A Multilanguage Static Analysis of Python/C Programs with Mopsa

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné
Introduction
### Specifications of the analyzer

**Inference**  of program properties such as the absence of run-time errors.

**Semantic**  based on a formal modelization of the language.

**Automatic**  no expert knowledge required.

**Sound**  covers all possible executions.

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

r1 = average([1, 2, 3])
r2 = average(['a', 'b', 'c'])
```

```
#include <string.h>

int main(int argc, char *argv[])
{
    int i = 0;
    for (char **p = argv; *p; p++) {
        strlen(*p); // valid string
        i++; // no overflow
    }
    return 0;
}
```

Type Error: unsupported operand type(s) for '+': 'int' and 'str'

No alarm
Dynamic programming languages

Most popular languages on GitHub

- JavaScript: 23%
- Python: 18%
- Java: 12%
- TypeScript: 8%
- Go: 8%
- C++: 6%
- Ruby: 6%
- PHP: 5%
- Other (< 5%): 23%
Dynamic programming languages

Most popular languages on GitHub

- JavaScript: 23%
- Python: 18%
- Java: 14%
- TypeScript: 12%
- Go: 8%
- C++: 6%
- Ruby: 6%
- PHP: 5%
- Other (< 5%): 23%

Features
- Object orientation
- Dynamic typing
- Dynamic object structure
- Introspection operators
- eval
Combining C and Python – motivation

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

- To bring better performance (numpy)
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

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 ($\mathbb{Z}$ vs. Int32)
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 ($\mathbb{Z}$ vs. Int32)
- Shared memory state
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 __getattr__(self, attr):
        if attr[0] == "_":
            raise AttributeError("...")
        return object.__getattr__(self, attr)

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

Dual type system

▶ Nominal (classes, MRO)

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)

```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

- $1 + "a" \rightarrow \text{TypeError}$
- $l[len(l) + 1] \rightarrow \text{IndexError}$

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())
```

Example Semantics – binary operators

1. \( a_1 = \text{eval } e_1; \ a_2 = \text{eval } e_2 \)
2. \( \text{has_field}(a_1, \text{__add__})? \)
   - Yes
   - \( \text{has_field}(a_2, \text{__radd__}) \) \&\& \( \text{type}(a_1) < \text{type}(a_2) \)?
     - Yes
     - \( a_3 = \text{call } a_1\text{'s } \text{__add__}\text{ on } a_1, a_2 \)
     - No
     - \( a_3 == \text{NotImplemented} \)?
       - Yes
       - Type Error
       - No
3. \( \text{Result is } a_3 \)
4. \( \text{has_field}(a_2, \text{__radd__}) \) \&\& \( \text{type}(a_1) \neq \text{type}(a_2) \)?
   - Yes
   - \( a_3 = \text{call } a_2\text{'s } \text{__radd__}\text{ on } a_1, a_2 \)
   - No
   - \( a_3 == \text{NotImplemented} \)?
     - Yes
     - Type Error
     - No
Custom infix operators

```python
class Infix(object):
    def __init__(self, func): self.func = func
    def __or__(self, other): return self.func(other)
    def __ror__(self, other): return Infix(lambda x: self.func(other, x))

instanceof = Infix(isinstance)
b = 5 | instanceof | int

@Infix
def padd(x, y):
    print(f"{x} + {y} = {x + y}"")
    return x + y
c = 2 | padd | 3
```

Credits tomerfiliba.com/blog/Infix-Operators/
Overview of our value analysis for Python

<table>
<thead>
<tr>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect runtime errors: uncaught raised exceptions</td>
</tr>
</tbody>
</table>

Supported constructs:
- Objects
- Exceptions
- Dynamic typing
- Introspection
- Permissive semantics
- Dynamic attributes
- Generators
- super
- Metaclasses

Unsupported constructs:
- Recursive functions
- eval
- Finalizers
Overview of our value analysis for Python

**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
# Overview of our value analysis for Python

## 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
Mopsa
A program analysis workflow

Averaging 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)
```
A program analysis workflow

Averaging 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)
```

Proved safe?

- $m \div (i+1)$
- $l[i]$
A program analysis workflow

Averaging 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

$m \mapsto @_{\text{int}}# \quad i \mapsto @_{\text{int}}# \quad \text{els}(l) \mapsto @_{\text{int}}#$
A program analysis workflow

Averaging numbers

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

Searching for a loop invariant (l. 4)

Stateless domains: list content,

Environment abstraction

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

Numeric abstraction (intervals)

\[ m \in [0, +\infty) \quad \text{els}(l) \in [0, 20] \quad i \in [0, +\infty) \]

Proved safe?

- \( m \div (i+1) \)
- \( l[i] \)
A program analysis workflow

Averaging 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

Environment abstraction

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

Numeric abstraction (intervals)

$m \in [0, +\infty) \quad \text{els}(l) \in [0, 20]$

$\text{len}(l) \in [5, 10] \quad i \in [0, 10]$

Searching for a loop invariant (l. 4)

Proved safe?

- $m \div (i+1)$
- $l[i]$
A program analysis workflow

### Averaging 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]`

### Searching for a loop invariant (l. 4)

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

**Environment abstraction**

- `m ↦ @_{int\#}`
- `i ↦ @_{int\#}`
- `els(l) ↦ @_{int\#}`

**Numeric abstraction (polyhedra)**

- `m ∈ [0, +∞)`
- `els(l) ∈ [0, 20]`
- `0 ≤ i < len(l)`
- `5 ≤ len(l) ≤ 10`
A program analysis workflow

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**

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

**Numeric abstraction (polyhedra)**

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

**Attributes abstraction**

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

Proved safe?

- \( m \div (i+1) \)
- \( l[i].\text{weight} \)
A program analysis workflow

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**

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

*Conclusion*

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

**Attributes abstraction**

\[ @\text{Task} \mapsto \{\text{weight}\}, \emptyset \]

Proved safe?

- \( m \div (i+1) \)
- \( l[i].\text{weight} \)
Overview of Mopsa

Modular Open Platform for Static Analysis\textsuperscript{1}

gitlab.com/mopsa/mopsa-analyzer

\textsuperscript{1}Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. 2019.
Overview of Mopsa

Modular Open Platform for Static Analysis\(^1\)
gitlab.com/mopsa/mopsa-analyzer

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

\(^1\)Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. 2019.
Overview of Mopsa

**Modular Open Platform for Static Analysis**¹

gitlab.com/mopsa/mopsa-analyzer

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

▶ Unified domain signature
  - PEN: Semantic rewriting
  - PUZZLE-PIECE: Loose coupling
  - MICROSCOPE: Observability

Overview of Mopsa

Modular Open Platform for Static Analysis\(^1\)

gitlab.com/mopsa/mopsa-analyzer

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

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

- DAG of abstract domains
  - Composition
  - Cooperation

\(^1\)Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. 2019.
Overview of Mopsa

**Modular Open Platform for Static Analysis**

[gitlab.com/mopsa/mopsa-analyzer](http://gitlab.com/mopsa/mopsa-analyzer)

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

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

- **DAG of abstract domains**
  - Composition
  - Cooperation

---

Dynamic, semantic iterators with delegation

C.iterators.loops

Rewrite and analyze recursively

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

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
Dynamic, semantic iterators with delegation

for(init; cond; incr) body

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

for(init; cond; incr) body

C.iterators.loops

Rewrite and analyze recursively

Universal.Iterators.Loops

Matches while(...) {...}
Computes fixpoint using widening
Dynamic, semantic iterators with delegation

for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

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

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

for (init; cond; incr) { body }

C.iterators.loops
Rewrite and analyze recursively

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

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
Dynamic, semantic iterators with delegation

for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

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

for target in iterable: body

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

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

\[
\text{for}(\text{init}; \text{cond}; \text{incr}) \text{ body}
\]

\[
\text{C.iterators.loops}
\]

Rewrite and analyze recursively

\[
\text{init;}
\text{while(\text{cond})}\
\text{body;}
\text{incr;}
\]

\[
\text{Universal.Iterators.Loops}
\]

Matches \text{while(...)}\{...\}

Computes fixpoint using widening

\[
\text{for target in iterable: body}
\]

\[
\text{Python.Desugar.Loops}
\]

\[
\circ \text{Rewrite and analyze recursively}
\]

\[
\circ \text{Optimize for some semantic cases}
\]

\[
\text{it} = \text{iter(\text{iterable})}
\text{while(1)}\
\try: \text{target} = \text{next(\text{it})}
\except \text{StopIteration: break}
\text{body}
\]

\[
\text{clean it}
\]
Expression rewriting

$\xi_{\text{env}}[m = m + l[i].weight]_{\text{env}}\sigma^\#$

$\xi_{\text{env}}[m = m + l[i].weight]_{\text{env}}\sigma^\#$
Expression rewriting

\[ S_{\text{env}} \left[ m = m + l[i].\text{weight} \right] \sigma^R_{\text{env}} \]
\[ \xrightarrow{\text{E}_{\text{binop}}} m + l[i].\text{weight} \sigma^R_{\text{env}} \]
Expression rewriting

\[ S_{\text{env}}^\#(m \equiv m + l[i].\text{weight})_{\text{env}}^{\sigma^\sharp} \]

\[ \xrightarrow{E_{\text{binop}}} m + l[i].\text{weight} \sigma^\sharp \]

\[ \xrightarrow{E_{\text{env}}} m \sigma^\sharp \]

\[ \langle @^{\sharp}_{\text{int}}, \text{int}(m) \rangle, \sigma^\sharp \]
Expression rewriting

\[ \mathcal{E}_{\text{env}}[m = m + l[i].\text{weight}]_{\text{env}} \sigma^2 \]
\[ \mathcal{E}_{\text{binop}}[m + l[i].\text{weight}]_{\sigma^2} \]
\[ \mathcal{E}_{\text{env}}[m]_{\sigma^2} \]
\[ (\@_{\text{int}}, \text{int}(m)), \sigma^2 \]
\[ \mathcal{E}_{\text{attr}}[l[i].\text{weight}]_{\sigma^2} \]
Expression rewriting

\[ \text{env}_m^{\#} \xrightarrow{\text{binop}} \text{env}_m^{\#} + \text{env}_{[i]}^{\#} \]

\[ \xrightarrow{\text{env}_m^{\#}} \]

\[ \langle @\text{int}, \text{int}(m) \rangle, \sigma^{\#} \]

\[ \xrightarrow{\text{env}_m^{\#}} \]

\[ \langle @\text{int}, \text{int}(m) \rangle, \sigma^{\#} \]

\[ \xrightarrow{\text{env}_m^{\#}} \]

\[ \langle @\text{int}, \text{int}(m) \rangle, \sigma^{\#} \]
Expression rewriting

\[ S_{\text{env}}^\# \left[ m = m + \ell[i].weight \right]_{\text{env}}^\# \sigma^z \]
\[ \rightarrow E_{\text{binop}}^\# \left[ m + \ell[i].weight \right]_{\sigma^z}^\# \]
\[ \rightarrow E_{\text{env}}^\# \left[ m \right]_{\sigma^z}^\# \]
\[ \left\langle @_{\text{int}}, \text{int}(m), \sigma^z \right\rangle \]
\[ \rightarrow E_{\text{attrs}}^\# \left[ \ell[i].weight \right]_{\sigma^z}^\# \]
\[ \rightarrow E_{\text{index}}^\# \left[ \ell[i] \right]_{\sigma^z}^\# \]
\[ \left\langle \sigma^z \right\rangle \]
\[ \left\langle @_{\text{list,r}}, \bot, \sigma^z \right\rangle \]
Expression rewriting

\[ S_{\text{env}}^\# \{ m = m + l[i].\text{weight} \} \}_{\text{env}}^\# \sigma^z \]

\[ \xrightarrow{E_{\text{binop}}^\#} m + l[i].\text{weight} \}_{\text{env}}^\# \sigma^z \]

\[ \xrightarrow{E_{\text{env}}^\#} \] \( m \}_{\text{env}}^\# \sigma^z \)

\[ \langle @_{\text{int}}, \text{int}(m) \rangle, \sigma^z \]

\[ \xrightarrow{E_{\text{attrs}}^\#} [l[i].\text{weight}] \}_{\text{env}}^\# \sigma^z \]

\[ \xrightarrow{E_{\text{index}}^\#} [l[i]] \}_{\text{env}}^\# \sigma^z \]

\[ \xrightarrow{E_{\text{env}}^\#} [l] \}_{\text{env}}^\# \sigma^z \]

\[ \langle @_{\text{list}}, \bot \rangle, \sigma^z \]

\[ \xrightarrow{E_{\text{env}}^\#} [i] \}_{\text{env}}^\# \sigma^z \]

\[ \langle @_{\text{int}}, \text{int}(i) \rangle, \sigma^z \]
Expression rewriting

$$S_{\text{env}}^\# [ m = m + (l[i].weight)]^\#_{\text{env}} \sigma^z$$

$$E_{\text{binop}}^\# m + (l[i].weight) \sigma^z$$

$$E_{\text{env}}^\# m \sigma^z$$

$$\langle @\text{int}^\# \cdot \text{int}(m) \rangle, \sigma^z$$

$$E_{\text{attr}}^\# (l[i].weight) \sigma^z$$

$$E_{\text{index}}^\# (l[i]) \sigma^z$$

$$E_{\text{env}}^\# l \sigma^z$$

$$\langle @\text{list, r}^\# \cdot \bot \rangle, \sigma^z$$

$$E_{\text{env}}^\# i \sigma^z$$

$$\langle @\text{int}^\# \cdot \text{int}(i) \rangle, \sigma^z$$

$$E_{\text{list}}^\# \text{list.__getitem__}(l[i]) \sigma^z$$

$$S_{\text{num}}^\# \text{assume} \ 0 \leq \text{int}(i) < \text{len}(\text{@list, r}) \sigma^z$$

$$E_{\text{env}}^\# \text{els}(\text{@list, r}, \bot) \sigma^z$$

$$\langle @\text{task, r}^\# \cdot \bot \rangle, \sigma^z$$
Expression rewriting

\[
S^\#_{\text{env}}[m = m + \langle [l].weight \rangle_{\text{env}} \sigma^z] \\
\xrightarrow{E^\#_{\text{binop}}}[m + \langle [l].weight \rangle]_{\sigma^z} \\
\xrightarrow{E^\#_{\text{env}}}[m]_{\sigma^z} \\
\langle \@_{\text{int}}, \text{int}(m) \rangle, \sigma^z \\
\xrightarrow{E^\#_{\text{attrs}}}[\langle [l].weight \rangle]_{\sigma^z} \\
\xrightarrow{E^\#_{\text{index}}}[\langle [l] \rangle]_{\sigma^z} \\
\xrightarrow{E^\#_{\text{env}}}[\langle [l] \rangle]_{\sigma^z} \\
\langle \@_{\text{list}, r}, \bot \rangle, \sigma^z \\
\langle \@_{\text{int}}, \text{int}(i) \rangle, \sigma^z \\
\xrightarrow{E^\#_{\text{list}}}[\text{list}_{\text{getitem}}(\langle [l], \text{"weight"} \rangle)]_{\sigma^z} \\
\xrightarrow{E^\#_{\text{num}}}[\text{assume} \ 0 \leq \text{int}(i) < \text{len}(\@_{\text{list}, r})]_{\sigma^z} \\
\xrightarrow{E^\#_{\text{env}}}[\text{els}(\@_{\text{list}, r})]_{\sigma^z} \\
\langle \@_{\text{task}, r}, \bot \rangle, \sigma^z \\
\xrightarrow{E^\#_{\text{object}}}[\text{object}_{\text{getattr}}(\langle [l], \text{"weight"} \rangle)]_{\sigma^z} \\
\xrightarrow{E^\#_{\text{heap}}}[\text{get_field}(\langle [l], \text{"weight"} \rangle)]_{\sigma^z} \\
\xrightarrow{E^\#_{\text{env}}}[\langle [l] \cdot \text{weight} \rangle]_{\sigma^z} \\
\langle \@_{\text{int}}, \text{int}(\@_{\text{task}, r} \cdot \text{weight}) \rangle, \sigma^z
\]
Towards a Multilanguage Analysis
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()
```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);
}
```

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

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

counter.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

from counter import Counter
from random import randrange

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

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

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

```python
import Counter
from random import randrange

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

▶ 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
```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);
}
```

```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

▶ 32 ≤ power ≤ 64: OverflowError: signed integer is greater than maximum

▶ power ≥ 64: OverflowError: Python int too large to convert to C long
Multilanguage code – example

counter.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

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

```python
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 ➞ missed errors
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  ⇒  missed errors
▶ Only types
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
### 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 previous work:
  Monat, Ouadjaout, and Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP 2020.
# 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

```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)
### 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
- 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
How to analyze multilanguage programs?

- Type annotations
- Rewrite into Python code
- Drawbacks of the current approaches
  - Not the real code

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?

<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?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite into Python code</td>
</tr>
</tbody>
</table>

## 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><strong>Drawbacks of the current approaches</strong></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><strong>Our approach</strong></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?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Rewrite into Python code</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Drawbacks of the current approaches</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Our approach</th>
</tr>
</thead>
</table>

- 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
High-level idea

**Difficulty: shared memory**

- Two distinct visions of a shared state
- Synchronization? We could perform a full state translation, but
  - the cost would be high in the analysis
  - some abstractions can be shared between Python and C
High-level idea

**Difficulty: shared memory**

- Two distinct visions of a shared state
- Synchronization? We could perform a full state translation, but
  - the cost would be high in the analysis
  - some abstractions can be shared between Python and C

**State separation ǂ reduced synchronization**

- Observation: structures are directly dereferenceable by one language only
- Switch to other language otherwise (\texttt{c.incr()} ǂ \texttt{self->count += 1})
  
  Additional hypothesis: C accesses to Python objects through the API
- Synchronization: only when objects change language for the first time
Implementation & Experimental Evaluation
From distinct Python and C analyses...
... to a multilanguage analysis!

CPython API

- Py.program
- Py.desugar
- Py.exceptions
- Py.libraries
- Py.objects
- Py.data_model

C.program
- C.desugar
- C.goto
- C.stubs
- C.libraries
- C.files

U.intraproc
- U.loops
- U.interproc
- U.recency
- U.intervals

\[\times\] Universal
\[\circ\] C specific
\[\bullet\] Python specific

\[\otimes\] Sequence
\[\times\] Reduced product
\[\otimes\] Cartesian product
\[\circlearrowleft\] Composition

Implementation LOC

Universal: 5600
Python: 12600
C: 11700
Multilanguage: 2500

Part LOC

Framework

13200
... to a multilanguage analysis!
... to a multilanguage analysis!

**CPython API**

- Py.program
- Py.deugar
- Py.exceptions
- Py.libraries
- Py.objects
- Py.data_model

**C.program**

- C.goto
- C.stubs
- C.libraries
- C.files

**C.cells**

- C.strings

**U.intraproc**

- U.loops
- U.interproc

**U.recency**

- U.intervals

**Implementation LOC**

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

**Part LOC**

- Composition
- Reduced product
- Cartesian product
- Universal
- C specific
- Python specific
... 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>

### Python API

- `Py.program`
- `Py.desugar`
- `Py.exceptions`
- `Py.libraries`
- `Py.objects`
- `Py.data_model`

### C API

- `C.program`
- `C.desugar`
- `C.goto`
- `C.stubs`
- `C.files`
- `C.cells`
- `C.strings`

### LOCs

- **Universal**: 5600 LOC
- **Python specific**: 12600 LOC
- **C specific**: 11700 LOC
- **Sequence**: 438 LOC
- **Reduced product**: 250 LOC
- **Cartesian product**: 18 LOC
- **Composition**: 18 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>
<tr>
<td>C</td>
<td>11700</td>
</tr>
</tbody>
</table>

### Implementation LOC Breakdown

- **Universal**: 5600 LOC
- **C** specific: 11700 LOC
- **Python** specific: 12600 LOC
- **Multilanguage**: 2500 LOC

### Framework LOC

- **Framework**: 13200 LOC

### Part LOC

- **Universal**: 5600 LOC
- **C**: 11700 LOC
- **Python** specific: 12600 LOC
- **Multilanguage**: 2500 LOC

### Implementation LOC Notes

- **Universal**: 5600 LOC
- **C** specific: 11700 LOC
- **Python** specific: 12600 LOC
- **Multilanguage**: 2500 LOC

### Framework LOC Breakdown

- **Universal**: 5600 LOC
- **C**: 11700 LOC
- **Python** specific: 12600 LOC
- **Multilanguage**: 2500 LOC

### Part LOC Breakdown

- **Universal**: 5600 LOC
- **C** specific: 11700 LOC
- **Python** specific: 12600 LOC
- **Multilanguage**: 2500 LOC

### Implementation LOC Diagram

- **Universal**: 5600 LOC
- **C** specific: 11700 LOC
- **Python** specific: 12600 LOC
- **Multilanguage**: 2500 LOC

### Framework LOC Diagram

- **Universal**: 5600 LOC
- **C**: 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>
<tr>
<td>C</td>
<td>11700</td>
</tr>
<tr>
<td>Python</td>
<td>12600</td>
</tr>
</tbody>
</table>

- Universal
- C specific
- Python specific

- Universal

- Reduced product
- Cartesian product
- Composition
... 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>

**Implementation LOC**

- Universal: 5600 LOC
- C specific: 11700 LOC
- Python specific: 12600 LOC
- Multilanguage: 2500 LOC

**Implementation LOC**

- Framework: 13200 LOC
- Universal: 5600 LOC
- C: 11700 LOC
- Python: 12600 LOC
- Multilanguage: 2500 LOC

**Implementation LOC**

- Universal: 5600 LOC
- C specific: 11700 LOC
- Python specific: 12600 LOC
- Multilanguage: 2500 LOC
## Benchmarks

### Corpus selection

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

<table>
<thead>
<tr>
<th>Library</th>
<th>C + Py. Loc</th>
<th>Tests</th>
<th>📊/test</th>
<th># proved checks/ # checks</th>
<th>%</th>
<th># checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>noise</td>
<td>1397</td>
<td>15/15</td>
<td>1.2s</td>
<td>99.7%</td>
<td>6690</td>
<td></td>
</tr>
<tr>
<td>cdistance</td>
<td>2345</td>
<td>28/28</td>
<td>4.1s</td>
<td>98.0%</td>
<td>13716</td>
<td></td>
</tr>
<tr>
<td>llist</td>
<td>4515</td>
<td>167/194</td>
<td>1.5s</td>
<td>98.8%</td>
<td>36255</td>
<td></td>
</tr>
<tr>
<td>ahocorasick</td>
<td>4877</td>
<td>46/92</td>
<td>1.2s</td>
<td>96.7%</td>
<td>6722</td>
<td></td>
</tr>
<tr>
<td>levenshtein</td>
<td>5798</td>
<td>17/17</td>
<td>5.3s</td>
<td>84.6%</td>
<td>4825</td>
<td></td>
</tr>
<tr>
<td>bitarray</td>
<td>5841</td>
<td>159/216</td>
<td>1.6s</td>
<td>94.9%</td>
<td>25566</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion
Conclusion

Difficulties

- Concrete semantics
- Memory interaction

---

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

<table>
<thead>
<tr>
<th>Difficulties</th>
<th>Previous works</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Concrete semantics</td>
<td>▶ Type/exceptions analyses for the JNI</td>
</tr>
<tr>
<td>▶ Memory interaction</td>
<td>▶ No detection of runtime errors in C</td>
</tr>
</tbody>
</table>

---

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

### Difficulties
- Concrete semantics
- Memory interaction

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

### 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, Ouadiaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021
A Multilanguage Static Analysis of Python/C Programs with Mopsa

Questions

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné