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,

Python, from a PL perspective

- #2 language on Github,
- Object oriented,
Python, from a PL perspective

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

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,

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,

---

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,

---

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.

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
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.
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('...')
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.
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
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 → T | o.__fspath__ : unit → T, 
\quad T ∈ \{ str, bytes \} \}\
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 analyses successfully work on critical embedded C software. Python leaves less information in the syntax.
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 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 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: \( e_1 + e_2 \)

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

```
has_field(a_1, __add__)?
```

- **Yes**
  - \( \text{has_field}(a_2, \text{__radd__}) \land \text{type}(a_1) < \text{type}(a_2) \)?
    - **Yes**
      - \( a_3 = \text{call } a_2\text{\text{'}s } \text{__radd__} \text{ on } a_1, a_2 \)
    - **No**
      - \( a_3 = \text{call } a_1\text{\text{'}s } \text{__add__} \text{ on } a_1, a_2 \)
- **No**
  - \( a_3 = \text{call } a_1\text{\text{'}s } \text{__add__} \text{ on } a_1, a_2 \)

```
a_3 == \text{NotImplemented}?
```

- **Yes**
  - \( a_3 = \text{call } a_2\text{\text{'}s } \text{__radd__} \text{ on } a_1, a_2 \)
- **No**
  - \( a_3 == \text{NotImplemented} ? \)

```
\text{Result is } a_1
```

```
\text{Type Error}
```
Semantics – Example: $e_1 + e_2$

\[
\begin{align*}
\mathbb{E}[e_1 + e_2](f, \epsilon, \sigma) & \overset{\text{def}}{=} \\
& \text{if } f \neq \text{cur} \text{ then } (f, \epsilon, \sigma) \text{ else} \\
& \text{letif } (f, \epsilon, \sigma, a_1) = \mathbb{E}[e_1](f, \epsilon, \sigma) \text{ in} \\
& \text{letif } (f, \epsilon, \sigma, a_2) = \mathbb{E}[e_2](f, \epsilon, \sigma) \text{ in} \\
& \text{if } \text{hasattr}(\sigma(a_1), \_\text{add}_\_\_) \text{ then} \\
& \text{if } \text{hasattr}(\sigma(a_2), \_\text{radd}_\_) \land \text{type}(a_1) < \text{type}(a_2) \text{ then} \\
& \text{letif } (f, \epsilon, \sigma, a_r) = \mathbb{E}[a_2._\text{radd}_\_(a_1)] \text{ in} \\
& \text{if } \sigma(a_r) = \text{NotImpl} \text{ then empty_addr } \circ \mathbb{S}[\text{raise TypeError}](f, \epsilon, \sigma) \\
& \text{else } (f, \epsilon, \sigma, a_r) \\
& \text{else letif } (f, \epsilon, \sigma, a_r) = \mathbb{E}[a_1._\text{add}_\_(a_2)] \text{ in} \\
& \text{if } \sigma(a_r) = \text{NotImpl} \text{ then} \\
& \text{if } \text{hasattr}(\sigma(a_2), \_\text{radd}_\_) \land \text{type}(a_1) \neq \text{type}(a_2) \text{ then} \\
& \text{letif } (f, \epsilon, \sigma, a_r) = \mathbb{E}[a_2._\text{radd}_\_(a_1)] \text{ in} \\
& \text{if } \sigma(a_r) = \text{NotImpl} \text{ then empty_addr } \circ \mathbb{S}[\text{raise TypeError}](f, \epsilon, \sigma) \\
& \text{else } (f, \epsilon, \sigma, a_r) \\
& \text{else if } \text{hasattr}(\sigma(a_2), \_\text{add}_\_) \land \text{type}(a_1) \neq \text{type}(a_2) \text{ then} \\
& \text{letif } (f, \epsilon, \sigma, a_r) = \mathbb{E}[a_2._\text{add}_\_(a_1)] \text{ in} \\
& \text{if } \sigma(a_r) = \text{NotImpl} \text{ then empty_addr } \circ \mathbb{S}[\text{raise TypeError}](f, \epsilon, \sigma) \\
& \text{else } (f, \epsilon, \sigma, a_r) \\
& \text{else empty_addr } \circ \mathbb{S}[\text{raise TypeError}](f, \epsilon, \sigma)
\end{align*}
\]
Type Analysis of Python
Features of the Analysis

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

---

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
1 def dint(x):
2     if isinstance(x, int): return x*2
3     else: raise TypeError
4
5 try: z2 = dint('a')
6 except TypeError: z2 = dint(1)
7 # z2: int
```

---

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

```python
1  def dint(x):
2      if isinstance(x, int): return x*2
3      else: raise TypeError
4
5  try: z2 = dint('a')
6  except TypeError: z2 = dint(1)
7  # 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
1   def dint(x):
2       if isinstance(x, int): return x*2
3       else: raise TypeError
4
5   try: z2 = dint('a')
6   except TypeError: z2 = dint(1)
7   # z2: int

⇒ Flow-sensitive analysis including exceptions.
```

```python
1   class A:
2       def __init__(self): self.v = 0
3       x = A()
4       y = x.v  # y: int
5       z = x
6       z.v = 'a'
7       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')
ext 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**).

```python
1 def dint(x):
2     if isinstance(x, int): return x*2
3     else: raise TypeError
4
try: z2 = dint('a')
extcept TypeError: z2 = dint(1)
# z2: int

⇒ Flow-sensitive analysis including exceptions.
```

```python
1 class A:
2     def __init__(self): self.v = 0
3     x = A()
4     y = x.v  # y: int
5     z = x
6     z.v = 'a'
7     assert(z.v == x.v)

⇒ Handle addresses and aliasing.
```

---

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)
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)
```

```
p \mapsto \{ \text{@str, @Path} \}
@Path \mapsto \{ \text{__fspath__} \}
@Path\cdot__fspath__\_ \mapsto \{ \text{@meth} \}
```
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)
```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)
```

```
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)
```

```
p ↔ \{[@str, @Path]\}
@Path ↔ {__fspath__}
@Path.__fspath__ ↔ \{@meth\}
p ↔ \{[@str]\}
p ↔ \{[@Path]\}; r ↔ \{[@int]\}
@Path ↔ {__fspath__}
@Path.__fspath__ ↔ \{@meth\}
```
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)

\[
\begin{align*}
\text{Example} & \quad \text{Path} \\
\text{class} & \quad \text{Path} \\
\quad \text{def} & \quad \text{__fspath__} (\text{self}) : \quad \text{return} \; 42 \\
\text{p} & \quad = \quad \text{"/dev" if random() else Path()} \\
\text{def} & \quad \text{fspath} (\text{p}) : \\
\quad & \quad \text{if} \quad \text{isinstance} (\text{p}, (\text{str}, \text{bytes})) : \\
\quad & \quad \quad \text{return} \; \text{p} \\
\quad & \quad \quad \text{elif} \quad \text{hasattr} (\text{p}, \text{"__fspath__"}) : \\
\quad & \quad \quad \quad \quad \text{r} = \text{p}.\text{__fspath__} () \\
\quad & \quad \quad \quad \quad \quad \text{if} \quad \text{isinstance} (\text{r}, (\text{str}, \text{bytes})) : \\
\quad & \quad \quad \quad \quad \quad \quad \text{return} \; \text{r} \\
\quad & \quad \quad \quad \quad \quad \quad \quad \text{raise} \; \text{TypeError} \\
\quad & \quad \quad \quad \quad \text{r} = \text{fspath} (\text{p})
\end{align*}
\]
```python
class Path:
    def __fspath__(self): return 42

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

def fspath(p):
    if isinstance(p, (str, bytes)):  # p \mapsto \{str\}
        return p
    elif hasattr(p, "__fspath__"):
        r = p.__fspath__()  # p \mapsto \{Path\}
        if isinstance(r, (str, bytes)):
            return r
        raise TypeError

r = fspath(p)  # r \mapsto \{str\}
```

Simplified Analysis: Example

$$\begin{array}{l}
p \mapsto \{str, Path\}
\end{array}$$

$$\begin{array}{l}
Path \mapsto \{__fspath__\}
\end{array}$$

$$\begin{array}{l}
Path \_\_fspath\_\_ \mapsto \{meth\}
\end{array}$$

$$\begin{array}{l}
p \mapsto \{str\}
\end{array}$$

$$\begin{array}{l}
p \mapsto \{Path\}; r \mapsto \{int\}
\end{array}$$

$$\begin{array}{l}
Path \mapsto \{__fspath__\}
\end{array}$$

$$\begin{array}{l}
Path \_\_fspath\_\_ \mapsto \{meth\}
\end{array}$$

TypeError ∨ $$r \mapsto \{str\}$$
Addresses & aliasing\(^1\): \(\text{Addr}^\# \overset{\text{def}}{=} \text{Location} \times \{ r, o \}\)
Analysis State: Technical Overview

▶ Addresses & aliasing: $\text{Addr}^\# \overset{\text{def}}{=} \text{Location} \times \{ r, o \}$

- Recent address, strong update

---

Addresses & aliasing\(^1\): \( \text{Addr}^\# \overset{\text{def}}{=} \text{Location} \times \{ r, o \} \)

- Recent address, strong update
- Old address, weak updates

---

Analysis State: Technical Overview

- Addresses & aliasing\(^1\): \(\text{Addr}^\# \overset{\text{def}}{=} \text{Location} \times \{ r, o \} \)
  - Recent address, strong update
  - Old address, weak updates

- Keeping types:

Addresses & aliasing\(^1\): \( \text{Addr}^\# \overset{\text{def}}{=} \text{Location} \times \{ r, o \} \)
- Recent address, strong update
- Old address, weak updates

Keeping types:
- Nominal: \( \text{ObjN}^\# \overset{\text{def}}{=} \text{Class}(c) \cup \text{Inst}(a), a \in \text{Addr}^\# \)

Addresses & aliasing\(^1\): \(\text{Addr}\)^# \(\stackrel{\text{def}}{=} \text{Location} \times \{r, o\}\)

- Recent address, strong update
- Old address, weak updates

Keeping types:

- Nominal: \(\text{ObjN}\)^# \(\stackrel{\text{def}}{=} \text{Class}(c) \cup \text{Inst}(a), a \in \text{Addr}\)^#
- Structural: \(\text{ObjS}\)^# \(\stackrel{\text{def}}{=} \{\top\} \cup (\mathcal{P}(\text{string}) \times (\text{string} \rightarrow \text{Addr}\)^#))

Addresses & aliasing: $\text{Addr}^\# \overset{\text{def}}{=} \text{Location} \times \{ \text{r, o} \}$
- Recent address, strong update
- Old address, weak updates

Keeping types:
- Nominal: $\text{ObjN}^\# \overset{\text{def}}{=} \text{Class}(c) \cup \text{Inst}(a), a \in \text{Addr}^\#$
- Structural: $\text{ObjS}^\# \overset{\text{def}}{=} \{ \top \} \cup (\mathcal{P}(\text{string}) \times (\text{string} \rightarrow \text{Addr}^\#))$

---

Addresses & aliasing\(^1\): \(\text{Addr}^\# \stackrel{\text{def}}{=} \text{Location} \times \{r, o\}\)

- Recent address, strong update
- Old address, weak updates

Keeping types:

- Nominal: \(\text{ObjN}^\# \stackrel{\text{def}}{=} \text{Class}(c) \cup \text{Inst}(a), a \in \text{Addr}^\#\)
- Structural: \(\text{ObjS}^\# \stackrel{\text{def}}{=} \{\top\} \cup (\mathcal{P}(\text{string}) \times (\text{string} \rightarrow \text{Addr}^\#))\)

---

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

Keeping types:\n- Nominal: \[ \text{ObjN}^\# \overset{\text{def}}{=} \text{Class}(c) \cup \text{Inst}(a), a \in \text{Addr}^\# \]
- Structural: \[ \text{ObjS}^\# \overset{\text{def}}{=} \{ \top \} \cup (\mathcal{P}(\text{string}) \times (\text{string} \rightarrow \text{Addr}^\#)) \]

Flow-sensitive analysis: state partitioning by flow tokens\n\[ F^\# \overset{\text{def}}{=} \{ \text{cur}, \text{ret}, \text{brk}, \text{cont}, \text{exn} \in \text{Addr}^\# \} \]

---

## Comparison with Classical Typing Approaches

<table>
<thead>
<tr>
<th>Our Approach</th>
<th>“Classical” Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No restriction on the language</td>
<td>Valid programs may be rejected</td>
</tr>
</tbody>
</table>

- **Our Approach**
  - No restriction on the language

- **“Classical” Typing**
  - Valid programs may be rejected
Comparison with Classical Typing Approaches

<table>
<thead>
<tr>
<th>Our Approach</th>
<th>“Classical” Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No restriction on the language</td>
<td>Valid programs may be rejected</td>
</tr>
<tr>
<td>Type errors are catchable exceptions</td>
<td>Type errors are fatal</td>
</tr>
</tbody>
</table>
Comparison with Classical Typing Approaches

<table>
<thead>
<tr>
<th>Our Approach</th>
<th>“Classical” Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No restriction on the language</td>
<td>Valid programs may be rejected</td>
</tr>
<tr>
<td>Type errors are catchable exceptions</td>
<td>Type errors are fatal</td>
</tr>
<tr>
<td>Flow-sensitive analysis (dynamic typing &amp; exceptions)</td>
<td>Flow-insensitive analysis</td>
</tr>
</tbody>
</table>
## Comparison with Classical Typing Approaches

<table>
<thead>
<tr>
<th>Our Approach</th>
<th>“Classical” Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No restriction on the language</td>
<td>Valid programs may be rejected</td>
</tr>
<tr>
<td>Type errors are <strong>catchable exceptions</strong></td>
<td>Type errors are <strong>fatal</strong></td>
</tr>
<tr>
<td>Flow-sensitive analysis (dynamic typing &amp; exceptions)</td>
<td>Flow-insensitive analysis</td>
</tr>
<tr>
<td>Dynamic attribute addition <strong>changes types</strong></td>
<td><strong>Immutable</strong> types</td>
</tr>
</tbody>
</table>
### Comparison with Classical Typing Approaches

<table>
<thead>
<tr>
<th>Our Approach</th>
<th>“Classical” Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No restriction on the language</td>
<td>Valid programs may be rejected</td>
</tr>
<tr>
<td>Type errors are catchable exceptions</td>
<td>Type errors are fatal</td>
</tr>
<tr>
<td>Flow-sensitive analysis (dynamic typing &amp; exceptions)</td>
<td>Flow-insensitive analysis</td>
</tr>
<tr>
<td>Dynamic attribute addition changes types</td>
<td>Immutable types</td>
</tr>
<tr>
<td>Similar, relational domain</td>
<td>Parametric polymorphism</td>
</tr>
</tbody>
</table>
### Comparison with Classical Typing Approaches

<table>
<thead>
<tr>
<th>Our Approach</th>
<th>“Classical” Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No restriction on the language</td>
<td>Valid programs may be rejected</td>
</tr>
<tr>
<td>Type errors are catchable exceptions</td>
<td>Type errors are fatal</td>
</tr>
<tr>
<td>Flow-sensitive analysis (dynamic typing &amp; exceptions)</td>
<td>Flow-insensitive analysis</td>
</tr>
<tr>
<td>Dynamic attribute addition changes types</td>
<td>Immutable types</td>
</tr>
<tr>
<td>Similar, relational domain</td>
<td>Parametric polymorphism</td>
</tr>
<tr>
<td>More costly, context-sensitive interprocedural analysis</td>
<td>Functions are analyzed in isolation</td>
</tr>
<tr>
<td>Top-down analysis</td>
<td>Bottom-up analysis</td>
</tr>
</tbody>
</table>
Extensions of the Analysis
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)

\[
\begin{array}{l}
  \{ r \in \{ \text{@str} \} \} \\
  \{ \text{sep} \in \{ \text{@str} \} \} \\
  \text{if branch}
\end{array}
\]
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)

\[
\begin{align*}
\begin{cases}
    r \in \{\emptyset_{\text{str}}\} \\
    \text{sep} \in \{\emptyset_{\text{str}}\}
\end{cases} & \quad \text{if branch} \\
\begin{cases}
    r \in \{\emptyset_{\text{bytes}}\} \\
    \text{sep} \in \{\emptyset_{\text{bytes}}\}
\end{cases} & \quad \text{else branch}
\end{align*}
\]
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)

\[
\begin{align*}
\{ r \in \{ \texttt{str} \} \} \quad \text{if branch} \\
\{ \text{sep} \in \{ \texttt{str} \} \} \\
\{ r \in \{ \texttt{bytes} \} \} \quad \text{else branch} \\
\{ \text{sep} \in \{ \texttt{bytes} \} \}
\end{align*}
\]
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)

\[
\begin{align*}
\begin{cases}
    r \in \{ \mathbb{str} \} \\
    sep \in \{ \mathbb{str} \}
\end{cases}
\end{align*}
\]
\[
\begin{align*}
\begin{cases}
    r \in \{ \mathbb{bytes} \} \\
    sep \in \{ \mathbb{str} \}
\end{cases}
\end{align*}
\]
if branch

\[
\begin{align*}
\begin{cases}
    r \in \{ \mathbb{str}, \mathbb{bytes} \} \\
    sep \in \{ \mathbb{str}, \mathbb{bytes} \}
\end{cases}
\end{align*}
\]
else branch

Then, we can analyze \( \text{res} = r + sep \) without false alarms. The polymorphism improves the precision.
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)

\[
\begin{align*}
\begin{cases}
 r \in \{ \text{@str} \} \\
 sep \in \{ \text{@str} \}
\end{cases} & \quad \sqsubseteq \\
\begin{cases}
 r \in \{ \text{@bytes} \} \\
 sep \in \{ \text{@bytes} \}
\end{cases} \\
\begin{cases}
 r \in \{ \text{@str}, \text{@bytes} \} \\
 sep \in \{ \text{@str}, \text{@bytes} \}
\end{cases} \quad \land \\
r \equiv sep
\end{align*}
\]

Then, we can analyze \( res = r + sep \) without false alarms. The polymorphism improves the precision.
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)
```

Then, we can analyze `res = r + sep` without false alarms. The polymorphism improves the precision.
What about a context-insensitive analysis?

def f(e1, e2): return e1 + e2
What about a context-insensitive analysis?

```python
def f(e1, e2): return e1 + e2
e1.__add__ can be any function
```
What about a context-insensitive analysis?

```python
def f(e1, e2): return e1 + e2
⇒ We focus on a context-sensitive analysis.
```
What about a context-insensitive analysis?

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

⇒  We focus on a context-sensitive analysis.

Inlining most precise, but costly.
What about a context-insensitive analysis?

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

⇒ 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
Modular Open Platform for Static Analysis\(^2\)

- 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

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\(^7\).

---


\(^4\) Hassan et al. “MaxSMT-Based Type Inference for Python 3”. CAV 2018.


\(^7\) Fromherz et al. “Static Value Analysis of Python Programs by Abstract Interpretation”. NFM 2018.
# Experimental Evaluation

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

*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.

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