ETH Zürich

Bachelor Project Description

Static Type Inference for Python

Mostafa Hassan
mostafa.hassan@inf.ethz.ch

supervised by:
Dr. Caterina Urban
Marco Eilers

March 17, 2017

1 Introduction

In Python, being a dynamically-typed language, the variables are bound to their type during execution time. This is appealing because programs have more type flexibility, and programmers do not need to explicitly write variable types, leading to shorter and quicker to write code. However, this comes at the cost of losing many static guarantees of program correctness. Some bugs, for example, that can be detected early enough in a statically-typed system, may hide from the developer in a dynamically-typed one, leading to a harder debugging experience.

Static type inference is the ability to automatically deduce the type of program expressions statically from their context (without the need to run the program), following a predefined static type system, which leads to stronger guarantees of the program correctness, and makes the programs less error-prone. See the following example:

```
def add_random(x):
        r = randint(0,9)
        return x + r
```

The function `add_random` takes an integer parameter. Assume that another function that returns a string is accidentally called instead of `randint`. Depending only on dynamic type binding, this mistake will hide before the execution, leading the program to crash at run-time.

In this project, we are designing and implementing a static type inference for Python 3, as part of Lyra and VerifySCION, two ongoing projects at the Chair of Programming Methodology at ETH Zurich*, which aim to develop a static analyzer and a program verifier for Python programs. In particular, we build on previous work [1][2], which provides a description of a static type inference system for a subset of Python, similar to RPython†.

The thesis [1] describes a type system developed for a restricted version of RPython, and presents a type inference implementation for this type system. The approach depends on the built-in Python AST module for providing the abstract syntax tree (AST) for Python programs, where each node in the AST denotes a construct occurring in the program code. This AST is then traversed in a depth-first manner to infer the types of its nodes. This implementation describes the inference for expressions (like numbers, lists, dictionaries, binary and unary operations, etc.), assignment statements, conditional statements, function and class definitions, function calls, and class instantiation. However, the approach has the following restrictions:

1. The type inference is context insensitive. The inferred parameter types and the return type of a function are independent of the context the function is being called in. See the following example:

   ```
   class A:
   ```

*`http://www.pm.inf.ethz.ch/research`

†`https://en.wikipedia.org/wiki/PyPy#RPython`

```
        def foo(self):
                return 1
class B:
        def foo(self):
                return "some string"

def f(some_class):
        return some_class().foo()
```

The inference presented in [1] will infer the return type of the function `f` to be either an integer or a string, independent of which class is passed as its argument. So with a function call such as `x = f(A)`, the type of `x` is inferred to be either an integer or a string, where in fact it should be just an integer.

2. The variables are not allowed to change type. For example, the following is not allowed:

```
x = 1
x = "some string"
```

`x` is first bound to an integer type, so it is not possible to rebind it to a string.

3. There is no multiple inheritance.

4. Class definitions have to be written before any statement that uses them.

```
def f():
        x = A()
        x.a()
class A:
        def a(self):
                print("Hello World!")

f()
```

The above code will raise an error because the definition of the class `A` comes after its instantiation in the function `f`.
It is also not mentioned in [1] how to handle the same situation with function calls.

2 Core goals

1. **Implement the work presented in [1] for Python 3.**

   The thesis [1] describes ideas for implementing a static type inference for RPython. However, it omits a lot of implementation details. It also does not describe how to handle the subtype relationships. The first goal is to provide a complete implementation for the ideas described in [1], targeting Python 3.

2. **Extend the implementation to include iterators, tuples and generators.** The paper [2] mentions these constructs, but no implementation details are given in [1].

3. **Lift the class and function order restriction (d).** That is to allow the classes and functions to be written in no specific order, and allow statements to use subsequent class and function definitions.

4. **Evaluate the implemented platform with multiple Python code examples.** We may use already existing PyPy[‡] or SCION[§] code.

3 Possible Extensions

1. **Extend the inference to enable multiple inheritance.**

   Within a complex inheritance hierarchy, the diamond problem arises. Python 3 uses the ***C3 linearization***[¶] algorithm to establish a method resolution order, that is to obtain the order in which methods and attributes are inherited in the presence of multiple inheritance. This method resolution order has to be statically resolved during the type inference.

2. **Extend the inference to allow variables to change type** (i.e., lift restriction (a)) by associating variable types to program points.

3. **Extend the inference to also target Python 2.**

   There are many differences between Python 2.x and 3.x. For example, the new-style classes[‖] have been introduced in Python 2.7, and the old-style classes have been removed in Python 3. There are also many other differences that affect the print function, integer division, exceptions handling, etc. We consider expanding the scope the project to include Python 2 as well.

4. **Extend the inference to be context-sensitive.** Refering back to the example given in restriction (a), the type of function call `f(A)` should be only an integer.

[‡] `https://pypy.org/index.html`
[§] `https://www.scion-architecture.net/`
[¶] `https://www.python.org/download/releases/2.3/mro/`
[‖] `https://www.python.org/doc/newstyle/`

4 Core Goals Timeline

By	Should be done
3/10/2017	Provide class and behaviour diagrams.
3/14/2017	Write initial class structure.
3/16/2017	Develop a strategy to tackle subtype relationships.
3/31/2017	Implement the inference for expressions, including but not limited to: numbers, lists, tuples, binary operations, etc.
4/14/2017	Implement the inference for statements, including but not limited to: return, for, while, if, etc.
4/21/2017	Implement support for function definitions and calls.
4/28/2017	Implement support for class definitions and instantiations.
5/05/2017	Wrap the whole project pipeline and test with code samples.

References

[1] Eva Maia. *Inferência de tipos em Python.* (Portuguese) [*Inference of types in Python*]. University of Porto, 2010.

[2] Eva Maia, Nelma Moreira & Rogério Reis. *A Static Type Inference for Python.* University of Porto.