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

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 is 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 Python example:

```python
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 small 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 supported Python programs.

The type inference is based on a nominal static type system which is a subset from the type system of Python 3. It follows the semantics introduced in PEP 484 [1]. The type inference is intended to be integrated into Lyra and Nagini, 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. We make use of Satisfiability Modulo Theories (SMT) solving to assign inferred types to program constructs. Each sentence and expression in the program impose one or more new constraints, then the SMT solver is queried for a satisfying solution to these constraints. We will go through the details of the approach and the SMT encoding in later chapters.

This thesis is divided into six 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 MT encoding of the type system that we support. 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 constraints 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 dynamically-typed languages, specifically 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 their similarities and differences with 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

// TODO

Cartesian Product

// TODO

2.1.2 PEP 484

// TODO

2.1.3 Mypy [5]

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 and local variables 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:

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

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

What mypy intends to provide is closely related to one of the goals 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 by inferring the types of all these constructs in the program.

2.1.4 Inferência de tipos em Python [2]

The thesis [2] describes a static type system defined for a restricted version of RPython, which is a subset of Python, and presents static type inference ideas based on this type system. The work presented in [2] 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 calls to polymorphic and non-polymorphic functions, class definitions and class instantiation. However, the approach they take has a handful of limitations and is not applicable to real Python code. It failed to provide a type inference implementation for the ideas it proposed. Also, 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.

See the following example for illustration:

```
def add(x, y):
        return x + y
```

The addition operation in Python can only be applied on numeric type as arithmetic addition or on sequences as sequence concatenation. Therefore, the inferred types for the function arguments `x` and `y` should be one of the following possibilities:

- `x <: complex`, `y <: complex`, `x <: y`, `return == y`
- `x <: complex`, `y <: complex`, `y <: x`, `return == x`
- `x <: Seqence`, `y <: Sequence`, `x <: y`, `return == y`
- `x <: Seqence`, `y <: Sequence`, `y <: x`, `return == x`

Note that for simplicity, we consider `complex` to be a super type of all numeric types in Python. This is not precisely true with respect to the Python type system. However, this is acceptable because all numeric types are type-compatible with `complex` types, that is all numeric types can be used whenever `complex` is expected.

[2] does not give a way for handling the above constraints. It only states that the function arguments are inferred in a separate context without giving any insights into how the function body would affect the inferred types for the function arguments.

2.1.5 Starkiller

// TODO

2.1.6 PyType

// TODO

2.2 SMT Solving with Z3 [4]

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.

The SMT model is a mapping from the formula symbols to some values which satisfy the imposed constraints.

Z3 [4] is an efficient SMT solver, developed by Microsoft Research in 2007 with built-in support many theories like linear and nonlinear arithmetic, bit vectors, arrays, data-types, quantifiers, strings, etc.

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. [3]

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 constructs

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

Sorts

A sort is the building component of the 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`.

Constants

A constant is the symbol that builds the first-order formula which we are trying to solve with Z3. A Z3 model 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 model is of the same sort as this constant. 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.

The following example declares two constant, namely `x` and `y`, of type `Int` and queries Z3 for a solution to 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())
```

Axioms

An axiom 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 qualifies as a Z3 axiom. For instance, `x < y + x + 2`, `y != 0` and `x <= y` are all Z3 axioms.

Logical Connectives

Z3 supports the usual 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`

Uninterpreted 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 and similar to constants, functions in Z3 are also uninterpreted.

Z3 functions map values from one or more sort (type) of the domain to values from 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.

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 attributes carrying 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()
```

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.

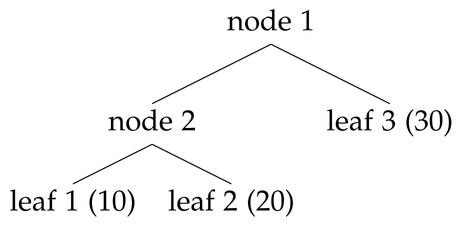
```
leaf_constructor = Tree.leaf
node_constructor = Tree.inner_node
```

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

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 = node_constructor(leaf_1, leaf_2)
node_1 = 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 syntactically matched during the search for a satisfying model for the formulas.

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 formulas 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 a set of rules that checks the assignment of types to different constructs of the program, such that constructs which have the same type share common behavioral properties. Type systems are 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.

Type systems can be classified as structural type systems, nominal type systems or a hybrid of both. We explain both classes of type systems shortly.

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 Nominal and Structural Type Systems

In a **nominal type system**, equivalence of types comes from an explicit declaration, and one type `a` is said to be subtype of another type `b` if and only if `a` is explicitly declared to be a subtype of `b`. Examples of languages that use a nominal type system include: C++, Java, C#, etc.

In a **structural type system**, equivalence of types comes from the structure of the types, such that a type `a` is equivalent to another type `b` if for every property in `a`, there exists and identical property in `b`. OCaml, for example, uses a structural type system.

A pseudo-code example to illustrate the difference between both type systems is given below:

```
class A {f() {return 1}}
class B {f() {return 1}}

A x = A()
B y = x
```

A nominal type system would reject the above program, because there is no explicit subtype relationship between classes `A` and `B`, so the variable `x` of type `A` cannot be assigned to the variable `y` of type `B`. However, a structural type system would allow it because the properties of the two classes are identical.

2.3.2 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. Also, static typing contributes to the program readability and, as consequence, to its maintainability.
The following Java example would be rejected at compile-time:

```
int x = 1
String y = "string"

x += y
```

On the other hand, **dynamic type checking** is performed during runtime, where each object gets assigned to its type during the program execution. One of the advantages of dynamically-typed languages over the statically typed ones is that programs tend to be simpler and more flexible.
The following Python example would be rejected at run time:

```
x = 1
y = "string"

x += y
```

2.3.3 Subtyping

Subtyping is a feature which exists in most programming paradigms. It allows a form of *substitution* principle, that is if a type A is a subtype of a type B, denoted by `A <: B`, then any expression if type A can be safely used in any context where a type B is expected. The type system of each programming language defines its own subtyping rules.
For example, in some programming languages, (e.g., Java), `int <: float`, so an integer type can be used in any context where a floating-point type is expected. Therefore, the following Java method is valid according to the subtyping rules of the Java type system.

```
float add(float x, float y) {
       return x + y;
}

float sum = add(1, 2.5F)
```

Method `add` is expecting its two arguments to be of `float` type, whereas an `int` is passed as its first argument.

2.3.4 Static Type System for Python 3

As mentioned before, our inference tool is based on a static type system that we have defined for Python 3. Since we intend to provide inference for statically-typed Python code, some dynamic features of Python have to be omitted in our type system. Below is a listing of the limitations we imposed on the dynamic nature of Python:

1. A variable should have a single type in the whole program.
2. Dictionaries map a set of keys of the same type to a set of values of the same type.
3. Elements in list or a set should have the same type.
4. Using reflective or introspective properties of Python is not allowed.
5. Modifying global variables using `global` keyword is not supported.
6. It is not possible to dynamically create and infer modules during runtime.
7. `exec` and `eval` commands are not supported.
8. Inheriting from built-in types is not supported.

Syntax

We present here the allowed syntax in our type system. Our tool supports all Python 3 syntax for expressions and statements except the following:

- Starred arguments in function definitions and function calls.
- Keyword arguments in function calls.
- `global` keyword.

Following the structure of the built-in Python `ast` module, we support the following collection of the Python 3 syntax:

```
stmt = FunctionDef | AsyncFunctionDef | ClassDef | Return | Delete
       | Assign | AugAssign | AnnAssign | For | AsyncFor | While | If
       | With | AsyncWith | Raise | Try | Assert | Import | ImportFrom
       | Expr | Pass | Break | Continue


expr = BoolOp | BinOp | UnaryOp | Lambda | IfExp | Dict | Set | ListComp
       | SetComp | DictComp | GeneratorExp | Await | Yield | YieldFrom
       | Compare | Call | Num | Str | FormattedValue | JoinedStr | Bytes
       | NameConstant | Ellipsis | Constant | Attribute | Subscript
       | Starred | Name | List | Tuple
```

The above listing follows the syntax for the class structure in the `ast` module. In order to comprehend the corresponding syntax in Python, one can read the documentation of the module [6].

Rules

Following the syntax of type hints introduced in PEP 484 [1], below is a listing of the types that we currently support:

```
t = None | object | bool | int | float | complex | string | bytes
       | Tuple[t*] | List[t] | Set[t] | Dict[$t_1$, $t_2$] | Callable[[t*], t]
       | Type[t] | T
```

Where `t*` represents a collection of types of arbitrary length and `T` represents an instance of a user-defined type.

Note that most other built-in types belong to user-defined types domain, since they are inferred as normal user-defined classes from our stub files.

The subtype relationships between the above types are defined by the following rules:

```
bool <: int
int <: float
float <: complex
complex <: string
$t_i$ <: $t_i^{'}$ : 1 <= i <= n $\rightarrow$ Tuple[$t_1$, .., $t_n$] <: Tuple[$t_1^{'}$, .., $t_n^{'}$]

t <: $t^{'}$ ^ $t_i^{'}$ <: $t_i$ : 1 <= i <= n $\rightarrow$
       Callable[[$t_1$, .., $t_n$], t] <: Callable[[$t_1^{'}$, .., $t_n^{'}$], $t^{'}$]

T <: U iff extends(T, U)
       where extends(a, b) is True if class a explicitly extends class b,
       and false otherwise

$\forall$t t <: object
```

Note that, and as mentioned earlier, there is not subtype relationship between `int` and `float` and between `float` and `complex` in the Python type system. However we claim this relationship for simplicity since these types are type-compatible with each other.

It is worth mentioning that the subtype relationship is both reflexive and transitive. Formally:

$$\forall x \ \texttt{subtype(x, x)}$$
$$\forall x, y, z \ \texttt{subtype(x, y)} \land \texttt{subtype(y,z)} \rightarrow \texttt{subtype(x,z)}$$

In the next chapter, we will explain the encoding of all the above rules in Z3.

3 Type System Encoding in Z3

4 Type Inference

5 Evaluation

6 Future Work

7 Conclusion

List of Figures

List of Tables

Bibliography

[1] Jukka Lehtosalo Guido van Rossum and Łukasz Langa. *PEP 484 - Type Hints*. 2014.

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

[3] Leonardo de Moura. *Experiments in Software Verification using SMT Solvers*. Microsoft Research. 2008.

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

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

[6] *Python AST docs*. `http://greentreesnakes.readthedocs.io`.