Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries

Raphaël Monat

PhD defense
22 November 2021
Introduction
Software is ubiquitous

- Transportation
Software is ubiquitous

- Transportation
Software is ubiquitous

- Transportation
Software is ubiquitous

- Transportation
- Communication
Software

Software is ubiquitous

- Transportation
- Communication
- ...

Bugs

When a program does not work as intended
Software verification

Cheap approach: test $S[\text{prog}]$.

Some bugs may go undetected!

Would there be a way to automatically prove programs correct?
Software verification

$S[prog]$  Bad states

Cheap approach: test prog. Some bugs may go undetected!

Would there be a way to automatically prove programs correct?
Software verification

Cheap approach: test \textit{prog}.
Software verification

Cheap approach: test \textit{prog}.
Software verification

Cheap approach: test \texttt{prog}.

Bad states

\( \mathcal{S}[\texttt{prog}] \)
Software verification

Cheap approach: test $prog$.

Bad states
Cheap approach: test $prog$. Some bugs may go undetected! Would there be a way to automatically prove programs correct?
Software verification

Cheap approach: test \( S[\text{prog}] \).
Some bugs may go undetected!
Software verification

Cheap approach: test \textit{prog}.
Some bugs may go undetected!

Would there be a way to automatically prove programs correct?
Rice’s theorem (1953)
All non-trivial semantic properties of programs are undecidable.
Rice’s theorem (1953)

All non-trivial semantic properties of programs are undecidable.
It is impossible to have a system which is:
- Automatic: no user interaction required, terminates in finite time.
Rice’s theorem (1953)

All non-trivial semantic properties of programs are undecidable. It is impossible to have a system which is:

- Automatic: no user interaction required, terminates in finite time.
- Sound: derived properties are true on the program.
### Rice’s theorem (1953)

All non-trivial semantic properties of programs are undecidable.

It is impossible to have a system which is:
- **Automatic**: no user interaction required, terminates in finite time.
- **Sound**: derived properties are true on the program.
- **Complete**: all properties of the program can be derived.
Rice’s theorem (1953)

All non-trivial semantic properties of programs are undecidable.
It is impossible to have a system which is:
▶ Automatic: no user interaction required, terminates in finite time.
▶ Sound: derived properties are true on the program.
▶ Complete: all properties of the program can be derived.

Mitigating Rice’s theorem

Our choice: sound and automatic approaches.
Aimed at certifying programs correct.
Abstract interpretation – the big picture

\[ \mathcal{S}[\text{prog}] \]

\[ \mathcal{D} \text{ (concrete)} \]

Bad states

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

Program proved safe

\[ S[\text{prog}] \]  
\[ \mathcal{D} \text{ (concrete)} \]

\[ S\#[\text{prog}] \]  
\[ \mathcal{D}\# \text{ (abstract)} \]

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

\[ S[\text{prog}] \]

\[ D (\text{concrete}) \]

\[ S^\#[\text{prog}] \]

\[ D^\# (\text{abstract}) \]

\[ \gamma \]

Bad states

True alarm

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

\[ S[\text{prog}] \]

\( \mathcal{D} \) (concrete)

Bad states

\( \gamma \)

\[ S[#][\text{prog}] \]

\( \mathcal{D}^\# \) (abstract)

Bad states

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

\[ S[\text{prog}] \]

Bad states

\[ \mathcal{D} \text{ (concrete)} \]

\[ S^\#[\text{prog}] \]

Bad states

\[ \mathcal{D}^\# \text{ (abstract)} \]

False alarm (Abstraction too coarse)

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

$\mathcal{D}$ (concrete)  

$\mathcal{S}[\text{prog}]$

Bad states

$\mathcal{D}^\#$ (abstract)  

$\mathcal{S}^\#[\text{prog}]$

Bad states

Unsound analysis  
(shouldn’t happen)

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub

https://octoverse.github.com/#top-languages
Dynamic programming languages

Growing popularity
JavaScript #1, Python #2 on GitHub¹

New features
► Object orientation,

¹https://octoverse.github.com/#top-languages
Dynamic programming languages

Growing popularity
JavaScript #1, Python #2 on GitHub¹

New features
- Object orientation,
- Dynamic typing,

¹https://octoverse.github.com/#top-languages
## Dynamic programming languages

### Growing popularity

JavaScript #1, Python #2 on GitHub

### New features

- Object orientation,
- Dynamic typing,
- Dynamic object structure,

---

1[https://octoverse.github.com/#top-languages](https://octoverse.github.com/#top-languages)
Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub¹

New features

- Object orientation,
- Dynamic typing,
- Dynamic object structure,
- Introspection operators,

¹https://octoverse.github.com/#top-languages
Dynamic programming languages

Growing popularity
JavaScript #1, Python #2 on GitHub

New features
▶ Object orientation,
▶ Dynamic typing,
▶ Dynamic object structure,
▶ Introspection operators,
▶ eval.

¹https://octoverse.github.com/#top-languages
State of the art

Well-established & industrialized analysis of static programming languages

- Java: Julia (2010)
State of the art

### Well-established & industrialized analysis of static programming languages

- **C**: Polyspace (1999), Astrée (2003), Frama-C (2008)
- **Java**: Julia (2010)

### Around JavaScript

- **First**: Jensen, Møller, and Thiemann. “Type Analysis for JavaScript”. SAS 2009
- **Bodin et al.** “A trusted mechanised JavaScript specification”. POPL 2014
## State of the art

### Well-established & industrialized analysis of static programming languages
- **C:** Polyspace (1999), Astrée (2003), Frama-C (2008)
- **Java:** Julia (2010)

### Around JavaScript
- **First:** Jensen, Møller, and Thiemann. “Type Analysis for JavaScript”. SAS 2009
- **Bodin et al.** “A trusted mechanised JavaScript specification”. POPL 2014

### Why Python?
- Used a lot in
  - Scientific computing
  - Scripts and automation
Outline

1. Introduction
2. A Taste of Python
3. Analyzing Python Programs
4. Analyzing Python Programs with C Libraries
5. Conclusion
A Taste of Python
## Python’s specificities

### No standard

- CPython is the reference
  - → manual inspection of the source code and handcrafted tests
Python’s specificities

No standard

► CPython is the reference

⇒ manual inspection of the source code and handcrafted tests

Operator redefinition

► Calls, additions, attribute accesses

► Operators eventually call overloaded `__methods__`

```python
class Protected:
    def __init__(self, priv):
        self._priv = priv
    def __getattribute__(self, attr):
        if attr[0] == "_": raise AttributeError("protected")
        return object.__getattribute__(self, attr)

a = Protected(42)
a._priv  # AttributeError raised
```
## Python’s specificities (II)

### Dual type system

- **Nominal (classes, MRO)**

### Exceptions

- Exceptions rather than specific values
  - `1 + "a"` → `TypeError`
  - `l[len(l) + 1]` → `IndexError`

### Fspath (from standard library)

```python
class Path:
    def __fspath__(self): return 42

def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        r = p.__fspath__()
        if isinstance(r, (str, bytes)):
            return r
        raise TypeError
    fspath("/dev" if random() else Path())
```

---

Python’s specificities (II)

Dual type system

- Nominal (classes, MRO)
- Structural (attributes)

Exceptions rather than specific values

- `1 + "a"` ↛ `TypeError`
- `l[len(l) + 1]` ↛ `IndexError`

Fspath (from standard library)

```python
class Path:
    def __fspath__(self): return 42

def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        r = p.__fspath__()
        if isinstance(r, (str, bytes)):
            return r
        raise TypeError

fspath("/dev" if random() else Path())
```

## Python’s specificities (II)

### Dual type system

- Nominal (classes, MRO)
- Structural (attributes)

### Exceptions

Exceptions rather than specific values

- `1 + "a"` ➞ `TypeError`
- `l[len(l) + 1]` ➞ `IndexError`

---

Fspath (from standard library)

```python
1 class Path:
2     def __fspath__(self): return 42
3
4 def fspath(p):
5     if isinstance(p, (str, bytes)):
6         return p
7     elif hasattr(p, "__fspath__"):
8         r = p.__fspath__()
9         if isinstance(r, (str, bytes)):
10            return r
11     raise TypeError
12
13 fspath("/dev" if random() else Path())
```

---

Guth. “A formal semantics of Python 3.3”. 2013
Implementation within the K framework.
Previous works on Python 3

Guth. “A formal semantics of Python 3.3”. 2013
Implementation within the K framework.

Politz et al. “Python: The full monty”. OOPSLA 2013
Complex desugaring into $\lambda_\pi$.
May incur losses of precision in the abstract interpreter.
### Previous works on Python 3

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guth</td>
<td>“A formal semantics of Python 3.3”.</td>
<td>2013</td>
</tr>
<tr>
<td></td>
<td>Implementation within the K framework.</td>
<td></td>
</tr>
<tr>
<td>Politz et al.</td>
<td>“Python: The full monty”.</td>
<td>OOPSLA 2013</td>
</tr>
<tr>
<td></td>
<td>Complex desugaring into $\lambda_\pi$.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>May incur losses of precision in the abstract interpreter.</td>
<td></td>
</tr>
<tr>
<td>Köhl</td>
<td>“An Executable Structural Operational Formal Semantics for Python”.</td>
<td>2021</td>
</tr>
<tr>
<td></td>
<td>Semantics of Python, using a Python framework, developed concurrently.</td>
<td></td>
</tr>
</tbody>
</table>
Previous works on Python 3

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guth</td>
<td>“A formal semantics of Python 3.3”</td>
<td>2013</td>
</tr>
<tr>
<td></td>
<td>Implementation within the K framework.</td>
<td></td>
</tr>
<tr>
<td>Politz et al.</td>
<td>“Python: The full monty”</td>
<td>2013</td>
</tr>
<tr>
<td></td>
<td>Complex desugaring into $\lambda_\pi$.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>May incur losses of precision in the abstract interpreter.</td>
<td></td>
</tr>
<tr>
<td>Köhl</td>
<td>“An Executable Structural Operational Formal Semantics for Python”</td>
<td>2021</td>
</tr>
<tr>
<td></td>
<td>Semantics of Python, using a Python framework, developed concurrently.</td>
<td></td>
</tr>
</tbody>
</table>

Different goal

These works focus on the concrete semantics. This is not our endgoal.
Previous works on Python 3

Guth. “A formal semantics of Python 3.3”. 2013
Implementation within the K framework.

Politz et al. “Python: The full monty”. OOPSLA 2013
Complex desugaring into $\lambda\pi$.
May incur losses in the abstract interpreter.

Semantics of Python, using a Python framework, developed concurrently.

Different goal
These works focus on the concrete semantics. This is not our endgoal.

Moving to our own semantics
- Cost of understanding the code (vs CPython)
- Trust in the code (CPython’s tests?)
- Insights of the papers
Our approach

Interpreter-like semantics
Easily convertible to an abstract interpreter.

---

Our approach

**Interpreter-like semantics**
Easily convertible to an abstract interpreter.

**Major extension of the work of Fromherz, Ouadjaout, and Miné**

- Separation between core and builtins
- $2.3 \times$ more cases (*with* statement, bidirectional generators, ...)
- Improved some cases (+, boolean casts of conditionals, data descriptors, ...)

---

Our approach

**Interpreter-like semantics**
Easily convertible to an abstract interpreter.

**Major extension of the work of Fromherz, Ouadjaout, and Miné**
- Separation between core and builtins
- $2.3 \times$ more cases (with statement, bidirectional generators, ...)
- Improved some cases (+, boolean casts of conditionals, data descriptors, ...)

**Correctness**
- Strived to make it auditable (with links to the source).
- Tested only through the abstract analysis yet (no concrete execution).

---

Example – attribute access

\[ \mathcal{E}_{\text{cur}[\text{x}.\text{s}]}(\text{cur}, \text{e}, \text{h}) \overset{\text{def}}{=} \text{LOAD\_ATTR } \text{PyObject\_GetAttr} \left( \text{slot\_tp\_getattr\_hook} \right) \]

- \text{letb} (\text{cur}, \text{e}, \text{h}), \@_{\text{x}} = \mathcal{E}[\text{x}](\text{cur}, \text{e}, \text{h}) \text{ in}
- \text{letb} (\text{cur}, \text{e}, \text{h}), \@_{\text{c}} = \mathcal{E}[\text{mro\_search(type(\@_{\text{x}}),"\_\_getattribute\_\_"})](\text{cur}, \text{e}, \text{h}) \text{ in}
- \text{letcases} (\text{f}, \text{e}, \text{h}), \@_{\text{x}.\text{s}} = \mathcal{E}[\text{\@}_{\text{c}}(\@_{\text{x}}, \text{s})](\text{cur}, \text{e}, \text{h}) \text{ in}

match \text{f} \text{ with}
- \cdot \; \text{exn} \; \@_{\text{exc}} \text{ when } \text{isinstance}(\@_{\text{exc}}, \text{AttributeError}) \Rightarrow
  \text{let} (\text{f}, \text{e}, \text{h}), \@_{\text{d}} = \mathcal{E}[\text{mro\_search(type(\@_{\text{x}}),"\_\_getattr\_\_"})](\text{f}, \text{e}, \text{h}) \text{ in}
  \text{if } d \neq \perp \text{ then return } \mathcal{E}[\text{\@}_{\text{d}}(\@_{\text{x}}, \text{s})](\text{cur}, \text{e}, \text{h})
  \text{ else return } (\text{f}, \text{e}, \text{h}), \perp
- \cdot \Rightarrow \text{return } (\text{f}, \text{e}, \text{h}), \@_{\text{x}.\text{s}}
Example – attribute access

\[ E_{\text{cur}[x.s]}(\text{cur}, e, h) \overset{\text{def}}{=} \text{LOAD\_ATTR} \ \text{PyObject\_GetAttr} \ (\text{slot\_tp\_getattr\_hook}) \]

\[ \text{letb} \ (\text{cur}, e, h), \text{obj}_x = E[x](\text{cur}, e, h) \text{ in} \]

\[ E_{\text{object\._getattribute\_}}(\text{obj}, \text{name})(\text{cur}, e, h) \overset{\text{def}}{=} \text{tp\_field\ _PyObject\_GenericGetAttrWithDict} \]

\[ \text{letb} \ (\text{cur}, e, h), \text{obj}_o = E[\text{obj}](\text{cur}, e, h) \text{ in} \]

\[ \text{letb} \ (\text{cur}, e, h), \text{obj}_n = E[\text{name}](\text{cur}, e, h) \text{ in} \]

\[ \text{if} \ \neg \text{isinstance}(\text{obj}_n, \text{str}) \text{ then return } S[\text{raise TypeError}](\text{cur}, e, h), \bot \text{ else} \]

\[ \text{let str}(n) = \text{fst} \circ h(\text{obj}_n) \text{ in} \]

\[ \text{letcases} \ (f, e, h), \text{obj}_\text{descr} = E[\text{mro\_search}(\text{type}(\text{obj}_o), n)](f, e, h) \text{ in} \]

\[ \text{if} \ \text{obj}_\text{descr} \neq \bot \text{ then} \]

\[ \text{if} \ \text{hasattr(\text{type}(\text{obj}_\text{descr}),"\_\_get\_\_")}\land \]

\[ (\text{hasattr(\text{type}(\text{obj}_\text{descr}),"\_\_set\_\_")}) \lor \text{hasattr(\text{type}(\text{obj}_\text{descr}),"\_\_delete\_\_")}) \text{ then} \]

\[ \text{return } E[\text{type}(\text{obj}_\text{descr})\_\_get\_\_](\text{obj}_\text{descr}, \text{obj}_o, \text{type}(\text{obj}_o))(f, e, h) \]
Example – attribute access

\[
E_{\text{cur}[x.s]}(\text{cur}, e, h) \overset{\text{def}}{=} \text{LOADATTR \ PyObject\_GetAttr}(\text{slot\_tp\_getattr\_hook})
\]

\[
\begin{align*}
\text{letb} & \ (\text{cur}, e, h), \ O_x = E[x](\text{cur}, e, h) \text{ in} \\
\text{letb} & \ (\text{cur}, e, h), \ O_o = E[\text{obj}](\text{cur}, e, h) \text{ in} \\
\text{letb} & \ (\text{cur}, e, h), \ O_{\text{name}} = E[\text{name}](\text{cur}, e, h) \text{ in} \\
\text{letb} & \ (\text{cur}, e, h), \ O_{\text{meta}} = E[\text{mro\_search}(\text{type}(O_{\text{typ}}), O_{\text{name}})](\text{cur}, e, h) \text{ in}
\end{align*}
\]

\[
\begin{align*}
\text{if } O_{\text{meta}} \neq \bot \text{ then} \\
\text{if } \text{hasattr}(\text{type}(O_{\text{meta}}), "\_\_get\_\_") \lor \\
(\text{hasattr}(\text{type}(O_{\text{meta}}), "\_\_set\_\_") \lor \text{hasattr}(\text{type}(O_{\text{meta}}), "\_\_delete\_\_")) \text{ then}
\end{align*}
\]

\[
\begin{align*}
\text{E}_{\text{cur}[\text{object\_\_getattribute\_\_}]}(\text{obj}, name)](\text{cur}, e, h) \overset{\text{def}}{=} & \\
\text{tp\_field \ _PyObject\_GenericGetAttrWithDict} & \\
\text{letb} & \ (\text{cur}, e, h), \ O_{\text{typ}} = E[\text{typ}](\text{cur}, e, h) \text{ in} \\
\text{letb} & \ (\text{cur}, e, h), \ O_{\text{name}} = E[\text{name}](\text{cur}, e, h) \text{ in} \\
\text{letb} & \ (\text{cur}, e, h), \ O_{\text{meta}} = E[\text{mro\_search}(\text{type}(O_{\text{typ}}), O_{\text{name}})](\text{cur}, e, h) \text{ in}
\end{align*}
\]

\[
\begin{align*}
\text{if } O_{\text{meta}} \neq \bot \text{ then} \\
\text{if } \text{hasattr}(\text{type}(O_{\text{meta}}), "\_\_get\_\_") \lor \\
(\text{hasattr}(\text{type}(O_{\text{meta}}), "\_\_set\_\_") \lor \text{hasattr}(\text{type}(O_{\text{meta}}), "\_\_delete\_\_")) \text{ then}
\end{align*}
\]
Analyzing Python Programs
Goal

Detect runtime errors: uncaught raised exceptions
### Goal

Detect runtime errors: uncaught raised exceptions

### Supported constructs

Our analysis supports:

- Objects
- Exceptions
- Dynamic typing
- Introspection
- Permissive semantics
- Dynamic attributes
- Generators
- `super`
- Metaclasses

Unsupported constructs:

- Recursive functions
- `eval`
- Finalizers
Analysis | Overview

**Goal**
Detect runtime errors: uncaught raised exceptions

**Supported constructs**
Our analysis supports:
- Objects
- Exceptions
- Dynamic typing
- Introspection
- Permissive semantics
- Dynamic attributes
- Generators
- super
- Metaclases

**Unsupported constructs**
- Recursive functions
- eval
- Finalizers
Avering numbers

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m

l = [randint(0, 20)
     for i in range(randint(5, 10))]

m = average(l)
```
Avering numbers

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Proved safe?

- $m // (i+1)$
- $l[i]$
Avering numbers

```
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Stateless domains: \textit{list content}, \textit{list length}, \textit{int}, \textit{int}, \textit{task}

Environment abstraction

\[
m \mapsto \text{@int} \quad i \mapsto \text{@int} \quad \text{els}(l) \mapsto \text{@int}
\]

---

Proved safe?

\[
\begin{align*}
& \quad m \div (i+1) \\
& \quad l[i]
\end{align*}
\]
Analysis | Domains required

Avering numbers

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Proved safe?

- $m \div (i+1)$
- $l[i]$

Searching for a loop invariant (l. 4)

Stateless domains: list content,

**Environment abstraction**

$m \mapsto \#_{\text{int}}$  \ 
$i \mapsto \#_{\text{int}}$  \ 
$\text{els}(l) \mapsto \#_{\text{int}}$

**Numeric abstraction (intervals)**

$m \in [0, +\infty)$  \ 
$\text{els}(l) \in [0, 20]$  \ 
$i \in [0, +\infty)$
Analysis | Domains required

Avering numbers

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Stateless domains: list content, **list length**

Searching for a loop invariant (l. 4)

Proved safe?

- \(m // (i+1)\)
- \(l[i]\)

Environment abstraction

\[m \mapsto @int\quad i \mapsto @int\quad \text{els}(l) \mapsto @int\]

Numeric abstraction (intervals)

\[m \in [0, +\infty)\quad \text{els}(l) \in [0, 20]\]
\[\text{len}(l) \in [5, 10]\quad i \in [0, 10]\]
Avering numbers

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Proved safe?

- \( m \div (i+1) \)
- \( l[i] \)

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

\[ m \mapsto \mathbb{Z}, \quad i \mapsto \mathbb{Z}, \quad \text{els}(l) \mapsto \mathbb{Z} \]

Numeric abstraction (polyhedra)

\[ m \in [0, +\infty), \quad \text{els}(l) \in [0, 20], \quad 0 \leq i < \text{len}(l), \quad 5 \leq \text{len}(l) \leq 10 \]
Analysis | Domains required

Averaging tasks

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

    def average(l):
        m = 0
        for i in range(len(l)):
            m = m + l[i].weight
        m = m // (i + 1)
        return m

l = [Task(randint(0, 20))
     for i in range(randint(5, 10))]

m = average(l)
```

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

\[
\begin{align*}
  m &\mapsto \int, \
  i &\mapsto \int, \
  \text{els}(l) &\mapsto \text{Task} \\
  \text{Task} \cdot \text{weight} &\mapsto \int
\end{align*}
\]

Numeric abstraction (polyhedra)

\[
\begin{align*}
  m &\in [0, +\infty) \\
  0 &\leq i < \text{len}(l) \\
  5 &\leq \text{len}(l) \leq 10 \\
  0 &\leq \text{Task} \cdot \text{weight} \leq 20
\end{align*}
\]

Attributes abstraction

\[
\begin{align*}
  \text{Task} &\mapsto (\{\text{weight}\}, \emptyset)
\end{align*}
\]

Proved safe?

- m // (i+1)
- l[i].weight
Analysis | Domains required

Averaging tasks

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i].weight
    m = m // (i + 1)
    return m

l = [Task(randint(0, 20)) for i in range(randint(5, 10))]
m = average(l)
```

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

\[ m \mapsto \#\text{Task} \]

\[ i \mapsto \#\text{int} \]

\[ i \mapsto \#\text{int} \]

\[ l \mapsto \#\text{list} \]

\[ \text{Task} \cdot \text{weight} \mapsto \#\text{int} \]

Conclusion

- Different domains depending on the precision
- Use of auxiliary variables (underlined)

Attributes abstraction

\[ \#\text{Task} \mapsto (\{\text{weight}\}, \emptyset) \]

Proved safe?

- \[ m \div (i+1) \]
- \[ l[i].weight \]
Modular Open Platform for Static Analysis

Modular Open Platform for Static Analysis\textsuperscript{3}

▶ One AST to analyze them all
- Flag: Multilanguage support
- FILE-CODE: Expressiveness
- RECYCLE: Reusability

\textsuperscript{3} Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. VSTTE 2019.
Modular Open Platform for Static Analysis\(^3\)

- One AST to analyze them all
  - Multilanguage support
  - Expressiveness
  - Reusability

- Unified domain signature
  - Semantic rewriting
  - Loose coupling
  - Observability

---
\(^3\) Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. VSTTE 2019.
Modular Open Platform for Static Analysis

▶ One AST to analyze them all
- Multilanguage support
- Expressiveness
- Reusability

▶ Unified domain signature
- Semantic rewriting
- Loose coupling
- Observability

▶ DAG of abstract domains
- Composition
- Cooperation

---

for(init; cond; incr) body

C.iterators.loops

Rewrite and analyze recursively

init;
while(cond) {
    body;
    incr;
}
clean init

for target in iterable: body

Python.Desugar.Loops
- Rewrite and analyze recursively
- Optimize for some semantic cases
it = iter(iterable)
while(1) {
    try: target = next(it)
    except StopIteration: break
    body
}
clean it

Universal.Iterators.Loops

Matches `while(...) {...}`
Computes fixpoint using widening
for(init; cond; incr) body

Universal.Iterators.Loops
Matches `while(...) {...}`
Computes fixpoint using widening
for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

Universal.Iterators.Loops
Matches `while(...) {...}`
Computes fixpoint using widening
for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

init;
while(cond) {
  body;
  incr;
}
clean init

Universal.Iterators.Loops
Matches while(...){...}
Computes fixpoint using widening
for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

init;
while(cond) {
    body;
    incr;
}
clean init

for target in iterable: body

Universal.Iterators.Loops
Matches while(...){...}
Computes fixpoint using widening
Mopsa | Dynamic, semantic iterators with delegation

**for(init; cond; incr) body**

- C.iterators.loops
  - Rewrite and analyze recursively

- init;
- while(cond) {
  - body;
  - incr;
- }
- clean init

**for target in iterable: body**

- Python.Desugar.Loops
  - Rewrite and analyze recursively
  - Optimize for some **semantic** cases

- it = iter(iterable)
- while(1) {
  - try: target = next(it)
  - except StopIteration: break
  - body
- }
- clean it

- Universal.Iterations.Loops
  - Matches **while(...) {...}**
  - Computes fixpoint using widening
### Mopsa | Dynamic, semantic iterators with delegation

#### for(init; cond; incr) body

- **C.iterators.loops**
  - Rewrite and analyze recursively

- **init**; while(cond) {
  - body;
  - incr;
}
- **clean init**

#### for target in iterable: body

- **Python.Desugar.Loops**
  - Rewrite and analyze recursively
  - Optimize for some semantic cases

- it = iter(iterable)
- while(1) {
  - try: target = next(it)
  - except StopIteration: break
  - body
- }
- **clean it**

#### Universal.Iterators.Loops

- Matches while(...){...}
- Computes fixpoint using widening
Types | Analysis

- Dynamicity:
  - type inference first
- Flow-sensitive
- Context-sensitive
Types | Analysis

- Dynamicity:
  - type inference first
- Flow-sensitive
- Context-sensitive

Sequence
Cartesian product
Composition
Universal
Python specific
Types | Related work

- Similar in essence to TAJS.\(^4\)
- Dataflow analysis by Fritz and Hage.\(^5\)
- Typpete: SMT-based type inference.\(^6\)
- Pytype, type inference tool used by Google.\(^7\)
- RPython: efficient compilation of a static subset of Python.\(^8\)
- Value analysis by Fromherz et al.\(^9\)

\(^6\) Hassan, Urban, Eilers, and Müller. “MaxSMT-Based Type Inference for Python 3”. CAV 2018.
\(^7\) Kramm et al. Pytype. 2019.
## Experimental evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Mopsa</th>
<th>Fritz &amp; Hage</th>
<th>Pytype</th>
<th>Typette</th>
<th>Fromherz et al.</th>
<th>RPython</th>
</tr>
</thead>
<tbody>
<tr>
<td>bellman_ford.py</td>
<td>61</td>
<td>0.24s</td>
<td>1.4s</td>
<td>0.99s</td>
<td>1.4s</td>
<td>2.4m</td>
<td>7.1s</td>
</tr>
<tr>
<td>float.py</td>
<td>63</td>
<td>82ms</td>
<td>1.7s</td>
<td>0.92s</td>
<td>1.3s</td>
<td>0.84s</td>
<td>5.6s</td>
</tr>
<tr>
<td>coop_concat.py</td>
<td>64</td>
<td>43ms</td>
<td>1.8s</td>
<td>0.81s</td>
<td>1.3s</td>
<td>20ms</td>
<td></td>
</tr>
<tr>
<td>crafting.py</td>
<td>132</td>
<td>0.41s</td>
<td>1.6s</td>
<td>0.97</td>
<td>1.7s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>0.80s</td>
<td>1.7s</td>
<td>1.3s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chaos.py</td>
<td>324</td>
<td>2.3s</td>
<td>13s</td>
<td>11s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scimark.py</td>
<td>416</td>
<td>0.55s</td>
<td>8.5s</td>
<td>4.4s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>richards.py</td>
<td>426</td>
<td>5.0s</td>
<td>38s</td>
<td>2.4s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>4.2s</td>
<td>1.1s</td>
<td>7.4s</td>
<td>2.7s</td>
<td>14s</td>
<td></td>
</tr>
<tr>
<td>go.py</td>
<td>461</td>
<td>15s</td>
<td>8.5s</td>
<td>3.4s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>22s</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regex_v8.py</td>
<td>1792</td>
<td>15s</td>
<td>4.9s</td>
<td>1.7m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>processInput.py</td>
<td>1417</td>
<td>4.8s</td>
<td>2.4s</td>
<td>11s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>choose.py</td>
<td>2562</td>
<td>46s</td>
<td>1.7s</td>
<td>15s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

支撑：unsupported by the analyzer (crash)  时限：timeout (after 1h)

Smashed exceptions: KeyError, IndexError, ValueError
Types | Experimental evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Mopsa</th>
<th>Fritz &amp; Hage</th>
<th>Pytype</th>
<th>Typpete</th>
<th>Fromherz et al.</th>
<th>RPython</th>
</tr>
</thead>
<tbody>
<tr>
<td>bellman_ford.py</td>
<td>61</td>
<td>0.24s</td>
<td>1.4s</td>
<td>0.99s</td>
<td>1.4s</td>
<td>2.4m</td>
<td>7.1s</td>
</tr>
<tr>
<td>float.py</td>
<td>63</td>
<td>82ms</td>
<td>1.7s</td>
<td>0.92s</td>
<td>1.3s</td>
<td>0.84s</td>
<td>5.6s</td>
</tr>
<tr>
<td>coop_concat.py</td>
<td>64</td>
<td>43ms</td>
<td>1.8s</td>
<td>0.81s</td>
<td>1.3s</td>
<td>20ms</td>
<td></td>
</tr>
<tr>
<td>crafting.py</td>
<td>132</td>
<td>0.41s</td>
<td>1.6s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.8s</td>
<td></td>
</tr>
<tr>
<td>chaos.py</td>
<td>324</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.7s</td>
<td>1.3s</td>
</tr>
<tr>
<td>scimark.py</td>
<td>416</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14s</td>
<td></td>
</tr>
<tr>
<td>richards.py</td>
<td>426</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7.8s</td>
<td></td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>go.py</td>
<td>461</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regex_v8.py</td>
<td>1792</td>
<td>15s</td>
<td>4.9s</td>
<td></td>
<td></td>
<td>1.7m</td>
<td></td>
</tr>
<tr>
<td>processInput.py</td>
<td>1417</td>
<td>4.8s</td>
<td>7s</td>
<td>2.4s</td>
<td>11s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>choose.py</td>
<td>2562</td>
<td>46s</td>
<td>17s</td>
<td>1.7s</td>
<td>15s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unsupported by the analyzer (crash) ☢️ Timeout (after 1h)
Smashed exceptions: KeyError 🕰️, IndexError ⬆️, ValueError 🕳️
Thanks to Mopsa, switching from types to values is straightforward!
Thanks to Mopsa, switching from types to values is straightforward!

Monat, Ouadjaout, and Miné. “Value and allocation sensitivity in static Python analyses”. 
Types $\rightsquigarrow$ values | Configurations

Thanks to Mopsa, switching from types to values is straightforward!

Monat, Ouadjaout, and Miné. “Value and allocation sensitivity in static Python analyses”. 20
Types ⇔ values | Comparing the analyses

### Averaging tasks

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i].weight
    m = m // (i + 1)
    return m

l = []
for i in range(randint(5, 10)):
    l.append(Task(randint(0, 20)))
m = average(l)
```

### Type analysis

- **ValueError (l. 3)**

- **IndexError (l. 9)**

- **ZeroDivisionError (l. 10)**

- **NameError (l. 10)**
Types ∼ values | Comparing the analyses

Averaging tasks

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

    def average(l):
        m = 0
        for i in range(len(l)):
            m = m + l[i].weight
        m = m // (i + 1)
        return m

l = []
for i in range(randint(5, 10)):
    l.append(Task(randint(0, 20)))

m = average(l)
```

Type analysis

- **ValueError (l. 3)**
- **IndexError (l. 9)**
Types $\mapsto$ values | Comparing the analyses

### Averaging tasks

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i].weight
    m = m // (i + 1)
    return m

l = []
for i in range(randint(5, 10)):
    l.append(Task(randint(0, 20)))
    m = average(l)
```

### Type analysis

- **ValueError (l. 3)**
- **IndexError (l. 9)**
- **ZeroDivisionError (l. 10)**

---

Non-relational value analysis

Relational value analysis

No alarm!
### Averaging tasks

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i].weight
    m = m // (i + 1)
    return m

l = []
for i in range(randint(5, 10)):
    l.append(Task(randint(0, 20)))
    m = average(l)
```

### Type analysis

- **ValueError (l. 3)**
- **IndexError (l. 9)**
- **ZeroDivisionError (l. 10)**
- **NameError (l. 10)**
Averaging tasks

1. **Class Task:**
   ```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight
```

2. **Def average(l):**
   ```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i].weight
    m = m // (i + 1)
    return m
```

3. **List l:**
   ```python
l = []
for i in range(randint(5, 10)):
    l.append(Task(randint(0, 20)))
```

**Type analysis**
- **ValueError (l. 3)**
- **IndexError (l. 9)**
- **ZeroDivisionError (l. 10)**
- **NameError (l. 10)**

**Non-relational value analysis**
**IndexError (l. 9)**
Averaging tasks

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i].weight
    m = m // (i + 1)
    return m

l = []
for i in range(randint(5, 10)):
    l.append(Task(randint(0, 20)))
    m = average(l)
```

### Type analysis

- **ValueTypeError** (l. 3)
- **IndexError** (l. 9)
- **ZeroDivisionError** (l. 10)
- **NameError** (l. 10)

### Non-relational value analysis

**IndexError** (l. 9)

### Relational value analysis

No alarm!
## Types $\mapsto$ values | Comparing the analyses (II)

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Type Analysis</th>
<th>Non-relational Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Type</td>
<td>Index</td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>1.5s</td>
<td>3MB</td>
</tr>
<tr>
<td>scimark.py</td>
<td>416</td>
<td>1.4s</td>
<td>12MB</td>
</tr>
<tr>
<td>richards.py</td>
<td>426</td>
<td>13s</td>
<td>112MB</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>8.3s</td>
<td>7MB</td>
</tr>
<tr>
<td>go.py</td>
<td>461</td>
<td>27s</td>
<td>345MB</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>1.1m</td>
<td>525MB</td>
</tr>
<tr>
<td>regex_v8.py</td>
<td>1792</td>
<td>23s</td>
<td>18MB</td>
</tr>
<tr>
<td>processInput.py</td>
<td>1417</td>
<td>10s</td>
<td>64MB</td>
</tr>
<tr>
<td>choose.py</td>
<td>2562</td>
<td>1.1m</td>
<td>1.6GB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>9294</td>
<td>4.0m</td>
<td>2.8GB</td>
</tr>
</tbody>
</table>
### Types ⇝ values | Comparing the analyses (II)

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time</th>
<th>Mem.</th>
<th>Exceptions detected</th>
<th>Non-relational Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Type Analysis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Mem.</td>
<td>Exceptions detected</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Type</td>
<td>Index</td>
<td>Key</td>
<td>Type</td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>1.5s</td>
<td>3MB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>scimark.py</td>
<td>416</td>
<td>1.4s</td>
<td>12MB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>richards.py</td>
<td>426</td>
<td>13s</td>
<td>112MB</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>8.3s</td>
<td>7MB</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>go.py</td>
<td>461</td>
<td>27s</td>
<td>345MB</td>
<td>33</td>
<td>20</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>1.1m</td>
<td>525MB</td>
<td>0</td>
<td>46</td>
</tr>
<tr>
<td>regex_v8.py</td>
<td>1792</td>
<td>23s</td>
<td>18MB</td>
<td>0</td>
<td>2053</td>
</tr>
<tr>
<td>processInput.py</td>
<td>1417</td>
<td>10s</td>
<td>64MB</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>choose.py</td>
<td>2562</td>
<td>1.1m</td>
<td>1.6GB</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>9294</td>
<td>4.0m</td>
<td>2.8GB</td>
<td>59</td>
<td>2214</td>
</tr>
</tbody>
</table>

**Conclusion**

The non-relational value analysis

- does not remove false type alarms
- significantly reduces index errors
- is \(\simeq 3 \times\) costlier
## Types ↦ values | Comparing the analyses (II)

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Type Analysis</th>
<th>Non-relational Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>1.5s</td>
<td>3MB</td>
</tr>
<tr>
<td>scimark.py</td>
<td>416</td>
<td>1.4s</td>
<td>12MB</td>
</tr>
<tr>
<td>richards.py</td>
<td>426</td>
<td>13s</td>
<td>112MB</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>8.3s</td>
<td>7MB</td>
</tr>
<tr>
<td>go.py</td>
<td>461</td>
<td>27s</td>
<td>345MB</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>1.1m</td>
<td>525MB</td>
</tr>
<tr>
<td>regex_v8.py</td>
<td>1792</td>
<td>23s</td>
<td>18MB</td>
</tr>
<tr>
<td>processInput.py</td>
<td>1417</td>
<td>10s</td>
<td>64MB</td>
</tr>
<tr>
<td>choose.py</td>
<td>2562</td>
<td>1.1m</td>
<td>1.6GB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>9294</td>
<td>4.0m</td>
<td>2.8GB</td>
</tr>
</tbody>
</table>

### Conclusion

The non-relational value analysis

- does not remove false type alarms
- significantly reduces index errors
- is \( \simeq 3 \times \) costlier

### Heuristic packing and relational analyses

- Static packing, using function’s scope
- Rules out all 145 alarms of `regex_v8.py` (1792 LOC) at 2.5× cost
The recency abstraction

- Finite number of abstract addresses

---

# The recency abstraction

- Finite number of abstract addresses
- Precise analysis of object initialization

---


The recency abstraction

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:

- by allocation site
- through a recency criterion:
  - \((l, r)\) most recent allocation (with strong updates)
  - \((l, o)\) older addresses (summarized)

Initially designed for analysis of low-level code (binaries, C)
Also used in Type Analysis for JavaScript

---

## The recency abstraction\(^\text{10}\)

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:
  - by allocation site \( l \in L \)


The recency abstraction

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:
  - by allocation site $l \in \mathbb{L}$
  - through a recency criterion: $(l, r)$ most recent allocation (with strong updates)
  - $(l, o)$ older addresses (summarized)

---


The recency abstraction

<table>
<thead>
<tr>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Finite number of abstract addresses</td>
</tr>
<tr>
<td>- Precise analysis of object initialization</td>
</tr>
<tr>
<td>- Twofold partitioning:</td>
</tr>
<tr>
<td>- by allocation site $l \in \mathbb{L}$</td>
</tr>
<tr>
<td>- through a recency criterion: $(l, r)$ most recent allocation (with strong updates)</td>
</tr>
<tr>
<td>- $(l, o)$ older addresses (summarized)</td>
</tr>
<tr>
<td>- Initially designed for analysis of low-level code (binaries, C)</td>
</tr>
</tbody>
</table>

---


The recency abstraction

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:
  - by allocation site $l \in L$
  - through a recency criterion: $(l, r)$ most recent allocation (with strong updates)
  - $(l, o)$ older addresses (summarized)
- Initially designed for analysis of low-level code (binaries, C)
- Also used in Type Analysis for JavaScript

---

Shared abstractions | Variations around the recency abstraction

**Type analysis**

Nominal types used in abstract addresses. No need for allocation-site in `Tasks`. But helpful for lists!

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

m = [1, 2]
l = [Task(i) for i in m]
l.append(Task(3))
```
Task creation

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

m = [1, 2]
l = [Task(i) for i in m]
l.append(Task(3))
```

Type analysis
Nominal types used in abstract addresses. No need for allocation-site in Tasks. But helpful for lists!

Value analysis
Use allocation sites for range objects.
Shared abstractions | Variations around the recency abstraction

Task creation

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

m = [1, 2]
l = [Task(i) for i in m]
l.append(Task(3))
```

Type analysis
Nominal types used in abstract addresses. No need for allocation-site in Tasks. But helpful for lists!

Value analysis
Use allocation sites for `range` objects.

Variable allocation policies

- Type-based (nominal) and/or location-based partitioning.
- Different configurations depending on type/value analysis.
### List abstraction

- Summarization of the content (auxiliary variable)
- Auxiliary length variable
List abstraction

- Summarization of the content (auxiliary variable)
- Auxiliary length variable

Dictionaries in Python

- Keys can be *any object* (JavaScript: strings or symbols)
- Key/value summarization currently used
Two soundnesses

- Modelization of the semantics from CPython
- Implementation of this semantics within Mopsa
Soundness

Two soundnesses

- Modelization of the semantics from CPython
- Implementation of this semantics within Mopsa

Our approach

- Test only in the abstract
- Issue of overapproximations and unproved assertions

Unsupported constructs

- `eval`
- Recursive functions
- Finalizers
Soundness

Two soundnesses
- Modelization of the semantics from CPython
- Implementation of this semantics within Mopsa

Our approach
- Test only in the abstract
- Issue of overapproximations and unproved assertions

Unsupported constructs
- `eval`
- Recursive functions
- Finalizers

Tests from previous works
- 450/586 tests supported
- 268/586 assertions proved

Official tests from CPython
- 325/416 tests supported (17 chosen files)
- 389/702 assertions proved
Analyzing Python Programs with C Libraries
Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code
Combining C and Python – motivation

<table>
<thead>
<tr>
<th>One in five of the top 200 Python libraries contains C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>- To bring better performance (numpy)</td>
</tr>
</tbody>
</table>
Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)
Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Pitfalls
Combining C and Python – motivation

<table>
<thead>
<tr>
<th>One in five of the top 200 Python libraries contains C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>► To bring better performance (numpy)</td>
</tr>
<tr>
<td>► To provide library bindings (pygit2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pitfalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>► Different values (arbitrary-precision integers in Python, bounded in C)</td>
</tr>
</tbody>
</table>
# Combining C and Python – motivation

<table>
<thead>
<tr>
<th>One in five of the top 200 Python libraries contains C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>► To bring better performance (numpy)</td>
</tr>
<tr>
<td>► To provide library bindings (pygit2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pitfalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>► Different values (arbitrary-precision integers in Python, bounded in C)</td>
</tr>
<tr>
<td>► Different object representations (Python objects, C structs)</td>
</tr>
</tbody>
</table>
Combining C and Python – motivation

<table>
<thead>
<tr>
<th>One in five of the top 200 Python libraries contains C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>► To bring better performance (numpy)</td>
</tr>
<tr>
<td>► To provide library bindings (pygit2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pitfalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>► Different values (arbitrary-precision integers in Python, bounded in C)</td>
</tr>
<tr>
<td>► Different object representations (Python objects, C structs)</td>
</tr>
<tr>
<td>► Different runtime-errors (exceptions in Python)</td>
</tr>
</tbody>
</table>
Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Pitfalls

- Different values (arbitrary-precision integers in Python, bounded in C)
- Different object representations (Python objects, C structs)
- Different runtime-errors (exceptions in Python)
- Garbage collection
A combined static analysis of C/Python¹²

- Targeting C extensions using the CPython API

A combined static analysis of C/Python

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the “glue”)

---

A combined static analysis of C/Python\textsuperscript{12}

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the “glue”)
- Observations

\textsuperscript{12}Monat, Ouadjaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021.
A combined static analysis of C/Python\textsuperscript{12}

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the “glue”)
- Observations
  - allocated objects are shared in the memory

\textsuperscript{12}Monat, Ouadjiaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021.
A combined static analysis of C/Python\textsuperscript{12}

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the “glue”)
- Observations
  - allocated objects are shared in the memory
  - but each language has different abstractions

\textsuperscript{12}Monat, Ouadjaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021.
A combined static analysis of C/Python$^{12}$

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the “glue”)
- Observations
  - allocated objects are shared in the memory
  - but each language has different abstractions

⇒ Share universal domains and synchronize abstractions

---

Combining C and Python – example

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject *
CounterIncr(Counter *self, PyObject *args)
{
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
        return NULL;
    self->count += i;
    Py_RETURN_NONE;
}

static PyObject *
CounterGet(Counter *self)
{
    return Py_BuildValue("i", self->count);
}
```

count.py

```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```
### Combining C and Python – example

<table>
<thead>
<tr>
<th>counter.c</th>
<th>count.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>```c</td>
<td>```py</td>
</tr>
<tr>
<td>1 typedef struct {</td>
<td>1 from counter import Counter</td>
</tr>
<tr>
<td>PyObject_HEAD;</td>
<td>2 from random import randrange</td>
</tr>
<tr>
<td>int count;</td>
<td>3</td>
</tr>
<tr>
<td>} Counter;</td>
<td>4 c = Counter()</td>
</tr>
<tr>
<td>5 static PyObject*</td>
<td>5 power = randrange(128)</td>
</tr>
<tr>
<td>CounterIncr(Counter *self, PyObject *args)</td>
<td>6 c.incr(2**power-1)</td>
</tr>
<tr>
<td>7 {</td>
<td>7 c.incr()</td>
</tr>
<tr>
<td>int i = 1;</td>
<td>8 r = c.get()</td>
</tr>
<tr>
<td>if(!PyArg_ParseTuple(args, &quot;</td>
<td>i&quot;, &amp;i))</td>
</tr>
<tr>
<td>return NULL;</td>
<td></td>
</tr>
<tr>
<td>self-&gt;count += i;</td>
<td></td>
</tr>
<tr>
<td>Py_RETURN_NONE;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>static PyObject*</td>
<td></td>
</tr>
<tr>
<td>CounterGet(Counter *self)</td>
<td></td>
</tr>
<tr>
<td>8 {</td>
<td></td>
</tr>
<tr>
<td>return Py_BuildValue(&quot;i&quot;, self-&gt;count);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

▶ power \( \leq 30 \) \( \Rightarrow \) \( r = 2^{\text{power}} \)
### Combining C and Python – example

**counter.c**

```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject*
CounterIncr(Counter *self, PyObject *args)
{
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
        return NULL;
    self->count += i;
    Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
    return Py_BuildValue("i", self->count);
}
```

**count.py**

```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

- **power** ≤ 30 ⇒ \( r = 2^{\text{power}} \)
- **32 ≤ power ≤ 64**: OverflowError: signed integer is greater than maximum
- **power** ≥ 64: OverflowError: Python int too large to convert to C long
### Combining C and Python – example

**counter.c**
```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject*
CounterIncr(Counter *self, PyObject *args) {
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
        return NULL;
    self->count += i;
    Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self) {
    return Py_BuildValue("i", self->count);
}
```

**count.py**
```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

- **power ≤ 30 ⇒ r = 2^power**
- **power = 31 ⇒ r = −2^{31}**
- **32 ≤ power ≤ 64**: OverflowError: signed integer is greater than maximum
- **power ≥ 64**: OverflowError: Python int too large to convert to C long
How to analyze multilanguage programs?

Type annotations

class Counter:
    def __init__(self): ...
    def incr(self, i: int = 1): ...
    def get(self) -> int: ...
How to analyze multilanguage programs?

Type annotations

class Counter:
    def __init__(self): ...
    def incr(self, i: int = 1): ...
    def get(self) -> int: ...

- No raised exceptions $\Rightarrow$ missed errors
How to analyze multilanguage programs?

Type annotations

```python
class Counter:
    def __init__(self): ...
    def incr(self, i: int = 1): ...
    def get(self) -> int: ...
```

- No raised exceptions ➞ missed errors
- Only types
How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Counter:</td>
</tr>
<tr>
<td>def <strong>init</strong>(self): ...</td>
</tr>
<tr>
<td>def incr(self, i: int = 1): ...</td>
</tr>
<tr>
<td>def get(self) -&gt; int: ...</td>
</tr>
</tbody>
</table>

- No raised exceptions ➞ missed errors
- Only types
- Typeshed: type annotations for the standard library
How to analyze multilanguage programs?

Type annotations

```python
class Counter:
    def __init__(self): ...
    def incr(self, i: int = 1): ...
    def get(self) -> int: ...
```

- No raised exceptions ⇒ missed errors
- Only types
- Typeshed: type annotations for the standard library, used in the single-language analysis before
How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

```python
class Counter:
    def __init__(self):
        self.count = 0
    def get(self):
        return self.count
    def incr(self, i=1):
        self.count += i
```

No integer wrap-around in Python

Some effects can't be written in pure Python (e.g., read-only attributes)

Our approach

- Analyze both the C and Python sources
- Switch from one language to the other just as the program does
- Reuse previous analyses of C and Python
- Detect runtime errors in Python, in C, and at the boundary
How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

```python
class Counter:
    def __init__(self):
        self.count = 0
    def get(self):
        return self.count
    def incr(self, i=1):
        self.count += i
```

- No integer wrap-around in Python
How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

class Counter:
    def __init__(self):
        self.count = 0
    def get(self):
        return self.count
    def incr(self, i=1):
        self.count += i

- No integer wrap-around in Python
- Some effects can’t be written in pure Python (e.g., read-only attributes)
<table>
<thead>
<tr>
<th>How to analyze multilanguage programs?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type annotations</strong></td>
</tr>
<tr>
<td><strong>Rewrite into Python code</strong></td>
</tr>
<tr>
<td><strong>Drawbacks of the current approaches</strong></td>
</tr>
</tbody>
</table>
How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite into Python code</td>
</tr>
<tr>
<td>Drawbacks of the current approaches</td>
</tr>
</tbody>
</table>

- Not the real code
### How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite into Python code</td>
</tr>
<tr>
<td>Drawbacks of the current approaches</td>
</tr>
<tr>
<td>▶ Not the real code</td>
</tr>
<tr>
<td>▶ Not automatic: manual conversion</td>
</tr>
</tbody>
</table>
How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

▶ Not the real code
▶ Not automatic: manual conversion
▶ Not sound: some effects are not taken into account
How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite into Python code</td>
</tr>
<tr>
<td>Drawbacks of the current approaches</td>
</tr>
<tr>
<td>- Not the real code</td>
</tr>
<tr>
<td>- Not automatic: manual conversion</td>
</tr>
<tr>
<td>- Not sound: some effects are not taken into account</td>
</tr>
</tbody>
</table>

Our approach
How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

Our approach

- Analyze both the C and Python sources
How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite into Python code</td>
</tr>
<tr>
<td>Drawbacks of the current approaches</td>
</tr>
<tr>
<td>- Not the real code</td>
</tr>
<tr>
<td>- Not automatic: manual conversion</td>
</tr>
<tr>
<td>- Not sound: some effects are not taken into account</td>
</tr>
<tr>
<td>Our approach</td>
</tr>
<tr>
<td>- Analyze both the C and Python sources</td>
</tr>
<tr>
<td>- Switch from one language to the other just as the program does</td>
</tr>
</tbody>
</table>
How to analyze multilanguage programs?

**Type annotations**

**Rewrite into Python code**

**Drawbacks of the current approaches**
- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

**Our approach**
- Analyze both the C and Python sources
- Switch from one language to the other just as the program does
- Reuse previous analyses of C and Python
### How to analyze multilanguage programs?

#### Type annotations

#### Rewrite into Python code

#### Drawbacks of the current approaches
- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

#### Our approach
- Analyze both the C and Python sources
- Switch from one language to the other just as the program does
- Reuse previous analyses of C and Python
- Detect runtime errors in Python, in C, and at the boundary
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject* CounterIncr(Counter *self, PyObject *args) {
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
        return NULL;
    self->count += i;
    Py_RETURN_NONE;
}

static PyObject* CounterGet(Counter *self) {
    return Py_BuildValue("i", self->count);
}

from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
typedef struct {
PyObject_HEAD;
    int count;
} Counter;

static PyObject* CounterIncr(Counter *self, 
{
    int i = 1;
    if(!PyArg_ParseTuple(a
        return NULL;
    
    self->count += i;
    Py_RETURN_NONE;
}

static PyObject* CounterGet(Counter *self) 
{
    return Py_BuildValue("i",self->count);
}

from counter import Counter
from random import randrange

power = randrange(128)
c = Counter()
c.incr(2**power-1)
c.incr()
r = c.get()
## Multilanguage semantics

<table>
<thead>
<tr>
<th>Concrete definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>➤ Builds upon the Python and C semantics</td>
</tr>
</tbody>
</table>

### Limitations

- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
- Manual modelization from CPython's source code
Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views

Limitations

- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
- Manual modelization from CPython's source code
Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

Limitations

- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
- Manual modelization from CPython's source code
## Multilanguage semantics

### Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

### Limitations

- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
- Manual modelization from CPython’s source code
Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

Limitations

- Garbage collection not handled
Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

Limitations

- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
## Multilanguage semantics

### Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- **Boundary functions** when objects switch views for the first time

### Limitations

- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
- Manual modelization from CPython’s source code
From distinct Python and C analyses...
... to a multilanguage analysis!
... to a multilanguage analysis!
... to a multilanguage analysis!

Implementation LOC

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
</tbody>
</table>

13200 LOC

Universal: 5600 LOC

C specific: 11700 LOC

Python specific: 12600 LOC

Multilanguage: 2500 LOC
... to a multilanguage analysis!

### Implementation LOC

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
<tr>
<td>Universal</td>
<td>5600</td>
</tr>
</tbody>
</table>
... to a multilanguage analysis!

Implementation LOC

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
<tr>
<td>Universal</td>
<td>5600</td>
</tr>
<tr>
<td>Python</td>
<td>12600</td>
</tr>
</tbody>
</table>

Part LOC

- Framework: 13200
- Universal: 5600
- Python: 12600

Framework LOC

- Implementation: 13200
- Part: 5600
- Framework: 13200
- Universal: 5600
- Python: 12600
- Multilanguage: 2500
... to a multilanguage analysis!

**Implementation LOC**

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
<tr>
<td>Universal</td>
<td>5600</td>
</tr>
<tr>
<td>C</td>
<td>11700</td>
</tr>
<tr>
<td>Python</td>
<td>12600</td>
</tr>
</tbody>
</table>
... to a multilanguage analysis!

Implementation LOC

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
<tr>
<td>Universal</td>
<td>5600</td>
</tr>
<tr>
<td>C</td>
<td>11700</td>
</tr>
<tr>
<td>Python</td>
<td>12600</td>
</tr>
<tr>
<td>Multilanguage</td>
<td>2500</td>
</tr>
</tbody>
</table>

Part LOC

Framework 13200
Universal 5600
C 11700
Python 12600
Multilanguage 2500
Benchmarks

Corpus selection

- Popular, real-world libraries available on GitHub, averaging 412 stars.
- Whole-program analysis: we use the tests provided by the libraries.

| Library         | |C|  | |Py|  | Tests | safe C checks | total C checks | Assertions | Py ⇄ C | average # transitions between Python and C |
|-----------------|---|---|-------------------|-------------------|-----------------|----------------|-------------------|-----------------|-----------------|-----------------|-----------------|
| noise           | 722 | 675 | 15/15 | 18s | 99.6% | (4952) | 100.0% | (1738) | 0/21 | 6.5 |
| ahocorasic      | 3541 | 1336 | 46/92 | 54s | 93.1% | (1785) | 98.0% | (4937) | 30/88 | 5.4 |
| levenshtein     | 5441 | 357 | 17/17 | 1.5m | 79.9% | (3106) | 93.2% | (1719) | 0/38 | 2.7 |
| cdistance       | 1433 | 912 | 28/28 | 1.9m | 95.3% | (1832) | 98.3% | (11884) | 88/207 | 8.7 |
| llist           | 2829 | 1686 | 167/194 | 4.2m | 99.0% | (5311) | 98.8% | (30944) | 235/691 | 51.7 |
| bitarray        | 3244 | 2597 | 159/216 | 4.2m | 96.3% | (4496) | 94.6% | (21070) | 100/378 | 14.8 |

safe C checks

total C checks

total C checks

average # transitions between Python and C per test
Related work

Theoretical frameworks

- Matthews and Findler\textsuperscript{13} boundary functions as value conversions between two languages.
- Buro, Crole, and Mastroeni\textsuperscript{14} generic framework for combining analyses of different languages.

\textsuperscript{13} Matthews and Findler. “Operational semantics for multi-language programs”. 2009.
Related work (II)

Around the Java Native Interface (JNI)

Static translation of some of C’s effects, injected back into the Java analysis.

- Effects of C code on Java heap modelized using JVML\(^\text{15}\)
- Type inference of Java objects in C code\(^\text{16}\)
- Extraction of C callbacks to Java\(^\text{17}\)

- Modular analyses
- No numeric information
- Missing C runtime errors

\(^{15}\text{Tan and Morrisett. “Ilea: inter-language analysis across Java and C”. OOPSLA 2007.}\)
\(^{16}\text{Furr and Foster. “Checking type safety of foreign function calls”. 2008.}\)
Conclusion
Difficulties

- Size of the semantics
- CPython’s source code
**Contribution: concrete semantics of Python**

**Difficulties**
- Size of the semantics
- CPython’s source code

**Previous works**
- Executable semantics of Python
- Handcrafted tests
Contribution: concrete semantics of Python

<table>
<thead>
<tr>
<th>Difficulties</th>
<th>Previous works</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the semantics</td>
<td>Executable semantics of Python</td>
</tr>
<tr>
<td>CPython’s source code</td>
<td>Handcrafted tests</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Our results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics suitable for abstract interpretation</td>
</tr>
<tr>
<td>Written and explained in the manuscript (70 cases)</td>
</tr>
<tr>
<td>Backreferences to the source code</td>
</tr>
<tr>
<td>Preliminary tests using CPython’s suite</td>
</tr>
</tbody>
</table>
Contribution: type & value analyses of Python

Difficulties

- Dynamicity
- Dual type system
- Size of the semantics

Previous works

- JS: type and constant analysis
- Python: no scalability or support of dynamicity

Our results

- Type analysis
- Numeric value analysis & new sensitivities for the recency abstraction
- Relational value analysis with packing (manuscript)
- Scale to small, real-world benchmarks

---

18 Monat, Ouadjaout, and Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP 2020.
**Contribution:** type & value analyses of Python

### Difficulties
- Dynamicity
- Dual type system
- Size of the semantics

### Previous works
- JS: type and constant analysis
- Python: no scalability or support of dynamicity

---

18 Monat, Ouadjaout, and Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP 2020.

## Contribution: type & value analyses of Python

### Difficulties
- Dynamicity
- Dual type system
- Size of the semantics

### Previous works
- JS: type and constant analysis
- Python: no scalability or support of dynamicity

### Our results
- Type analysis\(^{18}\),
- **Numeric value analysis** & new sensitivities for the recency abstraction\(^{19}\)
- **Relational value analysis** with packing (manuscript)
- Scale to small, real-world benchmarks

---

\(^{18}\) Monat, Ouadjaout, and Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP 2020.

\(^{19}\) Monat, Ouadjaout, and Miné. “Value and allocation sensitivity in static Python analyses”. SOAP@PLDI 2020.
Contribution: multilanguage Python/C analysis

Difficulties

- Concrete semantics
- Memory interaction

---

Monat, Ouadjaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021
Contribution: multilanguage Python/C analysis

Difficulties
- Concrete semantics
- Memory interaction

Previous works
- Type/exceptions analyses for the JNI
- No detection of runtime errors in C

Monat, Ouadjaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021
Contribution: multilanguage Python/C analysis

**Difficulties**
- Concrete semantics
- Memory interaction

**Previous works**
- Type/exceptions analyses for the JNI
- No detection of runtime errors in C

**Our results**
- Careful separation of the states and modelization of the API
- Lightweight domain on top of off-the-shelf C and Python analyses
- Shared underlying abstractions (numeric, recency)
- Scale to small, real-world libraries (using client code)

---

Monat, Ouadjaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021
Some future works

**Executable concrete semantics**

- Split soundness testing (CPython – concrete semantics – analyzer)
- Conformance tests
Some future works

### Executable concrete semantics
- Split soundness testing (CPython – concrete semantics – analyzer)
- Conformance tests

### Dictionary abstractions
- Beyond key/value summarization
- Empirical study of dictionary use (use of non-string keys)
### Some future works

<table>
<thead>
<tr>
<th>Executable concrete semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Split soundness testing (CPython – concrete semantics – analyzer)</td>
</tr>
<tr>
<td>- Conformance tests</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dictionary abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Beyond key/value summarization</td>
</tr>
<tr>
<td>- Empirical study of dictionary use (use of non-string keys)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multilanguage library analyses</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Other interoperability frameworks (Cffi, Swig, Cython)</td>
</tr>
<tr>
<td>- Infer Typeshed’s annotations</td>
</tr>
<tr>
<td>- Library analysis without client code</td>
</tr>
</tbody>
</table>
Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries

Questions

xkcd.com/353

Raphaël Monat