A Multilanguage Static Analysis of Python/C Programs with Mopsa

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné
Introduction
**Static Program Analysis**

average.py

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

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

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

argslen.c

```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;
}
```

No alarm

**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**: cover all possible executions.
Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub

1 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 GitHub

### New features

- Object orientation,
- Dynamic typing,
Dynamic programming languages

Growing popularity
JavaScript #1, Python #2 on GitHub\(^1\)

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

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

Growing popularity
JavaScript #1, Python #2 on GitHub\(^1\)

New features

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

\(^1\)https://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.

\(^1\)https://octoverse.github.com/#top-languages
## Python’s specificities

### No standard

- CPython is the reference
  - manual inspection of the source code and handcrafted tests

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

a = Protected(42)
a._priv  # AttributeError raised
```
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("...")
        return object.__getattribute__(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)

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

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

---

Combining C and Python – motivation

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

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

- To bring better performance (numpy)

Pitfalls
- Different values (arbitrary-precision integers in Python, bounded in C)
- Different runtime-errors (exceptions in Python)
- Garbage collection
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

**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 runtime-errors (exceptions in Python)
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 runtime-errors (exceptions in Python)
- Garbage collection
Outline

1 Introduction

2 Mopsa

3 A Concrete Example

4 Concrete Multilanguage Semantics

5 Implementation & Experimental Evaluation

6 Conclusion
Mopsa
A program analysis workflow

Avering numbers

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

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))]

```bash
m = average(l)
```

Proved safe?

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

Searching for a loop invariant (l. 4)

Environment abstraction

\[ m \mapsto \int, \quad i \mapsto \int \]

Conclusion

- Different domains depending on the precision
- Use of auxiliary variables (underlined)
A program analysis workflow

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: **list content**, **list length**

Environment abstraction

```
m \rightarrow \@_{\text{int}}\# \quad i \rightarrow \@_{\text{int}}\# \quad \text{els}(l) \rightarrow \@_{\text{int}}\#
```

Searching for a loop invariant (l. 4)

Proved safe?

- `m // (i+1)`
- `l[i]`
A program analysis workflow

Avering numbers

```
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 (intervals)

- m ∈ [0, +∞)
- els(l) ∈ [0, 20]
- i ∈ [0, +∞)

Conclusion

- Different domains depending on the precision
- Use of auxiliary variables (underlined)
A program analysis workflow

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)

- **Environment abstraction**
  
  \[ m \mapsto \mathbb{#}_{\text{int}}, \quad i \mapsto \mathbb{#}_{\text{int}}, \quad \text{els}(l) \mapsto \mathbb{#}_{\text{int}} \]

- **Numeric abstraction (intervals)**
  
  \[ m \in [0, +\infty), \quad \text{els}(l) \in [0, 20], \quad \text{len}(l) \in [5, 10], \quad i \in [0, 10] \]

Proved safe?

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

### Avering numbers

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

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

Proved safe?

- ▶ \( m \div (i+1) \)
- ▶ \( l[i] \)

Searching for a loop invariant (l. 4)

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

**Environment abstraction**

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

**Numeric abstraction (polyhedra)**

\[
m \in [0, +\infty) \quad \texttt{els}(l) \in [0, 20] \\
0 \leq i < \texttt{len}(l) \quad 5 \leq \texttt{len}(l) \leq 10
\]
A program analysis workflow

### Averaging tasks

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

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

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

**Proved safe?**

- `m // (i+1)`
- `l[i].weight`

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

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

#### Environment abstraction

- `m` ↦ `int`
- `i` ↦ `int`
- `els(l)` ↦ `int`
- `Task·weight` ↦ `int`

#### Numeric abstraction (polyhedra)

- `m ∈ [0, +∞)`
- `0 ≤ i < len(l)`
- `5 ≤ len(l) ≤ 10`
- `0 ≤ Task·weight ≤ 20`

#### Attributes abstraction

- `Task` ↦ `{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

\[
m \mapsto @\text{Task} \cdot \text{weight} \\rightarrow \\{\text{weight}\} \\mapsto \emptyset
\]

#### Attributes abstraction

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

**Conclusion**

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

**Proved safe?**

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

**Modular Open Platform for Static Analysis**

gitlab.com/mopsa/mopsa-analyzer

---

Overview of Mopsa

Modular Open Platform for Static Analysis

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

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

---

Overview of Mopsa

**Modular Open Platform for Static Analysis**
[gitlab.com/mopsa/mopsa-analyzer](gitlab.com/mopsa/mopsa-analyzer)

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

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

---

Overview of Mopsa

Modular Open Platform for Static Analysis

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

---

Overview of Mopsa

Modular Open Platform for Static Analysis

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

for(init; cond; incr) body

C.iterators.loops

Rewrite and analyze recursively

C.iterators.loops

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, target

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

```
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, target

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;
}
clean init

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

for(init; cond; incr) body

for target in iterable: body

C.iterators.loops

Rewrite and analyze recursively

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

Universal.Iterators.Loops

Matches while(...) {...}
Computes fixpoint using widening

Python.Desugar.Loops

◦ Rewrite and analyze recursively

◦ Optimize for some semantic cases
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

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;
}
clean init

Universal.Iterators.Loops
Matches while(...){...}
Computes fixpoint using widening

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, target
A Concrete Example
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rom random import randrange
c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```
# Combining C and Python – example

```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** ⇒ \( r = 2^\text{power} \)
- **power ≥ 64**: OverflowError: Python int too large to convert to C long

\[ \text{power} \geq 32 : \text{OverflowError: signed integer is greater than maximum} \]
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**

- **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

```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.

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

```python
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

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

- No raised exceptions $\Rightarrow$ missed errors
- Only types
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 $\implies$ missed errors
- Only types
- Typeshed: type annotations for the standard library
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
- 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

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

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)
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
How to analyze multilanguage programs?

- **Type annotations**
- **Rewrite into Python code**
- **Drawbacks of the current approaches**
  - 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><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>
</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>
<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>
### 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>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th><strong>Our approach</strong></th>
</tr>
</thead>
<tbody>
<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>
<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>
<tr>
<td>▶ Reuse previous analyses of C and Python</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
- 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, 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()

d = randrange(128)
c.incr(2**d-1)
c.incr()
r = c.get
Concrete Multilanguage Semantics
Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
Multilanguage semantics

Concrete definition

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

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

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
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
Implementation & Experimental Evaluation
From distinct Python and C analyses...
... to a multilanguage analysis!
... to a multilanguage analysis!

Implementation LOC

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
</table>

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

Universal
C specific
Python specific

Sequence
Reduced product
Cartesian product
Composition

Implementation LOC

CPython

Py.desugar
Py.objects
Py.libraries
Py.environment
Py.lists
Py.dicts
Py.tuples

C.goto
C.files
C.strings
C.pointers
C.machineNum
C.cells

U.loops
U.interproc
U.loops
U.intervals

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>
</tbody>
</table>

- Universal LOC: 13200
- Python specific LOC: 12600
- C specific LOC: 11700
- Sequence LOC: 3878
- Reduced product LOC: 443
- Cartesian product LOC: 2022
- Composition LOC: 1859
- Part LOC: 5600

CPython

Py program
Py.destructure
Py.exceptions
Py.libraries
Py.objects
Py.data_model
Py.Environment
Py.Attributes
Py.lists
Py.dicts
Py.tuples

C.program
C.desugar
C.goto
C.libraries
C.files
C.strings
C.machineNum
C.pointers

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

Sequence
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>
</tbody>
</table>
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 C</td>
<td>5600</td>
</tr>
<tr>
<td>C</td>
<td>11700</td>
</tr>
</tbody>
</table>

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>
<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>

15
... 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>
## 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</th>
<th>Py</th>
<th>Tests</th>
<th>Safe C checks</th>
<th>Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>noise</td>
<td>722</td>
<td>675</td>
<td>15/15</td>
<td>99.6% (4952)</td>
<td>0/21</td>
</tr>
<tr>
<td>ahocorasick</td>
<td>3541</td>
<td>1336</td>
<td>46/92</td>
<td>93.1% (1785)</td>
<td>30/88</td>
</tr>
<tr>
<td>levenshtein</td>
<td>5441</td>
<td>357</td>
<td>17/17</td>
<td>79.9% (3106)</td>
<td>0/38</td>
</tr>
<tr>
<td>cdistance</td>
<td>1433</td>
<td>912</td>
<td>28/28</td>
<td>95.3% (1832)</td>
<td>88/207</td>
</tr>
<tr>
<td>llist</td>
<td>2829</td>
<td>1686</td>
<td>167/194</td>
<td>99.0% (5311)</td>
<td>235/691</td>
</tr>
<tr>
<td>bitarray</td>
<td>3244</td>
<td>2597</td>
<td>159/216</td>
<td>96.3% (4496)</td>
<td>100/378</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>18s</th>
<th>54s</th>
<th>1.5m</th>
<th>1.9m</th>
<th>4.2m</th>
<th>4.2m</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>average # transitions between Python and C</td>
<td>6.5</td>
<td>5.4</td>
<td>2.7</td>
<td>8.7</td>
<td>51.7</td>
<td>14.8</td>
<td></td>
</tr>
</tbody>
</table>

- **Total C checks:**
  - Safe C checks: 15/15, 46/92, 17/17, 28/28, 167/194, 159/216
  - Total C checks: 15/15, 46/92, 17/17, 28/28, 167/194, 159/216

- **Safe C checks %:**
  - Noise: 99.6%
  - Ahocorasick: 93.1%
  - Levenshtein: 79.9%
  - Cdistance: 95.3%
  - Llist: 99.0%
  - Bitarray: 96.3%

- **Total C checks:**
  - Noise: 18s, 54s, 1.5m, 1.9m, 4.2m
  - Ahocorasick: 54s, 1.5m, 1.9m, 4.2m
  - Levenshtein: 1.5m, 1.9m, 4.2m
  - Cdistance: 1.9m, 4.2m
  - Llist: 4.2m
  - Bitarray: 4.2m

- **Assertions:**
  - Noise: 0/21
  - Ahocorasick: 30/88
  - Levenshtein: 0/38
  - Cdistance: 88/207
  - Llist: 235/691
  - Bitarray: 100/378
Conclusion
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

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
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

### 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
## Future works

### Multilanguage analyses

- Other interoperability frameworks (Cffi, Swig, Cython)
- Bigger applications
Future works

Multilanguage analyses

- Other interoperability frameworks (Cffi, Swig, Cython)
- Bigger applications

Library analyses

- Library analysis without client code
- Infer Typeshed’s annotations
A Multilanguage Static Analysis of Python/C Programs with Mopsa

Looking for
- RA / Postdoc position
- Starting fall 2022
- In the UK or Europe
- raphael.monat@lip6.fr

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

Facebook TAV
1 December 2021