Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries

Raphaël Monat

PhD defense

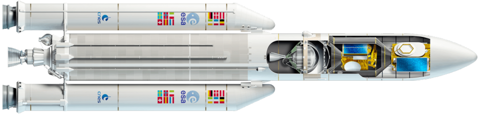
22 November 2021

Introduction

Software

Software is ubiquitous

- Transportation



Software

Software is ubiquitous

- Transportation

Software

Software is ubiquitous

- Transportation

Software

Software is ubiquitous

- Transportation
- Communication

Software

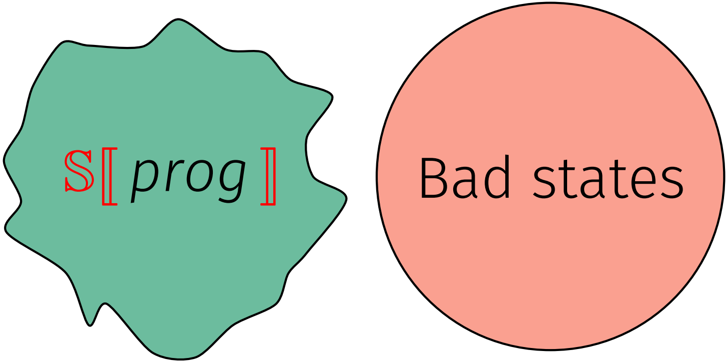
Software is ubiquitous

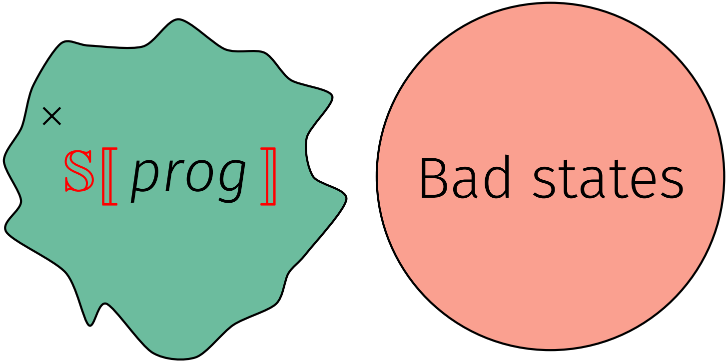
- Transportation
- Communication
- …

Bugs

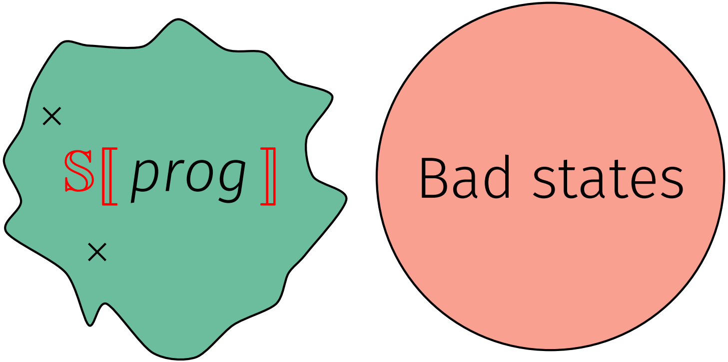
When a program does not work as intended

Software verification

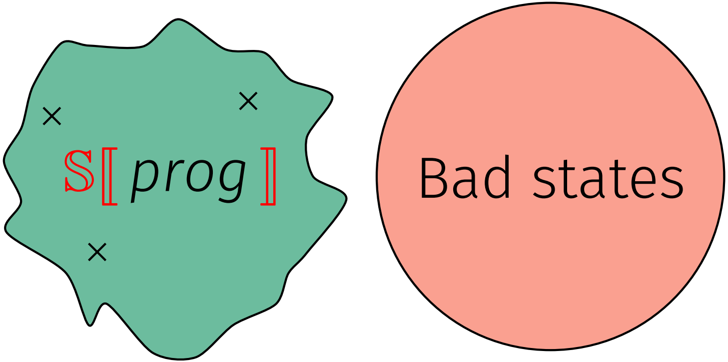

$\mathbb{S}[\![prog]\!]$
Bad states



Cheap approach: test *prog*.

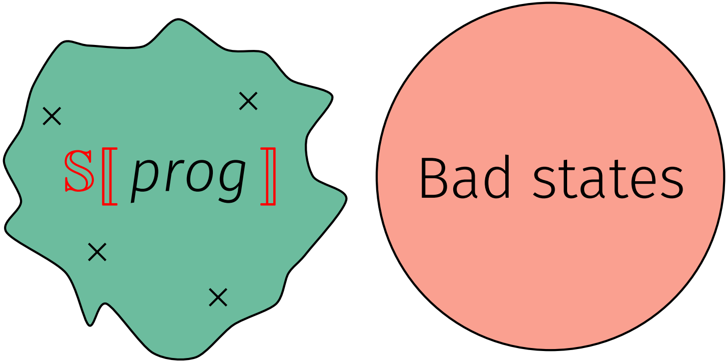


Cheap approach: test *prog*.

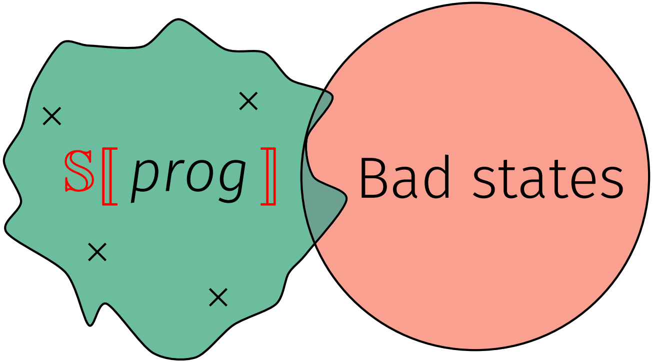


Cheap approach: test *prog*.

Software verification

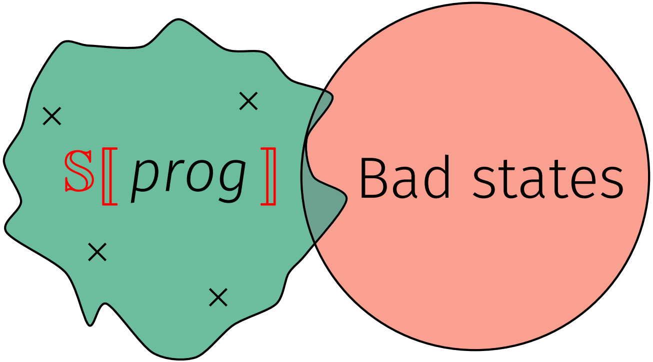


Cheap approach: test *prog*.



Cheap approach: test *prog*.

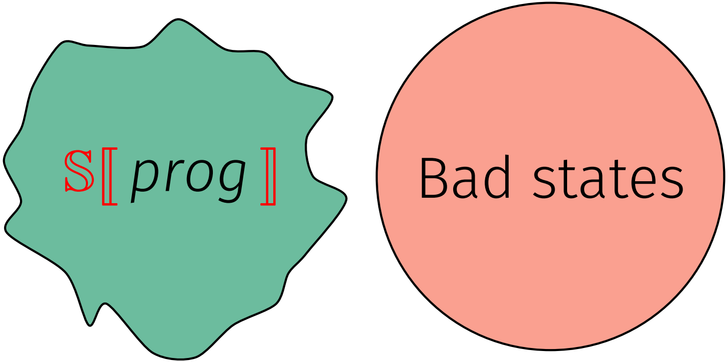
Software verification



Cheap approach: test *prog*.

Some bugs may go undetected!

Software verification



Cheap approach: test *prog*.

Some bugs may go undetected!

Would there be a way to automatically prove programs correct?

An impossibility theorem

Rice's theorem (1953)

All non-trivial semantic properties of programs are undecidable.

An impossibility theorem

Rice's theorem (1953)

All non-trivial semantic properties of programs are undecidable.

It is impossible to have a system which is:

- Automatic: no user interaction required, terminates in finite time.

An impossibility theorem

Rice's theorem (1953)

All non-trivial semantic properties of programs are undecidable.

It is impossible to have a system which is:

- Automatic: no user interaction required, terminates in finite time.
- Sound: derived properties are true on the program.

An impossibility theorem

Rice's theorem (1953)

All non-trivial semantic properties of programs are undecidable.

It is impossible to have a system which is:

- Automatic: no user interaction required, terminates in finite time.
- Sound: derived properties are true on the program.
- Complete: all properties of the program can be derived.

An impossibility theorem

Rice's theorem (1953)

All non-trivial semantic properties of programs are undecidable.

It is impossible to have a system which is:

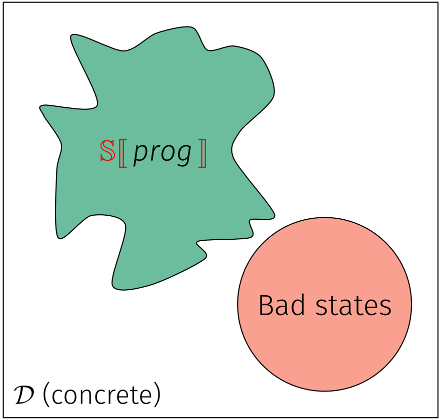
- Automatic: no user interaction required, terminates in finite time.
- Sound: derived properties are true on the program.
- Complete: all properties of the program can be derived.

Mitigating Rice's theorem

Our choice: sound and automatic approaches.

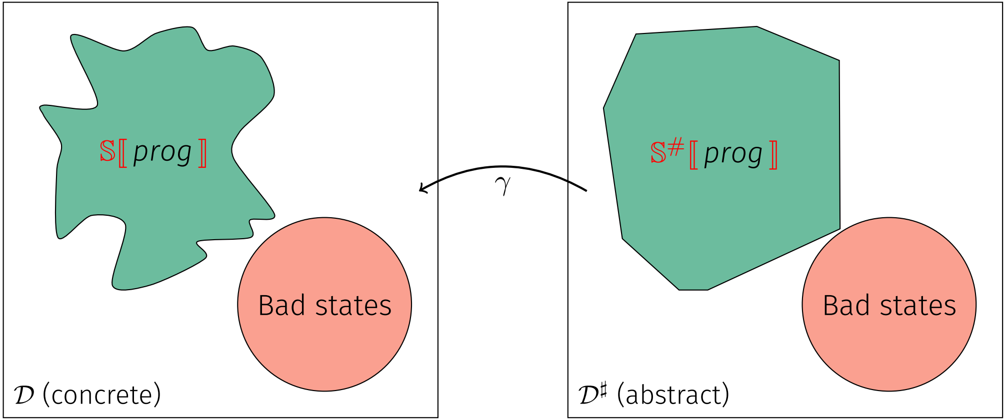
Aimed at certifying programs correct.

Abstract interpretation – the big picture



P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". POPL 1977

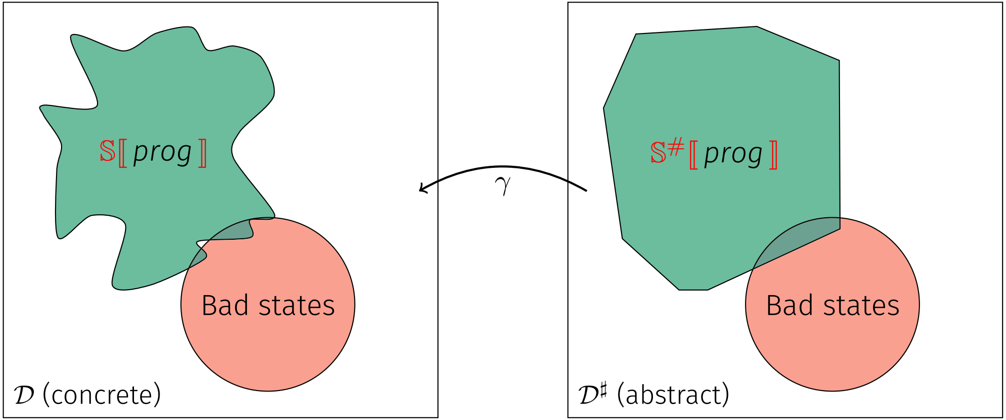
Abstract interpretation – the big picture



Program proved safe

P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". POPL 1977

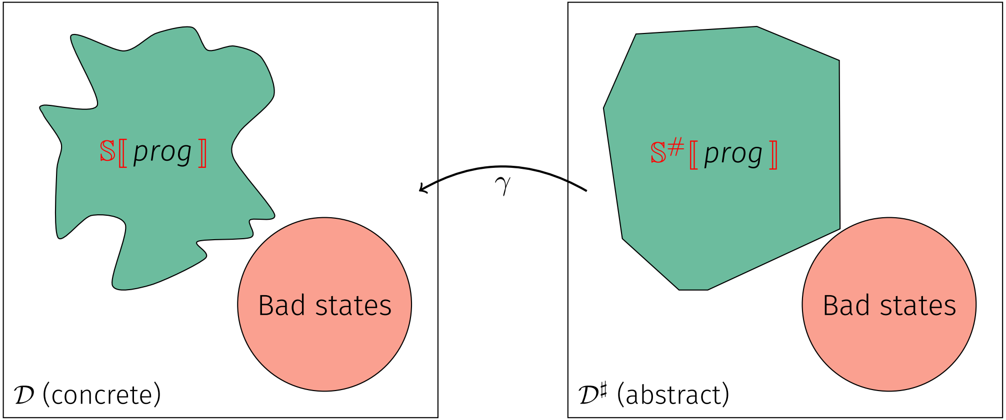
Abstract interpretation – the big picture



True alarm

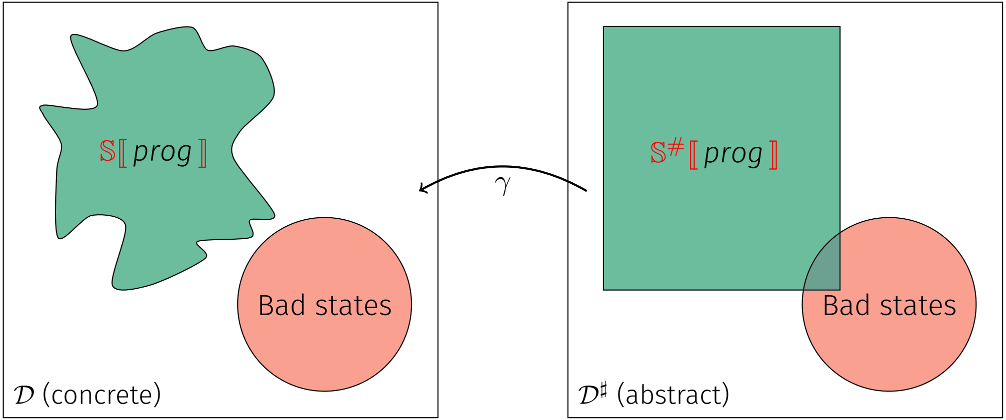
P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". POPL 1977

Abstract interpretation – the big picture



P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977

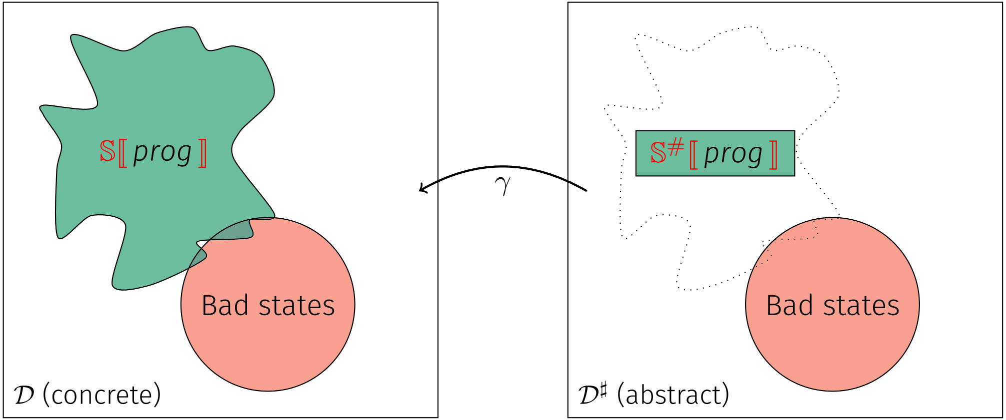
Abstract interpretation – the big picture



False alarm (Abstraction too coarse)

P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. POPL 1977

Abstract interpretation – the big picture



Unsound analysis
(shouldn't happen)

P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". POPL 1977

Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub[1]

[1] https://octoverse.github.com/#top-languages

Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub[1]

New features

- Object orientation,

[1] `https://octoverse.github.com/#top-languages`

Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub[1]

New features

- Object orientation,
- Dynamic typing,

[1] `https://octoverse.github.com/#top-languages`

Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub[1]

New features

- Object orientation,
- Dynamic typing,
- Dynamic object structure,

[1] https://octoverse.github.com/#top-languages

Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub[1]

New features

- Object orientation,
- Dynamic typing,
- Dynamic object structure,
- Introspection operators,

[1] https://octoverse.github.com/#top-languages

Dynamic programming languages

Growing popularity

JavaScript #1, Python #2 on GitHub[1]

New features

- Object orientation,
- Dynamic typing,
- Dynamic object structure,
- Introspection operators,
- `eval`.

[1] https://octoverse.github.com/#top-languages

State of the art

Well-established & industrialized analysis of static programming languages

- C: Polyspace (1999), Astrée (2003), Frama-C (2008)
- Java: Julia (2010)

State of the art

Well-established & industrialized analysis of static programming languages

- C: Polyspace (1999), Astrée (2003), Frama-C (2008)
- Java: Julia (2010)

Around JavaScript

- First: Jensen, Møller, and Thiemann. “Type Analysis for JavaScript”. SAS 2009
- Bodin et al. “A trusted mechanised JavaScript specification”. POPL 2014

State of the art

Well-established & industrialized analysis of static programming languages

- C: Polyspace (1999), Astrée (2003), Frama-C (2008)
- Java: Julia (2010)

Around JavaScript

- First: Jensen, Møller, and Thiemann. “Type Analysis for JavaScript”. SAS 2009
- Bodin et al. “A trusted mechanised JavaScript specification”. POPL 2014

Why Python?

Used a lot in

- Scientific computing
- Scripts and automation

Outline

A Taste of Python

Python's specificities

No standard

- CPython is the reference

$\Longrightarrow$ manual inspection of the source code and handcrafted tests

Python's specificities

No standard

- CPython is the reference

  $\Longrightarrow$ manual inspection of the source code and handcrafted tests

Operator redefinition

- Calls, additions, attribute accesses
- Operators eventually call overloaded `__methods__`

Protected attributes

```python
class Protected:
  def __init__(self, priv):
    self._priv = priv
  def __getattribute__(self, attr):
    if attr[0] == "_": raise AttributeError("protected")
    return object.__getattribute__(self, attr)

a = Protected(42)
a._priv # AttributeError raised
```

Python's specificities (II)

Dual type system

- Nominal (classes, MRO)

Fspath (from standard library)

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

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

fspath("/dev" if random() else Path())
```

Barrett et al. "A Monotonic Superclass Linearization for Dylan". OOPSLA 1996

Python's specificities (II)

Dual type system

- Nominal (classes, MRO)
- **Structural (attributes)**

Fspath (from standard library)

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

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

fspath("/dev" if random() else Path())
```

Barrett et al. "A Monotonic Superclass Linearization for Dylan". OOPSLA 1996

Python's specificities (II)

Dual type system

- Nominal (classes, MRO)
- Structural (attributes)

Exceptions

Exceptions rather than specific values

- `1 + "a"` ⇝ `TypeError`
- `l[len(l) + 1]` ⇝ `IndexError`

Fspath (from standard library)

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

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

fspath("/dev" if random() else Path())
```

Barrett et al. “A Monotonic Superclass Linearization for Dylan”. OOPSLA 1996

Previous works on Python 3

Guth. “A formal semantics of Python 3.3”. 2013

Implementation within the K framework.

Previous works on Python 3

Guth. "A formal semantics of Python 3.3". 2013

Implementation within the K framework.

Politz et al. "Python: The full monty". OOPSLA 2013

Complex desugaring into λ_π.

May incur losses of precision in the abstract interpreter.

Previous works on Python 3

Guth. "A formal semantics of Python 3.3". 2013

Implementation within the K framework.

Politz et al. "Python: The full monty". OOPSLA 2013

Complex desugaring into λ_π.

May incur losses of precision in the abstract interpreter.

Köhl. "An Executable Structural Operational Formal Semantics for Python". 2021

Semantics of Python, using a Python framework, developped concurrently.

Previous works on Python 3

Guth. "A formal semantics of Python 3.3". 2013

Implementation within the K framework.

Politz et al. "Python: The full monty". OOPSLA 2013

Complex desugaring into λ_π.

May incur losses of precision in the abstract interpreter.

Köhl. "An Executable Structural Operational Formal Semantics for Python". 2021

Semantics of Python, using a Python framework, developped concurrently.

Different goal

These works focus on the concrete semantics. This is not our endgoal.

Previous works on Python 3

Guth. "A formal semantics of Python 3.3". 2013

Implementation within the K framework.

Politz et al. "Py

Complex desug

May incur losse

Köhl. "An Execu **Python". 2021**

Semantics of Python, using a Python framework, developped concurrently.

Moving to our own semantics

- Cost of understanding the code (vs CPython)
- Trust in the code (CPython's tests?)
- Insights of the papers

Different goal

These works focus on the concrete semantics. This is not our endgoal.

Our approach

Interpreter-like semantics

Easily convertable to an abstract interpreter.

[2] Fromherz, Ouadjaout, and Miné. "Static Value Analysis of Python Programs by Abstract Interpretation". NFM 2018.

Our approach

Interpreter-like semantics

Easily convertable to an abstract interpreter.

Major extension of the work of Fromherz, Ouadjaout, and Miné[2]

- Separation between core and builtins
- $2.3\times$ more cases (`with` statement, bidirectional generators, …)
- Improved some cases ($+$, boolean casts of conditionals, data descriptors, …)

[2] Fromherz, Ouadjaout, and Miné. "Static Value Analysis of Python Programs by Abstract Interpretation". NFM 2018.

Our approach

Interpreter-like semantics

Easily convertable to an abstract interpreter.

Major extension of the work of Fromherz, Ouadjaout, and Miné[2]

- Separation between core and builtins
- 2.3$\times$ more cases (`with` statement, bidirectional generators, …)
- Improved some cases ($+$, boolean casts of conditionals, data descriptors, …)

Correctness

- Strived to make it auditable (with links to the source).
- Tested only through the abstract analysis yet (no concrete execution).

[2] Fromherz, Ouadjaout, and Miné. "Static Value Analysis of Python Programs by Abstract Interpretation". NFM 2018.

Example – attribute access

$$\mathbb{E}_{\text{cur}}[\![\, x.s \,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{LOAD\_ATTR PyObject\_GetAttr (slot\_tp\_getattr\_hook)}$$

$\text{letb } (cur, e, h), @_x = \mathbb{E}[\![\, x \,]\!](cur, e, h) \text{ in}$

$\text{letb } (cur, e, h), @_c = \mathbb{E}[\![\, mro\_search(type(@_x), \text{"}\texttt{\_\_getattribute\_\_}\text{"}) \,]\!](cur, e, h) \text{ in}$

$\text{letcases } (f, e, h), @_{x.s} = \mathbb{E}[\![\, @_c(@_x, s) \,]\!](cur, e, h) \text{ in}$

$\text{match } f \text{ with}$

- $exn\ @_{exc} \text{ when } isinstance(@_{exc}, \texttt{AttributeError}) \Rightarrow$

  $\text{let } (f, e, h), @_d = \mathbb{E}[\![\, mro\_search(type(@_x), \text{"}\texttt{\_\_getattr\_\_}\text{"}) \,]\!](f, e, h) \text{ in}$

  $\text{if } d \neq \bot \text{ then return } \mathbb{E}[\![\, @_d(@_x, s) \,]\!](cur, e, h)$

  $\text{else return } (f, e, h), \bot$

- $\_ \Rightarrow \text{return } (f, e, h), @_{x.s}$

Example – attribute access

$\mathbb{E}_{cur}[\![x.s]\!](cur, e, h) \stackrel{\text{def}}{=}$ `LOAD_ATTR PyObject_GetAttr (slot_tp_getattr_hook)`

$\text{letb } (cur, e, h), @_x = \mathbb{E}[\![x]\!](cur, e, h) \text{ in}$

$\mathbb{E}_{cur}[\![\texttt{object.\_\_getattribute\_\_}(obj, name)]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field _PyObject_GenericGetAttrWithDict`

$\text{letb } (cur, e, h), @_o = \mathbb{E}[\![obj]\!](cur, e, h) \text{ in}$

$\text{letb } (cur, e, h), @_n = \mathbb{E}[\![name]\!](cur, e, h) \text{ in}$

$\text{if } \neg isinstance(@_n, \text{str}) \text{ then return } \mathbb{S}[\![\texttt{raise TypeError}]\!](cur, e, h), \bot \text{ else}$

$\text{let } \texttt{str}(n) = \text{fst} \circ h(@_n) \text{ in}$

$\text{letcases } (f, e, h), @_{descr} = \mathbb{E}[\![mro\_search(type(@_o), n)]\!](f, e, h) \text{ in}$

$\text{if } @_{descr} \neq \bot \text{ then}$

$\text{if } hasattr(type(@_{descr}), \texttt{"\_\_get\_\_"}) \land$

$(hasattr(type(@_{descr}), \texttt{"\_\_set\_\_"}) \lor hasattr(type(@_{descr}), \texttt{"\_\_delete\_\_"})) \text{ then}$

$\text{return } \mathbb{E}[\![type(@_{descr}).\texttt{\_\_get\_\_}(@_{descr}, @_o, type(@_o))]\!](f, e, h)$

Example – attribute access

$\mathbb{E}_{cur}[\![x.s]\!](cur, e, h) \stackrel{def}{=}$ `LOAD_ATTR PyObject_GetAttr (slot_tp_getattr_hook)`

$\quad$ letb $(cur, e, h), @_x = \mathbb{E}[\![x]\!](cur, e, h)$ in

$\mathbb{E}_{cur}[\![$ `object.__getattribute__` $(obj, name)]\!](cur, e, h) \stackrel{def}{=}$ `tp_field _PyObject_GenericGetAttrWithDict`

$\quad$ letb $(cur, e, h), @_o = \mathbb{E}[\![obj]\!](cur, e, h)$ in

$\quad$ letb $(cur, e, h), @_n = \mathbb{E}[\![name]\!]($

$\mathbb{E}_{cur}[\![$ `type.__getattribute__` $(typ, name)]\!](cur, e, h) \stackrel{def}{=}$ `tp_field type_getattro`

$\quad$ letb $(cur, e, h), @_{typ} = \mathbb{E}[\![typ]\!](cur, e, h)$ in

$\quad$ letb $(cur, e, h), @_{name} = \mathbb{E}[\![name]\!](cur, e, h)$ in

$\quad$ letb $(cur, e, h), @_{meta} = \mathbb{E}[\![\text{mro\_search}(type(@_{typ}), @_{name})]\!](cur, e, h)$ in

$\quad$ if $@_{meta} \neq \bot$ then

$\quad\quad$ if $hasattr(type(@_{meta}), \text{"\_\_get\_\_"}) \wedge$

$\quad\quad\quad (hasattr(type(@_{meta}), \text{"\_\_set\_\_"}) \vee hasattr(type(@_{meta}), \text{"\_\_delete\_\_"}))$ then

$\quad\quad\quad (@_{meta}, @_{typ}, type(@_{typ}))]\!](cur, e, h)$

Analyzing Python Programs

Goal

Detect runtime errors: uncaught raised exceptions

Goal

Detect runtime errors: uncaught raised exceptions

Supported constructs

Our analysis supports:

- Objects
- Exceptions
- Dynamic typing
- Introspection
- Permissive semantics
- Dynamic attributes
- Generators
- `super`
- Metaclasses

Analysis | Overview

Goal

Detect runtime errors: uncaught raised exceptions

Supported constructs

Our analysis supports:

- Objects
- Exceptions
- Dynamic typing
- Introspection
- Permissive semantics
- Dynamic attributes
- Generators
- `super`
- Metaclasses

Unsupported constructs

- Recursive functions
- `eval`
- Finalizers

Analysis | Domains required

Avering numbers

```
def average(l):
  m = 0
  for i in range(len(l)):
    m = m + l[i]
  m = m // (i + 1)
  return m

l = [randint(0, 20)
  for i in range(randint(5, 10))]
m = average(l)
```

Analysis | Domains required

Avering numbers

```python
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

- `m // (i+1)`
- `l[i]`

Searching for a loop invariant (l. 4)

Environment abstraction

$m \mapsto @^\sharp_{\mathtt{int}^\sharp} \quad i \mapsto @^\sharp_{\mathtt{int}^\sharp}$

Analysis | Domains required

Avering numbers

```
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

- `m // (i+1)`
- `l[i]`

Searching for a loop invariant (l. 4)

Stateless domains: **list content**,

Environment abstraction

$m \mapsto @^\sharp_{\mathtt{int}^\sharp} \quad i \mapsto @^\sharp_{\mathtt{int}^\sharp} \quad \underline{\mathsf{els}}(l) \mapsto @^\sharp_{\mathtt{int}^\sharp}$

Analysis | Domains required

Avering numbers

```
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

- `m // (i+1)`
- `l[i]`

Searching for a loop invariant (l. 4)

Stateless domains: list content,

Environment abstraction

$m \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad i \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad \underline{\mathsf{els}}(l) \mapsto @^\sharp_{\texttt{int}^\sharp}$

Numeric abstraction (intervals)

$m \in [0, +\infty)$ $\quad \underline{\mathsf{els}}(l) \in [0, 20]$ $\quad i \in [0, +\infty)$

Analysis | Domains required

Avering numbers

```
1   def average(l):
2     m = 0
3     for i in range(len(l)):
4       m = m + l[i]
5     m = m // (i + 1)
6     return m
7
8   l = [randint(0, 20)
9     for i in range(randint(5, 10))]
10  m = average(l)
```

Proved safe?

- `m // (i+1)`
- `l[i]`

Searching for a loop invariant (l. 4)

Stateless domains: list content, **list length**

Environment abstraction

$m \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad i \mapsto @^\sharp_{\texttt{int}^\sharp}$ $\quad \underline{\text{els}}(l) \mapsto @^\sharp_{\texttt{int}^\sharp}$

Numeric abstraction (intervals)

$m \in [0, +\infty)$ $\quad \underline{\text{els}}(l) \in [0, 20]$

$\underline{\textbf{len}}(l) \in \mathbf{[5, 10]}$ $\quad i \in \mathbf{[0, 10]}$

Analysis | Domains required

Avering numbers

```
1  def average(l):
2    m = 0
3    for i in range(len(l)):
4      m = m + l[i]
5    m = m // (i + 1)
6    return m
7
8  l = [randint(0, 20)
9    for i in range(randint(5, 10))]
10 m = average(l)
```

Proved safe?

- `m // (i+1)`
- `l[i]`

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

$m \mapsto @^\sharp_{\mathtt{int}^\sharp} \quad i \mapsto @^\sharp_{\mathtt{int}^\sharp} \quad \underline{\mathrm{els}}(l) \mapsto @^\sharp_{\mathtt{int}^\sharp}$

Numeric abstraction (polyhedra)

$m \in [0, +\infty) \qquad \underline{\mathrm{els}}(l) \in [0, 20]$

$0 \leq i < \underline{\mathrm{len}}(l) \qquad 5 \leq \underline{\mathrm{len}}(l) \leq 10$

Analysis | Domains required

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(len(l)):
9      m = m + l[i].weight
10   m = m // (i + 1)
11   return m
12
13 l = [Task(randint(0, 20))
14   for i in range(randint(5, 10))]
15 m = average(l)
```

Proved safe?

- `m // (i+1)`
- `l[i].weight`

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

$m \mapsto @^\sharp_{\mathtt{int}^\sharp}$ $\quad i \mapsto @^\sharp_{\mathtt{int}^\sharp}$ $\quad \underline{\mathsf{els}}(l) \mapsto @^\sharp_{\mathtt{Task}}$

$\underline{@^\sharp_{\mathtt{Task}} \cdot \mathtt{weight}} \mapsto @^\sharp_{\mathtt{int}^\sharp}$

Numeric abstraction (polyhedra)

$m \in [0, +\infty)$

$0 \leq i < \underline{\mathsf{len}}(l) \qquad 5 \leq \underline{\mathsf{len}}(l) \leq 10$

$0 \leq \underline{@^\sharp_{\mathtt{Task}} \cdot \mathtt{weight}} \leq 20$

Attributes abstraction

$@^\sharp_{\mathtt{Task}} \mapsto (\{\mathtt{weight}\}, \emptyset)$

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(le
9      m = m + l[i].wei
10   m = m // (i + 1)
11   return m
12
13 l = [Task(randint(0,
14   for i in range(randint(5, 10))]
15 m = average(l)
```

Proved safe?

- `m // (i+1)`
- `l[i].weight`

Searching for a loop invariant (l. 4)

Stateless domains: list content, list length

Environment abstraction

$m \mapsto @^\sharp$ … $i \mapsto @^\sharp$ … $@^\sharp_{\texttt{Task}}$

Conclusion

- Different domains depending on the precision
- Use of auxiliary variables (underlined)

$0 \leq i < \underline{\text{len}(l)} \qquad 5 \leq \underline{\text{len}(l)} \leq 10$

$0 \leq \underline{@^\sharp_{\texttt{Task}} \cdot \texttt{weight}} \leq 20$

Attributes abstraction

$@^\sharp_{\texttt{Task}} \mapsto (\{\texttt{weight}\}, \emptyset)$

Modular Open Platform for Static Analysis[3]

[3] Journault, Miné, Monat, and Ouadjaout. "Combinations of reusable abstract domains for a multilingual static analyzer". VSTTE 2019.

Modular Open Platform for Static Analysis[3]

- One AST to analyze them all
  - Multilanguage support
  - Expressiveness
  - Reusability

[3] Journault, Miné, Monat, and Ouadjaout. "Combinations of reusable abstract domains for a multilingual static analyzer". VSTTE 2019.

Modular Open Platform for Static Analysis[3]

- One AST to analyze them all
  - Multilanguage support
  - Expressiveness
  - Reusability
- Unified domain signature
  - Semantic rewriting
  - Loose coupling
  - Observability

[3] Journault, Miné, Monat, and Ouadjaout. “Combinations of reusable abstract domains for a multilingual static analyzer”. VSTTE 2019.

Modular Open Platform for Static Analysis[3]

- One AST to analyze them all
  - Multilanguage support
  - Expressiveness
  - Reusability
- Unified domain signature
  - Semantic rewriting
  - Loose coupling
  - Observability
- DAG of abstract domains
  - Composition
  - Cooperation

[3] Journault, Miné, Monat, and Ouadjaout. "Combinations of reusable abstract domains for a multilingual static analyzer". VSTTE 2019.

Universal.Iterators.Loops
Matches `while(...){...}` Computes fixpoint using widening

Mopsa | Dynamic, semantic iterators with delegation

```
for(init; cond; incr) body
```

Universal.Iterators.Loops
Matches `while(...){...}` Computes fixpoint using widening

Mopsa | Dynamic, semantic iterators with delegation

```
for(init; cond; incr) body
```

C.iterators.loops

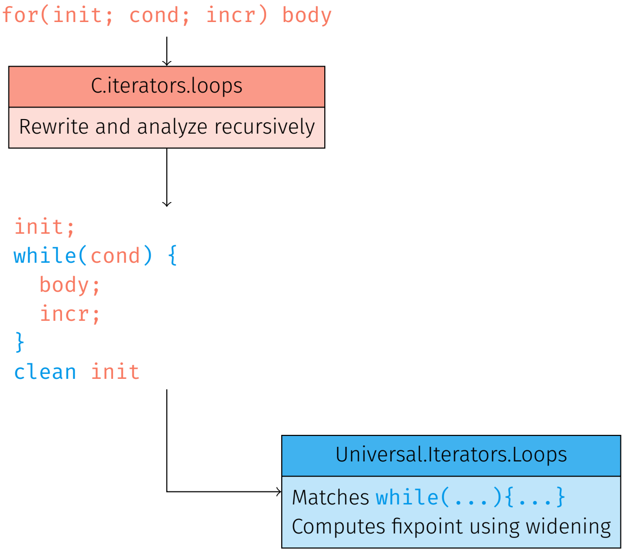
Rewrite and analyze recursively

Universal.Iterators.Loops

Matches `while(...){...}`

Computes fixpoint using widening

Mopsa | Dynamic, semantic iterators with delegation



Mopsa | Dynamic, semantic iterators with delegation

```
for(init; cond; incr) body
```

```
for target in iterable: body
```

C.iterators.loops

Rewrite and analyze recursively

```
init;
while(cond) {
  body;
  incr;
}
clean init
```

Universal.Iterators.Loops

Matches `while(...){...}`

Computes fixpoint using widening

Mopsa | Dynamic, semantic iterators with delegation

```
for(init; cond; incr) body
```

C.iterators.loops

Rewrite and analyze recursively

```
init;
while(cond) {
  body;
  incr;
}
clean init
```

Universal.Iterators.Loops

Matches `while(...){...}`

Computes fixpoint using widening

```
for target in iterable: body
```

Python.Desugar.Loops

- Rewrite and analyze recursively
- Optimize for some semantic cases

Mopsa | Dynamic, semantic iterators with delegation

```
for(init; cond; incr) body
```

C.iterators.loops

Rewrite and analyze recursively

```
init;
while(cond) {
  body;
  incr;
}
clean init
```

```
for target in iterable: body
```

Python.Desugar.Loops

- Rewrite and analyze recursively
- Optimize for some semantic cases

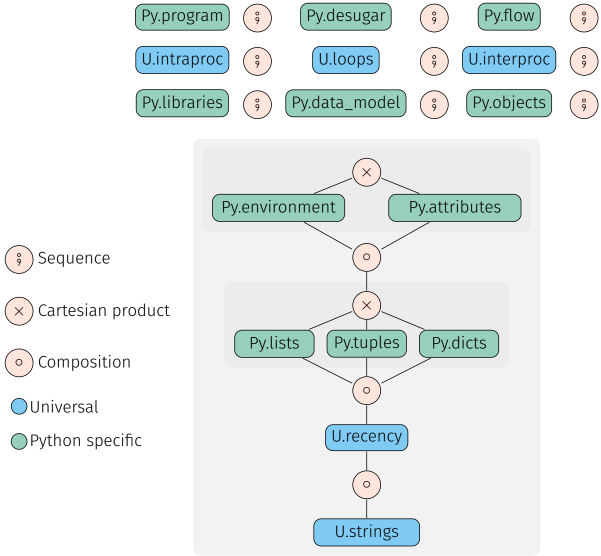
```
it = iter(iterable)
while(1) {
 try: target = next(it)
 except StopIteration: break
 body
}
clean it
```

Universal.Iterators.Loops

Matches `while(...){...}`
Computes fixpoint using widening

- Dynamicity:
  type inference first
- Flow-sensitive
- Context-sensitive

Types | Analysis



- Dynamicity: type inference first
- Flow-sensitive
- Context-sensitive

Types | Related work

- Similar in essence to TAJS.[4]
- Dataflow analysis by Fritz and Hage.[5]
- Typpete: SMT-based type inference.[6]
- Pytype, type inference tool used by Google.[7]
- RPython: efficient compilation of a static subset of Python.[8]
- Value analysis by Fromherz et al.[9]

---

[4] Jensen, Møller, and Thiemann. "Type Analysis for JavaScript". SAS 2009.

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

[6] Hassan, Urban, Eilers, and Müller. "MaxSMT-Based Type Inference for Python 3". CAV 2018.

[7] Kramm et al. Pytype. 2019.

[8] Ancona, Ancona, Cuni, and Matsakis. "RPython: a step towards reconciling dynamically and statically typed OO languages". DLS 2007.

[9] Fromherz, Ouadjaout, and Miné. "Static Value Analysis of Python Programs by Abstract Interpretation". NFM 2018.

Types | Experimental evaluation

Name	LOC	Mopsa	⚠	Fritz & Hage	Pytype	Typpete	Fromherz et al.	RPython
bellman_ford.py	61	0.24s	0†	1.4s	0.99s	1.4s	2.4m	7.1s
float.py	63	82ms	0†	1.7s	0.92s	1.3s	0.84s	5.6s
coop_concat.py	64	43ms	0†	1.8s	0.81s	1.3s	20ms	🐞
crafting.py	132	0.41s	0†🔑	1.6s	0.97	1.7s	🐞	🐞
nbody.py	157	0.80s	1†🔑±	1.7s	1.3s	🐞	🐞	🐞
chaos.py	324	2.3s	0†±	13s	11s	🐞	🐞	🐞
scimark.py	416	0.55s	2†	8.5s	4.4s	🐞	🐞	🐞
richards.py	426	5.0s	2†±	38s	2.4s	🐞	🐞	7.8s
unpack_seq.py	458	4.2s	0±	1.1s	7.4s	2.7s	14s	🐞
go.py	461	15s	32†±	8.5s	3.4s	🐞	🐞	🐞
hexiom.py	674	22s	25†🔑±	🐞	4.2s	🐞	🐞	🐞
regex_v8.py	1792	15s	0†	4.9s	🕓	1.7m	🐞	🐞
processInput.py	1417	4.8s	7†🔑±	2.4s	11s	🐞	🐞	🐞
choose.py	2562	46s	17🔑†±	1.7s	15s	🐞	🐞	🐞

🐞 unsupported by the analyzer (crash) 🕓 timeout (after 1h)
Smashed exceptions: KeyError 🔑, IndexError †, ValueError ±

Types | Experimental evaluation

Name	LOC	Mopsa	⚠	Fritz & Hage	Pytype	Typpete	Fromherz et al.	RPython
bellman_ford.py	61	0.24s	0†	1.4s	0.99s	1.4s	2.4m	7.1s
float.py	63	82ms	0†	1.7s	0.92s	1.3s	0.84s	5.6s
coop_concat.py	64	43ms	0†	1.8s	0.81s	1.3s	20ms	🐞
crafting.py	132	0.41s	0†🔑				🐞	🐞
nbody.py	157						🐞	🐞
chaos.py	324						🐞	🐞
scimark.py	416						🐞	🐞
richards.py	426						🐞	7.8s
unpack_seq.py	458						14s	🐞
go.py	461						🐞	🐞
hexiom.py	674					🐞	🐞	🐞
regex_v8.py	1792	15s	0†	4.9s	🕓	1.7m	🐞	🐞
processInput.py	1417	4.8s	7†🔑⋆	2.4s	11s	🐞	🐞	🐞
choose.py	2562	46s	17🔑†⋆	1.7s	15s	🐞	🐞	🐞

🐞 unsupported by the analyzer (crash) 🕓 timeout (after 1h)
Smashed exceptions: KeyError 🔑, IndexError †, ValueError ⋆

Conclusion

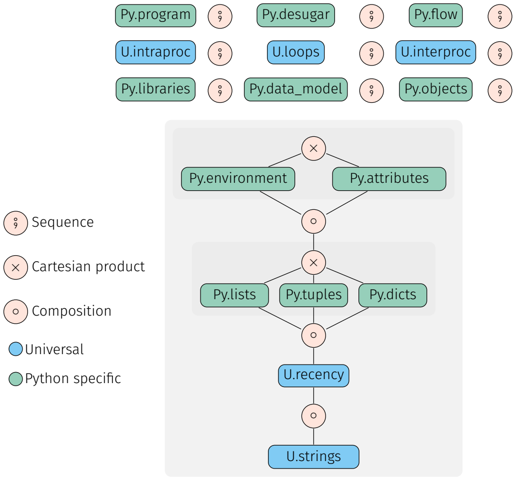
- Handling Python's dynamicity
- Good scalability (w.r.t. other semantic tools)

Thanks to Mopsa, switching from types to values is straightforward!

Monat, Ouadjaout, and Miné. “Value and allocation sensitivity in static Python analyses”.

Types ⇝ values | Configurations

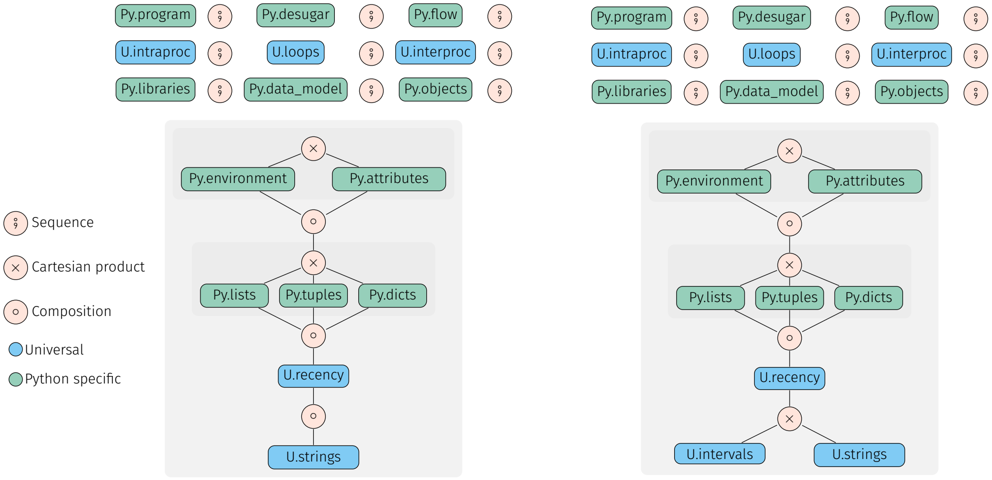
Thanks to Mopsa, switching from types to values is straightforward!



Monat, Ouadjaout, and Miné. “Value and allocation sensitivity in static Python analyses”.

Types ⇝ values | Configurations

Thanks to Mopsa, switching from types to values is straightforward!



Monat, Ouadjaout, and Miné. “Value and allocation sensitivity in static Python analyses”.

Types ⇝ values | Comparing the analyses

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(len(l)):
9      m = m + l[i].weight
10   m = m // (i + 1)
11   return m
12
13 l = []
14 for i in range(randint(5, 10)):
15   l.append(Task(randint(0, 20)))
16 m = average(l)
```

Type analysis

- ▶ `ValueError` (l. 3)

Types ⇝ values | Comparing the analyses

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(len(l)):
9      m = m + l[i].weight
10   m = m // (i + 1)
11   return m
12
13 l = []
14 for i in range(randint(5, 10)):
15   l.append(Task(randint(0, 20)))
16 m = average(l)
```

Type analysis

- `ValueError` (l. 3)
- `IndexError` (l. 9)

Types $\rightsquigarrow$ values | Comparing the analyses

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(len(l)):
9      m = m + l[i].weight
10   m = m // (i + 1)
11   return m
12
13 l = []
14 for i in range(randint(5, 10)):
15   l.append(Task(randint(0, 20)))
16 m = average(l)
```

Type analysis

- `ValueError` (l. 3)
- `IndexError` (l. 9)
- `ZeroDivisionError` (l. 10)

Types ⇝ values | Comparing the analyses

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(len(l)):
9      m = m + l[i].weight
10   m = m // (i + 1)
11   return m
12
13 l = []
14 for i in range(randint(5, 10)):
15   l.append(Task(randint(0, 20)))
16 m = average(l)
```

Type analysis

- `ValueError` (l. 3)
- `IndexError` (l. 9)
- `ZeroDivisionError` (l. 10)
- `NameError` (l. 10)

Types ⇝ values | Comparing the analyses

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(len(l)):
9      m = m + l[i].weight
10   m = m // (i + 1)
11   return m
12
13 l = []
14 for i in range(randint(5, 10)):
15   l.append(Task(randint(0, 20)))
16 m = average(l)
```

Type analysis

- `ValueError` (l. 3)
- `IndexError` (l. 9)
- `ZeroDivisionError` (l. 10)
- `NameError` (l. 10)

Non-relational value analysis

`IndexError` (l. 9)

Types $\rightsquigarrow$ values | Comparing the analyses

Averaging tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  def average(l):
7    m = 0
8    for i in range(len(l)):
9      m = m + l[i].weight
10   m = m // (i + 1)
11   return m
12
13 l = []
14 for i in range(randint(5, 10)):
15   l.append(Task(randint(0, 20)))
16 m = average(l)
```

Type analysis

- `ValueError` (l. 3)
- `IndexError` (l. 9)
- `ZeroDivisionError` (l. 10)
- `NameError` (l. 10)

Non-relational value analysis

`IndexError` (l. 9)

Relational value analysis

No alarm!

Types ⇝ values | Comparing the analyses (II)

Name	LOC	Type Analysis					Non-relational Value Analysis				
		Time	Mem.	Exceptions detected			Time	Mem.	Exceptions detected		
				Type	Index	Key			Type	Index	Key
nbody.py	157	1.5s	3MB	0	22	1	5.7s	9MB	0	**1**	1
scimark.py	416	1.4s	12MB	1	1	0	3.4s	27MB	1	**0**	0
richards.py	426	13s	112MB	1	4	0	17s	149MB	1	**2**	0
unpack_seq.py	458	8.3s	7MB	0	0	0	9.4s	6MB	0	0	0
go.py	461	27s	345MB	33	20	0	2.0m	1.4GB	33	20	0
hexiom.py	674	1.1m	525MB	0	46	3	4.7m	3.2GB	0	**21**	3
regex_v8.py	1792	23s	18MB	0	2053	0	1.3m	56MB	0	**145**	0
processInput.py	1417	10s	64MB	7	7	1	12s	85MB	7	**4**	1
choose.py	2562	1.1m	1.6GB	12	22	7	2.9m	3.7GB	12	**13**	7
Total	9294	4.0m	2.8GB	59	2214	12	13m	9.1GB	59	228	12

Name	LOC	Type Analysis					Non-relational Value Analysis				
		Time	Mem.	Exceptions detected			Time	Mem.	Exceptions detected		
				Type	Index	Key			Type	Index	Key
nbody.py	157	1.5								**1**	1
scimark.py	416	1.4								**0**	0
richards.py	426	13								**2**	0
unpack_seq.py	458	8.3								0	0
go.py	461	27s								20	0
hexiom.py	674	1.1m								**21**	3
regex_v8.py	1792	23s								**145**	0
processInput.py	1417	10s								**4**	1
choose.py	2562	1.1m			22	7	2.9m	3.7GB	12	**13**	7
Total	9294	4.0m	2.8GB	59	2214	12	13m	9.1GB	59	228	12

Conclusion

The non-relational value analysis

- does not remove false type alarms
- significantly reduces index errors
- is $\simeq 3\times$ costlier

Types ⇝ values | Comparing the analyses (II)

Name	LOC	Type Analysis: Time	Mem.	Exceptions detected: Type	Index	Key	Non-relational Value Analysis: Time	Mem.	Exceptions detected:	Index	Key
nbody.py	157	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**1**	1
scimark.py	416	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**0**	0
richards.py	426	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**2**	0
unpack_seq.py	458	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0
go.py	461	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	20	0
hexiom.py	674	1.1m	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**21**	3
regex_v8.py	1792	23s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**145**	0
processInput.py	1417	10s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**4**	1
choose.py	2562	1.1m	[illegible]	[illegible]	[illegible]	[illegible]	2.9m	3.7GB	12	**13**	7
Total	9294	4.0m	2.8GB	59	2214	12	13m	9.1GB	59	228	12

Conclusion

The non-relational value analysis

- does not remove false type alarms
- significantly reduces index errors
- is $\simeq 3\times$ costlier

Heuristic packing and relational analyses

- Static packing, using function's scope
- Rules out all 145 alarms of `regex_v8.py` (1792 LOC) at $2.5\times$ cost

The recency abstraction[10]

- Finite number of abstract addresses

[10] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.
[11] Jensen, Møller, and Thiemann. “Type Analysis for JavaScript”. SAS 2009.

The recency abstraction[10]

- Finite number of abstract addresses
- Precise analysis of object initialization

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

[11] Jensen, Møller, and Thiemann. "Type Analysis for JavaScript". SAS 2009.

The recency abstraction[10]

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:

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

[11] Jensen, Møller, and Thiemann. "Type Analysis for JavaScript". SAS 2009.

The recency abstraction[10]

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:
  - by allocation site $l \in \mathbb{L}$

[10] Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS 2006.
[11] Jensen, Møller, and Thiemann. “Type Analysis for JavaScript”. SAS 2009.

The recency abstraction[10]

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:
  - by allocation site $l \in \mathbb{L}$
  - through a recency criterion: $(l, \boldsymbol{r})$ most recent allocation (with strong updates)
    $(l, \boldsymbol{o})$ older addresses (summarized)

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

[11] Jensen, Møller, and Thiemann. “Type Analysis for JavaScript”. SAS 2009.

The recency abstraction[10]

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:
  - by allocation site $l \in \mathbb{L}$
  - through a recency criterion: $(l, \boldsymbol{r})$ most recent allocation (with strong updates)
    $(l, \boldsymbol{o})$ older addresses (summarized)
- Initially designed for analysis of low-level code (binaries, C)

[10] Balakrishnan and Reps. "Recency-Abstraction for Heap-Allocated Storage". SAS 2006.
[11] Jensen, Møller, and Thiemann. "Type Analysis for JavaScript". SAS 2009.

The recency abstraction[10]

- Finite number of abstract addresses
- Precise analysis of object initialization
- Twofold partitioning:
  - by allocation site $l \in \mathbb{L}$
  - through a recency criterion: $(l, \boldsymbol{r})$ most recent allocation (with strong updates)
    $(l, \boldsymbol{o})$ older addresses (summarized)
- Initially designed for analysis of low-level code (binaries, C)
- Also used in Type Analysis for JavaScript[11]

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

[11] Jensen, Møller, and Thiemann. "Type Analysis for JavaScript". SAS 2009.

Task creation

```python
class Task:
  def __init__(self, weight):
    if weight < 0: raise ValueError
    self.weight = weight

m = [1, 2]
l = [Task(i) for i in m]
l.append(Task(3))
```

Type analysis

Nominal types used in abstract addresses. No need for allocation-site in `Tasks`. But helpful for lists!

Task creation

```
class Task:
  def __init__(self, weight):
    if weight < 0: raise ValueError
    self.weight = weight

m = [1, 2]
l = [Task(i) for i in m]
l.append(Task(3))
```

Type analysis

Nominal types used in abstract addresses. No need for allocation-site in `Tasks`. But helpful for lists!

Value analysis

Use allocation sites for `range` objects.

Task creation

```python
class Task:
  def __init__(self, weight):
    if weight < 0: raise ValueError
    self.weight = weight

m = [1, 2]
l = [Task(i) for i in m]
l.append(Task(3))
```

Type analysis

Nominal types used in abstract addresses. No need for allocation-site in `Tasks`. But helpful for lists!

Value analysis

Use allocation sites for `range` objects.

Variable allocation policies

- Type-based (nominal) and/or location-based partitioning.
- Different configurations depending on type/value analysis.

List abstraction

- Summarization of the content (auxiliary variable)
- Auxiliary length variable

List abstraction

- Summarization of the content (auxiliary variable)
- Auxiliary length variable

Dictionaries in Python

- Keys can be any object (JavaScript: strings or symbols)
- Key/value summarization currently used

Soundness

Two soundnesses

- Modelization of the semantics from CPython
- Implementation of this semantics within Mopsa

Soundness

Two soundnesses

- Modelization of the semantics from CPython
- Implementation of this semantics within Mopsa

Our approach

- Test only in the abstract
- Issue of overapproximations and unproved assertions

Unsupported constructs

- `eval`
- Recursive functions
- Finalizers

Soundness

Two soundnesses

- Modelization of the semantics from CPython
- Implementation of this semantics within Mopsa

Our approach

- Test only in the abstract
- Issue of overapproximations and unproved assertions

Unsupported constructs

- `eval`
- Recursive functions
- Finalizers

Tests from previous works

- 450/586 tests supported
- 268/586 assertions proved

Official tests from CPython

- 325/416 tests supported (17 chosen files)
- 389/702 assertions proved

Analyzing Python Programs with C Libraries

Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)

Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Pitfalls

Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Pitfalls

- Different values (arbitrary-precision integers in Python, bounded in C)

Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Pitfalls

- Different values (arbitrary-precision integers in Python, bounded in C)
- Different object representations (Python objects, C structs)

Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Pitfalls

- Different values (arbitrary-precision integers in Python, bounded in C)
- Different object representations (Python objects, C structs)
- Different runtime-errors (exceptions in Python)

Combining C and Python – motivation

One in five of the top 200 Python libraries contains C code

- To bring better performance (numpy)
- To provide library bindings (pygit2)

Pitfalls

- Different values (arbitrary-precision integers in Python, bounded in C)
- Different object representations (Python objects, C structs)
- Different runtime-errors (exceptions in Python)
- Garbage collection

Combining C and Python – first look

A combined static analysis of C/Python[12]

- Targeting C extensions using the CPython API

[12] Monat, Ouadjaout, and Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". SAS 2021.

Combining C and Python – first look

A combined static analysis of C/Python[12]

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the “glue”)

[12] Monat, Ouadjaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021.

Combining C and Python – first look

A combined static analysis of C/Python[12]

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the "glue")
- Observations

[12] Monat, Ouadjaout, and Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". SAS 2021.

Combining C and Python – first look

A combined static analysis of C/Python[12]

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the "glue")
- Observations
  - allocated objects are shared in the memory

[12] Monat, Ouadjaout, and Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". SAS 2021.

Combining C and Python – first look

A combined static analysis of C/Python[12]

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the "glue")
- Observations
  - allocated objects are <u>shared</u> in the memory
  - but each language has <u>different abstractions</u>

[12] Monat, Ouadjaout, and Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". SAS 2021.

Combining C and Python – first look

A combined static analysis of C/Python[12]

- Targeting C extensions using the CPython API
- To detect runtime errors (in C, Python, and the "glue")
- Observations
  - allocated objects are shared in the memory
  - but each language has different abstractions

$\Rightarrow$ Share universal domains and synchronize abstractions

[12] Monat, Ouadjaout, and Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". SAS 2021.

Combining C and Python – example

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject*
CounterIncr(Counter *self, PyObject *args)
{
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
       return NULL;

    self->count += i;
    Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
    return Py_BuildValue("i", self->count);
}
```

count.py

```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

Combining C and Python – example

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject*
CounterIncr(Counter *self, PyObject *args)
{
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
       return NULL;

    self->count += i;
    Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
    return Py_BuildValue("i", self->count);
}
```

count.py

```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$

Combining C and Python – example

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject*
CounterIncr(Counter *self, PyObject *args)
{
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
       return NULL;

    self->count += i;
    Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
    return Py_BuildValue("i", self->count);
}
```

count.py

```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$
- $32 \leq \texttt{power} \leq 64$: OverflowError: signed integer is greater than maximum
- $\texttt{power} \geq 64$: OverflowError: Python int too large to convert to C long

Combining C and Python – example

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject*
CounterIncr(Counter *self, PyObject *args)
{
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
       return NULL;

    self->count += i;
    Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
    return Py_BuildValue("i", self->count);
}
```

count.py

```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$
- $\texttt{power} = 31 \Rightarrow \texttt{r} = -2^{31}$
- $32 \leq \texttt{power} \leq 64$: OverflowError: signed integer is greater than maximum
- $\texttt{power} \geq 64$: OverflowError: Python int too large to convert to C long

How to analyze multilanguage programs?

Type annotations

```
class Counter:
  def __init__(self): ...
  def incr(self, i: int = 1): ...
  def get(self) -> int: ...
```

How to analyze multilanguage programs?

Type annotations

```
class Counter:
  def __init__(self): ...
  def incr(self, i: int = 1): ...
  def get(self) -> int: ...
```

- No raised exceptions $\Longrightarrow$ missed errors

How to analyze multilanguage programs?

Type annotations

```python
class Counter:
  def __init__(self): ...
  def incr(self, i: int = 1): ...
  def get(self) -> int: ...
```

- No raised exceptions $\implies$ missed errors
- Only types

How to analyze multilanguage programs?

Type annotations

```
class Counter:
  def __init__(self): ...
  def incr(self, i: int = 1): ...
  def get(self) -> int: ...
```

- No raised exceptions $\Longrightarrow$ missed errors
- Only types
- Typeshed: type annotations for the standard library

How to analyze multilanguage programs?

Type annotations

```
class Counter:
  def __init__(self): ...
  def incr(self, i: int = 1): ...
  def get(self) -> int: ...
```

- No raised exceptions $\implies$ missed errors
- Only types
- Typeshed: type annotations for the standard library, used in the single-language analysis before

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

```
class Counter:
  def __init__(self):
    self.count = 0
  def get(self):
    return self.count
  def incr(self, i=1):
    self.count += i
```

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

```
class Counter:
  def __init__(self):
    self.count = 0
  def get(self):
    return self.count
  def incr(self, i=1):
    self.count += i
```

- No integer wrap-around in Python

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

```
class Counter:
  def __init__(self):
    self.count = 0
  def get(self):
    return self.count
  def incr(self, i=1):
    self.count += i
```

- No integer wrap-around in Python
- Some effects can't be written in pure Python (e.g., read-only attributes)

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

Our approach

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

Our approach

- Analyze both the C and Python sources

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

Our approach

- Analyze both the C and Python sources
- Switch from one language to the other just as the program does

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

Our approach

- Analyze both the C and Python sources
- Switch from one language to the other just as the program does
- Reuse previous analyses of C and Python

How to analyze multilanguage programs?

Type annotations

Rewrite into Python code

Drawbacks of the current approaches

- Not the real code
- Not automatic: manual conversion
- Not sound: some effects are not taken into account

Our approach

- Analyze both the C and Python sources
- Switch from one language to the other just as the program does
- Reuse previous analyses of C and Python
- Detect runtime errors in Python, in C, and at the boundary

Analysis result

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject*
CounterIncr(Counter *self, PyObject *args)
{
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
       return NULL;

    self->count += i;
    Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
    return Py_BuildValue("i", self->count);
}
```

count.py

```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

Analysis result

counter.c

```c
1  typedef struct {
2      PyObject_HEAD;
3      int count;
4  } Counter;
5
6  static PyObject*
7  CounterIncr(Counter *self,
8  {
9      int i = 1;
10     if(!PyArg_ParseTuple(a
11        return NULL;
12
13     self->count += i;
14     Py_RETURN_NONE;
15 }
16
17 static PyObject*
18 CounterGet(Counter *self)
19 {
20     return Py_BuildValue("
21 }
```

count.py

```python
1  from counter import Counter
2  from random import randrange
```

```
⚠Check #430:
./counter.c: In function 'CounterIncr':
./counter.c:13.2-18: warning: Integer overflow

 13:    self->count += i;
        ^^^^^^^^^^^^^^^^
 '(self->count + i)' has value [0,2147483648] that is larger
   than the range of 'signed int' = [-2147483648,2147483647]
 Callstack:
       from count.py:8.0-8: CounterIncr


✗ Check #506:
count.py: In function 'PyErr_SetString':
count.py:6.0-14: error: OverflowError exception

 6: c.incr(2**p-1)
    ^^^^^^^^^^^^^^
 Uncaught Python exception: OverflowError: signed integer is greater than maximum
 Uncaught Python exception: OverflowError: Python int too large to convert to C long
 Callstack:
       from ./counter.c:17.6-38::convert_single[0]: PyParseTuple_int
       from count.py:7.0-14: CounterIncr
 +1 other callstack
```

Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics

Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions

Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views

Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary <u>views</u>
- <u>Boundary functions</u> when objects switch views for the first time

Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

Limitations

Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

Limitations

- Garbage collection not handled

Multilanguage semantics

Concrete definition

- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

Limitations

- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)

Multilanguage semantics

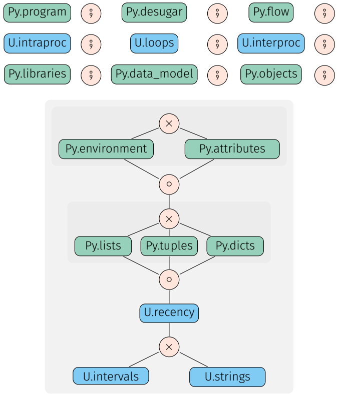
Concrete definition

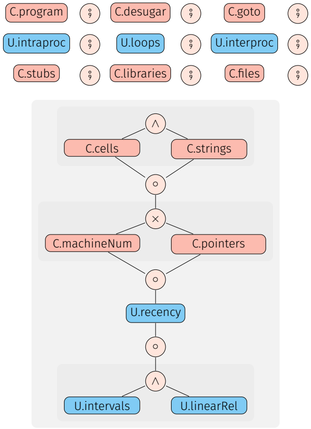
- Builds upon the Python and C semantics
- Defines the API: calls between languages, value conversions
- Shared heap, with disjoint, complementary views
- Boundary functions when objects switch views for the first time

Limitations

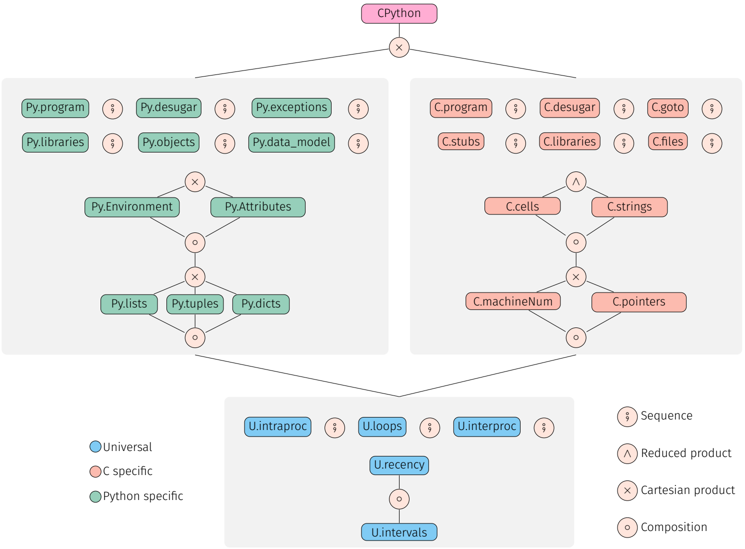
- Garbage collection not handled
- C access to Python objects only through the API (verified by Mopsa)
- Manual modelization from CPython's source code

From distinct Python and C analyses...

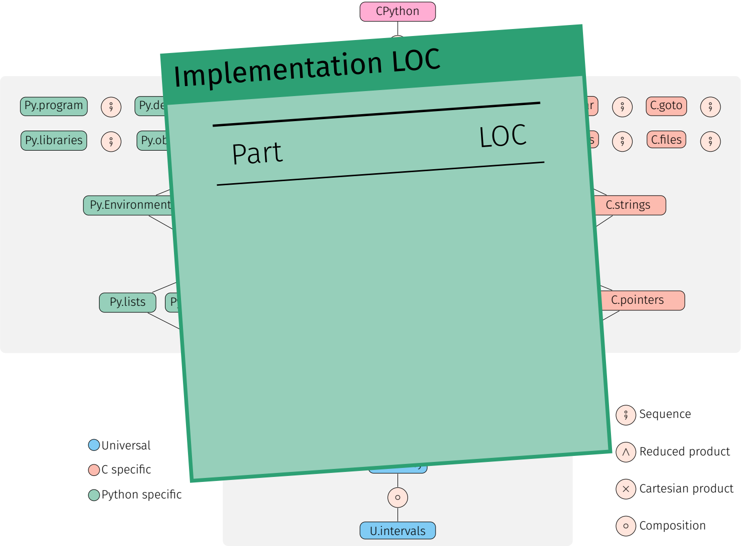




… to a multilanguage analysis!



… to a multilanguage analysis!



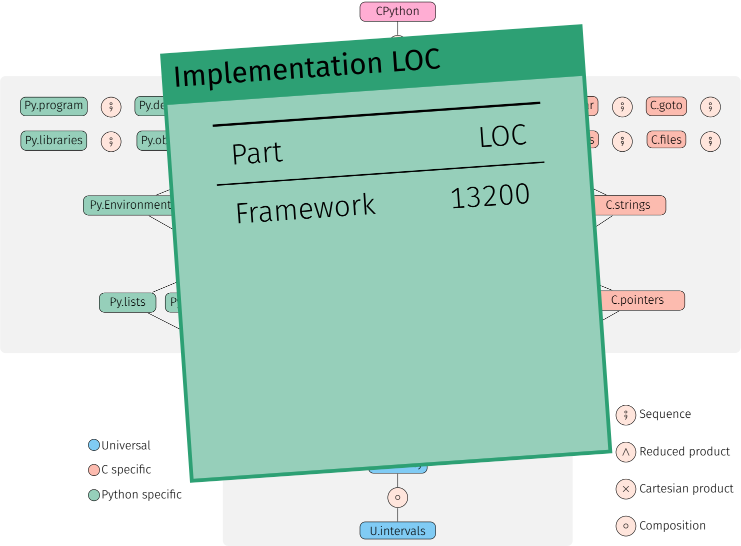
Implementation LOC

Part	LOC

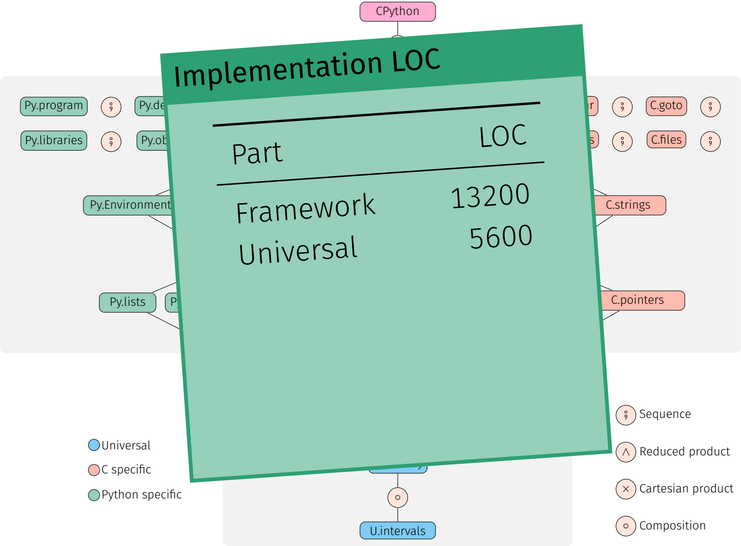
… to a multilanguage analysis!

Implementation LOC

Part	LOC
Framework	13200



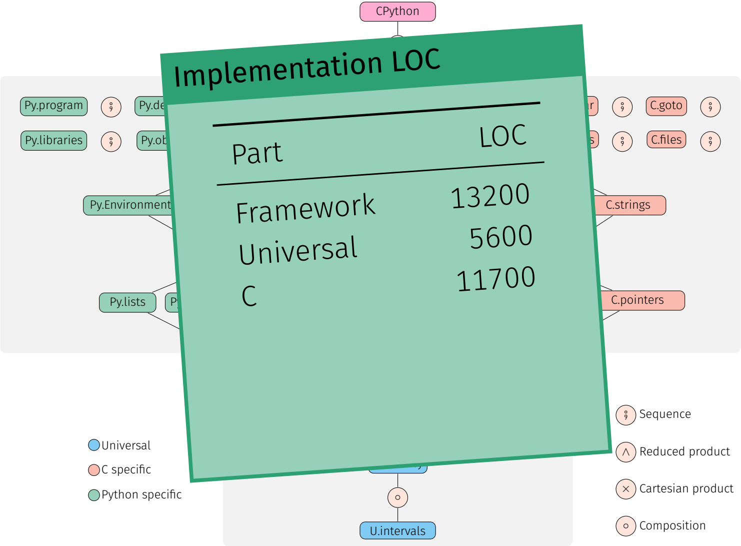
… to a multilanguage analysis!



Implementation LOC

Part	LOC
Framework	13200
Universal	5600

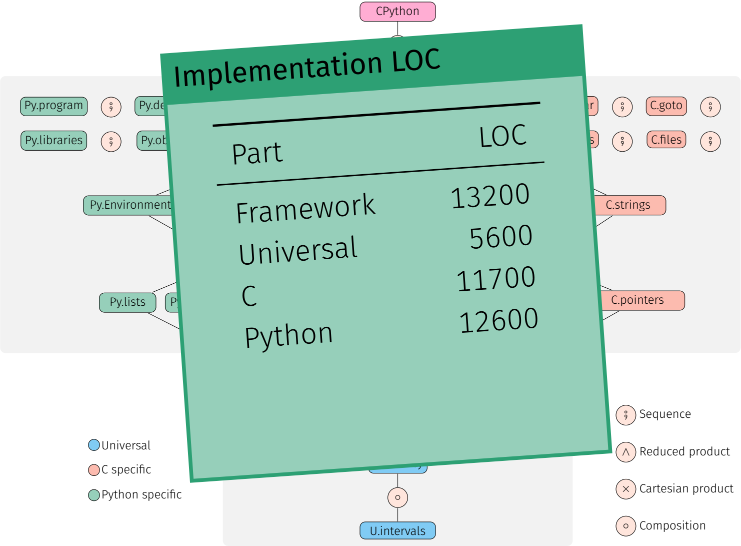
... to a multilanguage analysis!



Implementation LOC

Part	LOC
Framework	13200
Universal	5600
C	11700

... to a multilanguage analysis!



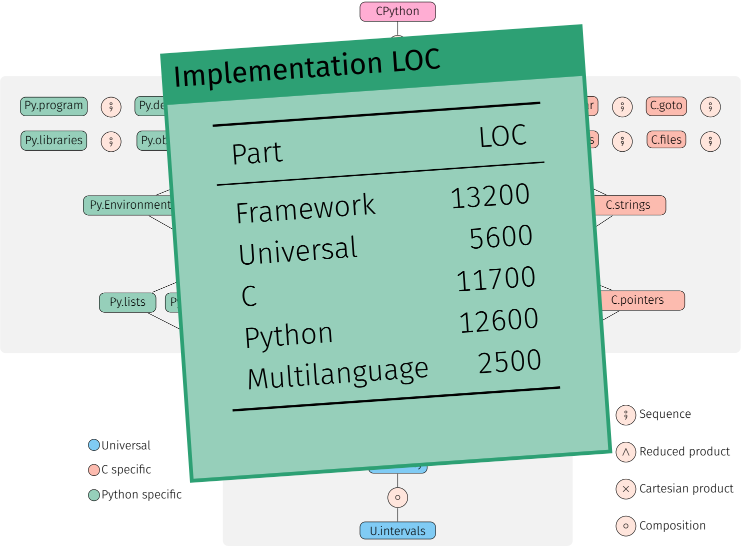
Implementation LOC

Part	LOC
Framework	13200
Universal	5600
C	11700
Python	12600

… to a multilanguage analysis!

Implementation LOC

Part	LOC
Framework	13200
Universal	5600
C	11700
Python	12600
Multilanguage	2500



Benchmarks

Corpus selection

- Popular, real-world libraries available on GitHub, averaging 412 stars.
- Whole-program analysis: we use the tests provided by the libraries.

Library	|C|	|Py|	Tests						Assertions	Py ⇝ C
`noise`	722	675	15/15	18s	99.6%	(4952)	100.0%	(1738)	0/21	6.5
`ahocorasick`	3541	1336	46/92	54s	93.1%	(1785)	98.0%	(4937)	30/88	5.4
`levenshtein`	5441	357	17/17	1.5m	79.9%	(3106)	93.2%	(1719)	0/38	2.7
`cdistance`	1433	912	28/28	1.9m	95.3%	(1832)	98.3%	(11884)	88/207	8.7
`llist`	2829	1686	167/194	4.2m	99.0%	(5311)	98.8%	(30944)	235/691	51.7
`bitarray`	3244	2597	159/216	4.2m	96.3%	(4496)	94.6%	(21070)	100/378	14.8

$\frac{\text{safe C checks}}{\text{total C checks}}\%$

total C checks

average # transitions between Python and C per test

Related work

Theoretical frameworks

- Matthews and Findler[13] boundary functions as value conversions between two languages.
- Buro, Crole, and Mastroeni[14] generic framework for combining analyses of different languages.

[13] Matthews and Findler. “Operational semantics for multi-language programs”. 2009.
[14] Buro, Crole, and Mastroeni. “On Multi-language Abstraction - Towards a Static Analysis of Multi-language Programs”. SAS 2020.

Related work (II)

Around the Java Native Interface (JNI)

Static translation of some of C's effects, injected back into the Java analysis.

- Effects of C code on Java heap modelized using JVML[15]
- Type inference of Java objects in C code[16]
- Extraction of C callbacks to Java[17]

- Modular analyses
- No numeric information
- Missing C runtime errors

[15] Tan and Morrisett. "Ilea: inter-language analysis across Java and C". OOPSLA 2007.

[16] Furr and Foster. "Checking type safety of foreign function calls". 2008.

[17] Lee, Lee, and Ryu. "Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis". ASE 2020.

Conclusion

Contribution: concrete semantics of Python

Difficulties

- Size of the semantics
- CPython's source code

Contribution: concrete semantics of Python

Difficulties

- Size of the semantics
- CPython's source code

Previous works

- Executable semantics of Python
- Handcrafted tests

Contribution: concrete semantics of Python

Difficulties

- Size of the semantics
- CPython's source code

Previous works

- Executable semantics of Python
- Handcrafted tests

Our results

- Semantics suitable for abstract interpretation
- Written and explained in the manuscript (70 cases)
- Backreferences to the source code
- Preliminary tests using CPython's suite

Contribution: type & value analyses of Python

Difficulties

- Dynamicity
- Dual type system
- Size of the semantics

[18] Monat, Ouadjaout, and Miné. "Static Type Analysis by Abstract Interpretation of Python Programs". ECOOP 2020.
[19] Monat, Ouadjaout, and Miné. "Value and allocation sensitivity in static Python analyses". SOAP@PLDI 2020.

Contribution: type & value analyses of Python

Difficulties

- Dynamicity
- Dual type system
- Size of the semantics

Previous works

- JS: type and constant analysis
- Python: no scalability or support of dynamicity

[18] Monat, Ouadjaout, and Miné. "Static Type Analysis by Abstract Interpretation of Python Programs". ECOOP 2020.
[19] Monat, Ouadjaout, and Miné. "Value and allocation sensitivity in static Python analyses". SOAP@PLDI 2020.

Contribution: type & value analyses of Python

Difficulties

- Dynamicity
- Dual type system
- Size of the semantics

Previous works

- JS: type and constant analysis
- Python: no scalability or support of dynamicity

Our results

- Type analysis[18],
- Numeric value analysis & new sensitivities for the recency abstraction[19]
- Relational value analysis with packing (manuscript)
- Scale to small, real-world benchmarks

[18] Monat, Ouadjaout, and Miné. "Static Type Analysis by Abstract Interpretation of Python Programs". ECOOP 2020.
[19] Monat, Ouadjaout, and Miné. "Value and allocation sensitivity in static Python analyses". SOAP@PLDI 2020.

Contribution: multilanguage Python/C analysis

Difficulties

- Concrete semantics
- Memory interaction

Monat, Ouadjaout, and Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. SAS 2021

Contribution: multilanguage Python/C analysis

Difficulties

- Concrete semantics
- Memory interaction

Previous works

- Type/exceptions analyses for the JNI
- No detection of runtime errors in C

Monat, Ouadjaout, and Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". SAS 2021

Contribution: multilanguage Python/C analysis

Difficulties

- Concrete semantics
- Memory interaction

Previous works

- Type/exceptions analyses for the JNI
- No detection of runtime errors in C

Our results

- Careful separation of the states and modelization of the API
- Lightweight domain on top of off-the-shelf C and Python analyses
- Shared underlying abstractions (numeric, recency)
- Scale to small, real-world libraries (using client code)

Monat, Ouadjaout, and Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". SAS 2021

Some future works

Executable concrete semantics

- Split soundness testing (CPython – concrete semantics – analyzer)
- Conformance tests

Some future works

Executable concrete semantics

- Split soundness testing (CPython – concrete semantics – analyzer)
- Conformance tests

Dictionary abstractions

- Beyond key/value summarization
- Empirical study of dictionary use (use of non-string keys)

Some future works

Executable concrete semantics

- Split soundness testing (CPython – concrete semantics – analyzer)
- Conformance tests

Dictionary abstractions

- Beyond key/value summarization
- Empirical study of dictionary use (use of non-string keys)

Multilanguage library analyses

- Other interoperability frameworks (Cffi, Swig, Cython)
- Infer Typeshed’s annotations
- Library analysis without client code

Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries

Questions

xkcd.com/353

Raphaël Monat

PhD defense
22 November 2021