Department of Computer Science
ETH Zurich

Department of Computer Science and Engineering
German University in Cairo

Bachelor Thesis

A Static Type Inference for Python 3

August, 2017

Author: Mostafa Hassan
Supervisors: Marco Eilers
Dr. Caterina Urban
Prof. Dr. Peter Muller

I confirm that this bachelor thesis is my own work and I have documented all sources and material used.

Zurich, August, 2017 Mostafa Hassan

Acknowledgments

Abstract

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Background Information **3**
2.1 Related Work 3
2.1.1 Type Inference Algorithms 3
2.1.2 Mypy [3] 3
2.1.3 Inferência de tipos em Python [1] 3
2.2 SMT Solving with Z3 [2] 4
2.2.1 Z3 semantics 4
2.3 Type System 7
2.3.1 Static Type and Dynamic Type Checking 8

3 Type System **9**

4 Type Inference **10**

5 Evaluation **11**

6 Future Work **12**

7 Conclusion **13**

1 Introduction

"The cost to fix an error found after product release was four to five times as much as one uncovered during design, and up to 100 times more than one identified in the maintenance phase.", reported by the System Science Institute at IBM. This fact justifies the increasing investments in software analysis, software verification and the need to make programs more reliable and safe.

In Python, being a dynamically-typed language, the variables are bound to their types during the execution time. This may look appealing because programs have more type flexibility, and they do not need to contain the writing overhead for the type system, leading to shorter and quicker to write code. However, this comes at the cost of losing many static guarantees of program correctness. Dynamically-typed languages perform type checking at runtime, while statically typed languages perform type checking at compile time. Therefore, some type errors that can be detected at compile time in a statically-typed system, may lead the system to crash at runtime in a dynamically-typed one, incurring high costs and a harder debugging experience.

See the following example:

```
num = 1
num = num + "2"
```

The intention of the above program was to add the number 2 to the variable `num`, not the `string` representation of this number. This tiny mistake goes unnoticed at compile time, and leads the program to raise an exception during runtime.

In this thesis, we are presenting a tool for static type inference and static type checking for a subset of Python 3. The aim of the tool is to gain the benefits of static typing while maintaining some (yet not all) dynamic features of Python. We discuss later the details of the dynamic limitations imposed on the inferable Python programs.

The type inference is based on a nominal static type system that we define in the next chapter. The type inference is intended to be integrated into 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.

We present a new approach for tackling the type inference problem, solely based Satisfiability Modulo Theories (SMT) solving. We also augment the SMT solver with several enhancements to account for the dynamic nature for Python. We will go through the approach details and the SMT encoding in later chapters.

This thesis describes the design and the implementation of the static type inference for Python 3, using Z3 SMT solver. It is divided into seven chapters. The second chapter presents the background information that will help the reader comprehend the rest of the thesis. It reviews the already existing type inference algorithms and the past work done in this area, explains the syntax and the type system rules of the subset of Python 3 that our tool supports and explains the SMT concepts that we will be using throughout the thesis.

In the third chapter, we introduce the encoding of the type system that we support in the SMT solver. We also state and justify the limitations of this type system.

In the forth chapter, we describe the design and the implementation of the type inference algorithm in depth. We explain the components of the tool and all the SMT axioms for all the language constructs that we support.

The fifth chapter explains the experiments we have done to test the tool. We also highlight the current limitations of the type inference and problems it faces with certain types of programs.

Finally in the sixth chapter, we review our work and suggest more improvements in the future.

2 Background Information

2.1 Related Work

Many attempts have been made to infer types for Python, each of which had its own goals and limitations. We discuss here some work that we have studied, and we present some of their limitations and how similar and/or different they are from our tool.

2.1.1 Type Inference Algorithms

There are two type inference algorithms primarily used at the time of writing this thesis: Hindley-Milner algorithm and the Cartesian Product algorithm.

Hindley-Milner

Cartesian Product

2.1.2 Mypy [3]

Mypy is a static type checker for Python. It depends on defining type annotations for almost all the constructs in the Python program to be checked. In addition, it performs local type inference. However, this type inference cannot be extended beyond local scopes. It requires that function definitions to be fully type-annotated and cannot infer function calls whose return type annotation is not specified. For example, mypy will fail to infer the type of variable x in the following program:

```python
def f():
        return "string"

x = f() # Infer type Any for x
```

What mypy intends to provide is closely related to the goal of our tool, that is to provide static type checking for the program. However, we aim to reduce (and sometimes eliminate) the writing overhead in defining the type annotations for the program constructs.

2.1.3 Inferência de tipos em Python [1]

The thesis [1] describes a static type system defined for a restricted version of RPython, and presents static type inference ideas based on this type system. The work presented in [1] also describes type inference implementation for Python expressions (like numbers, lists, dictionaries, binary and unary operations, etc.), assignment statements and conditional statements. It also gives an idea about inferring polymorphic and non-polymorphic function calls, class definitions and class instantiation. However, the approach they take has a handful of limitations and is not applicable to real Python code. It does not describe inferring function arguments, which is a critical step in the inference of function definitions and function calls. Accordingly, and similar to mypy, the inference they present is not extensible beyond local scopes inference.

2.2 SMT Solving with Z3 [2]

Satisfiability Modulo Theories (SMT) is a decision problem for first-order logic formulas. Which means, it is the problem which determines whether a given first-order logic formula, whose variables may have several interpretations, is satisfiable or not.

SMT solving is a generalization of boolean satisfiability (SAT) solving. It can reason about a larger set of first-order theories than SAT theories, like those involving real numbers, integers, bit vectors and arrays.

Z3 [2] is an efficient SMT solver, developed by Microsoft Research in 2007 with built-in support for theories of linear and nonlinear arithmetic, bit vectors, arrays, data-types, quantifiers and strings.

Z3 is now widely used in software analysis and program verification. For instance, 50 bugs were found in Windows kernel code after using Z3 to verify Windows components.

In our static type inference tool, we depend primarily on Z3 to provide a types model that satisfies all the Python program semantics.

2.2.1 Z3 semantics

We explain here all the relevant Z3 semantics that we will be using in our tool. For convenience, we will provide the explanation of these semantics in Z3Py, a Python interface for the Z3 solver, since we will be using this interface semantics throughout this thesis. This section targets those who are new to Z3. Those who are already familiar with these Z3 semantics can skip this section.

Z3 Sort

It is the building component of Z3 type system. **Sorts** in Z3 are equivalent to **data types** in most programming languages. Examples of a sort in Z3 include `Bool`, `Int` and `Real`.

Constant

It is a symbol that builds the first-order formula which we are trying to solve with Z3. A Z3 solution to the SMT problem will assign a value to this constant that satisfies the given formula.

Each constant in Z3 has its own type (sort), and the value assigned to it in the SMT solution is of the same sort as this constant. The following example declares two constant, namely `x` and `y`, of type `Int` and queries Z3 for a solution for the given constraints.

```
x = Int("x")
y = Int("x")
solve(x == 1, y == x + 1)

# model: x = 1, y = 2
```

A constant of any sort can be created with the following syntax:

```
x = Const("x", some_sort)
y = Const("y", IntSort())
```

Axiom

It is the constraint imposed on problem constants that needs to be satisfied by values assigned to these constants. In the example above, `x == 1` and `y == x + 1` are two axioms.

Any Z3 expression that can evaluate to the Z3 `Bool` sort can qualify as a Z3 axiom. For instance, `x < y + x + 2`, `y != 0` and `x <= y` are all Z3 axioms.

Logical Connectives

Z3 supports most commonly used logical connectives in first-order logic. It supports negation (not), conjunction (and), disjunction (or), implication and bi-implication (equivalence). The syntax for these connectives in Z3Py is given below.
Negation: `Not(some_axiom)`
Conjunction: `And(one_or_more_axioms)`
Disjunction: `Or(one_or_more_axioms)`
Implication: `Implies(first_axiom, second_axiom`
Bi-implication: `first_axiom == second_axiom`

Functions

Functions are the basic building blocks of the SMT formula. Every constant can be considered as a function which takes no arguments and returns this constant. Z3 functions are **total** that is they are defined for all the domain elements. Moreover, functions (and constants) in Z3 are called **uninterpreted**, that is they allow any interpretation (may be more than one) which is consistent with the imposed constraints. Which means there is no prior interpretation attached before solving the SMT problem. Therefore, we may use the terms uninterpreted constant and variable interchangeably.

Z3 functions map one or more sort (type) of the domain to a result sort.

Below is an example that illustrates uninterpreted functions and constants.

```
x = Int('x')
y = Int('y')
f = Function('f', IntSort(), IntSort())
solve(f(f(x)) == x, f(x) == y, x != y)

# model:
# x = 0, y = 1, f = [0 -> 1, 1 -> 0, else -> 1]
```

Data-types

Z3 provides a convenient way for declaring algebraic data-types (a kind of composite data-types), which is a sort that can constructed from other sorts or data-types.

Before going through an example, it is important to define two constructs in Z3 data-types: Constructors and accessors. With a **constructor**, different variants of the data-type can be created. Each of these variants may have its own typed attributes. An **accessor** is a function that can fetch these attributes stored within a data-type instance.

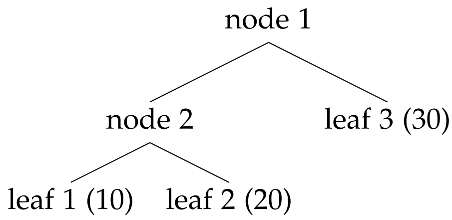
The following example demonstrates declaring and using data-types in Z3. We create a data-type representing a binary tree. The node of this tree may have two variants: Either a leaf with some value attached to it, or an inner node with left and right references two its left and right subtrees respectively.

```
Tree = Datatype("Tree")

Tree.declare("leaf", ("value", IntSort()))
Tree.declare("inner_node", ("left", Tree), ("right", Tree))

Tree = Tree.create()

leaf_constructor = Tree.leaf
inner_node_constructor = Tree.inner_node

left_accessor = Tree.left
right_accessor = Tree.right
value_accessor = Tree.value
```

A constructor is declared for each variant of the tree node. The leaf has an `Int` attribute representing the value it carries. The `inner_node` constructor has two arguments. Each attribute has its own accessor function.

Below on the left is an example of encoding the tree on the right using the example above.

```
leaf_1 = leaf_constructor(10)
leaf_2 = leaf_constructor(20)
leaf_3 = leaf_constructor(30)

node_2 = inner_node_constructor(leaf_1, leaf_2)
node_1 = inner_node_constructor(node_2, leaf_3)
```



Quantifiers

In addition to quantifier-free formulas, Z3 can also solve formulas involving quantifiers. Z3 uses different approaches to solve formulas with quantifiers. The only one which we are concerned with and we will be using in our type inference tool is the *pattern-based quantifier instantiation* approach. This approach works by annotating the quantified formula with some pattern annotations, and these formulas are only instantiated when these patterns are matched during the search context.

Z3 supports two kinds of quantifiers: *Universal* and *Existential* quantifiers.

Below is an example demonstrating using both kinds of quantifiers in Z3Py.

```
x = Int(’x’)
f = Function(’x’, IntSort(), IntSort())

ForAll(x, f(x) == x, patterns=[f(x)])

y = Int(’y’)
Exists(y, x + y == 2)
```

The above two axioms are equivalent to the below in first-order logic syntax:

$$\forall x \in \mathbb{Z}, \texttt{f(x) = x}$$
$$\exists y \in \mathbb{Z}, \texttt{x + y = 2}$$

2.3 Type System

A type system is the set of rules that assign the types to different constructs of the program, such that constructs which have the same type share common behavioral properties. The type system is useful in preventing the occurrences of certain types of errors before or during the program execution.

Each programming language defines the rules for its type system, and the language compilers and/or interpreters are built based on this type system. The process of verifying that the program satisfies the rules enforced by the language's type system is called *type checking*. There are two types of type checking: *static type checking* and *dynamic type checking*. Accordingly, programming languages are divided to *statically-typed* and *dynamically-typed* languages according to the type checking they perform.

2.3.1 Static Type and Dynamic Type Checking

Static type checking is done at compile time. Therefore, the types for every construct in the program must be available before compiling the code. Most statically-typed programming languages, like Java, enforce the programmer to declare the types for every construct. However, there are some languages, like Haskell, that employ type inference to statically deduce the types of the program constructs.

One benefit of static type checking is the early detection of type errors. Therefore, statically-typed languages are generally type-safer than dynamically-typed ones. Also, static typing contributes to the program readability and accordingly its maintainability.

On the other hand, **Dynamic type checking** is performed during runtime, where each object gets assigned to its type during the program execution. Programming languages that perform only dynamic type checking are classified as dynamically-typed languages. One of the advantages of dynamically-typed languages over the statically typed ones is that programs tend to be simpler and more flexible.

3 Type System

4 Type Inference

5 Evaluation

6 Future Work

7 Conclusion

List of Figures

List of Tables

Bibliography

[1] Eva Catarina Gomes Maia. “Inferência de tipos em Python.” MA thesis. the Netherlands: Universidade do Porto, 2010.

[2] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*. Microsoft Research, 2008.

[3] *mypy - Optional Static Typing for Python*. `http://mypy-lang.org`.