From Python to the French Tax Code: Applying Formal Methods on Real Systems

Raphaël Monat
SyCoMoRES team
rmonat.fr
Introduction

Curriculum

- ENS de Lyon
- PhD student, LIP6
- ATER, LIP6 CRCN, Inria Lille


Hongseok Yang @ Oxford, 3 months
Eva Darulova @ MPI-SWS, 5 months
Antoine Miné

Research field: formal methods
⇒ Improve confidence in software.
⇒ Theory: formal definition and reasoning over systems
⇒ Practice: software development
Introduction

Curriculum

- ENS de Lyon
- PhD student, LIP6
- ATER, LIP6 CRCN, Inria Lille


Hongseok Yang @ Oxford, 3 months
Eva Darulova @ MPI-SWS, 5 months
Antoine Miné

Research field: formal methods

⇒ Improve confidence in software.
## Research field: formal methods

⇒ Improve confidence in software.

Means

- Theory: formal definition and reasoning over systems
- Practice: software development
## Personal methodology

Constant back and forth between theory and practice

1. Find interesting bugs, properties or systems to study (GitHub, ...)
2. Theoretical study and solution
3. Implementation and experimental validation (on 1)

### Studied systems

- Python programs using C libraries:
  - Static analysis
    - Abdelraouf Ouadjaout (LIP6)
    - Antoine Miné (LIP6)
- Implementation of the French tax code:
  - Compiler, modernization
    - Denis Merigoux (Inria Prosecco)
    - Jonathan Protzenko (MSR)
Introduction

Personal methodology
Constant back and forth between theory and practice

1. Find interesting bugs, properties or systems to study (GitHub, ...)
2. Theoretical study and solution
3. Implementation and experimental validation (on 1)

Studied systems

- Python programs using C libraries → static analysis
  - Abdelraouf Ouadjaout (LIP6)
  - Antoine Miné (LIP6)
- Implementation of the French tax code → compiler, modernization
  - Denis Merigoux (Inria Prosecco)
  - Jonathan Protzenko (MSR)
## Introduction

### Personal methodology

Constant back and forth between theory and practice

1. Find interesting bugs, properties or systems to study (GitHub, ...)
2. Theoretical study and solution
3. Implementation and experimental validation (on 1)

### Studied systems

- **Python programs using C libraries (∼ static analysis**
  - Abdelraouf Ouadjaout (LIP6)
  - Antoine Miné (LIP6)

- **Implementation of the French tax code (∼ compiler, modernization**
  - Denis Merigoux (Inria Prosecco)
  - Jonathan Protzenko (MSR)
Software verification

Some bugs may go undetected!

Would there be a way to automatically prove programs correct?
Software verification

$S[[\text{prog}]]$

Bad states

Cheap approach: test $\text{prog}$.

Some bugs may go undetected!

Would there be a way to automatically prove programs correct?
Software verification

Cheap approach: test \textit{prog}.

Some bugs may go undetected!
Software verification

Cheap approach: test \textit{prog}.

Bad states
Software verification

Cheap approach: test $prog$.

Bad states
Cheap approach: test $prog$. 

Would there be a way to automatically prove programs correct?
Cheap approach: test \textit{prog}.

Some bugs may go undetected!
Software verification

Cheap approach: test $prog$. Some bugs may go undetected!
Software verification

Cheap approach: test \textit{prog}. Some bugs may go undetected!

Would there be a way to automatically prove programs correct?
All reported errors are true errors.

Sound

All true errors are reported
An impossibility theorem

All reported errors are true errors

Complete

Sound

All true errors are reported
An impossibility theorem

Guaranteed Termination

Complete

Sound

All reported errors are true errors

All true errors are reported
An impossibility theorem

Rice’s theorem

Guaranteed Termination

Sound

Complete

All reported errors are true errors

All true errors are reported
An impossibility theorem

Guaranteed Termination

Complete

Sound

Rice’s theorem

∅

Abstract Interpretation

All reported errors are true errors

All true errors are reported

Rice’s theorem
Abstract interpretation – the big picture

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

\[ D \text{ (concrete)} \rightarrow_\gamma S^\sharp [\text{prog}] \]

\[ S^\sharp [\text{prog}] \rightarrow D^\# \text{ (abstract)} \]

Program proved safe

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

\[ D (\text{concrete}) \]

\[ S[\text{prog}] \]

\[ \text{Bad states} \]

\[ \gamma \]

\[ S^\#[\text{prog}] \]

\[ D^\# (\text{abstract}) \]

\[ \text{Bad states} \]

True alarm

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

\[ \mathcal{D} \text{ (concrete)} \]

Bad states

\[ \mathcal{S}[prog] \]

\[ \mathcal{D}^\# \text{ (abstract)} \]

Bad states

\[ \mathcal{S}^\#[prog] \]

False alarm (Abstraction too coarse)

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Abstract interpretation – the big picture

$D$ (concrete)

$\mathcal{S}[\text{prog}]$

Bad states

$\gamma$

$\mathcal{D}^\#$ (abstract)

$\mathcal{S}^\#[\text{prog}]$

Bad states

Unsound analysis
(shouldn’t happen)

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977
Critical software certification through static analysis

Bertrane, P. Cousot, R. Cousot, Feret, Mauborgne, A. Miné, and Rival. “Static analysis and verification of aerospace software by abstract interpretation”. AIAA Infotech@Aerospace (I@A 2010) 2010
Critical software certification through static analysis

Democratizing static analysis?

- General C software (dynamic allocation, ...)
- Other languages
- Framework to implement analyses

Embedded C
- Generated code
- Dynamic allocation

Bertrane, P. Cousot, R. Cousot, Feret, Mauborgne, A. Miné, and Rival. “Static analysis and verification of aerospace software by abstract interpretation”. AIAA Infotech@Aerospace (I@A 2010) 2010
Critical software certification through static analysis

Democratizing static analysis?
- General C software (dynamic allocation, ...)
- Other languages
- Framework to implement analyses

Bertrane, P. Cousot, R. Cousot, Feret, Mauborgne, A. Miné, and Rival. “Static analysis and verification of aerospace software by abstract interpretation”. AIAA Infotech@Aerospace (I@A 2010) 2010
Dynamic programming languages

Most popular languages on GitHub

- **JavaScript**: 18%
- **Python**: 14%
- **Java**: 12%
- **TypeScript**: 8%
- **Go**: 8%
- **C++**: 6%
- **Ruby**: 6%
- **PHP**: 5%
- **Other (<5%)**: 23%
Dynamic programming languages

Most popular languages on GitHub

- JavaScript: 23%
- Python: 18%
- Java: 14%
- TypeScript: 12%
- Go: 8%
- C++: 6%
- Ruby: 6%
- PHP: 5%
- Other (< 5%): 23%

New features:
- Dynamic typing
- Dynamic object structure
Outline

1. Introduction
2. A Taste of Python
3. Analyzing Python Programs
4. Analyzing Python Programs with C Libraries
5. A Modern Compiler for the French Tax Code
6. Conclusion
A Taste of Python
Python’s specificities

No standard

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

```python
class Protected:
    def __init__(self, priv):
        self._priv = priv

    def __getattribute__(self, attr):
        if attr[0] == '_':
            raise AttributeError('...
        return object.__getattribute__(self, attr)

a = Protected(42)
a._priv  # AttributeError raised
```
Python’s specificities

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__

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

a = Protected(42)
a._priv  # AttributeError raised
```
Python’s specificities (II)

Dual type system

- Nominal (classes, MRO)

```
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)**

#### 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

- \(1 + "a" \rightsquigarrow \text{TypeError}\)
- \(l[len(l) + 1] \rightsquigarrow \text{IndexError}\)

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())
```

Example Semantics – binary operators

\[ a_1 = \text{eval} e_1; \ a_2 = \text{eval} e_2 \]

- If \( \text{has\_field}(a_1, \_\_\_\_add\_\_)? \) is Yes:
  - If \( \text{has\_field}(a_2, \_\_\_\_radd\_\_)? \) is Yes, then:
    - \( a_3 = \text{call} \ a_1\text{'s \_\_\_\_add\_\_ on } a_1, a_2 \)
    - If \( a_3 == \text{NotImplemented} \) is Yes, then:
      - Result is \( a_3 \)
    - If \( a_3 == \text{NotImplemented} \) is No, then:
      - Type Error
  - If \( \text{has\_field}(a_2, \_\_\_\_radd\_\_) \&\& \text{type}(a_1) \neq \text{type}(a_2) \) is Yes, then:
    - \( a_3 = \text{call} \ a_2\text{'s \_\_\_\_radd\_\_ on } a_1, a_2 \)
    - If \( a_3 == \text{NotImplemented} \) is Yes, then:
      - Result is \( a_3 \)
    - If \( a_3 == \text{NotImplemented} \) is No, then:
      - Type Error
- If \( \text{has\_field}(a_1, \_\_\_\_add\_\_)? \) is No:
  - If \( \text{has\_field}(a_2, \_\_\_\_radd\_\_) \&\& \text{type}(a_1) < \text{type}(a_2) ? \) is Yes:
    - \( a_3 = \text{call} \ a_1\text{'s \_\_\_\_add\_\_ on } a_1, a_2 \)
    - If \( a_3 == \text{NotImplemented} \) is Yes, then:
      - Result is \( a_3 \)
    - If \( a_3 == \text{NotImplemented} \) is No, then:
      - Type Error
- If \( \text{has\_field}(a_1, \_\_\_\_add\_\_)? \) is No:
  - If \( \text{has\_field}(a_2, \_\_\_\_radd\_\_) \&\& \text{type}(a_1) \neq \text{type}(a_2) ? \) is No:
    - Result is \( a_3 \)
  - If \( \text{has\_field}(a_2, \_\_\_\_radd\_\_) \&\& \text{type}(a_1) < \text{type}(a_2) ? \) is No:
    - \( a_3 = \text{call} \ a_2\text{'s \_\_\_\_radd\_\_ on } a_1, a_2 \)
    - If \( a_3 == \text{NotImplemented} \) is Yes, then:
      - Result is \( a_3 \)
    - If \( a_3 == \text{NotImplemented} \) is No, then:
      - Type Error
class Infix(object):
    def __init__(self, func):
        self.func = func
    def __or__(self, other):
        return self.func(other)
    def __ror__(self, other):
        return Infix(lambda x: self.func(other, x))

instanceof = Infix(isinstance)
b = 5 | instanceof | int
@Infix
def padd(x, y):
    print(f"{x} + {y} = {x + y}")
    return x + y
c = 2 | padd | 3

Credits tomerfiliba.com/blog/Infix-Operators/
Analyzing Python Programs
Goal
Detect runtime errors: uncaught raised exceptions
Analysis | Overview

Goal
Detect runtime errors: uncaught raised exceptions

Supported constructs
Our analysis supports:
- Objects
- Exceptions
- Dynamic typing
- Introspection
- Permissive semantics
- Dynamic attributes
- Generators
- `super`
- Metaclasses
Analysis | Overview

**Goal**
Detect runtime errors: uncaught raised exceptions

**Supported constructs**
Our analysis supports:
- Objects
- Exceptions
- Dynamic typing
- Introspection
- Permissive semantics
- Dynamic attributes
- Generators
- `super`
- Metaclasses

**Unsupported constructs**
- Recursive functions
- `eval`
- Finalizers
Averaging numbers

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

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

m = average(l)
```

Averaging numbers

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

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

Searching for a loop invariant (l. 4)

Environment abstraction

\[ m \mapsto \text{int} \quad i \mapsto \text{int} \]

Proved safe?

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

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

Stateless domains: `list content`,

Environment abstraction:

\[
\begin{align*}
m & \mapsto \mathbb{\#_{int}} \quad i & \mapsto \mathbb{\#_{int}} \quad \text{els}(l) & \mapsto \mathbb{\#_{int}}
\end{align*}
\]

Proved safe?

- \[m // (i+1)\]
- \[m + l[i]\]
Analysis | Domains required

Averaging numbers

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

Stateless domains: list content, list length

Environment abstraction

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

Numeric abstraction (intervals)

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

Searching for a loop invariant (l. 4)

Proved safe?

- \( m \mapsto (i+1) \)
- \( m + l[i] \)
Analysis | Domains required

Averaging numbers

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

Stateless domains: list content, list length

Proved safe?

▶ m // (i+1)
▶ m + l[i]

Searching for a loop invariant (l. 4)

Environment abstraction

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

Numeric abstraction (intervals)

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

Conclusion

▶ Different domains depending on the precision
▶ Use of auxiliary variables (underlined)
Averaging 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)
Stateless domains: list content, list length

<table>
<thead>
<tr>
<th>Environment abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m \mapsto \mathbb{Z}_{\text{int}}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Numeric abstraction (polyhedra)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m \in [0, +\infty)$</td>
</tr>
<tr>
<td>$0 \leq i &lt; \text{len}(l)$</td>
</tr>
</tbody>
</table>

Proved safe?

▶ $m \, \text{//} \, (i+1)$
▶ $m \, + \, l[i]$
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
```

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

Environment abstraction

<table>
<thead>
<tr>
<th>Domain</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>$\oplus_{\text{int}}$</td>
</tr>
<tr>
<td>$i$</td>
<td>$\oplus_{\text{int}}$</td>
</tr>
<tr>
<td>$\text{els}(l)$</td>
<td>$\oplus_{\text{Task}}$</td>
</tr>
<tr>
<td>$\text{Task} \cdot \text{weight}$</td>
<td>$\oplus_{\text{int}}$</td>
</tr>
</tbody>
</table>

Numeric abstraction (polyhedra)

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

Attributes abstraction

$\text{Task} \mapsto (\{\text{weight}\}, \emptyset)$
Averaging tasks

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

Conclusion

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

Attributes abstraction

@Task → (weight, 0)

Proved safe?

► m // (i+1)
► m + l[i].weight
Modular Open Platform for Static Analysis\textsuperscript{1}

\textsuperscript{1}Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. 2019.
Modular Open Platform for Static Analysis

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

\[ \text{Py.list_len} \quad \text{Py.list_els} \]
\[ \varpi \text{numeric} \]
\[ \text{Reduced product} \]
\[ \text{Composition} \]

\(^1\)Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. 2019.
Modular Open Platform for Static Analysis

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

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

---

1Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. 2019.
Modular Open Platform for Static Analysis\(^1\)

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

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

- DAG of abstract domains
  - Composition
  - Cooperation

\(^1\)Journault, Miné, Monat, and Ouadjiaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. 2019.
Mopsa | Overview

Modular Open Platform for Static Analysis¹

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

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

- DAG of abstract domains
  - Composition
  - Cooperation

Mopsa | Dynamic, semantic iterators with delegation

<table>
<thead>
<tr>
<th>Universal.Iterators.Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matches <code>while(...) {...}</code></td>
</tr>
<tr>
<td>Computes fixpoint using widening</td>
</tr>
</tbody>
</table>
for(init; cond; incr) body
for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

Universal.Iterators.Loops
Matches while(...){...}
Computes fixpoint using widening
for(init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

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

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

for(init; cond; incr) body for target in iterable: body

C.iterators.loops
Rewrite and analyze recursively

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

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

Universal.Iterators.Loops
Matches while(...){...}
Computes fixpoint using widening
for(init; cond; incr) body

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

for target in iterable: body

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

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

for (init; cond; incr) body

C.iterators.loops
Rewrite and analyze recursively

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

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
Definition of the Value Analysis

Py.program × Py.desugar × Py.flow

U.intraproc × U.loops × U.interproc

Py.libraries × Py.data_model × Py.objects

× Py.environment × Py.attributes

× Py.lists × Py.tuples × Py.dicts

Universal Ⓓ Python specific

Sequence Ⓑ Cartesian product Ⓒ Composition

× U.recency

× U.intervals × U.strings

17
Comparison of the type and value analyses

Type analysis

▶ **ValueError (l. 3)**

R. Monat, A. Ouadjaout, and A. Miné. “Value and allocation sensitivity in static Python analyses”. SOAP@PLDI 2020
Comparison of the type and value analyses

<table>
<thead>
<tr>
<th>Averaging tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. class Task:</td>
</tr>
<tr>
<td>2. def <strong>init</strong>(self, weight):</td>
</tr>
<tr>
<td>3. if weight &lt; 0: raise ValueError</td>
</tr>
<tr>
<td>4. self.weight = weight</td>
</tr>
<tr>
<td>5.</td>
</tr>
<tr>
<td>6. def average(l):</td>
</tr>
<tr>
<td>7. m = 0</td>
</tr>
<tr>
<td>8. for i in range(len(l)):</td>
</tr>
<tr>
<td>9. m = m + l[i].weight</td>
</tr>
<tr>
<td>10. m = m // (i + 1)</td>
</tr>
<tr>
<td>11. return m</td>
</tr>
<tr>
<td>12.</td>
</tr>
<tr>
<td>13. l = []</td>
</tr>
<tr>
<td>14. for i in range(randint(5, 10)):</td>
</tr>
<tr>
<td>15. l.append(Task(randint(0, 20)))</td>
</tr>
<tr>
<td>16. m = average(l)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ValueError (l. 3)</td>
</tr>
<tr>
<td>2. IndexError (l. 9)</td>
</tr>
</tbody>
</table>

R. Monat, A. Ouadjaout, and A. Miné. “Value and allocation sensitivity in static Python analyses”. SOAP@PLDI 2020
Comparison of the type and value analyses

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

Type analysis

- ValueError (l. 3)
- IndexError (l. 9)
- ZeroDivisionError (l. 10)

R. Monat, A. Ouadjaout, and A. Miné. “Value and allocation sensitivity in static Python analyses”. SOAP@PLDI 2020
Comparison of the type and value analyses

<table>
<thead>
<tr>
<th>Class Task:</th>
</tr>
</thead>
<tbody>
<tr>
<td>def <strong>init</strong>(self, weight):</td>
</tr>
<tr>
<td>if weight &lt; 0: raise ValueError</td>
</tr>
<tr>
<td>self.weight = weight</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>def average(l):</td>
</tr>
<tr>
<td>m = 0</td>
</tr>
<tr>
<td>for i in range(len(l)):</td>
</tr>
<tr>
<td>m = m + l[i].weight</td>
</tr>
<tr>
<td>m = m // (i + 1)</td>
</tr>
<tr>
<td>return m</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>l = []</td>
</tr>
<tr>
<td>for i in range(randint(5, 10)):</td>
</tr>
<tr>
<td>l.append(Task(randint(0, 20)))</td>
</tr>
<tr>
<td>m = average(l)</td>
</tr>
</tbody>
</table>

Type analysis

- **ValueError** (l. 3)
- **IndexError** (l. 9)
- **ZeroDivisionError** (l. 10)
- **NameError** (l. 10)

R. Monat, A. Ouadjaout, and A. Miné. “Value and allocation sensitivity in static Python analyses”. SOAP@PLDI 2020
Comparison of the type and value analyses

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 = []
for i in range(randint(5, 10)):
    l.append(Task(randint(0, 20)))

m = average(l)
```

Type analysis

- **ValueError** (l. 3)
- **IndexError** (l. 9)
- **ZeroDivisionError** (l. 10)
- **NameError** (l. 10)

Non-relational value analysis

**IndexError** (l. 9)

R. Monat, A. Oualjaout, and A. Miné. “Value and allocation sensitivity in static Python analyses”. SOAP@PLDI 2020
Comparison of the type and value analyses

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 = []
for i in range(randint(5, 10)):
    l.append(Task(randint(0, 20)))
```

Type analysis

- ValueError (l. 3)
- IndexError (l. 9)
- ZeroDivisionError (l. 10)
- NameError (l. 10)

Non-relational value analysis

IndexError (l. 9)

Relational value analysis

No alarm!

R. Monat, A. Ouadjaout, and A. Miné. “Value and allocation sensitivity in static Python analyses”. SOAP@PLDI 2020
Comparison of the type and value analyses (II)

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Type Analysis</th>
<th>Non-relational Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>1.5s</td>
<td>3MB</td>
</tr>
<tr>
<td>scimark.py</td>
<td>416</td>
<td>1.4s</td>
<td>12MB</td>
</tr>
<tr>
<td>richards.py</td>
<td>426</td>
<td>13s</td>
<td>112MB</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>8.3s</td>
<td>7MB</td>
</tr>
<tr>
<td>go.py</td>
<td>461</td>
<td>27s</td>
<td>345MB</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>1.1m</td>
<td>525MB</td>
</tr>
<tr>
<td>regex_v8.py</td>
<td>1792</td>
<td>23s</td>
<td>18MB</td>
</tr>
<tr>
<td>processInput.py</td>
<td>1417</td>
<td>10s</td>
<td>64MB</td>
</tr>
<tr>
<td>choose.py</td>
<td>2562</td>
<td>1.1m</td>
<td>1.6GB</td>
</tr>
<tr>
<td>Total</td>
<td>9294</td>
<td>4.0m</td>
<td>2.8GB</td>
</tr>
</tbody>
</table>
### Comparison of the type and value analyses (II)

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Type Index Key</th>
<th>Type Analysis</th>
<th>Non-relational Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time</td>
<td>Mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td></td>
<td>1.5s</td>
<td>3MB</td>
</tr>
<tr>
<td>scimark.py</td>
<td>416</td>
<td></td>
<td>1.4s</td>
<td>12MB</td>
</tr>
<tr>
<td>richards.py</td>
<td>426</td>
<td></td>
<td>13s</td>
<td>112MB</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td></td>
<td>8.3s</td>
<td>7MB</td>
</tr>
<tr>
<td>go.py</td>
<td>461</td>
<td></td>
<td>27s</td>
<td>345MB</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td></td>
<td>1.1m</td>
<td>525MB</td>
</tr>
<tr>
<td>regex_v8.py</td>
<td>1792</td>
<td></td>
<td>23s</td>
<td>18MB</td>
</tr>
<tr>
<td>processInput.py</td>
<td>1417</td>
<td></td>
<td>10s</td>
<td>64MB</td>
</tr>
<tr>
<td>choose.py</td>
<td>2562</td>
<td></td>
<td>1.1m</td>
<td>1.6GB</td>
</tr>
<tr>
<td>Total</td>
<td>9294</td>
<td></td>
<td>4.0m</td>
<td>2.8GB</td>
</tr>
</tbody>
</table>

## Conclusion

The non-relational value analysis
- does not remove false type alarms
- significantly reduces index errors
- is $\approx 3 \times$ costlier
Comparison of the type and value analyses (II)

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Type Analysis</th>
<th>Non-relational Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Type</td>
<td>Index</td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>1.5s</td>
<td>3MB</td>
</tr>
<tr>
<td>scimark.py</td>
<td>416</td>
<td>1.4s</td>
<td>12MB</td>
</tr>
<tr>
<td>richards.py</td>
<td>426</td>
<td>13s</td>
<td>112MB</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>8.3s</td>
<td>7MB</td>
</tr>
<tr>
<td>go.py</td>
<td>461</td>
<td>27s</td>
<td>345MB</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>1.1m</td>
<td>525MB</td>
</tr>
<tr>
<td>regex_v8.py</td>
<td>1792</td>
<td>23s</td>
<td>18MB</td>
</tr>
<tr>
<td>processInput.py</td>
<td>1417</td>
<td>10s</td>
<td>64MB</td>
</tr>
<tr>
<td>choose.py</td>
<td>2562</td>
<td>1.1m</td>
<td>1.6GB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9294</strong></td>
<td><strong>4.0m</strong></td>
<td><strong>2.8GB</strong></td>
</tr>
</tbody>
</table>

**Conclusion**

The non-relational value analysis
- does not remove false type alarms
- significantly reduces index errors
- is $\approx 3 \times$ costlier

**Heuristic packing and relational analyses**
- Static packing, using function’s scope
- Rules out all 145 alarms of `regex_v8.py` (1792 LOC) at $2.5 \times$ cost
Analyzing Python Programs with C Libraries
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 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 runtime-errors (exceptions in Python)
Combining C and Python – motivation

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

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Pitfalls

- Different values (arbitrary-precision integers in Python, bounded in C)
- Different runtime-errors (exceptions in Python)
- Garbage collection
### Combining C and Python – example

<table>
<thead>
<tr>
<th>counter.c</th>
<th>count.py</th>
</tr>
</thead>
</table>
| **typedef struct {**<br> | **from counter import Counter**<br>  
|     PyObject_HEAD;<br>     int count;<<br> } Counter; | **from random import randrange**<br>  
| **static PyObject**<br> CounterIncr(Counter *self, PyObject *args)<br> {<br>     int i = 1;<br>     if(!PyArg_ParseTuple(args, "|i", &i))<br>         return NULL;<br>     self->count += i;<br>     Py_RETURN_NONE;<br> }<br> **static PyObject**<br> CounterGet(Counter *self)<br> {<br>     return Py_BuildValue("i", self->count);<br> }<br> | **c = Counter()**<br>  
| **power = randrange(128)**<br>  
| **c.incr(2**<sup>**power-1**</sup>)<br>  
| **c.incr()**<br>  
| **r = c.get()**<br>  
| ▶ power ≤ 30 ⇒ r = 2<sup>power</sup><br>  
| ▶ power = 31 ⇒ r = −2<sup>31</sup><br>  
| ▶ 32 ≤ power ≤ 64: OverflowError: signed integer is greater than maximum<br>  
| ▶ power ≥ 64: OverflowError: Python int too large to convert to C long<br>  |
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()
```

▶ $\text{power} \leq 30 \Rightarrow r = 2^{\text{power}}$

$\text{power} = 31 \Rightarrow r = -2^{31}$

$32 \leq \text{power} \leq 64: \text{OverflowError: signed integer is greater than maximum}$

$\text{power} \geq 64: \text{OverflowError: Python int too large to convert to C long}$
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()
```

- **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()
```

- **power** ≤ 30 ⇒ \( r = 2^{\text{power}} \)
- **power** = 31 ⇒ \( r = -2^{31} \)
- 32 ≤ **power** ≤ 64: OverflowError: signed integer is greater than maximum
- **power** ≥ 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: ...

No raised exceptions
⇒ missed errors

Typeshed: type annotations for the standard library, used in the single-language analysis before

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 $\implies$ missed errors
How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Counter:</td>
</tr>
<tr>
<td>def <strong>init</strong>(self): ...</td>
</tr>
<tr>
<td>def incr(self, i: int = 1): ...</td>
</tr>
<tr>
<td>def get(self) -&gt; int: ...</td>
</tr>
</tbody>
</table>

- 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 $\Rightarrow$ 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 the single-language analysis before
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?

<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>
</tbody>
</table>

- 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?

<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>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>
</tbody>
</table>
### How to analyze multilanguage programs?

<table>
<thead>
<tr>
<th>Type annotations</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>
<tr>
<td>- Not sound: some effects are not taken into account</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

### 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?

<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>
</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

## 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?

<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>
<tr>
<td>▶ Not automatic: manual conversion</td>
</tr>
<tr>
<td>▶ Not sound: some effects are not taken into account</td>
</tr>
<tr>
<td><strong>Our approach</strong></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>
<tr>
<td>▶ Reuse previous analyses of C and Python</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

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
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);
}

from counter import Counter
from random import randrange

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

<table>
<thead>
<tr>
<th>counter.c</th>
<th>count.py</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>typedef struct {</code></td>
<td><code>from counter import Counter</code></td>
</tr>
<tr>
<td><code>   PyObject_HEAD;</code></td>
<td><code>from random import randrange</code></td>
</tr>
<tr>
<td><code>   int count;</code></td>
<td><code>Counter</code></td>
</tr>
<tr>
<td><code>} Counter;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>from counter import Counter</code></td>
</tr>
<tr>
<td>static PyObject* CounterIncr(Counter *self, `</td>
<td><code>CounterIncr(Counter *self, </code></td>
</tr>
<tr>
<td>{ `</td>
<td>`{</td>
</tr>
<tr>
<td>int i = 1; `</td>
<td>`int i = 1;</td>
</tr>
<tr>
<td>if(!PyArg_ParseTuple(args, `</td>
<td>`if(!PyArg_ParseTuple(args,</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot; , &amp;i))`</td>
</tr>
<tr>
<td>return NULL;`</td>
<td><code>return NULL;</code></td>
</tr>
<tr>
<td>self-&gt;count += i;`</td>
<td><code>self-&gt;count += i;</code></td>
</tr>
<tr>
<td>}`</td>
<td><code>Py_RETURN_NONE;</code></td>
</tr>
<tr>
<td>static PyObject* CounterGet(Counter *self) `</td>
<td><code>CounterGet(Counter *self) </code></td>
</tr>
<tr>
<td>{ `</td>
<td>`{</td>
</tr>
<tr>
<td>return Py_BuildValue(&quot;i&quot;, `</td>
<td><code>return Py_BuildValue(&quot;i&quot;, </code></td>
</tr>
<tr>
<td>self-&gt;count);`</td>
<td><code>self-&gt;count);</code></td>
</tr>
<tr>
<td>}`</td>
<td>`}</td>
</tr>
</tbody>
</table>

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

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

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 ./counter.c:17.6-38:convert_single[0]: PyParseTuple_int
  from ./counter.c:17.6-38:convert_single[0]: PyParseTuple_int
  from ./counter.c:17.6-38:convert_single[0]: PyParseTuple_int
+1 other callstack
Difficulty: shared memory

- Each language may change the memory state, and has a different view of it.
- Synchronization? We could perform a full state translation, but:
  - the cost would be high in the analysis
  - some abstractions can be shared between Python and C.
High-level idea

**Difficulty: shared memory**
- Each language may change the memory state, and has a different view of it
- Synchronization? We could perform a full state translation, but
  - the cost would be high in the analysis
  - some abstractions can be shared between Python and C

**State separation → reduced synchronization**
- Observation: structures are directly dereferenceable by one language only
- Switch to other language otherwise ($c$.incr() → self->count += 1)
  Additional hypothesis: C accesses to Python objects through the API
- Synchronization: only when objects change language for the first time
- Mopsa supports shared abstractions
Concrete definition

- Builds upon the Python and C semantics
Multilanguage semantics

Concrete definition

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

### Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- **Boundary functions** handling the reduced synchronization
Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Boundary functions handling the reduced synchronization

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
- **Boundary functions** handling the reduced synchronization

### Limitations

- Garbage collection not handled
### Multilanguage semantics

#### Concrete definition
- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- **Boundary functions** handling the reduced synchronization

#### 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
- **Boundary functions** handling the reduced synchronization

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

Py:program

U.intraproc

Py.desugar

U.loops

Py.flow

U.interproc

Py.libraries

Py.data_model

Py.objects

×

Py.environment

Py.attributes

◦

×

Py.lists

Py.tuples

Py.dicts

U.recency

×

U.intervals

U.strings

Sequence

Reduced product

Cartesian product

Composition

C:program

C.desugar

C.goto

U.intraproc

U.loops

U.interproc

C.stubs

C.libraries

C.files

∧

C.machineNum

C.pointers

U.recency

∧

U.intervals

U.linearRel

Implementation LOC

Part LOC

Framework 13200

Universal 5600

C 11700

Python 12600

Multilanguage 2500
From distinct Python and C analyses... to a multilanguage analysis!
From distinct Python and C analyses... 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.

<table>
<thead>
<tr>
<th>Library</th>
<th>C + Py. Loc</th>
<th>Tests</th>
<th>1/test</th>
<th># proved checks</th>
<th>%</th>
<th># checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>noise</td>
<td>1397</td>
<td>15/15</td>
<td>1.2s</td>
<td>99.7%</td>
<td>(6690)</td>
<td></td>
</tr>
<tr>
<td>cdistance</td>
<td>2345</td>
<td>28/28</td>
<td>4.1s</td>
<td>98.0%</td>
<td>(13716)</td>
<td></td>
</tr>
<tr>
<td>llist</td>
<td>4515</td>
<td>167/194</td>
<td>1.5s</td>
<td>98.8%</td>
<td>(36255)</td>
<td></td>
</tr>
<tr>
<td>ahocorasicick</td>
<td>4877</td>
<td>46/92</td>
<td>1.2s</td>
<td>96.7%</td>
<td>(6722)</td>
<td></td>
</tr>
<tr>
<td>levenshtein</td>
<td>5798</td>
<td>17/17</td>
<td>5.3s</td>
<td>84.6%</td>
<td>(4825)</td>
<td></td>
</tr>
<tr>
<td>bitarray</td>
<td>5841</td>
<td>159/216</td>
<td>1.6s</td>
<td>94.9%</td>
<td>(25566)</td>
<td></td>
</tr>
</tbody>
</table>
Contributions around the static analysis of Python programs

### Python’s semantics

- Reverse-engineering CPython (160kLoc C)
- Backlinks to source code (auditability)
- On-paper formalization (∼ 44 pages)
Contributions around the static analysis of Python programs

**Python’s semantics**

- Reverse-engineering CPython (160kLoc C)
- Backlinks to source code (auditability)
- On-paper formalization (∼ 44 pages)

**Python’s analyses**

- Type and value analyses
- Combining numerous abstractions
- Analysis of real programs
Contributions around the static analysis of Python programs

**Python’s semantics** ▶️ $\text{\pygame} \rightsquigarrow [\cdot]_{\text{py}}$
- Reverse-engineering CPython (160kLoc C)
- Backlinks to source code (auditability)
- On-paper formalization ($\approx$ 44 pages)

**Python’s analyses** $[\cdot]_{\text{py}} \rightsquigarrow [\cdot]_{\text{py}}$
- Type and value analyses
- Combining numerous abstractions
- Analysis of real programs

**Multilanguage, Python/C analysis**
- First real multilanguage analysis
- Reuses off-the-shelf Python and C analyses
- Analysis of real-world libraries
### Contributions around the static analysis of Python programs

<table>
<thead>
<tr>
<th>Python’s semantics</th>
<th>Python’s analyses</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Reverse-engineering CPython (160kLoc C)</td>
<td>▶ Type and value analyses</td>
</tr>
<tr>
<td>▶ Backlinks to source code (auditability)</td>
<td>▶ Combining numerous abstractions</td>
</tr>
<tr>
<td>▶ On-paper formalization (∼ 44 pages)</td>
<td>▶ Analysis of real programs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multilanguage, Python/C analysis</th>
<th>Implementation into Mopsa</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ First real multilanguage analysis</td>
<td>▶ Open-source analyzer for C and Python</td>
</tr>
<tr>
<td>▶ Reuses off-the-shelf Python and C analyses</td>
<td>▶ Factors abstractions between languages</td>
</tr>
<tr>
<td>▶ Analysis of real-world libraries</td>
<td>▶ Sharing between abstractions</td>
</tr>
</tbody>
</table>
A Modern Compiler for the French Tax Code
Research field: formal methods
⇒ Improve confidence in software.
### Research field: formal methods

⇒ Improve confidence in software.

### Personal methodology

Constant back and forth between theory and practice

1. Find interesting bugs, properties or systems to study (GitHub, ...)
2. Theoretical study and solution
3. Implementation and experimental validation (on 1)
### French income tax

- 38M households, 75Md€ of income
- Made public in April 2016: ≃ 92kLoc M, custom language

⚠ Computation not reproducible in 2019
## Legal implementations

### French income tax

- 38M households, 75Md€ of income
- Made public in April 2016: \( \approx 92 \text{kLoc M, custom language} \)

⚠ Computation not reproducible in 2019

### Trusting the computation?

- Reproducibility of the computation?
- Accurate simulation of tax reforms?
- Compliance with the law, acting as specification?
### Variable declaration

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRNETBIS</td>
<td>calculee primrest = 0 : &quot;IRNET avant bidouille du 8ZI&quot; ;</td>
</tr>
<tr>
<td>8ZI</td>
<td>&quot;Impot net apres depart a l'etranger (non residents)&quot; ;</td>
</tr>
</tbody>
</table>
Variable declaration

IRNETBIS : calculee primrest = 0 : "IRNET avant bidouille du 8ZI" ;
8ZI : "Impot net apres depart a l'etranger (non residents)" ;

Computation rule

rule 221220:
application : iliad ;
IRNETBIS = max(0, IRNETTER -
  PIR * positif(SEUIL_12 - IRNETTER + PIR)
  * positif(SEUIL_12 - PIR)
  * positif_ou_nul(IRNETTER - SEUIL_12));
The core of $M$: arithmetic expressions assigned to variables.
The core of M: **arithmetic expressions** assigned to variables.

**M quirks**

- Static-size arrays (size defined at declaration)
M, briefly

The core of M: **arithmetic expressions** assigned to variables.

**M quirks**

- Static-size arrays (size defined at declaration)
- Small, unrollable loops
The core of M: **arithmetic expressions** assigned to variables.

**M quirks**

- Static-size arrays (size defined at declaration)
- Small, unrollable loops
- Use of floating-point numbers, booleans are zero and one
The core of M: arithmetic expressions assigned to variables.

M quirks

- Static-size arrays (size defined at declaration)
- Small, unrollable loops
- Use of floating-point numbers, booleans are zero and one
- `undef` value
A formal semantics for M

We reverse-engineered the semantics:

► At first, using the online simulator²
► Later, using the private tests DGFiP sent us (August 7, 2019)

A formal semantics for M

We reverse-engineered the semantics:

- At first, using the online simulator\(^2\)
- Later, using the private tests DGFiP sent us (August 7, 2019)

⇒ a $\mu$M kernel, its semantics formalized in the Coq proof assistant.

\(^2\)https://www3.impots.gouv.fr/simulateur/calcul_impot/2020/index.htm
A formal semantics for M

We reverse-engineered the semantics:

- At first, using the online simulator\(^2\)
- Later, using the private tests DGFiP sent us (August 7, 2019)

⇒ a \(\mu M\) kernel, its semantics formalized in the Coq proof assistant.

The **undef** value

- Used for: default inputs, runtime errors & missing cases in inline conditionals
- Fun facts: \(f + \text{undef} = f, f \div 0 = 0, x[|x| + 1] = \text{undef}, x[-1] = 0\...\)

\(^2\)https://www3.impots.gouv.fr/simulateur/calcul_impot/2020/index.htm
DGFiP’s legacy architecture

After 9 months of negotiations, we’re in!

"rules" M files → "rules" C files → Shared state → "calculette" Shared library → “inter” C files

DGFiP’s internal compiler

GCC

35kLoc of C to bypass M’s lack of functions.

Security concerns ⇝ no publication

How to extract the logic of the code?

DSLs to the rescue! Introducing M++

▶ High-level, no mutable state under the hood
▶ Tailored for the needs of the “inter” files and DGFiP devs

6,000 lines of “inter” C code ⇒ 100 lines of M++
After 9 months of negotiations, we’re in!

DGFiP’s legacy architecture

“inter” files
35kLoc of C to bypass M’s lack of functions.

DGFiP’s internal compiler

 GCC
DGFiP’s legacy architecture

After 9 months of negotiations, we’re in!

“rules” M files

“rules” C files

Shared state

“calculette” Shared library

“inter” C files

DGFiP’s internal compiler

GCC

“inter” files

35kLoc of C to bypass M’s lack of functions.

Security concerns ⇝ no publication

How to extract the logic of the code?

DSLs to the rescue! Introducing M ++

▶ High-level, no mutable state under the hood

▶ Tailored for the needs of the “inter” files and DGFiP devs

6,000 lines of “inter” C code
⇒ 100 lines of M ++
DGFiP’s legacy architecture

After 9 months of negotiations, we’re in!

“rules” files
“rules” C files

Shared state
“inter” C files
“inter” files

35kLoc of C to bypass M’s lack of functions.

Security concerns ⇝ no publication
How to extract the logic of the code?

“inter” files

“calculette” Shared library

DGFiP’s internal compiler

GCC

DSLs to the rescue! Introducing M++

- High-level, no mutable state under the hood
- Tailored for the needs of the “inter” files and DGFiP devs
- 6,000 lines of “inter” C code ⇒ 100 lines of M++
MLANG’s architecture

MLANG: written in OCaml, 10k lines of code
https://github.com/MLanguage/mlang
MLANG: written in OCaml, 10k lines of code
https://github.com/MLanguage/mlang
MLANG’s correctness

How to check that MLANG is correct?

- 476 tests from DGFiP
- Generation of our own tests
- Quality measure: value coverage

It works (precise down to the euro)!
All backends validated, on all tests

Number of distinct values assigned

<table>
<thead>
<tr>
<th>Percentage of assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
</tr>
<tr>
<td>20%</td>
</tr>
<tr>
<td>40%</td>
</tr>
<tr>
<td>60%</td>
</tr>
<tr>
<td>80%</td>
</tr>
</tbody>
</table>

DGFiP Private (476 tests)
Randomized (1267 tests)
Fuzzer-generated (275 tests)
How to check that MLANG is correct?

- 476 tests from DGFiP
- Generation of our own tests
- Quality measure: value coverage

It works (precise down to the euro)!
All backends validated, on all tests
MLANG’s correctness

How to check that MLANG is correct?

- 476 tests from DGFiP
- Generation of our own tests
- Quality measure: value coverage

It works (precise down to the euro)!
All backends validated, on all tests

![Graph showing percentage of assignments](image-url)
Code optimization

Compiler optimizations

- Global value numbering
- Dead code elimination
- Partial evaluation
- Dataflow defined-ness analysis

<table>
<thead>
<tr>
<th>Spec. name</th>
<th># inputs</th>
<th># outputs</th>
<th># instructions</th>
<th>% reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>2,459</td>
<td>10,411</td>
<td>129,683</td>
<td>80.2%</td>
</tr>
<tr>
<td>Selected outs</td>
<td>2,459</td>
<td>11</td>
<td>99,922</td>
<td>84.8%</td>
</tr>
<tr>
<td>Tests</td>
<td>1,635</td>
<td>646</td>
<td>111,839</td>
<td>83.0%</td>
</tr>
<tr>
<td>Simplified</td>
<td>228</td>
<td>11</td>
<td>4,172</td>
<td>99.4%</td>
</tr>
<tr>
<td>Basic</td>
<td>3</td>
<td>1</td>
<td>553</td>
<td>99.9%</td>
</tr>
</tbody>
</table>
## Code optimization

### Compiler optimizations

- Global value numbering
- Dead code elimination
- Partial evaluation
- Dataflow defined-ness analysis

<table>
<thead>
<tr>
<th>Spec. name</th>
<th># inputs</th>
<th># outputs</th>
<th># instructions</th>
<th>% reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>2,459</td>
<td>10,411</td>
<td>129,683</td>
<td>80.2%</td>
</tr>
<tr>
<td>Selected outs</td>
<td>2,459</td>
<td>11</td>
<td>99,922</td>
<td>84.8%</td>
</tr>
<tr>
<td>Tests</td>
<td>1,635</td>
<td>646</td>
<td>111,839</td>
<td>83.0%</td>
</tr>
<tr>
<td>Simplified</td>
<td>228</td>
<td>11</td>
<td>4,172</td>
<td>99.4%</td>
</tr>
<tr>
<td>Basic</td>
<td>3</td>
<td>1</td>
<td>553</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

# of instructions with optimizations disabled (2018 code): 656,020.
## Code optimization

### Compiler optimizations

- Global value numbering
- Dead code elimination
- Partial evaluation
- Dataflow defined-ness analysis

<table>
<thead>
<tr>
<th>Spec. name</th>
<th># inputs</th>
<th># outputs</th>
<th># instructions</th>
<th>% reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>2,459</td>
<td>10,411</td>
<td>129,683</td>
<td>80.2%</td>
</tr>
<tr>
<td>Selected outs</td>
<td>2,459</td>
<td>11</td>
<td>99,922</td>
<td>84.8%</td>
</tr>
<tr>
<td>Tests</td>
<td>1,635</td>
<td>646</td>
<td>111,839</td>
<td>83.0%</td>
</tr>
<tr>
<td>Simplified</td>
<td>228</td>
<td>11</td>
<td>4,172</td>
<td>99.4%</td>
</tr>
<tr>
<td>Basic</td>
<td>3</td>
<td>1</td>
<td>553</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

# of instructions with optimizations disabled (2018 code): 656,020.
## Contributions around the French tax code

<table>
<thead>
<tr>
<th>Component</th>
<th>Published by DGFiP?</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>yes</td>
<td>Reverse-engineering $M \rightsquigarrow \mu M$ in Coq</td>
</tr>
<tr>
<td>“inter” code (C)</td>
<td>no</td>
<td>DSL $M^{++}$ (6kLoc C $\rightsquigarrow$ 100 M++)</td>
</tr>
<tr>
<td>M compiler</td>
<td>no</td>
<td>MLANG (10kLoc OCaml) with optimizations</td>
</tr>
<tr>
<td>Tests</td>
<td>no</td>
<td>Fuzzing-generated tests (better coverage)</td>
</tr>
</tbody>
</table>

$\Rightarrow$ Now reproducible outside DGFiP!
Contributions around the French tax code

<table>
<thead>
<tr>
<th>Component</th>
<th>Published by DGFiP?</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>yes</td>
<td>Reverse-engineering $M \rightsquigarrow \mu M$ in Coq</td>
</tr>
<tr>
<td>“inter” code (C)</td>
<td>no</td>
<td>$\text{DSL } M^{++}$ ($6k\text{Loc C} \rightsquigarrow 100 \text{ M}^{++}$)</td>
</tr>
<tr>
<td>M compiler</td>
<td>no</td>
<td>$\text{MLang (10kLoc OCaml) with optimizations}$</td>
</tr>
<tr>
<td>Tests</td>
<td>no</td>
<td>Fuzzing-generated tests (better coverage)</td>
</tr>
</tbody>
</table>

$\Rightarrow$ Now reproducible outside DGFiP!

# Contributions around the French tax code

<table>
<thead>
<tr>
<th>Component</th>
<th>Published by DGFiP?</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>yes</td>
<td>Reverse-engineering $M \rightsquigarrow \mu M$ in Coq</td>
</tr>
<tr>
<td>“inter” code (C)</td>
<td>no</td>
<td><strong>DSL</strong> $M++$ (6kLoc C $\rightsquigarrow$ 100 $M++$)</td>
</tr>
<tr>
<td>M compiler</td>
<td>no</td>
<td><strong>MLANG</strong> (10kLoc OCaml) with optimizations</td>
</tr>
<tr>
<td>Tests</td>
<td>no</td>
<td>Fuzzing-generated tests (better coverage)</td>
</tr>
</tbody>
</table>

$\implies$ Now reproducible outside DGFiP!

## Interacting with DGFiP

- Long term work: 9 months to access the missing C code
- Pedagogy required: very legal environment

Contributions around the French tax code

<table>
<thead>
<tr>
<th>Component</th>
<th>Published by DGFiP?</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>yes</td>
<td>Reverse-engineering $M \leadsto \mu M$ in Coq</td>
</tr>
<tr>
<td>&quot;inter&quot; code (C)</td>
<td>no</td>
<td>DSL $M^{++}$ (10kLoc OCaml) with optimizations</td>
</tr>
<tr>
<td>$M$ compiler</td>
<td>no</td>
<td>Fuzzing-generated tests (better coverage)</td>
</tr>
</tbody>
</table>

- Ongoing work to bring $MLANG$ at DGFiP.
  - 30-day mission between January and August 2022
  - Supervision of 3 developers (2 OCamlPro, 1 DGFiP)

Interacting with DGFiP:
- Long term work: 9 months to access the missing C code
- Pedagogy required: very legal environment

Future DSLs

Does the implementation comply with the law?

Law (specification)  \(\xrightarrow{GF-1A}\)  Technical description  \(\xrightarrow{BSI4}\)  Implementation

Future DSLs

Does the implementation comply with the law?

- No structural correspondance
- 2019～2020 : 30% of 90kLoc M changed
Does the implementation comply with the law?

Catala, another DSL to the rescue

Article D521-1 du code de la sécurité sociale

I - Pour l’application de l’article L. 521-1, le montant des allocations familiales et de la majoration pour âge prévue à l’article L. 521-3 est défini selon le barème suivant :

1° Lorsque le ménage ou la personne a disposé d’un montant de ressources inférieur ou égal au plafond défini au I de l’article D. 521-3, les taux servant au calcul des allocations familiales sont fixés, en pourcentage de la base mensuelle prévue à l’article L. 551-1, à :
a) 32 % pour le deuxième enfant à charge ;

```catala
champ d'application AllocationsFamiliales :
  définition montant_initial_base_deuxième_enfant sous condition
  ressources_ménage ≤ € plafond_I_d521_3
  conséquence égal à
  si nombre de enfants_à_charge_droit_ouvert_prestation_familiale ≥ 2
  alors prestations_familiales.base_mensuelle ×€ 32 %
  sinon 0 €
```
Future DSLs

Does the implementation comply with the law?

Catala, another DSL to the rescue

Article D521-1 du code de la sécurité sociale

1° Lorsque le ménage ou la personne a disposé d’un montant de ressources inférieur ou égal au plafond défini au 1° de l’article D. 521-3, les taux servant au calcul des allocations familiales sont fixés, en pourcentage de la base mensuelle prévue à l’article L. 551-1, à :

a) 32 % pour le deuxième enfant à charge ;

```catala
champ d'application AllocationsFamiliales :
  définition montant_initial_base_deuxième_enfant sous condition
    ressources_ménage ≤€ plafond_I_d521_3
  conséquence égal à
    si nombre de enfants_à_charge_droit_ouvert_prestation_familiale ≥ 2
    alors prestations_familiales.base_mensuelle ×€ 32 %
    sinon 0 €
```

Conclusion
Summary of the contributions

<table>
<thead>
<tr>
<th>Static analysis of Python programs using C libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Ouadjaout and Miné (LIP6, Sorbonne Université)</td>
</tr>
<tr>
<td>▶ SAS’21, ECOOP’20, VSTTE’19 (invited), SOAP@PLDI’20 (award), JFLA’21 (fr, tool)</td>
</tr>
<tr>
<td>▶ Mopsa (LGPL v3, 60kLoc OCaml), main contributor since September 2018</td>
</tr>
</tbody>
</table>
### Summary of the contributions

#### Static analysis of Python programs using C libraries

- Ouadjaout and Miné (LIP6, Sorbonne Université)
- SAS’21, ECOOP’20, VSTTE’19 (invited), SOAP@PLDI’20 (award), JFLA’21 (fr, tool)
- Mopsa (LGPL v3, 60kLoc OCaml), main contributor since September 2018

#### A modern compiler for the French tax code

- Merigoux (Prosecco, Inria Paris) et Protzenko (Microsoft Research)
- CC’21, JFLA’20 (fr), JFLA’21 (fr, tool)
- Mlang (GPL v3, 10kLoc OCaml), main contributor since May 2019
- Ongoing transfer work
  - 30 days mission between January and August 2022
  - Supervision of 3 developers (2 OCamlPro, 1 DGFiP)
Looking forward... an important question
Looking forward... an important question

What kind of research for tomorrow’s world?
What kind of research for tomorrow’s world?

⚠️ Avoiding rebound effect
What kind of research for tomorrow’s world?

⚠ Avoiding rebound effect
❓ Impact?
What kind of research for tomorrow’s world?

⚠ Avoiding rebound effect
❓ Impact? Difficult to estimate a priori
From Python to the French Tax Code: Applying Formal Methods on Real Systems

Discussion

Raphaël Monat
SyCoMoRES team
rmonat.fr

DI Seminar
22 March 2023
From Python to the French Tax Code:
Applying Formal Methods on Real Systems

Discussion

Interested in internships or PhDs?
Come have a chat or email me!
raphael.monat@inria.fr

Raphaël Monat
SyCoMoRES team
rmonat.fr

DI Seminar
22 March 2023