A Multilanguage Static Analysis of Python/C Programs with Mopsa

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

Binsec Seminar
26 November 2021
Introduction
average.py

```python
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 r1 = average([1, 2, 3])
9 r2 = average(['a', 'b', 'c'])
```

TypeError: unsupported operand type(s) for '+': 'int' and 'str'

argslen.c

```c
1
#include <string.h>
2
3 int main(int argc, char *argv[])
4 {
5     int i = 0;
6     for (char **p = argv; *p; p++) {
7         strlen(*p); // valid string
8         i++; // no overflow
9     }
10    return 0;
11 }
```

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.
## Dynamic programming languages

### Growing popularity

JavaScript #1, Python #2 on GitHub

[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 GitHub¹

### New features

- Object orientation,
- Dynamic typing,
- Dynamic object structure,

¹https://octoverse.github.com/#top-languages
## Dynamic programming languages

### Growing popularity

JavaScript #1, Python #2 on GitHub¹

### New features

- Object orientation,
- Dynamic typing,
- Dynamic object structure,
- Introspection operators,

¹[https://octoverse.github.com/#top-languages](https://octoverse.github.com/#top-languages)
Dynamic programming languages

Growing popularity
JavaScript #1, Python #2 on GitHub

New features
- Object orientation,
- Dynamic typing,
- Dynamic object structure,
- Introspection operators,
- `eval`.

---

¹https://octoverse.github.com/#top-languages
No standard

- CPython is the reference
  → manual inspection of the source code and handcrafted tests
Python’s specificities

No standard

- CPython is the reference
  ➞ manual inspection of the source code and handcrafted tests

Operator redefinition

- Calls, additions, attribute accesses
- Operators eventually call overloaded __methods__

```
class Protected:
    def __init__(self, priv):
        self._priv = priv
    def __getattr__(self, a):
        if a[0] == "_": raise AttributeError("protected")
        return object.__getattribute__(self, a)

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())
```

Dual type system

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

Exceptions

- 1 + "a" ⇝ TypeError
- l[len(l) + 1] ⇝ IndexError

Fspath (from standard library)

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

def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        r = p.__fspath__()
        if isinstance(r, (str, bytes)):
            return r
        raise TypeError

fspath("/dev" if random() else Path())
```

Python’s specificities (II)

Dual type system

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

Exceptions

Exceptions rather than specific values

- `1 + "a"` ➝ `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())
```

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)

- 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
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 object representations (Python objects, C structs)
# Combining C and Python – motivation

<table>
<thead>
<tr>
<th>One in five of the top 200 Python libraries contains C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ To bring better performance (numpy)</td>
</tr>
<tr>
<td>▶ To provide library bindings (pygit2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pitfalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Different values (arbitrary-precision integers in Python, bounded in C)</td>
</tr>
<tr>
<td>▶ Different object representations (Python objects, C structs)</td>
</tr>
<tr>
<td>▶ Different runtime-errors (exceptions in Python)</td>
</tr>
</tbody>
</table>
## 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 object representations (Python objects, C structs)
- Different runtime-errors (exceptions in Python)
- Garbage collection
1. Introduction
2. Mopsa
3. A Concrete Example
4. Concrete Multilanguage Semantics
5. Experimental Evaluation
6. Conclusion
Mopsa
A program analysis workflow

Avering 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)
```

Proved safe?

▶ $m // (i+1)$

▶ $l[i]$

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

$m$ ↦ $\sharp$

$i$ ↦ $\sharp$

els ($l$) ↦ $\sharp$

Task · weight ↦ $\sharp$

Numeric abstraction (intervals)

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

els ($l$) ∈ $[0, 20]$

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

$0 \leq \sharp$ Task · weight $\leq 20$

Attributes abstraction

$\sharp$ Task ↦ $(\{weight\}, \emptyset)$

Conclusion

▶ Different domains depending on the precision

▶ Use of auxiliary variables (underlined)
A program analysis workflow

Avering numbers

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Searching for a loop invariant (l. 4)

Environment abstraction

\[ m \mapsto @_{\text{int}}^\# \quad i \mapsto @_{\text{int}}^\# \]

Proved safe?

- \( m \div (i+1) \)
- \( l[i] \)
A program analysis workflow

Avering numbers

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Proved safe?

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

Searching for a loop invariant (l. 4)

Stateless domains: list content,

Environment abstraction

```
m ↦ int#  i ↦ int#  els(l) ↦ int#
```

Conclusion

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

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Proved safe?

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

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

$m \mapsto \int \quad i \mapsto \int \quad els(l) \mapsto \int$

Numeric abstraction (intervals)

$m \in [0, +\infty) \quad els(l) \in [0, 20] \quad i \in [0, +\infty)$
A program analysis workflow

Searching for a loop invariant (l. 4)
Stateless domains: list content, list length

Environment abstraction

$$m \longmapsto @_{\text{int}^#} \quad i \longmapsto @_{\text{int}^#} \quad \text{els}(l) \longmapsto @_{\text{int}^#}$$

Numeric abstraction (intervals)

$$m \in [0, +\infty) \quad \text{els}(l) \in [0, 20]$$
$$\text{len}(l) \in [5, 10] \quad i \in [0, 10]$$

Avering numbers

```python
def average(l):
    m = 0
    for i in range(len(l)):
        m = m + l[i]
    m = m // (i + 1)
    return m
```

Proved safe?

- $$m \div (i+1)$$
- $$l[i]$$
A program analysis workflow

Avering numbers

```python
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) for i in range(randint(5, 10))]
9  m = average(l)
```

Searching for a loop invariant (l. 4)
Stateless domains: list content, list length

**Environment abstraction**

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

**Numeric abstraction (polyhedra)**

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

Proved safe?

- \( m // (i+1) \)
- \( l[i] \)
A program analysis workflow

Averaging tasks

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

    def average(l):
        m = 0
        for i in range(len(l)):
            m = m + l[i].weight
        m = m // (i + 1)
        return m

l = [Task(randint(0, 20)) for i in range(randint(5, 10))]
m = average(l)
```

Searching for a loop invariant (l. 4)
Stateless domains: list content, list length

Environment abstraction

\[
m \mapsto \int \quad i \mapsto \int \
\text{els}(l) \mapsto \text{Task} \\
\text{Task} \cdot \text{weight} \mapsto \int
\]

Numeric abstraction (polyhedra)

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

Attributes abstraction

\[
\text{Task} \mapsto (\{\text{weight}\}, \emptyset)
\]

Proved safe?

- m // (i+1)
- l[i].weight
A program analysis workflow

Averaging tasks

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

    def average(l):
        m = 0
        for i in range(len(l)):
            m = m + l[i].weight
        m = m // (i + 1)
        return m

l = [Task(randint(0, 20)) for i in range(randint(5, 10))]
m = average(l)
```

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

- m ↦ @# m
- i ↦ @# i
- l ↦ @# l
- Task ↦ @# Task

Conclusion

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

Attributes abstraction

- @# Task ↦ (weight

Proved safe?

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

Modular Open Platform for Static Analysis

---

Overview of Mopsa

Modular Open Platform for Static Analysis²

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

Overview of Mopsa

**Modular Open Platform for Static Analysis**

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

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

---

Overview of Mopsa

**Modular Open Platform for Static Analysis**

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

---

Dynamic, semantic iterators with delegation

for (init; cond; incr) body

C.iterators.loops

Rewrite and analyze recursively

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

for target in iterable: body

Python.Desugar.Loops

◦ Rewrite and analyze recursively

◦ Optimize for some semantic cases

it = iter(iterable)
while (1) {
    try: target = next (it)
    except StopIteration: break
    body
}
clean it

Universal.Iterators.Loops

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

Computes fixpoint using widening
Dynamic, semantic iterators with delegation

\[
\text{for}(\text{init}; \text{cond}; \text{incr}) \text{ body}
\]

- **Python.Desugar.Loops**
  - Rewrite and analyze recursively
  - Optimize for some semantic cases

\[
\text{it} = \text{iter}(\text{iterable})
\]

\[
\text{while}(1)\
\quad \text{try: target} = \text{next}(\text{it})\
\quad \text{except StopIteration: break}\
\quad \text{body}
\]

- **Universal.Iterators.Loops**
  - Matches `while(...){}...`
  - Computes fixpoint using widening
Dynamic, semantic iterators with delegation

for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

Universal.Iterators.Loops
Matches while(...) {...}
Computes fixpoint using widening
Dynamic, semantic iterators with delegation

for(init; cond; incr) body

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

C.iterators.loops

Rewrite and analyze recursively

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

for target in iterable: body

Universal.Iterators.Loops

Matches while (...){...}
Computes fixpoint using widening
Dynamic, semantic iterators with delegation

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

```
C.iterators.loops

Rewrite and analyze recursively
```

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

```
Python.Desugar.Loops

- Rewrite and analyze recursively
- Optimize for some semantic cases
```

```
Universal.Iterators.Loops

Matches while(...) {...}
Computes fixpoint using widening
```
Dynamic, semantic iterators with delegation

for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

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

for target in iterable: body

Python.Desugar.Loops
◦ Rewrite and analyze recursively
◦ Optimize for some semantic cases

it = iter(iterable)
while(1) {
  try: target = next(it)
  body
  except StopIteration: break
}
clean it

Universal.Iterators.Loops
Matches while(...) {...}
Computes fixpoint using widening
A Concrete Example
Combining C and Python – example

counter.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

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

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

- **power ≤ 30** \(\Rightarrow r = 2^{\text{power}}\)
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()
```

- **power ≤ 30 ⇒ r = 2^{power}**
- **32 ≤ power ≤ 64**: OverflowError: signed integer is greater than maximum
- **power ≥ 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()
```

- $\text{power} \leq 30 \Rightarrow r = 2^{\text{power}}$
- $\text{power} = 31 \Rightarrow r = -2^{31}$
- $32 \leq \text{power} \leq 64$: OverflowError: signed integer is greater than maximum
- $\text{power} \geq 64$: 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: ...
```

- **No raised exceptions**: $\Rightarrow$ missed errors
- **Only types**: 
- **Typeshed**: type annotations for the standard library, used in previous work:

Our approach

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

Type annotations

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

- No raised exceptions $\Rightarrow$ 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 \(\Rightarrow\) 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 $\implies$ 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 ⇒ 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

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

No integer wrap-around in Python

Some effects can't be written in pure Python (e.g., read-only attributes)

Our approach

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

Type annotations

Rewrite into Python code

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

- No integer wrap-around in Python
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?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite into Python code</td>
</tr>
<tr>
<td><strong>Drawbacks of the current approaches</strong></td>
</tr>
<tr>
<td>► Not the real code</td>
</tr>
</tbody>
</table>
### How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th><strong>Type annotations</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rewrite into Python code</strong></td>
</tr>
<tr>
<td><strong>Drawbacks of the current approaches</strong></td>
</tr>
<tr>
<td>- Not the real code</td>
</tr>
<tr>
<td>- Not automatic: manual conversion</td>
</tr>
</tbody>
</table>
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
- Switch from one language to the other just as the program does
- Reuse previous analyses of C and Python
- Detect runtime errors in Python, in C, and at the boundary
### How to analyze multilanguage programs?

**Type annotations**

**Rewrite into Python code**

**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?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite into Python code</td>
</tr>
<tr>
<td>Drawbacks of the current approaches</td>
</tr>
<tr>
<td>▶ Not the real code</td>
</tr>
<tr>
<td>▶ Not automatic: manual conversion</td>
</tr>
<tr>
<td>▶ Not sound: some effects are not taken into account</td>
</tr>
<tr>
<td>Our approach</td>
</tr>
<tr>
<td>▶ Analyze both the C and Python sources</td>
</tr>
<tr>
<td>▶ Switch from one language to the other just as the program does</td>
</tr>
</tbody>
</table>
How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Rewrite into Python code</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Drawbacks of the current approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>► Not the real code</td>
</tr>
<tr>
<td>► Not automatic: manual conversion</td>
</tr>
<tr>
<td>► Not sound: some effects are not taken into account</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>► Analyze both the C and Python sources</td>
</tr>
<tr>
<td>► Switch from one language to the other just as the program does</td>
</tr>
<tr>
<td>► Reuse previous analyses of C and Python</td>
</tr>
</tbody>
</table>
How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite into Python code</td>
</tr>
<tr>
<td>Drawbacks of the current approaches</td>
</tr>
<tr>
<td>Our approach</td>
</tr>
</tbody>
</table>

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

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

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

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

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```
```c
typedef struct {
  PyObject_HEAD;
  int count;
} Counter;

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

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

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

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

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

```python
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

```python
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

---

**Limitations**

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

Limitations
- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
- Manual modelization from CPython's source code
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
**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
Experimental Evaluation
From distinct Python and C analyses...

**Python Analyses**
- Py.program
- Py.desugar
- Py.flow
- Py.intraproc
- Py.loops
- Py.interproc
- Py.libraries
- Py.data_model
- Py.objects

**Python Specific**
- Py.environment
- Py.attributes
- Py.lists
- Py.tuples
- Py.dicts

**Universal Analyses**
- U.intraproc
- U.loops
- U.interproc

**Universal Specific**
- U.recency
- U.intervals

**Cartesian Product**
- Py.environment × Py.attributes
- Py.lists × Py.tuples × Py.dicts

**Reduced Product**
- Py.environment ◦ U.recency
- U.intervals ◦ U.strings

**C Analyses**
- C.program
- C.desugar
- C.goto
- C.intraproc
- C.loops
- C.interproc
- C.stubs
- C.libraries
- C.files

**C Specific**
- C.cells
- C.strings

**Universal Analyses**
- C.machineNum
- C.pointers

**Universal Specific**
- U.recency

**Universal Specific**
- U.intervals
- U.linearRel
... to a multilanguage analysis!
... to a multilanguage analysis!

Implementation LOC

Part LOC

− Universal
− C specific
− Python specific

− Sequence
− Reduced product
− Cartesian product
− Composition
... to a multilanguage analysis!

**Implementation LOC**

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
</tbody>
</table>

**Universal Implementation LOC**

<table>
<thead>
<tr>
<th>Framework LOC</th>
<th>Universal LOC</th>
<th>C Specific LOC</th>
<th>Python Specific LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>13200</td>
<td>5600</td>
<td>11700</td>
<td>12600</td>
</tr>
</tbody>
</table>

- Universal: Sequence
- C Specific: Reduced product
- Python Specific: Cartesian product
- Composition: Composition

**Diagram Details**

- CPython
- PyObject
- PyEnvironment
- PyProgram
- PyLibraries
- PyMethods
- PyDesugar
- PyExceptions
- PyObjects
- PyDataModel
- PyEnvironment
- CProgram
- CDesugar
- CGoto
- CStubs
- CLibraries
- CFiles
- CStrings
- CPointers
- CMachineNum
- UIntraproc
- ULoops
- UInterproc
- URecency
- UIntervals
- Sequence
- Reduced product
- Cartesian product
- Composition

**Implementation LOC**

- Implementation LOC: 13200
- Framework LOC: 13200
- Universal LOC: 5600
- C Specific LOC: 11700
- Python Specific LOC: 12600
... to a multilanguage analysis!

**Implementation LOC**

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
<tr>
<td>Universal</td>
<td>5600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation LOC</th>
<th>13200</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>5600</td>
</tr>
</tbody>
</table>
... to a multilanguage analysis!

Implementation LOC

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

Part LOC

- Framework: 13200 LOC
- Universal: 5600 LOC
- C: 11700 LOC

Python specific

- Py.object
- Py.data_model

C specific

- C-files
- C.stubs

Universal

- C goto
- Composition
- Reduced product
- Cartesian product
- Sequence

Implementation LOC

- Universal: 5600 LOC
- C: 11700 LOC
- Python: 12600 LOC
- Multilanguage: 2500 LOC

Universal LOC

- Implementation LOC: 13200 LOC
- Framework: 13200 LOC
- Universal: 5600 LOC
- C: 11700 LOC

C LOC

- Framework: 13200 LOC
- Universal: 5600 LOC
- C: 11700 LOC
... to a multilanguage analysis!

**Implementation LOC**

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
<tr>
<td>Universal</td>
<td>5600</td>
</tr>
<tr>
<td>C</td>
<td>11700</td>
</tr>
<tr>
<td>Python</td>
<td>12600</td>
</tr>
</tbody>
</table>

- Universal
- C specific
- Python specific
- Universal LOC: 5600
- C LOC: 11700
- Python LOC: 12600
- Sequence
- Reduced product
- Cartesian product
- Composition
... to a multilanguage analysis!

**Implementation LOC**

<table>
<thead>
<tr>
<th>Part</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>13200</td>
</tr>
<tr>
<td>Universal</td>
<td>5600</td>
</tr>
<tr>
<td>C</td>
<td>11700</td>
</tr>
<tr>
<td>Python</td>
<td>12600</td>
</tr>
<tr>
<td>Multilanguage</td>
<td>2500</td>
</tr>
</tbody>
</table>
## 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 | unsafe C checks | safe C checks | total C checks | total C checks | Assertions | \(\text{Py \leftrightarrow C}\) |
|--------------------|--------|--------|-------|-----------------|---------------|----------------|----------------|------------|---------------------------------|
| noise              | 722    | 675    | 15/15 | 18s             | 99.6%         | (4952)         | 100.0%         | (1738)     | 0/21                            |
| ahocorasick        | 3541   | 1336   | 46/92 | 54s             | 93.1%         | (1785)         | 98.0%         | (4937)     | 30/88                           |
| levenshtein        | 5441   | 357    | 17/17 | 1.5m            | 79.9%         | (3106)         | 93.2%         | (1719)     | 0/38                            |
| cdistance          | 1433   | 912    | 28/28 | 1.9m            | 95.3%         | (1832)         | 98.3%         | (11884)    | 88/207                          |
| llist              | 2829   | 1686   | 167/194 | 4.2m           | 99.0%         | (5311)         | 98.8%         | (30944)    | 235/691                         |
| bitarray           | 3244   | 2597   | 159/216 | 4.2m          | 96.3%         | (4496)         | 94.6%         | (21070)    | 100/378                         |
Theoretical frameworks

- Matthews and Findler\textsuperscript{3} boundary functions as value conversions between two languages.
- Buro, Crole, and Mastroeni\textsuperscript{4} generic framework for combining analyses of different languages.


Related work (II)

Around the Java Native Interface (JNI)

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

- Effects of C code on Java heap modelized using JVML\(^5\)
- Type inference of Java objects in C code\(^6\)
- Extraction of C callbacks to Java\(^7\)

- Modular analyses
- No numeric information
- Missing C runtime errors

---


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

Questions

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné
<table>
<thead>
<tr>
<th></th>
<th>Table of backups</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Usual implementations</td>
</tr>
<tr>
<td>3</td>
<td>Analysis of the multilanguage example</td>
</tr>
<tr>
<td>4</td>
<td>Example of recency abstraction</td>
</tr>
</tbody>
</table>
Usual implementations

Programming Language(s) C

Static Simplification

Simplified IR Clight

Iterator

Abstract Domain 1
Arrays

Abstract Domain 2
Intervals

Abstract Domain 3
Congruences

Static simplification
- Infer’s SIL IR for Java/C/C++
- May reduce the precision

Tree of abstract domains
No sharing (e.g., len(l), els(l), i)

Layered signatures
Different signatures throughout the abstract tree, and the iterator
Analysis of the multilanguage 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) {
    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()
```
### Analysis of the multilanguage 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) {
    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()
```

---

**Python**

**Attributes**

- @CounterCls $\leftrightarrow \{\text{get, incr}\}$

**Environment**

- Counter $\leftrightarrow \{@\text{CounterCls}\}$
- @CounterCls.get $\leftrightarrow \{@\text{c function CounterGet}\}$
- @CounterCls.incr $\leftrightarrow \{@\text{c function CounterIncr}\}$

**Heap (Recency)**

- @CounterCls @CounterIncr
- @CounterGet

**Intervals**

- Counter $\leftrightarrow \{\text{CounterCls}\}$
- @CounterCls $\leftrightarrow \{\text{Counter_methods}\}$

---

**Universal**

- Counter $\leftrightarrow \{\text{CounterCls}\}$
- @CounterCls $\leftrightarrow \{\text{Counter_methods}\}$

---

**Pointers**

- {CounterCls, 8, ptr} : {PyType_Type}
- {CounterCls, 232, ptr} : {Counter_methods}
Analysis of the multilanguage 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) {
    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()
```

**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

Pointers
- `{CounterCls, 8, ptr}` : `{PyType_Type}
- `{CounterCls, 232, ptr}` : `{Counter_methods}`
Analysis of the multilanguage 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) {
    return Py_BuildValue("i", self->counter);
}
```

**count.py**

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

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

**Flow**

- **Attributes**
  - `CounterCls` → `{get, incr}`

- **Environment**
  - `Counter` → `{@CounterCls}`
  - `@CounterCls` → `CounterGet`
  - `@CounterCls` → `CounterIncr`

- **Points**
  - `{CounterCls, 8, ptr} : {PyType_Type}
  - `{CounterCls, 232, ptr} : {Counter_methods}
  - `{@I{CounterCls}, 8, ptr} : {CounterCls}`
Analysis of the multilanguage 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)
{
    return Py_BuildValue("i", self->counter);
}
```

**count.py**
```python
def Counter():
    pass
def randrange(min):
    pass
c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

**Python**
- Attributes
  - @CounterCls → {get, incr}
- Environment
  - Counter → @CounterCls
  - Counter → @I{CounterCls}
- Universal
  - Heap (Recency)
    - @CounterCls
    - @CounterIncr
    - @CounterGet
  - Intervals
    - ⟨@I{CounterCls},16,s32⟩ ↦→ [0, 0]

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

**Universal**
- C

- ⟨I{CounterCls},16,s32⟩ ↦→ [0, 0]
Analysis of the multilanguage 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) {
    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()
```

**Universal**

- **Attributes**
  - `@CounterCls` ➞ `{get, incr}
  - `@I{CounterCls}` ➞ ∅

- **Environment**
  - `Counter` ➞ `{@CounterCls}
  - `Counter.cls` ➞ `{c function CounterGet}
  - `CounterCls.incr` ➞ `{c function CounterIncr}

- **Pointers**
  - `CounterCls, 8, ptr` : `{PyType_Type}
  - `CounterCls, 232, ptr` : `{Counter_methods}
  - `@I{CounterCls}, 8, ptr` : `{CounterCls}`

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

- **Intervals**
  - `{@I{CounterCls}, 16, s32}` ➞ `[0, 0]`
### Analysis of the multilanguage 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)
{
    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()
```

---

**Python**

- **Attributes**
  - `@CounterCls` $\leftrightarrow$ `get, incr`
  - `@I{CounterCls}` $\leftrightarrow$ `∅`

- **Environment**
  - `Counter` $\leftrightarrow$ `{@CounterCls}`
  - `CounterCls.get` $\leftrightarrow$ `{@c function CounterGet}`
  - `CounterCls.incr` $\leftrightarrow$ `{@c function CounterIncr}`
  - `c` $\leftrightarrow$ `{@I{CounterCls}}`

**Universal**

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

- **Intervals**
  - `{@I{CounterCls}, 16, s32}` $\mapsto$ `[0, 0]`

**Pointers**

- `{CounterCls, 8, ptr}` : `{PyType_Type}`
- `{CounterCls, 232, ptr}` : `{Counter_methods}`
- `{@I{CounterCls}, 8, ptr}` : `{CounterCls}`
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) {
    return Py_BuildValue("i", self->counter);
}

from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]
```

Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]
Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]
```

Initialization:
Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]
```

Allocation: $\#(\text{Task}, r) \sim \#(\text{Task}, o)$
Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]
```

Initialization:

\[
\begin{align*}
\@\#(\text{Task}, r) \cdot \text{weight} & \mapsto [2, 2] \\
\@\#(\text{Task}, o) \cdot \text{weight} & \mapsto [2, 2]
\end{align*}
\]
Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]

Allocation: @#(Task, r) \sim \oslash@#(Task, o)
```

```
@#(Task, r) \cdot weight \mapsto [2, 2]
@#(Task, r) \cdot weight \mapsto [1, 1]
@#(Task, o) \cdot weight \mapsto [2, 2]
@#(Task, r) \mapsto [2, 2] \sqcup [1, 1]
```
Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]
```

Initialization:

\[
\begin{align*}
\&(\text{Task}, r) \cdot \text{weight} &\mapsto [2, 2] \\
\&(\text{Task, o}) \cdot \text{weight} &\mapsto [1, 1] \\
\&(\text{Task, r}) \cdot \text{weight} &\mapsto [2, 2] \\
\&(\text{Task, o}) \cdot \text{weight} &\mapsto [4, 4] \\
\&(\text{Task, o}) \cdot \text{weight} &\mapsto [1, 2]
\end{align*}
\]
Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]

Allocation: @^#(Task, r) \sim \@^#(Task, o)

\{ 
\begin{align*}
@^#(Task, r) \cdot weight & \mapsto [2, 2] \\
@^#(Task, o) \cdot weight & \mapsto [1, 1] \\
\end{align*}
\}
```

```
Example of recency abstraction

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

l = [Task(2), Task(1), Task(4), Task(5)]
```

Initialization:

\[
\begin{align*}
\&\#(\text{Task}, r) \cdot \text{weight} &\rightarrow [2, 2] \\
&\#(\text{Task}, o) \cdot \text{weight} &\rightarrow [1, 1] \\
&\#(\text{Task}, r) \cdot \text{weight} &\rightarrow [1, 4] \\
&\#(\text{Task}, o) \cdot \text{weight} &\rightarrow [1, 2] \\
&\#(\text{Task}, r) \cdot \text{weight} &\rightarrow [4, 4] \\
&\#(\text{Task}, o) \cdot \text{weight} &\rightarrow [2, 2] \\
&\#(\text{Task}, r) \cdot \text{weight} &\rightarrow [5, 5] \\
&\#(\text{Task}, o) \cdot \text{weight} &\rightarrow [1, 2]
\end{align*}
\]