A Multilanguage Static Analysis of Python Programs with Native C Extensions

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

Static Analysis Symposium
18 October 2021

rmonat.fr/sas21

Introduction

Static Program Analysis

sum.py

```python
def sum(l):
  s = 0
  for x in l:
    s += x
  return s

r1 = sum([1, 2, 3])
r2 = sum(['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.

Automatic no expert knowledge required.

Semantic based on a formal modelization of the language.

Sound cover all possible executions.

Python

- #2 language on Github[1],

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

Python

- #2 language on Github[1],
- Object oriented,

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

Python

- #2 language on Github[1],
- Object oriented,
- Dynamic typing,

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

Python

- #2 language on Github[1],
- Object oriented,
- Dynamic typing,
- Operator redefinition,

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

Python

- #2 language on Github[1],
- Object oriented,
- Dynamic typing,
- Operator redefinition,
- Introspection operators,

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

Python

- #2 language on Github[1],
- Object oriented,
- Dynamic typing,
- Operator redefinition,
- Introspection operators,
- Dynamic object structure,

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

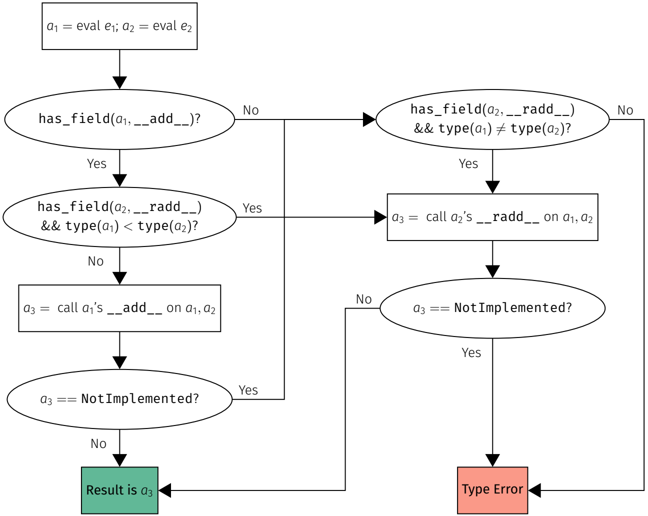
Python

- #2 language on Github[1],
- Object oriented,
- Dynamic typing,
- Operator redefinition,
- Introspection operators,
- Dynamic object structure,
- `eval`.

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

Python

- #2 language on Github[1],
- Object oriented,
- Dynamic typing,
- Operator redefinition,
- Introspection operators,
- Dynamic object structure,
- `eval`.



$a_1 =$ eval e_1; $a_2 =$ eval e_2

has_field(a_1, __add__)?

No

has_field(a_2, __radd__) && type(a_1) $\neq$ type(a_2)?

No

Yes

Yes

has_field(a_2, __radd__) && type(a_1) $<$ type(a_2)?

Yes

$a_3 =$ call a_2's __radd__ on a_1, a_2

No

$a_3 =$ call a_1's __add__ on a_1, a_2

No

$a_3 ==$ NotImplemented?

Yes

$a_3 ==$ NotImplemented?

Yes

No

Result is a_3

Type Error

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

Combining C and Python

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

Combining C and Python

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

- To bring better performance (numpy),

Combining C and Python

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

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

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

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 object representations (Python objects, C structs),

Combining C and Python

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 object representations (Python objects, C structs),
- Different runtime-errors (exceptions in Python),

Combining C and Python

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 object representations (Python objects, C structs),
- Different runtime-errors (exceptions in Python),
- Garbage collection.

Outline

1. Introduction
2. A Concrete Example
3. Concrete Multilanguage Semantics
4. Mopsa, a Multilanguage Analyzer
5. Experimental Evaluation
6. Conclusion

A Concrete Example

Combining C and Python – Counter 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 – Counter 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 – Counter 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 – Counter 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 $\implies$ 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 $\implies$ missed errors,
- Only types,

How to analyze multilanguage programs?

Type annotations

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

- No raised exceptions ⟹ missed errors,
- Only types,
- Typeshed: type annotations for the standard library

How to analyze multilanguage programs?

Type annotations

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

- No raised exceptions ⟹ missed errors,
- Only types,
- Typeshed: type annotations for the standard library,
  used in previous work: Monat et al. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP 2020.

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

```python
class Counter:
    def __init__(self):
        self.count = 0
    def get(self):
        return self.count
    def incr(self, i=1):
        self.count += i
```

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

```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?

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.

Our approach

counter.c

```c
 1  typedef struct {
 2      PyObject_HEAD;
 3      int count;
 4  } Counter;
 5
 6  static PyObject*
 7  CounterIncr(Counter *self, PyObject *args)
 8  {
 9      int i = 1;
10      if(!PyArg_ParseTuple(args, "|i", &i))
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("i", self->count);
21  }
```

count.py

```python
1  from counter import Counter
2  from random import randrange
3
4  c = Counter()
5  power = randrange(128)
6  c.incr(2**power-1)
7  c.incr()
8  r = c.get()
```

```
⚠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

Multilanguage Semantics

- Builds upon the Python and C semantics.

Multilanguage Semantics

Multilanguage Semantics

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

Multilanguage Semantics

Multilanguage Semantics

- Builds upon the Python and C semantics.
- Defines the API: calls between languages, value conversions.
- Shared heap, with disjoint, complementary views.

Multilanguage Semantics

Multilanguage Semantics

- 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.

Multilanguage Semantics

Multilanguage Semantics

- 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.

Limitations

Multilanguage Semantics

Multilanguage Semantics

- 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.

Limitations

- Garbage collection not handled.

Multilanguage Semantics

Multilanguage Semantics

- 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.

Limitations

- Garbage collection not handled.
- C access to Python objects only through the API.

Multilanguage Semantics

Multilanguage Semantics

- 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.

Limitations

- Garbage collection not handled.
- C access to Python objects only through the API.
- Manual modelization from Python’s source code.

Multilanguage Semantics

Multilanguage Semantics

- 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.

Limitations

- Garbage collection not handled.
- C access to Python objects only through the API.
- Manual modelization from Python’s source code.

$\Longrightarrow$ details in the paper.

Mopsa, a Multilanguage Analyzer

Our Framework: MOPSA

Modular **O**pen **P**latform for **S**tatic **A**nalysis

- Multi-language support (C and Python)

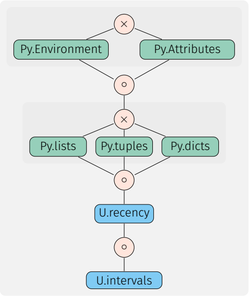
Expressiveness	Keep the original AST of the program.
Reusability	Reuse abstractions among languages.

- Flexible architecture

Loose coupling	Divide into interchangeable components.
Composition	Create complex components from simpler ones.
Cooperation	Components can communicate and delegate tasks.
Observability	Pluggable hooks observe the analysis.

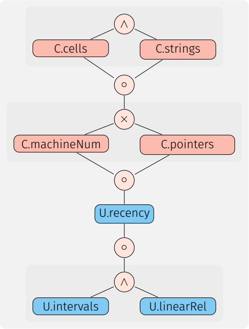
From distinct Python and C analyses...

Py.program ⨟ Py.desugar ⨟ Py.exceptions ⨟
U.intraproc ⨟ U.loops ⨟ U.interproc ⨟
Py.libraries ⨟ Py.objects ⨟ Py.data_model ⨟



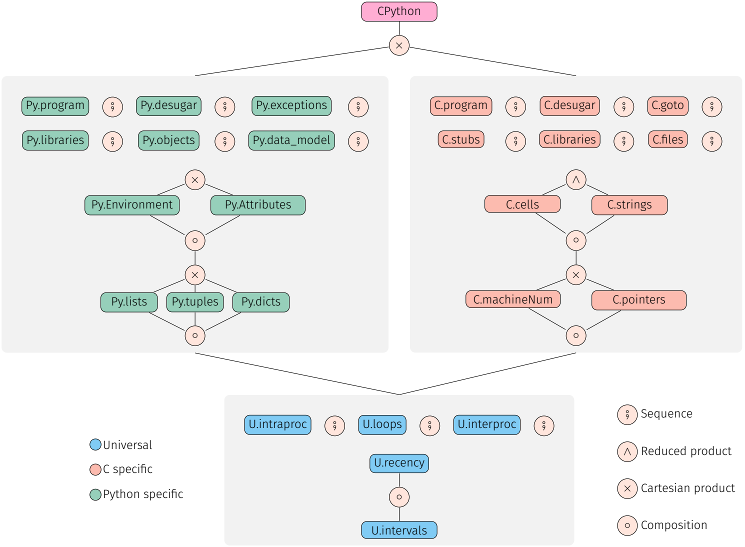
×
Py.Environment Py.Attributes
∘
×
Py.lists Py.tuples Py.dicts
∘
U.recency
∘
U.intervals

C.program ⨟ C.desugar ⨟ C.goto ⨟
U.intraproc ⨟ U.loops ⨟ U.interproc ⨟
C.stubs ⨟ C.libraries ⨟ C.files ⨟



∧
C.cells C.strings
∘
×
C.machineNum C.pointers
∘
U.recency
∘
∧
U.intervals U.linearRel

… to a multilanguage analysis!



🔵 Universal

🔴 C specific

🟢 Python specific

⨾ Sequence

∧ Reduced product

× Cartesian product

∘ Composition

Analysis of the 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->counter += i;
  Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
```

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()
```

$\mathbb{E}^{\#}_{\text{Py.call}}[\![\, \texttt{Counter()} \,]\!]\sigma^{\sharp}$

C

```
Pointers
 ⟨CounterCls,8,ptr⟩ : {PyType_Type}
 ⟨CounterCls,232,ptr⟩ : {Counter_methods}
```

Universal

```
Heap (Recency)
 @CounterCls @CounterIncr
 @CounterGet

Intervals
```

Python

```
Attributes
 @CounterCls ↦ {get, incr}


Environment
 Counter ↦ {@CounterCls}
 @CounterCls·get ↦
 {@c function CounterGet}
 @CounterCls·incr ↦
 {@c function CounterIncr}
```

Analysis of the 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->counter += i;
  Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
```

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()
```

$\mathbb{E}^{\#}_{\text{C.call}}[\![\, \texttt{tp\_new\_wrapper}(\texttt{type}, \texttt{tuple(Counter)}, \texttt{NULL}) \,]\!]\sigma^{\sharp}$

C

```
Pointers
 ⟨CounterCls,8,ptr⟩ : {PyType_Type}
 ⟨CounterCls,232,ptr⟩ : {Counter_methods}
```

Universal

```
Heap (Recency)
 @CounterCls @CounterIncr
 @CounterGet

Intervals
```

Python

```
Attributes
 @CounterCls ↦ {get, incr}

Environment
 Counter ↦ {@CounterCls}
 @CounterCls·get ↦
 {@c function CounterGet}
 @CounterCls·incr ↦
 {@c function CounterIncr}
```

Analysis of the 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->counter += i;
  Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
```

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()
```

$\mathbb{E}^{\#}_{\text{C.call}}[\![\, \texttt{PyType\_GenericNew(CounterCls}, \texttt{NULL}, \texttt{NULL)} \,]\!]\sigma^{\sharp}$

C

```
Pointers
 ⟨CounterCls,8,ptr⟩ : {PyType_Type}
 ⟨CounterCls,232,ptr⟩ : {Counter_methods}
```

Universal

```
Heap (Recency)
 @CounterCls @CounterIncr
 @CounterGet @I{CounterCls}

Intervals
```

Python

```
Attributes
 @CounterCls ↦ {get, incr}


Environment
 Counter ↦ {@CounterCls}
 @CounterCls·get ↦
 {@c function CounterGet}
 @CounterCls·incr ↦
 {@c function CounterIncr}
```

Analysis of the 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->counter += i;
  Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
```

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()
```

$\mathbb{E}^{\#}_{\text{C.cells}}[\![\ \texttt{@I\{CounterCls\}->ob\_type} = \texttt{CounterCls}\ ]\!]\sigma^{\sharp}$

Python

```
Attributes
 @CounterCls ↦ {get, incr}


Environment
 Counter ↦ {@CounterCls}
 @CounterCls·get ↦
 {@c function CounterGet}
 @CounterCls·incr ↦
 {@c function CounterIncr}
```

Universal

```
Heap (Recency)
 @CounterCls @CounterIncr
 @CounterGet @I{CounterCls}

Intervals
```

C

```
Pointers
 ⟨CounterCls,8,ptr⟩ : {PyType_Type}
 ⟨CounterCls,232,ptr⟩ : {Counter_methods}
 ⟨@I{CounterCls},8,ptr⟩ : {CounterCls}
```

Analysis of the 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->counter += i;
  Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
```

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()
```

$\mathbb{E}^{\#}_{\text{C.cells}}[\![\ \texttt{@I\{CounterCls\}->count} = 0\ ]\!]\sigma^{\sharp}$

C

```
Pointers
 ⟨CounterCls,8,ptr⟩ : {PyType_Type}
 ⟨CounterCls,232,ptr⟩ : {Counter_methods}
 ⟨@I{CounterCls},8,ptr⟩ : {CounterCls}
```

Universal

```
Heap (Recency)
 @CounterCls @CounterIncr
 @CounterGet @I{CounterCls}

Intervals
 ⟨@I{CounterCls},16,s32⟩ ↦ [0, 0]
```

Python

```
Attributes
 @CounterCls ↦ {get, incr}


Environment
 Counter ↦ {@CounterCls}
 @CounterCls·get ↦
 {@c function CounterGet}
 @CounterCls·incr ↦
 {@c function CounterIncr}
```

Analysis of the 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->counter += i;
  Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
```

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()
```

${}^{c}\hookrightarrow_{p}(\text{@I\{CounterCls\}}, \sigma^{\sharp})$

Python

Attributes
 @CounterCls ↦ {get, incr}
 @I{CounterCls} ↦ ∅

Environment
 Counter ↦ {@CounterCls}
 @CounterCls·get ↦
 {@c function CounterGet}
 @CounterCls·incr ↦
 {@c function CounterIncr}

Universal

Heap (Recency)
 @CounterCls @CounterIncr
 @CounterGet @I{CounterCls}

Intervals
 ⟨@I{CounterCls},16,s32}⟩ ↦ [0, 0]

C

Pointers
 ⟨CounterCls,8,ptr⟩ : {PyType_Type}
 ⟨CounterCls,232,ptr⟩ : {Counter_methods}
 ⟨@I{CounterCls},8,ptr⟩ : {CounterCls}

Analysis of the 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->counter += i;
  Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
```

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()
```

$\mathbb{S}^{\#}_{\text{Py.env}}[\![c = @\text{I}\{\text{CounterCls}\}]\!]\sigma^{\sharp}$

C

```
Pointers
 ⟨CounterCls,8,ptr⟩ : {PyType_Type}
 ⟨CounterCls,232,ptr⟩ : {Counter_methods}
 ⟨@I{CounterCls},8,ptr⟩ : {CounterCls}
```

Universal

```
Heap (Recency)
 @CounterCls @CounterIncr
 @CounterGet @I{CounterCls}

Intervals
 ⟨@I{CounterCls},16,s32⟩ ↦ [0, 0]
```

Python

```
Attributes
 @CounterCls ↦ {get, incr}
 @I{CounterCls} ↦ ∅

Environment
 Counter ↦ {@CounterCls}
 @CounterCls·get ↦
 {@c function CounterGet}
 @CounterCls·incr ↦
 {@c function CounterIncr}
 c ↦ {@I{CounterCls}}
```

Analysis of the 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->counter += i;
  Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
```

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()
```

$\mathbb{S}^{\#}_{\text{Py}}[\![\ \text{power} = \text{randrange}(128)\ ]\!]\sigma^{\sharp}$

C

```
Pointers
 ⟨CounterCls,8,ptr⟩ : {PyType_Type}
 ⟨CounterCls,232,ptr⟩ : {Counter_methods}
 ⟨@I{CounterCls},8,ptr⟩ : {CounterCls}
```

Universal

```
Heap (Recency)
 @CounterCls @CounterIncr
 @CounterGet @I{CounterCls}

Intervals
 ⟨@I{CounterCls},16,s32⟩ ↦ [0, 0]
 power ↦ [0, 127]
```

Python

```
Attributes
 @CounterCls ↦ {get, incr}
 @I{CounterCls} ↦ ∅

Environment
 Counter ↦ {@CounterCls}
 @CounterCls·get ↦
 {@c function CounterGet}
 @CounterCls·incr ↦
 {@c function CounterIncr}
 c ↦ {@I{CounterCls}}
 power ↦ {@I{int}}
```

Experimental Evaluation

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

$\underbrace{\quad}_{\frac{\text{safe C checks}}{\text{total C checks}}\%}$

$\underbrace{\quad}_{\text{total C checks}}$

$\underbrace{\quad}_{\text{average \# transitions between Python and C per test}}$

Conclusion

Conclusion

Previous works (JNI)

Static translation of some of C’s effects, injected back into the Java analysis.

Conclusion

Previous works (JNI)

Static translation of some of C’s effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

Conclusion

Previous works (JNI)

Static translation of some of C's effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

- Combines previous C and Python analyses,

Conclusion

Previous works (JNI)

Static translation of some of C's effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

- Combines previous C and Python analyses,
- Allocated objects are shared in the memory,

Conclusion

Previous works (JNI)

Static translation of some of C’s effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

- Combines previous C and Python analyses,
- Allocated objects are shared in the memory,
- Each language has different abstractions,

Conclusion

Previous works (JNI)

Static translation of some of C's effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

- Combines previous C and Python analyses,
- Allocated objects are shared in the memory,
- Each language has different abstractions,
- These abstractions co-exist and collaborate.

Conclusion

Previous works (JNI)

Static translation of some of C’s effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

- Combines previous C and Python analyses,
- Allocated objects are shared in the memory,
- Each language has different abstractions,
- These abstractions co-exist and collaborate.

Future work

Conclusion

Previous works (JNI)

Static translation of some of C's effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

- Combines previous C and Python analyses,
- Allocated objects are shared in the memory,
- Each language has different abstractions,
- These abstractions co-exist and collaborate.

Future work

- Analyze larger applications,

Conclusion

Previous works (JNI)

Static translation of some of C's effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

- Combines previous C and Python analyses,
- Allocated objects are shared in the memory,
- Each language has different abstractions,
- These abstractions co-exist and collaborate.

Future work

- Analyze larger applications,
- Validate typeshed's annotations,

Conclusion

Previous works (JNI)

Static translation of some of C's effects, injected back into the Java analysis.

Our approach: dynamic switch between the analyses of Python and C

- Combines previous C and Python analyses,
- Allocated objects are shared in the memory,
- Each language has different abstractions,
- These abstractions co-exist and collaborate.

Future work

- Analyze larger applications,
- Validate typeshed's annotations,
- Apply to other multilanguage settings (JNI).