Department of Computer Science

ETH Zurich

Department of Computer Science and Engineering

German University in Cairo

Bachelor Thesis

# SMT-based 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

To teach is to touch a life forever. Therefore, my eternal gratitude goes to my supervisors, Dr. Caterina Urban and Marco Eilers. Thank you for your continuous support and patience, for believing in me, for always listening to and valuing my opinion and for everything you have taught me. I could not have done this work without your input and contributions.

I would also like to express my gratitude to Prof. Peter Müller for giving me the opportunity to be a part of his group and for trusting me to take the responsibility of this work.

A big thank you goes to my mother, father, brother and sister for their continuous encouragement and support. You are the reason I am who I am today.

I would also like to thank my dear friends. The ones who were always there when I needed encouragement, and the ones who held me when I was almost breaking down.

Last but not least, I am grateful to the programming methodology group at ETH Zurich for welcoming me for the past six months as the youngest member in the group.

I am utterly grateful to all of you!

# Abstract

Static types are useful in many programming languages. They can contribute to the type safety of the language by catching type errors in an early stage and contribute to the documentation of the programs written in it. Python is a dynamically typed language in which types are assigned to variables at runtime. Python is very popular nowadays and is used in different fields especially data science. This justifies the increasing attention towards performing static analysis for Python programs.

The aim of this thesis is to provide a static type inference for Python 3. We make use of SMT solving to perform the type inference by encoding the type system of the language in an SMT problem, collecting constraints from the given program and using Z3 SMT solver to provide a types model. We do whole program type inference such that the types model given by the SMT solution includes the types for all the variables that appear in the whole program.

We present a static type system for Python which is essential for defining the static type inference rules. We provide an encoding for this type system in the SMT solver Z3 and present a complete implementation of the type inference algorithm based on this encoding. The type inference currently supports many complicated features in Python like multiple inheritance and operator overloading. The proposed type inference is sound, but there remains some Python features which we do not support. The type inference proved to perform well on several programs and an open source project which uses different Python features.

# Contents


**Acknowledgments** **iii**

**Abstract** **iv**

**1 Introduction** **1**

**2 Background Information** **4**

2.1 Related Work . . . . . 4

2.1.1 Type Inference Algorithms . . . . . 4

2.1.2 PEP 484 [11] . . . . . 5

2.1.3 Mypy [9] . . . . . 5

2.1.4 Inferência de tipos em Python [5] . . . . . 6

2.1.5 Starkiller [12] . . . . . 7

2.2 SMT Solving with Z3 [8] . . . . . 7

2.2.1 Z3 constructs . . . . . 8

2.3 Type System . . . . . 12

2.3.1 Nominal and Structural Type Systems . . . . . 13

2.3.2 Static Type and Dynamic Type Checking . . . . . 13

2.3.3 Subtyping . . . . . 14

2.3.4 Static Type System for Python 3 . . . . . 15

**3 Type System Encoding in Z3** **18**

3.1 Types Encoding . . . . . 18

3.1.1 Built-ins . . . . . 18

3.1.2 Functions and Tuples . . . . . 19

3.1.3 User-defined Types . . . . . 20

3.2 Subtyping Rules Encoding . . . . . 21

3.2.1 Builtins with Generic Types . . . . . 23

3.2.2 User-defined Types . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**4 Type Inference 25**

4.1 Type Inference Design . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.1.1 Abstract Syntax Tree (AST) . . . . . . . . . . . . . . . . . . . . . . . 25
4.1.2 Pre-analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.1.3 Context Hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.1.4 Z3 Solver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.1.5 Import Handler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.1.6 Stubs Handler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.1.7 Annotations Resolver . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.1.8 Inference Configuration . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.1.9 Hard Constraints vs. Soft Constraints . . . . . . . . . . . . . . . . . 31
4.2 Type Inference Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2.1 Expressions Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2.2 Statements Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.2.3 Function Definition Inference . . . . . . . . . . . . . . . . . . . . . 48
4.2.4 Class Definition Inference . . . . . . . . . . . . . . . . . . . . . . . 52
4.2.5 Function Calls and Class Instantiation Inference . . . . . . . . . . 62
4.2.6 Attribute Access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.2.7 Module Importing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.2.8 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.2.9 Operator Overloading . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.3 Inference Output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.3.1 Typed AST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.3.2 Error Reporting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.4 Implementation Overview . . . . . . . . . . . . . . . . . . . . . . . . . 76

**5 Evaluation 78**

5.1 Experimentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.1.1 IMP Interpreter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.1.2 Functionality Specific Programs . . . . . . . . . . . . . . . . . . . . 79
5.2 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

**6 Conclusion and Future Work** **82**
6.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.2.1 Error Reporting . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.2.2 Modularity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
6.2.3 Infer Built-in Types with Class Definition Rules . . . . . . . . . . 83

# 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.", according to 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, 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. If this small mistake goes unnoticed at compile time, it will lead the program to raise an exception during runtime.

In this thesis, we present 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 in this thesis the details of the dynamic limitations imposed on the supported Python programs.

The goal of the thesis is to provide type inference rules that are sound in terms of static type checking, that is they reject any program which has any type error. It is worth mentioning that we perform whole program analysis, that is we do not provide local or modular type inference. The whole program must be available and given to the type inference. The reason for this is that the types of some constructs in the program might depend on other parts in the program. See the following example for illustration:

```python
class A:
  z = 1
class B:
  z = "string"

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

We have two classes `A` and `B`. Each of them defines a class attribute `z`. The type of `z` in class `A` is `int`, while the one in `B` has the type `str`. Function `f` takes a variable `x`, accesses its attribute `z` and adds `y` to it. From this function definition, we can know that the type of the parameter `x` is either `A` or `B`, the only classes which contain the attribute `z`. The type of `y` depends on the type assigned to `x`. If `x` is of type `A`, then `y` must be numeric type (e.g., `int`, `float`). If `x` is an instance of `B`, `y` must be `str`. Note that we cannot give polymorphic type to the function `f` by having the type of `x` to be `Union[A, B]` and the type of `y` to be `Union[int, str]`, because this allows a function call like `f(B(), 1)`, which will fail at runtime. It is also not correct to give one of these two type assignments to the function `f` because the other one might be required by other parts of the program. Now assume that this function `g` exists in the program:

```python
def g(x):
  f(x, "string")
```

The call to `f` inside `g` assigns the type `str` to `y`. Accordingly, the type of `x` is function `f` is inferred to be of type `B` not `A`. This clarifies the need to do whole program inference instead of a local one in order to have sound type inference rules.

The type inference is based on a nominal static type system which is a subset of the type system of Python 3. It follows the semantics introduced in PEP 484 [11]. 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 statement and expression in the program imposes 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 SMT encoding of the type system that we support. We also state the limitations of this type system.

In the fourth chapter, we describe the design and the implementation of the type inference algorithm in depth. We explain the components of the tool and the SMT constraints generated 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 the problems it faces with certain types of programs.

Finally in the sixth chapter, we review our work, discuss the conclusion of this thesis and we suggest the future work and research that is encouraged to be performed on our tool.

# 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 their limitations and their similarities and differences with our tool.

### 2.1.1 Type Inference Algorithms

There are two type inference algorithms primarily used nowadays: Hindley-Milner algorithm and the Cartesian Product algorithm.

#### Hindley-Milner

Hindley-Milner (HM) is a type system for the labmda calculus, defined by J. Roger Hindley and Robin Milner [4, 6]. They invented a type inference algorithm for this type system that is constraint-based. The HM algorithm is known for inferring the most general type for the programs providing parametric polymorphism without the need to write any type annotations. The HM algorithm is primarily used in functional programming languages like Haskell and ML.

The standard HM algorithm is not applicable to object oriented languages, since the concept of subtyping which is the basis of object oriented languages are is considered in the algorithm. Also inferring the most general type within a nominal type system is not always possible. For example, as discussed in the introduction, within a nominal type system, there may be many types given to the following function:

```
def f(x):
  return x.a
```

**Cartesian Product**

The Cartesian Product is an algorithm for the inference of the types of function calls presented by Agesen [1]. It is usually coupled with an iterative analysis for the inference of function bodies. It works by allocating type variables for all the program variables, then it builds a graph in which the nodes represent type variables and the edges represent constraints between these type variables. The types that a variable may achieve at runtime are determined by following the constraints which start from the type variable of this node. The Cartesian Product algorithm is useful in determining all the possible types that a variable may achieve at runtime. This is useful in compilation and program analysis, however it does not guarantee a sound type checking.

### 2.1.2 PEP 484 [11]

In Python, functions and variables can have type annotations to indicate the type that they will have at runtime. By default, these type annotations do not provide any static type checking. They are useful for documentation, static analyzers and static type checkers. PEP 484 defines the syntax that governs writing these type annotations. It introduces a module to provide these standard definitions. In this thesis, as we will discuss in the upcoming chapters, all the type representations and the types included in our type system are compatible with the definitions introduced in PEP 484. For example, the annotation representing a list whose elements have some type `T` is written as `List[T]`. For a complete guide of the syntax, we advice the reader to refer back to the complete proposal of PEP 484 [11].

### 2.1.3 Mypy [9]

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 function definitions and local variables to be fully type-annotated and cannot infer the types of 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
```

Mypy cannot infer the type of the function call `f()`, which is `str`. It says that the type of this call can be of *any* type.

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 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 [5]

The thesis [5] 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 [5] also describes a type inference implementation for Python expressions (like numbers, lists, dictionaries, binary and unary operations, etc.), assignment statements and conditional statements. They also give an idea about inferring the types of 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. They fail to provide a type inference implementation for the ideas they proposed. Also, they do not describe inferring the types of function parameters, which is a critical step in the inference of the types of function definitions and function calls. Accordingly, and similarly 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, for types of the function parameters `x` and `y`, we have 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.

[5] does not describe a way for handling the above constraints. they only state that the types of function parameters are inferred in a separate context without giving any insights into how the function body would affect the inferred types of the function parameters. In addition, there is no implementation available for their proposed type inference.

### 2.1.5 Starkiller [12]

Starkiller is a type inferencer and compiler for Python. It is based on the Cartesian Product algorithm to infer the types that every variable may have at runtime with some limitations. The goal of the type inference in Starkiller is not to provide any static type checking, but to give any possible type that any variable can achieve in order to be used in optimizing the compiler it presents. In addition, it is flow insensitive. Therefore, it does not provide sound type inference in terms of static type checking. For example, Starkiller's type inference and compiler will not reject the following program:

```
x = 1
x = "string"
y = x + 3
```

Because according to its type inference algorithm, `x` can achieve the type `int` at runtime (from the first assignment statement), and being flow insensitive, the order of the two assignment statements does not matter. So the addition operation in the third line is valid. Starkiller focuses on improving the runtime performance, but at the cost of the runtime safety.

## 2.2 SMT Solving with Z3 [8]

**Satisfiability Modulo Theories (SMT)** is a decision problem for first-order logic formulas. It is the problem to determine 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. An SMT model is a mapping from the formula symbols to some values which satisfy the imposed constraints.

**Z3 [8]** is an efficient SMT solver, developed by Microsoft Research 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 [7].

In our static type inference tool, we collect constrains from the Python program, and we depend primarily on Z3 to provide a types model that satisfies all these constraints.

### 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)
```

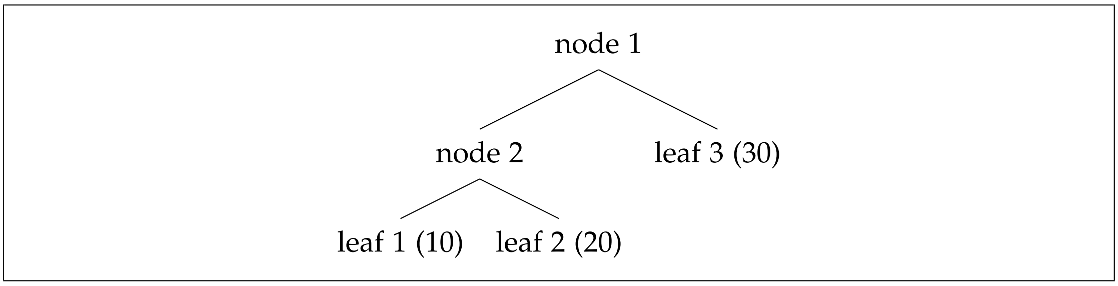


Figure 2.1: Tree Encoding with Z3 data-types

**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 kinds 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.

Python uses a structural type system. However in our type inference, we depend on a nominal type system that. We will define this type system in 2.3.4.

### 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 erroneous Java program 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 more flexible and simpler in terms of the syntax and the code size.

The following erroneous Python program 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 follows the *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 rejected by 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.

   Note that elements in sets, lists and dictionaries are allowed to have different runtime type. However during the type inference, we should be able to give them the same static type which allows any operation performed on any of these elements.

4. Inheriting from built-in types is not supported.

5. Attribute deletion is not supported because in a nominal static type system, all instances of a certain type are expected to have the same attribute everywhere in the program.

#### 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 [10].

### Rules

Following the syntax of type hints introduced in PEP 484 [11], 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[t1, t2] | 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 the 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 other functionalities like built-ins and low level code.

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

```
bool <: int
int <: float
float <: complex
complex <: string
```

$\mathtt{t}_i$ <: $\mathtt{t}_i'$ : 1 <= i <= n $\rightarrow$ Tuple[$\mathtt{t}_1$, .., $\mathtt{t}_n$] <: Tuple[$\mathtt{t}_1'$, .., $\mathtt{t}_n'$]

$$\texttt{t <: t}' \wedge \texttt{t}_i' \texttt{ <: t}_i \texttt{ : 1 <= i <= n} \rightarrow$$
$$\texttt{Callable[[t}_1\texttt{, .., t}_n\texttt{], t] <: Callable[[t}_1'\texttt{, .., t}_n'\texttt{], t}'\texttt{]}$$

$$\texttt{T <: U iff extends(T, U)} \vee \exists \texttt{V: extends(T, V)} \wedge \texttt{V <: U}$$

$$\forall \texttt{t t <: object}$$

Where `extends(a, b)` is True if class a explicitly inherits from class b, and False otherwise.

Note that as mentioned earlier, there is not subtype relationship between `int` and `float` and between `float` and `complex` in the Python type system. However, for simplicity, we claim this relationship since these types are type-compatible with each other, so that we do not need to define special rules for numeric types which will be the same as the subtyping rules defined for other types.

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

$$\forall x \; \texttt{x <: x}$$
$$\forall x, y, z \; \texttt{x <: y} \wedge \texttt{y <: z} \rightarrow \texttt{x <: z}$$

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 constraints imposed by the type system. In order to have such kind of constraints, the types of the type system are simulated using Z3 data-types, 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 *type sort*. From this data-type, multiple constructors are declared, each representing a corresponding type in the Python type system.

### 3.1.1 Built-ins

Most built-in types are simply declared with a *type sort* constructor. If a built-in type is composed of other one or more types (e.g., lists, dictionaries, etc.), accessor arguments representing these types are declared within the constructor declaration. 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. This introduces the need to perform some pre-analysis for the program being inferred to get knowledge of the types that are naturally not available before inspecting the source code. We will discuss these types in the upcoming sections.

### 3.1.2 Functions and Tuples

One thing that is not straightforward to encode is types that consist of a collection of other types, like tuples or functions, since tuples and function parameters may have arbitrary number of elements.

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

Now having the maximum length of functions parameters and tuples, the types of functions and tuples can be encoded as follows:

```
# 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)
```

A constructor is created for every length of functions and tuples, and accessor for each of their arguments is added to the corresponding constructor.

### 3.1.3 User-defined Types

Similarly, 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 before running the inference algorithm, we use the program pre-analyzer to provide a listing of all classes that are used in the whole program. Then a type constructor is created for each of these classes.

```
for cls in get_all_classes(program):
```

```
type_sort.declare("class_{}".format(cls))
```

Moreover, the pre-analyzer provides information about the methods and the attributes of these classes. 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 types of the type system defined in the previous chapter are encoded in Z3. Note that, and as discussed earlier, other built-in types 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 the way user-defined class are encoded, since every type in Python 3 is a class definition. However, the inference of the types of user-defined classes has significant costs in terms of performance. Therefore, it was more convenient to encode the most common built-in types, like `int`, `float`, etc., with separate constructors.

## 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 a directed acyclic graph (DAG) representing the inheritance, 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 (inherits from) 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:

$$subtype(x,y) = \begin{cases} True \ : x == y \lor edge(y,x) \lor \exists z : edge(y,z) \land subtype(x,z) \\ False : otherwise \end{cases}$$

Such that for every type, disjunctions for its super classes and sub classes are added to the Z3 solver. For example, the following disjunctions are added for the `int` type:

```
ForAll(x, subtype(x, int) ==
  Or(
    x == bool,
    x == int
  )
)

ForAll(x, subtype(int, x) ==
  Or(
    x == int,
    x == float,
    x == complex,
    x == object
  )
)
```

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

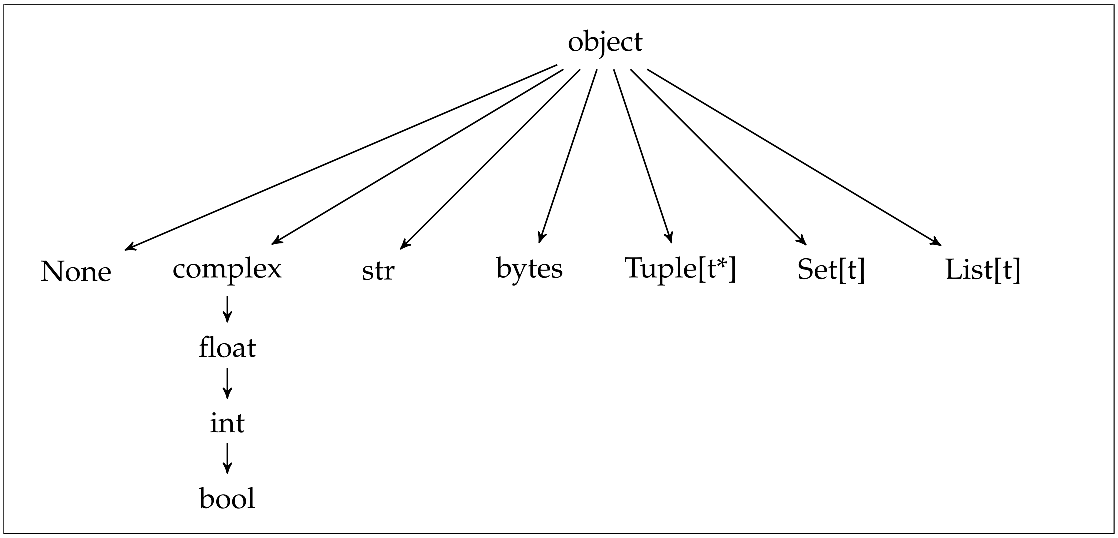


Figure 3.1: Subgraph of the constructed inheritance DAG

### 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) ==
  Or(
    y == List(x)
    y == object,
  )
)
```

### 3.2.2 User-defined Types

The pre-analysis provides a mapping from every class to its super class(es). If a class has no explicitly declared super classes, it is mapped to `object` type. Then the edges for every direct subtype relationship deduced from the pre-analysis are added to the 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 resolved in the presence of multiple inheritance, and how it is handled in our type inference. We will also discuss the variance relationships between the overriding and the 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. There are two phases for the type inference: constraints collection and constraints solving. In the phase of constraints collection, we traverse the input program and generate constraints from the program constructs according to the type system. In the phase of constraints solving, we give the collected constraints to the SMT solver. If these constraints are satisfiable, a model is given in which a type is assigned to each variable in the program. Otherwise, the SMT solver reports the reason for the unsatisfiability. We discuss the methodology for constraints collection and constraints solving in this chapter.

## 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 describing 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 defined by the type system along the way.

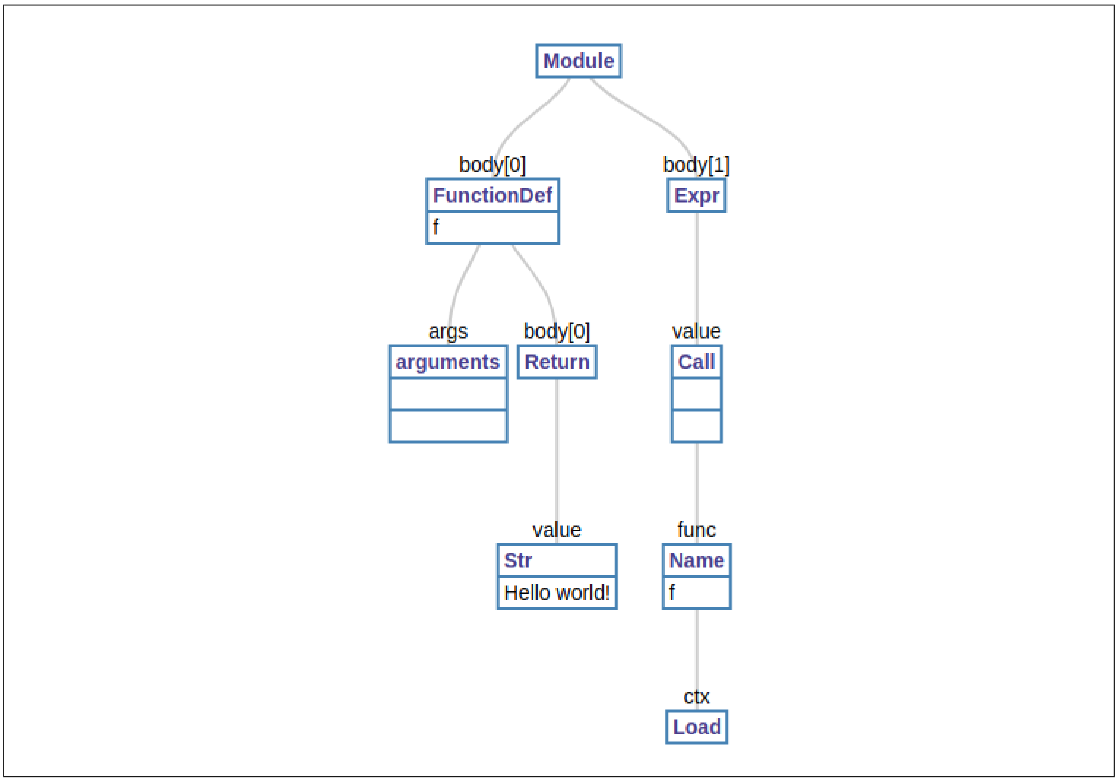


Figure 4.1: Abstract Syntax Tree (AST) for a Python program

### 4.1.2 Pre-analysis

Before we attempt to infer the types of the input Python program, some pre-analysis is needed to configure 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 number of function parameters 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 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 methods, like accessing `self` argument in the above example. The inference can be pre-configured to only detect instance-level attributes that are defined within the `__init__` method. This kind of configurations is explained in 4.1.8

Note that as far as the type inference is concerned, instance-level attributes also include class-level ones, 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 or to `object` if they do not explicitly inherit from other classes.
- The inheritance DAG which is used to generate the subtyping constraints discussed in the previous chapter.

In addition, the pre-analyzer does the following pre-processing to the user-defined classes:

- Adds a default `__init__` method to classes which neither contain nor inherit one.

  This default `__init__` method has the following form:

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

  Note that Python does not add this method for the classes which do not contain `__init__`. However as what will be clear shortly, we add it to make the inference of the class instantiation easier.

- 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 scope 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.

```
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]
```

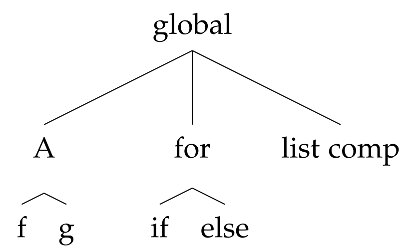


Figure 4.2: Context hierarchy for a Python program

### 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 reporting that they are unsatisfiable.

We extend the solver class of the Z3 Python interface, 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 (model) to all the added constraints.

### 4.1.5 Import Handler

As the name suggests, the import handler is responsible for 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 stores the types of its contents in the appropriate context, otherwise, it reads the imported module from the disk and runs the inference algorithm to infer its types in a separate context, but within the same SMT problem.

Later in this chapter, we will discuss how different types of import statements are handled in our type inference.

### 4.1.6 Stubs Handler

As discussed earlier, **stubs** are files containing code which simulates other functionalities, like built-ins or low level code. 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 which use built-ins. The stubs handler is the module responsible for providing the types of the relevant stubs which are required by the program being inferred.

### 4.1.7 Annotations Resolver

PEP 3107 [14] introduced the ability to add function type annotations, while PEP 484 [11] introduced the semantics of such annotations. The annotation resolver is responsible for translating a type annotation encountered in the program into the corresponding type-sort constructor.

For example, `type_sort.list(type_sort.int)` is the translation of the type annotation `List[int]`.

We will explain how these type annotations are useful in the type inference when we get to the inference of function definitions.

### 4.1.8 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 improvement in the inference performance, yet at the cost of rejecting a larger set of correct programs. We will discuss the current possible configurations when we get to the inference constraints related to these configurations.

### 4.1.9 Hard Constraints vs. Soft Constraints

An important addition to our type inference is 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 by the type inference. On the other hand, **soft constraints** are those that can be violated, but the SMT solver tries to maximize the number of satisfied soft constraints. So 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 list 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 list 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 object, or any too general super type of the right-hand side (which is correct and sound, but not precise). Since type annotations contribute to the program documentation and accordingly its maintainability, having a too general (unexpected) type annotation may be misleading and harmful, although it is correct in terms of the type system. So the purpose of the hard constraints is to provide a sound type inference, while that of the soft constraints is to increase its precision.

## 4.2 Type Inference Rules

Having explained the main components of the type inference, we are now ready to discuss the constraints 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 constraints 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 a super type of all the list (set) elements.

Assuming a function `infer` which takes the AST node of the expression and returns a Z3 uninterpreted constant which represents its type, the inference of the types 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_type = infer(element)
      add_constraint(subtype(current_type, elements_type))

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

For example, the type of the list literal `[1, 2.0, 3j]` can be `List[complex]` or even `List[object]`, since we assume `complex` to be a super type of `int` and `float` and every type is a subtype of `object`. With the soft constraint added, the type is restricted to `List[complex]`.

The inference of the types of sets is analogous to that of lists after replacing the list z3 constructor with the one for sets.

### Dictionary Literals

Similarly to lists and sets, dictionaries should have homogeneous keys set and values set, and the type inferred for each of these sets is a 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 a tuple includes information about the types of all its elements. So to get the type of a tuple, we first apply the expressions inference rules on all its elements. For example, the type of the tuple `(1, "string", object())` is `Tuple[int, str, object]`

### Binary Operations

Binary operations combine two expressions, called operands, to produce a result expression. The type inference for binary operations in Python is not straightforward because the result type of every operation depends on different combinations of the types of the operands. We discuss here the constraints 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, complex), subtype(right, left), result == left),
  And(subtype(right, complex), subtype(left, right), result == right)
)
```

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

- Lists concatenation: The result type is a list and the type of its elements is a super type of types of the elements in both operands.
- String (or byte string) concatenation: The two operands should be of the same type and so is the result.

- Tuple concatenation: The resulting type should be a tuple with the element types of both operands concatenated.

For brevity, we do not write the constraints for the sequence concatenation here.

Also, we do not consider here operator overloading. Section 4.2.9 explains how these constraints are extended to include classes which contain method `__add__`.

The listing below shows examples of different forms of 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, again without considering operator overloading, is either numeric multiplication or sequence repetition.

Similarly 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 a general Tuple type without specifying its elements. This is sound because as we will see shortly, any operation applied on this general tuple type (e.g., indexing) result in a sound general type like `object`.

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 constraints generation. For instance, the addition constraints are enhanced as follows:

```
Implies(
  And(left == bool, right == bool),
  result == int
```

```
)

Implies(
  Not(And(left == bool, right == bool)),
  Or(
    # Addition constraints explained above
  )
)

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 constraints generated by a division operation are given below:

```
And(
  types.subtype(left, complex), types.subtype(right, complex),
  Implies(Or(left == complex, right == complex), result == complex),
  Implies(Not(Or(left == complex, right == complex)), result == 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 constraints 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 applied on only one expression, called operand, and give 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 values: 0, 0.0, 0j
- Empty collections: [], , ()
- Any object which has `__len__` method which returns a zero or `__bool__` method which returns `False`.

Any other object has a `True` truth value.

The result from `and` operator is the first operand which has a `False` truth value. If all the operands have `True` values, then the result is the last operand. On the other hand, the result of `or` operator is the first operand which has a `True` truth value, or the last one if all have `False` truth values.

```
x = 1 and "str"
>>> x
"str"

x = 0 and "str"
>>> x
0

x = 1 or "str"
>>> x
1

x = 0 or "str"
>>> x
"str"
```

The constraints generated by a boolean operation are not simple to deduce since the result type 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 value zero at runtime, the type of `x` will be the same as `a`, `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 of a boolean operation is inferred to be the common super type between its operands. So the result of `1 and 2.0` is `float` and of `1 and "str"` is `object`.

### If (Conditional) Expression

An if expression is an expression whose value depends 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 type of the value of the if expression is the common super type of the types of the true (A) and the false (B) values.

### Subscripts

Subscript literals in Python 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 contains one or two colons `:`, e.g., `x[a:b]`. Any other subscript is an indexed subscript.

In Python, any type which contains the `__getitem__` method can be indexed or sliced. However, for simplicity, here we only consider built-in types. 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. As for the general tuple type `Tuple`, the result of indexing is constrained to be `object`.
- 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.

Disjunctions for all the above possibilities are added whenever an index subscript is encountered during the inference traversal.

As for **sliced** subscripts, only strings, lists and tuples support slicing. The slicing keys must be a subtype of `int`. For all types except tuples, the result type from slicing is the same as the sliced object. For tuples, the result is the general tuple type `Tuple` because resolving the slicing ends is impossible statically.

**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:

$$x = \{i * i \mid i \in y\}$$

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 constraining the generator target (i in the example above) in this local context according to the generator iterable (y). Then by having the Z3 constant corresponding to type of the target in the local context, the type of the comprehension element can be inferred by generating the expressions constraints explained above.

For the example above, assuming the type of y is constrained 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]. Assuming that x is the z3 constant representing the type of the iterable in the generator expression and y represents the type of the generator target, the following axion is generated during the inference of the generator expression:

```
And(
  subtype(x, iterable(y))
)
```

Now the local context of the comprehension is as follows:

```
context = {
  'i' : y
}
```

Then the multiplication constraints are added on the type of the variable i.

Note that, 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 list y contains the list x after flattening all its inner lists ([1, 2, 3, 4, 5, 6, ↪ 7, 8, 9]). The inference for chained generators works by applying the inference rules on the generator targets in the order they appear in the comprehension. So after solving all the constraints, 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 generator 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 first, and `j` is constrained to be `List[int]`. Now having the the z3 constant representing type of `j` in the local context of the comprehension, the type of the inner comprehension can be constrained with the same rule. Therefore after solving the SMT problem at the end, 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 consists of a mapping instead of a single expression. The following example creates a dictionary comprehension which maps every value in a list to its square.

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

After applying the inference rules on 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.

### Variable Inference

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

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

### 4.2.2 Statements Rules

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

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

#### Assignment Statements

There are several variations of the assignment statement in Python. However there is one important common constraint that is generated for 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 must be 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 already exists in the context, the above subtyping constraint is imposed on the Z3 constant which this variable is mapped to in the context, otherwise, a new Z3 constant is created on which the subtyping constraint 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 constraint 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 constraint 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 is similar 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 the 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 similarly to subscript expressions inference, we only consider built-in types here. We explain later how to extend the inference to 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, a constraint 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` ↪ ). The Z3 constant corresponding to the attribute `x` in the instance `a` is used as the type of the assignment target, and the normal assignment constraint is applied on it.

It is worth noticing the role of the soft constraints in the assignment inference. The result of the subtype constraint 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 starting with the same indentation tabs. We assume that every statement in Python has a `None` type, except the `return` statement which has the type of the value it returns. This makes it easier to infer the type of a block and also, as we will explain shortly, the return type of functions. The type of a code block is the common super type of all non-`None` (non `return`) 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 constraints 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 decides on two or more paths to follow during the program execution. 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 resolve statically. We present here what constraints are generated for the control-flow statements in Python, and how the context is affected by the paths choice.

There are three variants of control-flow statements in Python: `if` statements, `for` loops and `while` loops. However, the constraints generated for these statements are the same, except that an additional constraint for the loop variable is generated in the case of `for` loops.

Each control-flow statement has a required body block and an optional else block, and a separate local context is created for each. If one branch introduces a new variable, it will only be added to the parent context (which contains the control-flow statement) if the other branch also defines this variable. This prevents attempting to use a variable which might have not been defined. See the following example for illustration:

```
if some_condition:
  x = 1 # This is the first time x is defined
  print(x)
else:
  pass

print(x) # Invalid: x might have not been defined.
```

In the above example, using the variable `x` outside the local scope of the `if` body block is not allowed, since the flow might have taken the `else` path, where in this case variable x would not exist.

The type of a control-flow statement is a common super type between its branches. For example, the type of the following if statement is object, which is the super type of both int and str.

```
if some_condition:
  return 1
else:
  return "string"
```

### Deletion

The dynamic nature of Python allows the programmer to delete a value at runtime. The type inference supports all kinds of deletion except attribute deletion. Attribute deletion is not supported because in a nominal static type system, all instances of a certain type are expected to have the same attribute everywhere in the program.

The simplest kind of deletion is normal **variable deletion**. The deleted variable is simply removed from the context. Also, when a variable is deleted in one branch of a control-flow statement, it is also deleted from the context, because determining which branch was taken is not possible statically, so it is safer to assume that it was deleted.

For example, using the variable x after the following if statement is unsafe, because the then branch might have been taken, where the variable x is deleted.

```
x = 1
if some_condition:
  del x
else:
  pass

print(x) # Invalid: x might have been deleted.
```

The second variant of deletion is **subscript deletion**. Every subscriptable object, with the exception of immutable objects, can be used in subscript deletion. So constraint is generated when a subscript deletion is encountered that the subscripted object is mutable.

### 4.2.3 Function Definition Inference

The inference for function definitions is the main motivation behind adopting SMT to solve the type inference problem. For a program composed of only assignment statements, the type for every variable can be easily resolved by tracking the type of the value it is assigned to. However for function parameters, there is no origin for the type to track. The types of these parameters are solely based on how they are used in the function body and how this function is called; hence the idea of constraint solving evolved.

As stated earlier, a local context for the function is created. Each parameter in the function is mapped to a newly created Z3 constant in this local context, where this fresh constant is totally free from any constraints. Then the statement inference rules are applied on the body of the function. These rules impose constraints on the types of the function parameters. The return type of the function is the constrained type of the body block of the function. If a function has no `return` statement, then the return type is `None`. At the end, Z3 will assign types to the constants corresponding to the types of the arguments which satisfy all the imposed constraints. Let us illustrate the functions inference with the following example:

```
def f(x, y, z):
  a = x + y
  z += [1, 2]
  return z[a]
```

Initially, we create the context which contains the Z3 constants representing the types of function parameters.

```
context = {
  'x': x_z3_constant,
  'y': y_z3_constant,
  'z': z_z3_constant
}
```

The first statement `a = x + y` generates the addition constraint, and the type of variable `a` is stored in the context.

```
context = {
  'x': x_z3_constant,
```

```
  'y': y_z3_constant,
  'z': z_z3_constant,
  'a': addition_result
}
```

The second statement `z += [1, 2]` generates another addition constraint. According to the addition constraints explained before, lists can only be added to lists. Therefore, the type of `z` is constrained to be `List[T]` such that `T` is a new Z3 constant and `int <: T`. The last `return` statement generates the subscript constraint. From the subscript rules, we know that lists can only be indexed with subtypes of `int`. Putting this constraint together with the addition constraint on which the type of variable `a` depends, we restrict the type of `a` to be a subtype of `int`, and so are the types of `x` and `y`, and the return type is the same as the type of the elements of `z`, `T`. Therefore the type of the function `f` is inferred to be, according to the syntax of PEP 484 [11], `Callable[[A, B, ↪ List[T]], T]`, such that `A <: int`, `B <: int` and `int <: T`. Z3 will pick only one model for `A`, `B` and `T`. So, for instance, the types `Callable[[int, int, List[int]], ↪ int]`, `Callable[[bool, int, List[float]], float]` and `Callable[[bool, bool ↪ , List[complex]], complex]` are all valid types for `f`. Further constrains from function calls might narrow down these types and exclude some of them.

In order to increase the accuracy and to make the type inference more deterministic, we make use of soft constraints in all the contexts that give multiple possibilities because of the subtyping relationship. Specifically in this example, a soft constraint is added in the addition operation that the type of the addition result is the same as the two operands. So with the soft constraints being added, Z3 will assign the type `Callable[[ ↪ int, int, List[int]], int]`. to `f`.

### Default Arguments

A function in Python can have zero or more default values for the function parameters. If these default values exist, they are used to constrain the initial types of the function arguments. The type of every parameter which has a default value must be a super type of the type of this value. See the following example:

```
def f(x, y=1):
  return x + y
```

From the default value of the argument `y`, and assuming that `T` denotes the Z3 constant representing the type of `y` in the context, a new constraint is generated that `int <: T`. Additionally, a soft constraint is added here that both `T` and `int` are the same. So the type of the function `f` given by the Z3 model is `Callable[[int, int], int]`.

In order to accommodate the default arguments in the function type encoding in Z3, a new attribute, which has `Int` sort, is introduced in the function constructor of the *type sort* data-type declaration. This attribute represents the number of default arguments this function has. Recalling the function constructor declaration from chapter 3, the new accessor is added to the accessors array of the constructor.

```
accessors = [("func_{}_defaults_args".format(cur_len), IntSort())]
```

### Function Type Annotations

The programmer also has the ability to write type annotations for the function parameters as well as the return type. PEP 484 [11] introduced the semantics for writing the type annotations. For example, a list of integers will have the type annotation `List[int]`. By default, these type annotations are not type-checked. In our tool, these annotations are useful in reducing the number of generated constraints, and hence improving the performance of the type inference. When a parameter is annotated in a function definition, the indicated type is used instead of creating a new uninterpreted Z3 constant.

```
def f(x: int, y: int, z):
  ...
```

The above function will have an initial local context with the following contents:

```
context = {
  'x': int,
  'y': int,
  'z': z_z3_constant
}
```

**Parametric Polymorphism**

The dynamic structural typing of Python gives a polymorphic nature to its functions. Since all the types are evaluated at runtime and no types are attached to the function parameters or the return, a function can accept a value of any type as its arguments which allows the operations performed inside the function. See the following example for illustration:

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

a = add(1, 2) # type := int
b = add("a", "b") # type := str
c = add([1, 2], [3, 4]) # type = List[int]
```

The function `add` can accept different argument types, and the return type depends on which types are passed as the function arguments. One limitation of depending on SMT solving to infer the type of the function is that the solver picks only one model. So referring to the above example, the function `add` can only perform either numeric addition or sequence concatenation, but not both. Thus, the approach described so far prevents this kind of polymorphism, and the above program will be rejected. To work around this limitation, we introduced the ability to annotate the function with a generic type variable. The syntax for these type variables follow the syntax introduced in PEP 484 [11].

Re-writing the above example with generic type variables:

```
from typing import TypeVar, List

T = TypeVar("T")
U = TypeVar("U", [int, str, List[T]])

def add(x: U, y: U) -> U:
  return x + y
```

Now the function `add` can accept any type which is indicated in the possibilities of the type variable `U`, and it returns the type which is unified with this type variable during the function call. We will describe later in Section 4.2.5 how this unification algorithm works.

However, introducing generic type variables comes with its limitations. Functions containing generic type variables are handled separately from Z3. Accordingly, no Z3 constant is created for any function that contains a generic type variable, and any function that contains one must be fully type-annotated (i.e., there must be a type annotation for every argument and the return of the function). Another consequence of treating these functions separately from Z3 is that they cannot be used as a first class object, that is they cannot be passed as function arguments. The only supported way to use functions with generic type variables is via direct function calls.

### 4.2.4 Class Definition Inference

Classes provide a way for creating new types in Python. Each instance of any class has a set of attributes attached to it, which identify its state, behavior and properties. We explain here how the type inference works for defining new classes.

#### Pre-analysis

As mentioned in Section 4.1.2, the pre-analyzer provides the following information about class definitions prior to running the type inference algorithm:

- A mapping from classes to methods: `methods`
- A mapping from classes to instance-level data attributes: `instance_level_attrs`
- A mapping from classes to class-level attributes: `class_level_attrs`
- A mapping from classes to their base classes, or to `object` if they do not have any: `bases`

Then as described in Chapter 3, all the user-defined classes are encoded in the *type sort* data-type, and Z3 uninterpreted constants are created for every method and data attribute which will represent their corresponding types in the generated model, and each class is mapped to the Z3 constants representing its own attributes.

### Attribute and Method Inference

When a class definition statement is encountered, a local context is created for this class. Then the statements inside the class (its methods and attributes declarations) are also traversed depth-first starting from the class node itself. The type of each statement is then constrained according to the inference rules described for expressions, statements and functions. After the traversal of the class contents is done, the types added to the local context of the class are unified with the corresponding Z3 constants the class is mapped to, which represent the types of the attributes of this class. Let us discuss the following example for illustration:

```
class A:
  x = 1
  def __init__(self):
    self.a = 1
  def get_a(self):
    return self.a
```

The pre-analyzer gives the following mappings:

```
class_level_attrs = {
  'A': ['x', '__init__', 'get_a']
}
instance_level_attrs = {
  'A': ['x', 'a', '__init__', 'get_a']
}
methods = {
  'A': ['__init__', 'get_a']
}

bases = {
  'A': ['object']
}
```

And the mapping from the class to the Z3 constants is also created:

```
z3_consts = {
  'A': {
```

```
    'x': attr_x_z3_const,
    '__init__': attr_init_z3_const,
    'get_a': attr_get_a_z3_const,
    'a': attr_a_z3_const
  }
}
```

It is worth mentioning that the first argument in any instance method (handling static methods is discussed shortly), called the method receiver, is always automatically constrained to have the type of class itself. So `self` in the above examples will have the type `A`.

At the beginning, the local context of the class is empty. Then the class contents is traversed and the constraints are generated as described before. After the statement `x = 1` is encountered, attribute `x` is constrained to be a super type of `int`. In the function `__init__`, we constraint the type of the instance-level data attribute `self.a` to be a super type of `int`. The types of the methods `__init__` and `get_a` are constrained as described in Section 4.2.3. In the end, the local context of the class `A` looks as follows:

```
context = {
  'x': x_z3_constant,
  '__init__': init_z3_constant,
  'get_a': get_a_z3_consant
}
```

Then at the end of the class inference, the following unification constraints are added to the SMT problem to define the types of the class attributes:

```
context['x'] == z3_consts['A']['x']
context['__init__'] == z3_consts['A']['__init__']
context['get_a'] == z3_consts['A']['get_a']
```

Note that the type of the attribute `self.a` is not included in this unification, since it is already unified by the attribute assignment statement in the `__init__` method.

### Method Decorators

Decorators provide a mechanism to dynamically alter the functionality of a function. Decorators in general are not supported in our type inference, since they may dynamically alter type-related functionalities of the functions. Currently, the only supported decorators are `@staticmethod` and `@abstractmethod`.

The inference for the types of methods decorated with `@staticmethod` works like for instance methods, except that the first argument is not unified with the class instance. We also impose a restriction that static methods can only be invoked on class types; they cannot be called on class instances. The reason for such restriction is to provide more type flexibility by lifting the variance restrictions which are imposed on the inherited methods. The variance restrictions are to be discussed shortly. The inference of calling static methods will be explained when we get to method calls inference in Section 4.2.5.

As for `@abstractmethods`, the body type of the method is ignored, and any subclass is forced to override any abstract method that appears in any of its super classes.

### Inheritance

Every class definition can define zero or more classes to inherit from. Inheriting from a class explicitly defines a *subtype* relationship between the subclass and the parent class. All the methods which exist in the super classes and are not explicitly implemented in the subclass are inherited.

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


class B(A):
  pass


B().f()
```

Class `B` inherits from class `A` and does not implement the method `f`, so the method `f` is inherited by `B` from `A`. However, things get more complicated in the presence of multiple inheritance. See the following example:

```python
class A:
  def f(self):
    return 1

class B:
  def f(self):
    return "str"

class C(A, B):
  pass

x = C().f() # What is the value (and type) of ‘x‘?
```

Now determining which `f` function is being called is not trivial. This is where the method resolution order (MRO) becomes crucial. MRO is the order in which methods and attributes are resolved in the presence of multiple inheritance. Python 3 uses the *C3 linearization* algorithm to determine the MRO. We briefly explain how this algorithm works.

### C3 Linearization

The algorithm works by defining the linearization of every class. The linearization of a class is a list containing the class itself followed by a unique merge of the linearizations of the parents of this class (if any) and a list of these parents.

```
linearization[A] = [A] + merge(parents_linearizations, parents)
```

The merge function is responsible of merging several lists into one list. It works as follows:

- Select the first head of the lists which does not appear in the tail of any other list. Let us call this head the *good head*.
- Remove the selected item from all the lists where it appears as a head and add it to the output list.
- Repeat the above two steps until all the lists are empty

- If no head can be removed and the lists are not yet empty, then no consistent MRO is possible and the program is rejected.

For example, merging the lists `[B, A]` `[C, A]`, `[B, C]` will be performed as follows:

- Select `B` to be the good head and append it to the result list, since it does not appear in the tail of any list.
- Remove `B` from all the lists. Now the lists are `[A]`, `[C, A]` and `[C]`.
- Similarly, remove the head `C`. Now the lists are `[A]`, `[A]` and `[]`.
- Select `A`. Now all the lists are empty and the result of the merge is `B, C, A`.

Let us see a more concrete example:

```
class A:
  def f(self):
    return A

class B(A):
  pass

class C(A):
  def f(self):
    return C

class D(B, C):
  pass

x = D().f()
```

Method `f` in class `A` returns the type of the class `A` itself. Similarly, the one in `C` returns the type of class `C`. Now which of these is inherited by class `D`? Let us calculate the linearization of each class. Let function `L(x)` denote the linearization of class `x`.

```
L[A] = [A] + merge([object]) = [A, object]
L[B] = [B] + merge(L(A), [A])
     = [B] + merge([A, object], [A])
```

```
      = [B] + [A, object]
      = [B, A, object]
Similarly,
L[C] = [C, A, object]
L[D] = [D] + merge(L(B), L(C), [B, C])
      = [D] + merge([B, A, object], [C, A, object], [C, B])
      = [D] + [B, C, A, object]
      = [D, B, C, A, object]
```

Now to determine which method `f` is inherited by class `D`, we traverse the linearization of class `D` and stop at the first class we encounter which implements the method `f`. So class `D` inherits method `f` from class `C`, because it is the first class in the linearization of `D` which implements `f`, and the type of the variable `x` is inferred to be `Type[C]`.

### Overridden Methods Variance

Variance is a topic that usually comes with any nominal type system which defines subtype relationships between its types. There are four forms of variance: invariance, covariance, contra-variance and bi-variance. We will explain these forms in the context of functions. See the following classes structure for illustration:

```python
class A: ...
class B(A): ...
class C(B): ...
```

And the following function:

```python
def f(x: B): ...
```

Function `f` accepts an argument `x` of type `B`.

Functions with **invariant** functions parameters which are restricted to a certain type do not accept neither super types nor subtypes of these types. So an invariant `f` can only accept `B` instances, not `A` or `C`.

Functions with **bi-variant** parameters accept both super types and subtypes. So a bi-variant `f` will accept instances of `A`, `B` and `C`.

Functions with **covariant** parameters accept subtypes but not super types. So a covariant `f` accepts `B` and `C`, but not `A`.

Functions with **contra-variant** parameters accept super types but not subtypes. A contra-variant `f` accepts `A` and `B`, but not `C`.

Let us add some methods to the classes above:

```
class A:
  def f(self, x: int) -> int:
    return x

class B(A):
  def f(self, x):
    return x
```

Class `A` has a method `f` which accepts subtypes of `int` as its argument and returns an `int` type. Class `B` inherits from `A` and overrides the method `f`. Assume a function `foo` which accepts an instance of `A` as its argument.

```
def foo(a: A):
  x = a.f(1)
  # .. more statements which use ‘x‘ ..
```

Now `x` is expected to have the type `int` (the return type of method `f` in class `A`). Since `B <: A`, then `B` instances can be used whenever `A` instances are expected. So, `B` instances can be passed as the argument for `foo`. Therefore, the return type of method `f` in class `B` **must** also allow all the operations that are executed in the body of the function `foo`. So it must be a subtype of the return type of the overridden method `f` in class `A`. Therefore, the return type of overriding methods must be covariant with the return type of the overridden methods.

On the other hand, any argument passed to the method `f` in class `A` must be also compatible with the corresponding argument in class `B`. So the types of parameters in the overriding method in `B` **must** be super types of the corresponding ones in the overridden method. Therefore, the types of parameters of the overriding methods are contra-variant with the corresponding ones in the overridden methods.

One thing to consider is the use of default arguments in the presence of these variance constraints. Referring back to the example, any call to method `f` in class `A` takes into account that it might be a call to the method `f` in class `B` instead. So any call to `f` on an `A` instance must also be valid as a call to `f` on a `B` instance. Therefore, the number of required arguments which have no default value in the overriding method in the subclass must be less than or equal to the one in the overridden method, and the total number of arguments in the overriding method in the subclass must be greater than or equal to the total number of arguments in the super class. For example, assume the following method exists in class `A`:

```
def g(self, x, y=1):
  pass
```

Then the calls `a.g(1)` and `a.g(1, 2)` are both valid, where `a` is an instance of class `A`. These two calls must also be valid if they were invoked on any instance of a subclass of `A`. So if class `B` overrides method `g`, the maximum number of required arguments it can have is one, and the minimum number of total arguments it can have is two (ignoring the first `self` receiver argument).

Formally, if the number of required arguments and arguments with default values in the super class are $\mathtt{a}$ and $\mathtt{b}$ respectively, and those of the subclass are $\mathtt{a}'$ and $\mathtt{b}'$, then the following conditions must hold:

$$\mathtt{a}' \leq \mathtt{a}$$
$$\mathtt{a}' + \mathtt{b}' \geq \mathtt{a} + \mathtt{b}$$

In the presence of multiple inheritance, if a method exists in more than one super class, then the overriding methods must satisfy the variance conditions described above with the methods in all the super classes, because an instance of a subclass can be used where an instance of any of the super classes is expected. For example, if a class `D` extends both classes `B` and `C`, then an instance of `D` can be used where an instance of `B` or `C` is expected.

#### More on Multiple Inheritance

Supporting multiple inheritance comes with a lot of scenarios to consider. One of them is the call to `super()`. The function `super` returns the next class in the method resolution order. So a `super` call in a certain method can refer to many different classes, depending on from where this call is originating. See the following example:

```
class A:
  def f(self): ...
```

```
class B(A):
  def f(self):
    return super().f()

class C(A):
  def f(self):
    return super().f()

class D(B, C):
  def f(self):
    return super().f()
```

If asked which class super the method f in class B is referring to, the most intuitive answer is class A. However, this is not always the case. The super call might have originated from method f in class D, and as we just discussed, the super call refers to the next class in the MRO. Class D has the linearization [D, B, C, A]. So the super in D will refer to B, and that in B refers to C. This makes the inference for the super call very complicated, since the super-methods may have different number of arguments or may even not exist. Assume that the class C did not have the function f, then calling super from f in class B would still be fine since it may refer to f in class A, and such resolution is not always possible to resolve statically. Adding support for the super call is still under research.

Another tricky case that comes with multiple inheritance is to decide which data attributes are inherited by a subclass from the super classes. See the following example:

```
class A:
  def __init__(self):
    self.x = None
class B:
  def __init__(self):
    self.y = None

class C(A, B):
```

```
    pass
```

Class C inherits from both A and B, and according to the linearization of C, only the `__init__` method of A is inherited by C. So C has the attribute `self.x` only, and does not have the attribute `self.y` defined in class B. However, since `C <: B`, C is expected to have all the attributes of B, and any instance of C can be used where B is expected. So an attribute access on an instance of B, e.g., `B().y`, should also be applicable to all its subclasses, which does not apply in the example given above. This reduces the safety of the type system. For example, the following function is expecting an instance of B as its argument and it returns its `self.y` attribute. This function can be called with an instance of any subclass of class B.

```
def f(b: B):
  return b.y

f(B())
f(C())
```

The above example passes our static type checking, but fails at runtime.

### 4.2.5 Function Calls and Class Instantiation Inference

Having explained the inference rules for class and function definitions, we are now ready to discuss the rules for dealing with function calls and classes instantiation. A `Call` AST node can represent a call to a function (or a method) or an instantiation of a type. Therefore, disjunction of both possibilities are added to the SMT problem.

#### Function Calls

Whenever a `Call` node is encountered, the type of the called construct is inferred in the current context, and then the following constraints are generated:

- The number of arguments in a call must be greater than or equal to the number of required parameters in the called function, and less than or equal to the total number of parameters. Formally, if a denotes the number of arguments in the function call, b denotes the number of parameters of the called function and c denotes the number of default arguments in this function, then the following condition must hold:

$\texttt{a} \leq \texttt{b}$

$\texttt{b - a} \leq \texttt{c}$

- Every argument passed to the function call is a subtype of the corresponding parameter in the function type. Formally, for a function call with argument types $\texttt{[t}_1\texttt{, .., t}_a\texttt{]}$ to a function of type $\texttt{Callable[[t}'_1\texttt{, .., t}'_b\texttt{], t]}$, then th following relationship holds $\texttt{t}_i \texttt{ <: t}'_i \texttt{ : 1 <= i <= a}$.

- The resulting type from a function call is the inferred return type of the function.

### Class Instantiation

Instantiating a class is equivalent to a call to the `__init__` method of this class, and the type of the call expression is the instance of this class. A disjunction for every possible class is added, and Z3 picks the one which satisfies the constraints explained above. Considering all the classes during the instantiation may seem redundant and inefficient, however class instantiation may be encountered in situations where the type of instantiated object is completely unknown. See the following example for illustration:

```
class A: ...
class B: ...

def f(x):
  return x()
```

When the call `x()` is encountered, a disjunction of the following constraints added to the SMT solver:

- `x_type == Type[A]` $\wedge$ `f_return_type == A`

- `x_type == Type[B]` $\wedge$ `f_return_type == B`

- `x_type == Callable[[], T]` $\wedge$ `f_return_type == T`

Such that `T` can be any type, and Z3 picks only one.

In addition, when the type of the instantiated class is obvious, only one constraint for this class is added. So if the call in the above example is `A()` instead of `x()`, only the first constraint in the disjunction above is added.

### Calls to Functions with Generic Type Variables

As discussed earlier, function definitions containing generic type variables are treated separately from Z3. Accordingly, calls to these functions also have to be treated in a special way.

When a `TypeVar` declaration is encountered, it is stored in the annotation resolver. Additionally, one of two more information might be attached with every `TypeVar` declaration:

- The type variable possibilities, that is the possible types that the type unifying with this type variable can achieve. Example: `TypeVar("T", [int, str])`. Only `int` and `str` (and not their other subtypes) can unify with this type variable `T`.
- The upper bound of the type variable. This allows unification with a certain type and its subtypes. For example, the type variable `TypeVar("T", bound=int)` allows `int` and `bool` to unify with it.

When a call to a function containing a type variable annotation is encountered, a local dictionary is created which maps the generic type variable to the Z3 constant of the type unifying with it. Then, if a parameter or the return of the function has an annotation containing this type variable, it is unified with the type this type variable is mapped to. Let us see an example for illustration:

```
from typing import TypeVar

T = TypeVar('T', [int, str])

def f(x: T) -> List[T]: ...

a = f(2)
```

The function `f` takes an argument and returns a list of elements which have the same type as this argument. The argument must either be of type `int` or `str`. At first, the generic type variable `T` is saved in the annotation resolver. Then when the function call `f(2)` is reached during the type inference traversal, an empty dictionary for the type variable mapping is created. Now the unification algorithm starts. When we check the arguments, we begin to resolve the type annotation in the function definition. At the first argument `x: T`, it is the first time we encounter the type variable `T`, so a new Z3 constant is created and is added to the dictionary, then the constraint for the type variable possibilities is added to this constant:

```
Or(
  t_z3_const == int,
  t_z3_const == str
)
```

And the type variable `T` is added to the dictionary:

```
type_var_mapping = {
  'T': t_z3_const
}
```

Then the type of the call argument `2` is unified with this Z3 constant.

```
int == t_z3_const
```

For the return type, the annotation resolver attempts to recursively resolve the type of `List[T]`. When it reaches the generic type variable `T`, it sees that it already exists in the dictionary, so it returns this type from the dictionary, and the return type will be `List[t_z3_const]`. By solving all the above constraints together, the type of the function call evaluates to `List[int]`.

Similarly, a call `f("string")` will unify the type `str` with the generic type variable, and the resulting type will be `List[str]`.

### 4.2.6 Attribute Access

An attribute access is an attempt to access the contents of an object. The syntax of an attribute access in Python is `some_object.some_property`. In our type inference, the attribute access falls under three kinds of objects:

- Accessing attributes of user-defined classes.
- Accessing attributes of built-ins.
- Accessing contents of an imported module.

### User-defined Classes

For user-defined classes, two types of constraints are generated: constraints for the class types and for the class instances. As discussed earlier, the pre-analyzer provides mappings from the classes to their class-level and instance level-attributes. As for the constraints generated for the class types, disjunction of constraints are generated for all the classes that contain the accessed class-level attribute. For the class instances, similar disjunctions are generated but for instance-level attributes. See the following example:

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


class B:
  x = "str"
```

If we encounter the attribute access `a.x`, the following (simplified) disjunction is created:

```
Or(
  And(a == Type[A], a.x == int),
  And(a == Type[B], a.x == str),
  And(a == A, a.x == int),
  And(a == B, a.x == str)
)
```

Z3 picks a valid one in the end when solving all the imposed constraints. Note that disjunctions for instances are also added in the case of class-level attributes, because as mentioned before, a class-level attribute is also an instance-level attribute from the perspective of the type inference. An attribute access like `a.y` will only generate disjunctions for instances as follows:

```
Or(
  And(a == A, a.y == int)
)
```

### Built-ins

Since we handle built-in types separately from the classes inference, the attribute access for built-ins also has to be addressed in a special manner. Our inference is currently only able to handle built-in attribute access for instance methods calls. So an attribute access like `"string".upper()` is supported but the one like `"string".upper` is not.

We get types of built-in method calls from their stub files. A stub method of a built-in typically looks as follows:

```
def method(self: built_in_type, *args) -> return_type:
  ...
```

For example, the stub for `append` method in the `list` type looks as follows:

```
from typing import TypeVar

T = TypeVar('T')

def append(self: List[T], e: T) -> None:
  ...
```

And whenever we have an attribute-access call, like `x.y()`, disjunction of all the possible built-in types and the user-defined types which contain this called method is generated. Assume that we have the following class:

```
class A:
  def append(self, x):
    ...
```

Then when, for example, the attribute access `x.append(1)` is encountered, a disjunction of two constraints is generated, the first is that `x` is `List[T]`, where `int <: T` (since we are appending `int`), and the other is that `x` is of type `A`, and `int` is a subtype of the type of argument of the method `append` in class `A`.

```
Or(
  x == List[int],
  And(x == A, subtype(int, append_arg_x)
)
```

### Calling Static Methods

In our type inference, static methods can only be defined by using the `@staticmethod` decorator, and they can only be invoked on class types, not their instances. For every attribute-access call, a disjunction of possible static method calls is also added.

### 4.2.7 Module Importing

Importing a module is gaining access to the code in this module. Python provides several ways for importing modules, the most common way and the only one that we currently support is using the `import` statements. Another way for importing, which we do not support, is using functions like `importlib.import_module()` and `__import__()`.

There are two forms of import statements: `import` and `import-from`. In both cases, the import handler is responsible for the importing in our type inference. The handler searches for the imported file first in the stubs for the built-in libraries, then using the file system after resolving the path of the module. It then creates a new context in which it applies the inference rules on the contents of the imported module within the same SMT problem. What happens after the module inference depends on the type of the `import` statement:

`import module`

The context in which the module was inferred is added to the global context of the parent module. In the following example, the global context is initially empty.

```
import X
```

After the import statement, the context contains the context of the imported module.

```
context = {
  'X': module_x_context
}
```

So when the attribute access `X.y` is encountered, we know that `X` is a module context, so we infer the type of the expression `y` in the context of the module `X`.

Similarly to functions with generic type variables, treating this kind of `import` statements separately from Z3 disallows using the imported modules as a first-class object. The only way to use the imported module is by accessing its contents directly.

```
from module import ...
```

For this kind of `import`, its context is not saved into the context of the importing module, but a merging of both contexts takes place. For example:

```
from X import A, B
```

This will lead the global context to have the following contents:

```
context = {
  'A': A_z3_const,
  'B': B_z3_const
}
```

In both kinds of `import`, it is possible to have an alias for the imported module or objects.

```
import X as x
from X import A as a, B as b
```

In this case, the alias is added to the context instead of the actual name itself.

### 4.2.8 Interfaces

Our type system is a static nominal one. However, we support partial structural subtyping through interfaces. With an interface, structural properties, like iteration or hashing, can be added to user-defined classes. We support some of the interfaces defined in the `typing` module of Python. We support the following interfaces:

- `Sized`
- `Hashable`
- `Iterable[T]`

Interfaces are introduced as normal types in the type system and are encoded accordingly. For each interface, a corresponding magic method must be present in the class which implements this interface. A magic method is a special method which adds a certain property to a type. The name of magic methods usually begins with two underscores. Examples of magic methods are `__init__`, `__add__`, `__len__` and `__iter__`. If this magic method exists in a class definition, a subtype property is added between this class and the corresponding interface in the inheritance DAG introduced in Section 3.2.

### Sized

The magic method of the `Sized` interface is `__len__(self)`. So an edge between classes implementing this method and the `Sized` type is added to the inheritance DAG. Built-in types which internally implement this method are also connected to the `Sized` type in the DAG. The `Sized` interface is useful for restricting the `len` built-in function to only accept sized objects. See the following example:

```
class A:
  def __len__(self):
    pass

def get_size(x: Sized):
  return len(x)


size_of_a = get_size(A()) # type: int
size_of_list = get_size([1, 2, 3]) # type: int
```

### Hashable

The magic method of the `Hashable` interface is `__hash__(self)`. Similarly to `Sized`, subtyping relationships are added between the classes which implement this method and the `Hashable` interface in the DAG. The `Hashable` interface is useful for restricting the type of the keys in a dictionary and the type of elements in a set to be hashable.

**Iterable**

Similarly, the presence of `__iter__` method inside a class indicates that this class implements the `Iterable` interface. Iterable objects can act as iteration variables in `for` loops and in comprehensions. A user-defined type can be used as an iteration variable by implementing the `Iterable` interface. See the following example:

```
class A:
  def __iter__(self) -> List[int]: ...

def sum_elements(x):
  result = 0
  for i in x:
    result += i

  return result

sum_elements([1,2,3])
sum_elements(A())
```

The parameter `x` in the function `sum_elements` is inferred to have type `Iterable[int]`, since it is the common interface implemented by `List[int]` and `A`.

The encoding of `Iterable` interface is different from the other ones since it is composed of a generic type (e.g., `Iterable[int]`, `Iterable[str]`). In case of built-in types that are composed of generics (e.g., lists), a quantification is performed on the generic type in the subtyping constraint. Recall the lists subtyping constraints explained in Section 3.2.1, interfaces are included as follows:

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

For user-defined classes, the type of generic in the iterable is unified with the return type of the `__iter__` method which the class implements.

### 4.2.9 Operator Overloading

Similarly to interfaces, operator overloading adds structural functionalities to user-defined classes and is defined by implementing magic methods. However, it is not included into the interfaces because there are no equivalent definitions in the `typing` module for the classes which implement (overload) these functionalities. We currently support overloading these operations:

- Addition (+). Its magic method is `__add__(self, other)`.
- Multiplication (*). Its magic method is `__mul__(self, other)`.
- Bitwise or (|). Its magic method is `__or__(self, other)`.
- Bitwise xor (^). Its magic method is `__xor__(self, other)`.

Constraints of these operations explained in Section 4.2.1 are modified to include classes which implement the above methods. For example, the addition disjunction is augmented with the constraint that the left operand implements `__add__`, and the right operand is a subtype of the parameter in this implementation (the parameter `other` in the list above), and the result type is the return type of this method.

## 4.3 Inference Output

After all the type constraints are collected, the Z3 solver is queried to check the satisfiability of these constraints. If the constraints are satisfiable, then a model is generated in which a type is assigned to every Z3 constant in the SMT problem. These types are used to construct a typed abstract syntax tree, in which each variable has an annotation indicating its type. Otherwise, a list of the conflicting constraints are returned.

### 4.3.1 Typed AST

The typed AST is a normal AST in which all the function definitions and the targets of assignment statements are type-annotated.

PEP 526 [3] introduced a new syntax for annotating variables, that was integrated into Python starting from Python 3.6. This syntax only works with single assignment statements, such that tuple/list assignments or assignments with multiple targets are not supported.

```
x: List[int] = [] # valid

x: int, y: int = 1, 2 # invalid
```

Generating the typed AST works by storing a list of all the function definition and assignment nodes contained in every context. When the types model is generated, the context hierarchy is traversed, and the type of every node stored in the context is converted from a *type sort* instance back to a type annotation matching PEP 484 syntax. This annotation is then added to the AST node.

Optional unparsing of the typed AST can then take place to generate typed source code for the input program. For example, see the following input program:

```
"""Factorization. Fermat's, Pollard's methods"""


def factorize(n):
  factors = {}
  d = 2
  while n > 1:
    power = 0
    while n % d == 0:
      power += 1
      n //= d
    if power > 0:
      factors[d] = power
    d += 1
    if d * d > n:
      d = n
  return factors


def get_all_divisors(n):
```

```
  divisors = []
  d = 1
  while d * d <= n:
    if n % d == 0:
      divisors.append(d)
      if d * d != n:
        divisors.append(n // d)
    d += 1
  return sorted(divisors)

a = get_all_divisors(2)
```

The following is the generated typed program:

```
"""Factorization. Fermat's, Pollard's methods"""

def factorize(n: int) -> Dict[int, int]:
  factors: Dict[int, int] = {}
  d:int = 2
    while n > 1:
      power: int = 0
      while n % d == 0:
        power += 1
        n //= d
      if power > 0:
        factors[d]: int = power
      d += 1
      if d * d > n:
        d: int = n
  return factors


def get_all_divisors(n: int) -> List[int]:
  divisors: List[int] = []
  d: int = 1
```

```
    while d * d <= n:
      if n % d == 0:
        divisors.append(d)
        if d * d != n:
          divisors.append(n // d)
      d += 1
    return sorted(divisors)

a: List[int] = get_all_divisors(2)
```

### 4.3.2 Error Reporting

If it is impossible to satisfy all the generated hard constraints, the solver is not able to find a model, but gives a listing of the conflicting constraints, which is called *unsat core* in Z3.

Every constraint added to the Z3 solver is mapped to a certain message that describes what this constraint represents, and contains the line number in the code which generates this constraint. Then the messages of the constraints that appear in the *unsat core* are then returned to the programmer. For example:

```
x = 1.5
y = [1, 2, 3][x]
```

It is impossible to satisfy the constraints generated by the above program because arrays cannot be indexed with floats. So the unsat core would give that these two constraints are conflicting:

- Assignment in line 1
- Array indexing in line 2

A more advanced error reporting would try to give types to the rest of the program except the conflicting lines. However, this is not as easy as it seems because everything else in the program might depend on the lines generating the conflicting constraints. This advanced error reporting is not implemented yet, but its research is in progress.

## 4.4 Implementation Overview

We give here an overview of the implementation, and how the type inference components are pipelined. The following listing represents the life cycle of our type inference algorithm:

1. The path to the input program is given to the inference runner.
2. The inference runner parses the program and generates its AST.
3. The inference runner creates the Z3 Solver.
4. The solver creates the pre-analyzer which traverses the AST and returns the discussed analysis results.
5. The solver creates the stubs handler, which, given the pre-analysis, organizes and prepares all the relevant stubs.
6. The solver creates the *type sort* data-type, which in turns instantiates all the required constructors and subtype constraints according to the pre-analysis.
7. The solver creates and prepares the annotation resolver, which is used during the type inference to translate any encountered type annotation into the relevant *type sort* constructor.
8. The inference runner creates the global context for the program.
9. The inference runner then performs a depth-first traversal of the AST of the program and begins the generation of constraints for the program statements and expressions as explained in Section 4.2.
10. The Z3 constraints collected from the program semantics are given to the Z3 solver which tries to find a satisfying model.
11. If the constraints are satisfiable, the model is generated and construction of the typed AST takes place. Otherwise, the *unsat core* messages of the conflicting constraints are printed.

Figure 4.3 shows how the components of the type inference interact with each other.

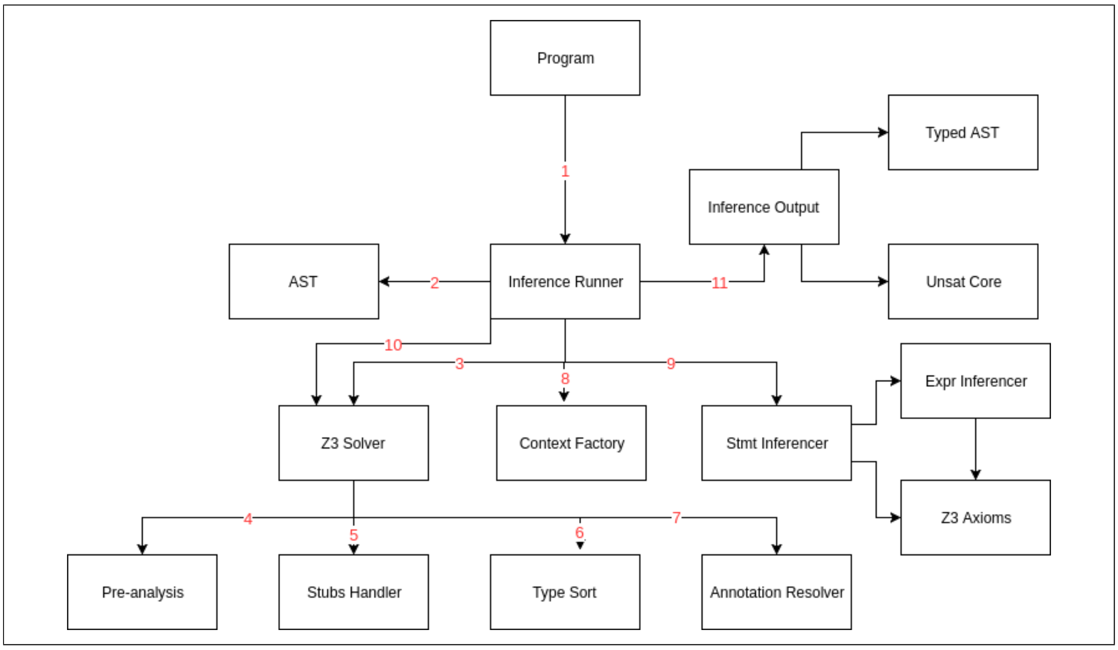


Figure 4.3: Components of the Type Inference

Having explained the static type system in Chapter 3 and the design and the rules of the type inference in this chapter, we present in the next chapter the experimentation we performed to evaluate the type inference, and we discuss the limitations of the type inference and the cases it cannot handle.

# 5 Evaluation

Having explained the encoding for the type system in Z3 and the type inference rules and the constraints generated by different Python constructs, we discuss here the experiments we have done with the tool and discuss the current type inference limitations.

## 5.1 Experimentation

We evaluated the type inference on a variety of programs, some of which focus on a single functionality (like multiple inheritance, function calls, etc.), while another is an open source project available on GitHub. However, the open source project that we use in our experimentation was not meant to be statically typed when it was written, so some parts of this project oppose the restrictions imposed by having a nominal static type system for Python. An example of these conflicts is the following:

```
class A:
  pass

class B(A):
  def f(self): ...

class C(A):
  def f(self): ...

def foo(x):
  x.f()

f(B())
f(C())
```

The type of the parameter `x` in function `foo` is inferred to be of type `A` (The super type of both `B` and `C`). However, class `A` does not implement the method `f`, so the call `x.f()` in the body of `foo` is invalid, although it will not fail at runtime. Accordingly, we had to modify some parts of the project we used to fit the limitations imposed by having a static type system. In the above example, we add an abstract method `f` to class `A`. After the types of these programs are inferred and the typed source code is generated, we run mypy [9] to statically check these types and verify that the inference produced a correct result.

### 5.1.1 IMP Interpreter

IMP [13] is a simple programming language developed in the 1970s. An interpreter for the language [2] was created by Jay Conrod as an example of building interpreters. This interpreter is an excellent testing material for our type inference for many reasons:

- It does not violate any of the restrictions discussed in 2.3.4.
- It extensively uses most of the Python patterns that we support, like inheritance, callable objects, operator overloading and using built-in libraries.
- It is composed of more than 1000 lines of code, which is comparable to most Python projects.

The inference for this project runs in 12 seconds on *8-core 3.6 GHz Intel Core i7* processor, which gives a prospect that the performance of the constraints solving is capable of handling a large portion of regular-sized projects.

### 5.1.2 Functionality Specific Programs

Some programs used for evaluation focus on specific functionalities supported by the type inference (e.g., multiple inheritance, inheritance variance relationships, built-ins, etc.). For example, the following program is used to test the method resolution order in the presence of multiple inheritance.

```
class A:
  def who_am_i(self):
    return A
```

```python
class B(A):
  pass


class C(A):
  def who_am_i(self):
    return C


class D(B, C):
  pass

d1 = D()
who = d1.who_am_i()

"""With the old MRO algorithm, the type of `who` would be Type[A],
because the naive DFS will reach A before C
With the C3 Linearization algorithm, C will appear first in the
search context, so the type of `who` is Type[C]
"""

# A := Type[A]
# B := Type[B]
# C := Type[C]
# D := Type[D]
# d1 := D
# who := Type[C]
```

To verify that the type inference is correct for these programs, we write the expected types for the variables in comments at the end of the program (e.g., `A := Type[A]`), then after the SMT problem is solved and the types model is given, we parse these special comments and compare the types given to each variable in the model against the specified ones in the comments.

## 5.2 Limitations

Section 2.3.4 states the restrictions imposed on the dynamic nature of Python by defining a static type inference for the language. We list here the limitations of the type inference which are allowed by the definition of the type system.

1. Using reflective or introspective properties of Python is not supported.
2. Modifying global variables using `global` keyword is not supported.
3. It is not possible to dynamically create and infer modules during runtime.
4. `exec` and `eval` commands are not supported.
5. It is not possible to define new function decorators, and using built-in decorators is currently limited to `@staticmethod` and `@abstractmethod` decorators.
6. Tuple assignments support only tuples not lists as assignment values. Similarly, list assignments support only lists as assignment values. For example, the following two assignments are not supported:

   ```
   a, b = [1, 2] # Cannot assign list to tuple
   [a, b] = 1, 2 # Cannot assign tuple to list
   ```

   This limitation is put to decrease the number of generated constraints when any of such kind of assignments is encountered, and hence increase the inference performance.
7. `super` calls are not supported due to the reasons discussed in Section 4.2.4.
8. Functions containing generic type variables and stub functions cannot be used as first class objects. They only support direct function calls.
9. Imported module via `import` statement cannot be used as first class objects.
10. Classes in the whole program must have unique names even if they belong to different modules.

# 6 Conclusion and Future Work

## 6.1 Conclusion

In this thesis, we presented a static type inference for Python based on SMT solving. We proved SMT to be capable of simulating a complex structure like the type system of Python and to be useful in solving a complicated system of constraints like the ones imposed by the type inference rules.

The goal of the thesis was to provide a static type inference which is sound in terms of the static type checking, but not necessarily complete. We took a new approach for tackling the type inference problem by encoding the type system in an SMT solver, doing whole program analysis, collecting constraints from the whole program and querying the SMT solver for the types that each variable is allowed to have at runtime which satisfy the imposed constraints.

Particularly, we defined a static type system for Python 3 which makes the static type inference possible. We also defined subtyping relationships between the types in this type system and presented the limitations imposed on the dynamic nature of Python due to having a static type system (Chapter 2).

In addition, we presented the encoding of this static type system in the SMT solver Z3 and how we make use of different Z3 constructs (e.g., data types and uninterpreted functions) to provide a concise encoding (Chapter 3).

We also provided type inference rules for all the Python constructs that we support and presented a complete implementation for the type inference algorithm (Chapter 4). We evaluated the type inference by giving it a variety of Python programs which focus on different Python features as well as one open source Python project (Chapter 5).

Although the type inference is still in an early stage and imposes some restrictions on the supported programs, being the first to tackle the type inference problem with SMT solving poses many interesting questions and opens multiple possibilities for future research. The work on this tool does not stop with this thesis. We will keep enlarging the subset of the Python language which we support and increase the completeness of the tool.

## 6.2 Future Work

The focus of this work was to provide a model for developing static type inference of dynamically typed languages. The goal was to define a static type system for Python and type inference rules that are sound in terms of static type checking. Having achieved this goal, our goal now is to make the type inference reliable for large-scale projects. We are faced with the challenge of supporting a larger subset of the Python programs. We present here some of the research and contributions to this tool that we plan to do in the future.

### 6.2.1 Error Reporting

Currently, if the generated constraints are unsatisfiable, the conflicting constraints are returned. However, the whole program is rejected in this case and no types are inferred. One possible future improvement is giving types to the program constructs except the ones contributing to the unsatisfiability of the constraints.

### 6.2.2 Modularity

The input program may be divided into independent components, such that a separate SMT problem is concerned with inferring the types of each component. So instead of having only one SMT solver and giving all the collected constraints to it, solving the SMT problem can be distributed among multiple solvers. This will lead to a significant improvement in the inference performance, and can also contribute to the error reporting discussed above. To have such modularity, advanced pre-analysis is required to determine the flow graph of the program, and divide the program components accordingly.

### 6.2.3 Infer Built-in Types with Class Definition Rules

The encoding presented in 3.1.1 treats the encoding of most commonly used built-in types separately from the encoding of class definitions, although every type in Python comes down to a class definition. The reason for this encoding is that the number of constraints generated for inferring the types of class definitions is huge compared to the ones generated by handling these built-in types separately. An interesting area of research in the future is applying the inference rules of the class definitions on these built-in types while maintaining fair performance. This will remove the need to handle a lot of special cases, and accordingly will make the type inference more natural with respect to the type system of Python. Also, this will lift some of the current type system restrictions, like inheriting from built-in types.

# List of Figures


2.1 Tree Encoding with Z3 data-types . . . . . . . . . . . . . . . . . . . . . . . 11

3.1 Subgraph of the constructed inheritance DAG . . . . . . . . . . . . . . . . 23

4.1 Abstract Syntax Tree (AST) for a Python program . . . . . . . . . . . . . 26
4.2 Context hierarchy for a Python program . . . . . . . . . . . . . . . . . . . 29
4.3 Components of the Type Inference . . . . . . . . . . . . . . . . . . . . . . . 77

# Bibliography


[1] O. Agesen, ed. *The Cartesian Product Algorithm*.

[2] J. Conrod. *IMP Interpreter*. URL: `https://github.com/jayconrod/imp-interpreter`.

[3] R. Gonzalez et al. *PEP 526 - Syntax for Variable Annotations*. 2016. URL: `https://www.python.org/dev/peps/pep-0526/`.

[4] J. R. Hindley. *The Principal Type-Scheme of an Object in Combinatory Logic*. 1969, 146: 29–60.

[5] E. Maia. "Inferência de tipos em Python." MA thesis. the Netherlands: Universidade do Porto, 2010.

[6] R. Milner. "A Theory of Type Polymorphism in Programming." In: *Journal of Computer and System Science (JCSS)* (1978).

[7] L. Moura. *Experiments in Software Verification using SMT Solvers*. Microsoft Research. 2008.

[8] L. Moura and N. Bjørner. *Z3: An Efficient SMT Solver*. Microsoft Research, 2008.

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

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

[11] G. Rossum, J. Lehtosalo, and Ł. Langa. *PEP 484 - Type Hints*. 2014. URL: `https://www.python.org/dev/peps/pep-0484/`.

[12] M. Salib. "Starkiller: A Static Type Inferencer and Compiler for Python." Massachusetts Institute of Technology, 2004.

[13] P.D. Stephens. "The IMP language and compiler." In: *The Computer Journal* ().

[14] C. Winter and T. Lownds. *PEP 3107 - Function Annotations*. 2006.