Project Final Report

ePYt: Automatic type-aware test input generator for python

Team 2
Michael Tegegn, MyeongGeun Shin
Sihoon Lee, Yongwoo Lee

Abstract

Python does not have a high quality automatic test input generator. It's because Python uses duck types for variables and provides much freedom to type systems. So, automatic test input generators cannot feed the desirable instances for input, and results in ill-formed test inputs and low coverage. So we suggest a way to enhance type inference utilizing attribute information, and use such information to generate test inputs. This can lead us to an automatic test input generator which can infer type and make desirable inputs.

Contents

1. Introduction

Software testing is an important step to check if a certain program works properly. This is important because it allows programmers to find and fix potential bugs before publishing. Developers can avoid the financial and time loss that can occur when a bug occurs after publishing the program. However, making test suites by hand is inefficient because there are too many cases to consider. Also, humans are biased and may not consider all cases. Therefore automating software testing is highly beneficial.

There are some automatic software testing tools such as EvoSuite for java and Klover for C. There are also some tools for Python but they have several limitations due to the nature of the Python itself. First, Python is a dynamically typed language. A single variable can have multiple types in different places. So we cannot specify the type of a variable. And second, Python is an object-oriented language, so there is a high possibility that there are many user defined classes. Due to these limitations, the existing twos do not perform well. So we devised a way to overcome these limitations.

First of all, a tool called Pynguin creates random test suites based on type annotation information if there is information. However, many python source codes do not have type annotations. Also, it is difficult to find out the type of a variable due to the aforementioned characteristics of Python. To overcome this, we implement ePYt that uses static analysis to infer the types of variables and generate type annotations including user defined classes. And automatically creates a test suite for python using Pynguin.

2. Solution

(a) In a glance

- Pre-analysis: Statically collect class and function definitions and Dynamically analyze collected classes along with their attribute information.
- Analyzer : Analyze attribute usages inside functions
- Type Inferrer : Guesses parameter type
- Annotator : Annotate functions with type information
- Generator : Generate unit tests for the annotated functions

In short, we take Python file(s), and for each function we infer parameter types. And generate unit tests utilizing inferred type information.

(b) Existing Tools

a. pyanalyze: https://github.com/quora/pyanalyze

pyanalyze is a static analyzer that aims to programmatically detect common mistakes in python code by checking type annotations and finding dead code. After inspecting pyanalyze we found that some components including the type inference component for python expressions were not implemented.

b. Microsoft pyright: https://github.com/microsoft/pyright

pyrite is a static type checker that is more useful for large code bases as it is fast and does little analysis. It Infers python symbol types based on value assignment, return type, expected type, etc. We thought about building on top of pyrite but it is written in TypeScript which meant that we could not simply extend it.

c. Pynguin: https://github.com/se2p/pynguin.git

Pynguin is a general python unit test generator. Its goal is to fill the gap of dynamically typed and statically typed languages when it comes to automatic test generation. It relies on a set of predefined algorithms to generate unit tests based coverage information and different fitness functions. We investigated some other tools but couldn't find a good tool to build on for our approach. So we decided the best strategy would be to gain more type information about functions using static analysis by ourselves and generate unit tests using an off the shelf test generator in Pynguin. Our aim is to improve test quality of existing test input generators.

(c) Solution Components

- **Pre-analysis**
  In this component, we statically collect class and function definitions and Dynamically analyze collected classes along with their attribute information.
  (Analysis)
  First of all, Preanalyzer collects class and function definitions and dynamically gets attributes and base classes using the ‘inspect’ module.
  (Base class)
  In analyzing base classes, it can get all the base classes as analyzing recursively. To figure out properties initialized in the constructor, it statically analyzes Assign AST nodes in ‘__init__’ method with node visitor.
  (Property initialization)
  It also analyzes method invocations in constructor so that it can collect potential property initializations. Finally, it merges collected properties in the constructor of base class into derived classes.

- **Analyze - FileInfo**
  Since we are targeting functions, we should collect all the functions in Python file(s). So we first iterate over python files, and for each file we collect class definitions and then iterate function definitions inside(method definitions). And we also collect functions outside the class definitions too.

e.g. Example python file to analyze

```python
def ClassA:
    def method1: pass
    def method2: pass

def ClassB:
    def method3: pass
    def method4: pass

def func1: pass
```

ePYt will have store information with one FileInfo consists of,

---

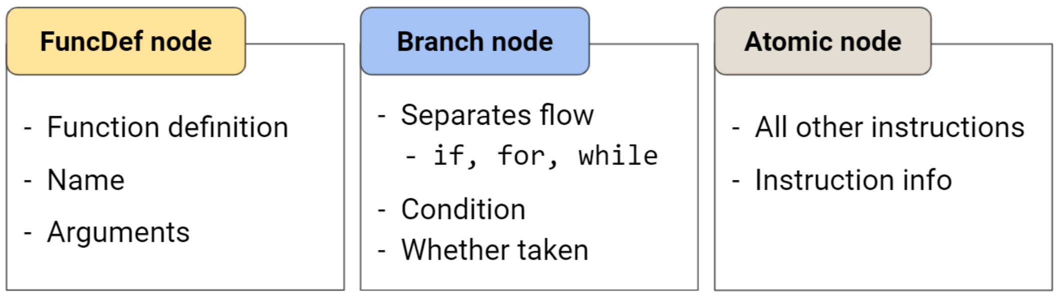
```
FileInfo [
	ClassDef “ClassA” [
		FuncDef “method1”
		FuncDef “method2”
	],
	ClassDef “ClassB” [
		FuncDef “method3”
		FuncDef “method4”
	],
	FuncDef “func1”
]
```

- **Analyze - Abstract domain**
  For analysis, we defined an abstract domain for variables. Since we only consider the type of the variable, abstract domain subsumes the attributes used in source code. So the abstract domain's class `BaseType` has a list of methods and properties. The difference between method and property is whether it is callable or not.
  Then we defined join, meet for the abstract value. And for variables that have already been typed(annotated or outer analyzer can annotate), we also made another type for that. And for the bottom value for analysis, we made `AnyType` which can be anything. (No information = Can be anything)
  And since python allows us to override a variable of another type(e.g. a=1; a=”str”) we added another class `FixedType`. If a function's parameter is overridden, we should not propagate further. So if such an assignment operation has occurred, we fix the abstract value with `FixedType` and stop propagating.
  And lastly, for the type inferring phase, a variable should be able to be inferred to a primitive type too. So we make an abstract value list for primitive which consists of `PrimitiveType`.
  For the ease of understanding, we will attach part of our code domain.py.

```python
class BaseType:
    def join(self, other):
        if isinstance(self, FixedType):
            return self
        if isinstance(other, FixedType):
            return other
        if isinstance(self, AnyType):
            return AnyType()
        if isinstance(other, AnyType):
            return AnyType()
        return self._join(self, other)

    def meet(self, other):
        if isinstance(self, AnyType):
            return deepcopy(other)
        if isinstance(other, AnyType):
            return deepcopy(self)
        return self._join(self, other)


class AnyType(BaseType):
    def __str__(self):
        return "AnyType"

    def __repr__(self):
        return "<Type AnyType>"


class HasAttr(BaseType):
    def __init__(self):
        self.properties = list()
        self.methods = list()
```

- **Analyze - Abstract memory**
  For each graph node, it should have an abstract memory which is a mapping from variable to abstract memory. And for abstract memory, it can be joined when merging incoming branches splitted in if-else-statements. Lastly, it should be able to be comparable when computing fixed point.
  So such functionalities are implemented in memory.py.

- **Analyze - Control Flow Graph**
  For inferring the type of variable, we need some static analyzer. In static analysis, Control Flow Graph is needed. Our CFG is builded by using AST module and represents flow with prev edge.
  First, we define three types of nodes. We replace all instructions with one of these three nodes.



FuncDef node is for funcion definition. Each FuncDef node has a function name and arguments list. Branch node is for branch statements such as if, for, while. They have a condition and a variable that indicates which branch it takes (True or False). Atomic node is for other instructions. And each node has a prev list of preceding instructions. Some AST nodes have a body, orelse element. They are the list of instructions and we think they can be a basic block because instructions in the same bodye are all executed or not executed together. In other words, when there are two instructions in one body, there is no case where only one is executed or only another is executed. So I link all nodes in the same body using prev edge.

```python
def parse(self, stmts):
    for stmt in stmts:
        if isinstance(stmt, self.handling_types):
            self.visit(stmt)
        else:
            node = Atomic([stmt], self.current_prev)
            self.nodes.append(node)
            self.current_prev = [node]
```

In some cases, we have to consider more than one prev node. For example, the instruction immediately following the if/else statement has both the last instruction of the if body and else body in prev list considering both the possibility of going to the True branch and False branch.
Reclusively linking nodes in this way, we can construct a control flow graph.

- **Analyzer - Semantic**
  Using CFG builded above, we have to transfer some information from one instruction to the entire code. Each node in CFG has a memory and a node gets prev's memory as input and updates it's memory as output. We iterate this process until all memory is unchanged.
  Memory is updated when the node has an instruction using one of our target variables(function arguments). When we meet an instruction that performs an addition operation on a variable, we know that the variable has an __add__ method. Or if we meet an instruction that accesses some property using ".", we can know that the variable has that property or method. We can't just see the using part because that variable name can be assigned different values and used. In this case, it should be fixed so that it is not further modified. So we consider three cases to update memory.
    - Builtin dunder method
      For example, in var + 1 statement, we can know that var has __add__ method. And from len(var) statement, we can know var has __len__ method.

So we saw all statements in atomic or branch nodes, and added dunder methods to target variables. This is an example code.

```
op_to_method = {
    ast.UAdd: '__pos__', ast.USub: '__neg__', ast.Invert: '__invert__',
    ...
    ast.In: '__contains__', ast.NotIn: '__contains_'}

func_to_method = {
    'abs': '__abs__', 'len': '__len__', 'int': '__int__', 'oct': '__oct__',
    ...
    'hash': '__hash__'}

def visit_UnaryOp(self, node):
    self.generic_visit(node)
    arg_key = ast.unparse(node.operand)
    if arg_key in self.args:
        self._add_method(arg_key, self.op_to_method[type(node.op)])
```

- User defined property or method
  When we meet var.foo or var.foo() statements, we can know var has property foo or method foo. For these user properties or user method, we parse and check ast.Call node and ast.Attribute node.

```
def visit_Call(self, node):
    fun_name = ast.unparse(node.func)
    if isinstance(node.func, ast.Attribute):
        arg_key = ast.unparse(node.func.value)
        if arg_key in self.args:
            self._add_method(arg_key, node.func.attr)
            return

def visit_Attribute(self, node):
    self.generic_visit(node)
    arg_key = ast.unparse(node.value)
    if arg_key in self.args:
        self._add_property(arg_key, node.attr)
```

- Fix
  As mentioned above, there are some cases where we have to fix the information. When there is a statement such as var+1; var=foo; var-1;. We have to __add__ method to var but we should not add __sub__ method to var.
  We detect it by using ctx attribute in ast.Name node. When the ctx is ast.Store node, that means the variable is changed to another value.

```
def visit_Name(self, node):
    if isinstance(node.ctx, ast.Store):
        self._has_assigned(node.id)
```

- **Type Inferrer**
  Type inferrer takes a function definition and conducts a static analysis. After static analysis, it determines types of function arguments through matching analyzed attributes the function uses with user-defined classes.

(Matching strategy)
Matching strategy is that user defined attributes should be a superset of analyzed attributes. Eventually, it infers the type of function arguments.

- **Annotator**
  It annotates inferred type-hints to source codes with analyzed information through inserting annotation nodes to AST nodes. Annotating multiple type-hints, it inserts an import statement which imports Union in the typing module, using it as argument type-hints.

```
def summarize_address_range(first: Union[IPv4Address, IPv4Interface, IPv6Address, IPv6Interface],
                            last: Union[IPv4Address, IPv4Interface, IPv6Address, IPv6Interface]):
    """Summarize a network range given the first and last IP addresses.

    Example:
        >>> list(summarize_address_range(IPv4Address('192.0.2.0'),
        ...                              IPv4Address('192.0.2.130')))
        ...                                #doctest: +NORMALIZE_WHITESPACE
        [IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'),
         IPv4Network('192.0.2.130/32')]
```

  In the case of annotating the types from external codes, it also inserts import statements. And, To address an error of not-defined class types, it adds empty classes which have the same name. Like this.

```
class IPv4Address:
    pass

class _BaseNetwork:
    pass
```

- **Generator**
  ePYt utilizes Pynguin which is an existing test case generator. It spawns a subprocess to run Pynguin and generates test cases. It provides code-level interface as well as subprocess spawning. But the default interface is spawning to avoid side-effects target codes cause. This is the result.

```
    var13 = {}
    var14 = module0._BaseV4(**var13)
    assert var14 is not None
    var15 = module0._IPv4Constants()
    assert var15 is not None
    var16 = None
    var17 = module0._BaseAddress()
    assert var17 is not None
    var18 = var17.__add__(var16)
    assert var18 is not None
```

3. Evaluation

For evaluation, we run two target scripts from the python standard library. These were ipaddress module and datetime module. These modules were were chosen as they are large files with multiple class definitions and methods each. To be exact, 15 custom class definitions with 150 methods for ipaddress and 7 custom class definitions with 160 methods for datetime. Loc is 2,290 for ipaddress and 2,524 for datetime. These two modules were chosen without further analysis to wholly estimate the performance of ePYt on larger python modules.

The most important part of ePYt is the attribute analysis to determine function signatures. After running the ePYt annotator. Results on the two modules can be seen in '*evaluate*' branch of our project Repo. Here are some examples of function signatures that are generated from the two files.

```
def __lt__(self, other: Union[IPv4Address, IPv4Interface, IPv6Address, IPv6Interface]):
    if not isinstance(other, _IPAddressBase):
        return NotImplemented
    if not isinstance(other, _BaseAddress):
        raise TypeError('%s and %s are not of the same type' % (self, other))
    if self._version != other._version:
        raise TypeError('%s and %s are not of the same version' % (self, other))
    if self._ip != other._ip:
        return self._ip < other._ip
    return False
```

ipaddress.py example function after annotation

Here, ePYt analyzer correctly infers _ip, and ._version attribute calls for the parameter 'other' and annotates it with Union type of all the available user defined types that contain these attributes. The function originally had no type information for its parameter. Many more of these types of signatures that accurately capture type information are present after annotation.

When the number of user defined classes are smaller (like in datetime module) with little inheritance, the type annotation can be even more accurate. See the example below.

```
def utcnow(cls: Union[datetime.datetime,]):
    """Construct a UTC datetime from time.time()."""
    t = _time.time()
    return cls.utcfromtimestamp(t)
```

datetime.py example function after annotation

If ePYt could not find accesses of parameter methods or attributes that are constraining, the annotated type signature can encompass a wide range of types. For example, type signatures like the following are quite commonly seen in smaller functions that do not give away too much information about the parameters. ePYt leaves parameters whose attributes or methods are not used in the function without type information.

```
def _fromtimestamp(cls, t: Union[datetime.timedelta, datetime.date,
    datetime.datetime, builtins.int, builtins.float, builtins.bool], utc, tz):
    ...
```

datetime.py example function after annotation

After fully annotating the functions, we ran pynguin and compared the results. We tried running pynguin with a 30 second budget with all the available test generation algorithms, namely DYNAMOSA, MIO, MOSA, RANDOM and WHOLE_SUITE. The results show that both the original files and ePYt annotated files have similar coverage with similar types of tests generated. For ipaddress, pynguin generated 73 passing 178 failing tests for original file and 67 passing and 190 failing tests for ePYt annotated file. However, the generated test cases still lack quality and are mostly characterized by having None value checks and only rarely called class methods or attributes. Examples are shown below.

```python
def test_case_30():
    var0 = module0.AddressValueError()
    assert var0 is not None
    var1 = module0._IPv4Constants()
    assert var1 is not None
    var2 = 'ma{jC'
    var3 = module0._BaseAddress()
    assert var3 is not None
    var4 = var3.__lt__(var2)
    assert var4 is not None
    var5 = module0._BaseNetwork(var3)
    assert var5 is not None
    var6 = module0._BaseV6()
    assert var6 is not None
```

```python
def test_case_30():
    var0 = module0._BaseV6()
    assert var0 is not None
    var1 = module0._BaseV6()
    assert var1 is not None
    var2 = module0._TotalOrderingMixin()
    assert var2 is not None
    var3 = b'\xde\x034&q\xae'
    var4 = module0._BaseNetwork(var3)
    assert var4 is not None
    var5 = var4.hosts()
    assert var5 is not None
```

Left: Pynguin test case with original file, Right: Pynguin test case with ePYt annotated file

The generated test cases failed to capture and capitalize on the function signature information. Perhaps a better test case generation algorithm is needed to efficiently use type hints. However, the accuracy of the type annotations suggest that ePYt can be even more powerful for modules with a large amount of custom classes. And the results also suggest that type annotations are not being fully utilized for test case generation and that further research in this area is needed.

4. Limitations and Future Work

- Circular imports

Currently, circular imports pose a threat to our pre-analysis component. Since we could not handle this with the time we have, we resorted to only dealing with one file python modules. Even though this issue does not pose any threat to our inference system(since modules can be compressed to one single file), it is needed to run ePYt easily on modules.
And also, it's hard to annotate directly into the file because of some circular imports. So we limited our version of ePYt to just target one file, not the directory(or a library) despite it's fully functional.

- Utilization of function signature information

Now, we do not use function signature information. In other words, if two methods with the same name receive different types of arguments, we cannot be distinguished. This can be solved using function signature information, and it is expected that it will be more helpful for type inference.

In addition, it can help type inference in other ways. If a target variable is used as an argument for another function call and we can know the function signature of that function, we can easily infer the type of target variable.

- Building the test generation engine

Currently, ePYt relies on Pynguin to generate the test cases. Since the results we got are not satisfactory enough, a test case generator can be built to focus on type annotations to generate inputs.

5. References

1. Pynguin: https://pynguin.readthedocs.io/en/latest/api.html
2. Lecture slides of IS593: Language-based Security https://github.com/prosyslab-classroom/is593-2020-spring
3. pyanalyze: https://github.com/quora/pyanalyze
4. Microsoft pyright: https://github.com/microsoft/pyright
5. Jedi: Static analyzer: https://github.com/davidhalter/jedi.git