MIR: A Formal and Minimal Intermediate Representation for Rigorous Python Program Analysis

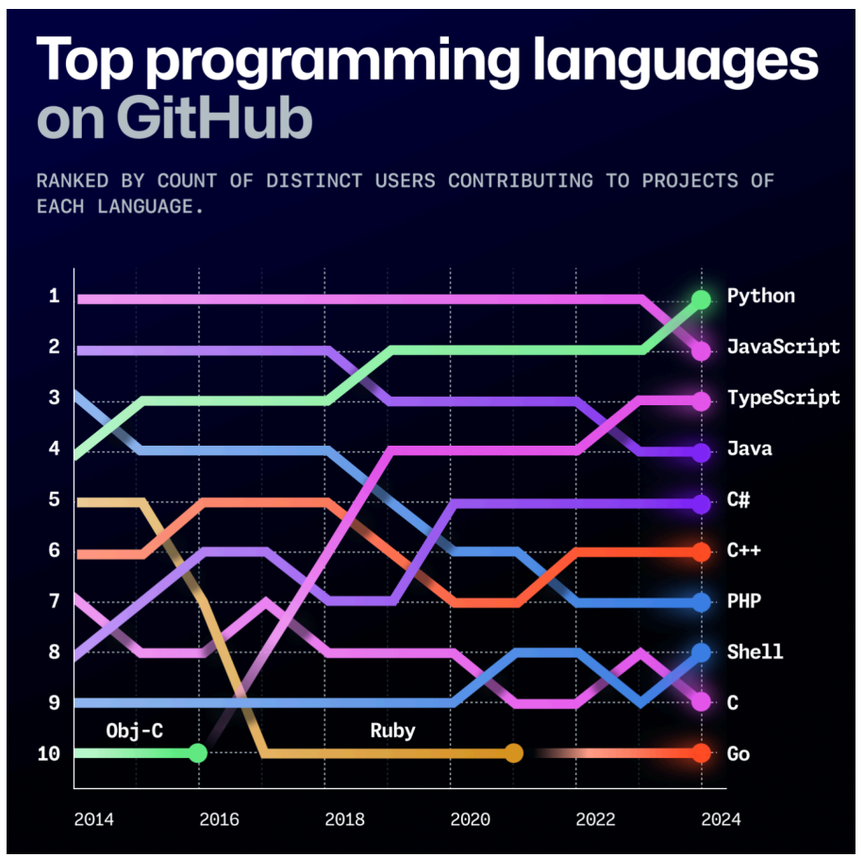
Hakjoo Oh
Korea University

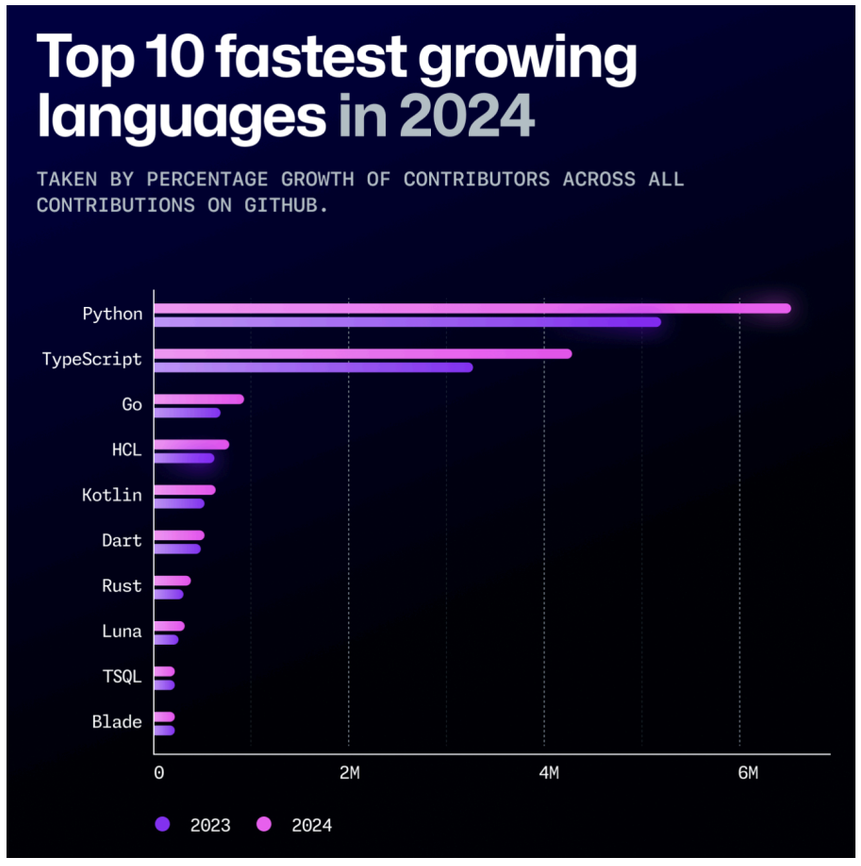
(w/ Seokhyun Lee, Jeongseop Lim, Jihyeok Park, Seokhyeon Moon, Yoonchan Jhi)

1-5 March 2026@IFIP WG 2.4 Meeting (Savannah)

Why Python?

- One of the most widely used and fastest-growing PL.
- Now mission-critical: AI agents, robotics, healthcare, etc.





Need for program analysis for correctness, safety, and speed

Challenge

- Developing program analysis for Python is challenging.

  (1) **Complex**: Supporting its diverse features and corner cases requires huge engineering effort.

  (2) **Implicit**: Even seemingly simple operations can trigger intricate dynamic behaviors.

  (3) **Informal**: No formal spec. Tool developers must rely on incomplete docs or examine the CPython implementation.

- Consequence: Building a sound and maintainable program analysis for Python remains difficult.

Example

- Consider a simple expression: **`a + b`**.

(1) Let `A` and `B` denote the types of `a` and `b`, respectively. If `B` is a subclass of `A` and `__radd__` is defined in class `B`, then Python calls `B.__radd__(b, a)`.
(2) If the above conditions do not hold, or if the call returns `NotImplemented`, and `__add__` is defined in `A`, then Python calls `A.__add__(a, b)`.
(3) If `A.__add__` is undefined or returns `NotImplemented`, and `__radd__` is defined in `B`, then `B.__radd__(b, a)` is called (even if `B` is not a subclass of `A`).
(4) If `__radd__` is undefined or returns `NotImplemented`, a `TypeError` is raised.

- Call expression **`f(x,...)`** behaves differently depending on whether **`f`** is a function, a callable object, or neither:

(1) If `f` is a function, the body of the function is executed.
(2) If `f` is not a function and it is not a callable object (the type of `f` does not define `__call__`), a `TypeError` is raised.
(3) If `f` is not a function but defines `__call__`, and `__call__` is a function, then `__call__(f, x, ...)` is evaluated by executing the body of the function.
(4) If `f` is not a function, defines `__call__`, and `__call__` is not a function, then `__call__(x, ...)` is evaluated recursively by going back to step 2.

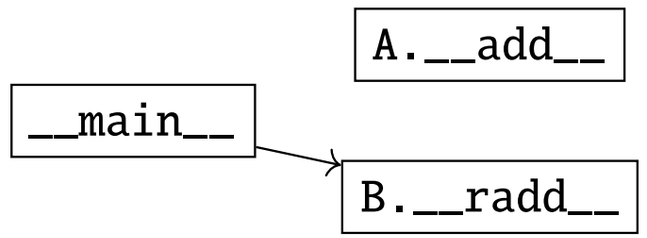
Even analyzing `a+b` involves more than a dozen semantic cases

Limitation of Existing Analysis Tools

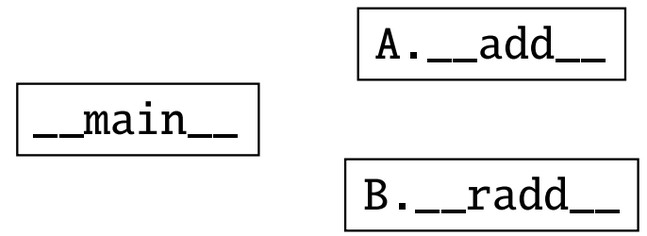
- Most, if not all, existing program analysis tools built on top of Python AST fail to correctly analyze Python programs.

- **Example 1**: Static call graph analysis (pointer analysis)

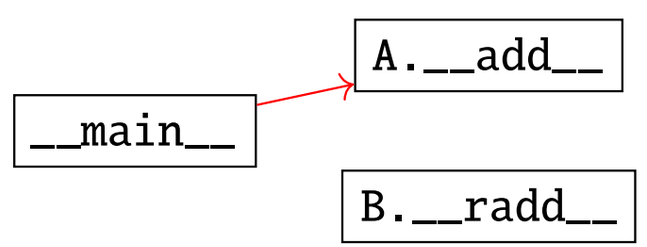
```
1  class A:
2      def __add__(self, other):
3          return self
4
5  class B(A):
6      def __radd__(self, other):
7          return self
8
9  a = A()
10 b = B()
11 assert (a + b) == a # False
```



Ground-truth



PyCG (ICSE 2021*)



Pyright (Microsoft)

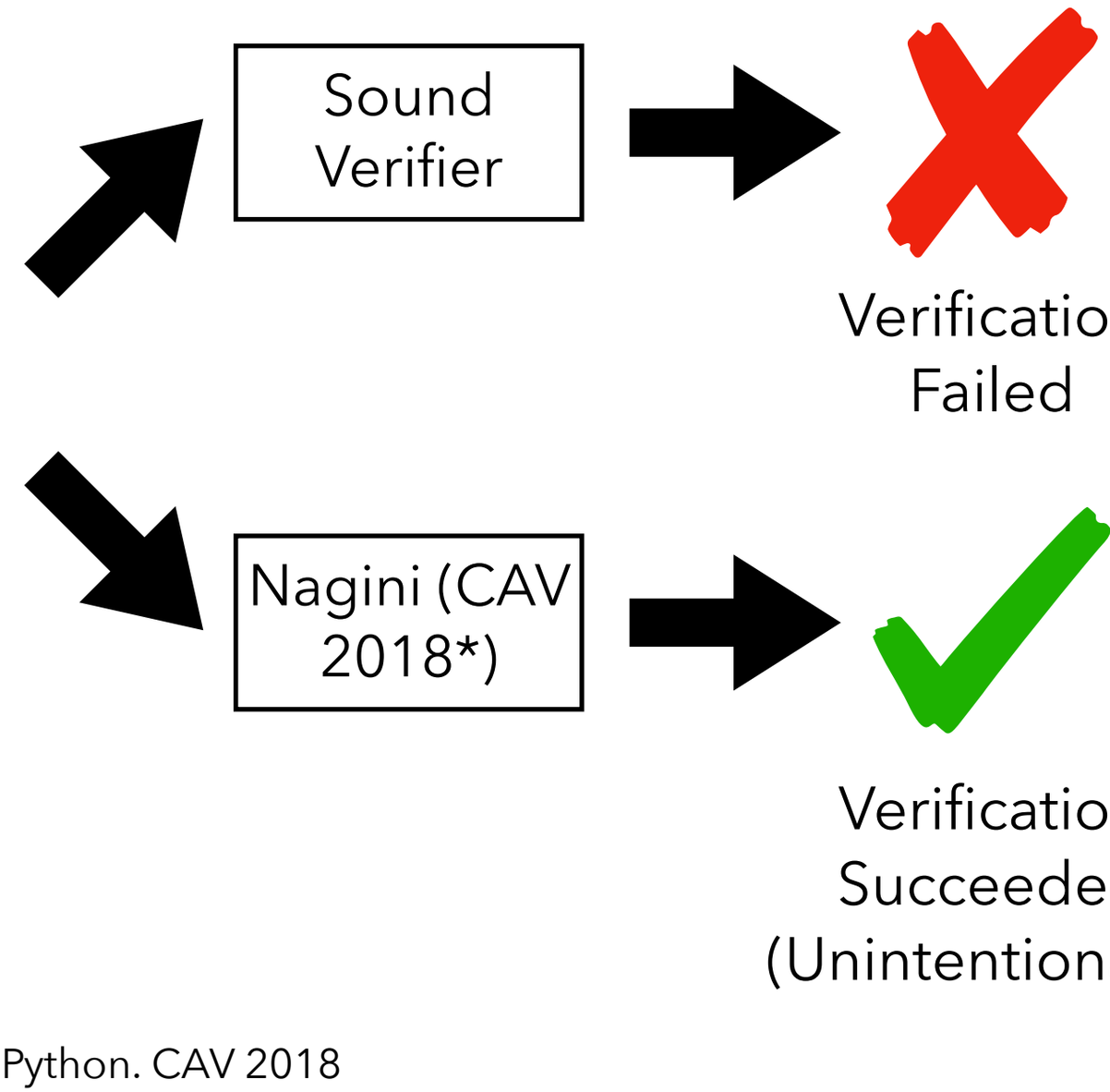
* Salis et al. PyCG: Practical Call Graph Generation in Python. ICSE 2021

Limitation of Existing Analysis Tools

- **Example 2**: Building a sound program verifier on top of Python AST is notoriously difficult, e.g.,

```
1  class A:
2      def __add__(self, other):
3          return self
4
5  class B(A):
6      def __radd__(self, other):
7          return self
8
9  a = A()
10 b = B()
11 assert (a + b) == a # False
```

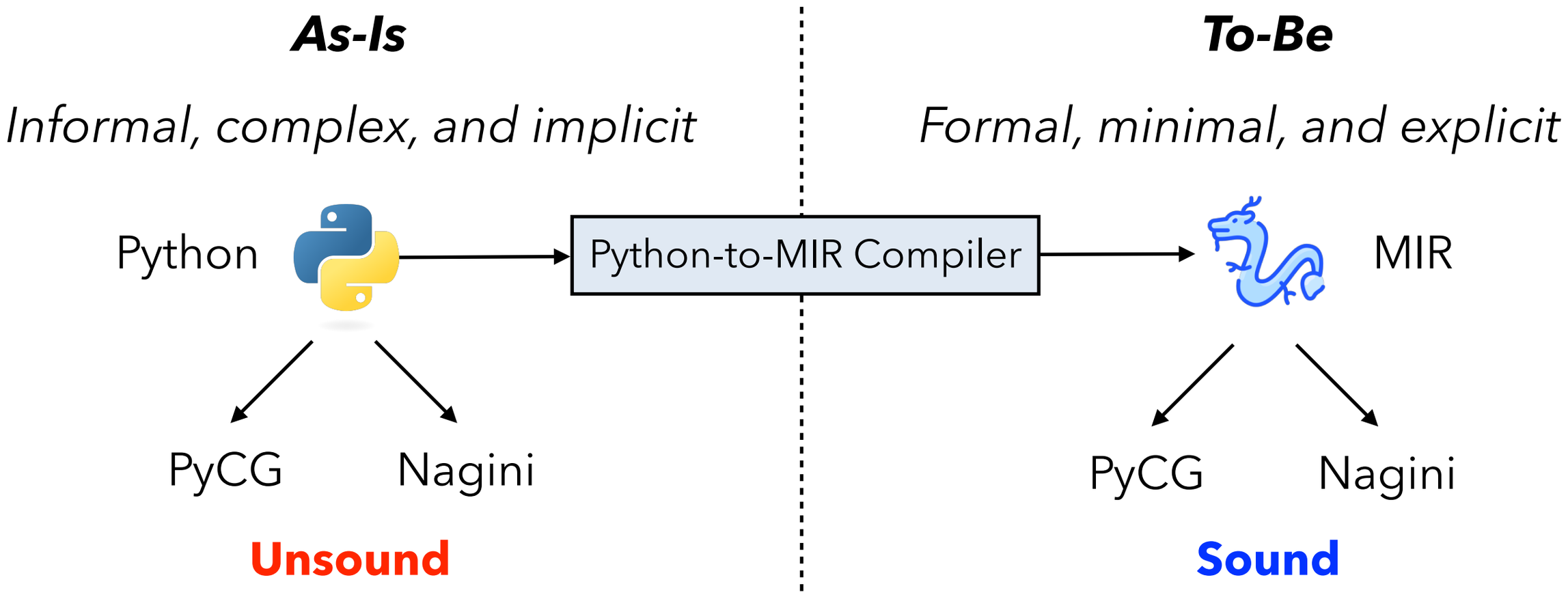
"Buggy" program



* Marco Eilers and Peter Müller. Nagini: A Static Verifier for Python. CAV 2018

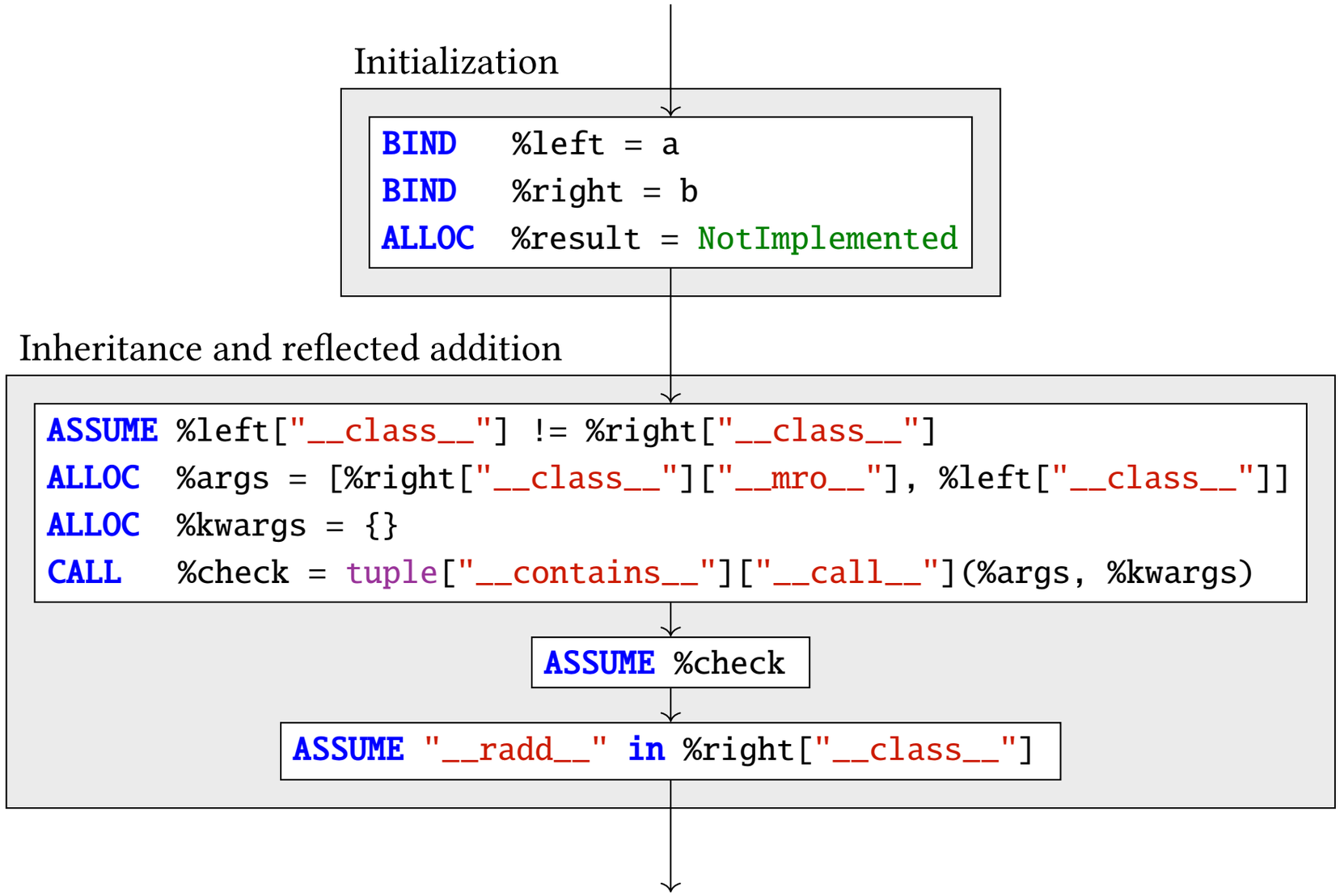
Goal: Easy and Rigorous Python Analysis

- Our Solution: MIR, a formal, minimal, and explicit IR.
  - **Formal**: Precise, mathematical reasoning is now possible
  - **Minimal**: Frees developers from huge implementation burden
  - **Explicit**: Eliminates subtleties that complicate static reasoning



Example

- How **`a + b`** is translated into MIR (excerpted):



MIR: Syntax and Design Principles

- Syntax: 7 core instructions with side-effect-free expressions

$$
\begin{array}{lcll}
I & \to & \textbf{skip} & \text{(skip)} \\
 & | & \textbf{assume}(E) & \text{(prune / assume)} \\
 & | & \textbf{alloc}(x, E) & \text{(allocation)} \\
 & | & \textbf{bind}(L_1, L_2) & \text{(binding / aliasing)} \\
 & | & \textbf{env}(x) & \text{(environment)} \\
 & | & \textbf{del}(L) & \text{(delete reference)} \\
 & | & \textbf{call}(x, E, L_1, L_2) & \text{(function call)} \\
L & \to & x & \text{(variable)} \\
 & | & L[E] & \text{(attribute access)}
\end{array}
$$

$$
\begin{array}{lcll}
E & \to & \{\} & \text{(new object)} \\
 & | & b \mid n \mid s & \text{(literals)} \\
 & | & \lambda(a, k, p).(G, z) & \text{(lambda)} \\
 & | & [L_1, \cdots, L_n] & \text{(list literal)} \\
 & | & \{x_1 : L_1, \cdots, x_n : L_n\} & \text{(dict literal)} \\
 & | & L & \text{(l-value)} \\
 & | & \neg E & \text{(logical negation)} \\
 & | & \textbf{len}(E) & \text{(size / length)} \\
 & | & E_1 \oplus E_2 & \text{(binary operation)} \\
\oplus & \to & + \mid - \mid = \mid < \mid \vee \mid \textbf{in} & \text{(binary operators)}
\end{array}
$$

- Key design principles
  - Separation of memory allocation and binding
  - Unified representation of function calls
  - Faithful modeling of objects
  - Environments as first-class values

Operational Semantics

$$
\begin{array}{rcll}
x & \in & \text{Id} & = \mathbb{S} \\
\rho & \in & \text{Env} & = \text{Id} \to \text{Addr} \\
\sigma & \in & \text{Memory} & = \text{Addr} \to \text{Val} \\
\alpha & \in & \text{Val} & = \text{Primitive} + \text{List} + \text{Dict} + \text{Obj} + \text{Closure} + \text{Env} \\
\pi & \in & \text{Primitive} & = \mathbb{B} + \mathbb{Z} + \mathbb{S} \\
o & \in & \text{Obj} & = \text{Id} \to \text{Addr} \\
l & \in & \text{List} & = \text{Addr}^* \\
d & \in & \text{Dict} & = \text{Id} \to \text{Addr} \\
\langle \lambda(a,k,p).(G,z), \ell_{Env} \rangle & \in & \text{Closure} & = \text{Id} \times \text{Id} \times \text{Id} \times \text{Cfg} \times \text{Id} \times \text{Addr}_{\text{Env}} \\
\ell & \in & \text{Addr} & \text{(memory addresses)} \\
\ell_{Env} & \in & \text{Addr}_{\text{Env}} & \text{(addresses that store environments)}
\end{array}
$$

Semantic domain

$$\text{I-Skip}\ \frac{}{\ell_{Env}, \sigma \vdash \textbf{skip} \Longrightarrow \sigma} \qquad \text{I-Assume}\ \frac{\ell_{Env}, \sigma \vdash E \Downarrow \textbf{true}}{\ell_{Env}, \sigma \vdash \textbf{assume}(E) \Longrightarrow \sigma}$$

$$\text{I-Alloc}\ \frac{\ell' \notin Dom(\sigma) \quad \ell_{Env}, \sigma \vdash E \Downarrow \alpha}{\ell_{Env}, \sigma \vdash \textbf{alloc}(x, E) \Longrightarrow \text{EnvUpdate}([\, \ell' \mapsto \alpha \,]\sigma, \ell_{Env}, x, \ell')}$$

$$\text{I-BindVar}\ \frac{\ell_{Env}, \sigma \vdash L \hookrightarrow \ell}{\ell_{Env}, \sigma \vdash \textbf{bind}(x, L) \Longrightarrow \text{EnvUpdate}(\sigma, \ell_{Env}, x, \ell)}$$

$$\text{I-BindAttr}\ \frac{\ell_{Env}, \sigma \vdash L_1 \hookrightarrow \ell_1 \quad \ell_{Env}, \sigma \vdash E \Downarrow \pi \quad \ell_{Env}, \sigma \vdash L_2 \hookrightarrow \ell_2}{\ell_{Env}, \sigma \vdash \textbf{bind}(L_1[E], L_2) \Longrightarrow \text{ConUpdate}(\sigma, \ell_1, \pi, \ell_2)}$$

$$\text{I-EnvRef}\ \frac{}{\ell_{Env}, \sigma \vdash \textbf{env}(x) \Longrightarrow \text{EnvUpdate}(\sigma, \ell_{Env}, x, \ell_{Env})}$$

$$\text{I-DelVar}\ \frac{\sigma(\ell_{Env}) = \rho \quad x \in Dom(\rho)}{\ell_{Env}, \sigma \vdash \textbf{del}(x) \Longrightarrow [\, \ell_{Env} \mapsto \rho \setminus \{x\} \,]\sigma}$$

$$\text{I-DelList}\ \frac{\ell_{Env}, \sigma \vdash L \hookrightarrow \ell \quad \sigma(\ell) = l \quad \ell_{Env}, \sigma \vdash E \Downarrow n \quad n \le |l|}{\ell_{Env}, \sigma \vdash \textbf{del}(L[E]) \Longrightarrow [\, \ell \mapsto \langle l_1, \ldots, l_{n-1}, l_{n+1}, \ldots, l_{|l|} \rangle \,]\sigma}$$

$$\text{I-DelMap}\ \frac{\ell_{Env}, \sigma \vdash L \hookrightarrow \ell \quad \sigma(\ell) = \alpha \quad \alpha \in \text{Dict} \cup \text{Obj} \cup \text{Env} \quad \ell_{Env}, \sigma \vdash E \Downarrow s \quad s \in Dom(\alpha)}{\ell_{Env}, \sigma \vdash \textbf{del}(L[E]) \Longrightarrow [\, \ell \mapsto \alpha \setminus \{s\} \,]\sigma}$$

$$\text{I-Call}\ \frac{\begin{array}{c}\ell'_{Env} \notin Dom(\sigma) \quad \ell_{Env}, \sigma \vdash E \Downarrow \langle \lambda(a,k,p).(G,z), \ell''_{Env} \rangle \\ \ell_{Env}, \sigma \vdash L_1 \hookrightarrow \ell_1 \quad \ell_{Env}, \sigma \vdash L_2 \hookrightarrow \ell_2 \quad \ell'_{Env}, [\, \ell'_{Env} \mapsto [\, a \mapsto \ell_1,\ k \mapsto \ell_2,\ p \mapsto \ell''_{Env} \,] \,]\sigma \vdash G \rightsquigarrow \sigma'\end{array}}{\ell_{Env}, \sigma \vdash \textbf{call}(x, E, L_1, L_2) \Longrightarrow \text{EnvUpdate}(\sigma', \ell_{Env}, x, \sigma'(\ell'_{Env})(z))}$$

Semantics of commands

$$\text{E-NewObj}\ \frac{}{\ell_{Env}, \sigma \vdash \{\} \Downarrow [\,]} \qquad \text{E-Bool}\ \frac{}{\ell_{Env}, \sigma \vdash b \Downarrow b} \qquad \text{E-Int}\ \frac{}{\ell_{Env}, \sigma \vdash n \Downarrow n} \qquad \text{E-String}\ \frac{}{\ell_{Env}, \sigma \vdash s \Downarrow s}$$

$$\text{E-Lambda}\ \frac{}{\ell_{Env}, \sigma \vdash \lambda(a,k,p).(G,z) \Downarrow \langle \lambda(a,k,p).(G,z), \ell_{Env} \rangle} \qquad \text{E-List}\ \frac{\ell_{Env}, \sigma \vdash L_1 \hookrightarrow \ell_1 \ \cdots\ \ell_{Env}, \sigma \vdash L_n \hookrightarrow \ell_n}{\ell_{Env}, \sigma \vdash [L_1, \cdots, L_n] \Downarrow \langle \ell_1, \cdots, \ell_n \rangle}$$

$$\text{E-Map}\ \frac{\ell_{Env}, \sigma \vdash L_1 \hookrightarrow \ell_1 \quad \cdots \quad \ell_{Env}, \sigma \vdash L_n \hookrightarrow \ell_n}{\ell_{Env}, \sigma \vdash \{x_1 : L_1, \cdots, x_n : L_n\} \Downarrow [x_1 \mapsto \ell_1, \cdots, x_n \mapsto \ell_n]} \qquad \text{E-Deref}\ \frac{\ell_{Env}, \sigma \vdash L \hookrightarrow \ell}{\ell_{Env}, \sigma \vdash L \Downarrow \sigma(\ell)}$$

$$\text{E-SizeList}\ \frac{\ell_{Env}, \sigma \vdash E \Downarrow l}{\ell_{Env}, \sigma \vdash \textbf{len}(E) \Downarrow |l|} \qquad \text{E-SizeMap}\ \frac{\ell_{Env}, \sigma \vdash E \Downarrow \alpha \quad \alpha \in \text{Dict} \cup \text{Obj} \cup \text{Env}}{\ell_{Env}, \sigma \vdash \textbf{len}(E) \Downarrow |Dom(\alpha)|}$$

$$\text{E-AddInt}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow n_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow n_2}{\ell_{Env}, \sigma \vdash E_1 + E_2 \Downarrow n_1 + n_2} \qquad \text{E-AddStr}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow s_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow s_2}{\ell_{Env}, \sigma \vdash E_1 + E_2 \Downarrow s_1 s_2}$$

$$\text{E-AddList}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow l_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow l_2}{\ell_{Env}, \sigma \vdash E_1 + E_2 \Downarrow l_1 +\!\!+ l_2} \qquad \text{E-EqBool}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow b_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow b_2}{\ell_{Env}, \sigma \vdash E_1 = E_2 \Downarrow b_1 = b_2}$$

$$\text{E-EqInt}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow n_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow n_2}{\ell_{Env}, \sigma \vdash E_1 = E_2 \Downarrow n_1 = n_2} \qquad \text{E-EqStr}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow s_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow s_2}{\ell_{Env}, \sigma \vdash E_1 = E_2 \Downarrow s_1 = s_2}$$

$$\text{E-EqObj}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow o_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow o_2}{\ell_{Env}, \sigma \vdash E_1 = E_2 \Downarrow Dom(o_1) = Dom(o_2) \wedge \forall x \in Dom(o_1).o_1(x) = o_2(x)}$$

$$\text{E-LtInt}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow n_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow n_2}{\ell_{Env}, \sigma \vdash E_1 < E_2 \Downarrow n_1 < n_2} \qquad \text{E-Or}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow b_1 \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow b_2}{\ell_{Env}, \sigma \vdash E_1 \vee E_2 \Downarrow b_1 \vee b_2}$$

$$\text{E-InMap}\ \frac{\ell_{Env}, \sigma \vdash E_1 \Downarrow s \quad \ell_{Env}, \sigma \vdash E_2 \Downarrow \alpha \quad \alpha \in \text{Dict} \cup \text{Obj} \cup \text{Env}}{\ell_{Env}, \sigma \vdash E_1 \textbf{ in } E_2 \Downarrow s \in Dom(\alpha)} \qquad \text{E-Not}\ \frac{\ell_{Env}, \sigma \vdash E \Downarrow b}{\ell_{Env}, \sigma \vdash \neg E \Downarrow \neg b}$$

Semantics of expressions

Language Coverage

Python Features	Examples	Supported
Literals	1, "hello", ...	Yes (with simplification)
Operations	1 + 2, "hello" == "world", ...	Yes
Containers	[1, 2, 3], d["key"], ...	Yes
Variables	x, x, y = 1, 2, ...	Yes
Control Flows	if, while, ...	Yes
Functions	def, f(x), @dec, ...	Yes
Iterators	yield, __next__, ...	Yes
Objects	obj.x, class, ...	Yes (with simplification)
Exceptions	raise, try, ...	Yes
Comprehensions	[x for ...], {x: y for ...}, ...	Yes
Imports	import lib, from lib import f, ...	Yes (with simplification)
Scoping	global, nonlocal	Yes
Context Manager	with	Yes
Dynamic Features	eval(), getattr(), ...	Yes
Asynchronous Features	async, await	No

Evaluation

- Research questions
  - **Correctness**: Is MIR expressive enough to cover Python language features, and can the MIR compiler correctly translate Python programs to MIR?
  - **Effectiveness**: Can it enable more rigorous analyses while reducing the engineering effort of building such tools?

Correctness

- Benchmarks
  - **MIR benchmarks**: 235 Python programs covering various features—especially corner cases
  - **MBPP and HumanEval**: 735 programs after filtering out the programs that use built-in functions that our MIR interpreter does not currently support
- Checked correctness via differential testing against CPython

Benchmark	Total	Applicable	Compiled	Executed	Correct
Mir benchmark	235	235	235	235	235
MBPP	974	619	619	619	619
HumanEval	163	116	116	116	116
Total	1,372	970	970	970	970

Effectiveness

- **MirCG**: Reimplemented version of PyCG (ICSE 2021) on top of MIR without modifying its underlying algorithm
  - Flow-, path-, and context-insensitive pointer analysis
- Compared MirCG and PyCG on two benchmarks:
  - **MirCG benchmark**: 100 Python programs designed to cover various scenarios of function calls, along with ground-truth call graphs
  - **Micro benchmarks** used in the PyCG paper: 112 programs with ground truth call graphs

Effectiveness

Benchmark	Total	MirCG			PyCG [36]		
		Sound	Complete	Time (s)	Sound	Complete	Time (s)
MirCG benchmark	100	**73**	79	30.7	19	73	7.7
Micro-benchmark	112	**78**	101	29.6	46	107	8.7
Total	212	**151 (71.2%)**	180 (84.9%)	60.3	65 (30.7%)	180 (84.9%)	16.4

- Sound: Count if all edges in the ground truth graph are present in the generated graph (no false negatives)
- Complete: Count if call edges in the generated graph are present in the ground truth graph (no false positives)

Migrating the PyCG algorithm from AST-based implementation to **MIR increases analysis soundness by 132%** (from 65 to 151)

Effectiveness

- Currently, MirCG is 4x slower than PyCG
  - This is because we currently prioritize correctness over optimization
  - Optimization plan:
    1. Optimizing the generated MIR code: ~2x
    2. Optimizing the compiler implementation: ~1.5x
    3. Optimizing the MirCG implementation: ~1.7x

~5x speed-up
after optimization

- MIR also reduced implementation cost
  - MirCG: 1,266 lines of code, developed in one week by a student
  - PyCG: 2,589 lines of code

Summary

- **Problem**: Python program analysis is hard due to the informal, complex, and implicit semantics.
- **Solution**: A formal, minimal, and explicit intermediate representation for Python.
- **Result**: Atop MIR, program analysis becomes more rigorous and simpler to develop.
- **Future work**: Optimization, abstract interpretation framework, program logic, cross-language translation, etc

Thank you!