Semantics and Static Analysis of Python

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

2nd July 2019
Introduction
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST "Hello, world!"

I DUNNO...
DYNAMIC TYPING?
WHITESPACE?

COME JOIN US!
PROGRAMMING IS FUN AGAIN!
IT'S A WHOLE NEW WORLD UP HERE!

BUT HOW ARE YOU FLYING?

I JUST TYPED
import antigravity

THAT'S IT?

...I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.

BUT I THINK THIS IS THE PYTHON.
Python is a Dynamic Programming Language

It features:

▶ A concise and efficient syntax,
▶ Dynamic typing: types are only known at runtime,
▶ Introspection,
▶ Self-modification.
def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        res = p.__fspath__()
        if isinstance(res, (str, bytes)):
            return res
        else:
            raise TypeError("...")
    else:
        raise TypeError("...")
def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        res = p.__fspath__()
        if isinstance(res, (str, bytes)):
            return res
        else:
            raise TypeError("...")
    else:
        raise TypeError("...")
def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        res = p.__fspath__()
        if isinstance(res, (str, bytes)):
            return res
        else:
            raise TypeError("...")
    else:
        raise TypeError("...")

Two notions of typing:

- Nominal, based on classes.
- Structural, based on attributes.
def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        res = p.__fspath__()
        if isinstance(res, (str, bytes)):
            return res
        else:
            raise TypeError("...")
    else:
        raise TypeError("...")

Two notions of typing:

► Nominal, based on classes.
► Structural, based on attributes.

Type of fspath?
def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        res = p.__fspath__()
        if isinstance(res, (str, bytes)):
            return res
        else:
            raise TypeError("...")
    else:
        raise TypeError("...")

Two notions of typing:

- Nominal, based on classes.
- Structural, based on attributes.

Type of fspath?

α → α, α ∈ \{str, bytes\} or an object having a method

__fspath__ returning α.
Bugs are Everywhere!
Motivation

- Detect all runtime errors,
- Without executing the programs.
Motivation

- Detect all runtime errors,
- Without executing the programs.

A theoretical hurdle: Rice’s theorem

“any non-trivial semantic property of programs is undecidable”
Static Analysis by Abstract Interpretation

Motivation

▶ Detect all runtime errors,
▶ Without executing the programs.

A theoretical hurdle: Rice’s theorem

“any non-trivial semantic property of programs is undecidable”

⇒ we will compute approximate results
Static Analysis by Abstract Interpretation

Motivation

- Detect all runtime errors,
- Without executing the programs.

A theoretical hurdle: Rice’s theorem

“any non-trivial semantic property of programs is undecidable”

$\implies$ we will compute approximate results

$\implies$ our approximate, pessimistic approach may yield false alarms.
Motivation

- Detect all runtime errors,
- Without executing the programs.

A theoretical hurdle: Rice’s theorem

“any non-trivial semantic property of programs is undecidable”

⇒ we will compute approximate results
⇒ our approximate, pessimistic approach may yield false alarms.

Goals

- Automatic analysis: no expert knowledge required.
- Sound analysis: if no bug is detected, none will occur.
Static analyses successfully work on critical embedded C software. Contrary to C, Python leaves less information in the syntax.
Static analyses successfully work on critical embedded C software. Contrary to C, Python leaves less information in the syntax.

Static analyses are especially helpful – though difficult – on dynamic programming languages.
Static analyses successfully work on critical embedded C software. Contrary to C, Python leaves less information in the syntax.

Static analyses are especially helpful – though difficult – on dynamic programming languages.

We present a static type analysis for Python...
Static analyses successfully work on critical embedded C software. Contrary to C, Python leaves less information in the syntax.

Static analyses are especially helpful – though difficult – on dynamic programming languages.

We present a static type analysis for Python… but first let’s take a look at Python’s semantics.
Semantics of Python
Semantics?  
A **mathematical** description of the behavior of Python operators.

Why?  
To relate static analyses with the actual program behavior, and prove that our static analyses are correct.
An Example: \( e_1 + e_2 \)

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

1. \( \text{has\_field}(a_1, \_\_add\_\_)? \)
   - **Yes**
     - \( \text{has\_field}(a_2, \_\_radd\_\_) \)
     - **Yes**
       - \( a_3 = \text{call } a_1 \_\_add\_\_ \) on \( a_1, a_2 \)
     - **No**
       - \( a_3 == \text{NotImplemented} \)
   - **No**
     - \( \text{has\_field}(a_2, \_\_radd\_\_) \)
     - **No**
       - \( \text{has\_field}(a_1, \_\_add\_\_) \)
       - **No**
         - \( \text{has\_field}(a_2, \_\_radd\_\_) \)
         - **Yes**
           - \( a_3 = \text{call } a_2 \_\_radd\_\_ \) on \( a_1, a_2 \)
         - **No**
           - \( a_3 == \text{NotImplemented} \)
       - **Yes**
         - \( \text{type}(a_1) \neq \text{type}(a_2) \)
         - **Yes**
           - Type Error
         - **No**
           - \( a_3 == \text{NotImplemented} \)
   - **No**
     - \( \text{type}(a_1) \neq \text{type}(a_2) \)
     - **Yes**
       - Type Error
     - **No**
       - \( a_3 == \text{NotImplemented} \)

Result is \( a_3 \)
Uncovering the semantics

- No standard, CPython is the reference interpreter.
- Done by reading the documentation and CPython’s source.
Uncovering the semantics

▶ No standard, CPython is the reference interpreter.
▶ Done by reading the documentation and CPython’s source.

Checking the semantics is correct

We are currently using CPython’s tests...
Uncovering the semantics

- No standard, CPython is the reference interpreter.
- Done by reading the documentation and CPython’s source.

Checking the semantics is correct

We are currently using CPython’s tests...

Other approaches

- Coq: extractable interpreter, proofs.
- K framework: interpreter, semantics coverage tests, deductive verification.
Uncovering the semantics

- No standard, CPython is the reference interpreter.
- Done by reading the documentation and CPython’s source.

Checking the semantics is correct
We are currently using CPython’s tests...

Other approaches

- Coq: extractable interpreter, proofs.
- K framework: interpreter, semantics coverage tests, deductive verification.

Both are time-consuming...
Static Type Analysis
def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        res = p.__fspath__()
        if isinstance(res, (str, bytes)):
            return res
        else:
            raise TypeError("...")
    else:
        raise TypeError("...")

if *:
    i = 'a'
elif *:
    i = b'path'
else:
    i = FSPPath()
r = fspath(i)

i : str or bytes or FSPath.

class FSPath:
    def __fspath__(self):
        return 42
Static Type Analysis Example

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

class FSPath:
    def __fspath__(self):
        return 42

if *:
    i = 'a'
elif *:
    i = b'path'
else:
    i = FSPath()
r = fspath(i)

$i$ : str or bytes or FSPath.
$r$ : str or bytes,
or a TypeError is raised.
Features of the Analysis

Our analysis:

- Detects uncaught exceptions (TypeError, AttributeError),
- Is flow-sensitive,
- Keeps track of aliasing,
- Proceeds by function inlining,
- Supports bounded polymorphism,
- Supports $\approx 200$ functions from the standard library.
Why don’t you use classical typing?

- We do not forbid some valid Python programs.
- Rather, we collect exceptions (that can be caught later on).
- Our analysis is flow-sensitive.
- Our analysis could be extended with other static analyses.
- Our analysis is not as modular as most type systems are (concerning functions, loops).
Modular Open Platform for Static Analysis

- Modular abstract domains are small “blocks”, handling everything from: abstract values to control-flow statements.
- Statements flow through these domains until one answers.
- The user can select the combination of abstract domains.
- Supports Python and C analysis (some parts are shared in a “universal” language).
Implementation size:

- 5500 lines of OCaml for Python’s semantics,
- 2500 for Python’s type abstract domain,
- 2100 for Python’s containers abstractions,
- 1800 for the universal language (loop & function analysis),
- 15000 for the modular framework.
## Official Python Benchmarks:

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time</th>
<th># A.</th>
<th># F.A.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fannkuch.py</td>
<td>59</td>
<td>0.07s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>float.py</td>
<td>63</td>
<td>0.06s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>spectral_n.py</td>
<td>74</td>
<td>0.33s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>1.5s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>chaos.py</td>
<td>324</td>
<td>5.9s</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>5.4s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>2.2m</td>
<td>0</td>
<td>52</td>
</tr>
</tbody>
</table>

A real bug was found: a piece of currently unused code was working in Python 2, but not in Python 3.
Official Python Benchmarks:

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time</th>
<th># A.</th>
<th># F.A.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fannkuch.py</td>
<td>59</td>
<td>0.07s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>float.py</td>
<td>63</td>
<td>0.06s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>spectral_n.py</td>
<td>74</td>
<td>0.33s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>1.5s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>chaos.py</td>
<td>324</td>
<td>5.9s</td>
<td>1(^1)</td>
<td>0</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>5.4s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>2.2m</td>
<td>0</td>
<td>52</td>
</tr>
</tbody>
</table>

\(^1\)A real bug was found: a piece of currently unused code was working in Python 2.x, but not in Python 3.x.
Conclusion
We have developed a static type analysis for Python.

It analyzes real-world benchmarks!

**Future Work**

- Better concrete semantics,
- Summary-based function analysis,
- Handle libraries,
- Analyze real-world programs.