()`
- all methods of Signs
- `glbAux()` and `wideningAux()` for lattice operators
- `satisfiesBinaryExpression()` to test whether an expression is satisfied
- `assumeBinaryExpression()` to refine environments when traversing a condition

Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

The Upper Bounds Domain

[LF10]

The domain of Upper Bounds keep tracks of relations of the form $x < y$

The Upper Bounds Domain

[LF10]

The domain of Upper Bounds keep tracks of relations of the form x < y

Despite this, it has a convenient non-relational-like representation:

$$\left\{ \begin{array}{l} x \mapsto \{y, z\} \\ z \mapsto \{w\} \end{array} \right\} \Longrightarrow \begin{array}{l} x < y \land x < z \\ z < w \end{array}$$

Values of the function are sets in $\wp(\mathsf{ID})$ with operators $\supseteq$, $\cap$, $\cup$, forming an inverse set lattice

The Upper Bounds Domain

[LF10]

The domain of Upper Bounds keep tracks of relations of the form x < y

Despite this, it has a convenient non-relational-like representation:

$$\left\{\begin{array}{l} x \mapsto \{y, z\} \\ z \mapsto \{w\} \end{array}\right\} \implies \begin{array}{l} x < y \land x < z \\ z < w \end{array}$$

Values of the function are sets in $\wp(\mathrm{ID})$ with operators $\supseteq$, $\cap$, $\cup$, forming an inverse set lattice

But the domain is relational: we cannot implement $\mathbb{S}^\sharp[\![\mathtt{x} = \bullet]\!]$ since we have to clean up occurrences of $\mathtt{x}$

Upper Bounds’ Implementation

Link to full Upper Bounds implementation (326 lines, 235 containing code)

`IdSet` as an `InverseSetLattice` to hold sets of variables

- Lattice operators for free, except `wideningAux()`

`StrictUpperBounds` as an `Environment` and a `ValueDomain` implementing:

- constructors
- `assign()` to cleanup invalid bounds and add new ones
- `smallStepSemantics()` as a no-op
- `satisfies()` to test whether an expression is satisfied
- `assume()` to add bounds when traversing a condition

Instead, `equals()`, `hashCode()`, and lattice operators are provided by `Environment`

Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

Products

A product is a combination of two or more analyses [CCF13]

- That run independently (i.e., a **Cartesian product**)
- That might exchange information (e.g., a **reduced** or **Granger product**)

Products

A product is a combination of two or more analyses [CCF13]

- That run independently (i.e., a **Cartesian product**)
- That might exchange information (e.g., a **reduced** or **Granger product**)

Regardless of the formalism, products in LiSA are just “nested” analyses

Products

A product is a combination of two or more analyses [CCF13]

- That run independently (i.e., a **Cartesian product**)
- That might exchange information (e.g., a **reduced** or **Granger product**)

Regardless of the formalism, products in LiSA are just “nested” analyses

- You define the individual analyses beforehand
- You define the product as a new analysis that contains the client ones in its state
- Lattice operators and abstract transformers forward to nested analysis, with optional closures and refinements
- This maintains modularity!

The Pentagons Domain

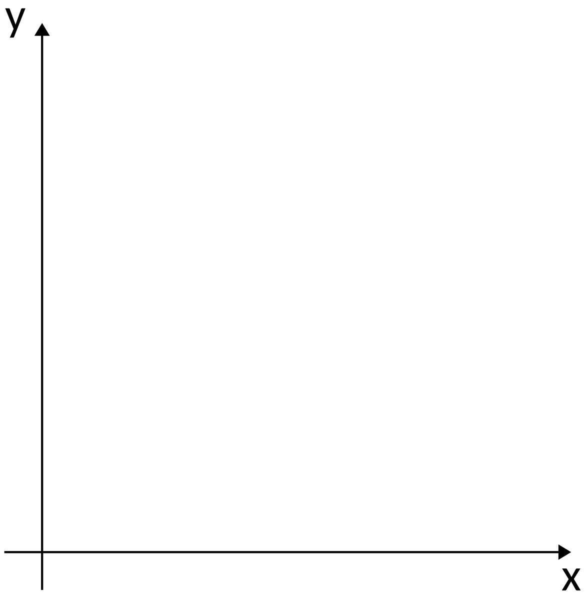
[LF10]

The Pentagons abstract domain is a relational domain defined as the reduced product of Intervals and Upper Bounds

The Pentagons Domain

[LF10]

The Pentagons abstract domain is a relational domain defined as the reduced product of Intervals and Upper Bounds



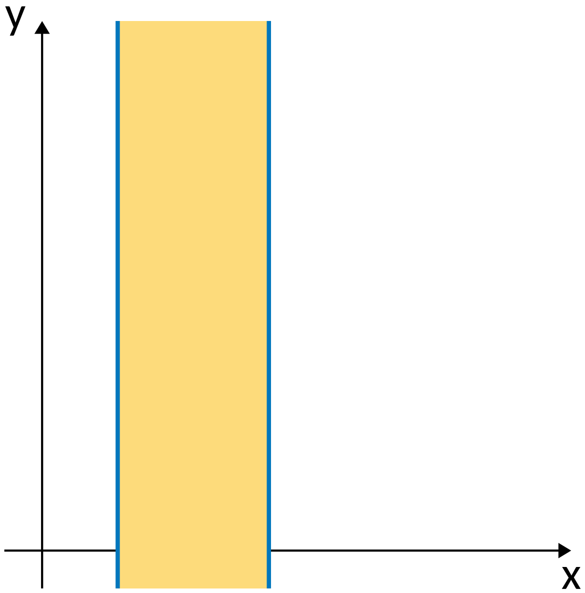
Intervals:

Upper Bounds:

The Pentagons Domain

[LF10]

The Pentagons abstract domain is a relational domain defined as the reduced product of Intervals and Upper Bounds



Intervals:

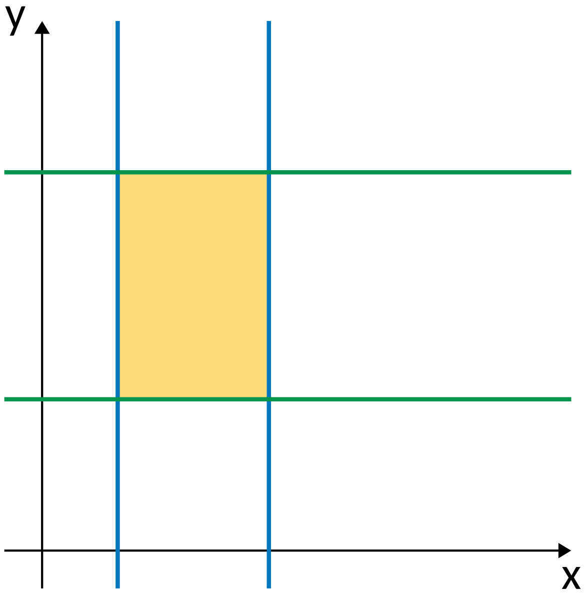
$$x \mapsto [3, 9]$$

Upper Bounds:

The Pentagons Domain

[LF10]

The Pentagons abstract domain is a relational domain defined as the reduced product of Intervals and Upper Bounds



Intervals:

$x \mapsto [3, 9]$

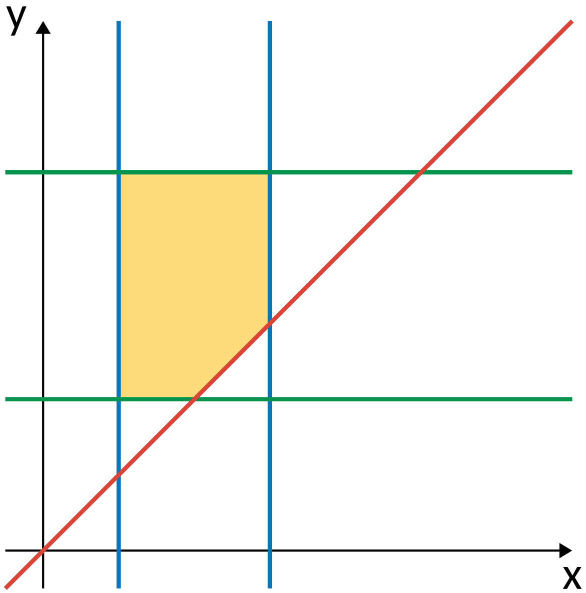
$y \mapsto [6, 15]$

Upper Bounds:

The Pentagons Domain

[LF10]

The Pentagons abstract domain is a relational domain defined as the reduced product of Intervals and Upper Bounds



Intervals:

$x \mapsto [3, 9]$

$y \mapsto [6, 15]$

Upper Bounds:

$x \mapsto \{y\}$

Pentagons’ Implementation

Link to full Pentagons implementation (307 lines, 247 containing code)

`Pentagons` as `ValueDomain` implementing:

- constructors, `equals()`, and `hashCode()`
- `top()` and `bottom()` propagating the calls
- `isTop()` and `isBottom()` matching the respective implementations
- `lessOrEqualsAux()` and `lubAux()` perform closures, while `wideningAux()` is just forwarded
- `assign()`, `smallStepSemantics()`, `satisfies()`, and `assume()` propagate the calls to the underlying analyses
- `assign()` adds the semantics for a new assignment

The remaining methods are mostly forwarding calls, and are needed for infrastructural reasons

Today’s Plan

1. **Components of a Static Analyzer**
2. **LiSA: a Library for Static Analysis**
3. **LiSA’s High-Level Architecture**
   - 3.1 Call resolution and evaluation
   - 3.2 Statement rewriting
   - 3.3 Memory and Value abstractions
4. **Putting it Into Code**
   - 4.1 The Signs Domain
   - 4.2 The Intervals Domain
   - 4.3 The Upper Bounds Domain
   - 4.4 The Pentagons Domain
   - 4.5 Information flow: the Taint analysis

Information Flow Analysis

Information Flow analyses aim at understanding how data flows from a location to another during the execution

- It does not care about values, but **how they are computed**

Information Flow Analysis

Information Flow analyses aim at understanding how data flows from a location to another during the execution

- It does not care about values, but **how they are computed**

Taint analysis is one instance of Information Flow that only considers explicit flows

- We mark relevant values as tainted
- We mark variables as tainted when they are assigned a tainted value
- After the analysis, we check where that taintedness reached

Example: SQL Injections

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

Example: SQL Injections

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

User input: this is dangerous!

Example: SQL Injections

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

Example: SQL Injections

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

Example: SQL Injections

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

Example: SQL Injections

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

Example: SQL Injections

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

Example: SQL Injections

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

Example: SQL Injections

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
  boolean success = false;
  String username = request.getParameter("username");
  String password = request.getParameter("password");
  String query = "SELECT * FROM users WHERE username='" + username
          + "' and password='" + password + "'";

  Statement stmt = null;
  Connection conn = DriverManager.getConnection("jdbc:mysql://" + DB_URL,
            DB_USER, DB_PWD);
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(query);
  // ...
}
```

Security hotspot: no user input allowed!

Implementing Taint Analysis

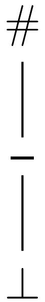
Taint analysis can be implemented in different ways (e.g., boolean formulas [SBE$^+$19])

Implementing Taint Analysis

Taint analysis can be implemented in different ways (e.g., boolean formulas [SBE+19])

We implement it as a non-relational domain:

- One bit of taintedness per variable
- Taintedness is propagated from one variable to another on assignments only

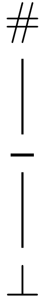


Implementing Taint Analysis

Taint analysis can be implemented in different ways (e.g., boolean formulas [SBE+19])

We implement it as a non-relational domain:

- One bit of taintedness per variable
- Taintedness is propagated from one variable to another on assignments only



We will use **annotations** for detecting sources
After the analysis, we will run a **semantic check** to find dangerous sinks

Taint’s Implementation

Link to full Taint implementation (231 lines, 162 containing code)
Link to full Semantic Check implementation (123 lines, 84 containing code)

`Taint` as `BaseNonRelationalValueDomain` implementing:

- constructors, `equals()`, and `hashCode()`
- same lattice operators of Signs and Intervals
- `evalIdentifier()` looks at the identifier’s annotations before looking inside the mapping
- other `evalX()` methods return the lub of the operands

`TaintCheck` as `SemanticCheck` inspecting each `Statement`:

- it skips everything that is not a call
- it checks parameters of each call for the sink annotation
- it issues warnings if a sink is reached by a tainted value

Thanks!

luca.negrini@unive.it • LiSA on GitHub • Today’s code

LiSA

Bibliography I

[BS96] David F. Bacon and Peter F. Sweeney. **Fast static analysis of C++ virtual function calls.** In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 324–341, New York, NY, USA, October 1996. Association for Computing Machinery.

[CC77] Patrick Cousot and Radhia Cousot. **Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.** In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[CC92] Patrick Cousot and Radhia Cousot. **Abstract interpretation and application to logic programs.** *The Journal of Logic Programming*, 13(2):103–179, jul 1992.

[CCF13] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. **A Survey on Product Operators in Abstract Interpretation.** *Electronic Proceedings in Theoretical Computer Science*, 129:325–336, September 2013. arXiv:1309.5146 [cs].

[Cou21] Patrick Cousot. ***Principles of Abstract Interpretation*.** MIT Press, 2021.

Bibliography II

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. **Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis.** In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101, Berlin, Heidelberg, 1995. Springer.

[Fer16] Pietro Ferrara. **A generic framework for heap and value analyses of object-oriented programming languages.** *Theoretical Computer Science*, 631:43–72, June 2016.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. **Call graph construction in object-oriented languages.** In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 108–124, New York, NY, USA, October 1997. Association for Computing Machinery.

[Hin01] Michael Hind. **Pointer analysis: haven't we solved this problem yet?** In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, jun 2001. Association for Computing Machinery.

Bibliography III

[KK16] Vini Kanvar and Uday P. Khedker. **Heap Abstractions for Static Analysis.** *ACM Comput. Surv.*, 49(2):29:1–29:47, jun 2016.

[LF10] Francesco Logozzo and Manuel Fähndrich. **Pentagons: A weakly relational abstract domain for the efficient validation of array accesses.** *Science of Computer Programming*, 75(9):796–807, September 2010.

[Neg23] Luca <1993> Negrini. ***A generic framework for multilanguage analysis*.** Doctoral Thesis, Università Ca' Foscari Venezia, January 2023. Accepted: 2022-12-12 Journal Abbreviation: A generic framework for multilanguage analysis.

[NFAC23] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. **LiSA: A Generic Framework for Multilanguage Static Analysis.** In Vincenzo Arceri, Agostino Cortesi, Pietro Ferrara, and Martina Olliaro, editors, *Challenges of Software Verification*, pages 19–42. Springer Nature, Singapore, 2023.

[PS81] M Pnueli and Micha Sharir. **Two approaches to interprocedural data flow analysis.** *Program flow analysis: theory and applications*, pages 189–234, 1981.

Bibliography IV

[SBE^{+}19] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. **Static Identification of Injection Attacks in Java.** *ACM Trans. Program. Lang. Syst.*, 41(3):18:1–18:58, July 2019.