Static Analysis of Python Programs

A Type Abstract Domain for Python

Raphaël Monat, Abdelraouf Ouadjiaout, Antoine Miné

7th October 2019
Introduction
YOU'RE FLYING! HOW?

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 SAMPL ED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

xkcd.com/353/
It features:

- A concise and efficient syntax,
- Class-based objects,
- A large standard library,
- Dynamic typing: types are only known at runtime,
- Metaprogramming features:
  - Introspection,
  - Self-modification,
  - Metaclasses,
  - eval.
```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("...")
```

Introspection, structural types.

Two notions of typing:

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

Type of `fspath`?

`str` → `str`; `bytes` → `bytes` and any object having a method `__fspath__` returning `str` or `bytes`. 
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
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, structural 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
        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?
str → str; bytes → bytes and any object having a method __fspath__ returning str or bytes.
Motivation

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

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

A theoretical hurdle: Rice’s theorem

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

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

A theoretical hurdle: Rice’s theorem

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

⇒ We will compute approximate results;
Motivation

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

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.
Static Analysis by Abstract Interpretation

Motivation

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

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 Analysis by Abstract Interpretation 101

\[ \mathcal{S} \] (concrete)

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

\[ \mathcal{D}^{\#} (\text{abstract}) \]
Static Analysis by Abstract Interpretation 101

$D$ (concrete) $\rightarrow \alpha \rightarrow \alpha(S)$ $\rightarrow D^\#$ (abstract)
Static Analysis by Abstract Interpretation 101

\[ \mathcal{D} \quad (\text{concrete}) \quad \xrightarrow{\alpha} \quad \mathcal{D}^\# \quad (\text{abstract}) \]

\[ S \quad \xrightarrow{\gamma} \quad \alpha(S) \]
Static Analysis by Abstract Interpretation 101

\[ \gamma \circ \alpha(S) \]

\[ S \]

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

\[ \alpha(S) \]

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

\[ \mathcal{D} = \mathcal{P}(\mathbb{Z}) \]

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

\[ \alpha(S) \]

\[ \gamma \circ \alpha(S) \]

\[ \gamma \]

\[ \alpha \]

\[ \gamma \circ \alpha(S) = \{ [a, b] \mid a \in \mathbb{Z}, b \in \mathbb{Z}, a \leq b \} \]
Static Analysis by Abstract Interpretation 101

\[ \gamma \circ \alpha(S) \]

\[ S \]

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

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

\[ \gamma \]

\[ \alpha \]

\[ \gamma \circ \alpha(S) \]

\[ D = \mathcal{P}(\mathbb{Z}) \]

\[ D^\# = \{ [a, b] \mid a \in \mathbb{Z}, b \in \mathbb{Z}, a \leq b \} \]
Static Analysis by Abstract Interpretation 101

\( D = \mathcal{P}(\mathbb{Z}) \)

\( D^\# = \{ [a, b] \mid a \in \mathbb{Z}, b \in \overline{\mathbb{Z}}, a \leq b \} \)

\( S = \{ 1, 3, 5 \} \)
Static Analysis by Abstract Interpretation 101

\[ \mathcal{D} = \mathcal{P}(\mathbb{Z}) \]

\[ \mathcal{D}^\# = \{ [a, b] \mid a \in \mathbb{Z}, b \in \overline{\mathbb{Z}}, a \leq b \} \]

\[ S = \{ 1, 3, 5 \} \]

\[ \alpha(S) = [1, 5] \]
**Static Analysis by Abstract Interpretation 101**

\[ \gamma \circ \alpha(S) \]

\[ \gamma \]

\[ \alpha(S) \]

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

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

\[ D = \mathcal{P}(\mathbb{Z}) \]

\[ D^\# = \{ [a, b] \mid a \in \mathbb{Z}, b \in \overline{\mathbb{Z}}, a \leq b \} \]

\[ S = \{ 1, 3, 5 \} \]

\[ \alpha(S) = [1, 5] \]

\[ \gamma \circ \alpha(S) = \{ 1, 2, 3, 4, 5 \} \]
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 could be 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 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. Contrary to C, 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? A mathematical description of the behavior of Python operators, acting like a collecting interpreter over program states.

Why? To relate static analyses with the actual program behavior, and prove that our static analyses are sound.

Issues

- No standard.
- Size: Python is a huge language.
- Some parts are not formalized yet (eval, ...).
- Correctness: concrete semantics not implemented. But we can run our analysis on CPython’s unittests.

⇒ We need a concrete semantics, but this is not our endgoal.
Semantics – Example: $e_1 + e_2$

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

\[\text{has_field}(a_1, \text{__add__})?\]

- Yes: $\text{has_field}(a_2, \text{__radd__}) \&\& \text{type}(a_1) < \text{type}(a_2)$?
  - Yes: $a_3 = \text{call } a_1\text{'s } \text{__add__} \text{ on } a_1, a_2$
    - Yes: $a_3 == \text{NotImplemented}$?
      - Yes: Type Error
      - No: $a_3 == \text{NotImplemented}$?
        - Yes: Type Error
        - No: $a_3 = \text{call } a_2\text{'s } \text{__radd__} \text{ on } a_1, a_2$
    - No: $\text{Result is } a_3$
  - No: $a_3 = \text{call } a_1\text{'s } \text{__add__} \text{ on } a_1, a_2$

- No: $\text{Result is } a_3$
Semantics – Example: $e_1 + e_2$

$$E[e_1 + e_2](f, \epsilon, \sigma) \overset{\text{def}}{=}$$

if $f \neq \text{cur}$ then $(f, \epsilon, \sigma)$ else

letif $(f, \epsilon, \sigma, a_1) = E[e_1](f, \epsilon, \sigma)$ in

letif $(f, \epsilon, \sigma, a_2) = E[e_2](f, \epsilon, \sigma)$ in

if hasattr($\sigma(a_1), \_\_\text{add}\_\_$) then

if hasattr($\sigma(a_2), \_\_\text{radd}\_\_$) $\land$ type($a_1) < \text{type}(a_2)$ then

letif $(f, \epsilon, \sigma, a_r) = E[a_2.\_\_\text{radd}\_\_(a_1)]$ in

if $\sigma(a_r) = \text{NotImpl}$ then empty _addr $\circ \text{S[ raise TypeError]}(f, \epsilon, \sigma)$

else $(f, \epsilon, \sigma, a_r)$

else letif $(f, \epsilon, \sigma, a_r) = E[a_1.\_\_\text{add}\_\_(a_2)]$ in

if $\sigma(a_r) = \text{NotImpl}$ then

if hasattr($\sigma(a_2), \_\_\text{radd}\_\_$) $\land$ type($a_1) \neq \text{type}(a_2)$ then

letif $(f, \epsilon, \sigma, a_r) = E[a_2.\_\_\text{radd}\_\_(a_1)]$ in

if $\sigma(a_r) = \text{NotImpl}$ then empty _addr $\circ \text{S[ raise TypeError]}(f, \epsilon, \sigma)$

else $(f, \epsilon, \sigma, a_r)$

else $(f, \epsilon, \sigma, a_r)$

else if hasattr($\sigma(a_2), \_\_\text{radd}\_\_$) $\land$ type($a_1) \neq \text{type}(a_2)$ then

letif $(f, \epsilon, \sigma, a_r) = E[a_2.\_\_\text{radd}\_\_(a_1)]$ in

if $\sigma(a_r) = \text{NotImpl}$ then empty _addr $\circ \text{S[ raise TypeError]}(f, \epsilon, \sigma)$

else $(f, \epsilon, \sigma, a_r)$

else empty _addr $\circ \text{S[ raise TypeError]}(f, \epsilon, \sigma)$
Type Abstract Domain
Our goal: Detect uncaught exceptions, such as TypeError, AttributeError. Have a sound analysis.
Features of the Analysis

**Our goal**: Detect uncaught exceptions, such as TypeError, AttributeError. Have a **sound** analysis.

```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
```

```python
class A:
    def __init__(self):
        self.val = 0
    def update(self, x):
        self.val = x

x = A()
c = x.val  # c: int
y = x  # x, y point to the same address
y.update('a')
z = x.val  # z: str
```

⇒ Handle addresses and aliasing.
Features of the Analysis

Our goal: Detect uncaught exceptions, such as TypeError, AttributeError. Have a sound analysis.

```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.
```
Features of the Analysis

**Our goal:** Detect uncaught exceptions, such as `TypeError`, `AttributeError`. Have a **sound** analysis.

```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.val = 0
    def update(self, x):
        self.val = x

x = A()
c = x.val # c: int
y = x
# x, y point to the same address
y.update('a')
z = x.val # z: str
```
Features of the Analysis

**Our goal:** Detect uncaught exceptions, such as `TypeError`, `AttributeError`. Have a **sound** analysis.

```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.

class A:
    def __init__(self):
        self.val = 0
    def update(self, x):
        self.val = x

x = A()
c = x.val # c: int
y = x
# x, y point to the same address
y.update('a')
z = x.val # z: str

⇒ Handle addresses and aliasing.
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
        else: raise TypeError
    else: 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
        else: raise TypeError
    else: 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
        else: raise TypeError
    else: 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
        else: raise TypeError
    else: 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
        else: raise TypeError
    else: 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
        else: raise TypeError
    else: raise TypeError

r = fspath(p)
Static Type 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
        else: raise TypeError
    else: 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
        else: raise TypeError
    else: 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
        else: raise TypeError
    else: raise TypeError

r = fspath(p)
```

```
 Classical Type Analysis Example

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
### Classical Typing vs Static Type Analysis

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

<table>
<thead>
<tr>
<th>“Classical” Typing</th>
<th>Our Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid programs may be <strong>rejected</strong></td>
<td>No restriction on the language</td>
</tr>
<tr>
<td>Type errors are <strong>fatal</strong></td>
<td>Type errors are <strong>catchable exceptions</strong></td>
</tr>
</tbody>
</table>
# Classical Typing vs Static Type Analysis

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

In addition, we can easily combine our analysis with other analyses (value analysis, ...).
## Classical Typing vs Static Type Analysis

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

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

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

In addition, we can easily combine our analysis with other analyses (value analysis, ...).
Abstract environment (nominal types)  $\mathcal{E}^\sharp = \text{Id} \rightarrow \mathcal{P}(\text{Addr}^\sharp)$
Abstract environment (nominal types) \( \mathcal{E}^\# = \text{Id} \rightarrow \mathcal{P}(\text{Addr}^\#) \)

Abstract Addresses (\texttt{Addr}^\#) have:

- A kind, representing the nominal type.
- A mode:
  - \textbf{weak}, meaning it \underline{summarizes} multiple concrete addresses (only weak updates are possible on those addresses),
  - \textbf{strong}, abstracting one concrete address.
Abstract environment (nominal types) \( \mathcal{E}^\# = \text{Id} \rightarrow \mathcal{P}(\text{Addr}^\#) \)

Abstract Addresses (\( \text{Addr}^\# \)) have:

- A kind, representing the nominal type.
- A mode:
  - **weak**, meaning it summarizes multiple concrete addresses (only weak updates are possible on those addresses),
  - **strong**, abstracting one concrete address.

We use the recency abstraction (Balakrishnan and Reps, “Recency-Abstraction for Heap-Allocated Storage”).
Attribute abstraction (structural types) \[ S^\# = \text{Addr}^\# \rightarrow \text{Attr}^\# \]
Description of the Type Abstract Domain

Attribute abstraction (structural types)  \( S^\# = \text{Addr}^\# \rightarrow \text{Attr}^\# \)

Attribute under/over-approximation  \( \text{Attr}^\# = \mathcal{P}(\text{string})^2 \)

\[ \gamma^\#_{\text{Attr}} : \begin{cases} \text{Attr}^\# & \rightarrow & \mathcal{P}(\mathcal{P}(\text{string})) \\ (u, o) & \mapsto & \{ s \mid u \subseteq s \subseteq o \} \end{cases} \]

\[ \gamma^\#_{\text{Attr}}(\{ a \}, \{ a, b, c \}) = \{ \{ a \}, \{ a, b \}, \{ a, c \}, \{ a, b, c \} \} \]
Abstract environment (nominal types)

\[ \mathcal{E}^\# = \text{Id} \to \mathcal{P}(\text{Addr}^\#) \]

Attribute abstraction (structural types)

\[ \mathcal{S}^\# = \text{Addr}^\# \to \text{Attr}^\# \]

\[ \mathcal{S}^\#[x = e](\epsilon, \sigma) \overset{\text{def}}{=} \]

\[ \begin{align*}
\text{let Obj } @, (\epsilon, \sigma) &= \mathcal{E}^\#[e](\epsilon, \sigma) \text{ in } \\
\epsilon[x \mapsto \{ @ \}], \sigma
\end{align*} \]
Abstract environment (nominal types)

\[ \mathcal{E}^\# = \text{Id} \to \mathcal{P}(\text{Addr}^\#) \]

Attribute abstraction (structural types)

\[ S^\# = \text{Addr}^\# \to \text{Attr}^\# \]

\[ S^\#[x = e] (\epsilon,\sigma) \overset{\text{def}}{=} \]

\[ \text{let Obj @} (\epsilon,\sigma) = \mathcal{E}^\#[e] (\epsilon,\sigma) \text{ in} \]

\[ \epsilon[x \mapsto \{ @}],\sigma \]

\[ \mathcal{E}^\#[\text{object.__new__}(e)] (\epsilon,\sigma) \overset{\text{def}}{=} \]

\[ \text{let ee,}(\epsilon,\sigma) = \mathcal{E}^\#[e] (\epsilon,\sigma) \text{ in} \]

\[ \text{if ee = Class } c \text{ then} \]

\[ \text{let @} (\epsilon,\sigma) = \mathcal{E}^\#[\text{alloc(Inst } c)] (\epsilon,\sigma) \text{ in Obj @} (\epsilon,\sigma) \]

\[ \text{else empty_eval} \circ S^\#[\text{raise TypeError}] (\epsilon,\sigma) \]
Description of the Type Abstract Domain

Abstract environment (nominal types) \[\mathcal{E}^\# = \text{Id} \to \mathcal{P}(\text{Addr}^\#)\]
Attribute abstraction (structural types) \[\mathcal{S}^\# = \text{Addr}^\# \to \text{Attr}^\#\]

\[\mathcal{S}^\#[\text{object.__setattribute__}(x \in \text{Id}, attr \in \text{string}, e)](\epsilon, \sigma) \overset{\text{def}}{=}\]

\[
\begin{align*}
&\text{let Obj @}x,(\epsilon,\sigma) = \mathcal{E}^\#[x](\epsilon,\sigma) \text{ in} \\
&\text{let attr_var = Var @}x,attr \text{ in} \\
&\text{if strong_addr @}x \text{ then} \\
&\quad \mathcal{S}^\#[\text{attr_var} = e](\epsilon,\text{add_under}(\sigma,@x,attr)) \\
&\text{else} \\
&\quad \mathcal{S}^\#[\text{attr_var} \overset{\text{weak}}{=} e](\epsilon,\text{add_over}(\sigma,@x,attr))
\end{align*}
\]
Description of the Type Abstract Domain – Polymorphism

```python
1  def get_sep(s):
2      if isinstance(s, str): return '/'
3      elif isinstance(s, bytes): return b'/'
4      else: raise TypeError
5
6      if *: r = '/dev/null'
7      else: r = b'/dev/null'
8
9      sep = get_sep(r)

\[
\begin{cases}
  r \in \{ @str \} \\
  sep \in \{ @str \}
\end{cases}
\]

if branch
```

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)

\[
\begin{cases}
    r \in \{ \text{str} \} \\
    \text{sep} \in \{ \text{str} \}
\end{cases}
\quad \text{if branch}
\]

\[
\begin{cases}
    r \in \{ \text{bytes} \} \\
    \text{sep} \in \{ \text{bytes} \}
\end{cases}
\quad \text{else branch}
\]

Then, we can analyze res = r + sep without false alarms. The polymorphism improves the precision.
Description of the Type Abstract Domain – Polymorphism

```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)
```

\[
\begin{aligned}
\left\{ r \in \{ \text{str} \} \right\} & \quad \sqcup \quad \left\{ r \in \{ \text{bytes} \} \right\} \\
sep \in \{ \text{str} \} & \quad \left\{ sep \in \{ \text{str} \} \right\} \\
sep \in \{ \text{bytes} \} & \quad \left\{ sep \in \{ \text{bytes} \} \right\}
\end{aligned}
\]

if branch else branch

Then, we can analyze res = r + sep without false alarms. The polymorphism improves the precision.
Description of the Type Abstract Domain – Polymorphism

```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)
```

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

Then, we can analyze \( r + sep \) without false alarms. The polymorphism improves the precision.
Description of the Type Abstract Domain – Polymorphism

```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)
```

\[
\begin{align*}
\begin{cases}
    r \in \{ \texttt{str} \} \\
    \text{sep} \in \{ \texttt{str} \}
\end{cases}
\quad \sqcup 
\begin{cases}
    r \in \{ \texttt{bytes} \} \\
    \text{sep} \in \{ \texttt{bytes} \}
\end{cases} =
\begin{cases}
    r \in \{ \texttt{str}, \texttt{bytes} \} \\
    \text{sep} \in \{ \texttt{str}, \texttt{bytes} \}
\end{cases} \land r \equiv \text{sep}
\end{align*}
\]
Description of the Type Abstract Domain – Polymorphism

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

5 if *: r = '/dev/null'
6 else: r = b'/dev/null'

7 sep = get_sep(r)
```

\[
\begin{cases}
    r \in \{ \texttt{@str} \} \\
    sep \in \{ \texttt{@str} \}
\end{cases}
\quad \sqcup 
\begin{cases}
    r \in \{ \texttt{@bytes} \} \\
    sep \in \{ \texttt{@bytes} \}
\end{cases}
= \begin{cases}
    r \in \{ \texttt{@str, @bytes} \} \\
    sep \in \{ \texttt{@str, @bytes} \}
\end{cases}
\wedge r \equiv sep
\]

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

Ideas:

- Smash each list into one weak, abstract contents variable.
- The contents variable is built upon the list’s abstract address.
- Delegate most of the work to the other domains.
A Modular List Domain

Ideas:

- Smash each list into one weak, abstract contents variable.
- The contents variable is built upon the list’s abstract address.
- Delegate most of the work to the other domains.

\[ E\#[[e_1, \ldots, e_n]^{loc}] s \overset{\text{def}}{=} \]

let  \( \#, s = E\#[[\text{alloc} (\text{List} \ loc)] \ s \text{ in} \]

let  contents = Var(\#, "contents") \ in \]

let  s = S\#[[ contents \overset{\text{weak}}{=} e_n ] \circ \ldots \circ S\#[[ contents \overset{\text{weak}}{=} e_1 ]] s \text{ in}\]

Obj  \( \#, s \)
Lists and Polymorphisms

1  if *: l = [1, 2, 3]
2  else: l = [0.1, 0.2, 0.3]
3  x = l[0]

After line 1:
\[
\begin{align*}
  l & \mapsto \{ \texttt{@list1} \} \\
  \texttt{@list1}.content & \mapsto \{ \texttt{@int} \}
\end{align*}
\]

After line 2:
\[
\begin{align*}
  l & \mapsto \{ \texttt{@list2} \} \\
  \texttt{@list2}.content & \mapsto \{ \texttt{@float} \}
\end{align*}
\]
Lists and Polymorphisms

1 \textbf{if} *: \texttt{l = [1, 2, 3]}
2 \textbf{else}: \texttt{l = [0.1, 0.2, 0.3]}
3 \texttt{x = l[0]}

After line 1:
\( l \mapsto \{ \text{@list1} \} \)
\( \text{@list1.content} \mapsto \{ \text{@int} \} \)

After line 2:
\( l \mapsto \{ \text{@list2} \} \)
\( \text{@list2.content} \mapsto \{ \text{@float} \} \)

After line 3 (join):
\[
\begin{cases}
\{ \text{@list1, @list2} \} \\
\text{@list1.content} \mapsto \{ \text{@int} \} \\
\text{@list2.content} \mapsto \{ \text{@float} \} \\
\text{x} \mapsto \{ \text{@int, @float} \}
\end{cases}
\]
Lists and Polymorphisms

```
1 if *: l = [1, 2, 3]
2 else: l = [0.1, 0.2, 0.3]
3 x = l[0]
```

After line 1:

\[
\begin{align*}
l &\mapsto \{ \text{@list1} \} \\
\text{@list1}.\text{content} &\mapsto \{ \text{@int} \}
\end{align*}
\]

After line 2:

\[
\begin{align*}
l &\mapsto \{ \text{@list2} \} \\
\text{@list2}.\text{content} &\mapsto \{ \text{@float} \}
\end{align*}
\]

After line 3 (join):

\[
\begin{align*}
l &\mapsto \{ \text{@list1}, \text{@list2} \} \\
\text{@list1}.\text{content} &\mapsto \{ \text{@int} \} \\
\text{@list2}.\text{content} &\mapsto \{ \text{@float} \} \\
x &\mapsto \{ \text{@int}, \text{@float} \}
\end{align*}
\]

After unification and join:

\[
\begin{align*}
l &\mapsto \{ \text{@list\{1,2\}} \} \\
\text{@list\{1,2\}}.\text{content} &\mapsto \{ \text{@int}, \text{@float} \} \\
x &\mapsto \{ \text{@int}, \text{@float} \} \\
x &\equiv \text{@list\{1,2\}}.\text{content}
\end{align*}
\]
Interprocedural Analysis

What about a context-insensitive analysis?

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

$f$ is valid whenever the evaluation hits the green box:

```
a_1 = eval e_1; a_2 = eval e_2
```

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

```
has_field(a_2, __radd__) & type(a_1) \neq type(a_2)
```

```
a_3 = call a_2's __radd__ on a_1, a_2
```

```
a_3 == NotImplemented?
```

```
a_3 == NotImplemented?
```

```
result is a_3
```

```
Type Error
```

Inlining most precise, but costly.

Towards function summaries a simple cache keeping the previous function analyses achieves up to 25x speedup on our benchmarks.
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?

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

⇒ We focus on a context-sensitive analysis.

**Inlining** most precise, but costly.
Interprocedural Analysis

What about a context-insensitive analysis?

```python
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 25x speedup on our benchmarks.
Experimental Evaluation
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 into MOPSA

- Py.program
- Py.desugar
- Py.exceptions
- U.intraproc
- U.loops
- U.interproc
- Py.libraries
- Py.objects
- Py.data_model

- U.recency_abstraction

- Sequence
- Reduced product
- Cartesian product

- Py.type_domain
- Py.polymorphism

- Py.lists
- Py dicts

- Py tuples

Universal
Python specific
### Official Python Benchmarks:

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time (inlining)</th>
<th>Time (cache)</th>
<th># Alarms</th>
<th># False Alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>fannkuch.py</td>
<td>59</td>
<td>0.07s</td>
<td>0.07s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>float.py</td>
<td>63</td>
<td>0.10s</td>
<td>0.06s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>spectral_n.py</td>
<td>74</td>
<td>3.9 s</td>
<td>0.33s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>2.6s</td>
<td>1.5s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>chaos.py</td>
<td>324</td>
<td>19s</td>
<td>5.9s</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>5.6s</td>
<td>5.4s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>61.7m</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 (inlining)</th>
<th>Time (cache)</th>
<th># Alarms</th>
<th># False Alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>fannkuch.py</td>
<td>59</td>
<td>0.07s</td>
<td>0.07s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>float.py</td>
<td>63</td>
<td>0.10s</td>
<td>0.06s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>spectral_n.py</td>
<td>74</td>
<td>3.9 s</td>
<td>0.33s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nbody.py</td>
<td>157</td>
<td>2.6s</td>
<td>1.5s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>chaos.py</td>
<td>324</td>
<td>19s</td>
<td>5.9s</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>unpack_seq.py</td>
<td>458</td>
<td>5.6s</td>
<td>5.4s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hexiom.py</td>
<td>674</td>
<td>61.7m</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.x, but not in Python 3.x.
Related Work
**Related Work**

**JavaScript:** more work has been done on JavaScript, including:

- Bodin et al.: semantics in Coq,

**Dynamic Analysis:** Pyannotate and MonkeyType collect the types of a program at runtime. The types are valid for the explored trace.

**Gradual Typing:** annotated program parts are typechecked, while other parts have an unknown “top” type, from which any static type can be cast to and from. In Python: Mypy, Pyre.
Related Work II

Static Analysis of Python

- Fromherz et al\(^3\): a value analysis for Python (more costly). We have started a modular implementation of this analysis using our type domain.
- Typpete\(^4\): encodes type inference into a MaxSMT instance.
- Fritz & Hage\(^5\): performs a dataflow analysis.
- Pytype, tool from Google, performing a dataflow analysis.

\(^3\)Fromherz, Ouadjaout, and Miné. “Static Value Analysis of Python Programs by Abstract Interpretation”. NFM 2018 Proceedings.

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

Related Work II

Static Analysis of Python

- Fromherz et al\(^3\): a value analysis for Python (more costly). We have started a modular implementation of this analysis using our type domain.
- Typpete\(^4\): encodes type inference into a MaxSMT instance.
- Fritz & Hage\(^5\): performs a dataflow analysis.
- Pytype, tool from Google, performing a dataflow analysis.

Our type analysis uniquely takes into account object mutability and control-flow.

\(^3\)Fromherz, Ouadjaout, and Miné. “Static Value Analysis of Python Programs by Abstract Interpretation”. NFM 2018 Proceedings.

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

Conclusion
Conclusion & Future Work

We have developed a static type analysis for Python.

▶ More precise than the state-of-the-art static type analyses.
▶ Analyzes real-world benchmarks!
▶ Sound: it relates the analysis and the concrete semantics.

Future Work

▶ Support official type annotations (PEP 484).
▶ More efficient, summary-based function analysis.
▶ Handle libraries.
▶ Analyze real-world programs.
Appendix
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.
Flow-sensitive analysis

**Flow-sensitive Analysis** Performed by induction over the syntax. Using flow tokens to label continuation-passed states.

\[ F^\# = \{ \text{cur, ret, brk, exn a, } a \in \text{Addr}^\# \} \]

- Abstract environment (nominal types) \[ E^\# = \text{Id} \rightarrow \mathcal{P}(\text{Addr}^\#) \]
- Attribute abstraction (structural types) \[ S^\# = \text{Addr}^\# \rightarrow \text{Attr}^\# \]

\[ E^\#[x](f \in F^\#, \epsilon \in E^\#, \sigma \in S^\#) \overset{\text{def}}{=} \bigcup_{a^\# \in \epsilon(x)} \text{Obj } a^\#, (f, \epsilon[x \mapsto \{ a^\# \}], \sigma) \]

\[ S^\#[x = e](f \in F^\#, \epsilon \in E^\#, \sigma \in S^\#) \overset{\text{def}}{=} \]

\[ \text{letif } (\text{Obj } \emptyset, (f, \epsilon, \sigma)) = E^\#[e](f, \epsilon, \sigma) \quad \text{in} \]

\[ (f, \epsilon[x \mapsto \emptyset], \sigma) \]
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)

\[
\begin{align*}
    r & \in \{ @_{str}^w \} \\
    sep & \in \{ @_{str}^s \}
\end{align*}
\]\n
if branch

Then, we can analyze 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)

\[
\begin{align*}
\left\{ r \in \{ \mathcal{O}_w^{str} \} \right\} \\
sep & \in \{ \mathcal{O}_s^{str} \}\quad \text{if branch} \\
\left\{ r \in \{ \mathcal{O}_w^{str} \} \right\} \\
sep & \in \{ \mathcal{O}_s^{str} \}\quad \text{else branch}
\end{align*}
\]
Description of the Type Abstract Domain – Polymorphism

```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)
```

\[
\begin{align*}
\begin{cases}
    r \in \{ \mathbb{A}_{str}^w \} \\
    sep \in \{ \mathbb{A}_{str}^s \}
\end{cases}
\quad \text{if branch}
\end{align*}
\]

\[
\begin{align*}
\begin{cases}
    r \in \{ \mathbb{A}_{str}^w \} \\
    sep \in \{ \mathbb{A}_{str}^s \}
\end{cases}
\quad \text{else branch}
\end{align*}
\]

Then, we can analyze \( r + sep \) 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)

\[
\begin{align*}
    & r \in \{ \emptyset^w_{\text{str}} \} \\
    & sep \in \{ \emptyset^s_{\text{str}} \} \\
\end{align*}
\]  

\[
\begin{align*}
    & r \in \{ \emptyset^w_{\text{str}} \} \\
    & sep \in \{ \emptyset^s_{\text{str}} \} \\
\end{align*}
\]  

\[
\begin{align*}
    & r \in \{ \emptyset^w_{\text{str}}, \emptyset^w_{\text{bytes}} \} \\
    & sep \in \{ \emptyset^s_{\text{str}}, \emptyset^s_{\text{bytes}} \}
\end{align*}
\]  

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

The polymorphism improves the precision.
Description of the Type Abstract Domain – Polymorphism

```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)
```

\[
\begin{cases}
    r \in \{ @^w_{str} \} \\
    sep \in \{ @^s_{str} \}
\end{cases}
\quad \sqcup \quad
\begin{cases}
    r \in \{ @^w_{str} \} \\
    sep \in \{ @^s_{str} \}
\end{cases}
= \begin{cases}
    r \in \{ @^w_{str}, @^w_{bytes} \} \\
    sep \in \{ @^s_{str}, @^s_{bytes} \}
\end{cases} \quad \land \quad r \equiv sep
\]
Description of the Type Abstract Domain – Polymorphism

```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 \( \text{res} = r + \text{sep} \) without false alarms. The polymorphism improves the precision.