A Multilanguage Static Analysis of Python/C Programs with Mopsa

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

Facebook TAV
1 December 2021

rmonat.fr

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'])
```

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 cover all possible executions.

Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub[1]

[1] https://octoverse.github.com/#top-languages

Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub[1]

New features

- Object orientation,

[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,

[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,

[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[1]

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

  $\Longrightarrow$ 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 et al. “A Monotonic Superclass Linearization for Dylan”. OOPSLA 1996

Python's specificities (II)

Dual type system

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

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 et al. "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 et al. "A Monotonic Superclass Linearization for Dylan". OOPSLA 1996

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 (arbitrary-precision integers in Python, bounded in C)

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

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

```
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)
```

Searching for a loop invariant (l. 4)

Environment abstraction

$m \mapsto @^{\sharp}_{\mathtt{int}^{\sharp}} \quad i \mapsto @^{\sharp}_{\mathtt{int}^{\sharp}}$

Proved safe?

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

A program analysis workflow

Avering 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

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

A program analysis workflow

Avering 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

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

Numeric abstraction (intervals)

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

A program analysis workflow

Avering 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

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

Numeric abstraction (intervals)

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

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

A program analysis workflow

Avering 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

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

Numeric abstraction (polyhedra)

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

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

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

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

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

Numeric abstraction (polyhedra)

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

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

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

Attributes abstraction

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

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(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

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

Conclusion

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

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

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

Attributes abstraction

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

Overview of Mopsa

Modular **O**pen **P**latform for **S**tatic **A**nalysis[2]

`gitlab.com/mopsa/mopsa-analyzer`

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

Modular Open Platform for Static Analysis[2]

`gitlab.com/mopsa/mopsa-analyzer`

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

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

Overview of Mopsa

Modular Open Platform for Static Analysis[2]

`gitlab.com/mopsa/mopsa-analyzer`

- One AST to analyze them all
  - Multilanguage support
  - Expressiveness
  - Reusability
- Unified domain signature
  - Semantic rewriting
  - Loose coupling
  - Observability

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

Overview of Mopsa

Modular Open Platform for Static Analysis[2]

`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

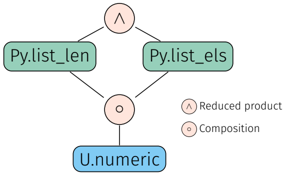
[2] Journault, Miné, Monat, and Ouadjaout. "Combinations of reusable abstract domains for a multilingual static analyzer". VSTTE 2019.

Overview of Mopsa

Modular Open Platform for Static Analysis[2]

`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



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

Dynamic, semantic iterators with delegation

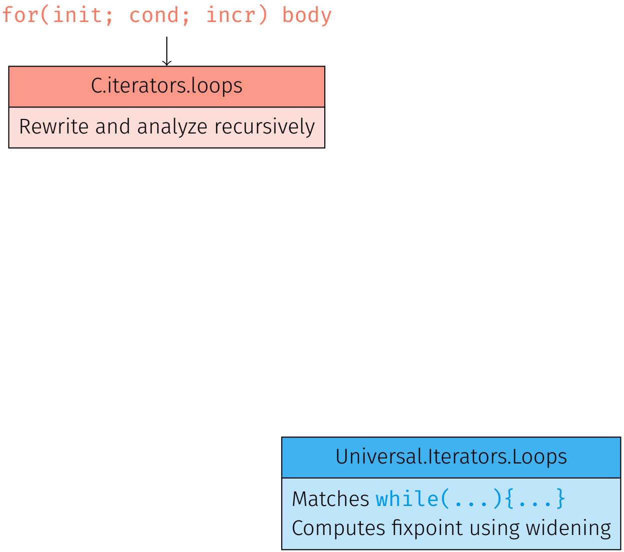
Universal.Iterators.Loops
Matches `while(...){...}` Computes fixpoint using widening

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

```
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

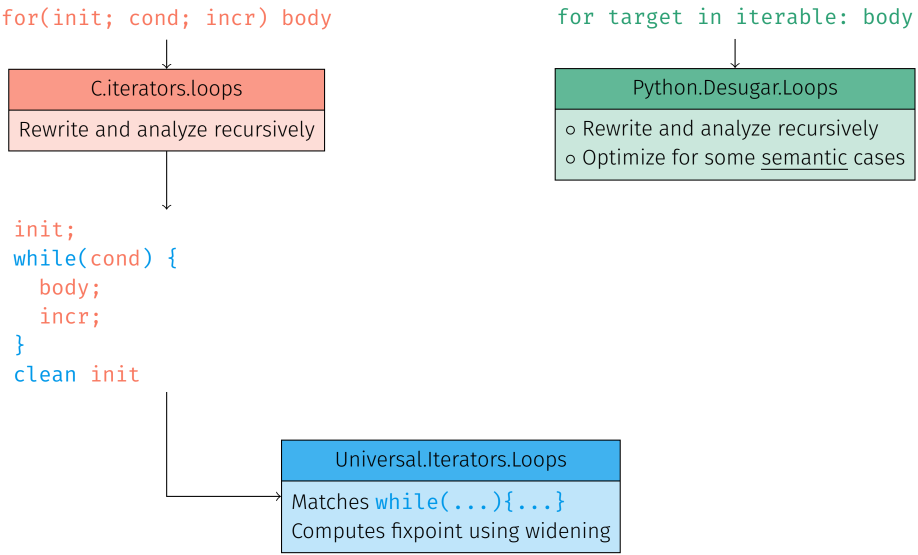
```
init;
while(cond) {
  body;
  incr;
}
clean init
```

Universal.Iterators.Loops

Matches `while(...){...}`

Computes fixpoint using widening

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
```

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

Python.Desugar.Loops

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

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
from random import randrange

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

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()
```

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

Combining C and Python – example

counter.c

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

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

    self->count += i;
    Py_RETURN_NONE;
}

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

count.py

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

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

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

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()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$
- $\texttt{power} = 31 \Rightarrow \texttt{r} = -2^{31}$
- $32 \leq \texttt{power} \leq 64$: 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

```
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

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

- No raised exceptions $\implies$ 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

```python
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

Analysis result

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()
```

Analysis result

counter.c

```c
1  typedef struct {
2      PyObject_HEAD;
3      int count;
4  } Counter;
5
6  static PyObject*
7  CounterIncr(Counter *self,
8  {
9      int i = 1;
10     if(!PyArg_ParseTuple(a
11        return NULL;
12
13     self->count += i;
14     Py_RETURN_NONE;
15 }
16
17 static PyObject*
18 CounterGet(Counter *self)
19 {
20     return Py_BuildValue("
21 }
```

count.py

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

```
⚠Check #430:
./counter.c: In function 'CounterIncr':
./counter.c:13.2-18: warning: Integer overflow

 13:    self->count += i;
        ^^^^^^^^^^^^^^^^
 '(self->count + i)' has value [0,2147483648] that is larger
   than the range of 'signed int' = [-2147483648,2147483647]
 Callstack:
       from count.py:8.0-8: CounterIncr


✗ Check #506:
count.py: In function 'PyErr_SetString':
count.py:6.0-14: error: OverflowError exception

 6: c.incr(2**p-1)
    ^^^^^^^^^^^^^^
 Uncaught Python exception: OverflowError: signed integer is greater than maximum
 Uncaught Python exception: OverflowError: Python int too large to convert to C long
 Callstack:
       from ./counter.c:17.6-38::convert_single[0]: PyParseTuple_int
       from count.py:7.0-14: CounterIncr
 +1 other callstack
```

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

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

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

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

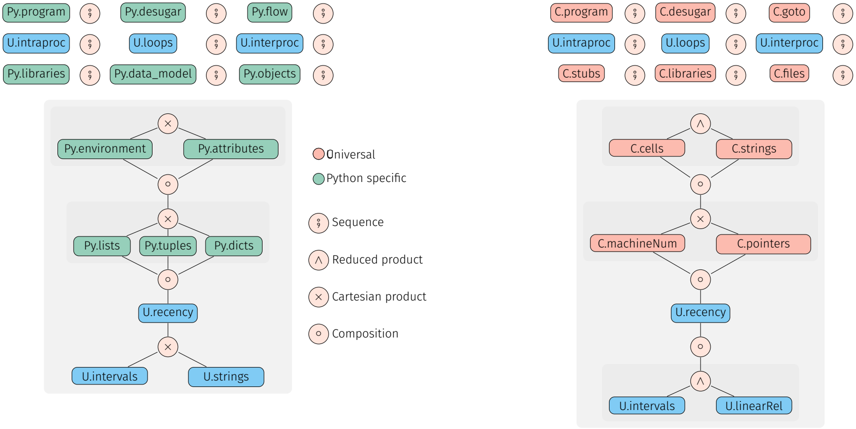
- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

Limitations

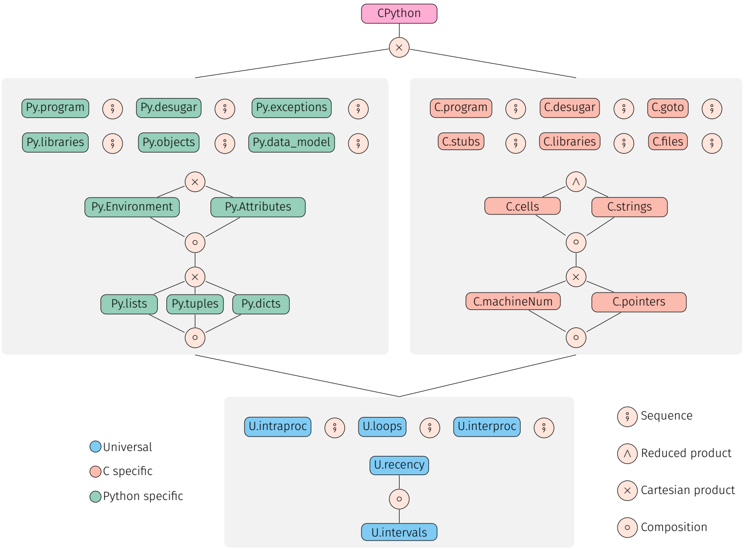
- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
- Manual modelization from CPython's source code

Implementation & Experimental Evaluation

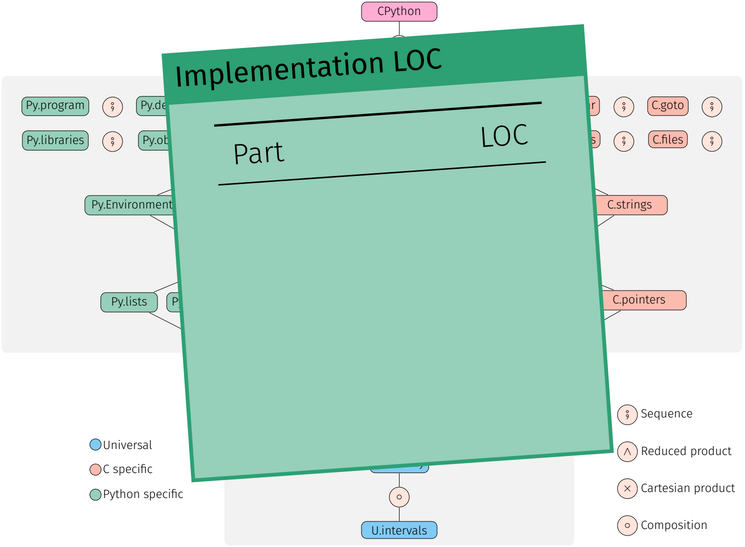
From distinct Python and C analyses...



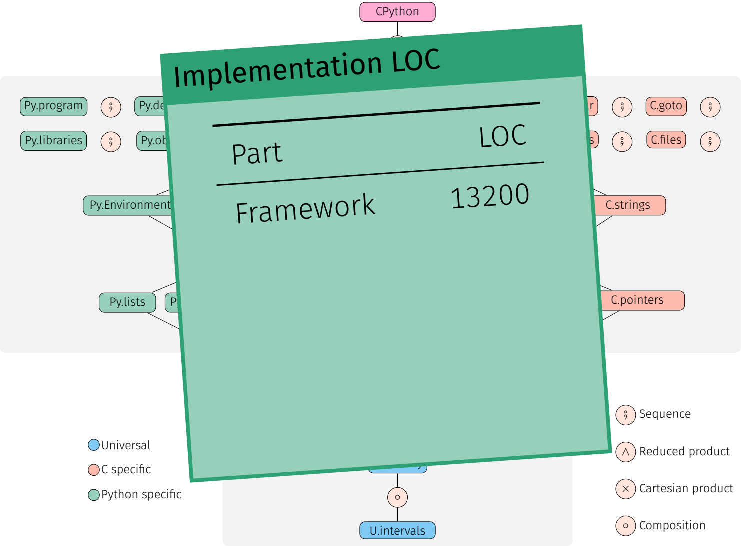
... to a multilanguage analysis!



... to a multilanguage analysis!



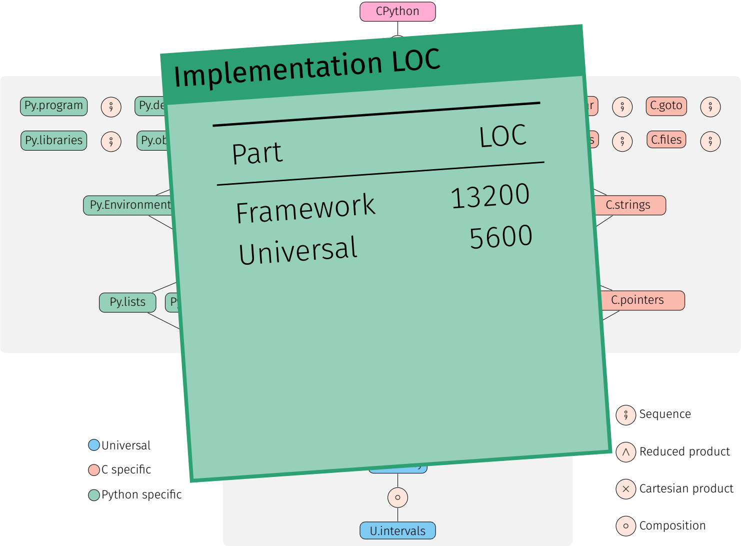
… to a multilanguage analysis!



Implementation LOC

Part	LOC
Framework	13200

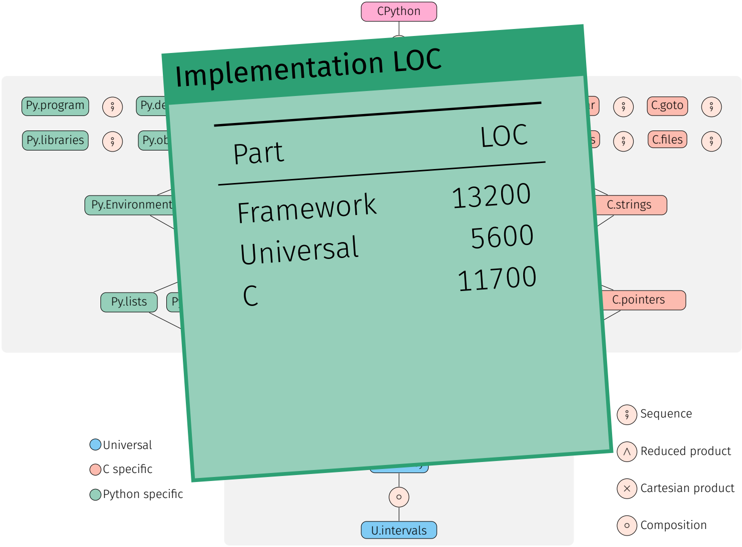
… to a multilanguage analysis!



Implementation LOC

Part	LOC
Framework	13200
Universal	5600

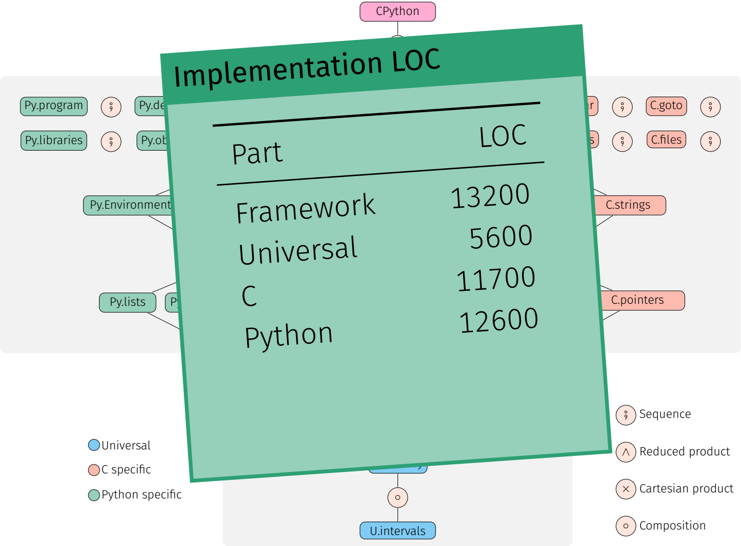
... to a multilanguage analysis!



Implementation LOC

Part	LOC
Framework	13200
Universal	5600
C	11700

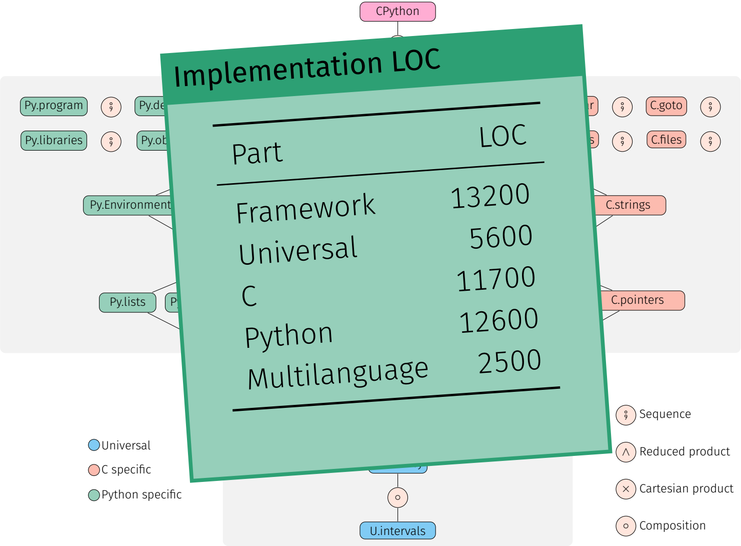
… to a multilanguage analysis!



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

$\frac{\text{safe C checks}}{\text{total C checks}}\%$

total C checks

average # transitions between Python and C per test

Conclusion

Contribution: multilanguage Python/C analysis

Difficulties

- Concrete semantics
- Memory interaction

Monat, Ouadjaout, and Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". SAS 2021

Contribution: multilanguage Python/C analysis

Difficulties

- Concrete semantics
- Memory interaction

Previous works

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

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

Contribution: multilanguage Python/C analysis

Difficulties

- Concrete semantics
- Memory interaction

Previous works

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

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

rmonat.fr