ETH Zürich
Department of Computer Science
ETH Zurich

GUC German University in Cairo
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:

```
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 is an example of encoding the tree in figure 2.1 using the tree data-type declared 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)
```

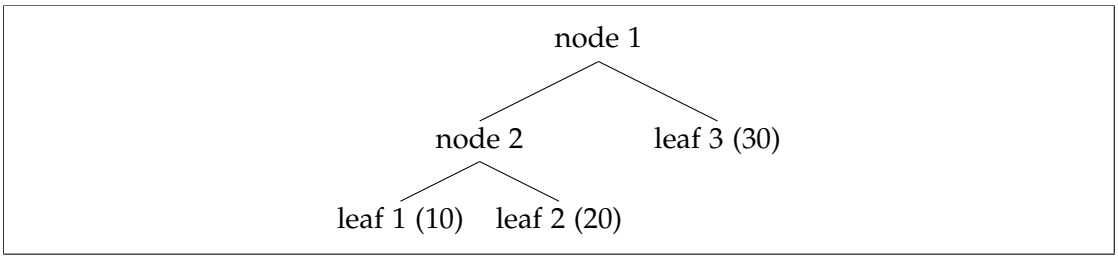

Figure 2.1: Tree Encoding with Z3 data-types

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. Stub files contain functions and classes that simulate built-in functionalities.

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 →  Tuple[t_1, .., t_n] <: Tuple[t_1', .., t_n']

t <: t' ∧ t_i' <: t_i : 1 <= i <= n →
       Callable[[t_1, .., t_n], t] <: Callable[[t_1', .., t_n'], t']

T <: U iff extends(T, U) ∨ ∃V: extends(T, V) ∧ V <: 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:

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

Although the above rules do not cover the whole Python type system, they are sufficient for most statically-typed Python programs. It remains to explain how we will build the type inference upon the above rules. In the next chapter, we will explain the encoding of all the above rules in Z3.

3 Type System Encoding in Z3

As mentioned earlier, we depend primarily on Z3 to provide a model in which a type is assigned to each construct in the Python program which satisfies the imposed constraints from the program semantics. In order to have such kind of constraints, the types of the type system of Python is simulated using Z3 data-types explained in the previous chapter, while the rules governing the subtype relationships in this type system are encoded with Z3 uninterpreted functions. We explain in this chapter the encoding of these types and rules in Z3.

3.1 Types Encoding

A python type is encoded in Z3 with a data-type declaration, which we call a *type sort*. From this data-type, multiple constructors are declared, each representing a corresponding type in Python type system.

3.1.1 Built-ins

Most built-in types are trivially declared with a *type sort* constructor. Below is a listing, written in Z3Py, of some types declarations from the *type sort* data-type.

```
type_sort = Datatype("type␣sort")

type_sort.declare("object")
type_sort.declare("type", ("instance", type_sort))
type_sort.declare("none")
type_sort.declare("complex")
type_sort.declare("float")
type_sort.declare("int")
type_sort.declare("bool")
type_sort.declare("str")
type_sort.declare("bytes")
type_sort.declare("list", ("list_type", type_sort))
type_sort.declare("set", ("set_type", type_sort))
type_sort.declare("dict", ("dict_key_type", type_sort), ("
    ↪ dict_value_type", type_sort))
```

```
...
type_sort = type_sort.create()
```

Notice that after defining all the constructors, we call the `create()` method to declare the actual data-type itself that we will be using in the inference. For this reason, all the types constructors have to be available before attempting to infer the program types.

3.1.2 Functions and Tuples

One thing that is not straightforward to encode is types that have a collection of generic types, like tuples or functions, since tuples and function arguments may have arbitrary length.

Unfortunately, Z3 does not currently support combining data-type declarations with arrays or sets. To workaround this limitation, we do pre-analysis for the whole input program which provides all possible lengths of tuples and the arguments of a function that appear in the program, then we manually declare a separate constructor for every possible length for a tuple and a function. Moreover, the user of the type inference himself has the ability to explicitly define the maximum number of function arguments and tuple elements without using that provided by the pre-analyzer.

Now having the maximum length of functions arguments and tuples elements, the types of functions and tuples can be encoded as below:

```
# Functions declaration
for cur_len in range(max_function_args + 1):
    accessors = []
    # create accessors for the argument types of the function
    for arg in range(cur_len):
            accessor = ("func_{}_arg_{}".format(cur_len, arg + 1),
                ↪ type_sort)
            accessors.append(accessor)
    # create accessor for the return type of the function
    accessors.append(("func_{}_return".format(cur_len), type_sort))
        ↪

    # declare type constructor for the function
    type_sort.declare("func_{}".format(cur_len), *accessors)

# Tuples declaration
for cur_len in range(max_tuple_length + 1):
    accessors = []
    # create accessors for the tuple
```

```
    for arg in range(cur_len):
            accessor = ("tuple_{}_arg_{}".format(cur_len, arg +
                ↪ 1), type_sort)
        accessors.append(accessor)

    # declare type constructor for the tuple
    type_sort.declare("tuple_{}".format(cur_len), *accessors)
```

3.1.3 User-defined Types

Similar to built-in types, each user-defined type has its own constructor declaration from the *type sort* data-type. However, since all the types constructors have to be available beforehand, we use the program pre-analyzer to provide a listing for all classes that are used in the whole program. Then a type constructor is created for each of these classes.

```
all_classes = some_pre_analysis_function(program)

for cls in all_classes:
    type_sort.declare("class_{}".format(cls))
```

Moreover, the pre-analyzer provides information about the methods and the attributes of these classes, then an uninterpreted constant is declared for each of these methods and attributes, and each class is mapped to the constants corresponding to its own attributes.

```
class_to_attrs = some_pre_analysis_function(program)
class_to_z3_consts = {}

for cls in class_to_attrs:
    attrs = class_to_attrs[cls]
    class_to_z3_consts[cls] = {}

    # Create Z3 constants for all attributes
    for attr in attrs:
        attribute = Const("class_{}_attr_{}".format(cls, attr),
            ↪  type_sort)
        class_to_z3_consts[cls][attr] = attribute
```

Now the type system defined in the previous chapter is wholly encoded in Z3. Note that, and as discussed earlier, other built-in types which are not mentioned here since they are inferred as user-defined types when they are encountered as class definitions in the stub files.

It is worth mentioning that all the built-in types could be encoded as class definitions, since every type in Python 3 is a class definition. However, classes inference has significant costs in terms of performance. Therefore, it was more convenient to encode the most common built-in types, like `int`, `float`, etc., separately.

3.2 Subtyping Rules Encoding

Having explained the encoding of the types in Z3, we explain here how the subtype relationships between these types are encoded. The subtype relationships discussed in the previous chapter are encoded using an uninterpreted function `subtype`, which takes two types, encoded in the *type sort* data-type, as its arguments and returns a `Bool` sort.

```
subtype = Function("subtype", type_sort, type_sort, BoolSort())
```

The encoding works by constructing an inheritance directed acyclic graph (DAG), such that each node in the DAG represents a type in the type system, and each edge from a node `x` to a node `y`, denoted by `edge(x,y)`, indicates that type `y` is a direct subtype of type `x`. Then using the information deduced from the DAG, the axioms for the subtyping are generated using the `subtype` function defined above according to the following conditions:

$$reachable(x,y) = \begin{cases} True & : x == y \vee edge(x,y) \vee \exists z : edge(x,z) \wedge reachable(z,y) \\ False & : otherwise \end{cases}$$

$$\forall x,y : reachable(x,y) == subtype(y,x)$$

Figure 3.1 shows a subgraph of the constructed DAG with a subset of the subtype relationships between built-in types.

3.2.1 Builtins with Generic Types

To generate the subtype axioms for types containing generics (e.g., lists, dicts and tuples), a universal quantification is performed over all possible types for the generics. The example below, in Z3Py syntax, shows the generated axiom for the subtype relationship between lists and `object`:

```
ForAll([x, y], subtype(List(x), y) == (y == object))
```

3.2.2 User-defined Types

The pre-analysis provides a mapping from every class to its super class(es). If it has no explicitly declared super classes, its mapped to `object` type, then the edges for every direct subtype relationship deduced from this mapping are added to the inheritance DAG.

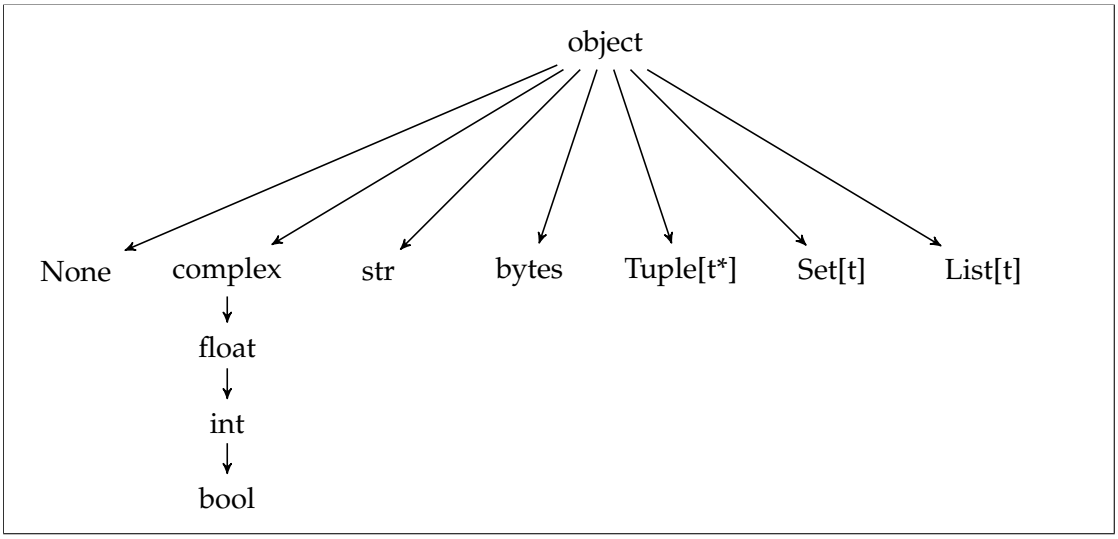

Figure 3.1: Subgraph of the constructed inheritance DAG

We will elaborate more in the next chapter on inheritance between user-defined classes. Specifically, we will talk about the method resolution order (MRO), which is the order in which methods and attributes are inherited in the presence of multiple inheritance, and how it is statically resolved. We will also discuss the variance relationships between overridden methods.

Now that the encoding of both types and subtype rules of the type system in Z3 is fully explained, we are ready to explain the design and the implementation of our type inference in the next chapter.

4 Type Inference

The previous chapter discusses the encoding of the type system in Z3. In this chapter, we explain how we build upon this encoding to provide a complete implementation for the static type inference.

4.1 Type Inference Design

We present here the main components of our type inference tool.

4.1.1 Abstract Syntax Tree (AST)

The AST of a program is a data structure which describes the structure of the source code, where each node in the tree represents a construct occurring in the program.

Figure 4.1 shows a visualization for the AST of the Python program below.

```
def f():
        return "Hello␣world!"
f()
```

Our type inference works by traversing the AST in a depth-first manner, and gathering constraints along the way.

4.1.2 Pre-analysis

Before we attempt to infer the types of the input Python program, some pre-analysis is needed to prepare the configurations of the type inference.

The pre-analyzer takes the AST of the input program and provides the following information:

- The maximum length of tuples that appear in the input program.
- The maximum length of function arguments that appear in the program.
- A mapping from all user-defined classes to their corresponding attributes and methods. It is important to differentiate between two kinds of these attributes: *class-level attributes* and *instance-level attributes*. Class-level attributes are those that are not specific to certain instances, and can be accessed with the class type itself, while instance-level attributes are those which are tied to the class instance during

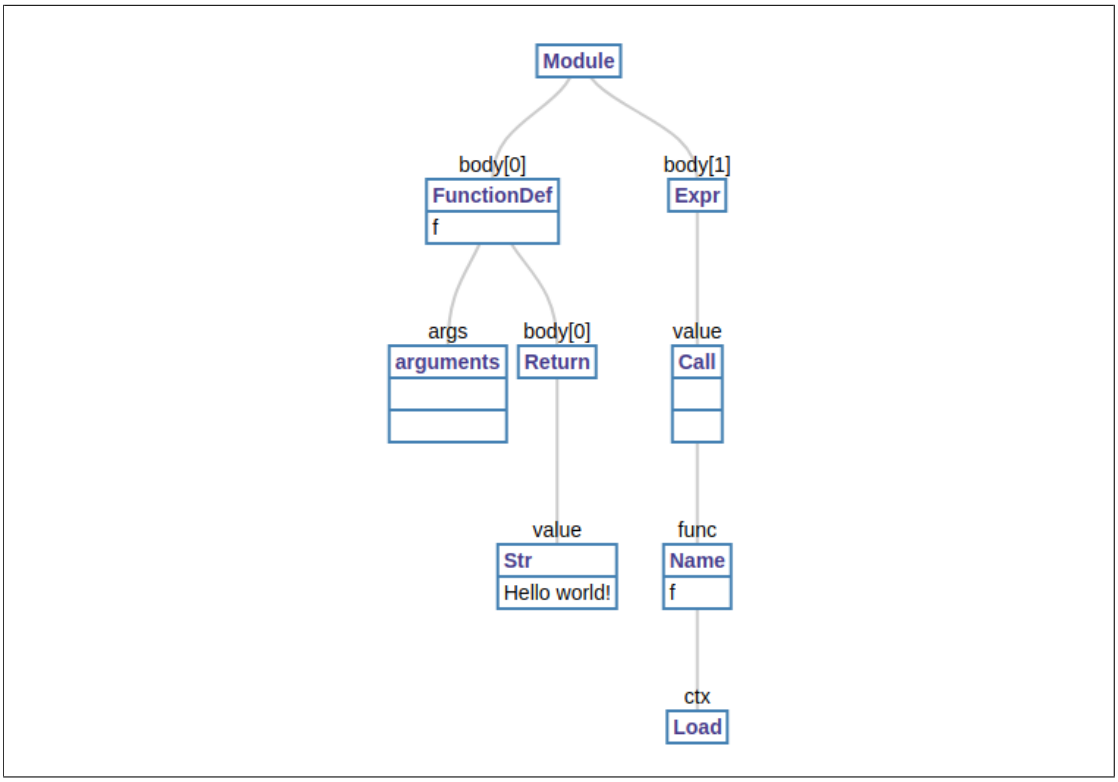

Figure 4.1: Abstract Syntax Tree (AST) for a Python program

its instantiation or later on. For instance, the class `A` in the following example has two attributes, namely `x` and `y`, where attribute `x` is a class-level attribute while `y` is an instance-level attribute.

```
class A:
        x = 1
        def __init__(self):
                self.y = 1

A.x # valid
A().x # valid
A().y # valid
A.y # invalid
```

Class-level attributes are detected by the pre-analyzer whenever it encounters an assignment statement in the top-level scope of the class in which the left-hand side is a normal variable.

Instance-level attributes are recognized whenever they are assigned by accessing the first argument in the class methods, like accessing `self` argument in the above example.

Note that class-level attributes represent a subset of the instance-level attributes, that is every instance of a certain class can access any class-level attribute. However, instance-level attributes cannot be accessed by the class type itself.

- A mapping from all user-defined classes to their base classes if they have any.
- The inheritance DAG which is used to generate the subtyping constraints discussed in the previous chapter.

In addition to the above information, the pre-analyzer also does the following preprocessing to the user-defined classes:

- Adds a default `__init__` method to classes which do not contain one.

  This default `__init__` method has the following source:

  ```
  def __init__(self):
          pass
  ```

- Propagates methods and attributes from base classes to their subclasses. The method resolution order which governs the order of this propagation is discussed later in this chapter.

4.1.3 Context Hierarchy

A context contains the information that a certain scope in the program holds. It contains a mapping from the variable names in this context to the Z3 variables representing their types, which are evaluated to the correct types after solving the SMT problem. Every context also has references to its children contexts (which are created inside the scope of this context) and a reference to its parent context.

Below is a listing of the constructs which create new contexts:

- `if` statements.
- `for` and `while` loops.
- Function definitions
- Class definitions
- List, set and dictionary comprehensions

Figure 4.2 shows a tree representing the context hierarchy for the Python program below.

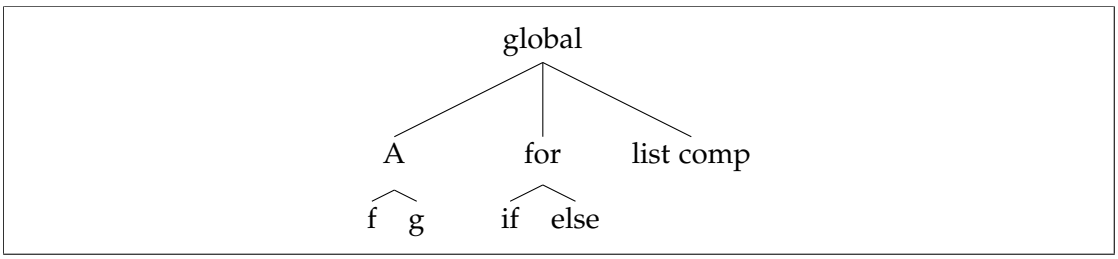

Figure 4.2: Context hierarchy for a Python program

```
x = [1, 2, 3]

class A:
        def f(self):
                pass
        def g(self):
                pass

for i in x:
        if True:
                pass
        else:
```

```
                pass

y = [i + 1 for i in x]
```

4.1.4 Z3 Solver

The Z3 solver is the main component in the type inference design. It is responsible for solving all the constraints imposed by the Python program semantics, or report that they are impossible to be satisfied.

We extend the default solver in Z3, such that during the instantiation of every solver instance, the following takes place:

- The pre-analysis defined above in processed.
- The *type sort* data-type is declared with all its constructors.
- The subtyping rules discussed in chapter 2 are initialized.

At the end of the program inference, this solver is queried for a solution to all the added constraints.

4.1.5 Import Handler

As the name suggests, the import handler is responsible for handling module importing during the type inference.

If the imported module is a built-in Python package, it retrieves the module from the corresponding stub file and infers the types of its contents, otherwise, it reads the imported module from the disk and infers its types in a separate context.

We will discuss later in this chapter different types of import statements in Python and how they are handled in our type inference.

4.1.6 Stubs Handler

As discussed earlier, **stubs** are files containing code which simulates built-in functionalities. A **stub function** is a function declaration which mocks some other function. The following function is a stub function which mocks the built-in function `len`:

```
def len(_: object) -> int:
        ...
```

Stubs enable the type inference to infer the types of programs using built-ins. The stubs handler is the module responsible for organizing the relevant stub files which are required by the program being inferred.

4.1.7 Inference Configuration

The user of the type inference has the ability to control the behavior of the type inference according to some pre-defined configurations. Each configuration is expected to have its gain and limitation. Some configurations, for instance, lead to a significant increase in the inference speed, yet at the cost of rejecting a larger set of correct programs. We will discuss the current possible configurations when we get to the rules related to these configurations.

4.1.8 Hard Constraints vs. Soft Constraints

An important addition to our type inference was introducing the ability to add soft constraints. **Hard constraints** are the constraints that **must** be satisfied by the program, such that if at least one hard constraint cannot be satisfied, the program is rejected and cannot type. On the other hand, **soft constraints** are those that are good to be satisfied, but they are not obligatory, such that a program violating some soft constraints is not rejected by the type inference. See the following example for illustration:

```
def f(x):
        y = [1, 2, 3]
        return y[x]
```

Here, the array `y` is indexed with variable `x`. Therefore, the type of `x` **must** be a subtype of `int`, so a program in which the type of `x` violates this constraint (e.g., having `x` as a float) would be rejected. The constraint in this case is a hard one. Another hard constraint is added in the assignment statement `y = [1, 2, 3]`, that the type of the array literal `[1, 2, 3]` is a subtype of the type of variable `y`. Moreover, a soft constraint is added that both the type of the array literal and the type of `y` are the same. Without this soft constraint, the type of `y` in the model given by Z3 might be an `object`, or any super type of the right-hand side (which is correct and sound, but not very accurate). So the purpose of the hard constraints is to provide a sound type inference, while that of the soft constraints is to increase its accuracy.

4.2 Type Inference Rules

Having explained the main components of the type inference, we are now ready to discuss the axioms added for every construct in the Python program.

4.2.1 Expressions Rules

An *expression* is any language construct that evaluates to a value. It can be a combination of one or more values, variables, operators and function calls. Every construct in Python that can be printed is an expression.

Below are some examples of Python expressions:

```
1 + 2 / 3
-a
[1.2, 2.0, b]
[[1.1, 2.5], c]
{(i, i * 2) for j in d for i in j}
2 & 3
d[f]
(g, 2.0, a)
"string"
(1 is 2) + 1
i[o]
```

We present here the axioms that are generated by every expression in Python.

List and Set Literals

As discussed earlier, the elements of a single list or a set have to be homogeneous, that is all these elements have to be of the same type.

So the elements type of a list (or a set) literal is the super type of all the list element.

Assuming a function `infer` which takes the AST node of the expression and returns its inferred type, the inference of list literals is implemented as follows:

```
def infer(node):
        ...
        if isinstance(node, ast.List):
                elements = node.elts
                elements_type = new_z3_constant()

                for element in elements:
                        current_element_type = infer(element)
                        add_axiom(subtype(current_element_type,
                            ↪ elements_type))

                return type_sort.list(elements_type)
        ...
```

For example, the type of the list literal `[1, 2.0, 3j]` is `List[complex]`, since we assume `complex` to be a super type of `int` and `float`.

The inference for sets is exactly the same after replacing the list z3 constructor with the set one.

Dictionary Literals

Similar to lists and sets, dictionaries should have homogeneous keys set and values set, and the type inferred for each of these sets is the super type of its elements. For example, the type inferred for the dictionary `{1: "string", 2: 3.6}` is `Dict[int, object]`, because `object` is the common super type between `str` and `float`.

Tuple Literals

The type of the tuple includes information about the types of all its elements. So to get the type of the tuple, we first infer the types of all its elements. For example, the type of the tuple `(1, "string", object())` is `Tuple[int, str, object]`

Binary Operations

Binary operations are the operations which combine two expressions, called operands, to produce a result expression. The inference of binary operations in Python is not very straightforward because the result type from every operation depends on different combinations of the types of the operands. We discuss here the axioms generated for every binary operation supported by Python.

Addition (+) The addition operation is either a numeric addition or a sequence concatenation. For the numeric addition, the type of the result is the super type of the types of the operands. This is encoded in Z3Py as follows:

```
Or(
        And(subtype(left, type_sort.complex), subtype(right, left),
            ↪  result == left),
        And(subtype(right, type_sort.complex), subtype(left, right)
            ↪ , result == right)
)
```

As for sequence concatenation, there are different cases to consider:

- Lists concatenation: The result type is a list in which the type of the elements is a super type of elements in both operands.
- String (or byte string) concatenation: The two operands should be of the same type.
- Tuple concatenation: The resulting type should be a tuple with the elements types of both operands concatenated.

For simplicity, we do not write the axioms for the sequence concatenation here.

Also, we do not currently consider operator overloading. However, this is not rejected by principle, these axioms can be extended to include classes which contain the method `__add__`.

The listing below shows examples for the addition operation and their inferred types:

```
1 + 1.0 # float
1j + 1.0 # complex
[1, 2, 3] + [4.0, 2] # List[float]
"a" + "b" # str
(1, "st") + (2.0, object()) # Tuple[int, str, float, object]
[1, 2, 3] + "a" # Invalid
```

Multiplication (*) Multiplication in Python, also without considering operator overloading, is either numeric multiplication or sequence repetition.

Similar to addition, the result type of numeric multiplication is the super type of the types of the operands. In case of sequence repetition, one of the operands must be a subtype of `int` and the other one should be the sequence. In all the sequences except tuples, the result type is the same as the sequence being multiplied. Ideally in case of tuples, the result is a tuple type with the argument types of the operand tuple repeated by the operand number. However, resolving the exact numeric value is impossible statically. Therefore, we consider the result of tuple multiplication to be the same type as the operand tuple. This is sound because no new types are introduced in the tuple arguments after multiplication, and as we will see shortly, any operation on tuple (e.g., indexing) does not care about the order of the types of the tuple elements. So the two tuple types `Tuple[int, str]` and `Tuple[int, str, int, str]` give the same output after applying any operation on the tuple.

An important thing to notice in both addition and multiplication is that applying these operations on two `bool` types results in an `int` type. So this needs special handling during the axioms generation..

```
3 * 4.0 # float
[1, 2, 3] * 3 # List[int]
(1, 2) * 2 # Tuple[int, int]
True * False # int
[1, 2] * 3.0 # invalid
```

Division (/) Division is only applicable on numeric types. The result is `complex` if at least one of the operands is of `complex` type, otherwise it is a float. Note that this is different from floor division (`//`).

The axioms generated by a division operation are given below:

```
And(
        types.subtype(left, type_sort.complex), types.subtype(right
            ↪ , type_sort.complex),
```

```
            Implies(Or(left == type_sort.complex, right == type_sort.
                ↪ complex), result == type_sort.complex),
            Implies(Not(Or(left == type_sort.complex, right ==
                ↪ type_sort.complex)), result == types.float)
    )
```

Other Arithmetic Operations (-, //, **, %) The remaining arithmetic operations (subtraction, floor division, exponentiation, modulo operator) exhibit similar behavior in terms of type inference. They can only be applied on numeric types and the result type is the super type of the types of the operands.

There is a single special case to consider in the modulo operator (%). In addition to giving the division remainder, it can also be used in string formating. Therefore, a disjunction is added to the axioms in this case that the left operand is a `str` type without restricting the right operand.

```
"A␣string␣which␣contains␣a␣number␣%i}" % 1
```

Bitwise Binary Operations (&, ^, |, «, ») Bitwise operations can only be applied on subtypes of `int` types (`int` and `bool`). The result is `int` in all cases except when we apply &, ^ or | on two `bool` types, where in such case the result is `bool`.

Unary Operations

Unary operations are the operations which are only applied on only one expression, called operand and gives a result expression. The supported unary operations in Python are unary - (minus), unary + (plus), unary ~ invert, and `not`.

For the plus and minus unary operations, the operand must be a subtype of `complex`, and the result is `int` if the type of the operand is `bool`, otherwise it is the same type as the operand. As for the unary invert, the operand must be a subtype of `int`, and the result is always of type `int`.

The `not` operation can by applied on any object and result in a `bool` type.

Boolean Operations

There are two boolean operators in Python: `and` and `or`.

Before explaining the inference for these operations, it is important to understand what a *truth value* of an object is.

In Python, every object can be tested for truth value, where each object can evaluate to `True` or `False` when used as test condition in `if` or `while` statements or in a boolean operation. The following values have a `False` truth value:

- `None`

- `False`
- Zero numeric value: 0, 0.0, 0j
- Any object which has `__len__` method which returns a zero.

Any other object has a `True` truth value.

The result type from a boolean operation is not simple to infer, since it totally depends on the values that the operands carry during runtime, and these values are impossible to be statically resolved. Specifically, the `and` operator keeps evaluating the operands until an operand with a `False` truth value is encountered. `or` operator does the opposite. See the following example for illustration:

```
a = function_which_returns_int()
b = function_which_returns_string()

x = a and b
y = a or b
```

If `a` has a zero value during runtime, then the type of `x` will be the same as `a`, which is `int`, otherwise it will be `str`. Conversely, `y` will have type `str` if `a` has a zero value, otherwise it will have an `int` type.

As mentioned, the values of `a` and `b` are impossible to be statically resolved. Therefore, the type of the result from a boolean operation is inferred to be the common super type between its operands.

So the result from `1 and 2.0` is `float` and from `1 and "str"` is `object`.

If (Conditional) Expression

With an if expression, conditional statements can be written as one statement which returns a value depending on whether a certain condition is true or false.

```
x = A if some_condition else B
```

This does the exact same thing as the following:

```
if some_condition:
        x = A
else:
        x = B
```

The inferred type for the if expression value is the common super type of the types of the true (A) and the false (B) values.

Subscripts

Subscript literals in Python are any literals which end in square brackets containing some expression. For example, all the following are subscript literals:

```
x[a]
x[a:b]
x[f()]
```

There are two kinds of subscripts in Python: *indexed* subscript and *sliced* subscript. A sliced subscript is that which contains one or two colons `:`. Any other subscript is an indexed subscript.

In python, any type which contains `__getitem__` method can be indexed or sliced. However for simplicity, we only consider built-in types here. We explain later in this chapter how we enhance the type inference to account for user-defined classes which implement the `__getitem__` method.

In our type system, only strings (and byte strings), lists, dictionaries and tuples support **indexing**:

- For strings, the index must be a subtype of `int` and the result is a `str`.
- For lists, the index is a subtype of `int` and the result is the type of the list elements.
- For tuples, the index must be a subtype of `int`, and the result is the common super type between all the tuple elements. This is because it is not possible to statically resolve the value of the index, so we cannot know which element the indexing is referring to. For this reason, the order (or the repetition) of the types of the tuple elements does not matter in indexing. For example, indexing both tuples of types `Tuple[int, float]` and `Tuple[int, float, int]` will give the same result type, `float`.
- For dictionaries, the index type should be the same as the inferred type of the dictionary keys, and the result type is the type of the dictionary values.

As for **sliced** subscripts, only strings, lists and tuples support slicing. The slicing one or more keys must be a subtype of `int`. The result type from slicing is the same as the sliced object. For the same reason stated above, slicing the tuple results in the same tuple type because resolving the slicing ends is impossible statically. This is also sound because any operation applied on the result from slicing a tuple should be compatible with every element type in the original tuple.

Comprehensions

Comprehensions are constructs which enable the programmer to create lists, sets or dictionaries in a natural and elegant way from any other iterable object.

The following set comprehension creates a set `x` which contains the square of all values in another iterable `y`.

```
x = {i * i for i in y}
```

This is equivalent to the following in a mathematical syntax:

```
x = {i * i | i ∈ y}
```

The expression `i in y` in the above example is called a *generator expression* while the expression `i * i` is called the comprehension element.

The inference for comprehensions works by creating a local context for the generator, and inferring the generator target (`i` in the example above) in this local context according to the generator iterable (`y`). Then by having the target inferred in the local context, the type of the comprehension element can be inferred by applying the expressions inference rules on it.

For the example above, assuming the type of `y` is inferred to be `List[int]`, then the type of `i` is inferred as `int` in the local context of the generator expression. Then the type of the set elements is inferred from the multiplication inference rules explained above: `int * int := int`. So the comprehension result will have a result of `Set[int]`.

Moreover, the generator expressions can be chained (nested). See the following example for illustration:

```
x = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

y = [i for j in x for i in j]
```

The array `y` contains the array `x` after flattening all its inner arrays (`[1, 2, 3, 4, 5, 6, 7, 8, 9]`). The inference for chained generators works by inferring the generator targets in the order they appear in the comprehension. So the type of `j` in the above example is inferred to be `List[int]` (because `x` is `List[List[int]]`), and the type of the second generator target `i` (which is also the comprehension element) is inferred to be `int`, and the type of `y` is then `List[int]`.

In addition to generators chaining, the comprehension itself can be nested.

```
x = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

y = [[i * 2 for i in j] for j in x]
```

The variable `y` is now a 2D array with the same dimensions as `x`, where each element in `y` is the double of the corresponding one in `x`.

The inference for comprehensions nesting works exactly the same as normal comprehensions: The comprehension `[[i * 2 for i in j]]` is treated at first as the larger comprehension element, and `j` is inferred to be `List[int]`. Now having the type of `j` in the local context of the comprehension, the type of the inner comprehension can be inferred with the same rule. Therefore, the inner comprehension will have the type `List[int]` and the outer one will have the type `List[List[int]]`, which is the type of `y`.

The inference for dictionary comprehension works the same way as lists and sets. The only difference is that the comprehension element is composed of a mapping instead of a single expression. The following example creates a dictionary comprehension which maps a every value in a list to its square.

```
{a: a * a for a in [1, 2, 3]}
```

After inferring the types in the generator expressions, the types for the dictionary keys and the values in the the comprehension element are inferred in the local context of the generators.

Variables Inference

The context contains a mapping from variable names to the Z3 constants that represent their types. The inference for variables in as simple as looking up the variable in this mapping.

The variables are stored in the context in the assignment statements, function definitions or class definitions. The inference for these constructs are explained in the upcoming sections.

4.2.2 Statements Axioms

A statement is a complete instruction that can be executed by the Python interpreter. It is a complete line of code that performs a certain action. Note that every expression is also a statement.

We explain here how the inference of different statements in our type system works.

Assignment Statements

There are several variations of the assignment statements in Python. However there is one important common axiom that is generated by all the variations, that is the right-hand side of the assignment must be a subtype of the left-hand side. In other words, the assignment target in any assignment statement is a super type of the assignment value.

```
target = value # value <: target
```

Simple Variable Assignment: The simplest variation of the assignment is the variable assignment:

```
variable = value
```

If the target variable is already existing in the context, the above subtyping axiom is added to the Z3 constant which this variable is mapped to in the context, otherwise, a new Z3 constant is created on which the subtyping axiom is applied.

For illustration, assume that the context initially has the following mapping:

```
{
        'x': x_z3_constant
}
```

Assume that these two assignment statements are encountered:

```
x = 1
y = "string"
```

Since variable `x` already exists in the context, no new Z3 constant is created for it, and the axiom is added on the Z3 constant it is mapped to `x_z3_constant`:

```
subtype(int, x_z3_constant)
```

As for the variable `y` not being in the context, a new Z3 constant is created for it and inserted to the context. The context now has the following contents:

```
{
        'x': x_z3_constant,
        'y': y_z3_constant
}
```

And the following axiom is also generated for the variable `y`:

```
subtype(str, y_z3_constant)
```

Tuple and List Assignment

```
x, y = 1, "str"
[a, b] = [1, 2]
```

The inference for this kind of assignment works in a similar manner to the variable assignment. The difference is that the target elements are checked one by one if they already exist in the context, and similarly, a new Z3 constant is created for the targets which do not exist in the context.

Since the lists in our type system have homogeneous elements, the types involved in list assignment must also have the same type. So the assignment `[a, b] = [1, "str"]` will lead both variables `a` and `b` to have type `object` (the common super type of `int` and `str`).

Subscript Assignment

```
a[x] = b
a[x:y:z] = b
```

Any class which implements `__setitem__` method is capable of performing subscript assignment. However for simplicity and similar to subscript expressions inference, we only consider built-in types here. We explain later how to extend the inference to support include user-defined classes as well.

Strings, byte strings and tuples are immutable objects, so they do not support subscript assignment. So whenever a subscript assignment is encountered, an axiom is generated that the subscripted object is not `str`, `bytes` or any `Tuple` type.

The last type of assignment statements is attribute assignment (e.g., `a.x = b`). We will discuss this type of assignment after explaining the inference for class definitions and the attribute access for class instances.

It is worth noticing the role of the soft constraints in the assignment inference. The result of the subtype axiom added for every assignment statement may make the inference not very accurate. For example, the assignment `x = 1` may lead `x` to have type `float` or even `object`, since `float` and `object` are super types of `int`. So a soft constraint is added that the type of `x` is `int`. See the following example:

```
x = 1
x = 2.5
```

The above code generates two hard constraints and two soft ones. The hard ones state that the type of x is a super type of both `int` and `float`. The first soft constraint assigns the type `int` to `x` and the second one assigns the type `float` to it. In the result model, only one soft constraint is satisfied (the second one), giving the type of `x` to be `float`, which satisfies the two hard constraints.

Body (Block of Statements) Type

A code block in Python is a group of Python statements which start with the same indentation tabs.

We assume that every non-expression statement in Python has a `None` type, except the `return` statement which has the type of the value it returns.

The type of a code block is the common super type of all non-`None` statements, or `None` if all the block statements have a type `None`.

See the following example for illustration:

```
def f(x, y):
        z = x + y
        return z
```

```
def g(x, y):
        x[0] = y
```

In the function `f`, the first statement sets the type of `z` to be the result from the addition axioms explained before between `x` and `y`. This assignment statement itself has a `None` type. The second statement has a type equal to the type value it returns: `z`. The type of the body of the function `f` is the super type of all the non-`None` statements in the body, which is the type of `z`.

In the function `g`, all the body statements have a `None` type, so the type of the function body is `None`.

Control-Flow Statements

A control-flow statement is the statement which results in a choice being made between two or more paths to follow. The inference for these kind of statements is tricky because the way the types in the context are affected depends on which path was taken, which is not always possible to be statically resolved. We present here how the axioms generated by the control-flow statements in Python, and how the context is affected by the paths choice.

4.2.3 Function Definitions Inference

4.2.4 Class Definitions Inference

4.2.5 Function Calls and Class Instantiation Inference

4.2.6 Attribute Access

4.2.7 Module Importing

4.3 Inference Output

4.3.1 Typed AST

4.3.2 Error Reporting

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`.