Utrecht University

Object-sensitive Type Analysis for Python

Xulei Liu
7009356

A thesis
submitted in partial fulfillment of the
requirements for the degree of
Master of Science

Supervisors:

Utrecht University	Prof. Dr. Jurriaan Hage
Utrecht University	Dr. Fernando Castor

Department of Information and Computing Sciences
Faculty of Science
Utrecht University
Friday 14^{th} October, 2022

Abstract

Python is an interpreted, interactive and object-oriented programming language that stresses code readability. In Python, objects interact with each other to accomplish various tasks. Such interaction is usually achieved by attribute lookup, storage and deletion.

We consider whether Python's attribute access semantics with different precision affect the precision of type analysis. Type inference for Python is hard due to the extensive use of external libraries and the dynamic language features. In this thesis we propose an object-sensitive type analysis for Python based on an extension of the notion of monotone frameworks to deal with dynamic flow manipulation. In addition, we also implement a type parser to partially support retrieving types from Python stub files. Our results show that the analysis precision is not improved substantially when employing sophisticated attribute access semantics.

Contents

1 Introduction

The attractiveness of dynamically typed languages like Python and Ruby stems from the flexible programming features they support. These include allowing one variable to take values of different types in different program locations. However, this comes at a price of losing the advantages of static typing. Programmers using these languages often suffer from the lack of type information during development. The enforcement of static typing may bring some benefits to dynamically typed languages:

- Static typing can make programs easier to read. In large projects, code is constantly modified and types can serve as documentation.
- Static typing can detect program bugs earlier. For instance, a portion of type errors will be reported before execution.
- Static typing can assist development tools like Integrated Development Environment (IDE). For example, IntelliJ IDEA can offer the information of refactoring candidates based on types.

We would like to mitigate the problems of dynamically typed languages described above and gain the advantages of static typing by performing type inference at compile time. The enforcement of static typing for dynamic languages has been widely studied in the academic world. In this thesis our contributions are summarized as follows:

1. We introduce the notion of a dynamic monotone framework to deal with dynamic flow discovery during data flow analysis.
2. We purpose a type analysis for Python as an instance of a dynamic monotone framework.
3. We implement a type inferencer on the basis of the type analysis for Python.
4. We evaluate the precision of the analysis result based on the coarseness of attribute access semantics.

The result of our work may be beneficial to type-related tools for Python. A majority of these tools are gradual type checkers. A gradual type checker [Siek and Taha, 2007] only type checks statically typed part of a program. Dynamically typed part of a program is considered to have a special type `Any` which is consistent with every type and vice versa. Our type inferencer is capable of inferring types from dynamically typed part of a program.

2 The Python Programming Language

Python[1] is a high-level, general-purpose programming language that stresses code readability. It supports procedural, object-oriented and functional programming paradigms. Python is considered versatile and widely used in artificial intelligence, web development, desktop programs and system applications. Python is now maintained by the *Python Software Foundation*.

2.1 History

Python was created in the early 1990s by *Guido van Rossum* at *Stichting Mathematisch Centrum* in the Netherlands. It was a successor of a language called ABC[2] but equipped with a simpler runtime.

The first release (version 0.9.0) was published in 1991. It had features such as classes, exception handling, and the core data types of list, dict and so on. In 1994, version 1.0 was released. The major new features of this release were the functional programming tools such as lambda, filter and reduce.

Python 2.0 released in 2000 introduced list comprehensions which were presented in functional programming languages like Haskell. In addition, it also added a full garbage collector. At this time, Python was evolving towards a reliable language.

Python 3.0 (also known as Python 3000 and Py3K) was available from December 3, 2008. The emphasis in Python 3.x had been on rectifying fundamental design flaws in Python 2.x, which made it backward-incompatible. However, Python 3.x came close to fulfilling a law of the Zen of Python[3]: "There should be one – and preferably only one – obvious way to do it.". Now the latest stable Python version is 3.10.6 released on August 2, 2022.

2.2 Implementations

CPython[4] is the reference implementation of Python written in C and Python. It can be regarded as both a compiler and an interpreter as it compiles Python code into bytecode which is then interpreted by the CPython virtual machine. There are a number of alternative implementations as well. For instance, Jython[5] is an alternative implementation written in Java and provides Python with a Java Virtual Machine Environment. PyPy[6] is another implementation that usually runs faster than CPython because it uses a just-in-time compiler.

In the rest of this thesis, if not mentioned explicitly, CPython is used as the default implementation.

[1] https://www.python.org/
[2] https://homepages.cwi.nl/~steven/abc/
[3] https://peps.python.org/pep-0020/
[4] https://github.com/python/cpython
[5] https://www.jython.org/
[6] https://www.pypy.org/

2.3 Data model

A *data model* arranges data elements and specifies how the data elements interact with one another in the course of computing.

2.3.1 Objects, values and types

In Python, data elements are abstracted as objects. Namely, everything is an *object*. Code Listing 1 defines a function object and an `int` object. On the contrary, Java does distinguish primitive types (`int`, `float`, etc.) from object types (`Integer`, `Float`, etc.).

Listing 1 Everything in Python is an object

```
# a_func is a function object
def a_func():
    pass

# a_value is an int object
a_value = 1
```

Each object has an *identity*, a *type* and a *value*. An object's identity can never change since its creation. The `is` operator compares the identity of two objects. For CPython, the identity of an object is its memory address.

An object's type determines the operations it supports. For instance, an `int` object and a `float` object can both be cast to a `bool` object but they both do not support random access.

The value of some objects may be unchangeable. Such objects are called *immutable*. Other objects are classified as *mutable*. An object's type determines mutability. As an example, `int` objects are immutable, while `list` objects are mutable. One special case is `tuple`. A `tuple` object is an immutable container but it may contain a reference to a mutable object. Therefore its value changes if the mutable object is changed.

An object in Python can not be destroyed manually. The Python garbage collector is in charge of reclaiming an object's memory resource when the object becomes unreachable. Currently, CPython uses a reference-counting scheme to decide when to garbage-collect objects.

2.3.2 Special methods

Special methods are a set of predefined methods that programmers can use to enrich class behaviors. Since they start and end with double underscores, such as `__init__` or `__len__`, special methods are also called *dunder methods* or *magic methods*. The functionality of special methods is similar to operator overloading, which allows the same operator to have different semantics. For instance, a custom class instance can perform addition by supporting the `__add__` special method.

Emulating numeric types

Table 1 and 2 display three sets of special methods of Python to emulate numeric types. The first column is the representation of arithmetic operators in the Python *abstract syntax tree*. The last column gives the arithmetic operators used by programmers. Other columns are the special methods used by Python under the hood to support arithmetic operations.

Abstract syntax node	Ordinary name	Reversed name	Symbol
`ast.Add`	`__add__`	`__radd__`	+
`ast.Sub`	`__sub__`	`__rsub__`	-
`ast.Mult`	`__mul__`	`__rmul__`	*
`ast.Div`	`__truediv__`	`__rtruediv__`	/
`ast.FloorDiv`	`__floordiv__`	`__rfloordiv__`	//
`ast.Mod`	`__mod__`	`__rmod__`	%
`ast.Pow`	`__pow__`	`__rpow__`	**
`ast.Lshift`	`__lshift__`	`__rlshift__`	<<
`ast.RShift`	`__rshift__`	`__rrshift__`	>>
`ast.BitAnd`	`__and__`	`__rand__`	&
`ast.BitXor`	`__xor__`	`__rxor__`	^
`ast.BitOr`	`__or__`	`__ror__`	|

Table 1: Ordinary and reversed arithmetic operators

Abstract syntax node	Augmented name	Symbol
`ast.Add`	`__iadd__`	+=
`ast.Sub`	`__isub__`	-=
`ast.Mult`	`__imul__`	*=
`ast.Div`	`__itruediv__`	/=
`ast.FloorDiv`	`__ifloordiv__`	//=
`ast.Mod`	`__imod__`	%=
`ast.Pow`	`__ipow__`	**=
`ast.Lshift`	`__ilshift__`	<<=
`ast.RShift`	`__irshift__`	>>=
`ast.BitAnd`	`__iand__`	&=
`ast.BitXor`	`__ixor__`	^=
`ast.BitOr`	`__ior__`	|=

Table 2: Augmented arithmetic operators

Emulating rich comparison

Table 3 shows six special methods to support comparison operations. It is worthwhile to mention that `is` and `is not` are predefined in the Python interpreter. So they do not have corresponding special methods.

Abstract syntax node	Name	Symbol
`ast.Lt`	`__lt__`	`<`
`ast.Le`	`__le__`	`<=`
`ast.Eq`	`__eq__`	`==`
`ast.NotEq`	`__ne__`	`!=`
`ast.Gt`	`__gt__`	`>`
`ast.Ge`	`__ge__`	`>=`
`ast.Is`		`is`
`ast.IsNot`		`is not`

Table 3: Rich comparison operators

Emulating container types

Table 4 shows special methods for emulating container types. The first column denotes the cases where calls to these methods may happen.

Statement	Name
`len(container)`	`__len__`
`length_hint(container)`	`__length_hint__`
`container[i]`	`__getitem__`
`container[i] = b`	`__setitem__`
`del container[i]`	`__delitem__`
`container[i]`	`__missing__`
`iter(container)`	`__iter__`
`reversed(container)`	`__reversed__`
`elt in container`	`__contains__`

Table 4: Special methods for emulating container types

As an example, executing `container[i]` leads to the special method call `__getitem__(container, i)`. Code Listing 2 shows a rough implementation of `__getitem__()`. It can be seen that if `i` is not present in `container`, `__getitem__(container, i)` will internally invoke `__missing__(container, i)` (if present) to obtain the missing value and then store the value into `container` with key `i`.

2.3.3 Special attributes

Special attributes are attributes that are usually accessed by the implementation and are not intended for general use. Programmers should not depend on these attributes since they are not guaranteed to be stable in future versions. Table 5 shows some frequently used special attributes in Python.

Listing 2 How __getitem__ works

```
def __getitem__(self, key):
    if key is not in self:
        # if key is not found
        obj = object()
        # retrieve __missing__
        __missing__ = getattr(type(self), "__missing__", obj)
        # if __missing__ is not found in self
        if __missing__ is obj:
            raise KeyError
        else:
            # if __missing__ is found
            value = __missing__(self, key)
            __setitem__(self, key, value)
    return self[key]
```

Name	Meaning
`__bases__`	a tuple containing the base classes
`__mro__`	a tuple containing the superclass linearization
`__module__`	the name of the module in which the object is defined
`__class__`	the class to which the class instance belongs
`__dict__`	a dictionary storing an object's attributes

Table 5: Some special attributes

2.3.4 Class creation

A *class* is a blueprint that defines what data and methods its instances have. A *class definition* defines a user-defined class object. By default a class is constructed by means of the metaclass `type`. Metaclasses are classes that create other classes. That is, they are classes' classes.

From the perspective of types, the type hierarchy in Python is shown in Figure 1. A class instance's type is its class. A class's type is its metaclass. A metaclass's type is the metaclass itself.

From the perspective of instances, the instance hierarchy in Python is shown in Figure 2. A metaclass can create a class object. A class object can create a class instance. A class object named `A_Class` can be created by a class definition `class A_Class: pass` or by a metaclass `type("A_Class", (), {})`.

Classes support multiple inheritance. Python follows the *method resolution order* (MRO) constructed by applying the C3 linearization algorithm[7] to search for a specific attribute. The output of the algorithm is stored in the special attribute `__mro__` of the class being initialized.

[7] https://www.python.org/download/releases/2.3/mro/

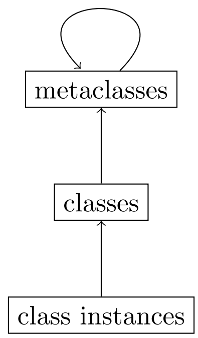


Figure 1: The type hierarchy of metaclasses, classes, and class instances

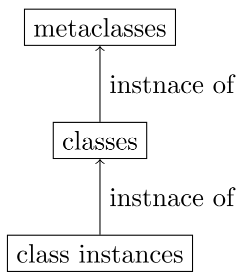


Figure 2: The instance hierarchy of metaclasses, classes, and class instances

2.3.5 Built-in functions

The Python interpreter has a set of built-in functions[8] (listed in Table 6) that are readily available for use. Most functions are extensions of special methods. For instance, `a = x.y` is equivalent to `a = getattr(x, "y")`. But the latter is more powerful since it has the third parameter `default`, which is returned if `x` has no `y`. In this way, conditional statements can be eliminated. Code Listing 2 has demonstrated one usage of `getattr`. In addition, some built-in functions are used to access system resources, such as `open()` to read or write system files.

2.4 The Python execution model

A Python *program* consists of code blocks. A *code block* is a group of statements executed as a unit. For instance, a Python script file is a code block. A function body is also a code block. A code block is executed in an *execution frame*. An execution frame records run-time information affecting the execution of a code block.

[8] https://docs.python.org/3.7/library/functions.html

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Table 6: Built-in functions

2.4.1 Naming and binding

An *identifier* in Python is called a *name*. A name is simply a human-readable string connected to an object. Names are introduced by name binding operations. Some constructs that could bind names are listed below. For instance, the statement `import mod` binds name `mod` to a module object.

- function parameters.
- class definitions.
- function definitions.
- assignment statements.
- import statements.

A *namespace* is a mapping from names to objects. Namespaces play a very import role in Python. As the *Zen of Python* said, “Namespaces are one honking great idea – let’s do more of those!”. Different namespace can co-exist at a given time but they are completely separated. Adding a name to a namespace is called *binding*. Changing the mapped object of a name in a namespace is called *rebinding* and removing a name is called *unbinding*. At present Python’s namespaces are implemented as dictionaries.

2.4.2 Name resolution

The *scope* of a name defines a region where the name can be accessed unambiguously. The *LEGB* rule is a name lookup procedure and determines the order in which the namespaces are searched during name resolution. The letters in LEGB stand for *Local Scope*, *Enclosing Scope*, *Global Scope* and *Built-in Scope*:

- Local scope is the current namespace.
- Enclosing scope is any namespace that encloses the current namespace, which may or may not exist.
- Global scope is the module-level namespace.
- Built-in scope is the namespace that is built into the Python interpreter. All built-in functions exist in the built-in scope.

When searching for an object by name, let's say `x`, Python first searches the local namespace. If `x` is not present, Python then searches any enclosing namespaces, starting from the enclosing namespace of the local namespace. Once all enclosing namespaces are searched but `x` is still not found, Python performs the search in the global namespace. If `x` is not defined in the global namespace, Python looks `x` up in the built-in namespace as a last resort. At last, if Python can not find `x`, an exception `NameError` would be raised. Figure 3 shows how the searching works. Code Listing 3 shows when executes `d = a`, `a` will be found in the outermost enclosing namespace.

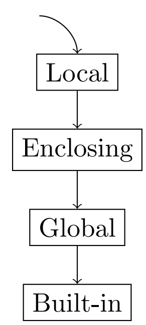


Figure 3: The LEGB rule

Python defines three types of *variables* based on how they are created and used. A name is a *local* variable in a block if it is bound to that block, unless declared as `nonlocal` or `global`. *Nonlocal* variables refer to those names defined within the nesting functions. Any name bound at the module level is a *global* variable. In the rest of this thesis, we use names and variables interchangeably.

One thing merits a mention. Though scopes are used dynamically, they are determined statically. That is, each variable in a program has been resolved at compile time by inspection only of the program's text.

Listing 3 Searches in enclosing namespaces

```python
# Two enclosing namespaces for the body of enclosing3
# (1) the body of enclosing1 containing an int object a
# (2) the body of enclosing2 containing an int object b
def enclosing1():
    a = 1
    def enclosing2():
        b = 2
        def enclosing3():
            c = 3
            d = a
        enclosing3()
    enclosing2()
enclosing1()
```

2.5 Type annotations and type hints

PEP 3107[9] introduced syntax for function annotations but the semantics were left undefined. Code Listing 4 shows an annotated function. Afterwards, PEP 484[10] standardized the syntax for *type annotations* and thus introduced *type hints*. Type hints are used to hint what the type of a variable should be expected to be. In this regard, Code Listing 4 states that the expected type of `name` and the return of `print_brand` are both `str`.

Listing 4 A function annotation

```python
def print_brand(name: str) -> str:
    return "This is " + name
```

Sometimes it is more appropriate to represent type hints in separately files ended with ".pyi". Such files are named *stub files*. They contain type hints that are only used by type checkers. Stub files basically have the same syntax as regular Python modules. Code Listing 5 defines a small stub file for Code Listing 4. In the rest of this thesis, we will use type hints to denote the type of a variable.

Listing 5 A stub file for `print_brand`

```python
def print_brand(name: str) -> str: ...
```

[9] https://peps.python.org/pep-3107/
[10] https://peps.python.org/pep-0484/

2.6 Language features

This section gives an overview of the Python programming language, in particular those features that are relevant to type analysis.

2.6.1 Type system

Python is a strongly, dynamically typed language. *Strong typing* means Python keeps track of all variable types. *Dynamic typing* means Python does not know the type of an object before the execution of code. Code Listing 6 demonstrates the above two properties.

Listing 6 Strong and dynamic typing

```python
# strong typing
res: str = "str"
res: int = 1 + 3

# dynamic typing
def different_return_value(flag) -> int | str:
    if flag:
        return 1
    else:
        return "string"

res: int = different_return_value(True)
res: str = different_return_value(False)
```

In addition, Python employs *duck typing* to determine whether an object can be used for a particular operation. The name comes from the duck test — "If it looks like a duck and it quacks like a duck, then it must be a duck". In short, in duck typing, only what objects can do are of interest, instead of what they are.

2.6.2 Function definitions

A *function* is a group of statements that performs a specific task. A *function definition* defines a user-defined function object. When a function definition is executed, the function name is bound to the function object in the current local namespace. Populating a function definition does not involve executing the function body. The latter will only be executed when the function gets called.

A function definition may have five types of *parameters*: positional or keyword parameters, keyword-only parameters, default parameters, arbitrary positional parameters and arbitrary keyword parameters. Figure 4 shows four types of parameters. Figure 5 shows keyword-only parameters that are prefixed with a $*$. Be aware that default arguments will be evaluated when a function definition is encountered.

```
def f(a, b, c=1, d=2, *args, **kwargs):
      ----  ---------- ----    -------
        |              |     |      |
positional or keyword|  arbitrary|positional
                       |            |
                  default   arbitrary keyword
```

Figure 4: Positional or keyword, default, arbitrary positional and arbitrary keyword parameters

```
def f(a, b, c, *, d, e):
      -------     -----
        |             |
positional or keyword |
                      - keyword-only
```

Figure 5: Positional or keyword and keyword-only parameters

2.6.3 Function calls

All its arguments are evaluated before a call is made. A call expression may have two types of *arguments*: positional arguments and keyword arguments.

Python has a set of rules to process arguments. At first, a list of unfilled slots is created for the formal parameters. Next, place positional arguments into these slots. After that, values of keyword arguments are placed into the slots according to the formal parameter names. If there is still any unfilled slot, it is filled with the default value from the function definition.

If there are more positional arguments than there are formal parameter slots and an arbitrary positional parameter is present, that parameter receives a tuple containing the excess positional arguments. If any keyword argument does not correspond to a formal parameter name and an arbitrary keyword parameter is present, that parameter receives a dictionary containing such excess keyword arguments.

Figure 6 gives an example of how function call semantics work. It is described in more detail in Calls[11].

2.6.4 Modules

A *module* refers to a file containing Python statements and definitions. A module name is a file name without the suffix `.py`. For instance, `example.py` is a module named `example`. A *package* is a directory that contains a collection of modules and has one additional `__init__.py`. A package can be nested into other packages. If so, nested modules and packages are named by using *dotted*

[11] https://docs.python.org/3.7/reference/expressions.html#calls

```python
def func(a, b, c=1, d=2, *args, **kwargs):
    ...

# original call expression
func(1, 2, d=5, c=1, address="Utrecht")

# positional arguments a and b are filled
a = 1, b = 2

# keyword arguments c and d are filled
c = 1, d = 5

# arbitrary positional arguments args is filled
args = ()

# arbitrary keyword arguments kwargs is filled
kwargs = {"address": "Utrecht"}

# so the function call is
func(1, 2, 1, 5, (), {"address":"Utrecht"})
```

Figure 6: An example of how Python processes the arguments of a call expression

module names. Figure 7 shows two packages `foo` and `foo.bar` and one module `foo.bar.example`. For details, refer to Modules[12].

```
foo # a package named foo
  -- __init__.py
  -- bar # a package named foo.bar
    -- __init__.py
      -- example.py # a module named foo.bar.example
```

Figure 7: Modules and packages

Packages are just special modules. The `import` keyword can import names within one module into other modules. For example, `import foo` introduces the module `foo` into the current local scope. In general a module is loaded only once before the Python interpreter exits. But programmers may use `importlib.reload()` to carry out the module initialization process again. The import system[13] and importlib[14] provide more information on the semantics of

[12] https://docs.python.org/3.7/tutorial/modules.html
[13] https://docs.python.org/3/reference/import.html
[14] https://docs.python.org/3/library/importlib.html

import.

2.7 Attribute access

Everything in an object is an attribute. Code Listing 7 shows a class object Attr with two attributes dummy and version.

Listing 7 A class with two attributes

```
class Attr:
    def dummy(self):
        pass

    version = "1.0"
```

2.7.1 Basic attribute access

In Python an attribute of an object may be called by dotted-syntax. For instance, obj.attr means looking attr up in obj. When an object does not contain the attribute, Python will raise AttributeError. For instance, based on Code Listing 7, Attr.version succeeds but Attr.unknown fails. Since unknown is not a valid attribute to Attr, Python raises AttributeError.

An attribute can also be created, overwritten or deleted by dotted syntax. For instance, Attr.unknown = 1 creates the new attribute unknown on Attr, Attr.dummy = "dummy" overwrites the value of dummy and del Attr.version deletes version from Attr.

Python's built-in module provides four functions hasattr, getattr, setattr, delattr for programmers to perform attribute access uniformly. For instance, getattr(Attr, "dummy") is semantically equivalent to Attr.dummy.

The default behavior for attribute access is to get, set or delete an attribute from an object's dictionary. However, the actual process in Python is more complex than that. Suppose obj is a class instance, obj.attr has a lookup chain starting from the method resolution order of its class (type(obj).__mro__) till the instance dictionary obj.__dict__. Code Listing 8 states how a name is looked up within __mro__. Code Listing 9 shows the default behavior of an attribute lookup process of an object in Python.

Listing 8 find_name_in_mro

```
def find_name_in_mro(cls, name, default):
    for base in cls.__mro__:
        if name in vars(base):
            return vars(base)[name]
    return default
```

Listing 9 The default behavior of an attribute lookup process

```
def basic_object_getattribute(obj, name):
    objtype = type(obj)
    null = object()
    cls_var = find_name_in_mro(objtype, name, null)

    if hasattr(obj, '__dict__') and name in vars(obj):
        return vars(obj)[name] # instance variable

    if cls_var is not null:
        return cls_var # class variable

    raise AttributeError(name)
```

2.7.2 Descriptors

It happens very often that the value of an attribute comes from other attributes. *Descriptors* provide a way to realize this requirement. A descriptor is an object that fulfills the *descriptor protocol* which consists of three special methods `__get__`, `__set__` and `__delete__`. Code Listing 10 shows the function signatures of the descriptor protocol.

Listing 10 Descriptor protocol

```
__get__(self, obj, type=None) -> value
__set__(self, obj, value) -> None
__delete__(self, obj) -> None
```

An object is considered a *data descriptor* if it defines `__set__` or `__delete__` and a *non-data descriptor* if it defines `__get__` only. Data descriptors and non-data descriptors differ in how Python resolves object's attribute access.

Descriptor invocation

A descriptor can be invoked directly just like any other function or method. But the preferred way is to invoke descriptors by an attribute access process automatically. Code Listing 11 and 12 show Python equivalents of `__getattribute__` and `__setattr__` of the built-in `object` class respectively[15]. It can be seen from Code Listing 11 and 12 that Python supports descriptors intrinsically and data descriptors take precedence over non-data descriptors.

The semantics of attribute access fully depend on the object type. As an example, given an object `obj`, `obj.x` searches for `x` in `type(obj).__mro__` and `obj.__dict__`.

[15] These two functions are adapted from Invocation from an instance

Listing 11 A pure Python equivalent of `object.__getattribute__`

```
def object_getattribute(obj, name):
    null = object()
    objtype = type(obj)
    cls_var = find_name_in_mro(objtype, name, null)
    descr_get = getattr(type(cls_var), '__get__', null)
    if descr_get is not null:
        if hasattr(type(cls_var), '__set__') or hasattr(type(cls_var), '__delete__'):
            # data descriptor
            return descr_get(cls_var, obj, objtype)

    if hasattr(obj, '__dict__') and name in vars(obj):
        return vars(obj)[name]   # instance variable

    if descr_get is not null:
        # non-data descriptor
        return descr_get(cls_var, obj, objtype)

    if cls_var is not null:
        return cls_var # class variable

    raise AttributeError(name)
```

Python comes with three descriptor-related built-in functions `property`, `classmethod` and `staticmethod`. They are also widely used in Python itself. For instance, `object.__new__` is decorated with `staticmethod`.

2.8 Relevant third-party projects

2.8.1 The project typeshed

Built-in functions like `len` and `hasattr` are written in C. So in general programmers have no type information about such functions. The project typeshed[16] consists of a collection of python stub files for standard libraries, built-ins and some third-party libraries, which provides a way to get type information for external code.

The type hints within `typeshed` can be used for type checking and type inference. Code Listing 13 is a piece of code excerpted from *builtins.pyi*. For instance, if one tries to obtain the type of `hasattr(obj, "name")`, the type should be `bool`.

[16] https://github.com/python/typeshed

Listing 12 A pure Python equivalent of `object.__setattr__`

```
def object_setattr(obj, name, value):
    null = object()
    objtype = type(obj)
    cls_var = find_name_in_mro(objtype, name, null)
    descr_set = getattr(type(cls_var), '__set__', null)
    if descr_set is not null:
        descr_set(cls_var, obj, value)

    if hasattr(obj, '__dict__'):
        vars(obj)[name] = value

    raise AttributeError(name)
```

Listing 13 An excerpt from `builtins.pyi`

```
def hasattr(__obj: object, __name: str) -> bool: ...
def hash(__obj: object) -> int: ...
def id(__obj: object) -> int: ...
def input(__prompt: object = ...) -> str: ...
```

2.8.2 The project isort

The project isort[17] provides functions to automatically recognize the *section* of an imported module. By default a module belongs to one of five sections: **FUTURE**, **STDLIB**, **THIRDPARTY**, **FIRSTPARTY**, **LOCALFOLDER**. In this thesis we only use the first two sections. They are described below.

- **FUTURE**. This section currently only contains one module `__future__`[18]. The module `__future__` is used to enable new features that will be available in the newer Python versions.
- **STDLIB**. It represents The Python Standard Library[19]. The library offers a wide range of modules to deal with general programming tasks.

Code Listing 14 shows how to use `isort` to acquire the section of a module.

2.8.3 The project typeshed-client

The project typeshed-client[20] provides a library for retrieving type information from `typeshed`. It can find the path of the stub file for a particular module,

[17] https://github.com/PyCQA/isort
[18] https://docs.python.org/3.7/library/__future__.html
[19] https://docs.python.org/3.7/library/
[20] https://github.com/JelleZijlstra/typeshed_client

Listing 14 How `isort` recognizes the section for a module

```python
# import isort module
import isort

# the section of __future__ is FUTURE
future_module = isort.place_module("__future__")

# the section of copy is STDLIB
stdlib_module = isort.place_module("copy")
```

collect all names in a stub file and resolve a name to its definition. Code Listing 15 shows how to utilize it.

However, the features provided by `typeshed-client` are not capable of addressing our tasks. How we refactor this project to meet our needs will be explained in the rest of this thesis.

Listing 15 How to extract type information from typeshed by means of typeshed-client

```python
# import typeshed_client module
import typeshed_client

# get a path to a stub file
stub_path = typeshed_client.get_stub_file("copy")

# get all names defined in a stub file
name_dict = typeshed_client.get_stub_names("copy")

# resolve a name to its definition
## get a resolver
resolver = typeshed_client.Resolver()
## the definition corresponding to the name
name_info = resolver.get_fully_qualified_name("copy.copy")
```

3 Related Work

3.1 Points-to analysis

Points-to analysis or *pointer analysis* is a program analysis technique that statically attempts to determine the possible run-time values with respect to each pointer variable [Smaragdakis, Balatsouras, et al., 2015]. It has many applications in compiler optimizations and error detection such as dead code elimination and memory leak detection. Moreover, many analyses like pointer alias analysis and escape analysis are defined on top of points-to analysis.

However, any static analysis that obtains non-trivial program behavior is undecidable [Rice, 1953]. For example, Landi proves that finding the alias that occurs on *all* executions of a program is not recursively enumerable [Landi, 1992]. Later, with the help of parenthesis-Post's Correspondence Problem, the undecidability of context-sensitive inter-procedural analyses has been certified as well [Reps, 2000]. Therefore, one has to safely approximate the behavior of a program in terms of precision and performance. For example, one way to make a good trade-off is to solicit clients' opinions [Hind, 2001].

3.2 Data flow analysis

Data flow analysis is a technique for gathering information of a program without actually executing it [Kam and Ullman, 1977]. It consists of three steps:

1. Construct a *control flow graph* of the program. A control flow graph is the graph-based abstract representation of a program. In the graph, each node attached a label ℓ represents a *basic block* and each edge indicates a *control flow*.

2. Write *data flow equations* for each node in the graph. These equations are used for collecting the desired facts. In general two equations are defined on each node b_ℓ: one equation specifies which information is true at the entry to b_ℓ and the other equation specifies which information is true at the exit of b_ℓ.

3. Solve these equations by repeatedly computing output based on the input at each node until a *fixed point* is reached.

The precision of data flow analysis can be enhanced by employing sensitivities. Three kinds of sensitivities are described in the following subsections.

3.2.1 Flow-sensitive analysis

In a flow-sensitive analysis the order the statements matters [Callahan, 1988]. For instance, a flow-sensitive analysis may determine that `x` in Code Listing 16 may refer to `a_list` after line 3. Instead, a flow-insensitive analysis may just determine that `x` may refer to `a_list`.

Listing 16 An example demonstrating flow-sensitive analysis

```
a_list = list()

x = a_list
```

3.2.2 Path-sensitive analysis

A path-sensitive analysis takes the predicates at conditional branches into account [Bodík and Anik, 1998]. For instance, a path-sensitive analysis may determine that `x` in Code Listing 17 may have an integer value after line 6.

Listing 17 An example demonstrating path-sensitive analysis

```
1 condition = True
2
3 if condition:
4     x = 1
5 else:
6     x = "hello"
```

3.2.3 Context-sensitive analysis

Context sensitivity is a primary approach to enhance the precision of inter-procedural data flow analysis without too much performance degradation. The idea is to encode *context information* to qualify paths taken. Four kinds of context sensitivities are *call site sensitivity*, *object sensitivity*, *type sensitivity* and *hybrid sensitivity*.

$$\delta \in \Delta \quad \texttt{context information}$$

In general two contexts are maintained in a context-sensitive points-to analysis: *calling context*(also referred to as *context*) used to qualify local variables and *heap context* for storing heap abstractions. Heap specialization is crucial to the success of points-to analysis since it is critical to the overall quality of accuracy and scalability [Nystrom, Kim, and Hwu, 2004]. For the sake of simplicity and uniformity, Smaragdakis, Bravenboer, and Lhoták propose two functions **Record** and **Merge** to manipulate contexts in object-sensitive analysis [Smaragdakis, Bravenboer, and Lhoták, 2011]. Their function signatures are:

$$\textbf{Record} : \text{Lab} \times \text{Context} \rightarrow \text{HContext} \tag{1}$$

$$\textbf{Merge} : \text{Lab} \times \text{HContext} \times \text{Context} \rightarrow \text{Context} \tag{2}$$

The function **Record** is used whenever an allocation site is encountered so as to create a new heap context. The function **Merge** is similar to **Record**, but it is invoked at each method invocation site and combines all available information to create a new calling context.

Call site sensitivity

Recording call sites is the first approach employed as context. The analysis encodes a sequence of call sites where each call site belongs to a method. By convention a context of k call sites is maintained, namely, the current call site of the method, the call site of the caller's method, etc., up to a constant value k [Smaragdakis, Balatsouras, et al., 2015].

In [Sharir, Pnueli, et al., 1978], Sharir, Pnueli, et al. consider a tuple of call blocks as call strings in inter-procedural analyses. Similarly, a variant of call strings called call cache is applied in Shivers's dissertation [Shivers, 1991]. Besides, Chapter 2 of [Nielson, Nielson, and Hankin, 2004] also uses call strings of bounded length as context to make inter-procedural analyses more precise.

By means of Equation 1 and 2, a 2-call-site-sensitive analysis with a 1 context-sensitive heap can be simply defined as:

$$\mathbf{Record}(lab, ctx) = first(ctx)$$

$$\mathbf{Merge}(lab, hctx, ctx) = pair(lab, first(ctx))$$

However, the precision of call-site-sensitive analyses is highly dependent on the syntactic patterns of programming languages. It is found that call-site-sensitive analyses for Java are less precise than object-sensitive analyses at the same depth because object-oriented languages usually stress encapsulation and inheritance, that is, indirect invocations, which weakens the usefulness of call sites [Lhoták and Hendren, 2008].

Object sensitivity

Milanova, Rountev, and Ryder propose another flavor of context sensitivity — object sensitivity for Java [Milanova, Rountev, and Ryder, 2002, 2005]. They utilize the receiver object at each method invocation site to distinguish calling contexts and then build up a parameterized k-object-sensitive DEF-USE analysis. However, only object sensitivity of depth 1 is implemented in their papers. The experiment shows that object sensitivity outperforms call-site sensitivity at the same depth because the former could compensate the precision loss of features like encapsulation and inheritance.

By 1 and 2, a 2-object-sensitive analysis with a 1-context-sensitive heap can be expressed as:

$$\mathbf{Record}(lab, ctx) = first(ctx)$$

$$\mathbf{Merge}(lab, hctx, ctx) = pair(lab, first(hctx))$$

Type sensitivity

Type sensitivity is a variant of object sensitivity [Smaragdakis, Bravenboer, and Lhoták, 2011]. The difference between object sensitivity and type sensitivity is that instead of qualifying contexts with allocation sites, type information is encoded. These types represent the classes containing the respective allocation sites with the help of an auxiliary function $\mathcal{T}$: heap $\rightarrow$ ClassName.

By means of types, contexts are coarser since the allocation sites with the same type will be merged, which avoids the replication of information between contexts [Smaragdakis et al., 2015]. According to the experiment result of [Smaragdakis et al., 2011], with insights into choosing appropriate types as contexts, type-sensitive analyses lead to almost no precision loss and are more performant than corresponding object-sensitive ones.

The counterpart of the aforementioned 2-object-sensitive analysis with a 1-context-sensitive heap is:

$$\mathbf{Record}(lab, ctx) = first(ctx)$$

$$\mathbf{Merge}(lab, hctx, ctx) = pair(\mathcal{T}(lab), first(hctx))$$

Hybrid sensitivity

Hybrid sensitivity combines several context sensitivities into one analysis [Kastrinis and Smaragdakis, 2013]. The intuition is to adjust different contexts based on different language features. For instance, static function calls in object-oriented languages may favor call-site sensitivity. Instead, virtual function calls may prefer object sensitivity. Such ideas can greatly extend the design space, which in turn allows more optimization [Smaragdakis, Balatsouras, et al., 2015].

One kind of context combination leads to *uniform hybrid analysis* where both object and call site contexts are maintained. The precision of this combination is at least as accurate as those non-hybrid equivalent ones such as uniform 1-object-sensitive hybrid analysis versus 1-object-sensitive analysis. But the overhead is also evident since two contexts are kept during analysis. Another flavor is *selective hybrid analysis*. In this way, different contexts are formed in line with different language features inside the same analysis.

Kastrinis and Smaragdakis present a more uniform representation of **Record** and **Merge** which takes invocation sites into account [Kastrinis and Smaragdakis, 2013]:

$$\mathbf{Record} : \text{Lab} \times \text{Context} \rightarrow \text{HeapContext}$$

$$\mathbf{Merge} : \text{Lab} \times \text{HeapContext} \times \text{Invocation} \times \text{Context} \rightarrow \text{Context}$$

In the evaluation of hybrid analyses in [Kastrinis and Smaragdakis, 2013], selective hybrid analyses surpass the corresponding base analyses being enhanced in both performance and precision. Besides, though selective ones can be unnoticeable less precise than uniform ones because of possible context information loss, the speed of the former is much higher than that of the latter. In addition, as [Lhoták and Hendren, 2006, 2008] suggest, call-site sensitivity is better to be added as extra context over object sensitivity and [Kastrinis and Smaragdakis, 2013] holds the same opinion that object-sensitive heap is more attractive than call-site-sensitive heap.

3.3 Type inference for dynamic languages

3.3.1 Python

Fritz and Hage present a static analysis to infer type information for Python programs [Fritz and Hage, 2017]. They focus on finding a sweet spot between cost and precision so that the analysis is suitable for interactive tools. The proposed analysis is a data flow analysis and the precision is controlled by three parameters of the widening operator: (1) the maximum number of types a variable may have, (2) the maximum number of attributes that the dictionary of an object may have, (3) the maximum nesting depth of types.

Fromherz, Ouadjaout, and Miné implement an analysis to compute abstract values of variables in a program [Fromherz, Ouadjaout, and Miné, 2018]. The analysis is also able to analyze generators by means of continuation-bases semantics. For this purpose, it maintains a tuple $\mathbf{Gen}(cont, frame, body, vars)$ representing each generator object. The resume location upon the following **next()** is stored in *cont*. Local variables are stored in *vars*. The mapping from local variables to values is stored in *frame* and *body* is the generator function body.

3.3.2 PHP

Van der Hoek and Hage present an object-sensitive type analysis for PHP [Van der Hoek and Hage, 2015]. The algorithm is based on an extended monotone framework which is able to discover call graphs dynamically during fixed point iteration. The analysis variants are parameterized by two context manipulation functions: **Record** and **Merge**. They specify full-object, plain-object and type sensitivities in their experimentation. The result shows full-object sensitive analysis is as least as fast as plain-object sensitive analysis. However, in terms of precision, they both yield basically the same result.

3.3.3 Javascript

Jensen, Møller, and Thiemann propose a static analysis that can infer detailed and sound type information for Javascript programs by means of abstract interpretation [Jensen, Møller, and Thiemann, 2009]. The analysis not only supports the full language defined in the ECMAScript standard but also all built-in functions. The analysis result can be used to detect common programming errors such as confusing numbers with booleans.

The analysis is implemented as an instance of a monotone framework with an elaborate lattice. The precision of the analysis result is further improved by employing *recency abstraction* [Balakrishnan and Reps, 2006]. With recency abstraction, each allocation site ℓ keeps track of two abstractions: singleton abstraction $\ell^{@}$ referring to the most recently allocated object from ℓ and ℓ^{*} referring to a summary abstraction of all older objects related to ℓ.

4 Research questions

When an attribute access is executed, Python under the hood performs a sequence of operations to get, set or delete the expected attribute. One interesting question arises: will attribute access semantics with different precision affect the analysis precision of our type inferencer?

In this section we first describe four special methods used in Python to carry out attribute access (Section 4.1) and then list 3 research questions related to attribute access (Section 4.2).

4.1 Attribute access semantics

Python has by default implemented 4 attribute access methods. These methods are embedded in built-in classes so that programmers are able to use them implicitly.

- `object.__getattribute__`. The method is called when the Python interpreter performs attribute lookup on a class instance.
- `object.__setattr__`. The method is called when the Python interpreter performs attribute storage and deletion on a class instance.
- `type.__getattribute__`. The method is called when the Python interpreter performs attribute lookup on a class object.
- `type.__setattr__`. The method is called when the Python interpreter performs attribute storage and deletion on a class object.

Code Listing 11 is a Python equivalent of `object.__getattribute__`. It states that the retrieved attribute may come from any of (data/non-data) descriptors, instance dictionaries or class dictionaries. For the stake of simplicity in implementation, one type inferencer may merge all three possible results. We define such analysis that merges all possible results as *crude analysis*. On the contrary, *refined analysis* refers to the analysis sticking to the default (path-sensitive) Python attribute access semantics.

4.2 Questions

4.2.1 Research question 1

Is refined analysis (substantially) slower than crude analysis?

4.2.2 Research question 2

Is refined analysis (substantially) more precise than crude analysis?

4.2.3 Research question 3

If the conclusion of research question 2 is true, which features may lead to such consequences?

5 Data Flow Analysis for Python

The type analysis described in this thesis is based on data flow analysis. In this section we first introduce monotone framework, embellished monotone framework [Nielson, Nielson, and Hankin, 2004] and extended monotone framework [Van der Hoek and Hage, 2015]. Then we describe how to adapt extended monotone framework to dynamic monotone framework.

Throughout the thesis we will use $\mathbf{P}_*$ to denote the program to be analyzed, $\mathbf{Lab}_*$ to denote all program labels in $\mathbf{P}_*$.

5.1 Basic definitions

Definition 1 (Partially ordered set[21])**.** A *partially ordered set* $(L, \sqsubseteq)$ is a set L with a partial ordering $\sqsubseteq: L \times L \to \{true, false\}$ that is reflexive $(\forall l : l \sqsubseteq l)$, transitive $(\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3)$, and anti-symmetric $(\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_3)$.

A subset Y of L has $l \in L$ as an *upper bound* if $\forall l' \in Y : l' \sqsubseteq l$ and as a *lower bound* if $\forall l' \in Y : l' \sqsupseteq l$. An upper bound l is called a *least upper bound* of Y if for all upper bounds l', $l \sqsubseteq l'$ and a *greatest lower bound* of Y if for all lower bounds $l' \in Y$, $l' \sqsubseteq l$.

Definition 2 (Complete lattice[22])**.** A *complete lattice* $L = (L, \sqsubseteq) = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a partially ordered set $(L, \sqsubseteq)$ such that all subsets have least upper bounds and greatest lower bounds.

Definition 3 (Ascending chain condition[23])**.** A partially ordered set P is said to satisfy the *ascending chain condition* if for every increasing sequence $l_1 \leq l_2 \leq l_3 \leq ...$ with $l_i \in P$, there is $n \in \mathbb{N}$ such that $l_n = l_{n+1} = ...$

5.2 Monotone framework

A *Monotone framework* allows a general pattern for data flow analysis by abstracting the commonalities and parameterizing the differences of different analyses.

An *instance*[24] of a monotone framework $(L, \mathcal{F}, F, E, \iota, f_\ell.)$ consists of:

- a complete lattice L that satisfies the ascending chain condition.
- a set of monotone functions $\mathcal{F}$ from L to L that includes the identity function and is closed under function compositions.
- a finite set of flow F.
- a finite set of *extremal labels* E.

[21] adapted from Appendix A of [Nielson, Nielson, and Hankin, 2004]
[22] adapted from Appendix A of [Nielson, Nielson, and Hankin, 2004]
[23] adapted from Appendix A of [Nielson, Nielson, and Hankin, 2004]
[24] borrowed from Section 2.3 of [Nielson, Nielson, and Hankin, 2004]

- an *extremal value* $\iota \in L$ for the extremal labels.
- a mapping f_ℓ. from labels in $\mathbf{Lab}_*$ to transfer functions in $\mathcal{F}$.

The instance gives rise to a *set of equations*[25] of the form:

$$\begin{aligned} A_\circ(\ell) =& \bigsqcup\{A_\bullet(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota_E^\ell \\ & \texttt{where } \iota_E^\ell = \begin{cases} \iota & \texttt{if } \ell \in E \\ \bot & \texttt{if } \ell \notin E \end{cases} \\ A_\bullet(\ell) =& f_\ell(A_\circ(\ell)) \end{aligned}$$

5.3 The worklist algorithm

The *worklist algorithm*[26] listed in Algorithm 1 is a general iterative algorithm to compute the least solution to the data flow equations for monotone frameworks.

The algorithm maintains a worklist W which contains a list of pairs obtained from F. The presence of each pair (ℓ, ℓ') indicates the analysis information has changed at the exit of the block labeled with ℓ. Therefore, the analysis information attached on the entry of the block labeled with ℓ' has to be recomputed to see if the information should be propagated.

5.4 Widening

The worklist algorithm always terminates if the analysis lattice satisfies the ascending chain condition. If this is not the case, widening operators can be employed to ensure termination. With a widening operator ∇[27], the construction of an iterative sequence is changed from $l_1, l_1 \sqcup l_2, (l_1 \sqcup l_2) \sqcup l_3$ to $l_1, l_1 \nabla l_2, (l_1 \nabla l_2) \nabla l_3$. The latter one guarantees stabilization.

When the worklist algorithm employs a widening operator, the analysis result is not necessarily the least fixed point. The precision of the approximated fixed point and the cost of computing both depend on the choice of the widening operator.

5.5 Interprocedural data flow analysis

Almost all modern programming languages support functions in some form. In a control flow graph, they are represented as follows: for each function definition f, there are two nodes ℓ_n denoting the entry to the body and ℓ_x the exit from the body. Each call to f also has two labels: ℓ_c marking the call and ℓ_r the return.

In the rest of this thesis we will employ and adapt the way of how [Fritz and Hage, 2017] represents control flow graphs. Figure 8 shows a function `func`

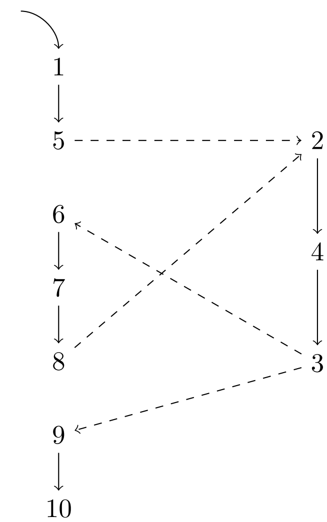
[25] borrowed from Section 2.3 of [Nielson, Nielson, and Hankin, 2004]
[26] borrowed from Section 2.4 of [Nielson, Nielson, and Hankin, 2004]
[27] see Section 4.2.1 of [Nielson, Nielson, and Hankin, 2004]

with the entry label 2 and the exit label 3. In addition, each function call site of `func` has two labels such as 8 and 9 corresponding to the call and exit labels respectively.

$[\texttt{def}]^1\ [\texttt{func(x)}]^2_3\texttt{:}$
$\quad [\texttt{return x}]^4$

$[\texttt{result1 = }[\texttt{func(1)}]^5_6]^7$
$[\texttt{result2 = }[\texttt{func("str")}]^8_9]^{10}$

(a) A program



(b) The control flow graph

Figure 8: A program and its control flow graph

However, naively applying intra-procedural techniques described in Section 5.2 may harm the analysis precision. Expressed in term of Figure 8b, nothing prevents the analysis from pushing information from ℓ_3 to ℓ_9. Thus the analysis may infer that `result2` may have types `str` and `int`. But in fact `result2` only has one type `str`.

5.6 Embellished monotone framework

By taking context Δ into account, a monotone framework turns into an *embellished monotone framework*. The following tuple represents an *instance*[28] of an embellished monotone framework:

$$(\widehat{L}, \widehat{\mathcal{F}}, F, E, \widehat{\iota}, \widehat{f_\ell}.)$$

- a complete lattice $\widehat{L} = \Delta \to L$ that satisfies the ascending chain condition.
- a set of monotone functions $\widehat{\mathcal{F}}$ from $\Delta \to L$ to $\Delta \to L$ that includes the identity function and is closed under function compositions.
- F and E remain the same.
- an extremal value $\widehat{\iota} = \Delta \to \iota$ in $\widehat{L}$ for the extremal labels.

[28] borrowed from Section 2.5 of [Nielson, Nielson, and Hankin, 2004]

- a mapping $\widehat{f_{\ell}.}$ from contexts in Δ and labels in $\mathbf{Lab}_*$ to transfer functions in $\widehat{\mathcal{F}}$.

5.7 Dynamic monontone framework

Based on an embellished monotone framework, Van der Hoek and Hage propose an *extended monotone framework* for PHP which supports dynamic flow adding [Van der Hoek and Hage, 2015]. Our work is similar to Van der Hoek and Hage's since Python and PHP both are dynamic languages and functions in them are first-class.

Our dynamic monotone framework[29] for Python supports adding and collecting inter-procedural control flow edges on the fly. An *instance* of a dynamic monotone framework consists of a tuple described below. In our setting, labels in F or E in a dynamic monotone framework are replaced with program points $\mathbf{PP}_*$.

$$program\_point \in \mathbf{PP}_* = \{(\ell, \delta) \mid \ell \in \mathbf{Lab}_*, \delta \in \Delta\}$$

$$(L, \mathcal{F}, F, E, \iota, f_\cdot, \Psi, \Phi)$$

- a complete lattice $L = \lambda\ell \to \lambda\delta \to L_{\ell,\delta}$ that satisfies the ascending chain condition.
- a set of monotone functions $\mathcal{F} = \lambda\ell \to \lambda\delta \to L_{\ell,\delta} \to L_{\ell,\delta}$ that includes the identity function and is closed under function compositions.
- a finite set of flow $F = \lambda\ell \to \lambda\delta \to F_{\ell,\delta}$.
- a finite set of extremal program points $E = \lambda\ell \to \lambda\delta \to \mathbf{PP}_*$.
- an extremal value $\iota = \lambda\ell \to \lambda\delta \to \iota_{\ell,\delta}$ in L for the extremal program points.
- a mapping $f_\cdot = \lambda\ell \to \lambda\delta \to f_{\ell,\delta}$ from labels in $\mathbf{Lab}_*$ and context elements in Δ to transfer functions in $\mathcal{F}$.
- a mapping $\Psi = \lambda\ell \to \lambda\delta \to \Psi_{\ell,\delta}$ from labels in $\mathbf{Lab}_*$ and context elements in Δ to dynamic inter-procedural flow creating functions in $\Psi_{\ell,\delta}$.
- a mapping $\Phi = \lambda\ell \to \lambda\delta \to \Phi_{\ell,\delta}$ from labels in $\mathbf{Lab}_*$ and context elements in Δ to flow collecting functions in $\Phi_{\ell,\delta}$.

Then the *data flow equations*[30] become:

$$A_\circ(\ell,\delta) = \bigsqcup\{A_\bullet(\ell',\delta') \mid ((\ell',\delta'),(\ell,\delta)) \in F\} \sqcup \iota_E^{\ell,\delta}$$
$$\texttt{where } \iota_E^{\ell,\delta} = \begin{cases} \iota & \texttt{if } \ell \in E \wedge \delta = \Lambda \\ \bot & \texttt{if } \ell \notin E \end{cases} \tag{3}$$

[29] adapted from Section 4.5 of [Van der Hoek, 2014]
[30] adapted from Section 4.6 of [Van der Hoek, 2014]

Equation 3 specifies how information flows from exits from program points to an entry to a program point.

$$A_\bullet(\ell, \delta) = f_{\ell,\delta}(A_\circ(\ell, \delta)) \tag{4}$$

Equation 4 specifies how information flows from the entry to the exit of a node except program points of function return.

$$\begin{aligned} A_\bullet(\ell_r, \delta_r) =& f_{(\ell_c,\delta_c),(\ell_r,\delta_r)}(A_\circ(\ell_c, \delta_c), A_\circ(\ell_r, \delta_r)) \\ & \forall\ (\ell_r, \delta_r) \in \mathit{ReturnProgramPoints} \end{aligned} \tag{5}$$

Equation 5 specifies given a return program point, how information flows from the entry to the exit of the node. The signature of the transfer function is $f_{(\ell_c,\delta_c),(\ell_r,\delta_r)} = \mathrm{L} \rightarrow \mathrm{L} \rightarrow \mathrm{L}$. The first parameter L represents the data flow information at the entry of a call and the second L represents the data flow information at the exit of the callee. The resulted lattice element depends on the semantics of a language.

$$\mathit{IF} = \Psi_{\ell,\delta}(\ell, \delta) \cup \mathit{IF},\ \forall\ (\ell, \delta) \in F \tag{6}$$

Equation 6 specifies how $\Psi_{\ell,\delta} : \lambda\ell \rightarrow \lambda\delta \rightarrow \mathrm{IF}$ adds inter-procedural flow to IF.

$$F = \{\Phi_{\ell,\delta}(e, \Lambda) \mid e \in E\} \cup \{\Psi_{\ell,\delta}(\ell', \delta') \mid ((\ell, \delta), (\ell', \delta')) \in F\} \tag{7}$$

Equation 7 specifies how all program flow is generated. It at first computes all initial edges related to extremal program points and then expands the flow by $\Phi_{\ell,\delta} : \lambda\ell \rightarrow \lambda\delta \rightarrow \mathrm{F}$ until a fixed point is reached.

Equations 3 to 7 are mutually dependent. Since the program flow F depends on the inter-procedural flow IF, the inter-procedural flow IF depends on the effect value $A_\bullet$, the effect value $A_\bullet$ depends on the context value $A_\circ$ and the context value $A_\bullet$ depends on the program flow F.

5.8 The worklist algorithm for dynamic monotone framework

The *worklist algorithm*[31] for dynamic monotone frameworks is listed in Algorithm 2. Given an instance of a dynamic monotone framework, it computes the least fixed point.

[31]adapted from Algorithm 2 of [Van der Hoek, 2014]

Algorithm 1 The worklist algorithm

Input: An instance of a monotone framework: $(L, \mathcal{F}, F, E, \iota, f_\ell.)$
Output: $MFP_\circ$, $MFP_\bullet$
Method:

Step 1: Initialization
$W \leftarrow nil$
for all (ℓ, ℓ') in F **do**
    $W \leftarrow cons((\ell, \ell'), W)$
end for
for all ℓ in F or E **do**
    if $\ell \in E$ **then**
        $A[\ell] \leftarrow \iota$
    else
        $A[\ell] \leftarrow \bot_L$
    end if
end for

Step 2: Iteration
while $W \neq nil$ **do**
    $\ell,\ \ell' \leftarrow fst(head(W)),\ snd(head(W))$
    $W \leftarrow tail(W)$
    if $f_\ell(A[\ell]) \not\sqsubseteq A[\ell']$ **then**
        $A[\ell'] \leftarrow A[\ell'] \sqcup f_\ell(A[\ell])$
        for all ℓ'' with (ℓ', ℓ'') in F **do**
            $W \leftarrow cons((\ell', \ell''), W)$
        end for
    end if
end while

Step 3: Presenting
for all ℓ in F or E **do**
    $MFP_\circ \leftarrow A[\ell]$
    $MFP_\bullet \leftarrow f_\ell(A[\ell])$
end for

Algorithm 2 The worklist algorithm for dynamic monotone frameworks

Input: An instance of a dynamic monotone framework: $(L, \mathcal{F}, F, E, \iota, f_{\cdot}, \Psi, \Phi)$
Output: $\mathit{MFP}_\circ$, $\mathit{MFP}_\bullet$
Method:

Step 1: Initialization

$W \leftarrow \mathit{nil}$
$\mathit{IF} \leftarrow \emptyset$
for all ℓ in E **do**
    $A[\ell,\ \Lambda] \leftarrow \iota$
    for all $((\ell,\delta),\ (\ell',\delta'))$ in $\Phi_{\ell,\delta}(\ell,\ \Lambda)$ **do**
        $W \leftarrow \mathit{cons}(((\ell,\delta),\ (\ell',\delta')),\ W)$
    end for
end for

Step 2: Iteration

while $W \neq \mathit{nil}$ **do**
    $(\ell,\delta),\ (\ell',\delta') \leftarrow \mathit{fst}(\mathit{head}(W)),\ \mathit{snd}(\mathit{head}(W))$
    $W \leftarrow \mathit{tail}(W)$
    if $(\ell,\delta) \in \mathit{ReturnProgramPoints}$ **then**
        $\ell_c \leftarrow \mathit{IF}(\ell_r)$
        $\mathit{Effect} \leftarrow f_{(\ell_c,\delta_c),(\ell_r,\delta_r)}(A_\circ(\ell_c,\delta_c),\ A_\circ(\ell_r,\delta_r))$
    else
        $\mathit{Effect} \leftarrow f_{\ell,\delta}(A_\circ(\ell,\delta))$
    end if

    if $\mathit{Effect} \not\sqsubseteq A[\ell',\delta']$ **then**
        $A[\ell',\delta'] \leftarrow A[\ell',\delta'] \sqcup \mathit{Effect}$
        $\mathit{IF} \leftarrow \Psi_{\ell,\delta}(\ell',\delta') \cup \mathit{IF}$
        for all $((\ell',\delta'),\ (\ell'',\delta''))$ in $\Phi_{\ell,\delta}(\ell',\delta')$ **do**
            $W \leftarrow \mathit{cons}(((\ell',\delta'),(\ell'',\delta'')),\ W)$
        end for
    end if
end while

Step 3: Presenting

for all (ℓ,δ) in F or E **do**
    $\mathit{MFP}_\circ(\ell) \leftarrow A[\ell,\delta]$
    if $(\ell,\delta) \in \mathit{ReturnProgramPoints}$ **then**
        $\ell_c \leftarrow \mathit{IF}(\ell_r)$
        $\mathit{MFP}_\bullet(\ell,\delta) \leftarrow f_{(\ell_c,\delta_c),(\ell_r,\delta_r)}(A_\circ(\ell_c,\delta_c),\ A_\circ(\ell_r,\delta_r))$
    else
        $\mathit{MFP}_\bullet(\ell,\delta) \leftarrow f_{\ell,\delta}(A_\circ(\ell,\delta))$
    end if
end for

6 Control flow graphs for Python

A control flow graph represents all possible execution paths in a program. In this section we first describe how to simplify Python by desugaring it into its core language. Then we exhibit the supported core language constructs and their control flow graphs. At last, some core constructs are further destructed to cater for the Python data model.

6.1 Desugaring of Language constructs

Syntactic sugar is high-level constructs that make code easier to read. Static analyzers often distill languages with those sugared constructs into their core languages before analyzing. By reducing a programming language to its essence, one analysis tool may have simpler implementation and thus is able to focus on crucial details.

This section describes how our control flow graph generator handles various high-level constructs. In the following discussion, we use special names such as `_tmpvar1`, `_tmpvar2`, ... to denote temporary variables. The desugaring order is in accordance with the evaluation order[32] of Python. The evaluation order is determined by the Python abstract syntax tree.

6.1.1 The assignment statement

The assignmen statement[33] is used to (re)bind names to values. Figure 9 shows how to transform a compound assignment statement into a sequence of simple statements.

```python
def func(x):
    return x

a = b = c = func(1)
```

(a) Original form

```python
def func(x):
    return x

a = func(1)
b = func(1)
c = func(1)
```

(b) Desugared form

Figure 9: How to desugar a compound assignment statement

A del statement[34] can have more than one expression such as `del a, b, c`. If so we desugar the del statement to make sure each del statement only has one expression.

[32] https://docs.python.org/3/reference/expressions.html#evaluation-order
[33] https://docs.python.org/3.7/reference/simple_stmts.html#assignment-statements
[34] https://docs.python.org/3.7/reference/simple_stmts.html#the-del-statement

6.1.2 The augassign statement

The augassign statement[35] combines an arithmetic operator with an assignment operator, which eliminates the need to define a temporary variable. For instance, `a = a + 1` can be shortened as `a += 1`. We transform all augassign statements into plain assignment statements. Figure 10 shows our approach of desugaring an augassign statement.

```
size += 5
```

(a) Original form

```
size = size + 5
```

(b) Desugared form

Figure 10: How to desugar an augassign statement

6.1.3 The annassign statement

The annassign statement[36] allows attaching type annotations to normal variables. An annassign statement has an optional right-hand-side expression. If the expression is not present, the whole statement is ignored since it has no effect in our analysis. Figure 11 shows two kinds of annassign statements.

```
name: str

address: str = "Utrecht"
```

(a) Original form

```
# name: str is ignored

address = "Utrecht"
```

(b) Desugared form

Figure 11: How to desugar two annassign statements

6.1.4 The expression statement

The expressio statement[37] is a sole expression without any target. We shall add a temporary variable to act as the target so that the analysis can handle expression statements and assignment statements uniformly. Figure 12 shows how to address an expression statement.

6.1.5 The assert statement

The assert statement[38] allows programmers to test if certain assumptions remain `True` while developing. Figure 13 shows the desugaring of an assert

[35] https://docs.python.org/3.7/reference/simple_stmts.html#augmented-assignment-statements

[36] https://docs.python.org/3.7/reference/simple_stmts.html#annotated-assignment-statements

[37] https://docs.python.org/3.7/reference/simple_stmts.html#expression-statements

[38] https://docs.python.org/3.7/reference/simple_stmts.html#the-assert-statement

```
a_func_call()
```

(a) Original form

```
_tmpvar1 = a_func_call()
```

(b) Desugared form

Figure 12: How to desugar an expression statement

statement.

```
assert number > 0
```

(a) Original form

```
if not number > 0:
    raise AssertionError
```

(b) Desugared form

Figure 13: How to desugar an assert statement

6.1.6 The import statement

The import statement[39] is used to find and load modules. Figure 14 shows how to desugar an import statement.

```
import mod1, mod2.mod3 as mod4
```

(a) Original form

```
import mod1
pass

import mod2.mod3 as mod4
pass
```

(b) Desugared form

Figure 14: How to desugar an import statement

6.1.7 The with statement

The with statement[40] is used to wrap the execution of a block with the help of a context manager. It simplifies the management of resources such as file resources. Figure 15 shows how to desugar a with statement.

Multiple items in a with statement is also possible. Figure 16 shows how to transform a complex with statement into its simpler form.

[39] https://docs.python.org/3.7/reference/simple_stmts.html#the-import-statement
[40] https://docs.python.org/3.7/reference/compound_stmts.html#the-with-statement

```python
with EXPRESSION as TARGET:
    BODY
```

(a) Original form

```python
manager = EXPRESSION
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    BODY
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

(b) Desugared form

Figure 15: How to desugar a with statement

```python
with A() as a, B() as b:
    BODY
```

(a) Original form

```python
with A() as a:
    with B() as b:
        BODY
```

(b) Desugared form

Figure 16: How to desugar a complex with statement

6.1.8 The for statement

The for statement[41] is used for iterating over iterable objects. Iterable objects are those objects that implement `__iter__` special method. Figure 17 shows the standard desugaring of a for statement. The evaluation process is as follows: at first `b` is evaluated into an iterator object. Then `c` is executed once for each target `a` until the iterator raises a `StopIteration` exception.

We shall transform all for statements into while statements in order to simplify the analysis. Figure 18 shows the desugaring in our analysis.

[41] https://docs.python.org/3.7/reference/compound_stmts.html#the-for-statement

```python
for a in b:
    c
```

(a) Original form

```python
_iter = iter(b)
while True:
    try:
        a = next(_iter)
    except StopIteration:
        break
    else:
        c
del _iter
```

(b) Desugared form

Figure 17: How to desugar a for statement in the standard way

```python
a_list = [1,"hello", True]
another_list = []
for elt in a_list:
    another_list.append(elt)
```

(a) Original form

```python
a_list = [1,"hello", True]
another_list = []

a_list_iter = iter(a_list)
while a_list_iter:
    elt = next(a_list_iter)
    another_list.append(elt)
```

(b) Desugared form

Figure 18: How to desugar a for statement in our analysis

6.1.9 The lambda expression

The lambda expression[42] is a small anonymous function with only one expression. Figure 19 shows how to transform a lambda expression into an ordinary function.

```python
lambda x, y = x + y
```

(a) Original form

```python
def _tmpvar1(x, y):
    return x + y
```

(b) Desugared form

Figure 19: How to desugar a lambda expression

[42] https://docs.python.org/3.7/reference/expressions.html#lambda

6.1.10 Decorators

Decorators are functions which modify the functionality of other functions or classes without modifying their structures. Figure 20 shows how to desugar a function with two decorators.

```
@f1
@f2
def func(x):
    return x
```

(a) Original form

```
def func():
    return x

func = f2(func)
func = f1(func)
```

(b) Desugared form

Figure 20: How to desugar a function with two decorators

We deal with the decorator `property` in a different manner. It is desugared in two steps. In the first step, our control flow generator collects all possible getters, setters and deleters. In the second step, an assignment statement is created with these getters, setters and deleters. One example can be seen in Figure 21.

```
@property
def name():
    ...

@name.setter
def name():
    ...

@name.deleter
def name():
    ...
```

(a) Original form

```
def name():
    ...

def name1():
    ...

def name2():
    ...

name = property(name, name1,
↪  name2)
```

(b) Desugared form

Figure 21: How to desugar `property` decorators

6.1.11 Expressions

Expressions are representations of values. The difference between expressions and statements is that a statement does something but an expression always yields a value. Expressions can also be chained. Such chained expressions may complicate the implementations of static analyzers. Therefore we deconstruct complex expressions into simpler ones. Code Listing 18 and 19 show how to desugar a complex expression.

Listing 18 Original form

```
length = to_integer('3').bit_length()
```

Listing 19 Desugared form of Code Listing 18

```
_tmpvar1 = '3'
_tmpvar2 = to_integer(_tmpvar1)
_tmpvar3 = _tmpvar2.bit_length
_tmpvar4 = _tmpvar3()
length = _tmpvar4
```

The expressions which have the form `x and y` or `x or y` are further desugared into if statements. The reason is these two kinds of expressions return the value of `x` or `y` rather than a `bool` object as the final evaluated value.

Similarly, each conditional expression `test if expression1 else expression2` is also transformed into a plain if statement.

6.1.12 Comprehensions

List comprehensions, *set comprehensions*, *dict comprehensions* and *generator comprehensions* provide a functional way to create lists, sets, dictionaries and generators respectively. They are more efficient and readable than equivalent loops. In our analysis, we shall translate them into verbose forms. Since they have basically the same structure, only desugaring a list comprehension is shown (refer to Figure 22).

```
even_nums = [
    x for x in range(20)
        if x % 2 == 0
]
```

(a) Original form

```
_tmpvar1 = list()
for x in range(20):
    if x % 2 == 0:
        _tmpvar1.append(x)
even_nums = _tmpvar1
```

(b) Desugared form

Figure 22: How to desugar a list comprehension

6.1.13 Literal collections

Python provides four kinds of literal collections: *list literals*, *tuple literals*, *set literals* and *dict literals*. Since they are very similar, we only explain list literals here. A list literal can store heterogeneous items. The values in a list literal are separated by a comma and enclosed within square brackets. Figure 23 shows how to desugar a list literal.

```
hybrid = [
    "Netherlands",
    1,
    True,
    1.0
]
```

(a) Original form

```
_tmpvar1 = list()
_tmpvar1.append("Netherlands")
_tmpvar1.append(1)
_tmpvar1.append(True)
_tmpvar1.append(1.0)
hybrid = _tmpvar1
```

(b) Desugared form

Figure 23: How to desugar a list literal

6.2 Supported language constructs

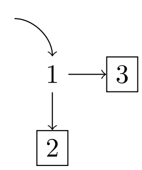
After applying the desugaring described in Section 6.1, we have only a set of core language constructs. In this section we describe our approach to representing each core construct in a control flow graph respectively.

6.2.1 The if statement

The if statement[43] is used for decision making. It selects exactly one body of code whose test expression is evaluated to `True`. Figure 24 shows a simple if statement and its control flow graph.

```
if [test_expression]^1:
    [a = 1]^2
else:
    [a = True]^3
```

(a) The code



(b) The control flow graph

Figure 24: A simple if statement and its control flow graph

An if statement may have zero or more `elif` clauses, which allows programmers to check multiple conditional expressions. The `elif` is a short notation for *else if*. Figure 25 shows an if statement with one `elif` clause and its control flow graph.

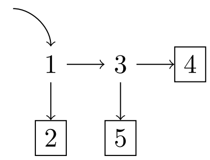
6.2.2 The while statement

The while statement[44] is used to iterate over a block of code as long as the test expression is evaluated to `True`. In addition, a `while` loop may have

[43] https://docs.python.org/3.7/reference/compound_stmts.html#the-if-statement
[44] https://docs.python.org/3.7/reference/compound_stmts.html#the-while-statement

```
if [test_expression1]^1:
    [a = 1]^2
elif [test_expression2]^3:
    [a = 2]^4
else:
    [a = 3]^5
```

(a) The code



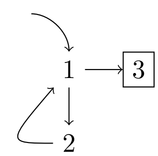
(b) The control flow graph

Figure 25: A complex if statement and its control flow graph

an optional `else` block. In general the `else` block will be executed if the test expression is evaluated to `False`. Figure 26 shows a `while` loop with an `else` part.

```
while [test_expression]^1:
    [a = 1]^2
else:
    [a = 2]^3
```

(a) The code



(b) The control flow graph

Figure 26: A while statement and its control flow graph

The break statement[45] and the continue statement[46] are used to alter the control flow of a loop. The break statement terminates the current loop. The continue statement skips the rest of the code of the loop body in the current iteration.

In a `while` loop, the `else` block will be ignored if the loop is terminated by a `break`. Figure 27 demonstrates the effects of `break` and `continue`.

6.2.3 The try statement

The try statement[47] allows programmers to take actions in case an error occurs. It is intrinsically hard to deal with in data flow analysis since exceptions break out of the normal control flow. To represent exception handling in the control flow graph, the analysis has to first identify program points where exceptions could occur, then locate the corresponding `catch` clauses and add edges from the former to the latter.

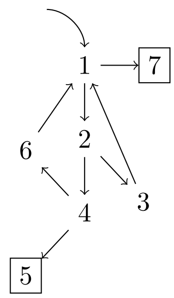
[45] https://docs.python.org/3.7/reference/simple_stmts.html#the-break-statement

[46] https://docs.python.org/3.7/reference/simple_stmts.html#the-continue-statement

[47] https://docs.python.org/3.7/reference/simple_stmts.html#the-continue-statement

```
while [test1]^1:
    if [test2]^2:
        [continue]^3
    elif [test3]^4:
        [break]^5
    [y = True]^6
else:
    [a = True]^7
```

(a) The code



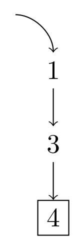
(b) The control flow graph

Figure 27: A complex while statement and its control flow graph

However, analyzing exceptions would complicate the analysis a lot. In our setting, we assume that exceptions happen in rare cases so `catch` clauses are ignored by the analysis. Figure 28 shows how to deal with a try statement.

```
try:
    [result = x // y]^1
except ZeroDivisionError:
    [result = -1]^2
else:
    [y = 0]^3
finally:
    [x = 0]^4
```

(a) The code



(b) The control flow graph

Figure 28: A try statement without considering exception handling

In addition, since the raise statement[48] may appear outside a `try` block. We observe that the pass statement[49] is just a null statement. So each `raise` is simply transformed into a `pass`. Figure 29 shows such a transformation. The exception handling part of each assert statement is handled similarly.

```
if x != 0:
    raise ValueError
```

(a) Original form

```
if x != 0:
    pass
```

(b) Transformed form

Figure 29: How to transform a raise statement

[48] https://docs.python.org/3.7/reference/simple_stmts.html#the-raise-statement
[49] https://docs.python.org/3.7/reference/simple_stmts.html#the-pass-statement

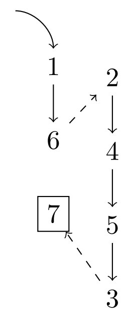
6.2.4 The global and nonlocal statements

The global statement[50] is used to tell Python to relate the listed identifiers to bound variables in the global scope. The nonlocal statement[51] causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope. Figure 30 shows a control flow graph containing a `global` and a `nonlocal`.

$[\texttt{def}]^1\ [\texttt{func()}]^2_3$:
$\quad[\texttt{global a}]^4$
$\quad[\texttt{nonlocal b}]^5$

$[\texttt{func()}]^6_7$

(a) The Code



(b) The control flow graph

Figure 30: A program containing a global and a nonlocal and its control flow graph

6.2.5 The classdef statement

When the Python interpreter encounters a class definition, its class body is executed in a new execution frame. When the class body finishes execution, the execution frame is discarded but the local namespace is saved. In data flow analysis, we shall set up a call label ℓ_c and a return label ℓ_r for each class definition and an entry label ℓ_n and an exit label ℓ_x for its class body. Figure 31 shows a class definition.

6.2.6 The functiondef statement

The control flow graph for a function definition is similar to that for a class definition except that the function body will only be executed when called. Figure 32 shows a control flow graph for a function definition. The flow $2 \rightarrow 4 \rightarrow 5 \rightarrow 3$ corresponds to the function body of `func`.

6.2.7 The return statement

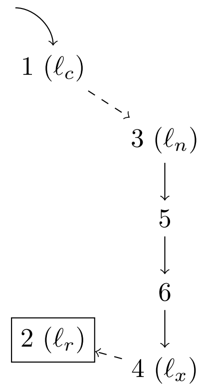
The return statement[52] is used to end the current execution of a function and return the result to its caller. Accordingly, in Figure 33, there is an edge $5 \rightarrow 3$ denoting that the execution has ended after a `return`.

[50] https://docs.python.org/3.7/reference/simple_stmts.html#the-global-statement
[51] https://docs.python.org/3.7/reference/simple_stmts.html#the-nonlocal-statement
[52] https://docs.python.org/3.7/reference/simple_stmts.html#the-return-statement

$[\texttt{class}]^1_2$ $[\texttt{Test}]^3_4$:
    $[\texttt{class\_variable = 1}]^5$

    $[\texttt{def}]^6$ $[\texttt{\_\_init\_\_(self)}]^7_8$:
        $[\texttt{pass}]^9$

(a) The code



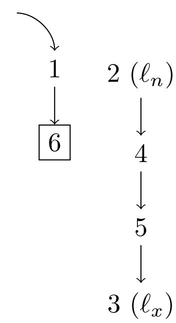
(b) The control flow graph

Figure 31: A class definition and its control flow graph

$[\texttt{def}]^1$ $[\texttt{func()}]^2_3$:
    $[\texttt{x = 1}]^4$
    $[\texttt{y = 2}]^5$

$[\texttt{pass}]^6$

(a) The code



(b) The control flow graph

Figure 32: A function definition and its control flow graph

6.2.8 The import statement

The import statement[53] is used to load modules and define names within an import statement. Figure 34 shows an import statement and its control flow graph.

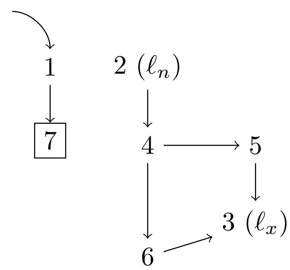
6.3 Special language constructs

Everything in Python is an object. The operations that an object supports are determined by its class. For instance, to evaluate the expression `x < y`, the Python interpreter first looks up the special method `__le__` on `type(x)`. If found, the interpreter will execute `x.__le__(y)` to get the comparison result. If not, `TypeError` will be raised. Code Listing 20 shows how the Python inter-

[53] https://docs.python.org/3.7/reference/simple_stmts.html#the-import-statement

```
[def]1 [func()]2 3:
    if [test_expression]4:
        [return 1]5
    [a = True]6


[pass]7
```

(a) The code

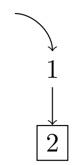


(b) The control flow graph

Figure 33: A program containing a return statement and its control flow graph

```
[from mod import var]1

[pass]2
```

(a) The code



(b) The control flow graph

Figure 34: An import statement and its control flow graph

preter retrieves and calls the special method `__le__` for `x < y` under the hood.

Since a special method could be invoked when an expression is being evaluated, we shall reflect this fact in our control flow graph. Our approach is to set up two nodes for each expression evaluation. In this section we describe how to create control flow graphs for these special constructs.

6.3.1 An expression on the right-hand side of an assignment

Evaluating the expression on the right-hand side of an assignment may result in a method call. Figure 35 shows the code block of a desugared assignment statement. Figure 36 shows two kinds of control flow graphs for code in Figure 35b. The idea is when the subscript expression `a_container[i]` is encountered, the analysis will try to retrieve the special method `__getitem__`. If found, the inter-procedural flow edges will be created such as 1 → `__getitem__` and `__getitem__` → 2.

6.3.2 An expression on the left-hand side of an assignment

Evaluating the target of an assignment could also lead to a special method invocation. For instance, performing `a[i] = b` implicitly invokes `__setitem__`. To reflect this, we also extend our control flow graph. Figure 37 shows an example for that. Figure 38 shows how to deal with the desugared expression in Figure 37.

Listing 20 How a special method is called implicitly

```
# get the class of the object x
x_class = type(x)

if hasattr(x_class, "__le__"):
    # if x_class defines the special method __le__
    # obtain __le__
    le_method = getattr(x_class, "__le__")
    # perform the special method with two arguments x and y
    return le_method(x, y)
else:
    # if not found, x does not support < operation
    raise TypeError
```

```
[result = a_container[i]]1
```

(a) Original form

```
# call node of a __getitem__
[a_container[i]]1

# return node of a __getitem__
[_tmpvar1]2

[result = _tmpvar1]3
```

(b) Desugared form

Figure 35: how to desugar a subscript expression on the right-hand side of an assignment statement

6.3.3 An expression in a del statement

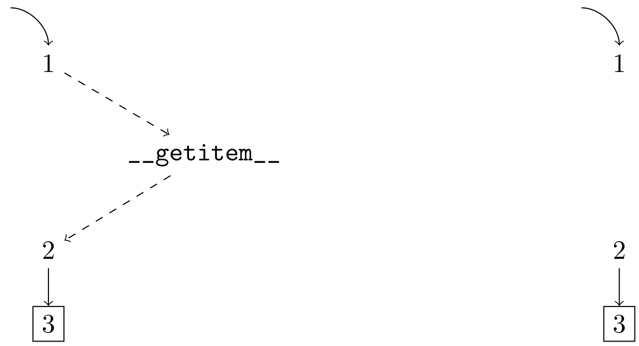
Executing a del statement may be accompanied by a special method invocation as well. Figure 39 shows how to desugar a del statement. Figure 40 shows how to deal with the desugared del statement in Figure 39.

6.3.4 The call expression

Python does not distinguish a class instantiation call from a function call. They both have the form of `A_Name(*args, **kwargs)`. In the following section we describe how to create control flow graphs for these two kinds of calls respectively.

A function call

Figure 41 and 42 show how to desugar a function call and its two possible control flow graphs.



(a) A __getitem__ is found (b) A __getitem__ is not found

Figure 36: Two control flow graphs of a subscript expression

```
[a[i] = v]^1
```

(a) Original form

```
# call node of a __setitem__
[a[i]]^1

# return node of a __setitem__
[_tmpvar1]^2

[result = _tmpvar1]^3
```

(b) Desugared form

Figure 37: how to desugar a subscript expression on the left-hand side of an assignment statement

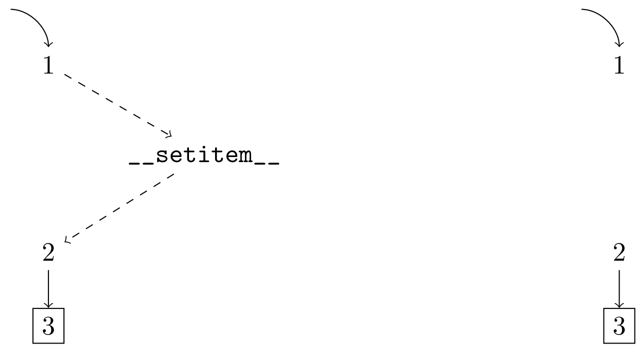
A class initialization call

A class initialization call undergoes two method calls. The first is `__new__` to reserve a memory region for the new class instance. The second is `__init__` which initializes the instance attributes. Figure 43 and 44 show how to desugar a class initialization call and its two possible control flow graphs.

It is possible that `__new__` or `__init__` is missing . In this situation we use artificial types explained in Section 7.2.3 to achieve the goal.

The uniform representation of a call

The control flow graph for a call can be represented as Figure 45. Edges are added dynamically during analyzing. However, functions such as built-in function `len()` have no function bodies. We address this issue by setting up dummy nodes in the control flow graph. The dummy nodes are used to receive the return value of a function call without the function body. Therefore we further extend Figure 45 to Figure 46.



(a) A __setitem__ is found (b) A __setitem__ is not found

Figure 38: Two control flow graphs of a subscript expression

```
[del a[i]]1
```

(a) Original form

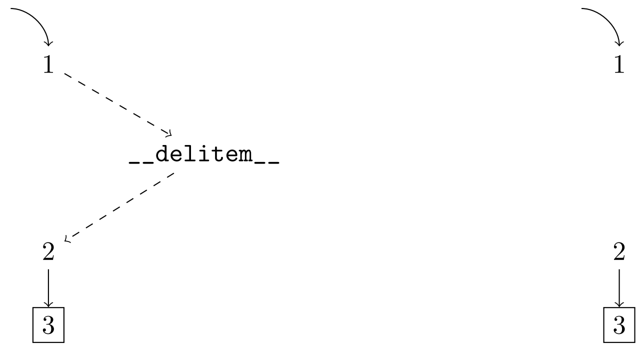
```
# call node of a __delitem__
[a[i]]1

# return node of a __delitem__
[_tmpvar1]2

[result = _tmpvar1]3
```

(b) Desugared form

Figure 39: How to desugar a del statement

6.4 Elimination of temporary variables

After desugaring, the control flow graph may contain a large number of temporary variables. For instance, Code Listing 19 contains four temporary variables `_tmpvar1` to `_tmpvar4`. We observe each temporary variable is used only once and has no effect in the future. Therefore, we enable deleting temporary variables in the desugared code. Our approach is to add del statements in the control flow graph. In this way, the analysis would handle temporary variable deletion by itself. Code Listing 21 shows a piece of desugared code with temporary variables deleted.



(a) A __delitem__ is found (b) A __delitem__ is not found

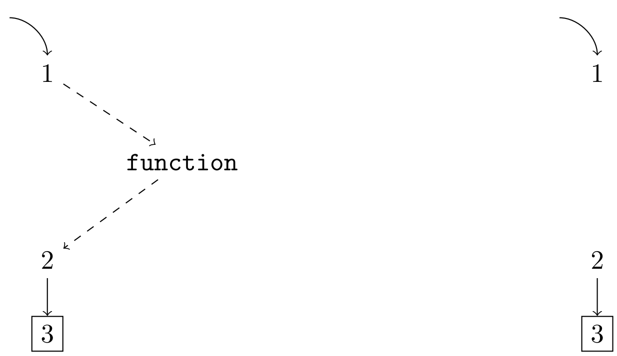
Figure 40: Evaluating a `del a[i]` invokes `__delitem__` implicitly

```
[res = func(*args, **kwargs)]1
```

```
# call node of a function
[func(*args, **kwargs)]1

# return node of a function
[_tmpvar1]2

[res = _tmpvar1]3
```

(a) Original form (b) Desugared form

Figure 41: How to desugar a function call



(a) The body of a function is found (b) The body of a function is not found

Figure 42: Two kinds of control flow graphs for a function call

```
[res = Class()]1
```

(a) Original form

```
# call node of __new__
[Class.__new__(Class)]1

# return node of __new__
[_tmpvar1]2

# call node of __init__
[Class.__init__(_tmpvar1)]3

# return node of __init__
[_tmpvar2]4

[res = _tmpvar2]5
```

(b) Desugared form

Figure 43: How to desugar a class initialization call

(a) (b)

Figure 44: Two kinds of control flow graphs for a class initialization

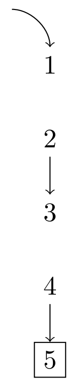


Figure 45: The uniform control flow graph for a call

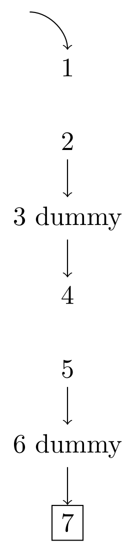


Figure 46: The extended uniform control flow graph for a call

Listing 21 Desugared code with temporary variables deleted

```
_tmpvar1 = '3'
_tmpvar2 = to_integer(_tmpvar1)
_tmpvar3 = _tmpvar2.bit_length
_tmpvar4 = _tmpvar3()
length = _tmpvar4

del _tmpvar1
del _tmpvar2
del _tmpvar3
del _tmpvar4
```

7 Type analysis for Python

Type analysis determines which types a variable may have at the exit of each program point [Van der Hoek and Hage, 2015]. However, Python is slightly different since the type of a variable in Python is the type of its bound value. Furthermore, in statically-typed languages such as Haskell or C, the type of a variable is determined at compile time. Python is dynamically typed, which means our type analysis is a kind of approximation. Our strategy is to statically collect all types of variables at each program point.

In this section we shall explain the type analysis operating on the control flow graphs described in Section 6. At first we describe an abstract value representing the run-time value with the help of a lattice. Secondly we formulate the type analysis as an instance of a dynamic monotone framework.

7.1 Points-to analysis

Code Listing 33 shows a method call `generate` on two different class instances `list_generator` and `set_generator`. In order to model such method calls in data flow analysis, the type of the receiver object should be known. We use points-to analysis to statically compute heap approximations. In fact, Python allocates all objects on the heap and these objects are recycled automatically by the Python garbage collector.

7.2 The analysis lattice

An abstract value is described by a **Value** tuple:

$$\textbf{Value} = \textbf{AnalysisType} \times \textbf{TypeshedType} \times \textbf{ArtificialType}$$

Each component of the tuple is described in the following sections.

7.2.1 AnalysisType

AnalysisType models types occurring in the source code to be analyzed. The analysis distinguishes six kinds of types. As an example, we create a Python file named `analysis_module.py`. Code Listing 34 shows the content of `analysis_module.py` and the comments explain which abstract types they have in our analysis.

1. **AnalysisModule**. Each represents a module object occurring in the source code.

2. **AnalysisClass**. Each represents a class object occurring in the source code.

3. **AnalysisFunction**. Each represents a function object or a generator object occurring in the source code.

4. **AnalysisMethod**. Each represents a method object occurring in the source code.

5. **AnalysisDescriptor**. Each represents a descriptor object occurring in the source code.

6. **AnalysisInstance**. Each represents a class instance occurring in the source code.

7.2.2 TypeshedType

TypeshedType models types occurring in the Python standard library. We retrieve the types from the project `typeshed`.

1. **TypeshedModule**. Each represents a module object occurring in the Python standard library.

2. **TypeshedClass**. Each represents a class object occurring in the Python standard library.

3. **TypeshedFunction**. Each represents a function object occurring in the Python standard library.

4. **TypeshedDescriptorGetter**. Each represents a descriptor object occurring in the Python standard library. We only implement the descriptors that support attribute lookup since attribute storage and deletion have no meaning in typeshed.

5. **TypeshedInstance**. Each represents a class instance occurring in the Python standard library.

Some types are used during parsing the typeshed project. They are described as follows:

1. **TypeshedAssign**. Each represents an assignment statement occurring in the typeshed project.

2. **TypeshedImportedModule**. Each represents an imported module module within a module occurring in the typeshed project.

3. **TypeshedImportedName**. Each represents an imported name within a module occurring in the typeshed project.

As an example, we create a Python stub file named `typeshed_module.pyi`. Code Listing 35 shows the content of `typeshed_module.pyi` and the comments explain which abstract types they have in our analysis.

7.2.3 ArtificialType

ArtificialType models some built-in types that we would like to enhance. For instance, according to typeshed, the built-in function `sum` has the signature listed in Code Listing 36. However, the return type is inferred as `Any` since our analysis has limited support for `typing.TypeVar`. We investigated our example projects and found that the return type of `sum` must be `int`. Our approach is to use an artificial function to represent `sum` to get more precise analysis result. Code Listing 37 shows the effect of an artificial class.

1. **ArtificialClass**. Each represents an enhanced class occurring in the typeshed project.
2. **ArtificialFunction**. Each represents an enhanced function occurring in the typeshed project.
3. **ArtificialMethod**. Each represents an enhanced method occurring in the typeshed project.

7.3 The analysis components

Abstract addresses

A variable may be bound to an abstract value that is allocated on heap. Every time a class instance is created, the **Record** function is used to create a heap context. Abstract addresses are elements of the set $\mathcal{P}(\mathbf{HContext})$.

$$hcontext \in \mathcal{P}(\mathbf{HContext})$$

Abstract scopes

A variable in Python can be classified into three scopes. They are:

$$scope \in \mathbf{Scope} = \{\mathbf{local},\ \mathbf{nonlocal},\ \mathbf{global}\}$$

Abstract names

An abstract name consists of the identifier of a variable and a scope.

$$name \in \mathbf{Name} = \mathbf{Identifier} \times \mathbf{Scope}$$

Abstract namespaces

An abstract namespace maps abstract names to abstract values.

$$namespace \in \mathbf{Namespace} = \mathbf{Name} \mapsto \mathbf{Value}$$

Abstract frames

An abstract frame is a four-tuple $(\mathbf{Locals} \times \mathbf{Back} \times \mathbf{Globals} \times \mathbf{Builtins})$. The type of each component is described as follows:

$$frame \in \mathbf{Frame} = (\mathbf{Namespace} \times \mathbf{Frame} \times \mathbf{Namespace} \times \mathbf{Namespace})$$

7.3.1 Abstract stacks

An abstract stack is a list of abstract frames. Code Listing 38 shows some operations supported by an abstract stack.

$$stack \in \mathbf{Stack} = [\mathbf{Frame}]$$

7.3.2 Abstract heaps

An abstract heap maps heap context elements to tuples of field names and abstract values. Each heap context element is abstraction of heap address. For simplicity, we use `Address` to denote heap context elements.

$$heap \in \mathbf{Heap} = \mathbf{Address} \mapsto (\mathbf{FieldName} \times \mathbf{Value})$$

7.3.3 Abstract states

The analysis will operate on an abstract state which consists of an abstract stack. Meanwhile, all heap objects share an abstract global heap.

$$\begin{aligned} state \in \mathbf{State} = &\mathbf{Stack} \\ &\texttt{with a global } \mathbf{Heap} \end{aligned}$$

Some nodes may be unreachable during data flow analysis. We use a least element $\bot$ to capture such cases. Therefore our new abstract state is define as:

$$\begin{aligned} state \in \mathbf{State} = &\mathbf{Stack}_\bot \\ &\texttt{with a global } \mathbf{Heap} \end{aligned}$$

The list of operations supported by an abstract state is listed in Code Listing 40.

7.4 The analysis instance

Our analysis[54] is an instance of a dynamic monotone framework.

$$(State,\ \mathcal{F}_{State},\ F,\ E,\ \iota,\ f.,\ \Psi,\ \Phi)$$

The instance gives rise to the following set of equations:

[54] adapted from Section 4.2 of [Van der Hoek and Hage, 2015]

$$
\begin{aligned}
A_\circ(\ell,\delta) &= \bigsqcup\{A_\bullet(\ell',\delta') \mid ((\ell',\delta'),(\ell,\delta)) \in F\} \sqcup \iota_E^{\ell,\delta} \\
&\quad \texttt{where } \iota_E^{\ell,\delta} = \begin{cases} \iota & \texttt{if } \ell \in E \land \delta = \Lambda \\ \bot & \texttt{if } \ell \notin E \end{cases} \\
A_\bullet(\ell,\delta) &= f_{\ell,\delta}(A_\circ(\ell,\delta)) \\
A_\bullet(\ell_r,\delta_r) &= f_{(\ell_c,\delta_c),(\ell_r,\delta_r)}(A_\circ(\ell_c,\delta_c), A_\circ(\ell_r,\delta_r)) \\
&\quad \texttt{where } (\ell_r,\delta_r) \in \mathit{ReturnProgramPoints} \\
\mathit{IF} &= \Psi_{\ell,\delta}(\ell,\delta) \cup \mathit{IF} \\
&\quad \texttt{where } (\ell,\delta)\ \mathit{in}\ F \\
F &= \{\Phi_{\ell,\delta}(e,\Lambda) \mid e \in E\} \cup \{\Psi_{\ell,\delta}(\ell',\delta') \mid ((\ell,\delta),(\ell',\delta')) \in F\}
\end{aligned}
$$

The analysis instance is represented as an instance of class `Analysis`. Code Listing 41 shows some operations supported by an `Analysis`.

7.5 The extremal value $\iota_{\ell,\delta}^{TA}$

The extremal value $\iota_{\ell,\delta}^{TA}$ specifies the available information when the analysis starts. It is an empty state which consists of an empty stack and an empty heap.

$$
\iota_{\ell,\delta}^{TA} = [] \quad \texttt{with a global } \mathbf{Heap} = \{\}
$$

7.6 The transfer function $f_{\ell,\delta}^{TA}$

The transfer function $f_{\ell,\delta}^{TA} = \mathbf{State} \to \mathbf{State}$ specifies how type information flows from the entry to the exit of a node based on ℓ and δ.

$$
f_{\ell,\delta}^{TA}(\mathit{state}) = \begin{cases} \bot & \texttt{if } \mathit{state} = \bot \\ \Omega_{\ell,\delta}^{TA}(\mathit{state}) & \texttt{otherwise} \end{cases}
$$

The type information should not be propagated if the state is $\bot$, which means the node is unreachable. Otherwise, $f_{\ell,\delta}^{TA}$ delegates the computation to $\Omega_{\ell,\delta}^{TA} : \mathbf{State} \to \mathbf{State}$.

For simplicity, we use `var_name` to represent a plain variable name, `func_name` to represent a function name and `class_name` to represent a class name.

7.6.1 Transfer functions for statements

Transfer functions for statements are used to deal with nodes with respect to intra-procedrual flow.

The transfer function for $[\texttt{ast.FunctionDef}]^\ell$

Code Listing 42 describes the transfer function for a function definition. The function performs a sequence of operations. At first it checks the function control flow graph out and adds the graph into the analysis. Secondly, the function default arguments, the function module and the function property (a generator function or a plain function) are computed. At last, it binds the function name to a value containing the function.

The transfer function for $[\texttt{ast.Return}]^\ell$

Code Listing 43 describes the transfer function for a return statement. The function first computes the value `node.value` to be returned and then writes the value into a special variable `RETURN_FLAG`.

The transfer function for $[\texttt{ast.Delete}]^\ell$

Code Listing 44 describes the transfer function for a del statement. The function first retrieves the name to be deleted and then performs a deletion by setting the written value to `None`.

The transfer function for $[\texttt{ast.Assign}]^\ell$

Code Listing 45 describes the transfer function for an assignment statement. The function proceeds by first identifying the type of `target` and then performing operations based on the type. Assigning value to a list or tuple is hard to deal with since value unpacking fully depends on run-time information. Therefore, if `target` has type `ast.List` or `ast.Tuple`, the function extracts all names within the target expression. These names will all be bound to `Any`.

The transfer function for $[\texttt{ast.Import}]^\ell$

Code Listing 46 describes the transfer function for an import statement. The function first extracts the imported module name and the module alias name. Then modules are loaded and returned by calling `import_a_module`. At last the name is bound to the module value. According to the language reference[55], if the alias name is not specified, the imported module will be a top-level module. For instance, `import a.b.c` actually binds name `a` to the module corresponding to `a`.

The transfer function for $[\texttt{ast.ImportFrom}]^\ell$

Code Listing 47 describes the transfer function for an importfrom statement. The function is more complex than that for the import statement because it allows both absolute importing and relative importing. The function first transforms `stmt.module` into an absolute module name. Second, it imports

[55] https://docs.python.org/3.7/reference/simple_stmts.html#the-import-statement

the module as that for $[\texttt{ast.Import}]^\ell$. At last it finds and binds names listed in `stmt.names`. It is possible that a name does not exist in the module. If so, the function will regard that name as a submodule and try to import it.

The transfer function for $[\texttt{ast.Global}]^\ell$

Code Listing 48 describes the transfer function for a global statement. The function first gets the name in the statement and then writes the name to `state` with an argument `"global"` indicating the analysis to find the name in the global scope.

The transfer function for $[\texttt{ast.Nonlocal}]^\ell$

Code Listing 49 describes the transfer function for a nonlocal statement. The function first gets the name in the statement and then writes the name to `state` with an argument `"nonlocal"` indicating the analysis to find the name in the enclosing scope.

The transfer function for $[\texttt{ast.While}]^\ell$, $[\texttt{ast.If}]^\ell$, $[\texttt{ast.Pass}]^\ell$, $[\texttt{ast.Break}]^\ell$ and $[\texttt{ast.Continue}]^\ell$

In the Python abstract syntax tree, the test condition of a while statement or an if statement has type `ast.expr` which can not introduce new bindings. After desugaring, the test condition of a while statement or an if statement is just a name. So it has no effect on an abstract state. Therefore $\Omega_{\ell,\delta}$ is an identity function.

The pass statement acts as a placeholder and causes no modification to an abstract state. In addition, `break` and `continue` alter the control flow graph but themselves have no effect on an abstract state.

To sum up, the transfer functions for these five statements are listed in Code Listing 50.

7.6.2 Transfer functions for inter-procedural counterparts

Transfer functions for statements are used to deal with nodes with respect to inter-procedrual flow.

The transfer function for $[\texttt{ast.ClassDef}]^{\ell_c}$

Code Listing 51 describes the transfer function for the call node of a class definition. The function prepares a new frame for executing the class body.

The transfer function for $[\texttt{ast.Call}]^{\ell_c}$

Code Listing 52 describes the transfer function for a function call node. The function obtains the call expression `call_expr` and prepares arguments in the local scope of the newly created frame.

In Python, $*$ can be used for unpacking positional arguments and $**$ can be used for unpacking keyword arguments. For instance, if `a = [1, "1", True]`, `func(*a)` is actually equivalent to `func(1, "1", True)`. In our analysis, all parameters of the function will be set to `Any` if there is any unpacking.

The transfer function for a right-hand-side expression $[\texttt{expr}]^{\ell_c}$

Code Listing 53 describes the transfer function for a call node of a right-hand-side expression. The branch `isinstance(call_expr, ast.Attribute)` corresponds to preparing arguments for possible descriptors. The other branches correspond to preparing arguments for possible special methods.

The transfer function for a left-hand-side expression $[\texttt{expr = value}]^{\ell_c}$

Code Listing 54 describes the transfer function for a call node of a left-hand-side expression. The function distinguishes two cases — `ast.Attribute` and `ast.Subscript`. In the former case, a descriptor setter may be called. In the latter case, `__setitem__` may be called.

The transfer function for a del statement $[\texttt{del expr}]^{\ell_c}$

Code Listing 55 describes the transfer function for a del statement. The function distinguishes descriptors and `__delitem__`.

The transfer function for an entry node $[\texttt{node}]^{\ell_n}$

Code Listing 56 describes the transfer function for an entry node of an inter-procedural call. If the entry node contains an `ast.arguments`, function arguments will be parsed based on the function parameters. Otherwise, the transfer function is just an identity function.

The transfer function for an exit node $[\texttt{node}]^{\ell_x}$

Code Listing 57 describes the transfer function for an exit node of an inter-procedural call. The function first calls `state.get_return_value()` to obtain `return_value` which is the value of the special variable `RETURN_FLAG`. If `return_value` is empty, it means the inter-procedural function will return the default value `None`.

The transfer function for $[\texttt{ast.ClassDef}]^{\ell_r}$

Code Listing 58 describes the transfer function for a return node which contains an `ast.ClassDef`. The function sequentially computes the class module, the class bases, the class body frame and the tuple containing the entry and exit labels of the class body.

The transfer function for $[\texttt{ast.Name}]^{\ell_r}$

Code Listing 58 describes the transfer function for a return node which contains an `ast.Name`. The function retrieves the value of `RETURN_FLAG` and writes it into the name `stmt.id`.

The transfer function for other return nodes $[\texttt{node}]^{\ell_r}$

Code Listing 60 describes the transfer function for other return nodes. The function just pops the last frame on the stack.

7.7 The flow creation function $\Psi^{TA}_{\ell,\delta}$

The $\Psi^{TA}_{\ell,\delta}$ function not only enables the analysis instance to discover new inter-flow edges and set up contexts during iteration, but also handles types that have no flows.

7.7.1 Context sensitivity manipulation functions

Section 3.2.3 has described how to apply context sensitivity by means of Equation 1 and 2. It works well in Java since every function belongs to some class. However, a function in Python can be declared outside a class so that it has no receiver object. We use the third function **MergeOrphan** to create a context when a function is called without a receiver object [Kashyap, Dewey, Kuefner, Wagner, Gibbons, Sarracino, Wiedermann, and Hardekopf, 2014].

$$\textbf{MergeOrphan} : \text{Lab} \times \text{HContext} \times \text{Context} \rightarrow \text{Context} \tag{8}$$

7.7.2 $\Psi^{TA}_{\ell,\delta}$ for $[\texttt{ast.ClassDef}]^{\ell_c}$

The function (Code Listing 61) is responsible for creating inter-procedural flows for a class definition.

7.7.3 $\Psi_{\ell,\delta}$ for $[\texttt{func\_name(*args, **kwargs)}]^{\ell_c}$

This function (Code Listing 62) is responsible for creating flows for standalone functions.

7.7.4 $\Psi_{\ell,\delta}$ for $[\texttt{class\_name(*args, **kwargs)}]^{\ell_c}$

This function (Code Listing 63) is responsible for creating flows for class initialization calls.

7.7.5 $\Psi_{\ell,\delta}$ for a right-hand-side expression $[\texttt{expr}]^{\ell_c}$

This function (Code Listing 64) is responsible for creating inter-procedural flows for special methods on the right-hand side of an assignment.

7.7.6 $\Psi_{\ell,\delta}$ for a left-hand-side $[\texttt{expr = value}]^{\ell_c}$

This function (Code Listing 65) is responsible for creating inter-procedural flows for special methods on the left-hand side of an assignment.

7.7.7 $\Psi_{\ell,\delta}$ for a del statement $[\texttt{del expr}]^{\ell_c}$

This function (Code Listing 66) is responsible for creating inter-procedural flows for special methods within a del assignment.

7.8 The flow collecting function $\Phi^{TA}_{\ell,\delta}$

Given a program point (ℓ, δ), the $\Phi^{TA}_{\ell,\delta}$ function enables collecting all flow edges within intra-procedural flow and inter-procedural flow . It behaves differently depending on the property of the node corresponding to ℓ.

$\Phi_{\ell,\delta}$ for a call node $[\texttt{node}]^{\ell_c}$

This function (Code Listing 67) collects flows corresponding to a call node.

$\Phi_{\ell,\delta}$ for an exit node $[\texttt{node}]^{\ell_x}$

This function (Code Listing 68) collects flows corresponding to an exit node.

$\Phi_{\ell,\delta}$ for any other node $[\texttt{node}]^{\ell}$

This function (Code Listing 69) collects flows corresponding to any other node.

$\Phi_{\ell,\delta}$ for every node $[\texttt{node}]^{\ell}$

In the control flow graph, a label ℓ can only belong to one kind of above three nodes. Therefore, our $\Phi_{\ell,\delta}$ function is defined as the function in Code Listing 70.

7.9 Typeshed types in the type analysis

We use typeshed to retrieve type information of objects within the Python standard library. Since typeshed-client is just a typeshed parser, we further extend it to support type information retrieval. Code Listing 71 gives a general idea of how we improve typeshed-client.

7.10 Artificial types in the type analysis

Artificial types are used to represent types that have effects during analysis. For instance, `object.__new__(cls)` is to create a class instance which has type `cls`. In our analysis, the return of `object.__new__(cls)` is an abstract heap address which represents an object allocated on heap. Code Listing 72 gives a general idea of how we make use of artificial types.

8 Experimental Evaluation

We at first made the following modifications to the example projects in order to circumvent some unsupported language features:

- Rewrite `super()`. In Python, each call to `super()` is transformed into `super(type1, type2)`[56]. We transform them manually since it is tough to perform the transformation in our analysis.
- Remove Python 2.x code. Some modules in Python 2.x do not exist in Python 3.7.
- Rewrite `import *` to a list of imported names based on the special attribute `__all__`[57].

We here describe a case where crude and refined analyses yield different values using Code Listing 22. In crude analysis, `ins.cls_var` returns a value containing `cls_var` which comes from the class body of `Cls` and an `int` object which comes from a call to `__get__`. On the contrary, in refined analysis, `ins.cls_var` returns `cls_var` only since instance variables take precedence over non-data descriptors.

Listing 22 Refined analysis and crude analysis yield different analysis results

```
class ClsVar:
    def __get__(self, obj, type=None):
        return 1

class Cls:
    # a non-data descriptor
    cls_var = ClsVar()

ins = Cls()
attr_value = ins.cls_var
```

In this section we describe the implementation of our type inferencer, the projects to be tested, the evaluation results and the answers to research questions.

8.1 The implementation

The implementation consists of three phases. In the first phase, an abstract syntax tree is obtained by first calling `ast.parse`[58] and then by desugaring each complex construct described in Section 6.1. In the second phase, the abstract

[56] https://docs.python.org/3.7/library/functions.html#super
[57] https://stackoverflow.com/questions/44834/what-does-all-mean-in-python
[58] https://docs.python.org/3.7/library/ast.html#ast.parse

syntax tree is transformed into a control flow graph with the help of our control flow graph generator. In the last phase, the control flow graph is read by the type inferencer which is written in Python.

The open source project is available at dmf[59]. It consists of three components which implement the phases described above.

8.2 The example projects

No.	Name	LOC
1	pyshorteners	522
2	google-api-python-client	2618
3	sqlparse	2614
4	feedparser	4089
5	configobj	1047
6	html2text	1359
7	twitter	2480

Table 7: Example projects

1. The project pyshorteners[60]. It is a Python library to shorten and expand urls.
2. The project google-api-python-client[61]. It provides developers with simple access to many Google's discovery based APIs.
3. The project sqlparse[62]. It is a SQL parser that offers support for parsing, splitting and formatting SQL statements.
4. The project feedparser[63]. It is a library for downloading and parsing syndicated feeds.
5. The project configobj[64]. It is a powerful ini file reader and writer.
6. The project html2text[65]. It is used to convert a HTML page into plain ASCII text.
7. The project twitter[66]. It is is an API access tool for Twitter that has a command-line program and an IRC bot.

[59] https://github.com/LayneInNL/dmf
[60] https://github.com/ellisonleao/pyshorteners
[61] https://github.com/googleapis/google-api-python-client
[62] https://github.com/andialbrecht/sqlparse
[63] https://github.com/kurtmckee/feedparser
[64] https://github.com/DiffSK/configobj
[65] https://github.com/aaronsw/html2text
[66] https://github.com/python-twitter-tools/twitter

8.3 Result

The experiments were performed on an machine with a Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz processor with 15.2 GiB of RAM running Fedora Linux 35.

8.3.1 Run time

As for each example project, we separately performed a 1-object-sensitive type analysis with a 1-context-sensitive heap and a 2-object-sensitive type analysis with a 1-context-sensitive heap. The results are shown in Table 8 and 9.

The second column and the third column of Table 8 and 9 show the run time of crude and refined analyses respectively. All run time is measured in seconds. `N/A` denotes the analysis can not finish the analyzing in 90 minutes (5400 seconds).

Name	Crude	Refined
pyshorteners	0.32	0.28
google-api-python-client	482.06	498.37
sqlparse	`N/A`	`N/A`
feedparser	`N/A`	`N/A`
configobj	125.58	113.75
html2text	160.01	166.45
twitter	214.73	221.86

Table 8: The run time of the 1-object-sensitive type analysis with a 1-context-sensitive heap

Name	Crude	Refined
pyshorteners	0.28	0.23
google-api-python-client	481.06	497.83
sqlparse	`N/A`	`N/A`
feedparser	`N/A`	`N/A`
configobj	115.25	115.87
html2text	158.31	165.63
twitter	216.03	222.91

Table 9: The run time of the 2-object-sensitive type analysis with a 1-context-sensitive heap

8.3.2 Type difference

Table 10 records two metrics 'Difference with temps' and 'Difference without temps' produced by our type inferencer taking each example project as input. We aim to calculate how many variables will have different types under crude analysis and refined analysis. We also would like to know how temporary variables affect the statistics.

For simplicity, we define a variable occurring in the analysis as a *considered variable*. A variable is a *distinct variable* if the inferred types are different between crude analysis and refined analysis. Each metric is a tuple containing the count of distinct variables and the count of considered variables. For instance, the cell $(45704, 400344)$ in row 7 and column 2 of Table 10 states that in all 400344 considered variables, there are 45704 distinct variables.

We use a simple program to explain how our metrics are computed. Code Listing 23 shows the possible analysis results of program point 1 and program point 2 in crude analysis and refined analysis respectively. There are two variables `_tmpvar1` and `res`. So the number of considered variables is 2. Furthermore, since only the temporary variable `_tmpvar1` has different types in two analyses, the number of distinct variables is 1. Therefore 'Difference with temps' in Code Listing 23 is $(1, 2)$. The metric 'Difference without temps' is computed similarly except that all temporary variables are not considered.

Listing 23 How metrics are counted in type difference

```
# curde analysis:
# program point 1
local_namespace1 = {_tmpvar: {AnalysisFunction, AnalysisMethod}}
# program point 2
local_namespace2 = {res: {int}}

# refined analysis:
# program point 1
local_namespace1 = {_tmpvar: {AnalysisMethod}}
# program point 2
local_namespace2 = {res: {int}}
```

Name	Difference with temps	Difference without temps
pyshorteners	(0, 1157)	(0, 390)
google-api-python-client	(22738, 346600)	(0, 82563)
sqlparse	N/A	N/A
feedparser	N/A	N/A
configobj	(45704, 400344)	(0, 54855)
html2text	(6240, 241895)	(0, 37249)
twitter	(61452, 301663)	(0, 23640)

Table 10: Type difference

8.4 Discussions

8.4.1 The answers to research question 1

It can be seen from Table 8 or 9 that except `sqlparse` and `feedparser`, our type inferencer can finish the analyzing in 500 seconds. In addition, though `google-api-python-client`, `sqlparse` and `twitter` have similar lines of code, the run time for `google-api-python-client` is at least two times longer than that for `twitter`. To make the situation worse, the analysis for `sqlparse` is even unfinished. After investigating these two projects, we found that `sqlparse` has a large number of configuration modules. The inferencer has to first initialize these modules and the classes within them. This process takes a very long time.

Comparing Table 8 with 9, the run time still remains stable under different object sensitivity depths. One possible reason is the example projects employ a large amount of external code so that the inferred types turns to `Any` in most cases.

We came to the conclusion that in general the running time of our analysis highly depends on the source code to be analyzed.

8.4.2 The answers to research question 2

The experimental result listed in the third column of Table 10 shows that crude analysis and refined analysis produced basically the same types. Especially the number of distinct variables were all 0. Therefore we came to the conclusion that as far as these example projects were considered, crude analysis and refined analysis were the same.

8.4.3 The answers to research question 3

After looking into the implementation of Python and the example projects, we found out several possible answers.

Python attribute lookup operations

A method call usually has the form `receiver.method(*args, **kwargs)`. We will use an example code to illustrate this situation. Code Listing 24 shows the type of `result` is determined by `a_ins.a_func()`. Crude analysis may infer that `a_ins.a_func` has two types `AnalysisFunction` and `AnalysisMethod`. However it will not affect the type of `result` since the call to `AnalysisFunction` fails due to no enough arguments.

Desugaring of language constructs

Due to desugaring, the method call `a_ins.a_func()` in Code Listing 24 is desugared into Code Listing 25. The inferencer only cares about variables occurring in the original programs. That is, the type of `result` is the same in both two kinds of analyses, though two `_tmpvar1` have different types.

Listing 24 The behavior of function calls may reduce type difference

```python
class AClass:
    def a_func(self):
        ...

a_ins = AClass()
# a_ins.a_func is actually a method
result = a_ins.a_func()
```

Listing 25 Desugaring may reduce type difference

```python
_tmpvar1 = a_ins.a_func

# in crude analysis
_tmpvar1: AnalysisFunction | AnalysisMethod = a_ins.a_func
# in refined analysis
_tmpvar1: AnalysisMethod = a_ins.a_func

result = _tmpvar1()
```

Python programming style

Python is dynamically typed and programmers may perform attribute access operations on almost any object. Code Listing 26 demonstrates a piece of valid Python code. The class instance `a_ins` has two attributes with the same name `a_field`: one is in the instance dictionary and the other is in the class dictionary.

However such a programming style may impair the readability. Programmers have to spend time on understanding which one will be used at each program point. In practice, programmers tend to use different names to represent variables of classes and variables of class instances.

Listing 26 Python programming style may reduce type difference

```python
class AClass:
    a_field = 1
    def __init__(self):
        self.a_field = 1
a_ins = AClass()
```

In addition, Python programmers tend to import and call external libraries to finish programming tasks, which complicates the type inference since our analysis does not support the external projects.

Incomplete class information

Our type inferencer can not infer all class type information statically. For instance, in Code Listing 27, `AClass` inherits from `deque`. Since the inferencer may know nothing about the bases of `deque`, `AClass.__mro__` is marked as incomplete. To keep sound, `AClass.__mro__` is replaced with `(AClass, Any)`. Besides, the class `__mro__` is accessed every time attribute access takes place. With incomplete class information, the resulted type of an attribute access operation tend to be `Any`. Any attribute access related to `Any` always returns `Any`.

Listing 27 Incomplete class information may reduce type difference

```
from collections import deque

# AClass.__mro__ is incomplete
class AClass(deque):
    pass

a_ins = AClass()
result: Any = a_ins.a_field
```

Not often used advanced features

Python is a scripting language and is designed for implementing new functionality quickly. Nevertheless, programmers usually use Python to deal with ordinary tasks which do not require advanced features. As a consequence, refined analysis contributes no precision enhancement to the analysis result.

One investigated project rich[67] whose lines of code are 19230 employs data descriptors. It is supposed that refined analysis would be more precise than crude analysis when taking such a feature into account.

[67] https://github.com/Textualize/rich

9 Conclusions

In this thesis we described a object-sensitive type analysis for Python. The feature of dynamic method resolution in Python implies the control flow graph is being constructed by means of data flow information. Furthermore, points-to information enriches the data flow information, which in turn expands the control flow graph. To this end we established a dynamic monotone framework by borrowing ideas from [Nielson, Nielson, and Hankin, 2004; Fritz and Hage, 2017; Van der Hoek and Hage, 2015].

We specified the type analysis as an instance of a dynamic monotone framework. On top of that, we simulated object-oriented design in Python to support all commonly used special methods.

Our experimental evaluation mainly aimed to answer whether crude analysis and refined analysis would yield significantly different types. If not, other type inference tools may just implement crude analysis. Our experiments showed that refined analysis did not improve the analysis precision a lot. At last we discussed some possible answers to why refined analysis failed to make an improvement based on our observations and experience.

10 Limitations

Python does not support tail recursion elimination[68]. The Python interpreter stack has the default maximum recursion depth of 1000. If the depth of function calls exceeds this number, a `RecursionError` will be raised. For instance, the maximum recursion depth may be exceeded when parsing a control flow graph. Users may change the limit[69] by calling `sys.setcursionlimit()`.

For memory-intensive applications, Python may crash due to stack overflow. Users may change the thread stack size by calling `resource.setrlimit()`[70] on Linux.

Our type inferencer now has only implemented a small set of methods of some built-in container types such as `list`, `tuple` and `set`. Supporting all of them takes great effort.

[68] https://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html
[69] https://docs.python.org/3.7/library/sys.html#sys.setrecursionlimit
[70] https://docs.python.org/3/library/resource.html#resource.setrlimit

11 Future Work

In this section we shall describe some features that can be improved or implemented in the future. Section 11.1 introduces three functions may be beneficial to the enhancement of the analysis precision. Section 11.2 introduces a helper tool that could make our tool more user-friendly. The last subsection lists some unsupported Python language features.

11.1 Enhancement of analysis precision

Recency abstraction

Our type inferencer only allows weak updates on fields of heap allocated objects. It is unknown that whether recency abstraction [Balakrishnan and Reps, 2006] could improve analysis precision for Python projects, although [Jensen, Møller, and Thiemann, 2009] has demonstrated that recency abstraction helps the analysis yield good precision for Javascript projects.

Exception analysis

Our type inferencer does not take exception handling into account and thus the analysis may be unsound. To make it sound, it would be good to add *exception analysis*. Actually exception analysis and points-to analysis rely on each other. The former depends on call graph information constructed in points-to analysis to complete exception handling. The points-to set produced by the latter is also influenced by exception analysis because of exception object assignments.

Due to significant overhead of exception-chain analysis [Fu and Ryder, 2007], Bravenboer and Smaragdakis purpose to embed an on-the-fly exception analysis in context-sensitive points-to analyses [Bravenboer and Smaragdakis, 2009]. However, the result shows that it results in a disproportionate amount of time being spent on exception analysis in consequence of a large number of exception objects [Smaragdakis, Bravenboer, and Lhoták, 2011]. So in practice, exception objects are coarsened relative to ordinary objects.

Class inheritance in typeshed

Our type inferencer can not handle class inheritance in typeshed. Therefore the tool may raise `AttributeError` if an attribute is not found. Adding support in class inheritance in typeshed eliminates such attribute lookup errors. Code Listing 28 shows the class `array` has two base classes `MutableSequence` and `Generic`. If the desired attribute is `array.__len__`, the analysis would find it in the base class `MutableSequence`.

Listing 28 An excerpt from array.pyi

```
from typing import Generic, MutableSequence, TypeVar

_T = TypeVar("_T", int, float, str)

class array(MutableSequence[_T], Generic[_T]):
    ...
```

11.2 Enhancement of tool usefulness

In general the type information inferred by our tool is incomplete due to insufficient information. One approach to enhancing the precision is to take type hints provided by programmers into account. Code Listing 29 shows the return value `res` is annotated with the intended return type `int`. However it is risky since user-provided type hints may be different from types inferred by type-related tools.

Listing 29 An excerpt from types.pyi

```
# user specifies the return type of eval as int
res: int = eval("1")
```

11.3 Type information integration

For now our type inferencer is only able to infer types of each variable at each program point. The project pytype[71] supplies a tool `merge-pyi` to merge stub files into the Python source code. To this end, the inferencer should first dump type information into stub files.

11.4 Unsupported Python language features

11.4.1 Closures

A closure is a function that has access to a free variable from the nesting function. Code Listing 30 demonstrates a closure `printer`. Under the hood, the Python interpreter makes use of `co_freevars` that is a tuple of names of free variables and `co_cellvars` that is a tuple of names of cell variables. Some detailed explaination can be found in Anatomize Python's Closures[72] and How are python closures implemented?[73]. It is nontrivial for a static analysis tool to keep track of free variables used in a closure.

[71] https://github.com/google/pytype
[72] https:\/\/medium.com/swlh/anatomize-pythons-closures-dbf0fa217d38
[73] https://stackoverflow.com/questions/70773695/how-are-python-closures-implemented

Listing 30 A closure in Python

```
def outer():
    version = "1.0"
    def printer():
        print(version)

    return printer

printer = outer()
printer()
```

11.4.2 Metaclasses

Programmers may create custom metaclasses by inheriting from the built-in metaclass `type`. Code Listing 31 shows a custom metaclass.

Listing 31 A custom metaclass in Python

```
class NewMetaclass(type):
    pass
```

11.4.3 The `ast.YieldFrom` Expression

The expression `yield from` enables delegating part of its evaluations to another generator. Its full semantics can be found in PEP 380[74]. Code Listing 32 shows the value of `res` is yielded from `generator1()`.

Listing 32 An example of `yield from`

```
# suppose generator1 is a generator
def generator1(): ...

# suppose generator2 is also a generator
def generator2():
    res = yield from generator1()
```

11.4.4 Coroutines with async and await syntax

The introduction of `async def`, `async for`, `async with` and `await` makes coroutines a native Python language feature. All example projects in the experimentation contain no asynchronous code.

[74] https://peps.python.org/pep-0380/

References

Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *International Static Analysis Symposium*, pages 221–239. Springer, 2006.

Rastisalv Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–251, 1998.

Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 1–12, 2009.

David Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 47–56, 1988.

Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for python. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 89–98, 2017.

Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. Static value analysis of python programs by abstract interpretation. In *NASA Formal Methods Symposium*, pages 185–202. Springer, 2018.

Chen Fu and Barbara G Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *29th International Conference on Software Engineering (ICSE'07)*, pages 230–239. IEEE, 2007.

Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, 2001.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta informatica*, 7(3):305–317, 1977.

Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: A static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 121–132, 2014.

George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices*, 48(6):423–434, 2013.

William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *International Conference on Compiler Construction*, pages 47–64. Springer, 2006.

Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):1–53, 2008.

Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, 2002.

Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer Science & Business Media, 2004.

Erik M Nystrom, Hong-Seok Kim, and Wen-Mei W Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, 2004.

Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22 (1):162–186, 2000.

Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ..., 1978.

Olin Grigsby Shivers. *Control-flow analysis of higher-order languages or taming lambda*. Carnegie Mellon University, 1991.

Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, 2011.

Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

Henk Erik Van der Hoek. Object sensitive type analysis for php. Master's thesis, Utrecht University, 2014.

Henk Erik Van der Hoek and Jurriaan Hage. Object-sensitive type analysis of php. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 9–20, 2015.

A All Code Listings in Section 7

Listing 33 Two class instances and two method calls

```
class ListGenerator:
    def generate(self): return list()

class SetGenerator:
    def generate(self): return set()

list_generator = ListGenerator()
set_generator = SetGenerator()

a_list = list_generator.generate()
a_set = set_generator.generate()
```

Listing 34 The content of `analysis_module.py`

```
# Cls is an instnace of AnalysisClass
class Cls:
    # func is an instance of AnalysisFunction
    def func(self):
        pass

    # name is an instance of AnalysisFunction
    def name(self):
        return "name"

    # name is an instance of AnalysisDescriptor
    name = property(name)

# ins is an instance of AnalysisInstance
ins = Cls()

# method is an instance of AnalysisMethod
method = ins.func
```

Listing 35 The content of `typeshed_module.pyi`

```
# sys is an instance of TypeshedImportedModule
import sys

# ABC is an instance of TypeshedImportedName
from abc import ABC

# Cls is an instnace of TypeshedClass
class Cls:
    # func is an instance of AnalysisFunction
    def func(self) -> Any: ...

    # alias is an instance of TypeshedAssign
    alias = func

    # name is an instance of TypeshedDescriptorGetter
    @property
    def name(self) -> str: ...


# ins is an instance of TypeshedInstance
ins = Cls()
```

Listing 36 The function signatures of the built-in function sum

```
_AddableT1 = TypeVar("_AddableT1", bound=SupportsAdd[Any, Any])
_AddableT2 = TypeVar("_AddableT2", bound=SupportsAdd[Any, Any])

@overload
def sum(
    __iterable: Iterable[bool | _LiteralInteger],
    __start: int = ...
) -> int: ...
@overload
def sum(
    __iterable: Iterable[_AddableT1], __start: _AddableT2
) -> _AddableT1 | _AddableT2: ...
```

Listing 37 The effect of the artificial function sum

```
# without ArtificialFunction, res has type Any.
res: Any = sum([1, 2, 3])

# with ArtificialFunction, res has type int.
res: int = sum([1, 2, 3])
```

Listing 38 The operations supported by a stack

```
class Stack:
    # write local name to stack
    def write_local_name(self, name: str,
        value: Value) -> None: ...

    # write nonlocal name to stack
    def write_nonlocal_name(self, name: str,
        value: Value) -> None: ...

    # write global name to stack
    def write_global_name(self, name: str,
        value: Value) -> None: ...

    # read name from stack
    def read_name(self, name: str) -> Value: ...

    # delete name from stack
    def delete_name(self, name: str) -> None: ...

    # add a new frame to stack
    def add_new_frame(self) -> None: ...

    # pop the last frame from stack
    def pop_frame(self) -> None: ...

    # check if the function body is related to generator
    def is_generator(self) -> bool: ...
```

Listing 39 The operations supported by a heap

```
class Heap:
    # write name to heap based on address
    def write_name(self, address: Address,
        name: str, value: Value) -> None: ...

    # read name from heap based on address
    def read_name(self, address: Address,
        name: str) -> Value: ...

    # delete name from heap based on address
    def delete_name(self, address: Address,
        name: str) -> None: ...
```

Listing 40 The operations supported by a state

```python
class State:
    # write name to stack, None represents deleting
    def write_name_to_stack(self, name: str,type: str, value: Value | None) -> None: ...

    # read name from stack
    def read_name_from_stack(self, name: str) -> Value: ...

    # write name to heap, None represents deleting
    def write_name_to_heap(self, address: Address, name: str, value: Value | None) -> None: ...

    # read name from heap
    def read_name_from_heap(self, address: Address, name: str) -> Value: ...

    # evaluate an expression expr
    def compute_expr(self, expr: ast.expr) -> Value: ...

    # evaluate function default arguments
    def compute_func_defaults(self, function: ast.FunctionDef) -> Tuple: ...

    # evaluate function call arguments
    def compute_func_args(self, expr: ast.Call) -> Tuple: ...

    # parse function call arguments
    def parse_call_args(self, expr: ast.arguments, args: Tuple) -> None: ...

    # evaluate class bases
    def compute_class_bases(self, stmt: ast.ClassDef) -> List: ...

    # get current module name
    def get_curr_module(self) -> str: ...

    # get current package name of the current module
    def get_curr_package(self) -> str: ...

    # get the value of the special variable RETURN_FLAG
    def get_return_value(self) -> Value: ...
```

Listing 41 The operations supported by an analysis

```
class Analysis:
    flow: Set[Tuple]
    inter_flow: Set[Tuple]

    def checkout_cfg(self, program_point: ProgramPoint) -> CFG: ...
    def add_cfg(self, cfg: CFG) -> Tuple[int, int]: ...
    def get_stmt_or_expr(self, program_point: ProgramPoint) -> ast.stmt | ast.expr: ...
    def name_extractor(self, expr: ast.List | ast.Tuple) -> Set[str]: ...

    # import a module based on name
    def import_a_module(self, name: str) -> Value: ...
    # resolve a relative module name to its absolute name
    def resolve_name(self, name: str | None, package: str, level: str) -> Value: ...

    # attribute access storage or deletion on a set of objects
    def analysis_setters(self, receiver: Value, attr: str, value: Value | None) -> Value: ...
    # attribute access storage or deletion on a single object
    def analysis_setter(self, receiver: Value, attr: str, value: Value | None) -> Value: ...
    # attribute access lookup on a set of objects
    def analysis_getters(self, receiver, attr: str) -> Value:
    # attribute access lookup on a single object
    def analysis_getter(self, receiver, attr: str) -> Value:

    def get_call_label(self, return_label: int) -> int: ...
    def get_return_label(self, call_label: int) -> int: ...
    def get_two_return_labels(self, call_label: int) -> Tuple: ...

    def add_classdef_flow(program_point: ProgramPoint, return_label: int): ...
    def add_function_flow(program_point: ProgramPoint, function: AnalysisFunction, return_label: int): ...
    def add_method_flow(program_point: ProgramPoint, method: AnalysisMethod, return_label: int): ...
    def add_descriptor_flow(program_point: ProgramPoint, descriptor: AnalysisDescriptor, return_label: int): ...
```

Listing 42 The sketch of `transfer_FunctionDef`

```
def transfer_FunctionDef(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, node: ast.FunctionDef
) -> State:
    tp_cfg: CFG = analysis.checkout_cfg(program_point)
    tp_code: Tuple = analysis.add_cfg(tp_cfg)

    tp_defaults: Tuple = state.compute_func_defaults(node)
    tp_module: str = state.get_curr_module()
    tp_is_generator: bool = func_cfg.is_generator

    value: Value = Value()
    analysis_functioin: AnalysisFunction = AnalysisFunction(
        tp_module, tp_code,
        tp_defaults, tp_is_generator
    )
    value.inject(analysis_function)

    state.write_name_to_stack(node.name, "local", value)

    return state
```

Listing 43 The sketch of `transfer_Return`

```

def transfer_Return(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, node: ast.Return
) -> State:
    tp_return: Value = state.compute_expr(node.value)
    state.write_name_to_stack(RETURN_FLAG, "local", tp_return)

    return state
```

Listing 44 The sketch of `transfer_Delete`

```
def transfer_Return(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, node: ast.Delete
) -> State:
    # get the name in the return statement
    tp_name:str = node.targets[0].id

    state.write_name_to_stack(tp_name, "local", None)

    return state
```

Listing 45 The sketch of `transfer_Assign`

```
def transfer_Assign(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, node: ast.Assign
) -> State:
    target: ast.expr = node.targets[0]
    if isinstance(target, ast.Name):
        value: Value = state.compute_expr(node.value)
        state.write_name_to_stack(target.id, "local", value)
    # list or tuple can not be handled perfectly
    elif isinstance(target, (ast.List, ast.Tuple)):
        names: Set[str] = analysis.name_extractor(target)
        for name in names:
            state.write_name_to_stack(name, "local", Any)

    return state
```

Listing 46 The sketch of `transfer_Import`

```
def transfer_Import(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, node: ast.Import
) -> State:
    # absolute module name
    name: str = node.names[0].name
    # possible alias name
    asname: str | None = node.names[0].asname

    module: Value = analysis.import_a_module(name)

    if asname:
        name: str = asname
    else:
        name: str = name.partition(".")[0]
        module: Value = import_a_module(name)

    state.write_name_to_stack(name, "local", module)

    return state
```

Listing 47 The sketch of `transfer_ImportFrom`

```python
def transfer_ImportFrom(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, stmt: ast.ImportFrom
) -> State:
    # possible relative module name
    name: str | None = stmt.module
    # if it's a relative import
    if node.level > 0:
        package: str = state.get_curr_package()
        name: str = analysis.resolve_name(stmt.module, package, stmt.level)

    module: Value = import_a_module(name)

    # the name in the fromlist
    sub_name = stmt.names[0].name
    # the possible alias name in the fromlist
    sub_asname = stmt.names[0].asname

    if sub_name not in module:
        # sub_name is not found in the current module
        sub_module: str = f"{name}.{sub_name}"
        module: Value = import_a_module(sub_module)
    else:
        module: Value = getattr(module, sub_name)

    if sub_asname:
        name: str = sub_asname

    state.write_name_to_stack(name, "local", module)

    return state
```

Listing 48 The sketch of `transfer_Global`

```
def transfer_Global(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, node: ast.Global
) -> State:
    name: str = names[0]
    value: Value = state.compute_expr(ast.Name(id=name))
    state.write_name_to_stack(name, "global", value)

    return state
```

Listing 49 The sketch of `transfer_Nonlobal`

```
def transfer_Nonlocal(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, node: ast.Nonlocal
) -> State:
    name = names[0]
    value = state.compute_expr(ast.Name(id=name))
    state.write_name_to_stack(name, "nonlocal", value)

    return state
```

Listing 50 The sketch of `transfer_Identity`

```
def transfer_Identity(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, node: ast.While | ast.If | ast.Pass | ast.Break
    ↪   | ast.Continue
) -> State:

    return state
```

Listing 51 The sketch of `transfer_call_classdef`

```
def transfer_call_classdef(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    state.stack.add_new_frame()

    return state
```

Listing 52 The sketch of `transfer_call_normal`

```
def transfer_call_normal(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    call_expr: ast.Call =
    ↪  analysis.get_stmt_or_expr(program_point)

    state.stack.add_new_frame()
    args: Tuple = state.compute_func_args(call_expr)
    state.parse_call_args(call_expr.arguments, args)

    return state
```

Listing 53 The sketch of `transfer_call_right_magic`

```
def transfer_call_right_magic(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    call_expr: ast.expr =
    ↪  analysis.get_stmt_or_expr(program_point)
    state.stack.add_new_frame()

    if isinstance(call_expr, ast.BinOp):
        rhs_value: Value = state.compute_expr(call_expr.right)
        state.write_name_to_stak("1", "local", rhs_value)
    elif isinstance(call_expr, ast.Attribute):
        receiver_value: Value =
        ↪  state.compute_expr(call_expr.value)
        descriptors: Value = getattrs(receiver_value,
        ↪  call_expr.attr)
        for descriptor in descriptors:
            for idx, arg in enumerate(descriptor.args, 1):
                state.write_name_to_stack(str(idx), "local", arg)
    elif isinstance(call_expr, ast.Subscript):
        slice_value: Value = state.compute_expr(call_expr.slice)
        state.write_name_to_stack("1", "local", slice_value)

    return state
```

Listing 54 The sketch of `transfer_call_left_magic`

```python
def transfer_call_left_magic(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    assign_stmt: ast.Assign = analysis.get_stmt_or_expr(program_point)
    state.stack.add_new_frame()

    target: ast.expr = assign_stmt.targets[0]
    if isinstance(target, ast.Attribute):
        receiver_value: Value = state.compute_expr(target.value)
        rhs_value: Value = state.compute_expr(call_expr.value)
        descriptors: Value = analysis_setters(receiver_value, call_expr.attr, rhs_value)
        for descriptor in descriptors:
            for idx, arg in enumerate(descriotpr.args, 1):
                state.write_name_to_stack(str(idx), "local", arg)
    elif isinstance(target, ast.Subscript):
        receiver_value: Value = state.compute_expr(target.value)
        rhs_value: Value = state.compute_expr(call_expr.value)
        slice_value: Value = state.compute_expr(call_expr.slice)
        state.write_name_to_stack("1", "local", slice_value)
        state.write_name_to_stack("2", "local", rhs_value)

    return state
```

Listing 55 The sketch of `transfer_call_del_magic`

```
def transfer_call_del_magic(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    del_stmt: ast.Delete =
    ↪ analysis.get_stmt_or_expr(program_point)
    state.stack.add_new_frame()

    target: ast.expr = del_stmt.targets[0]
    if isinstance(target, ast.Attribute):
        receiver_value: Value = state.compute_expr(target.value)
        rhs_value: Value = state.compute_expr(call_expr.value)
        descriptors: Value =
        ↪ analysis.analysis_setters(receiver_value,
        ↪ call_expr.attr, None)
        for descriptor in descriptors:
            for idx, arg in enumerate(descriotpr.args, 1):
                state.write_name_to_stack(str(idx), "local", arg)
    elif isinstance(target, ast.Subscript):
        slice_value: Value = state.compute_expr(target.slice)
        state.write_name_to_stack("1", "local", slice_value)

    return state
```

Listing 56 The sketch of `transfer_entry`

```
def transfer_entry(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    expr: ast.Pass | ast.arguments =
    ↪ analysis.get_stmt_or_expr(program_point)

    target: ast.expr = del_stmt.targets[0]
    if isinstance(target, ast.arguments):
        state.parse_call_args(target)

    return state
```

Listing 57 The sketch of `transfer_exit`

```
def transfer_exit(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    return_value: Value = state.get_return_value()

    if len(return_value) == 0:
        # means no explicit return statement in the function body
        # None_Instance is a special type corresponding to None
        ↪  in our analysis
        return_value.inject(None_Instance)
        state.write_name_to_stack(RETURN_FLAG, "local",
        ↪  return_value)

    return state
```

Listing 58 The sketch of `transfer_return_classdef`

```
def transfer_return_classdef(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    stmt: ast.ClassDef = analysis.get_stmt_or_expr(program_point)

    module: str = state.get_curr_module()
    bases: List = state.compute_class_bases()
    frame: Frame = state.stack.pop_frame()
    return_label: int = program_point[0]
    call_label: int = analysis.get_call_label(return_label)
    code: Tuple = (call_label, return_label)
    analysis_class: AnalysisClass = AnalysisClass(module, bases,
    ↪  frame, code)

    value: Value = Value()
    value.inject(analysis_class)
    state.write_name_to_stack(stmt.name, "local", value)

    return state
```

Listing 59 The sketch of `transfer_return_name`

```
def transfer_return_name(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    expr: ast.Name = analysis.get_stmt_or_expr(program_point)

    return_value: Value = state.read_name(RETURN_FLAG)
    state.stack.pop_frame()
    state.write_name_to_stack(expr.id, "local", value)

    return state
```

Listing 60 The sketch of `transfer_return_others`

```
def transfer_return_others(
    analysis: Analysis, program_point: ProgramPoint,
    state: State
) -> State:
    state.stack.pop_frame()

    return state
```

Listing 61 The sketch of `detect_classdef_flow_edges`

```
def detect_classdef_flow_edges(
    analysis: Analysis, program_point: ProgramPoint
) -> None:
    call_label, call_context = program_point
    entry_label, exit_label = analysis.add_cfg(call_label)
    return_label: int = analysis.get_return_label(call_label)
    analysis.add_classdef_flow(program_point, return_label)
```

Listing 62 The sketch of `detect_func_flow_edges`

```
def detect_func_flow_edges(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, dummy_value: Value
) -> None:
    call_label, call_context = program_point
    entry_label, exit_label = analysis.add_cfg(call_label)
    return_label, dummy_return_label =
    ↪  analysis.get_two_return_labels(call_label)

    call_expr: ast.expr =
    ↪  analysis.get_stmt_or_expr(program_point)
    func_value: Value = state.compute_expr(call_expr.func)

    for function in func_value:
        analysis.add_function_flow(program_point, function,
        ↪  return_label)
```

Listing 63 The sketch of `detect_class_flow_edges`

```
def detect_class_flow_edges(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, dummy_value: Value
) -> None:
    call_label, call_context = program_point
    return_label, dummy_return_label =
    ↪  analysis.get_two_return_labels(call_label)

    call_expr: ast.Call = analysis.get_stmt(program_point)
    class_value: Value = state.compute_expr(call_expr.func)

    for cls in class_value:
        new_methods: Value = analysis.analysis_getattr(cls,
        ↪  "__new__")
        for new_method in new_methods:
            analysis_method: AnalysisMethod =
            ↪  AnalysisMethod(new_method, cls)
            analysis.add_method_flow(program_point, function,
            ↪  return_label)
```

Listing 64 The sketch of `detect_right_magic_flow_edges`

```
def detect_right_magic_flow_edges(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, dummy_value: Value
) -> None:
    call_label, call_context = program_point
    return_label, dummy_return_label = analysis.get_two_return_labels(call_label)

    expr: ast.expr = analysis.get_stmt(program_point)
    if isinstance(expr, (ast.BinOp, ast.UnaryOp)):
        if isinstance(expr, ast.BinOp):
            receiver_value: Value = state.compute_expr(expr.left)
            operator_name: str = numeric_methods[type(expr.op)]
        else:
            receiver_value = state.compute_expr(expr.operand)
            operator_name = unary_methods[type(expr.op)]
        special_methods: Value = analysis.analysis_getattrs(receiver_value, operator_name)
        for special_method in special_methods:
            analysis.add_method_flow(program_point, special_method, return_label)
```

Listing 65 The sketch of `detect_left_magic_flow_edges`

```python
def detect_left_magic_flow_edges(
    analysis: Analysis, program_point: ProgramPoint,
    state: State, dummy_value: Value
) -> None:
    call_label, call_context = program_point
    return_label, dummy_return_label = analysis.get_two_return_labels(call_label)

    stmt: ast.Assign = analysis.get_stmt(program_point)
    target: ast.expr = stmt.targets[0]
    if isinstance(target, ast.Attribute):
        receiver_value: Value = state.compute_expr(target.value)
        rhs_value: Value = state.compute_expr(stmt.value)
        descriptors: Value = analysis.analysis_setattrs(receiver_value, target.attr, rhs_value)
        for descriptor in descriptors:
            analysis.add_descriptor_flow(program_point, descriptor, return_label)
    elif isinstance(target, ast.Subscript):
        receiver_value: Value = state.compute_expr(target.value)
        special_methods: Value = analysis.analysis_getattrs(receiver_value, "__selitem__")
        for special_method in special_methods:
            analysis.add_method_flow(program_point, special_method, return_label)
```

Listing 66 The sketch of `detect_del_magic_flow_edges`

```python
def detect_del_magic_flow_edges(analysis: Analysis, program_point: ProgramPoint,
    state: State, dummy_value: Value
) -> None:
    call_label, call_context = program_point
    return_label, dummy_return_label = analysis.get_two_return_labels(call_label)

    stmt: ast.Delete = analysis.get_stmt(program_point)
    target: ast.expr = stmt.targets[0]
    if isinstance(target, ast.Attribute):
        receiver_value: Value = state.compute_expr(target.value)
        descriptors: Value = analysis.analysis_setattrs(receiver_value, target.attr, None)
        for descriptor in descriptors:
            analysis.add_descriptor_flow(program_point, descriptor, return_label)
    elif isinstance(target, ast.Subscript):
        receiver_value: Value = state.compute_expr(target.value)
        special_methods: Value = analysis.analysis_getattrs(receiver_value, "__delitem__")
        for special_method in special_methods:
            analysis.add_method_flow(program_point, special_method, return_label)
```

Listing 67 The sketch of `collect_call_flow_edges`

```python
def collect_call_flow_edges(
    analysis: Analysis, program_point: ProgramPoint
) -> List[Flow]:
    edges: List[Flow] = []

    for call, entry, exit, return in analysis.inter_flow:
        if call_point == program_point:
            edges.append((call, entry))
            edges.append((exit, return))

    return edges
```

Listing 68 The sketch of `collect_exit_flow_edges`

```python
def collect_exit_flow_edges(
    analysis: Analysis, program_point: ProgramPoint
) -> List[Flow]:
    edges: List[Flow] = []

    for call, entry, exit, return in analysis.inter_flow:
        if exit == program_point:
            edges.append((exit, return))

    return edges
```

Listing 69 The sketch of `collect_intra_flow_edges`

```python
def collect_intra_flow_edges(
    analysis: Analysis, program_point: ProgramPoint
) -> List[Flow]:
    edges: List[Flow] = []

    label, context = program_point
    for first_label, second_label in analysis.flow:
        if first_label == label:
            edges.append(
                ((first_label, context), (second_label, context))
            )

    return edges
```

Listing 70 The sketch of `collect_flow_edges`

```python
def collect_flow_edges(
    analysis: Analysis, program_point: ProgramPoint
) -> List[Flow]:
    call_flow: List[Flow] =
    ↪  collect_call_flow_edges(program_point)
    exit_flow: List[Flow] =
    ↪  collect_exit_flow_edges(program_point)
    intra_flow: List[Flow] =
    ↪  collect_intra_flow_edges(program_point)

    all_flow: List[Flow] = intra_flow + call_flow + exit_flow

    return all_flow
```

Listing 71 How does our typeshed parser work on `math.floor`

```
import math

# type signature of math.floor
def floor(__x: _SupportsFloatOrIndex) -> int: ...

# without typeshed, result has type Any since we know nothing
↪  about module math.
result: Any = math.floor(7.5)

# without typeshed but with typeshed-client, result has type Any
↪  since we only know math.floor is a function.
result: Any = math.floor(7.5)

# with our typeshed parser, result has type int since we retrieve
↪  the return type from math.floor.
result: int = math.floor(7.5)
```

Listing 72 How does artificial `object.__new__` work

```
class Cls:
    ...

# without artificial types, according to typeshed,
↪  allocated_object has type Any.
allocated_object = object.__new__(Cls)

# with artificial types, allocated_object has type Cls.
allocated_object = object.__new__(Cls)
```