A Multilanguage Static Analysis of Python Programs with Native C Extensions

Antoine Miné, Abdelraouf Ouadjaout, Raphaël Monat

5th July 2021

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
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

Infer run-time errors (or other semantic properties).

Automatic no expert knowledge required.

Semantic based on a formal modelization of the language.

Sound cover all possible executions.

Static Analysis by Abstract Interpretation

How does an abstract interpreter work?

- Execution in approximate, computable domains,
- Program ⇝ Abstract state ⇝ Semantic property (alarms),
- Combine abstract domains to gain precision.

sum_indexed.py

```python
def sum(l):
  s = 0
  for i in range(len(l)):
    s += l[i]
  return s

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

Static Analysis by Abstract Interpretation

How does an abstract interpreter work?

- Execution in approximate, computable domains,
- Program ⇝ Abstract state ⇝ Semantic property (alarms),
- Combine abstract domains to gain precision.

sum_indexed.py

```python
def sum(l):
  s = 0
  for i in range(len(l)):
    s += l[i]
  return s

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

- Call with `[1, 2, 3]`

Static Analysis by Abstract Interpretation

How does an abstract interpreter work?

- Execution in approximate, computable domains,
- Program ⇝ Abstract state ⇝ Semantic property (alarms),
- Combine abstract domains to gain precision.

sum_indexed.py

```python
def sum(l):
  s = 0
  for i in range(len(l)):
    s += l[i]
  return s

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

- Call with `[1, 2, 3]`

$$\left.\begin{array}{l} \texttt{l} : \texttt{List[int]} \\ \texttt{i} : \texttt{int} \\ \texttt{s} : \texttt{int} \end{array}\right\} \text{types } \checkmark$$

Static Analysis by Abstract Interpretation

How does an abstract interpreter work?

- ▶ Execution in approximate, computable domains,
- ▶ Program ⇝ Abstract state ⇝ Semantic property (alarms),
- ▶ Combine abstract domains to gain precision.

sum_indexed.py

```python
def sum(l):
  s = 0
  for i in range(len(l)):
    s += l[i]
  return s

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

- ▶ Call with `[1, 2, 3]`

$$\left.\begin{array}{l} \texttt{l} : \texttt{List[int]} \\ \texttt{i} : \texttt{int} \\ \texttt{s} : \texttt{int} \end{array}\right\} \text{types } \checkmark$$

- ▶ Call with `[1, 'b', 3]`

$$\left.\begin{array}{l} \texttt{l} : \texttt{List[Union[int, str]]} \\ \texttt{s} : \texttt{int} \end{array}\right\} \texttt{int} + \texttt{str} \text{ invalid}$$

Static Analysis by Abstract Interpretation

How does an abstract interpreter work?

- Execution in approximate, computable domains,
- Program ⇝ Abstract state ⇝ Semantic property (alarms),
- Combine abstract domains to gain precision.

sum_indexed.py

```python
def sum(l):
  s = 0
  for i in range(len(l)):
    s += l[i]
  return s

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

- Call with `[1, 2, 3]`

$$\left.\begin{array}{l} \texttt{l} : \texttt{List[int]} \\ \texttt{i} : \texttt{int} \\ \texttt{s} : \texttt{int} \end{array}\right\} \text{types } \checkmark \quad \left.\begin{array}{l} \texttt{len(l)} = 3 \\ 0 \leq \texttt{i} < 3 \end{array}\right\} \text{valid list accesses}$$

$$\texttt{s} \geq 0$$

- Call with `[1, 'b', 3]`

$$\left.\begin{array}{l} \texttt{l} : \texttt{List[Union[int, str]]} \\ \texttt{s} : \texttt{int} \end{array}\right\} \texttt{int} + \texttt{str} \text{ invalid}$$

Python

- #2 language on Github,

Python

- #2 language on Github,
- Object oriented,

Python

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,

Python

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,

Python

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,
- Introspection operators,

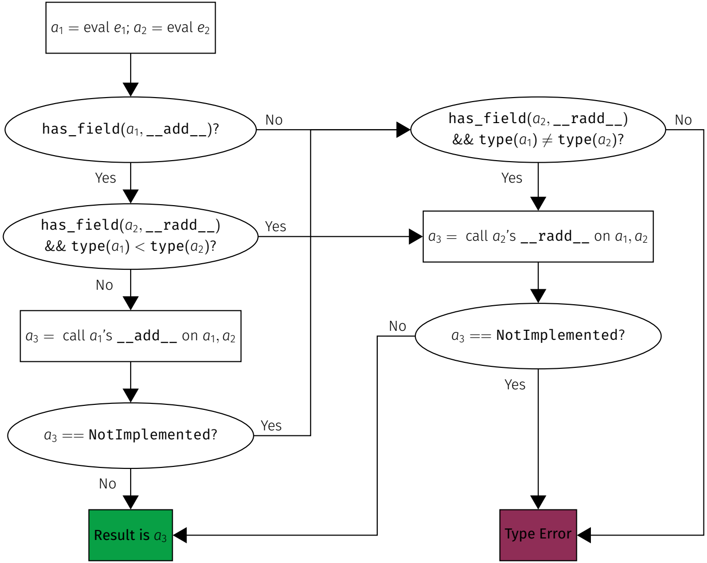
Python

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,
- Introspection operators,
- Dynamic object structure (attribute addition),

Python

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,
- Introspection operators,
- Dynamic object structure (attribute addition),
- `eval`.

Python is Complex! Semantics of $e_1 + e_2$



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

`has_field`(a_1, `__add__`)?

Yes

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

No

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

$a_3 ==$ `NotImplemented`?

No

Result is a_3

No

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

Yes

Yes

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

Yes

$a_3 ==$ `NotImplemented`?

No

Yes

Type Error

No

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. Interlude: the Mopsa Static Analyzer
4. Multilanguage Analysis
5. Conclusion

A Concrete Example

Combining C and Python – Counter Example

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int counter;
} 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)
{
    return Py_BuildValue("i", self->counter);
}
```

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 counter;
} 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)
{
    return Py_BuildValue("i", self->counter);
}
```

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()
```

⇒ Demo!

Combining C and Python – Counter Example

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int counter;
} 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)
{
    return Py_BuildValue("i", self->counter);
}
```

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()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$
- $\texttt{power} = 31 \Rightarrow \texttt{r} = -2^{31}$
- $32 \leq \texttt{power} \leq 62$: OverflowError: signed integer is greater than maximum
- $\texttt{power} \geq 63$: 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

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

- No raised exceptions $\implies$ missed errors,

How to analyze multilanguage programs?

Type annotations

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

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

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

```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,

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

- 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

- 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.

Interlude: the Mopsa Static Analyzer

Project’s target: experiment new development ideas

Modular Open Platform for Static Analysis

- Multi-language support (C and Python)

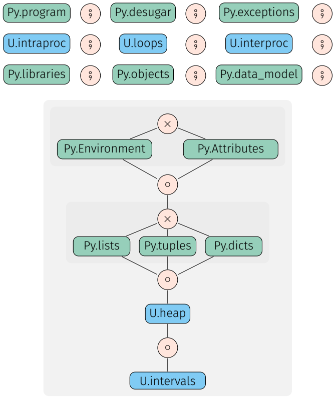
Expressiveness	Keep the original AST of the program.
Reusability	Reuse abstractions among languages.

- Flexible architecture

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

Multilanguage Analysis

Towards a Multilanguage Configuration



Py.program ⨟ Py.desugar ⨟ Py.exceptions ⨟

U.intraproc ⨟ U.loops ⨟ U.interproc ⨟

Py.libraries ⨟ Py.objects ⨟ Py.data_model ⨟

×

Py.Environment Py.Attributes

∘

×

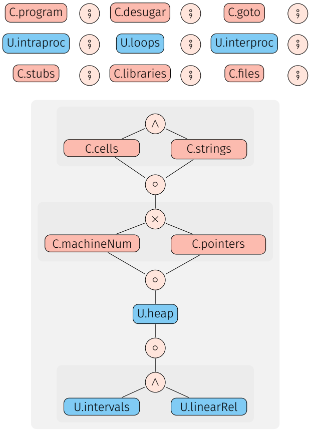
Py.lists Py.tuples Py.dicts

∘

U.heap

∘

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

∘

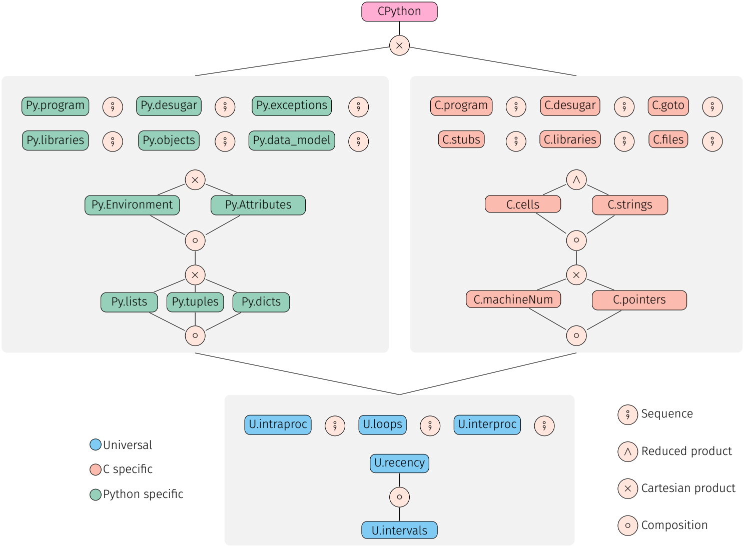
U.heap

∘

∧

U.intervals U.linearRel

Towards a Multilanguage Configuration



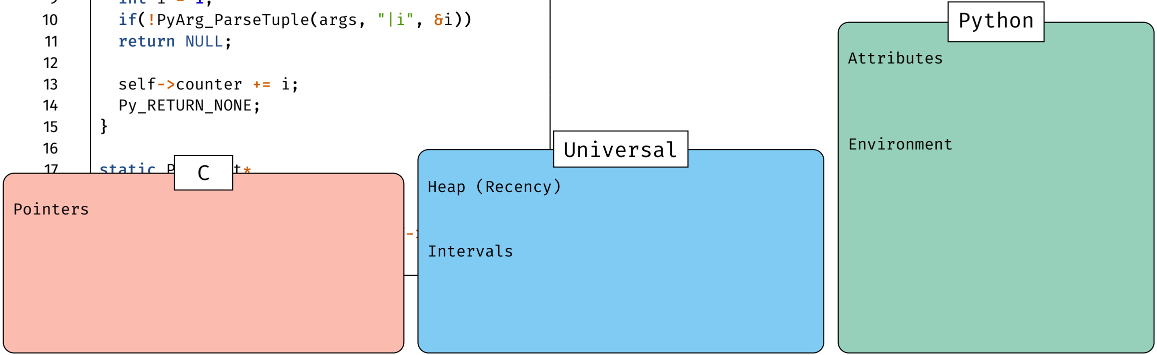
Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P      t*
```

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

Pointers

Universal

Heap (Recency)

Intervals

Python

Attributes

Environment

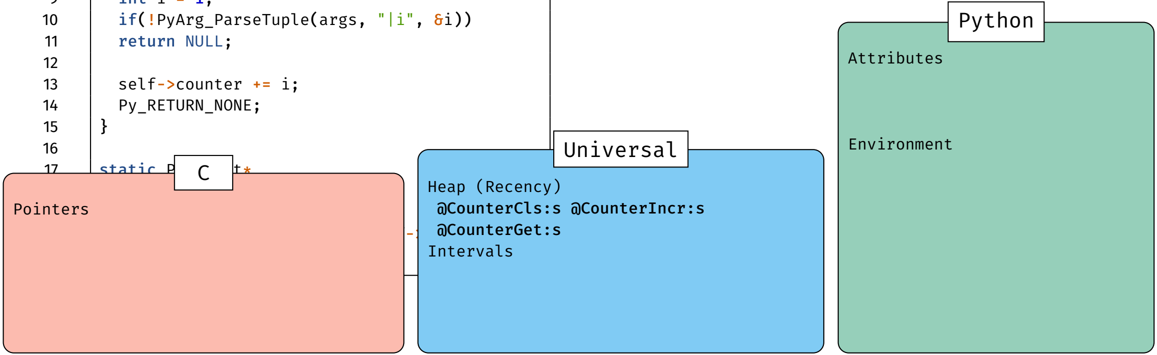
Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P      t*
```

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

Pointers

Universal

Heap (Recency)
@CounterCls:s @CounterIncr:s
@CounterGet:s
Intervals

Python

Attributes

Environment

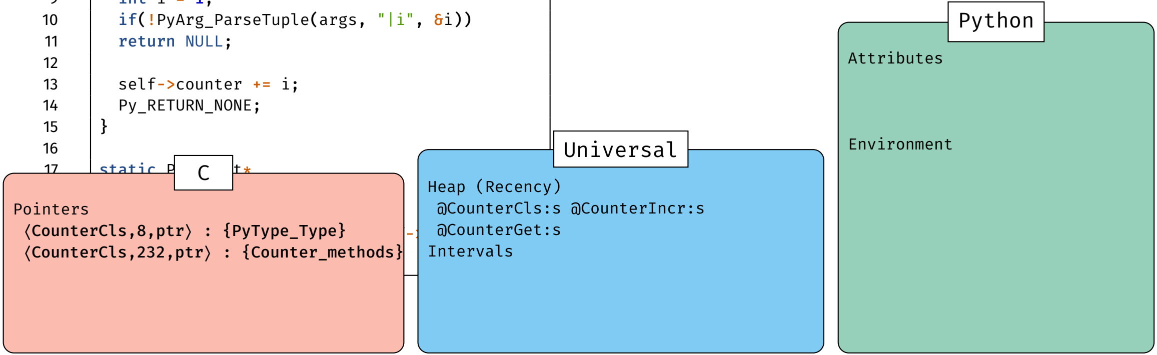
Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P    t*
```

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

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

Universal

Heap (Recency)
 @CounterCls:s @CounterIncr:s
 @CounterGet:s
Intervals

Python

Attributes

Environment

Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P      t*
```

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

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

Universal

```
Heap (Recency)
 @CounterCls:s @CounterIncr:s
 @CounterGet:s
Intervals
```

Python

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

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

Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P      t*
```

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

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

Universal

```
Heap (Recency)
 @CounterCls:s @CounterIncr:s
 @CounterGet:s
Intervals
```

Python

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


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

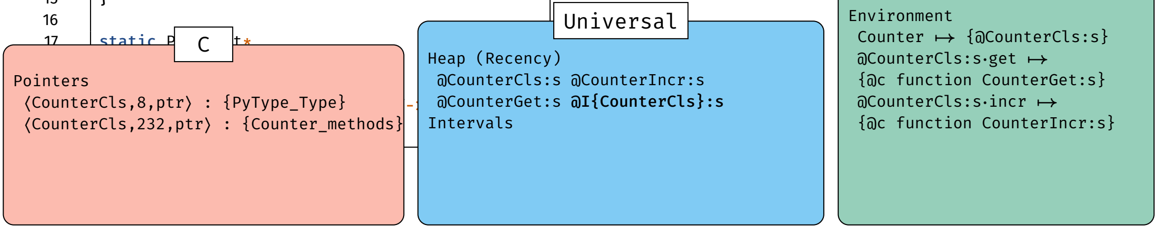
Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P     t*
```

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

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

Universal

Heap (Recency)
@CounterCls:s @CounterIncr:s
@CounterGet:s **@I{CounterCls}:s**
Intervals

Python

Attributes
@CounterCls:s ↦ {get, incr}

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

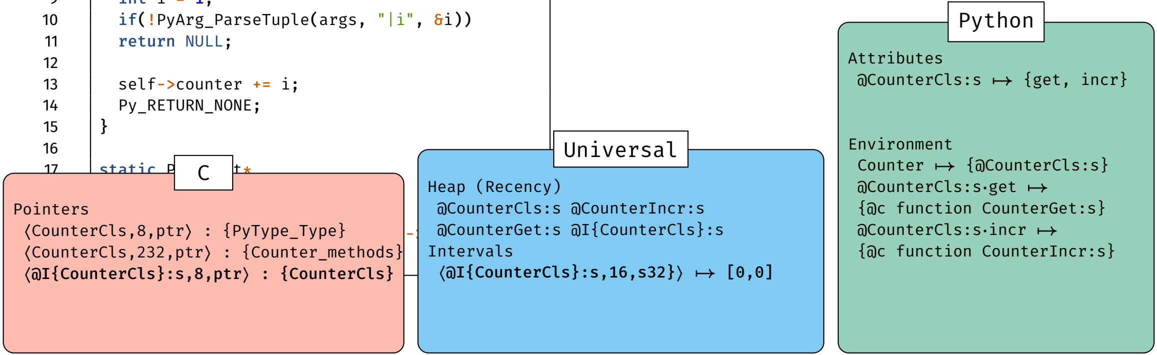
Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P
```

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

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

Universal

Heap (Recency)
@CounterCls:s @CounterIncr:s
@CounterGet:s @I{CounterCls}:s
Intervals
⟨@I{CounterCls}:s,16,s32}⟩ ↦ [0,0]

Python

Attributes
@CounterCls:s ↦ {get, incr}

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

Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P
```

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

Pointers

⟨CounterCls,8,ptr⟩ : {PyType_Type}

⟨CounterCls,232,ptr⟩ : {Counter_methods}

⟨@I{CounterCls}:s,8,ptr⟩ : {CounterCls}

Universal

Heap (Recency)

@CounterCls:s @CounterIncr:s

@CounterGet:s @I{CounterCls}:s

Intervals

⟨@I{CounterCls}:s,16,s32}⟩ ↦ [0,0]

Python

Attributes

@CounterCls:s ↦ {get, incr}

@I{CounterCls}:s ↦ ∅

Environment

Counter ↦ {@CounterCls:s}

@CounterCls:s·get ↦

{@c function CounterGet:s}

@CounterCls:s·incr ↦

{@c function CounterIncr:s}

c ↦ {@I{CounterCls}:s}

Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P t*
```

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

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

Universal

Heap (Recency)
@CounterCls:s @CounterIncr:s
@CounterGet:s @I{CounterCls}:s
Intervals
⟨@I{CounterCls}:s,16,s32}⟩ ↦ [0,0]

Python

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

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

Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P        t*
```

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

Pointers

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

Universal

Heap (Recency)

@CounterCls:s @CounterIncr:s
@CounterGet:s @I{CounterCls}:s **@I{int}:w**

Intervals

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

Python

Attributes

@CounterCls:s ↦ {get, incr}
@I{CounterCls}:s ↦ ∅

Environment

Counter ↦ {@CounterCls:s}
@CounterCls:s·get ↦
{@c function CounterGet:s}
@CounterCls:s·incr ↦
{@c function CounterIncr:s}
c ↦ {@I{CounterCls}:s}
power ↦ {@I{int}:w}

Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P        t*
```

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

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

Universal

```
Heap (Recency)
 @CounterCls:s @CounterIncr:s
 @CounterGet:s @I{CounterCls}:s @I{int}:w
Intervals
 ⟨@I{CounterCls}:s,16,s32}⟩ ↦ [0,0]
 power ↦ [0,127]
```

Python

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

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

Analysis of the Example

counter.c

```c
typedef struct {
  PyObject_HEAD;
  int counter;
} 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 P        t*
```

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

Pointers
- ⟨CounterCls,8,ptr⟩ : {PyType_Type}
- ⟨CounterCls,232,ptr⟩ : {Counter_methods}
- ⟨@I{CounterCls}:s,8,ptr⟩ : {CounterCls}
- **args : {@tuple[1]:s}**
- **self : {@I{CounterCls}:s}**

Universal

Heap (Recency)
- @CounterCls:s @CounterIncr:s **@tuple[1]:s**
- @CounterGet:s @I{CounterCls}:s @I{int}:w

Intervals
- ⟨@I{CounterCls}:s,16,s32⟩ ↦ [0,0]
- power ↦ [0,127]
- **@tuple[1]:s·[0] ↦ [0,2^{127} − 1]**

Python

Attributes
- @CounterCls:s ↦ {get, incr}
- @I{CounterCls}:s ↦ ∅

Environment
- Counter ↦ {@CounterCls:s}
- @CounterCls:s·get ↦ {@c function CounterGet:s}
- @CounterCls:s·incr ↦ {@c function CounterIncr:s}
- c ↦ {@I{CounterCls}:s}
- power ↦ {@I{int}:w}
- **@tuple[1]:s·[0] ↦ {@I{int}:w}**

Combining C and Python – Current analyses

Library	|C|	|Py|	Tests	🕒	✓		✅		Assertions	Py $\leftrightsquigarrow$ 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

Conclusion

Conclusion

An analysis of Python programs with C modules

- Combining previous C and Python analyses,

Conclusion

An analysis of Python programs with C modules

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

Conclusion

An analysis of Python programs with C modules

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

Conclusion

An analysis of Python programs with C modules

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

Conclusion

An analysis of Python programs with C modules

- Combining 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

An analysis of Python programs with C modules

- Combining 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

An analysis of Python programs with C modules

- Combining 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

An analysis of Python programs with C modules

- Combining 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 (Java/C).