A Multilanguage Static Analysis of Python/C Programs with Mopsa

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

Software Languages Lab, VUB
7 July 2022
Introduction
Program verification

Sound

All true errors are reported
Program verification

All reported errors are true errors

Complete

Sound

All true errors are reported
Program verification

Guaranteed Termination

Complete

Sound

All reported errors are true errors

All true errors are reported
Program verification

All reported errors are true errors

Guaranteed Termination

Sound

Complete

Rice’s theorem

All true errors are reported

∅
Program verification

All reported errors are true errors

Guaranteed Termination

Sound

Complete

Abstract Interpretation

Rice’s theorem

∅

All true errors are reported
Conservative 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'])
```

TypeError: 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;
}
```

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.
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%
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
Combining C and Python – motivation

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

▶ To bring better performance (numpy)
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

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 (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
Outline

1. Introduction
2. A Taste of Python
3. Mopsa
4. Towards a Multilanguage Semantics
5. Implementation & Experimental Evaluation
6. 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__

Protected attributes
```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)

Exceptions

Exceptions rather than specific values

\[1 + "a" \Rightarrow \text{TypeError}\]

\[l[l\text{en}(l) + 1] \Rightarrow \text{IndexError}\]

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())
```

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

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

---

Example Semantics – binary operators

\[ a_1 = \text{eval } e_1; \ a_2 = \text{eval } e_2 \]

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

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/
Mopsa
A program analysis workflow

Averaging 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

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)

Environment abstraction

```
m ↦ int
i ↦ int
```

Conclusion

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

Proved safe?

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

Searching for a loop invariant (l. 4)

Environment abstraction

$m \mapsto \@\text{int}\@$ $i \mapsto \@\text{int}\@$ $\text{els}(l) \mapsto \@\text{int}\@$
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
```

Proved safe?

▶ m // (i+1)
▶ l[i]

Searching for a loop invariant (l. 4)

Stateless domains: list content,

<table>
<thead>
<tr>
<th>Environment abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m \mapsto \mathbb{I}_{\text{int}} )</td>
</tr>
</tbody>
</table>

Numeric abstraction (intervals)

\( m \in [0, +\infty) \) \( \text{els}(l) \in [0, 20] \) \( i \in [0, +\infty) \)
A program analysis workflow

Averaging 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

Searching for a loop invariant (l. 4)

Proved safe?

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

Environment abstraction

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

Numeric abstraction (intervals)

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

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

Searching for a loop invariant (l. 4)

Proved safe?

- \( \frac{m}{i+1} \)
- \( l[i] \)

Environment abstraction

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

Numeric abstraction (polyhedra)

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

Averaging tasks

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

    def average(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)
```

Proved safe?

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

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

\[ m \mapsto \mathbb{I}_{\text{int}} \quad i \mapsto \mathbb{I}_{\text{int}} \quad \text{els}(l) \mapsto \mathbb{I}_{\text{Task}} \]

\[ \mathbb{I}_{\text{Task}} \cdot \text{weight} \mapsto \mathbb{I}_{\text{int}} \]

Numeric abstraction (polyhedra)

\[ m \in [0, +\infty) \]
\[ 0 \leq i < \text{len}(l) \quad 5 \leq \text{len}(l) \leq 10 \]
\[ 0 \leq \mathbb{I}_{\text{Task}} \cdot \text{weight} \leq 20 \]

Attributes abstraction

\[ \mathbb{I}_{\text{Task}} \mapsto (\{ \text{weight} \}, \emptyset) \]
A program analysis workflow

Averaging tasks

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

    def average(self, 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

- $\mathbf{m} \mapsto @\#\text{Task}$
- $i \mapsto @\#\text{int}$
- $\#\text{Task} \mapsto \#\text{Task}$
- $\#\text{weight} \mapsto @\#\text{int}$

Conclusion

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

Attributes abstraction

- $@\#\text{Task} \mapsto (\{\text{weight}\}, \emptyset)$

Proved safe?

- $m // (i+1)$
- $l[i].weight$
Overview of Mopsa

**Modular Open Platform for Static Analysis**

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

---

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

---

Overview of Mopsa

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

\[\text{gitlab.com/mopsa/mopsa-analyzer}\]

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

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

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

Modular Open Platform for Static Analysis¹

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

---

1Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. 2019.
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
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
```

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
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

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

- 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

Universal.Iterators.Loops
Matches while(...){...}
Computes fixpoint using widening
Towards a Multilanguage Semantics
```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
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()
```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**

<table>
<thead>
<tr>
<th>counter.c</th>
<th>count.py</th>
</tr>
</thead>
</table>
|```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
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
```

```
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
```
```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);
}
```
```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
- 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

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

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

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

- No raised exceptions ➔ missed errors
- Only types
- Typeshed: type annotations for the standard library, used in previous work: Monat, Ouadjiaout, 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**

```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
- 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>
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>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>
<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>
### How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rewrite into Python code</strong></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>
</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>
<tr>
<td><strong>Our approach</strong></td>
</tr>
<tr>
<td>► Analyze both the C and Python sources</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>
<tbody>
<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>Our approach</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><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>
<tr>
<td>► Reuse previous analyses of C and Python</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>
<tbody>
<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>Our approach</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>
<tr>
<td>▶ Reuse previous analyses of C and Python</td>
</tr>
<tr>
<td>▶ Detect runtime errors in Python, in C, and at the boundary</td>
</tr>
</tbody>
</table>
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 $\leadsto$ reduced synchronization

- Observation: structures are directly dereferenceable by one language only
- Switch to other language otherwise ($c$.incr() $\leadsto$ 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!
... 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>

CPython API

Universal
C specific
Python specific

Sequence
Reduced product
Cartesian product
Composition

Implementation LOC

Part LOC
Framework 13200
... 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>
<tr>
<td>Universal C</td>
<td>5600</td>
</tr>
<tr>
<td>C</td>
<td>11700</td>
</tr>
</tbody>
</table>

- Universal
- C specific
- Python specific

- 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>
<tr>
<td>C</td>
<td>11700</td>
</tr>
<tr>
<td>Python</td>
<td>12600</td>
</tr>
</tbody>
</table>

- **CPython API**
- **CPython**
- **Python**
- **C**
- **Universal**
- **C specific**
- **Python specific**
- **Implementation LOC**
- **Part LOC**
- **Framework LOC**
- **Universal LOC**
- **C LOC**
- **Python 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>
<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 + Py. Loc</th>
<th>Tests</th>
<th># proved checks/# checks</th>
<th># checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>noise</td>
<td>1397</td>
<td>15/15</td>
<td>99.7%</td>
<td>6690</td>
</tr>
<tr>
<td>cdistance</td>
<td>2345</td>
<td>28/28</td>
<td>98.0%</td>
<td>13716</td>
</tr>
<tr>
<td>llist</td>
<td>4515</td>
<td>167/194</td>
<td>98.8%</td>
<td>36255</td>
</tr>
<tr>
<td>ahocorasic</td>
<td>4877</td>
<td>46/92</td>
<td>96.7%</td>
<td>6722</td>
</tr>
<tr>
<td>levenshtein</td>
<td>5798</td>
<td>17/17</td>
<td>84.6%</td>
<td>4825</td>
</tr>
<tr>
<td>bitarray</td>
<td>5841</td>
<td>159/216</td>
<td>94.9%</td>
<td>25566</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

### 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
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, Ouadjaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021
I’ll be a permanent researcher at Inria Lille from September 2022
I’ll be a permanent researcher at Inria Lille from September 2022

Considered topics:

- Ease the sound analysis of new languages
- Make static analysis more user-friendly
- Apply formal methods to legal expert systems
Future work

I’ll be a permanent researcher at Inria Lille from September 2022

Considered topics:

- Ease the sound analysis of new languages
- Make static analysis more user-friendly
- Apply formal methods to legal expert systems

⇝ I would be glad to set up new collaborations!
A Multilanguage Static Analysis of Python/C Programs with Mopsa

Questions

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

Software Languages Lab, VUB
7 July 2022
rmonat.fr