Semantics and Static Analysis of Python

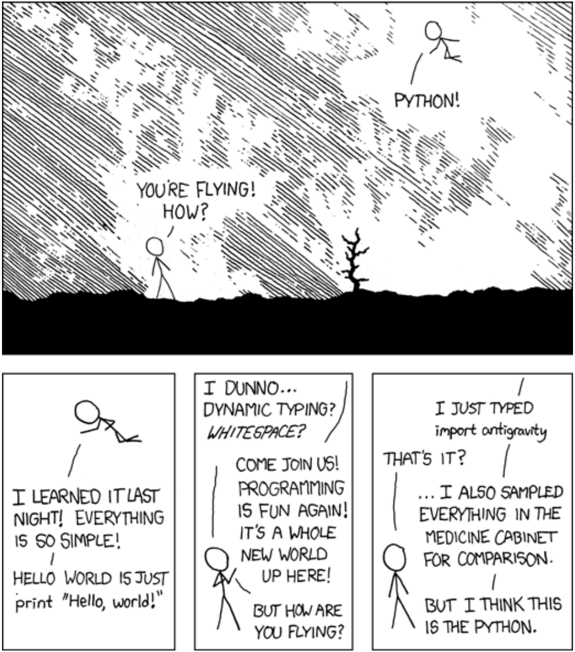
Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

2nd July 2019

European Research Council
Established by the European Commission

SORBONNE UNIVERSITÉ

Introduction


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "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.

xkcd.com/353/

Python is a Dynamic Programming Language

It features:

- A concise and efficient syntax,
- Dynamic typing: types are only known at runtime,
- Introspection,
- Self-modification.

Python Example from the “os” Library

```
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("...")
```

Python Example from the “os” Library

```
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("...")
```

Introspection!

Python Example from the "os" Library

```
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.

Python Example from the "os" Library

```
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?

Python Example from the "os" Library

```
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("...")
```

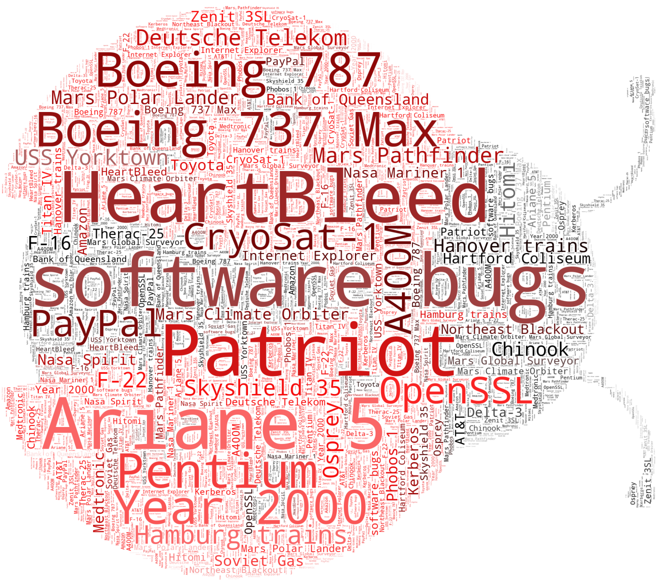
Two notions of typing:

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

Type of fspath?

$\alpha \to \alpha, \alpha \in \{\mathit{str}, \mathit{bytes}\}$ 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"

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

Motivation

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

A theoretical hurdle: Rice's theorem

"any non-trivial semantic property of programs is undecidable"

$\Longrightarrow$ we will compute approximate results

$\Longrightarrow$ 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"

$\Longrightarrow$ we will compute approximate results

$\Longrightarrow$ 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 Analysis & 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.

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 Analysis & 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...
but first let's take a look at Python's semantics.

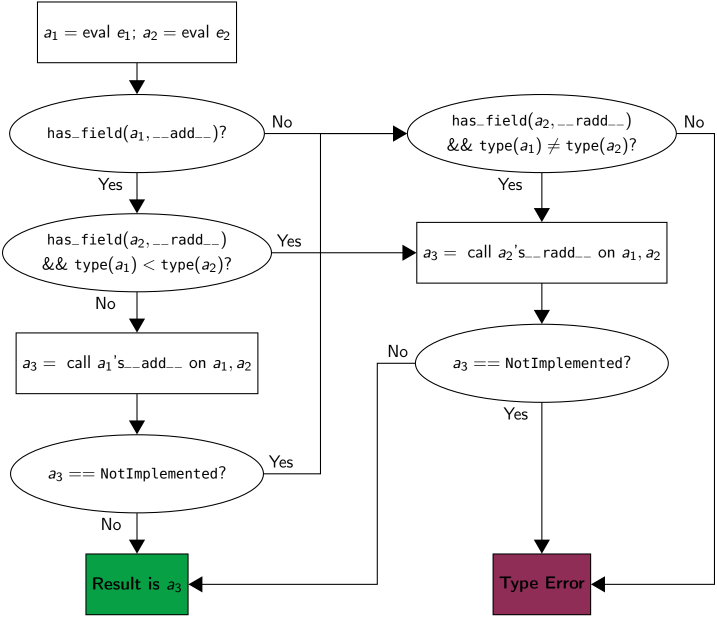
Semantics of Python

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: $e_1 + e_2$



Semantics – Challenges and Future Work

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...

Semantics – Challenges and Future Work

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.

Semantics – Challenges and Future Work

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

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.

Static Type Analysis Example

```python
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
```

```python
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.

Classical Typing vs Static Type Analysis

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).

Implementation into MOPSA

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

- 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 into MOPSA

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.

Benchmarks

Official Python Benchmarks:

Name	LOC	Time	# A.	# F.A.
fannkuch.py	59	0.07s	0	0
float.py	63	0.06s	0	0
spectral_n.py	74	0.33s	0	1
nbody.py	157	1.5s	0	1
chaos.py	324	5.9s	**1**	0
unpack_seq.py	458	5.4s	0	0
hexiom.py	674	2.2m	0	52

Official Python Benchmarks:

Name	LOC	Time	# A.	# F.A.
fannkuch.py	59	0.07s	0	0
float.py	63	0.06s	0	0
spectral_n.py	74	0.33s	0	1
nbody.py	157	1.5s	0	1
chaos.py	324	5.9s	**1**[1]	0
unpack_seq.py	458	5.4s	0	0
hexiom.py	674	2.2m	0	52

[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

Conclusion & Future Work

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.