Static Type Analysis by Abstract Interpretation of Python Programs

---

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

15th & 16th November 2020

`https://rmonat.fr/ecoop20/`

Introduction

Python, from a PL perspective

- #2 language on Github,

Tratt. “Dynamically Typed Languages”. Advances in Computers 2009

Python, from a PL perspective

- #2 language on Github,
- Object oriented,

Tratt. “Dynamically Typed Languages”. Advances in Computers 2009

Python, from a PL perspective

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

Tratt. “Dynamically Typed Languages”. Advances in Computers 2009

Python, from a PL perspective

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

Tratt. “Dynamically Typed Languages”. Advances in Computers 2009

Python, from a PL perspective

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

Tratt. “Dynamically Typed Languages”. Advances in Computers 2009

Python, from a PL perspective

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

Tratt. “Dynamically Typed Languages”. Advances in Computers 2009

Python, from a PL perspective

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

Tratt. “Dynamically Typed Languages”. Advances in Computers 2009

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
  raise TypeError('...')
```

Python Example from the “os” Library

```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
  raise TypeError('...')
```

Introspection, nominal types.

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
  raise TypeError('...')
```

Introspection, structural types.

Python Example from the “os” Library

```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
  raise TypeError('...')
```

Two notions of typing:

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

Python Example from the “os” Library

```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
  raise TypeError('...')
```

Two notions of typing:

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

Type of fspath?

Python Example from the “os” Library

```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
  raise TypeError('...')
```

Two notions of typing:

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

Type of fspath?

str → str; bytes → bytes;

Python Example from the “os” Library

```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
  raise TypeError('...')
```

Two notions of typing:

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

Type of fspath?

str → str; bytes → bytes;

$$\{\, o \to T \mid o.\texttt{\_\_fspath\_\_} : \mathit{unit} \to T,\; T \in \{\,\mathrm{str}, \mathrm{bytes}\,\}\,\}$$

Static Analysis by Abstract Interpretation

Motivation

- Detect all runtime errors,
- At “compile time”, before the program execution.

Static Analysis by Abstract Interpretation

Motivation

- Detect all runtime errors,
- At “compile time”, before the program execution.

Approach

- Interpret the program in a specific domain,
- Compute approximate results, which may yield false alarms.

Static Analysis by Abstract Interpretation

Motivation

- Detect all runtime errors,
- At “compile time”, before the program execution.

Approach

- Interpret the program in a specific domain,
- Compute approximate results, which may yield false alarms.

Goals

- Automatic analysis: no expert knowledge required.
- Sound analysis: if no bug is detected, none will occur.

Static Analysis & Dynamic Programming Languages

Static analyses successfully work on critical embedded C software.

Python leaves less information in the syntax.

Static Analysis & Dynamic Programming Languages

Static analyses successfully work on critical embedded C software.
Python leaves less information in the syntax.

**Static analyses could be especially helpful – though difficult –
on dynamic programming languages.**

Static Analysis & Dynamic Programming Languages

Static analyses successfully work on critical embedded C software.
Python leaves less information in the syntax.

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

We present a type abstract domain for Python...

Static Analysis & Dynamic Programming Languages

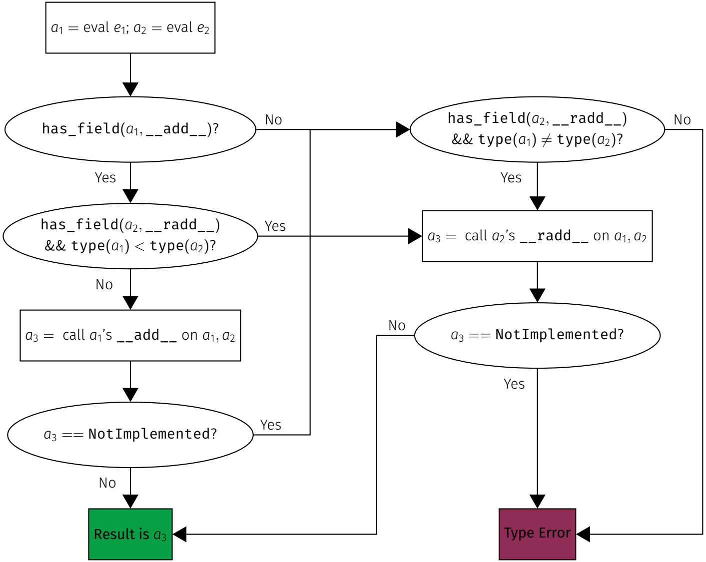
Static analyses successfully work on critical embedded C software.
Python leaves less information in the syntax.

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

We present a type abstract domain for Python…
but first let us take a look at Python's semantics.

Concrete Semantics of Python

Semantics – Example: $e_1 + e_2$



Semantics – Example: $e_1 + e_2$

```
$\mathbb{E}[\![ e_1 + e_2 ]\!] (f, \epsilon, \sigma) \overset{\text{def}}{=}$
    if $f \neq cur$ then $(f, \epsilon, \sigma)$ else
    letif $(f, \epsilon, \sigma, a_1) =$ $\mathbb{E}[\![ e_1 ]\!] (f, \epsilon, \sigma)$ in
    letif $(f, \epsilon, \sigma, a_2) =$ $\mathbb{E}[\![ e_2 ]\!] (f, \epsilon, \sigma)$ in
    if $hasattr(\sigma(a_1), \_\_add\_\_)$ then
        if $hasattr(\sigma(a_2), \_\_radd\_\_) \land type(a_1) < type(a_2)$ then
            letif $(f, \epsilon, \sigma, a_r) =$ $\mathbb{E}[\![ a_2.\_\_radd\_\_(a_1) ]\!]$ in
            if $\sigma(a_r) = \texttt{NotImpl}$ then $empty\_addr \circ \mathbb{S}[\![ \texttt{raise TypeError} ]\!](f, \epsilon, \sigma)$
            else $(f, \epsilon, \sigma, a_r)$
        else letif $(f, \epsilon, \sigma, a_r) =$ $\mathbb{E}[\![ a_1.\_\_add\_\_(a_2) ]\!]$ in
            if $\sigma(a_r) = \texttt{NotImpl}$ then
                if $hasattr(\sigma(a_2), \_\_radd\_\_) \land type(a_1) \neq type(a_2)$ then
                    letif $(f, \epsilon, \sigma, a_r) =$ $\mathbb{E}[\![ a_2.\_\_radd\_\_(a_1) ]\!]$ in
                    if $\sigma(a_r) = \texttt{NotImpl}$ then $empty\_addr \circ \mathbb{S}[\![ \texttt{raise TypeError} ]\!](f, \epsilon, \sigma)$
                    else $(f, \epsilon, \sigma, a_r)$
            else $(f, \epsilon, \sigma, a_r)$
    else if $hasattr(\sigma(a_2), \_\_radd\_\_) \land type(a_1) \neq type(a_2)$ then
        letif $(f, \epsilon, \sigma, a_r) =$ $\mathbb{E}[\![ a_2.\_\_radd\_\_(a_1) ]\!]$ in
        if $\sigma(a_r) = \texttt{NotImpl}$ then $empty\_addr \circ \mathbb{S}[\![ \texttt{raise TypeError} ]\!](f, \epsilon, \sigma)$
        else $(f, \epsilon, \sigma, a_r)$
    else $empty\_addr \circ \mathbb{S}[\![ \texttt{raise TypeError} ]\!](f, \epsilon, \sigma)$
```

Type Analysis of Python

Features of the Analysis

Our goal: Have a sound analysis, inferring both kind of types.
Detect uncaught exceptions (`TypeError`, `AttributeError`).

Features of the Analysis

Our goal: Have a sound analysis, inferring both kind of types.

Detect uncaught exceptions (`TypeError`, `AttributeError`).

```
def dint(x):
  if isinstance(x, int): return x*2
  else: raise TypeError

try: z2 = dint('a')
except TypeError: z2 = dint(1)
# z2: int
```

Features of the Analysis

Our goal: Have a sound analysis, inferring both kind of types.

Detect uncaught exceptions (`TypeError`, `AttributeError`).

```
def dint(x):
  if isinstance(x, int): return x*2
  else: raise TypeError

try: z2 = dint('a')
except TypeError: z2 = dint(1)
# z2: int
```

⟹ Flow-sensitive analysis including exceptions.

Features of the Analysis

Our goal: Have a sound analysis, inferring both kind of types.

Detect uncaught exceptions (`TypeError`, `AttributeError`).

```python
def dint(x):
  if isinstance(x, int): return x*2
  else: raise TypeError

try: z2 = dint('a')
except TypeError: z2 = dint(1)
# z2: int
```

⟹ Flow-sensitive analysis including exceptions.

```python
class A:
  def __init__(self): self.v = 0
x = A()
y = x.v # y: int
z = x
z.v = 'a'
assert(z.v == x.v)
```

Features of the Analysis

Our goal: Have a sound analysis, inferring both kind of types.
Detect uncaught exceptions (`TypeError`, `AttributeError`).

```python
def dint(x):
  if isinstance(x, int): return x*2
  else: raise TypeError

try: z2 = dint('a')
except TypeError: z2 = dint(1)
# z2: int
```

⟹ Flow-sensitive analysis including exceptions.

```python
class A:
  def __init__(self): self.v = 0
x = A()
y = x.v # y: int
z = x
z.v = 'a'
assert(z.v == x.v)
```

⟹ Handle addresses and aliasing.

Features of the Analysis

Our goal: Have a sound analysis, inferring both kind of types.

Detect uncaught exceptions (`TypeError`, `AttributeError`).

```
def dint(x):
  if isinstance(x, int): return x*2
  else: raise TypeError

try: z2 = dint('a')
except TypeError: z2 = dint(1)
# z2: int
```

⟹ Flow-sensitive analysis including exceptions.

```
class A:
  def __init__(self): self.v = 0
x = A()
y = x.v # y: int
z = x
z.v = 'a'
assert(z.v == x.v)
```

⟹ Handle addresses and aliasing.

Bravenboer and Smaragdakis. “Exception Analysis and Points-to Analysis: Better Together”. ISSTA 2009

Simplified Analysis: Example

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

p = "/dev" if random() else Path()

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

r = fspath(p)
```

Simplified Analysis: Example

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

p = "/dev" if random() else Path()

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

r = fspath(p)
```

Simplified Analysis: Example

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

p = "/dev" if random() else Path()  ●

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

r = fspath(p)
```

$$\begin{cases} p \mapsto \{@_{str}, @_{Path}\} \\ @_{Path} \mapsto \{\_\_\mathrm{fspath}\_\_\} \\ @_{Path}.\_\_\mathrm{fspath}\_\_ \mapsto \{@_{meth}\} \end{cases}$$

Simplified Analysis: Example

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

p = "/dev" if random() else Path() ●

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

r = fspath(p)
```

$$\begin{cases} p \mapsto \{@_{str}, @_{Path}\} \\ @_{Path} \mapsto \{\_\_\mathrm{fspath}\_\_\} \\ @_{Path}.\_\_\mathrm{fspath}\_\_ \mapsto \{ @_{meth} \} \end{cases}$$

Simplified Analysis: Example

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

p = "/dev" if random() else Path() ●

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

r = fspath(p)
```

$$\begin{cases} p \mapsto \{@_{str}, @_{Path}\} \\ @_{Path} \mapsto \{\text{\_\_fspath\_\_}\} \\ @_{Path}.\text{\_\_fspath\_\_} \mapsto \{@_{meth}\} \end{cases}$$

$$p \mapsto \{@_{str}\}$$

Simplified Analysis: Example

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

p = "/dev" if random() else Path() ●

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

r = fspath(p)
```

$$\begin{cases} p \mapsto \{@_{str}, @_{Path}\} \\ @_{Path} \mapsto \{\_\_fspath\_\_\} \\ @_{Path}.\_\_fspath\_\_ \mapsto \{@_{meth}\} \end{cases}$$

$$p \mapsto \{@_{str}\}$$

Simplified Analysis: Example

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

p = "/dev" if random() else Path() ●

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

r = fspath(p)
```

$$\begin{cases} p \mapsto \{@_{str}, @_{Path}\} \\ @_{Path} \mapsto \{\_\_\mathrm{fspath}\_\_\} \\ @_{Path}.\_\_\mathrm{fspath}\_\_ \mapsto \{@_{meth}\} \end{cases}$$

$$p \mapsto \{@_{str}\}$$

$$\begin{cases} p \mapsto \{@_{Path}\}; r \mapsto \{@_{int}\} \\ @_{Path} \mapsto \{\_\_\mathrm{fspath}\_\_\} \\ @_{Path}.\_\_\mathrm{fspath}\_\_ \mapsto \{@_{meth}\} \end{cases}$$

Simplified Analysis: Example

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

p = "/dev" if random() else Path() ●

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

r = fspath(p) ●
```

$$\begin{cases} p \mapsto \{@_{str}, @_{Path}\} \\ @_{Path} \mapsto \{\_\_fspath\_\_\} \\ @_{Path}.\_\_fspath\_\_ \mapsto \{ @_{meth} \} \end{cases}$$

$$p \mapsto \{@_{str}\}$$

$$\begin{cases} p \mapsto \{@_{Path}\}; r \mapsto \{@_{int}\} \\ @_{Path} \mapsto \{\_\_fspath\_\_\} \\ @_{Path}.\_\_fspath\_\_ \mapsto \{ @_{meth} \} \end{cases}$$

Simplified Analysis: Example

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

p = "/dev" if random() else Path() ●

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

r = fspath(p) ●
```

$$\begin{cases} p \mapsto \{@_{str}, @_{Path}\} \\ @_{Path} \mapsto \{\_\_\text{fspath}\_\_\} \\ @_{Path}.\_\_\text{fspath}\_\_ \mapsto \{@_{meth}\} \end{cases}$$

$$p \mapsto \{@_{str}\}$$

$$\begin{cases} p \mapsto \{@_{Path}\}; r \mapsto \{@_{int}\} \\ @_{Path} \mapsto \{\_\_\text{fspath}\_\_\} \\ @_{Path}.\_\_\text{fspath}\_\_ \mapsto \{@_{meth}\} \end{cases}$$

$$\text{TypeError} \vee r \mapsto \{@_{str}\}$$

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\mathsf{Addr}^\sharp \overset{\text{def}}{=} \mathsf{Location} \times \{\, \mathsf{r}, \mathsf{o} \,\}$

---

[1] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\mathbf{Addr}^\sharp \overset{\text{def}}{=} \mathbf{Location} \times \{\,\mathsf{r}, \mathsf{o}\,\}$
  - Recent address, strong update

---

[1] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\mathsf{\textbf{Addr}}^\sharp \overset{\text{def}}{=} \mathsf{\textbf{Location}} \times \{\, \mathsf{r}, \mathsf{o} \,\}$
  - Recent address, strong update
  - Old address, weak updates

---

[1] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\textbf{Addr}^\sharp \overset{\text{def}}{=} \textbf{Location} \times \{\,\textbf{r}, \textbf{o}\,\}$
  - Recent address, strong update
  - Old address, weak updates
- Keeping types:

---

[1] Balakrishnan and Reps. "Recency-Abstraction for Heap-Allocated Storage". SAS 2006.

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\mathsf{Addr}^\sharp \overset{\text{def}}{=} \mathsf{Location} \times \{\, \mathsf{r}, \mathsf{o} \,\}$
  - Recent address, strong update
  - Old address, weak updates
- Keeping types:
  - Nominal: $\mathsf{ObjN}^\sharp \overset{\text{def}}{=} \mathsf{Class}(c) \cup \mathsf{Inst}(a), a \in \mathsf{Addr}^\sharp$

---

[1] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\mathbf{Addr}^\sharp \overset{\text{def}}{=} \mathbf{Location} \times \{\, \mathbf{r}, \mathbf{o} \,\}$
  - Recent address, strong update
  - Old address, weak updates
- Keeping types:
  - Nominal: $\mathbf{ObjN}^\sharp \overset{\text{def}}{=} \mathbf{Class}(c) \cup \mathbf{Inst}(a), a \in \mathbf{Addr}^\sharp$
  - Structural: $\mathbf{ObjS}^\sharp \overset{\text{def}}{=} \{\, \top \,\} \cup (\ \mathcal{P}(\mathit{string}) \ \times (\mathit{string} \rightharpoonup \mathbf{Addr}^\sharp))$

---

[1] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\mathbf{Addr}^\sharp \overset{\text{def}}{=} \mathbf{Location} \times \{\, \mathsf{r}, \mathsf{o} \,\}$
  - Recent address, strong update
  - Old address, weak updates
- Keeping types:
  - Nominal: $\mathbf{ObjN}^\sharp \overset{\text{def}}{=} \mathbf{Class}(c) \cup \mathbf{Inst}(a), a \in \mathbf{Addr}^\sharp$
  - Structural: $\mathbf{ObjS}^\sharp \overset{\text{def}}{=} \{\, \top \,\} \cup (\, \mathcal{P}(string) \times (string \rightharpoonup \mathbf{Addr}^\sharp))$

---

[1] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\textbf{Addr}^\sharp \overset{\text{def}}{=} \textbf{Location} \times \{\, \textbf{r}, \textbf{o} \,\}$
  - Recent address, strong update
  - Old address, weak updates
- Keeping types:
  - Nominal: $\textbf{ObjN}^\sharp \overset{\text{def}}{=} \textbf{Class}(c) \cup \textbf{Inst}(a), a \in \textbf{Addr}^\sharp$
  - Structural: $\textbf{ObjS}^\sharp \overset{\text{def}}{=} \{\, \top \,\} \cup \left(\underbrace{\mathcal{P}(\mathit{string})}_{\text{must attributes}} \times (\mathit{string} \rightharpoonup \textbf{Addr}^\sharp)\right)$

---

[1] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.

Analysis State: Technical Overview

- Addresses & aliasing[1]: $\mathbf{Addr}^\sharp \overset{\text{def}}{=} \mathbf{Location} \times \{\, \mathbf{r}, \mathbf{o} \,\}$
  - Recent address, strong update
  - Old address, weak updates
- Keeping types:
  - Nominal: $\mathbf{ObjN}^\sharp \overset{\text{def}}{=} \mathbf{Class}(c) \cup \mathbf{Inst}(a), a \in \mathbf{Addr}^\sharp$
  - Structural: $\mathbf{ObjS}^\sharp \overset{\text{def}}{=} \{\, \top \,\} \cup (\, \underbrace{\mathcal{P}(string)}_{\text{must attributes}} \times (string \rightharpoonup \mathbf{Addr}^\sharp))$
- Flow-sensitive analysis: state partitioning by flow tokens

$$\mathcal{F}^\sharp \overset{\text{def}}{=} \{\, cur, ret, brk, cont, exn \;\; @^\sharp, @^\sharp \in \mathbf{Addr}^\sharp \,\}$$

---

[1] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.

Comparison with Classical Typing Approaches

Our Approach	"Classical" Typing
No restriction on the language	Valid programs may be rejected

Comparison with Classical Typing Approaches

Our Approach	“Classical” Typing
No restriction on the language	Valid programs may be rejected
Type errors are catchable exceptions	Type errors are fatal

Comparison with Classical Typing Approaches

Our Approach	“Classical” Typing
No restriction on the language	Valid programs may be rejected
Type errors are catchable exceptions	Type errors are fatal
Flow-sensitive analysis (dynamic typing & exceptions)	Flow-insensitive analysis

Comparison with Classical Typing Approaches

Our Approach	“Classical” Typing
No restriction on the language	Valid programs may be rejected
Type errors are catchable exceptions	Type errors are fatal
Flow-sensitive analysis (dynamic typing & exceptions)	Flow-insensitive analysis
Dynamic attribute addition changes types	Immutable types

Comparison with Classical Typing Approaches

Our Approach	“Classical” Typing
No restriction on the language	Valid programs may be rejected
Type errors are catchable exceptions	Type errors are fatal
Flow-sensitive analysis (dynamic typing & exceptions)	Flow-insensitive analysis
Dynamic attribute addition changes types	Immutable types
Similar, relational domain	Parametric polymorphism

Comparison with Classical Typing Approaches

Our Approach	"Classical" Typing
No restriction on the language	Valid programs may be rejected
Type errors are catchable exceptions	Type errors are fatal
Flow-sensitive analysis (dynamic typing & exceptions)	Flow-insensitive analysis
Dynamic attribute addition changes types	Immutable types
Similar, relational domain	Parametric polymorphism
More costly, context-sensitive interprocedural analysis	Functions are analyzed in isolation
Top-down analysis	Bottom-up analysis

Extensions of the Analysis

Relational Type Equality Domain

```
def get_sep(s):
  if isinstance(s, str): return '/'
  elif isinstance(s, bytes): return b'/'
  else: raise TypeError

if *: r = '/dev/null'
else: r = b'/dev/null'
sep = get_sep(r)
```

$$\underbrace{\begin{cases} r \in \{\, @_{str} \,\} \\ sep \in \{\, @_{str} \,\} \end{cases}}_{\text{if branch}}$$

Relational Type Equality Domain

```
def get_sep(s):
  if isinstance(s, str): return '/'
  elif isinstance(s, bytes): return b'/'
  else: raise TypeError

if *: r = '/dev/null'
else: r = b'/dev/null'
sep = get_sep(r)
```

$$\underbrace{\begin{cases} r \in \{ @_{str} \} \\ sep \in \{ @_{str} \} \end{cases}}_{\text{if branch}} \quad \underbrace{\begin{cases} r \in \{ @_{bytes} \} \\ sep \in \{ @_{bytes} \} \end{cases}}_{\text{else branch}}$$

Relational Type Equality Domain

```python
def get_sep(s):
  if isinstance(s, str): return '/'
  elif isinstance(s, bytes): return b'/'
  else: raise TypeError

if *: r = '/dev/null'
else: r = b'/dev/null'
sep = get_sep(r)
```

$$\underbrace{\begin{cases} r \in \{ @_{str} \} \\ sep \in \{ @_{str} \} \end{cases}}_{\text{if branch}} \sqcup \underbrace{\begin{cases} r \in \{ @_{bytes} \} \\ sep \in \{ @_{bytes} \} \end{cases}}_{\text{else branch}}$$

Relational Type Equality Domain

```
def get_sep(s):
  if isinstance(s, str): return '/'
  elif isinstance(s, bytes): return b'/'
  else: raise TypeError

if *: r = '/dev/null'
else: r = b'/dev/null'
sep = get_sep(r)
```

$$\underbrace{\begin{cases} r \in \{\, @_{str} \,\} \\ sep \in \{\, @_{str} \,\} \end{cases}}_{\text{if branch}} \sqcup \underbrace{\begin{cases} r \in \{\, @_{bytes} \,\} \\ sep \in \{\, @_{bytes} \,\} \end{cases}}_{\text{else branch}} = \begin{cases} r \in \{\, @_{str}, @_{bytes} \,\} \\ sep \in \{\, @_{str}, @_{bytes} \,\} \end{cases}$$

Relational Type Equality Domain

```
def get_sep(s):
  if isinstance(s, str): return '/'
  elif isinstance(s, bytes): return b'/'
  else: raise TypeError

if *: r = '/dev/null'
else: r = b'/dev/null'
sep = get_sep(r)
```

$$\underbrace{\begin{cases} r \in \{ @_{str} \} \\ sep \in \{ @_{str} \} \end{cases}}_{\text{if branch}} \sqcup \underbrace{\begin{cases} r \in \{ @_{bytes} \} \\ sep \in \{ @_{bytes} \} \end{cases}}_{\text{else branch}} = \begin{cases} r \in \{ @_{str}, @_{bytes} \} \\ sep \in \{ @_{str}, @_{bytes} \} \end{cases} \wedge \; r \equiv sep$$

Relational Type Equality Domain

```
def get_sep(s):
  if isinstance(s, str): return '/'
  elif isinstance(s, bytes): return b'/'
  else: raise TypeError

if *: r = '/dev/null'
else: r = b'/dev/null'
sep = get_sep(r)
```

$$\underbrace{\begin{cases} r \in \{\, @_{str} \,\} \\ sep \in \{\, @_{str} \,\} \end{cases}}_{\text{if branch}} \sqcup \underbrace{\begin{cases} r \in \{\, @_{bytes} \,\} \\ sep \in \{\, @_{bytes} \,\} \end{cases}}_{\text{else branch}} = \begin{cases} r \in \{\, @_{str}, @_{bytes} \,\} \\ sep \in \{\, @_{str}, @_{bytes} \,\} \end{cases} \wedge \; r \equiv sep$$

Then, we can analyze `res = r + sep` without false alarms.

The polymorphism improves the precision.

Interprocedural Analysis

What about a context-insensitive analysis?

```
def f(e1, e2): return e1 + e2
```

Interprocedural Analysis

What about a context-insensitive analysis?

```
def f(e1, e2): return e1 + e2
```

`e1.__add__` can be any function

Interprocedural Analysis

What about a context-insensitive analysis?

```
def f(e1, e2): return e1 + e2
```

$\implies$ We focus on a context-sensitive analysis.

Interprocedural Analysis

What about a context-insensitive analysis?

```
def f(e1, e2): return e1 + e2
```

$\Longrightarrow$ We focus on a context-sensitive analysis.

Inlining most precise, but costly.

Interprocedural Analysis

What about a context-insensitive analysis?

```
def f(e1, e2): return e1 + e2
```

$\implies$ We focus on a context-sensitive analysis.

Inlining most precise, but costly.

Towards function summaries a simple cache keeping the previous function analyses achieves up to 7x speedup on our benchmarks.

Experimental Evaluation

Implementation into Mopsa

Modular **O**pen **P**latform for **S**tatic **A**nalysis[2]

- Modular abstract domains are small “blocks”.
- The user can select the combination of abstract domains.
- Supports Python and C analysis
  (some parts are shared in a “universal” language).

Available now: `https://gitlab.com/mopsa/mopsa-analyzer`

---

[2] Journault et al. “Combinations of reusable abstract domains for a multilingual static analyzer”. VSTTE 2019.

Features and Limitations

Our tool is sound; it supports:

- Objects
- Exceptions
- Dynamic typing
- Introspection
- Permissive semantics
- Dynamic attributes

Currently, it does not support:

- Recursive functions
- `super`
- metaclasses
- `eval`

Related Work on Python Analysis

- Dataflow analysis by Fritz and Hage[3].
- Sound Typpete: SMT-based type inference[4].
- Pytype, type inference tool used by Google.[5].
- RPython: efficient compilation of a static subset of Python[6].
- Sound Value Analysis by Fromherz et al[7].

---

[3] Fritz and Hage. "Cost versus precision for approximate typing for Python". PEPM 2017.

[4] Hassan et al. "MaxSMT-Based Type Inference for Python 3". CAV 2018.

[5] Kramm et al. Pytype. 2019.

[6] Ancona et al. "RPython: a step towards reconciling dynamically and statically typed OO languages". DLS 2007.

[7] Fromherz et al. "Static Value Analysis of Python Programs by Abstract Interpretation". NFM 2018.

Experimental Evaluation

Name	LOC	Conf. 1	Conf. 2	⚠	Fritz & Hage	Pytype	Typpete	Fromherz et al.	RPython
🐇 bellman_ford.py	61	0.17s	0.24s	0†	1.4s	0.99s	1.4s	2.4m	7.1s
🐍 float.py	63	0.13s	82ms	0†	1.7s	0.92s	1.3s	0.84s	5.6s
🐇 coop_concat.py	64	45ms	43ms	0†	1.8s	0.81s	1.3s	20ms	🐞
🐍 spectral_norm.py	74	0.32s	0.19s	1	1.6s	0.98s	🐞	🐞	🐞
🐇 crafting.py	132	0.48s	0.41s	0†🔑	1.6s	0.97	1.7s	🐞	🐞
🐍 raytrace.py	411	3.5s	1.5s	7⚓	36s	2.8s	🐞	🐞	🐞
🐍 scimark.py	416	0.85s	0.55s	2†	8.5s	4.4s	🐞	🐞	🐞
🐍 richards.py	426	11s	5.0s	2†⚓	38s	2.4s	🐞	🐞	7.8s
🐍 unpack_seq.py	458	13s	4.2s	0⚓	1.1s	7.4s	2.7s	14s	🐞
🐍 go.py	461	4.0m	15s	32†⚓	8.5s	3.4s	🐞	🐞	🐞
🐍 hexiom.py	674	6.9m	22s	25†🔑⚓	🐞	4.2s	🐞	🐞	🐞
🐍 regex_v8.py	1792	8.2m	15s	0†	4.9s	🕒	1.7m	🐞	🐞
ⓕ processInput.py	1417	6.1s	4.8s	7†🔑⚓	2.4s	11s	🐞	🐞	🐞
ⓕ choose.py	2562	8.6m	46s	17🔑†⚓	1.7s	15s	🐞	🐞	🐞

Sound tool. 🐞 unsupported by the analyzer. 🕒 timeout (1h). Smashed Exceptions: KeyError 🔑, IndexError †, ValueError ⚓.

Conclusion

Conclusion

A new static type analysis for Python:

- Takes into account dynamic Python features.

Conclusion

A new static type analysis for Python:

- Takes into account dynamic Python features.
- Analyzes real-world benchmarks.

Conclusion

A new static type analysis for Python:

- Takes into account dynamic Python features.
- Analyzes real-world benchmarks.
- Sound: relation between analysis and the concrete semantics.

Conclusion

A new static type analysis for Python:

- Takes into account dynamic Python features.
- Analyzes real-world benchmarks.
- Sound: relation between analysis and the concrete semantics.

Future work:

Conclusion

A new static type analysis for Python:

- Takes into account dynamic Python features.
- Analyzes real-world benchmarks.
- Sound: relation between analysis and the concrete semantics.

Future work:

- Recursive functions, super, metaclasses, eval.

Conclusion

A new static type analysis for Python:

- Takes into account dynamic Python features.
- Analyzes real-world benchmarks.
- Sound: relation between analysis and the concrete semantics.

Future work:

- Recursive functions, super, metaclasses, eval.
- Summary-based function analysis.

Conclusion

A new static type analysis for Python:

- Takes into account dynamic Python features.
- Analyzes real-world benchmarks.
- Sound: relation between analysis and the concrete semantics.

Future work:

- Recursive functions, super, metaclasses, eval.
- Summary-based function analysis.
- Handle libraries.

Conclusion

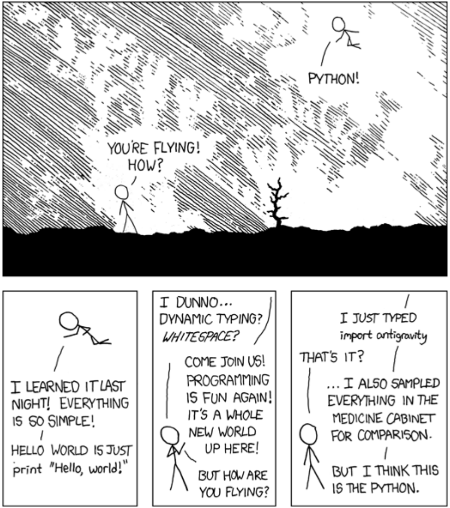
A new static type analysis for Python:

- Takes into account dynamic Python features.
- Analyzes real-world benchmarks.
- Sound: relation between analysis and the concrete semantics.

Future work:

- Recursive functions, super, metaclasses, eval.
- Summary-based function analysis.
- Handle libraries.
- Analyze bigger programs.

Thank you! Questions?



`xkcd.com/353/`