Static Analysis of Python Programs

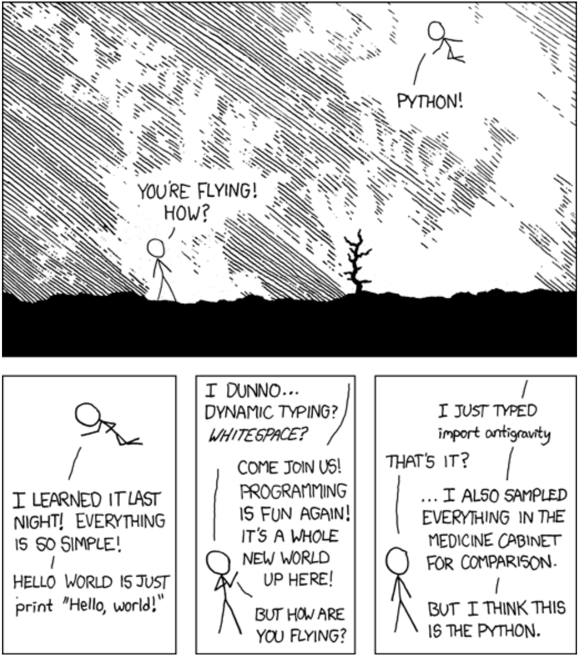
A Type Abstract Domain for Python

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

7th October 2019

European Research Council
Established by the European Commission

Introduction


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

xkcd.com/353/

Python is a Dynamic Programming Language

It features:

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

Python Example from the “os” Library

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

Python Example from the “os” Library

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
   else:
     raise TypeError("...")
 else:
   raise TypeError("...")
```

Python Example from the "os" Library

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("...")
```

Python Example from the “os” Library

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

Two notions of typing:

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

Python Example from the “os” Library

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

Two notions of typing:

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

Type of fspath?

Python Example from the “os” Library

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

Two notions of typing:

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

Type of fspath?

str $\rightarrow$ str; bytes $\rightarrow$ 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 <u>all</u> runtime errors,
- At "compile time", before the program execution.

A theoretical hurdle: Rice's theorem

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

$\Longrightarrow$ We will compute <u>approximate</u> 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".

$\Longrightarrow$ We will compute approximate results;

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

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".

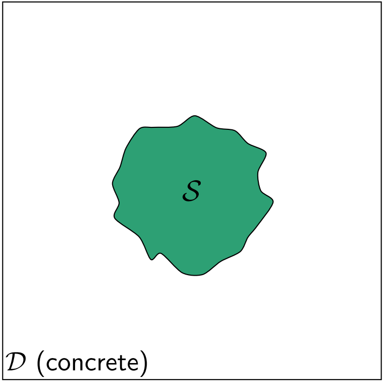
$\Longrightarrow$ We will compute approximate results;

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

Goals

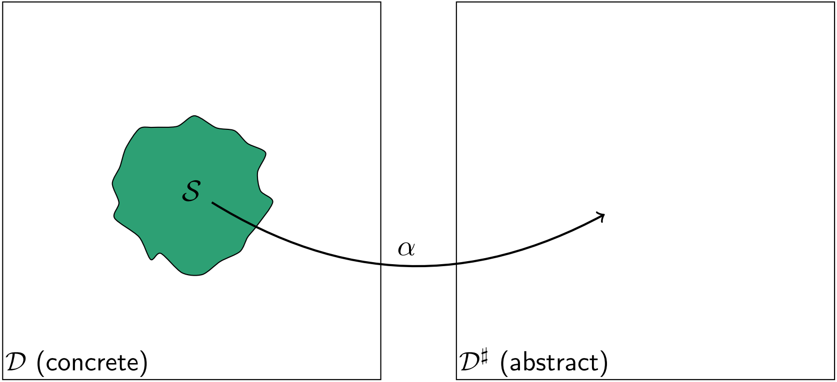
- Automatic analysis: no expert knowledge required.
- Sound analysis: if no bug is detected, none will occur.

Static Analysis by Abstract Interpretation 101

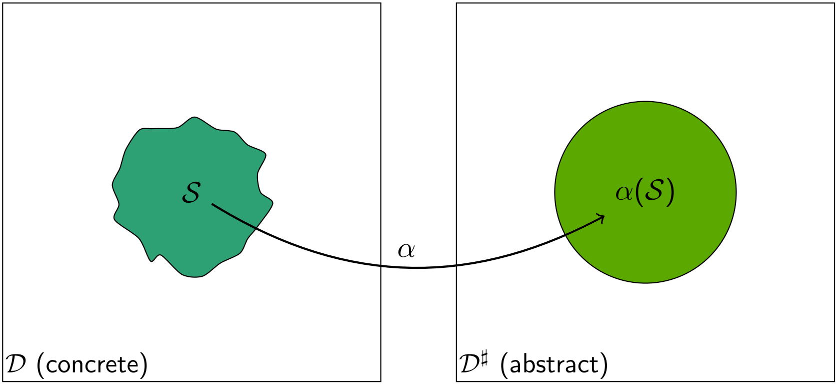


$\mathcal{D}^\sharp$ (abstract)

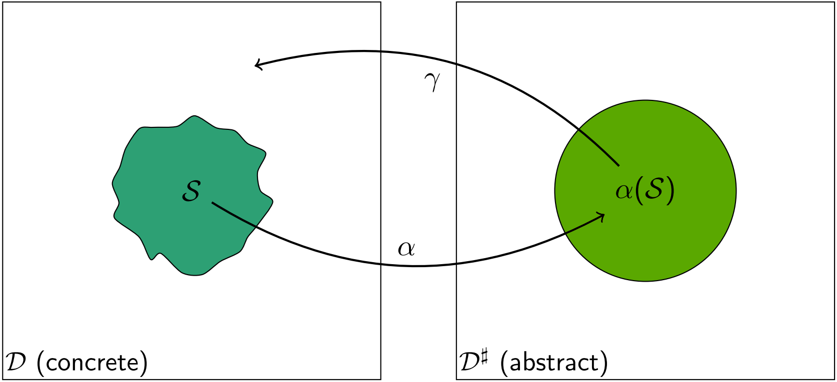
Static Analysis by Abstract Interpretation 101



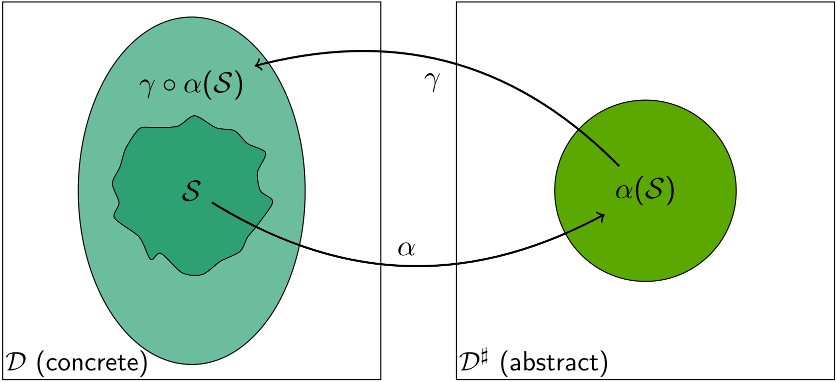
Static Analysis by Abstract Interpretation 101



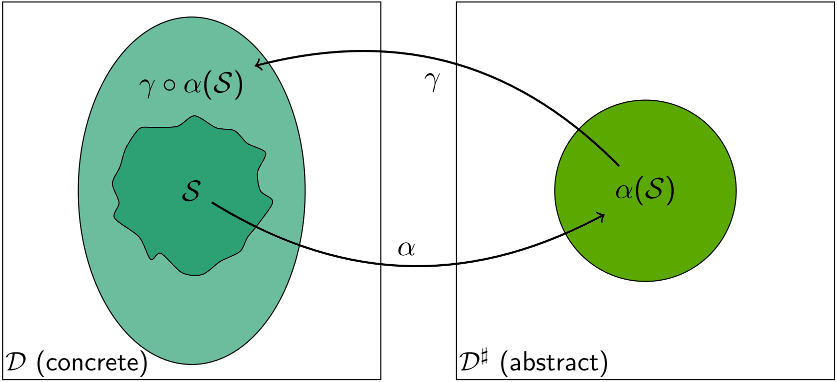
Static Analysis by Abstract Interpretation 101



Static Analysis by Abstract Interpretation 101

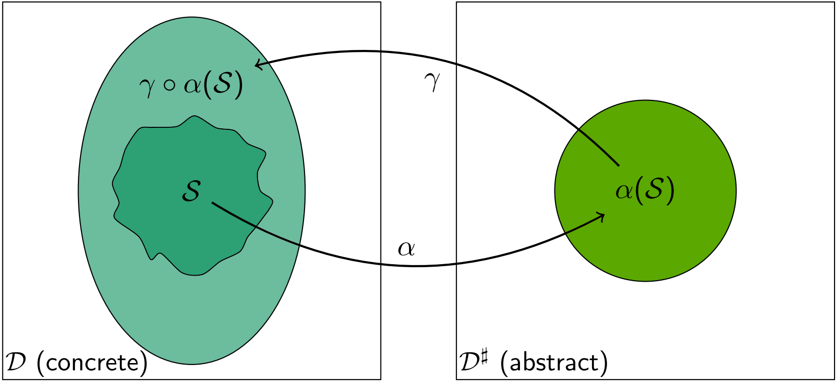


Static Analysis by Abstract Interpretation 101



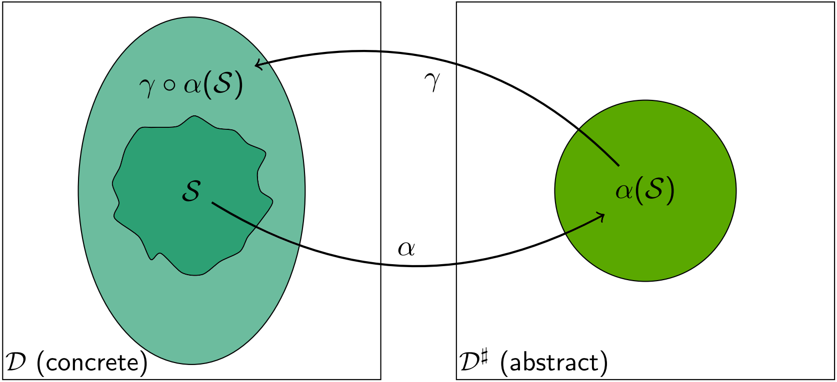
- $\mathcal{D} = \mathcal{P}(\mathbb{Z})$

Static Analysis by Abstract Interpretation 101

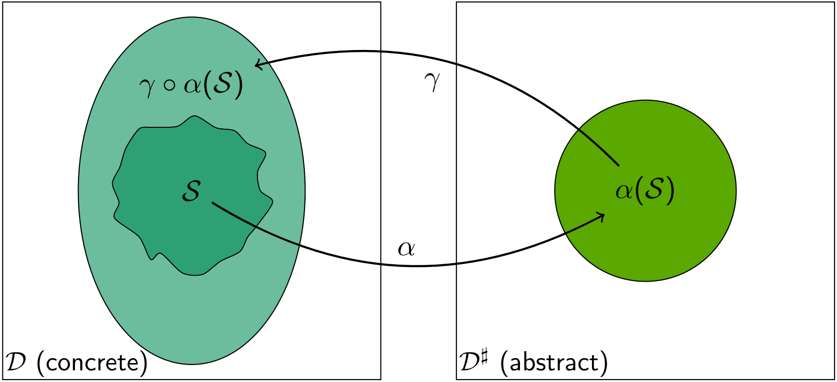


- $\mathcal{D} = \mathcal{P}(\mathbb{Z})$
- $\mathcal{D}^\sharp = \{\, [a, b] \mid a \in \underline{\mathbb{Z}}, b \in \overline{\mathbb{Z}}, a \leq b \,\}$

Static Analysis by Abstract Interpretation 101

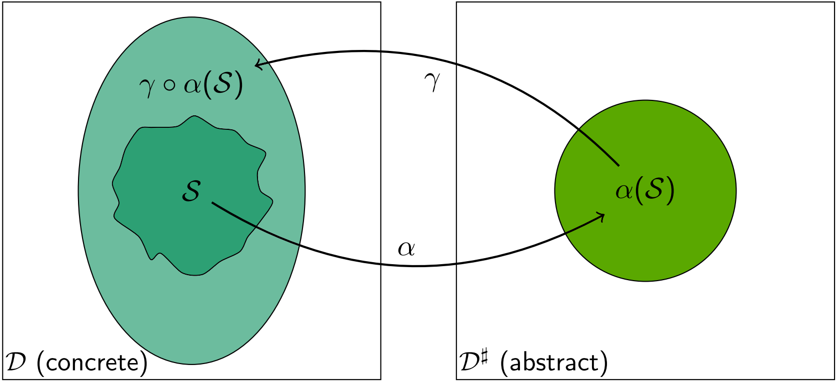


- $\mathcal{D} = \mathcal{P}(\mathbb{Z})$
- $\mathcal{D}^\sharp = \{ [a, b] \mid a \in \underline{\mathbb{Z}}, b \in \overline{\mathbb{Z}}, a \leq b \}$
- $\mathcal{S} = \{ 1, 3, 5 \}$



- $\mathcal{D} = \mathcal{P}(\mathbb{Z})$
- $\mathcal{D}^\sharp = \{\, [a, b] \mid a \in \underline{\mathbb{Z}}, b \in \overline{\mathbb{Z}}, a \leq b \,\}$
- $\mathcal{S} = \{\, 1, 3, 5 \,\}$
- $\alpha(\mathcal{S}) = [1, 5]$

Static Analysis by Abstract Interpretation 101



- $\mathcal{D} = \mathcal{P}(\mathbb{Z})$
- $\mathcal{D}^\sharp = \{\, [a, b] \mid a \in \underline{\mathbb{Z}}, b \in \overline{\mathbb{Z}}, a \leq b \,\}$
- $\mathcal{S} = \{\, 1, 3, 5 \,\}$
- $\alpha(\mathcal{S}) = [1, 5]$
- $\gamma \circ \alpha(\mathcal{S}) = \{\, 1, 2, 3, 4, 5 \,\}$

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

Concrete Semantics – Motivation

Semantics? A mathematical description of the behavior of Python operators, acting like a collecting interpreter over program states.

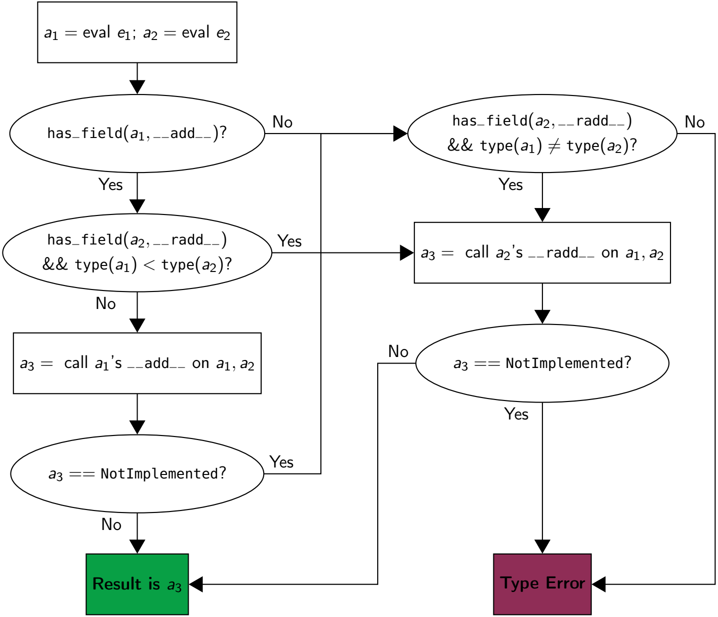
Why? To relate static analyses with the actual program behavior, and prove that our static analyses are sound.

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.

$\Longrightarrow$ We need a concrete semantics, but this is not our endgoal.

Semantics – Example: $e_1 + e_2$



Semantics – Example: $e_1 + e_2$

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

Type Abstract Domain

Features of the Analysis

Our goal: Detect uncaught exceptions, such as `TypeError`, `AttributeError`. Have a <u>sound</u> analysis.

Features of the Analysis

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

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

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

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

$\implies$ Flow-sensitive analysis including exceptions.

Features of the Analysis

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

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

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

$\implies$ 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
```

Features of the Analysis

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

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

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

$\implies$ 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
```

$\implies$ Handle addresses and aliasing.

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)
```

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)
```

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)
```

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

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)
```

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

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)
```

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

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

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)
```

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

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

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)
```

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

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

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

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)
```

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

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

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

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)
```

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

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

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

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

Classical Typing vs Static Type Analysis

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

Classical Typing vs Static Type Analysis

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

Classical Typing vs Static Type Analysis

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

Classical Typing vs Static Type Analysis

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

Classical Typing vs Static Type Analysis

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

Classical Typing vs Static Type Analysis

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

Classical Typing vs Static Type Analysis

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

In addition, we can easily combine our analysis
with other analyses (value analysis, . . .).

Description of the Type Abstract Domain

Abstract environment (nominal types)	$\mathcal{E}^\sharp = \textbf{Id} \to \mathcal{P}(\textbf{Addr}^\sharp)$

Description of the Type Abstract Domain

Abstract environment (nominal types)	$\mathcal{E}^\sharp = \mathbf{Id} \to \mathcal{P}(\mathbf{Addr}^\sharp)$

Abstract Addresses ($\mathbf{Addr}^\sharp$) 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.

Description of the Type Abstract Domain

$\boxed{\text{Abstract environment (nominal types)} \qquad \mathcal{E}^\sharp = \mathbf{Id} \to \mathcal{P}(\mathbf{Addr}^\sharp)}$

Abstract Addresses ($\mathbf{Addr}^\sharp$) have:

- A kind, representing the nominal type.
- A mode:
  - **weak**, meaning it <u>summarizes</u> 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").

Description of the Type Abstract Domain

$$\boxed{\text{Attribute abstraction (structural types)} \qquad \mathcal{S}^\sharp = \textbf{Addr}^\sharp \rightarrow \textbf{Attr}^\sharp}$$

Attribute abstraction (structural types) $\mathcal{S}^\sharp = \textbf{Addr}^\sharp \to \textbf{Attr}^\sharp$

Attribute under/over-approximation $\textbf{Attr}^\sharp = \mathcal{P}(\mathit{string})^2$

$$\gamma^\sharp_{\textbf{Attr}} : \begin{cases} \textbf{Attr}^\sharp & \to & \mathcal{P}(\mathcal{P}(\mathit{string})) \\ (u, o) & \mapsto & \{\, s \mid u \subseteq s \subseteq o \,\} \end{cases}$$

$$\gamma^\sharp_{\textbf{Attr}}(\{\, a \,\}, \{\, a, b, c \,\}) = \{\, \{\, a \,\}, \{\, a, b \,\}, \{\, a, c \,\}, \{\, a, b, c \,\} \,\}$$

Description of the Type Abstract Domain

Abstract environment (nominal types) $\mathcal{E}^\sharp = \textbf{Id} \rightarrow \mathcal{P}(\textbf{Addr}^\sharp)$

Attribute abstraction (structural types) $\mathcal{S}^\sharp = \textbf{Addr}^\sharp \rightarrow \textbf{Attr}^\sharp$

$$\mathbb{S}^\sharp[\![\, x = e \,]\!]\,(\epsilon \in \mathcal{E}^\sharp, \sigma \in \mathcal{S}^\sharp) \stackrel{\text{def}}{=}$$

$$\text{let } \textbf{Obj } @, (\epsilon, \sigma) = \mathbb{E}^\sharp[\![\, e \,]\!]\,(\epsilon, \sigma) \text{ in}$$

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

Description of the Type Abstract Domain

Abstract environment (nominal types) $\mathcal{E}^\sharp = \mathbf{Id} \to \mathcal{P}(\mathbf{Addr}^\sharp)$

Attribute abstraction (structural types) $\mathcal{S}^\sharp = \mathbf{Addr}^\sharp \to \mathbf{Attr}^\sharp$

$$\mathbb{S}^\sharp[\![\, x = e \,]\!]\, (\epsilon \in \mathcal{E}^\sharp, \sigma \in \mathcal{S}^\sharp) \stackrel{\text{def}}{=}$$

$$\quad \text{let } \mathbf{Obj}\ \texttt{@}, (\epsilon, \sigma) = \mathbb{E}^\sharp[\![\, e \,]\!]\, (\epsilon, \sigma) \text{ in}$$

$$\quad \epsilon[x \mapsto \{\, \texttt{@} \,\}], \sigma$$

$$\mathbb{E}^\sharp[\![\, \texttt{object.\_\_new\_\_}(e) \,]\!]\, (\epsilon, \sigma) \stackrel{\text{def}}{=}$$

$$\quad \text{let } ee, (\epsilon, \sigma) = \mathbb{E}^\sharp[\![\, e \,]\!]\, (\epsilon, \sigma) \text{ in}$$

$$\quad \text{if } ee = \mathbf{Class}\ c \text{ then}$$

$$\qquad \text{let } \texttt{@}, (\epsilon, \sigma) = \mathbb{E}^\sharp[\![\, \text{alloc}(\mathbf{Inst}\ c) \,]\!]\, (\epsilon, \sigma) \text{ in } \mathbf{Obj}\ \texttt{@},\ (\epsilon, \sigma)$$

$$\quad \text{else empty\_eval} \circ \mathbb{S}^\sharp[\![\, \texttt{raise TypeError} \,]\!](\epsilon, \sigma)$$

Description of the Type Abstract Domain

Abstract environment (nominal types) $\mathcal{E}^\sharp = \textbf{Id} \rightarrow \mathcal{P}(\textbf{Addr}^\sharp)$

Attribute abstraction (structural types) $\mathcal{S}^\sharp = \textbf{Addr}^\sharp \rightarrow \textbf{Attr}^\sharp$

$$\mathbb{S}^\sharp [\![\, \texttt{object.\_\_setattribute\_\_}(x \in \textbf{Id}, attr \in string, e) \,]\!] (\epsilon, \sigma) \stackrel{\text{def}}{=}$$

$$\text{let } \textbf{Obj } @_x, (\epsilon, \sigma) = \mathbb{E}^\sharp [\![\, x \,]\!] (\epsilon, \sigma) \text{ in}$$

$$\text{let attr\_var} = \text{Var } (@_x, \text{attr}) \text{ in}$$

$$\text{if strong\_addr } @_x \text{ then}$$

$$\mathbb{S}^\sharp [\![\, \text{attr\_var} = e \,]\!] (\epsilon, \text{add\_under}(\sigma, @_x, \text{attr}))$$

$$\text{else}$$

$$\mathbb{S}^\sharp [\![\, \text{attr\_var} \stackrel{\text{weak}}{=} e \,]\!] (\epsilon, \text{add\_over}(\sigma, @_x, \text{attr}))$$

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)
```

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

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)
```

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

Description of the Type Abstract Domain – Polymorphism

```
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)
```

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

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)
```

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

Description of the Type Abstract Domain – Polymorphism

```
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)
```

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

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)
```

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

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.

$$\mathbb{E}^\sharp[\![\, [e_1, \ldots, e_n]^{loc} \,]\!]\, s \stackrel{\text{def}}{=}$$

$$\textsf{let } @, \mathsf{s} = \mathbb{E}^\sharp[\![\, \mathsf{alloc}(\textbf{List } \mathit{loc}) \,]\!]\, s \textsf{ in}$$

$$\textsf{let } \mathsf{contents} = \mathsf{Var}(@, \texttt{"contents"}) \textsf{ in}$$

$$\textsf{let } \mathsf{s} = \mathbb{S}^\sharp[\![\, \mathsf{contents} \stackrel{\text{weak}}{=} e_n \,]\!] \circ \ldots \circ \mathbb{S}^\sharp[\![\, \mathsf{contents} \stackrel{\text{weak}}{=} e_1 \,]\!] s \textsf{ in}$$

$$\textbf{Obj } @, s$$

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:

$l \mapsto \{ @_{list1} \}$

$@_{list1}.\text{content} \mapsto \{ @_{int} \}$

After line 2:

$l \mapsto \{ @_{list2} \}$

$@_{list2}.\text{content} \mapsto \{ @_{float} \}$

Lists and Polymorphisms

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

After line 1:

$$l \mapsto \{ @_{list1} \}$$

$$@_{list1}.\text{content} \mapsto \{ @_{int} \}$$

After line 2:

$$l \mapsto \{ @_{list2} \}$$

$$@_{list2}.\text{content} \mapsto \{ @_{float} \}$$

After line 3 (join):

$$\begin{cases} l \mapsto \{ @_{list1}, @_{list2} \} \\ @_{list1}.\text{content} \mapsto \{ @_{int} \} \\ @_{list2}.\text{content} \mapsto \{ @_{float} \} \\ x \mapsto \{ @_{int}, @_{float} \} \end{cases}$$

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:

$$l \mapsto \{\, @_{list1} \,\}$$

$$@_{list1}.\text{content} \mapsto \{\, @_{int} \,\}$$

After line 2:

$$l \mapsto \{\, @_{list2} \,\}$$

$$@_{list2}.\text{content} \mapsto \{\, @_{float} \,\}$$

After line 3 (join):

$$\begin{cases} l \mapsto \{\, @_{list1}, @_{list2} \,\} \\ @_{list1}.\text{content} \mapsto \{\, @_{int} \,\} \\ @_{list2}.\text{content} \mapsto \{\, @_{float} \,\} \\ x \mapsto \{\, @_{int}, @_{float} \,\} \end{cases}$$

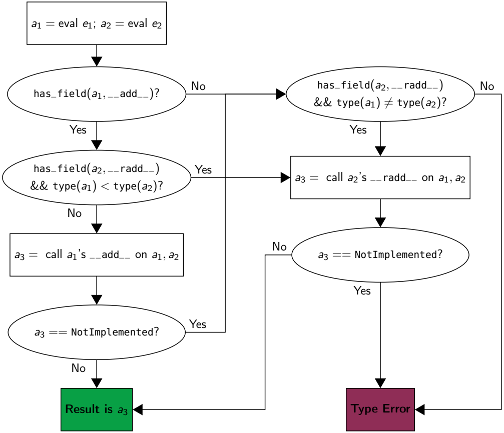
After unification and join:

$$\begin{cases} l \mapsto \{\, @_{list\{1,2\}} \,\} \\ @_{list\{1,2\}}.\text{content} \mapsto \{\, @_{int}, @_{float} \,\} \\ x \mapsto \{\, @_{int}, @_{float} \,\} \\ x \equiv @_{list\{1,2\}}.\text{content} \end{cases}$$

What about a context-insensitive analysis?

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

f is valid whenever the evaluation hits the green box:



What about a context-insensitive analysis?

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

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

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.

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

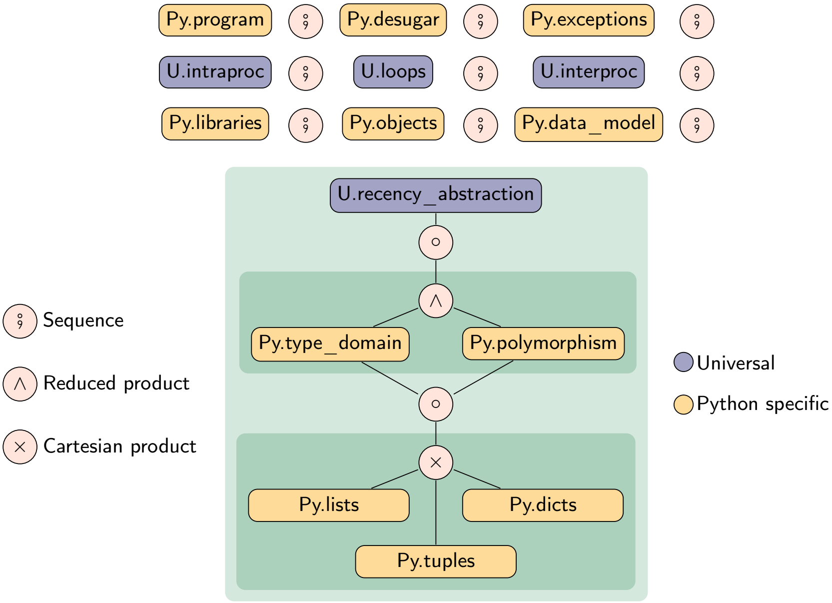
Experimental Evaluation

Modular **O**pen **P**latform for **S**tatic **A**nalysis[1]

- 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).

[1] Journault et al. "Combinations of reusable abstract domains for a multilingual static analyzer". VSTTE19 Proceedings.

Implementation into MOPSA



Benchmarks

Official Python Benchmarks:

Name	LOC	Time (inlining)	Time (cache)	# Alarms	# False Alarms
fannkuch.py	59	0.07s	0.07s	0	0
float.py	63	0.10s	0.06s	0	0
spectral_n.py	74	3.9 s	0.33s	0	1
nbody.py	157	2.6s	1.5s	0	1
chaos.py	324	19s	5.9s	**1**	0
unpack_seq.py	458	5.6s	5.4s	0	0
hexiom.py	674	61.7m	2.2m	0	52

Benchmarks

Official Python Benchmarks:

Name	LOC	Time (inlining)	Time (cache)	# Alarms	# False Alarms
fannkuch.py	59	0.07s	0.07s	0	0
float.py	63	0.10s	0.06s	0	0
spectral_n.py	74	3.9 s	0.33s	0	1
nbody.py	157	2.6s	1.5s	0	1
chaos.py	324	19s	5.9s	**1**[2]	0
unpack_seq.py	458	5.6s	5.4s	0	0
hexiom.py	674	61.7m	2.2m	0	52

[2] 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,
- Jensen, Møller, and Thiemann: Value Analysis for JavaScript.

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.

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

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

Implementation size:

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

Flow-sensitive analysis

Flow-sensitive Analysis Performed by induction over the syntax.

Using flow tokens to label continuation-passed states.

$\mathcal{F}^\sharp = \{\, cur, ret, brk, exn\ a, a \in \textbf{Addr}^\sharp \,\}$

- Abstract environment (nominal types) $\mathcal{E}^\sharp = \textbf{Id} \to \mathcal{P}(\textbf{Addr}^\sharp)$
- Attribute abstraction (structural types) $\mathcal{S}^\sharp = \textbf{Addr}^\sharp \to \textbf{Attr}^\sharp$

$$\mathbb{E}^\sharp[\![\, x \,]\!]\,(f \in \mathcal{F}^\sharp, \epsilon \in \mathcal{E}^\sharp, \sigma \in \mathcal{S}^\sharp) \stackrel{\text{def}}{=} \bigcup_{a^\sharp \in \epsilon(x)} \textbf{Obj}\ a^\sharp, (f, \epsilon[x \mapsto \{\, a^\sharp \,\}], \sigma)$$

$$\mathbb{S}^\sharp[\![\, x = e \,]\!]\,(f \in \mathcal{F}^\sharp, \epsilon \in \mathcal{E}^\sharp, \sigma \in \mathcal{S}^\sharp) \stackrel{\text{def}}{=}$$

$$\text{letif } (\textbf{Obj}\ @, (f, \epsilon, \sigma)) = \mathbb{E}^\sharp[\![\, e \,]\!]\,(f, \epsilon, \sigma) \text{ in}$$

$$(f, \epsilon[x \mapsto @], \sigma)$$

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)
```

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

Description of the Type Abstract Domain – Polymorphism

```
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)
```

$$\underbrace{\begin{cases} r \in \{ @^{w}_{str} \} \\ sep \in \{ @^{s}_{str} \} \end{cases}}_{\text{if branch}} \qquad \underbrace{\begin{cases} r \in \{ @^{w}_{str} \} \\ sep \in \{ @^{s}_{str} \} \end{cases}}_{\text{else branch}}$$

Description of the Type Abstract Domain – Polymorphism

```
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)
```

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

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)
```

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

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)
```

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

Description of the Type Abstract Domain – Polymorphism

```
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)
```

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

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

The polymorphism improves the precision.