Automated Type Inference for Python

Jifeng Wu, Caroline Lemieux

May 9, 2023

Software Practices Lab, University of British Columbia

Motivation

Python 3.5+ type annotations benefit:

- language safety [7]
- program documentation
- software testing [4]
- API recommendation [2, 3]

But they are:

- time-consuming to write [5]
- not widely adopted

Motivation

Python 3.5+ type annotations benefit:

- language safety [7]
- program documentation
- software testing [4]
- API recommendation [2, 3]

But they are:

- time-consuming to write [5]
- not widely adopted

Neural type recommendation [1, 6]

- textual hints
- correctness problem [8]

Static type recommendation[a b c d] [8]

- type checking and inference with typing rules or type seeds
- lack of type seeds or combinatorial explosion [8]
- too conservative [7], does not cover all language constructs [8]

[a] https://mypy.readthedocs.io/en/stable/index.html
[b] https://github.com/microsoft/pyright
[c] https://pyre-check.org/
[d] https://google.github.io/pytype/

Motivation

Python 3.5+ type annotations benefit:

- language safety [7]
- program documentation
- software testing [4]
- API recommendation [2, 3]

But they are:

- time-consuming to write [5]
- not widely adopted

Neural type recommendation [1, 6]

- textual hints
- correctness problem [8]

Static type recommendation[a][b][c][d] [8]

- type checking and inference with typing rules or type seeds
- lack of type seeds or combinatorial explosion [8]
- too conservative [7], does not cover all language constructs [8]

[a] https://mypy.readthedocs.io/en/stable/index.html
[b] https://github.com/microsoft/pyright
[c] https://pyre-check.org/
[d] https://google.github.io/pytype/

Python is a *duck-typed* language whose callables accept and return objects with certain *attributes*.

```
>>> def f(xs):
...     for x in xs: print(x)
...
>>> my_list = [1, 2]
>>> f(my_list)
1
2
>>> my_set = {1, 2}
>>> f(my_set)
1
2
>>> class MyClass:
...     def __iter__(self):
...         yield 1
...         yield 2
...
>>> f(MyClass())
1
2
```

Collect *attributes* to infer *types*!

Directly Accessed Attributes

```
import importlib.machinery
import os.path

def _strip_comments(source):
    buf = []
                        splitlines
    for line in source.splitlines():
                  strip
        l = line.strip()
                     startswith
        if not l or l.startswith(u'#'):
            continue
        buf.append(line)
    return u'\n'.join(buf)
```

Directly Accessed Attributes

```
import importlib.machinery
import os.path

def _strip_comments(source):
    buf = []
                        splitlines
    for line in source.splitlines():
                 strip
        l = line.strip()
                 startswith
        if not l or l.startswith(u'#'):
            continue
        buf.append(line)
    return u'\n'.join(buf)
```

Attributes Accessed Through Python Expressions (Subscriptions, Iterations, Unary and Binary Operations)

```
class LegacyModuleUtilLocator(ModuleUtilLocatorBase):
    # omitted code

    def _find_module(self, name_parts):
        rel_name_parts = self._get_module_utils_remainder_parts(name_parts)

        if len(rel_name_parts) == 1:
            paths = self._mu_paths
        else:
                                                __getitem__   __iter__
            paths = [os.path.join(p, *rel_name_parts[:-1]) for p in self._mu_paths]

            # omitted code
```

Alias Analysis 1

How do we infer attributes for the parameter *`name_parts`* of *`_find_module`*?

```
import importlib.machinery
import os.path

class ModuleUtilLocatorBase:
    def _find_module(self, name_parts):
        return False
```

How do we infer attributes for the parameter *name_parts* of *_find_module*?

```
import importlib.machinery
import os.path

class ModuleUtilLocatorBase:
    def _find_module(self, name_parts):
        return False
```

From an overwritten method in a derived class!

```
class LegacyModuleUtilLocator(ModuleUtilLocatorBase):
    def _find_module(self, name_parts):
        rel_name_parts = self._get_module_utils_remainder_parts(name_parts)

        if len(rel_name_parts) == 1:
            paths = self._mu_paths
        else:
            paths = [os.path.join(p, *rel_name_parts[:-1]) for p in self._mu_paths]

        # omitted code
```

Alias Analysis 2

How do we infer attributes for the arguments and returned values?

```
import importlib.machinery
import os.path

class LegacyModuleUtilLocator(ModuleUtilLocatorBase):
    def _find_module(self, name_parts):
        rel_name_parts = self._get_module_utils_remainder_parts(name_parts)

        # omitted code

        self.source_code = _slurp(path)
        return True
```

How do we infer attributes for the arguments and returned values?

```
import importlib.machinery
import os.path

class LegacyModuleUtilLocator(ModuleUtilLocatorBase):
    def _find_module(self, name_parts):
        rel_name_parts = self._get_module_utils_remainder_parts(name_parts)

        # omitted code

        self.source_code = _slurp(path)
        return True
```

By associating them with the parameters and return values of their definitions!

```
import importlib.machinery
import os.path

def _slurp(path):
    if not os.path.exists(path):
        # omitted code
    with open(path, 'rb') as fd:
        data = fd.read()
    return data

class LegacyModuleUtilLocator(ModuleUtilLocatorBase):
    def _get_module_utils_remainder_parts(self, name_parts):
        return name_parts[2:]
```

Assignments and Binary Operators

```
import importlib.machinery
import os.path

class LegacyModuleUtilLocator(ModuleUtilLocatorBase):
    def _find_module(self, name_parts):
        # omitted code

        if len(rel_name_parts) == 1:
            paths = self._mu_paths
        else:
            paths = [os.path.join(p, *rel_name_parts[:-1]) for p in self._mu_paths]

        self._info = info = importlib.machinery.PathFinder.find_spec(
            '.'.join(name_parts), paths
        )
        if info is not None \
        and os.path.splitext(info.origin)[1] in importlib.machinery.SOURCE_SUFFIXES:
            self.is_package = info.origin.endswith('/__init__.py')
            path = info.origin

        # omitted code
```

1. Python Modules

Our Proposed Approach

1. Python Modules

2. Numba SSA IR

```
Result.__init__:
0:
self = arg(0, self)
value = arg(1, value)
pos = arg(2, pos)
(self).value = value
(self).pos = pos
$const14.4 = const(NoneType, None)
$16return_value.5 = cast($const14.4)
return $16return_value.5
```

Our Proposed Approach

THE UNIVERSITY OF BRITISH COLUMBIA

1. Python Modules

2. Numba SSA IR

```
Result.__init__:
0:
self = arg(0, self)
value = arg(1, value)
pos = arg(2, pos)
(self).value = value
(self).pos = pos
$const14.4 = const(NoneType, None)
$16return_value.5 = cast($const14.4)
return $16return_value.5
```

3. Alias Analysis

```
def f(x, y):
    # ...
    return ...

def g(x, y):
    # ...
    c = f(a, b)
    # ...
```

Our Proposed Approach

THE UNIVERSITY OF BRITISH COLUMBIA

1. Python Modules

2. Numba SSA IR

```
Result.__init__:
0:
self = arg(0, self)
value = arg(1, value)
pos = arg(2, pos)
(self).value = value
(self).pos = pos
$const14.4 = const(NoneType, None)
$16return_value.5 = cast($const14.4)
return $16return_value.5
```

3. Alias Analysis

```
def f(x, y):
    # ...
    return ...

def g(x, y):
    # ...
    c = f(a, b)
    # ...
```

4. Typeshed[a] Lookup

```
re.Pattern        str | bytes
regex = re.compile(pattern)
re.Match          str|bytes, int
match = regex.match(char, pos)
```

[a] a collection of type stubs for callables within the Python standard library

Our Proposed Approach

THE UNIVERSITY OF BRITISH COLUMBIA

1. Python Modules

2. Numba SSA IR

```
Result.__init__:
0:
self = arg(0, self)
value = arg(1, value)
pos = arg(2, pos)
(self).value = value
(self).pos = pos
$const14.4 = const(NoneType, None)
$16return_value.5 = cast($const14.4)
return $16return_value.5
```

3. Alias Analysis

```
def f(x, y):
    # ...
    return ...

def g(x, y):
    # ...
    c = f(a, b)
    # ...
```

4. Typeshed[a] Lookup

```
re.Pattern          str | bytes
regex = re.compile(pattern)
re.Match            str|bytes, int
match = regex.match(char, pos)
```

[a] a collection of type stubs for callables within the Python standard library

5. Attribute Collection

```
        name          __contains__
if self.name in env:
                      __getitem__
    return env[self.name]
```

Our Proposed Approach

THE UNIVERSITY OF BRITISH COLUMBIA

1. Python Modules

2. Numba SSA IR

```
Result.__init__:
0:
self = arg(0, self)
value = arg(1, value)
pos = arg(2, pos)
(self).value = value
(self).pos = pos
$const14.4 = const(NoneType, None)
$16return_value.5 = cast($const14.4)
return $16return_value.5
```

3. Alias Analysis

```
def f(x, y):
    # ...
    return ...

def g(x, y):
    # ...
    c = f(a, b)
    # ...
```

4. Typeshed[a] Lookup

```
regex = re.compile(pattern)
match = regex.match(char, pos)
```

re.Pattern

str | bytes

re.Match

str|bytes, int

[a] a collection of type stubs for callables within the Python standard library

5. Attribute Collection

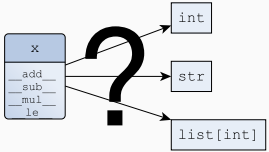
```
if self.name in env:
    return env[self.name]
```

name

__contains__

__getitem__

6. Type Query



TF-IDF Weights for Attributes

```
class int:
    def __init__(...): ...
    def __le__(...): ...
    def __add__(...): ...
    def __mul__(...): ...
    def __sub__(...): ...
    def __pow__(...): ...
```

```
class list:
    def __init__(...): ...
    def __le__(...): ...
    def __iter__(...): ...
    def __len__(...): ...
    def append(...): ...
    def extend(...): ...
```

```
class set:
    def __init__(...): ...
    def __le__(...): ...
    def __iter__(...): ...
    def __len__(...): ...
    def add(...): ...
    def update(...): ...
```

- rare attributes more suggestive of specific types
- types and type queries as N-dimensional vectors
- querying types as k-nearest neighbor search

Type Query

TF-IDF Weights for Attributes

```
class int:
    def __init__(...): ...
    def __le__(...): ...
    def __add__(...): ...
    def __mul__(...): ...
    def __sub__(...): ...
    def __pow__(...): ...
```

```
class list:
    def __init__(...): ...
    def __le__(...): ...
    def __iter__(...): ...
    def __len__(...): ...
    def append(...): ...
    def extend(...): ...
```

```
class set:
    def __init__(...): ...
    def __le__(...): ...
    def __iter__(...): ...
    def __len__(...): ...
    def add(...): ...
    def update(...): ...
```

Importing Modules Suggests Types Within Those Modules are Used

```
import collections -> collections.queue, collections.Counter

import ansible.playbook.handler -> ansible.playbook.handler.Handler
import ansible.playbook.task -> ansible.playbook.task.Task
import ansible.template -> ansible.template.Templar
import ansible.utils.display -> ansible.utils.display.Display
```

- rare attributes more suggestive of specific types
- types and type queries as *N*-dimensional vectors
- querying types as *k*-nearest neighbor search
- calculate the smallest Levenshtein distance between *the module of a type* and *any imported module*
- prioritize candidate types whose modules are similar to an imported module

M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao.
Typilus: Neural type hints.
In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105, 2020.

X. He, L. Xu, X. Zhang, R. Hao, Y. Feng, and B. Xu.
Pyart: Python api recommendation in real-time.
In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1634–1645. IEEE, 2021.

G. Li, H. Liu, G. Li, S. Shen, and H. Tang.
Lstm-based argument recommendation for non-api methods.
Science China Information Sciences, 63:1–22, 2020.

S. Lukasczyk and G. Fraser.
Pynguin: Automated unit test generation for python.
In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.


J.-P. Ore, S. Elbaum, C. Detweiler, and L. Karkazis.
Assessing the type annotation burden.
In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 190–201, 2018.


M. Pradel, G. Gousios, J. Liu, and S. Chandra.
Typewriter: Neural type prediction with search-based validation.
In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 209–220, 2020.


I. Rak-Amnouykit, D. McCrevan, A. Milanova, M. Hirzel, and J. Dolby.
Python 3 types in the wild: a tale of two type systems.
In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 57–70, 2020.


K. Sun, Y. Zhao, D. Hao, and L. Zhang.
Static type recommendation for python.
In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.