A Multilanguage Static Analysis of Python/C Programs with Mopsa

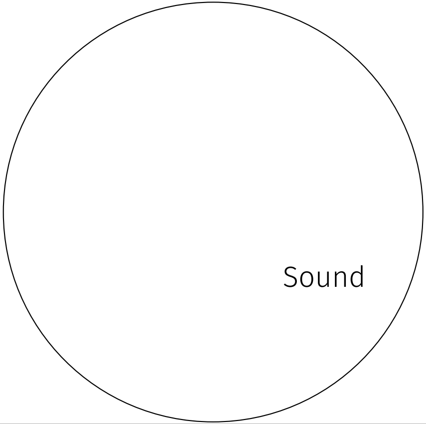
Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

Software Languages Lab, VUB
7 July 2022

rmonat.fr

Introduction

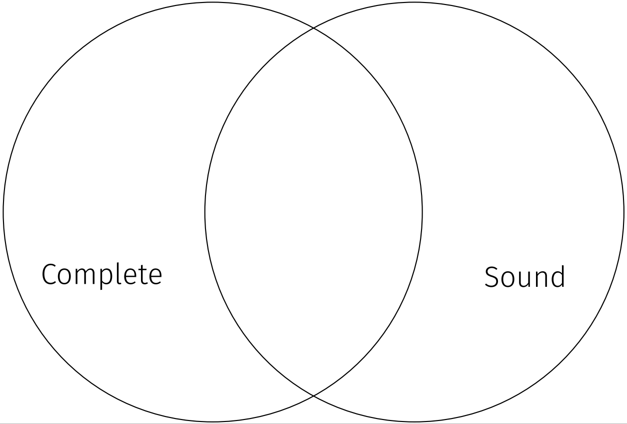
Program verification



All true errors
are reported

Program verification

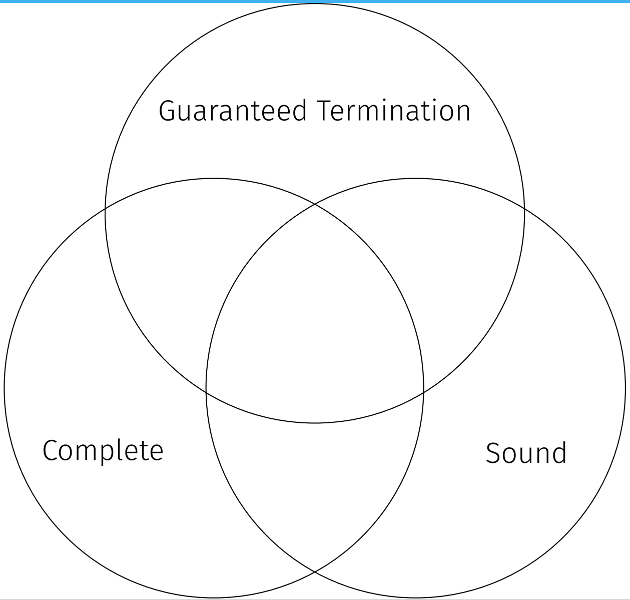
All reported errors are true errors



All true errors are reported

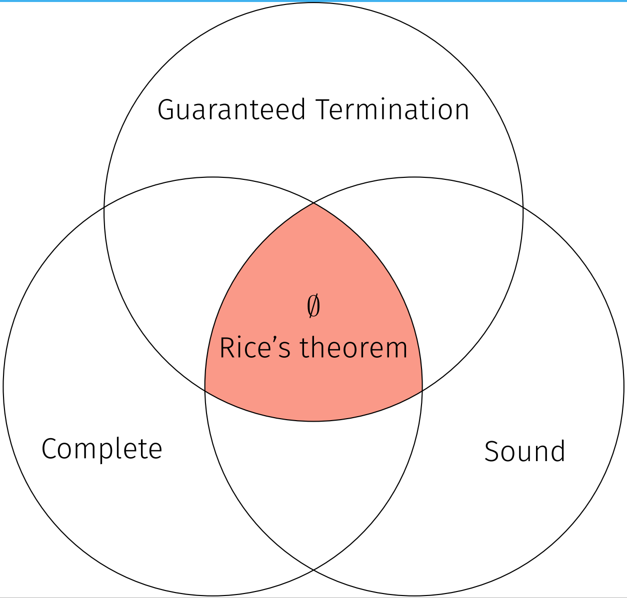
Program verification

All reported errors are true errors



All true errors are reported

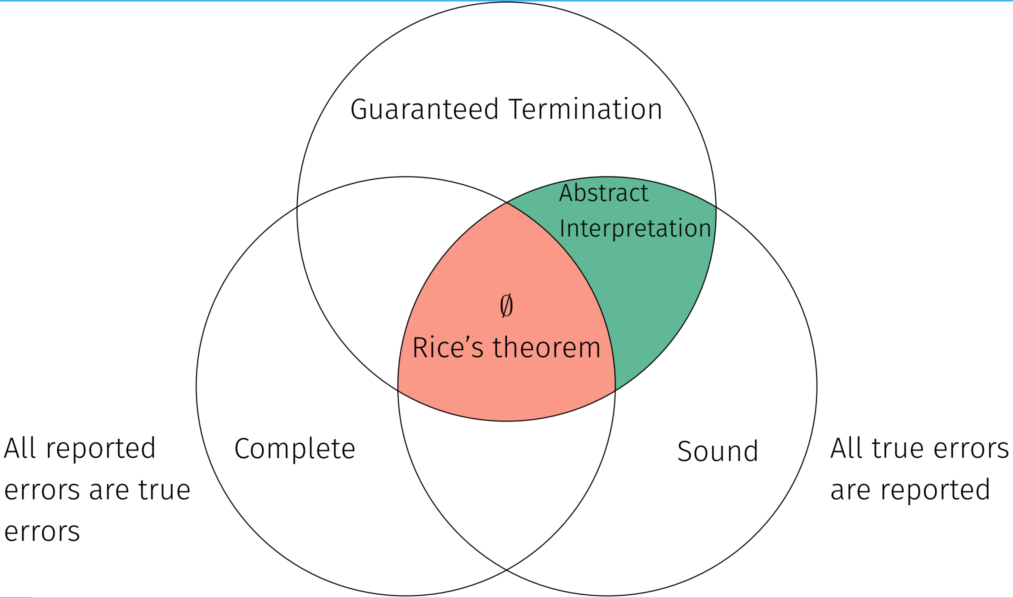
Program verification



All reported errors are true errors

All true errors are reported

Program verification



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;
}
```

No alarm

Specifications of the analyzer

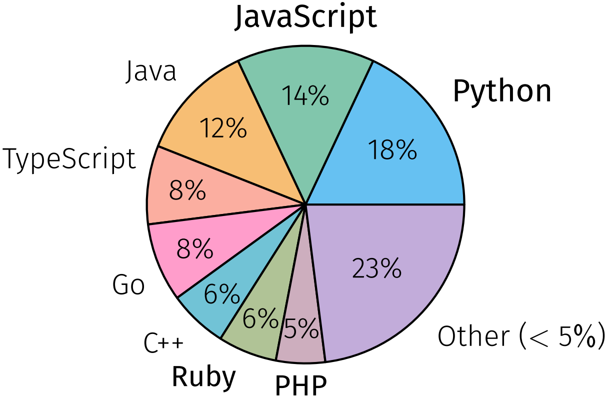
Inference of program properties such as the absence of run-time errors.

Semantic based on a formal modelization of the language.

Automatic no expert knowledge required.

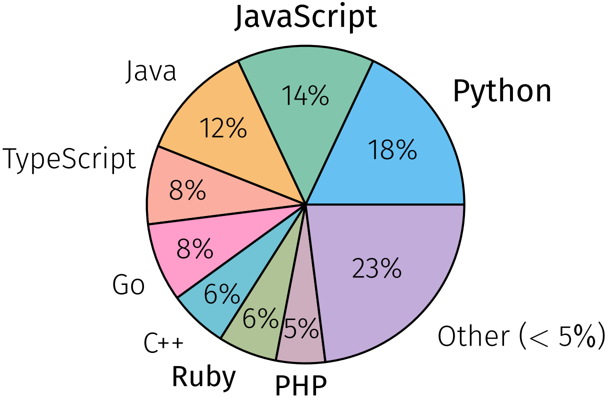
Sound covers all possible executions.

Dynamic programming languages



Most popular languages on GitHub

Dynamic programming languages



Most popular languages on GitHub

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 ($\mathbb{Z}$ 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

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

  $\Longrightarrow$ manual inspection of the source code and handcrafted tests

Python's specificities

No standard

- CPython is the reference

  $\Longrightarrow$ 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)

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())
```

Barrett, Cassels, Haahr, Moon, Playford, and Withington. "A Monotonic Superclass Linearization for Dylan". OOPSLA 1996

Python's specificities (II)

Dual type system

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

Fspath (from standard library)

```
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())
```

Barrett, Cassels, Haahr, Moon, Playford, and Withington. "A Monotonic Superclass Linearization for Dylan". OOPSLA 1996

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)

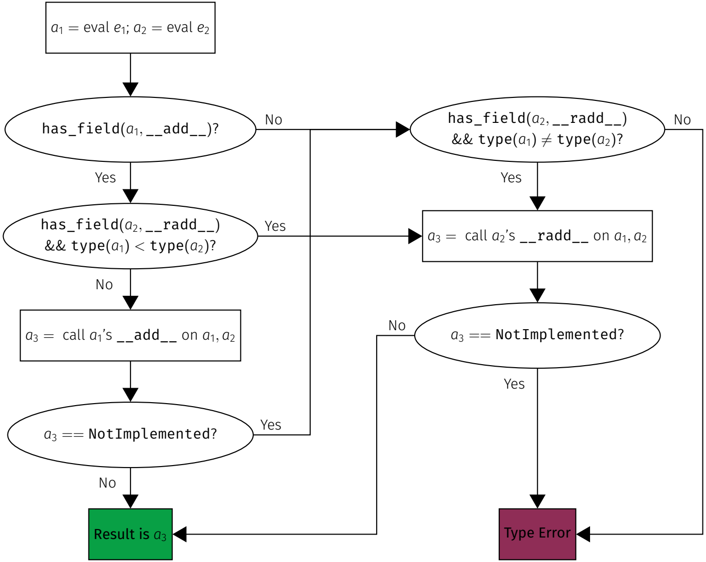
```python
class Path:
  def __fspath__(self): return 42

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())
```

Barrett, Cassels, Haahr, Moon, Playford, and Withington. "A Monotonic Superclass Linearization for Dylan". OOPSLA 1996

Example Semantics – binary operators



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/`

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

```
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

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

Searching for a loop invariant (l. 4)

Environment abstraction

$m \mapsto @^\sharp_{\texttt{int}^\sharp} \quad i \mapsto @^\sharp_{\texttt{int}^\sharp}$

A program analysis workflow

Averaging numbers

```
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

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

Searching for a loop invariant (l. 4)

Stateless domains: **list content**,

Environment abstraction

$m \mapsto @^\sharp_{\texttt{int}^\sharp} \quad i \mapsto @^\sharp_{\texttt{int}^\sharp} \quad \underline{\text{els}}(l) \mapsto @^\sharp_{\texttt{int}^\sharp}$

A program analysis workflow

Averaging numbers

```
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

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

Searching for a loop invariant (l. 4)

Stateless domains: list content,

Environment abstraction

$m \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad i \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad \underline{\text{els}}(l) \mapsto @^\sharp_{\texttt{int}^\sharp}$

Numeric abstraction (intervals)

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

A program analysis workflow

Averaging numbers

```
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

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

Searching for a loop invariant (l. 4)

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

Environment abstraction

$m \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad i \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad \underline{\text{els}}(l) \mapsto @^\sharp_{\texttt{int}^\sharp}$

Numeric abstraction (intervals)

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

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

A program analysis workflow

Averaging numbers

```
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

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

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

$m \mapsto @^\sharp_{\mathtt{int}^\sharp}$ $\quad i \mapsto @^\sharp_{\mathtt{int}^\sharp}$ $\quad \underline{\mathrm{els}}(l) \mapsto @^\sharp_{\mathtt{int}^\sharp}$

Numeric abstraction (polyhedra)

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

$\mathbf{0 \leq i < \underline{len}(l)}$ $\quad \mathbf{5 \leq \underline{len}(l) \leq 10}$

A program analysis workflow

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(len(l)):
9      m = m + l[i].weight
10   m = m // (i + 1)
11   return m
12
13 l = [Task(randint(0, 20))
14   for i in range(randint(5, 10))]
15 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

$m \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad i \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad \underline{\text{els}(l)} \mapsto @^\sharp_{\texttt{Task}}$

$\underline{@^\sharp_{\texttt{Task}} \cdot \texttt{weight}} \mapsto @^\sharp_{\texttt{int}^\sharp}$

Numeric abstraction (polyhedra)

$m \in [0, +\infty)$

$0 \le i < \underline{\text{len}}(l) \quad 5 \le \underline{\text{len}}(l) \le 10$

$0 \le \underline{@^\sharp_{\texttt{Task}} \cdot \texttt{weight}} \le 20$

Attributes abstraction

$@^\sharp_{\texttt{Task}} \mapsto (\{\texttt{weight}\}, \emptyset)$

A program analysis workflow

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(le
9      m = m + l[i].wei
10   m = m // (i + 1)
11   return m
12
13 l = [Task(randint(0,
14   for i in range(ran       5, 10))]
15 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

$m \mapsto @^\sharp \quad i \mapsto @^\sharp \quad @^\sharp_{\texttt{Task}}$

$0 \leq i < \underline{len}(l) \quad 5 \leq \underline{len}(l) \leq 10$

$0 \leq \underline{@^\sharp_{\texttt{Task}} \cdot \texttt{weight}} \leq 20$

Attributes abstraction

$@^\sharp_{\texttt{Task}} \mapsto (\{\texttt{weight}\}, \emptyset)$

Conclusion

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

Modular **O**pen **P**latform for **S**tatic **A**nalysis[1]

`gitlab.com/mopsa/mopsa-analyzer`

[1] Journault, Miné, Monat, and Ouadjaout. "Combinations of reusable abstract domains for a multilingual static analyzer". 2019.

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.

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

[1] Journault, Miné, Monat, and Ouadjaout. "Combinations of reusable abstract domains for a multilingual static analyzer". 2019.

Modular Open Platform for Static Analysis[1]

`gitlab.com/mopsa/mopsa-analyzer`

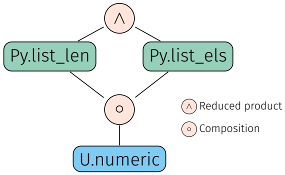
- One AST to analyze them all
  - Multilanguage support
  - Expressiveness
  - Reusability
- Unified domain signature
  - Semantic rewriting
  - Loose coupling
  - Observability
- DAG of abstract domains
  - Composition
  - Cooperation

[1] Journault, Miné, Monat, and Ouadjaout. "Combinations of reusable abstract domains for a multilingual static analyzer". 2019.

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.

Dynamic, semantic iterators with delegation

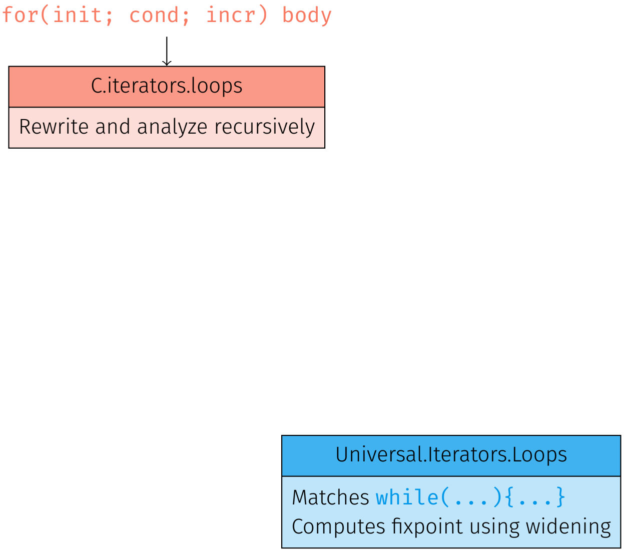
Universal.Iterators.Loops
Matches `while(...){...}` Computes fixpoint using widening

Dynamic, semantic iterators with delegation

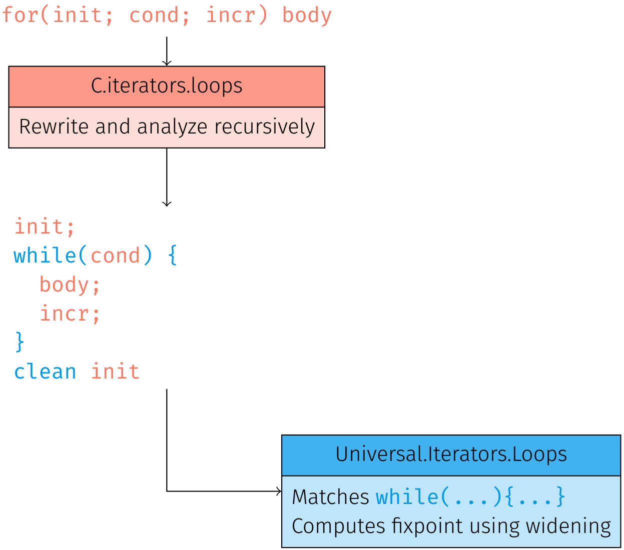
```
for(init; cond; incr) body
```

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

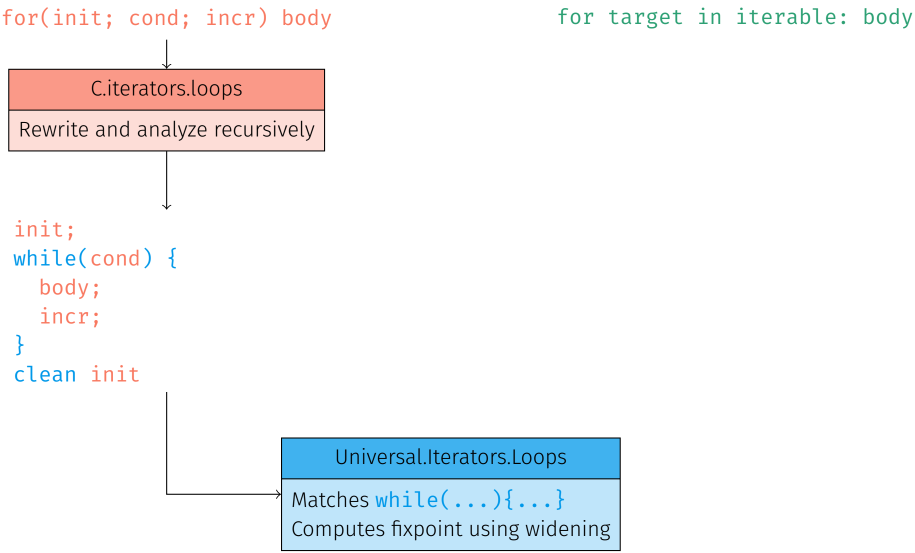
Dynamic, semantic iterators with delegation



Dynamic, semantic iterators with delegation



Dynamic, semantic iterators with delegation



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

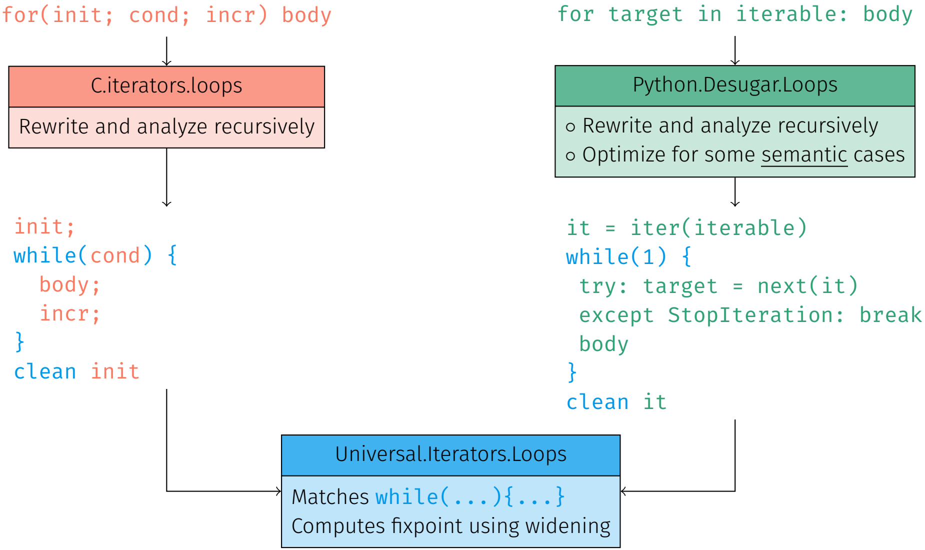
Matches `while(...){...}`
Computes fixpoint using widening

```
for target in iterable: body
```

Python.Desugar.Loops

- Rewrite and analyze recursively
- Optimize for some <u>semantic</u> cases

Dynamic, semantic iterators with delegation



Towards a Multilanguage Semantics

Multilanguage code – 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()
```

Multilanguage code – 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()
```

Multilanguage code – 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()
```

Multilanguage code – 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()
```

Multilanguage code – 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()
```

Multilanguage code – 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()
```

Multilanguage code – 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()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$

Multilanguage code – 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()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$
- $32 \leq \texttt{power} \leq 64$: OverflowError: signed integer is greater than maximum
- $\texttt{power} \geq 64$: OverflowError: Python int too large to convert to C long

Multilanguage code – 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()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$
- $\texttt{power} = 31 \Rightarrow \texttt{r} = -2^{31}$
- $32 \leq \texttt{power} \leq 64$: OverflowError: signed integer is greater than maximum
- $\texttt{power} \geq 64$: 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 $\Longrightarrow$ missed errors

How to analyze multilanguage programs?

Type annotations

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

- No raised exceptions $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ 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
```

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

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?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

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

How to analyze multilanguage programs?

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

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?

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

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

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

Our approach

- Analyze both the C and Python sources
- Switch from one language to the other just as the program does
- Reuse previous analyses of C and Python

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

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

Our approach

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

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
  - <u>some</u> abstractions can be shared between Python and C

High-level idea

Difficulty: shared memory

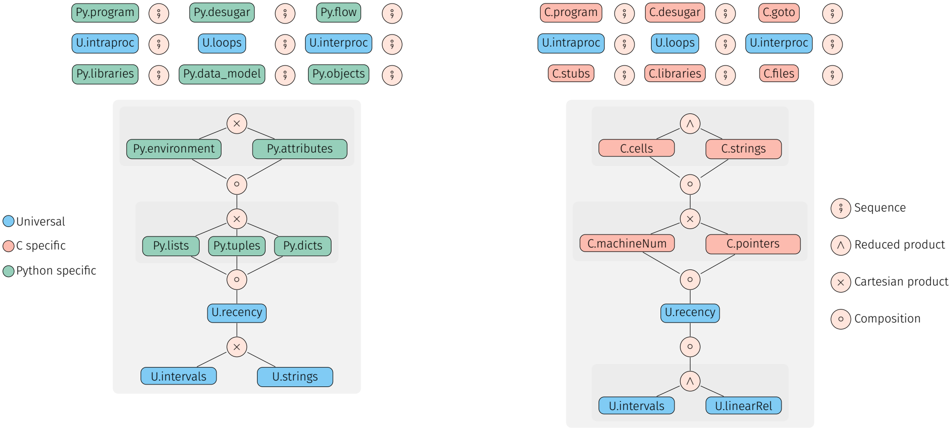
- Two distinct visions of a shared state
- Synchronization? We could perform a full state translation, but
  - the cost would be high in the analysis
  - some abstractions can be shared between Python and C

State separation $\rightsquigarrow$ reduced synchronization

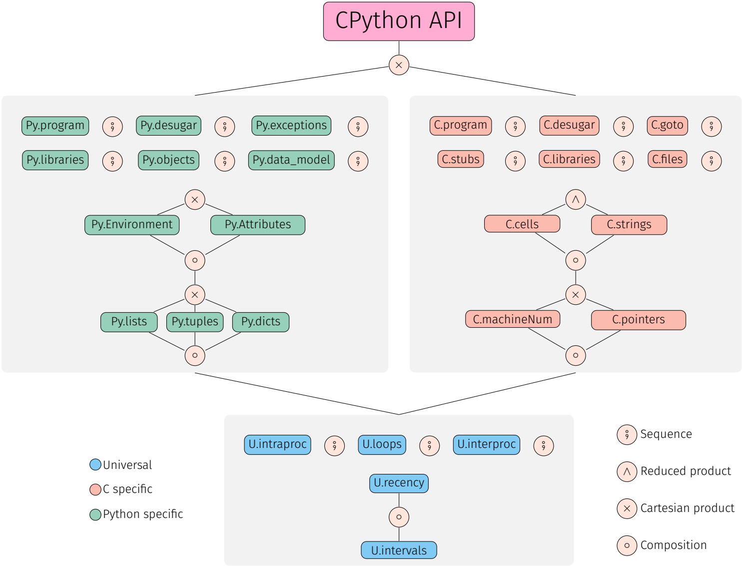
- Observation: structures are directly dereferenceable by one language only
- Switch to other language otherwise (`c.incr()` $\rightsquigarrow$ `self->count += 1`)
  Additional hypothesis: C accesses to Python objects through the API
- Synchronization: only when objects change language for the first time

Implementation & Experimental Evaluation

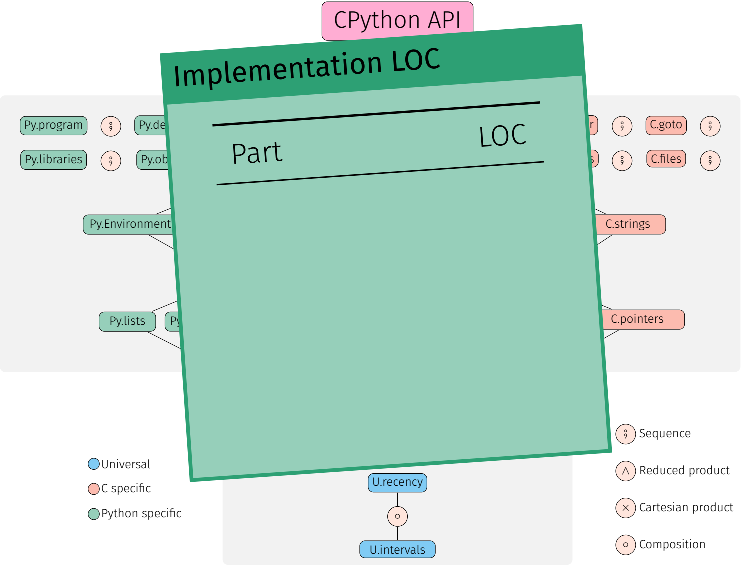
From distinct Python and C analyses...



... to a multilanguage analysis!



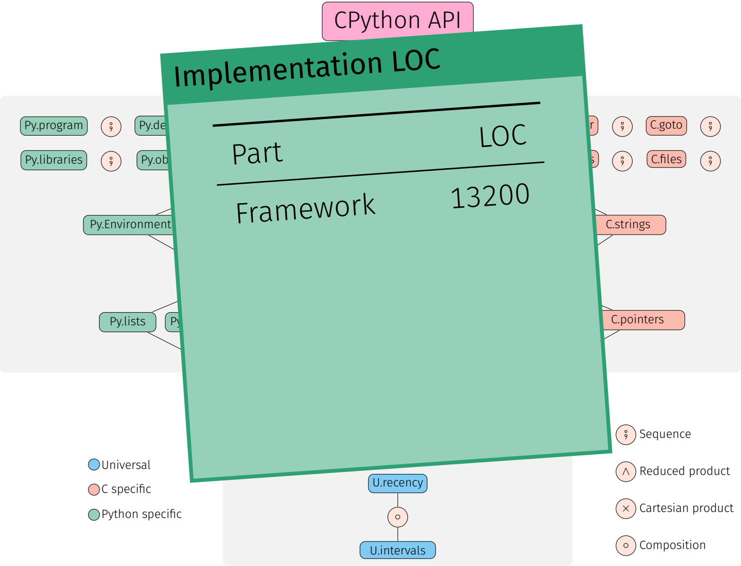
… to a multilanguage analysis!



Implementation LOC

Part	LOC

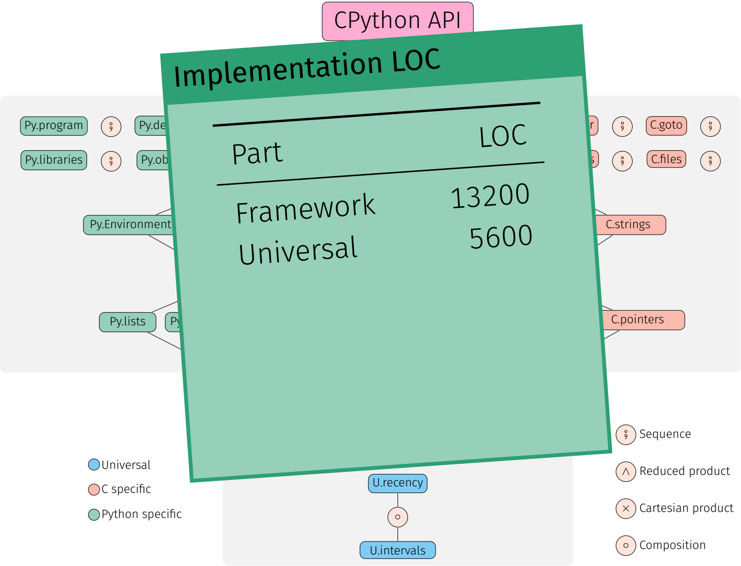
... to a multilanguage analysis!



Implementation LOC

Part	LOC
Framework	13200

… to a multilanguage analysis!



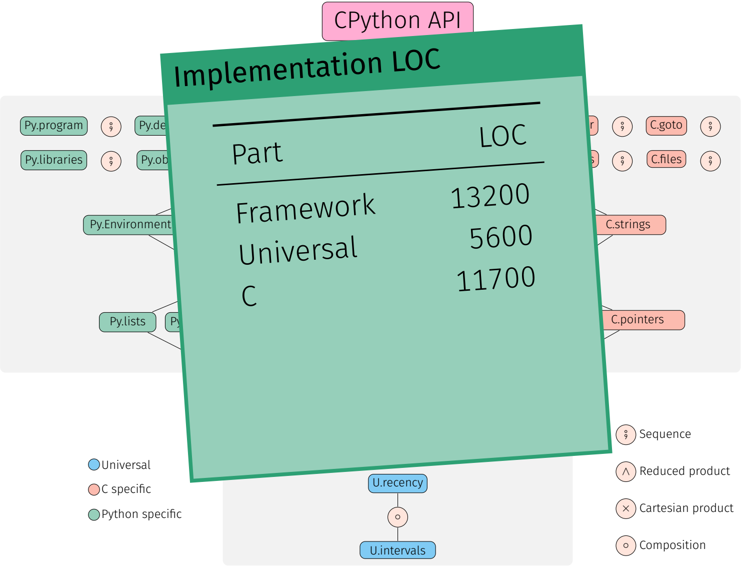
Implementation LOC

Part	LOC
Framework	13200
Universal	5600

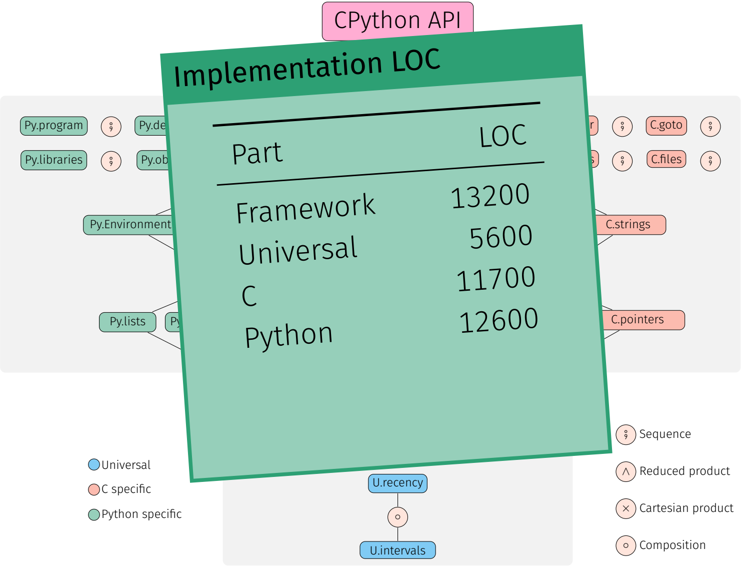
... to a multilanguage analysis!

Implementation LOC

Part	LOC
Framework	13200
Universal	5600
C	11700



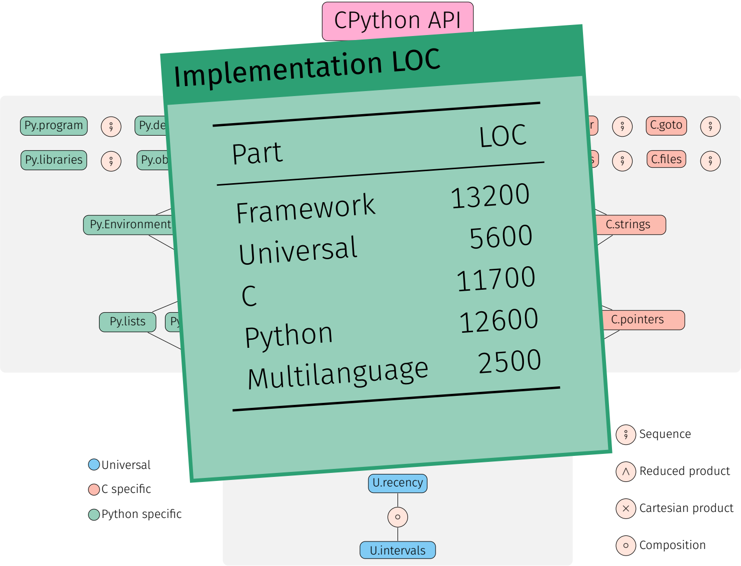
... to a multilanguage analysis!



Implementation LOC

Part	LOC
Framework	13200
Universal	5600
C	11700
Python	12600

… to a multilanguage analysis!



Implementation LOC

Part	LOC
Framework	13200
Universal	5600
C	11700
Python	12600
Multilanguage	2500

Benchmarks

Corpus selection

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

Library	C + Py. Loc	Tests	⏱/test	$\frac{\text{\# proved checks}}{\text{\# checks}}$ %	# checks
`noise`	1397	15/15	1.2s	99.7%	6690
`cdistance`	2345	28/28	4.1s	98.0%	13716
`llist`	4515	167/194	1.5s	98.8%	36255
`ahocorasick`	4877	46/92	1.2s	96.7%	6722
`levenshtein`	5798	17/17	5.3s	84.6%	4825
`bitarray`	5841	159/216	1.6s	94.9%	25566

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

Future work

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

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

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

$\rightsquigarrow$ 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

European Research Council
Established by the European Commission

SORBONNE UNIVERSITÉ