Comenius University, Bratislava

Faculty of Mathematics, Physics and Informatics

Type-Awareness in Dynamic Languages

Diploma Thesis

Dominik Kapišinský 2014

Comenius University, Bratislava

Faculty of Mathematics, Physics and Informatics

Type-Awareness in Dynamic Languages

Diploma Thesis

Program of Study: Informatics
Field of Study: 2508 Informatics
Department: Department of Informatics
Supervisor: RNDr. Tomáš Kulich, PhD.

Bratislava, 2014 Dominik Kapišinský

56333852

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Dominik Kapišinský
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický

Názov: Kontrola typov v dynamických jazykoch

Cieľ: Navrhnúť spôsob, ako do nie staticky typovaného jazyka zaviesť pre užívateľa nepovinnú informáciu o typoch a doménach hodnôt objektov, Na základe týchto informácií dedukovať možné miesta chýb, ktorých sa programátor dopustil.

Vedúci:	RNDr. Tomáš Kulich, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Dátum zadania:	21.11.2012
Dátum schválenia:	28.11.2012

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

....

študent vedúci práce

I hereby declare I wrote this thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis supervisor.

. . . .

I would like to thank my supervisor RNDr. Tomáš Kulich, PhD. for his great help, advices and supervising.

Abstract

The aim of the thesis is to propose and to implement a solution of a type inference in a certain dynamic language. It is not a hard problem to infer the types in a static language due its strict rules. On the other hand, almost each dynamic language is also a dynamically typed language. Such language is bounded only by a few rules regarding a type information and therefore a programmer is provided with a freedom that may be dangerous sometimes.

Our solution is a tool that is able to analyse a given source code of a Python program. The result is a list of the possible errors appearing in the input source code. Furthermore, our tool supports other functionalities suitable for a proper plugin supporting a development of the Python programs in a certain editor.

KEYWORDS: Dynamic programming languages, Python 3, Type inference

Abstrakt

Cieľom tejto práce je navrhnúť a implementovať riešenie dedukovania typov v určitom dynamickom jazyku. V prípade statických jazykov nie je ťažké dedukovať typy vďaka ich striktným pravidlám. Na druhej strane, takmer každý dynamický jazyk je taktiež dynamicky typovaným jazykom. Taký jazyk má len malé množstvo pravidiel týkajúcich sa typových informácií a teda ponúka programátorovi slobodu, čo môže byť niekedy nebezpečné.

Naším riešením je nástroj, ktorý je schopný analyzovať daný zdrojový kód Python programu. Výsledkom je zoznam možných výskytov chýb vo vstupnom zdrojovom kóde. Okrem toho náš nástroj podporuje aj ďalšiu funkcionalitu, ktorá je vhodná pre náležité rozšírenie podporujúce vývoj Python programov v určitom editore.

KĽÚČOVÉ SLOVÁ: dynamické programovacie jazyky, Python 3, dedukcia typov

List of Figures

Contents

Introduction

The dynamic programming languages register a sharp rise of users recently. These languages become more and more popular and therefore, there is a need of appropriate support for the programmers. Our main goal is to propose and to implement a solution that would be able to infer the types in the programs written in a certain dynamic language.

Majority of the dynamic languages is dynamically typed. That means, there are only a few rules regarding the dealing with types. With a great power comes a great responsibility. On the other hand, due to the strict rules of a usual static language it is relatively easy to provide a type control in IDE already during writing a source code. This control is very similar to that one executed during a compilation. A type error in the dynamically typed languages is detected by the period of runtime what presents a potential danger. For this reason, we want to come up with a solution that would be able to detect such a dangerous (smelling) code by the time of writing a program.

For our implementation, Python 3 was chosen from the family of the dynamic programming languages. Nowadays, Python is a very attractive language also for an educational activity. That means the increased number of beginners for whom a powerful code analysis would be helpful.

A relatively small Python community gives us a hope that the related solutions will provide us a space for improvements or even to come up with a completely new approach in this field.

In chapter 1, the basics of a type inference problem are explained and important features of Python and dynamic languages are introduced. Chapter 2 is dedicated to the related solutions and to their faults in type inference. Afterwards, we propose our ideas of a solution and the certain parts of the implementation. In the last chapter (chapter 4), our solution is compared with the existing projects already analysed in chapter 2.

1 Overview of the problem

Our thesis is focused on the dynamic languages. We take a closer look at the possibilities of how to infer a type of object in the programs written in the dynamic programming languages.

The dynamic programming languages are a class of the high-level programming languages and the difference between them and the static programming languages is significant, but the border between both of them is not strictly defined. The common features of a dynamic language include the eval function, the functional programming, the reflection, etc. Most of the dynamic languages are also dynamically typed. In the dynamic programming languages, many common programming behaviours are executed at the runtime which on the contrary the static programming languages perform during the compilation. One of the behaviours is a type system.

In programming languages, a type system is a set of rules that assign a type to the various constructions such as the variables, the expressions, the functions or the modules the computer programs are composed of. The main intention of a type system is to reduce the bugs of the computer programs by defining the interfaces among the different parts of the computer program. Afterwards, it is checked whether the parts are connected consistently. This checking may be executed statically (at compilation), dynamically (at runtime), or it can be performed as a combination of static and dynamic approaches (dynamically typed).

In the dynamic programming languages, an object is just some container, of which nothing is known, neither required. The variables in such languages are bound to this container (as shown in Figure 1.1).

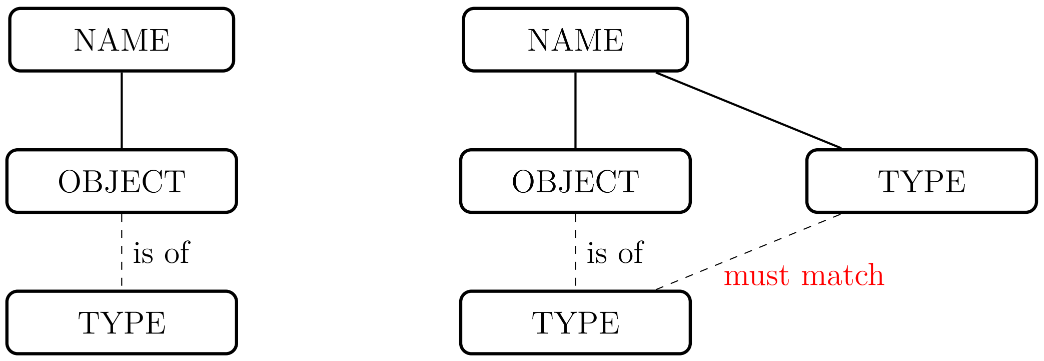

Figure 1.1: Dynamic vs. static languages

One of the main features is a dynamic addition of the attributes. This can be done in two ways:

- to add an attribute to an existing instance of an object
- to add an attribute directly to an object

As a consequence in the second case, all instances created from the altered object will consist also of the previously added attributes to the object.

1.1 How to infer a type of an object

When speaking about a type inference, there are mainly two possible ways how to realize it:

- a static approach
- a dynamic approach (to run the code and see)

In the static approach, the dynamics of a certain language is ignored and the specific rules are applied for inferencing the type. A dynamic approach means that a source code is executed and the result is analysed. At the first glance, it is clear that we do not want to ignore the dynamics when a dynamic programming language was chosen for some reasons. Why to give up the advantages provided by a certain dynamic language?

The idea of running the whole code does not seem safe and appropriate as well. What may happen? Let's consider a Python program that may cause the side effects, e.g. working with a database. In this case, each dynamic analysis, when willing to determine a type of an object, changes the entries in the database. That would be an unwanted surprise for a user of such a tool.

We would like to propose a better approach that would fit somewhere between the static and the dynamic approach. It should combine the safety of the static analysis and the power of the dynamic analysis. For the purposes of the type inference, it is possible to choose a technique of the extending syntax of a non-statically-typed language with an additional non-mandatory information on the object's types and domains. We have made a short review of the purposes in passing the additional information.

An extending of the syntax of a programming language is a significant intervention. Let's consider the following example in the Python-like syntax:

```
def fun(str x, bool y, int z):
    ...
    ...
```

This extended syntax of Python is not a valid Python program anymore. This change requires a lot of further work like an implementation of a new Python interpreter. A type inference would be more powerful thanks to this additional information, but the users of this tool would have to use also a customized Python interpreter when running this modified source code. Moreover, it seems like a step back to a static programming language. Therefore, this approach does not seem to be a proper one.

Learning from the previous not suitable approach, our second idea was to pass an extra information in the comments/docstrings. We avoided to a change in the syntax of the programming language that is a significant advantage in the comparison with the previous idea. The following example is again in the Python-like language:

```
class A:
 2        def bar(self):
 3            return ”bar”
 4  ”””
 5  type(obj=A)
 6  type(R=str)
 7  ”””
 8  def fun(obj):
 9      return obj.bar
10
11  a=A()
12  x=fun(a)
```

A format of the additional information is not important at this moment. On the line *5* there is an extra information for the function *fun*. This notation means that the argument *obj* is of a type *A*. The line *6* defines the return type of *fun*. This example shows a class definition *A* and the function definition *fun* and afterwards a creation of the instance *a* and passing it as an argument to the function *fun*. Without the ability to infer types meaningfully and with the additional information, it would be very risky to rely on such an approach. For instance, we would not know that variable *x* is of type *str* in the further possible source code. Therefore, there is a need to implement firstly the tool for the type inference and afterwards, to take into consideration the possibility of the additional information that would help us in some very complex programming constructions. These observations form the main aim of this thesis.

The goal is to suggest and to implement a powerful tool that will be able to infer a type of the objects in the complex programs written in the dynamically typed language.

1.2 Formulation of the problem

The main aim of our thesis is to propose a solution that would be able to infer a type of the objects and on the basis of the inference's results the possible sources of errors could be reported. Our vision is that such a tool could be a main part of a plugin for now undefined editor. This plugin would be a great support for the programmers using it in a combination with the given editor. However, this is not our part of work and therefore, we remain to focus on the tool of the type inference.

The idea of a suitable plugin solution and a communication between the plugin and the editor and much more is suggested and implemented in the thesis of my colleague, Bc. Ľubomír Žák [1]. Moreover, these two works create together a working solution of a plugin for the editor VIM. The name of the common solution is *pynfer*.

As a first step, we have to choose a certain dynamic language, the programs of which will be analysed by our tool. The next important decision is to choose the right programming language for the implementation of this tool. This choice is introduced in chapter 3. For now, we want to select a suitable dynamic language. After long a consideration of pros and cons, we have decided for a dynamic programming language Python. The reasons are as follows:

- Python is becoming more a more popular
- Python is used for the educative purposes
- Python's community is still relatively small
- a vision of a possibility to make a contribution in this area

The following section is devoted to the basics of Python.

1.3 Python

Python is a programming language [2] easy to learn and suitable for a widespread use. Moreover, a Python code is easy to read and write. Python is a very expressive language in comparison with e.g. Java or C++ . This means that a Python program usually has less lines of a source code than an equivalent program in the previously mentioned programming languages.

Python as a dynamically typed language can be used for programming in the object-oriented, procedural and functional style. Python is provided with a very complex standard library. This fact allows us to do such things as to download a file from Internet, to unpack a compressed archive file and to create a web server with just a few lines of code. In addition, there is also a lot of third party libraries providing a more powerful and sophisticated functionality than the standard library (NumPy, PLY) [3].

Python is a multi-platform language. In general, the same Python program can run on Windows and Unix systems such as Linux and Mac OS X. It is simply done by copying the files of the certain program/project to the target machine. On the other hand, it is possible to create the Python programs that use the specific platform functionality. However, this is rarely needed because the Python's standard library and most of the third-party libraries are fully and transparently multi-platform.

In the following section, we would like to explain the basics of the certain details in Python. Afterwards, the Python's modul AST is introduced.

1.3.1 Python scoping

In our thesis, scoping is a very important part. It is necessary to understand the rules of scoping in Python [4]. When a name is used in a program, Python creates, changes, or looks up the name in a namespace. Simply defined, the namespace is a place where the names live.

In Python, everything related to the names and the scope classification happens at an assignment time. The names in Python come into existence when a value is assigned firstly to the name. Besides, they must be assigned before they are used. Because the names are not declared before, Python uses the location of the assignment of a name to bind it to a particular namespace. That means that the place where the value is assigned to the name, determines the namespace for the name, called the scope of the visibility.

For instance, functions add an additional namespace to the programs to minimize the potential of the collisions among variables of the same name. The default rule says: all names assigned inside a function are associated with that function's namespace, and no other. The meaning is:

- a name assigned inside a definition of a function is seen by the code within that definition. It is not possible to refer to such a name from an outside function.
- a name assigned inside a definition of a function can not collide with the variables outside the definition, even if the same names are used elsewhere. A name X assigned outside the definition is a completely different variable from a name X assigned inside the definition.

When writing about the name's assignments, the variables may be assigned in three different places that correspond to three different scopes:

- If a variable is assigned inside a definition of a function, it is local to that function.
- If a variable is assigned in an enclosing definition of a function, it is non-local to the nested functions.
- If a variable is assigned outside all definitions, it is global to the entire file.

This approach is called a lexical scoping. The reason is that the variable scopes are determined entirely by the locations of the variables in the source code, not by the function calls.

In general, the Python's name-resolution scheme is called the *LEGB rule* (named after the scope names):

- The name assignments create or change the local names by default.
- The name references search at most four scopes: local (L), then the enclosing functions if any (E), then global (G) and the built-in scope (B). Python stops at the first place where the name is found (Figure 1.2). If the name is not found during this search, an error is reported.
- The name declared in the global and non-local statements maps the assigned name to the enclosing module and the function scopes, respectively (the local scope is the same as the global scope).

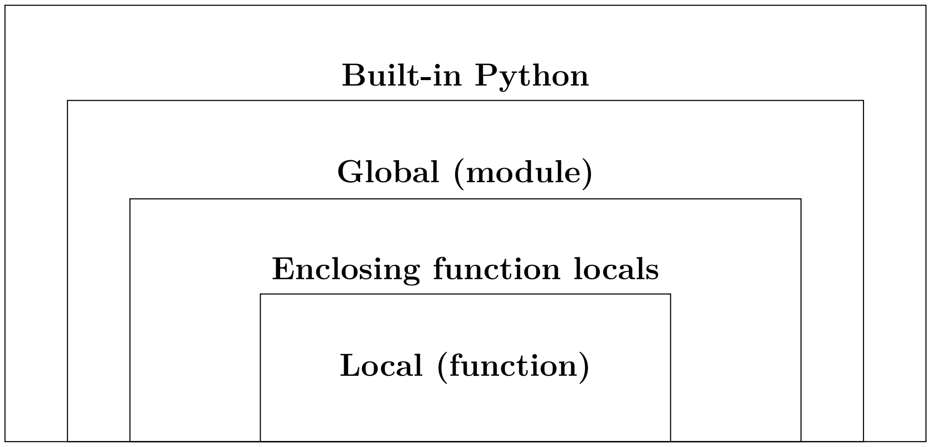

Figure 1.2: Encapsulating of the scopes - Built-in Python (The names preassigned in the built-in names module: open, range, TypeError...), Global (The names assigned at the top-level of a module file, or declared global in a definition of a function within the file), Enclosing function locals (The names in the local scope of any and all enclosing functions from inner to outer), Local (The names assigned in any way within a function and not declared global in that function)

This section introduces the most important rules that will be useful for us in chapter 3 by proposing our own solution. In the following section a useful module from Python of the standard library is introduced.

1.3.2 Python AST module

The shortcut AST means an abstract syntax tree. An abstract syntax tree is a tree representation of the abstract syntactic structure of the source code written in a certain programming language. Each kind of a node denotes some specific construction of a given programming language. This syntax is abstract in the sense of not representing every detail in the real syntax of the programming language.

The Python AST module [5] is a library that allows to obtain the AST of a source code written in the programming language Python. In other words, it is possible to inspect and to modify the Python code using the AST module. This module has a lot of the node kinds. These can be divided into the following groups:

- Literals

- Variables
- Expressions
- Statements
- Control flow
- Function and class definitions

Almost in each node, there is an information about the context in which a certain node is located. The context may be *Store*, *Load* or *Del*. Depending on the context, a behaviour of a certain program may be changed.

The literals are a group of the nodes that is responsible for dealing with the constants in the AST. This module supports following kinds:

- *ast.Num*, *ast.Str*, *ast.Bytes*
- *ast.List*, *ast.Tuple*, *ast.Set*, *ast.Dict*
- *ast.Ellipsis*

The variables are the nodes dealing with the names in Python. There two types of the names (simple and starred):

- *ast.Name*
- *ast.Starred*

The expressions are a big category of the nodes that encapsulate all the work with the expressions in Python. The main node is *ast.Expr*. Within this node, all operations are executed. The kind of the operations may be recognized by a nested node of the operation. The possibilities as follows:

- *ast.UnaryOp*
- *ast.BinOp*

- *ast.BoolOp(op,values)*
- *ast.Compare*
- *ast.Call*
- *ast.IfExp*
- *ast.Attribute*
- *ast.comprehension*
- *ast.Subscript*

Each one of the listed kind of the nodes consists of many various nested nodes. These nodes help to specify the source code sufficiently. As an example, we can take a closer look at the bool operation. The possible nested nodes for the operation are *ast.And* and *ast.Or*.

The group of the statements consists of:

- *ast.Assign*
- *ast.AugAssign*
- *ast.Print*
- *ast.Raise*
- *ast.Assert*
- *ast.Delete*
- *ast.Pass*
- *ast.Import*, *ast.ImportFrom*, *ast.alias*

The next group responds to a control flow in the Python program:

- *ast.If*
- *ast.For*, *ast.While*

- *ast.Break*, *ast.Continue*
- *ast.TryFinally*, *ast.TryExcept*
- *ast.ExceptHandler*

The last but not least, there is a group composed of the nodes dealing with the function and the class definition:

- *ast.FunctionDef*, *ast.Lambda*
- *ast.ClassDef*
- *ast.arguments*
- *ast.Return*, *ast.Yield*
- *ast.Global*, *ast.Nonlocal*

Such a powerful tool creates a perfect structure of a given Python source code. As we will see in chapter 3, the structure of the AST is very important for our solution. In the following chapter 2, the related solutions are introduced and a motivation is formulated as well.

2
Related works and motivation

This chapter is devoted to the related solutions in our field of work. We attempt to describe the logical part of other tools and to show the weak places by a demonstration on the short code snippets. In the last chapter 4, our solution is compared with the works already analysed in this chapter.

2.1 Related works

We have to admit that only the open source solutions were chosen. There are two main reasons why we have made such a decision. We could not afford to buy every software which might have a possible relation to our work. The second point is that we wanted to describe the logical part of each solution. Moreover, it is more likely to be able to take a look into the open source projects than into the licensed projects.

In the next few subsections four chosen solutions are introduced.

2.1.1 Pydev

As the first solution, the Pydev project [6] is presented. Pydev is a Python IDE for Eclipse, which may be used in the Python, Jython and IronPython development. It is not a big surprise that this tool is implemented in a programming language JAVA. The first release is dated on 1^{st} of October 2004 [7]. Pydev includes a lot of the functionality and according to the community forums it is one of the most used in the combination with the editor Eclipse.

The main functionality provided by Pydev:

- **Code completion**
- **Type hinting**
- **Code analysis**
- Code completion with auto import
- Go to definition
- Refactoring
- Debugger
- Code coverage

Our solution has an ambition to be a valuable tool in the field of the error detecting and the code completion. Therefore for our purposes, we analysed mainly the possibilities of the code completion, the type hinting and the code analysis. Pydev can be placed to the group of the tools that work as the statical analysis.

The code completion in Pydev provides the context-sensitive completions and offers a lot of settings to be personalized. In their approach, it is important to have the Python interpreter configured in order to make the code completion to work. In the case of builtins, Pydev communicates with the Python shell, therefore a firewall can prevent the code completion from working if it is blocking the communication from Eclipse to the shell.

The code analysis provides an error finding in the Python programs. According to the project page, it should find the common errors such as the undefined variables, the unused variables and the imports, the unresolved imports, etc. For each category of errors it is possible to set the priority of the error from the most important (Error) to less significant (Warning). The last but not least option is to ignore the given type of error. Moreover, if you do not care about some Python files in your project, Pydev provides you an option to add the Python comment in form *"#@PydevCodeAnalysisIgnore"* to the arbitrary Python file. As a result, the code analysis will skip the chosen file.

The code analysis and the code completion are significantly limited by the static analysis and can not detect a lot of common mistakes. Pydev has an answer also to this issue, but it can make the life of a programmer uncomfortable. In some cases, the special signs (flags) may be added to the code to provide the additional information for the code analysis. As a result, it may look as follows:

```
class Struct:
    '''Attributes passed in constructor.
    *@DynamicAttrs*
    '''
    def __init__(self, *entries):
        self.__dict__.update(entries)
```

The example above shows the usage of the flag *@DynamicAttrs*, which indicates the dynamic attributes in a class. As a consequence, the code analysis will not complain about the undefined variables when accessing any attribute from the class. Unfortunately, this is the only supported flag type.

The same idea of passing the additional information is used for the purpose of the code completion and it is called a "type hinting". For this purpose, Python docstrings are used by the commenting types with the Sphinx/Epydoc format. Below is an example of providing the type-hints using the Sphinx format:

```
class MyClass:
    def method(self, test):
     ':type test: TestCase' #Parameter type of 'test'
        ':rtype unittest.TestCase' #Return type of 'method'

        for unit in test:
            ': :type unit: GUITest' #Local variable
            unit.  #Appropriate code completion
```

As you can see, if you want to rely on the correctness of the code analysis and the code completion, you have to expend not exactly a small amount of effort to hold the additional information up to date. Furthermore, in some cases you do not want to specify the type of a variable because the part of the code is suitable for various objects. In the example bellow without an extra-information there are common errors absolutely ignored and the code completion malfunctioned:

```
class MyClass:
    def __init__(self):
        self.z=4
    def method(self,x):
        return x

m = MyClass()
print(m.www) #No error indication
y = m.method("test")
y. #No autocomplete options
```

These results show us a big space suitable for the improvement and provide the motivation for our work. The whole summary of the existing solutions and the code analysis properties is stated in the following section called Motivation (2.2). Now we will continue with the analysis of the tool named Pylint.

2.1.2 Pylint

Pylint [8] as a tool checks errors in the Python code, tries to enforce a coding standard and looks for the possible semantic errors. This tool is completely implemented in the programming language Python. Pylint was created in 2003 at Logilab [9], that funded Sylvain Thénault to lead its development up to now. It is possible to integrate Pylint with many editors (e.g. Eclipse, Vim etc.) or to use it as a single tool. This solution provides a wide range of the functionality listed bellow:

- Coding standard

- Editor integration
- **Error detection**
- Refactoring
- UML diagrams
- **Extensibility**

In this case, the main part of our interest is the error detection. For our purposes, it is sufficient and more practical to use Pylint as a single tool. Under Linux distributions like Ubuntu, Debian etc. it is really easy to install Pylint (also on Windows, OS X). Afterwards you get a powerful tool which is callable from a command line (Terminal) with many optional arguments. The optional arguments help to configure and to personalize requirements of each user. Of course, there is a possibility to edit/create *.pylintrc* file where you can set all your preferences and configurations. Below, you can see a short example of disabling the warning messages caused by the relative imports and **args*,**kwargs* support:

```
# Brain-dead errors regarding standard language features
#   W0142 = *args and **kwargs support
#   W0403 = Relative imports
disable-msg=W0142,W0403
```

Pylint belongs to the class of the tools which are using the static approach. As we have seen before, the static analysis makes nearly impossible to detect the type errors, the undefined variables, etc. To be more specific, it would need a large effort to detect e.g. the type errors properly (even the common type errors). This solution executes in-depth analysis of the abstract syntactic trees.

According to the specification, the error detection in Pylint should be a very strong tool. It checks if the declared interfaces are truly implemented, if the modules are imported and it supports many more error's messages. At this point, we would like to test Pylint on a small code snippet:

```
class A:
    def __init__(self):
        self.x=4
        self.y=10
    def decorate(self):
        self.decorated='Hello World'

a=A()
print(a.x)
print(a.y)
print(a.www) # warning showed, OK
a.z=10
print(a.z) # no warning showed, OK
print(a.w) # no warning showed, ERROR
a.w=20
print(a.decorated) #no warning showed, ERROR
print(len(a)) # len(a) does not make sense, however, no warning showed
```

The result of the Pylint's analysis is a disappointment. Despite the propositions stated on the website of the project, this tool could not properly analyse the simple (though not a trivial one) example above. What probably does Pylint? A dynamic addition of an attribute to an object is detected without any problems. Also a reading context of a such dynamically added attribute is processed properly. What is the issue for Pylint in this example? Almost certainly Pylint does not know in what sequence will be these statements executed and in some cases it can not determine which statements are executed. This is a very important observation, which can be used later on by the proposing of our solution.

The feature called the extensibility helps Pylint to understand things which Pylint does not know how to resolve them. The aim of this feature is to allow a registration of the functions that will be called after a module has been parsed. When running Pylint against the code using some favourite framework, you should expect a lot of the false positives because of the 'black magic' or whatever else. There are also places in the Python standard library where a dynamic code can cause the false positives in Pylint. The solution is that you can define a special plugin for Pylint [10]. We present the following example:

```
import hashlib

def hexmd5(value):
    """"return md5 checksum hexadecimal digest of the given value"""
    return hashlib.md5(value).hexdigest()
def hexsha1(value):
    """"return sha1 checksum hexadecimal digest of the given value"""
    return hashlib.sha1(value).hexdigest()
```

Pylint reports an error because of a non-existent member *md5* and *sha1* for module *hashlib*. The solution provided by Pylint is the following:

```
from logilab.astng import MANAGER
from logilab.astng.builder import ASTNGBuilder

def hashlib_transform(module):
    if module.name == 'hashlib':
    fake = ASTNGBuilder(MANAGER).string_build('''
class md5(object):
  def __init__(self, value): pass
  def hexdigest(self):
    return u''
class sha1(object):
  def __init__(self, value): pass
  def hexdigest(self):
    return u''
''')
    for hashfunc in ('sha1', 'md5'):
        module.locals[hashfunc] = fake.locals[hashfunc]

def register(linter):
    """called when loaded by pylint --load-plugins, register our tranformation
    function here
    """
    MANAGER.register_transformer(hashlib_transform)
```

Pylint provides us a powerful feature. As you can see, we have to write a fake python implementation by documenting the prototype of the desired class. The question is whether the ratio of the expended effort and the benefits we get is reasonable. Despite these facts, it is a very interesting idea to deal with the similar problems. We can learn from that and reconsider this approach by the proposing of our solution.

In the next subsection, the next Pythonic tool Pyflakes is analysed.

2.1.3 Pyflakes

Pyflakes [11] is a tool implemented in a programming language Python. The first release of Pyflakes is dated on 23th October 2006 . The installation of Pyflakes is really easy just like with Pylint. There is a possibility to use Pyflakes as a single tool in Console or to integrate it with favourite IDE. It is focused on identifying the common errors quickly without executing a Python code (the static analysis). Its primary advantage over Pychecker is the speed of checking. This solution does not control a code style, the refactoring possibilities and other features provided by two previous projects. There are two primary categories of errors reported by Pyflakes:

- the names, which are used, but not defined or used before they are defined
- the names, which are redefined without having been used

These errors may occur when you have forgotten an import, mistyped a variable name, defined two functions with the same name and so on. Pyflakes works by parsing the source file, not importing it, so it is safe to use it on modules with side effects. As far as we know, this project does not provide such the rich configuration possibilities as Pydev or Pylint.

The best way how to determine the power or abilities of a tool is to pass some testing code through it. Consider the following example:

```
import whatever #non-existent module, correctly reported
```

```
class Car(object):
    def __init__(self, fuel, price):
        self.fuel = fuel
        self.price = price
    def addFeature(self, feature):
self.feature = feature

a = Car(100,20000)
b = Car(100) # wrong number of arguments, nothing reported
print(a.price) # no error reported, correct
print(a.feature) # non-existent attribute, nothing reported
a.addFeature('radio')
print(a.color) #non-existent attribute, nothing reported
a.color='black'
a.speed = x #non-existent variable x, correctly reported
```

The result is unsatisfactory. Two trivial cases are correctly detected but other common mistakes are again not recognized. The dynamical adding of an attribute to an object or a class definition is not correctly processed in Pyflakes as well as in Pydev and Pylint. All of these projects are working on the similar principle (talking about the error detection). Therefore, it is very probable that the source of this limitation is the static analysis. In the conclusion of this analysis, we have to admit that Pyflakes seems to be the weakest tool we have analysed till now.

Now, Pychecker will be analysed as the last related work. In this part another approach of detecting the errors in the Python programs will be experienced.

2.1.4 Pychecker

Pychecker [12] finds the problems that are typically caught by a compiler in the less dynamic languages like C++, Java. Pychecker was born in 2001 but the last commit is dated on 8th of January 2011 [13]. Till the year 2003, new releases were published regularly but since then it occurred only very rarely. Despite this fact, Pychecker is very interesting for our purposes because it uses the dynamic analysis. This chosen project has the same installation possibilities like Pylint. We prefer using Pychecker as a single tool without an integration with editor.

The dynamic analysis is a strong approach. As we have discussed in the previous section 1.1, it is not a suitable tool for a project using the modules with the 'side effects' or dangerous constructions. The next interesting point is what Pychecker is able to detect by running the given Python program. Does it work sophistically? Is this tool able to detect multiple afterwards following errors? Let's consider a following dummy example as an input for Pychecker:

```
x=5
while(x):
    print("I am not going to stop!!!")
x=y
```

It is not hard to guess what just happened. Pychecker executes the infinite loop and it depends only on the operating system when this checking of the code is going to stop. Moreover, we do not get any information on the uninitialized variable y at the end of the input. On the other hand, the previous example was not a very common mistake made by a programmer. For this reason, one meaningful example without the side effects is considered:

```
def computeGrade(score):
    if score >= 80:
        grade = 'A'
    elif score >= 70:
        grade = 'B'
    elif score >= 55:
        grade = 'C'
    elif score >= 50:
        grade = 'Pass'
    else:
        grade = 'Fail'
    return grade
```

```
grade = computeGrade(50)
bonus = 10
grade = grade + bonus #grade is not a int, correctly reported
computeGrade(89,66) # too many arguments, nothing reported
```

It comes out that only the first occurrence of an error is reported because the source code is executed only once until the Python interpreter returns an error exception. That is not handy, because we do not get the next error message till we repair/rewrite the wrong code placed before. The advantage is that if Pychecker does not complain about the given code, the code is not going to fail during the runtime.

In the next section we show the reasons of our work. Our motivation is strongly connected with not sufficient results of the current related works.

2.2 Motivation

This part is devoted to the mistakes made by the related solutions. We demonstrate the failures of other projects on the short code snippets. For each snippet we explain errors made by each solution if there are any. At the end of this section it should be clear what the current state in this field of the problem is. Moreover, this analysis provides us the main goals for our solution.

Consider the following class definition A:

```
class A:
    def __init__(self):
        self.x=4
        self.y=10
    def decorate(self):
        self.decorated='Hello'
```

The following statements are:

```
a=A()
print(a.x)
print(a.y)
```

```
10 print(a.www)
```

The variable *a* is an instance of the class *A*. Class *A* does not have any method/attribute *www*. Only Pychecker and Pylint correctly reported an error on the line *10*. In the following statements we exclude Pychecker because it is not able to report more than one semantic error in one analysis. The next statements are focused on the dynamical attribute addition:

```
11 a.z=10
12 print(a.z)
13 print(a.w)
14 a.w=20
```

The attribute *z* is dynamically added to the variable *a*. As it was expected, none of the related solutions reported an error on the line *12*. The sequence of statements on the line *13*,*14* are twisted. None of the projects raised an error on the line *14*.

```
15 print(a.decorated)
16 print(len(a))
```

On the line *15* we try to load the value of the attribute *decorated*. This attribute was not added on previous lines (neither the method *decorate* of *A* was not called) and therefore an error should be reported. Line *16* does not make any sense. The instance of the class *A* does not define a method __len__ . None of the chosen solutions has reported an error in both cases.

```
17 print(i[3])
18 i=3
19 print(i.x)
20 print(i[2])
```

Three errors are located in this piece of a code. On the line *17* we try to index variable *i* that is even not defined. All remaining tools reported an error. Afterwards the variable *i* is defined as an integer. Only Pylint reported correctly an error on the line *19* because of a non-existent attribute *x*. The last error (integers does not support indexing) was not reported by any solution. In the next part we take look at the collections:

```
21  lst=[a,a,a]
22  b=lst[0]
23  b.www
```

The variable *b* is *a*. Despite this fact, the tools did not reported error on the line *23*. The information about *b* being *a* is lost because of working with the list. A similar problem occurs when dealing with the dictionaries. None of the tools reported an error on the line *23*. The similar simulation for the collection *tuple*:

```
24  b,bb,bbb = (a,a,a)
25  b.www
```

Only Pylint was able to detect that variable *b* refers to an instance *a* of the class *A*. Therefore Pylint reported an error on the line *25* because of a non-existent attribute *www*. This observation gives us hope that at least Pylint performs working with the tuples correctly. What about the operations on tuples?

```
26  (_,_,b)=(a,a)+(a,)
27  b.www
28  b=(a,a,a)[0:3]
29  b.www
```

Even the slightest manipulations with a tuple destroy this kind of information. As a result, Pylint and also all other three projects are not able to determine correctly the kind of the variable *b* and do not report the evident error on the line *27*. In the next statement the variable *b* is redefined as the *tuple*. Nevertheless, an error on the line *29* is obvious, none of the tools reported a warning to that. Let's consider the last example. We defined the average function as follows:

```
30  def avg(x,y):
31      return (x+y)/2
32  c=avg(1,'Johny')
33  c=avg(2,4)
34  c.www
```

The passing of the arguments to the function *avg* on the line *32* does make an error because of adding *int* and *string* on the line *31*. Unfortunately, the chosen solutions did not report anything. A variable *c* on the line *33* is clearly kind of *int*. At least

Pylint was able to detect an error on the line *34*. In addition, if the line *33* is dropped, only Pylint did report an error. At a first glance it may look like a fault. But the opposite is true. The Python interpreter stops before assigning some value to the variable c. A logical solution is to consider the variable c in the following statements as some special *any* type. Therefore in this special case the reporting of no error on the last line seems to be a good approach.

In this chapter we wanted to show/analyse the weakness of the related solutions. Despite the fact that Pylint looks like the most powerful tool among others, there are still many possibilities how improve it. We will consider all of these troubles by proposing our own solution (tool).

3 Proposed solution and implementation

This chapter is devoted to the introduction of our ideas in the context of a proposing solution. At first we will summarize the problem definition.

3.1 Problem definition

Our aim of the work is to come up with a way of how to determine a type of a variable in some Python code. As we have seen in chapter 2, the static analysis and the dynamic analysis have both some advantages and disadvantages. In general, the static analysis is more useful and safe for the users (programmers). On the other hand, the dynamic analysis provides the most accurate result as an exchange for safety and running the whole given code while checking it. Moreover, it may cause the unpleasant side effects if working with a database for instance.

Prior proposing our solution, we present our choice of a programming language in which this project will be implemented. We have decided for Python. The reasons are as follows:

- the useful modules like PLY, AST implemented in Python
- a plugin for VIM may consist of a Python code
- a consistency of our solution and the second part [1]
- a possibility to learn a new technology

In chapter 1, we have mentioned that our tool in the combination with the work of our colleague Ľubomír Žák [1] forms a complex plugin customized for the editor VIM. In the context of the final product (*pynfer*), the input for our tool is a retrieved AST from the second part of the solution. For the purposes of the presentation only of our work, we process the input in the following way. The assumption is that the given source code is syntactically correct. The certain AST is retrieved by the Python AST modul (section 1.3.2) from the given source code. Dealing with a syntactical non-valid source code is solved also in the second part.

The working name of our tool is the *Type Inference.* In the following text, our tool can be also referenced by the name *Type Inference* .

3.2 Symbolic execution

In this part, an idea of a symbolic execution is introduced. The symbolic execution is placed somewhere between the static and the dynamic analysis. The dynamic analysis without the need to run the whole code would be a very applicative solution. In the symbolic execution, the Python interpreter will be simulated, but:

- instead of the variables only the type will be used
- the standard library will be replaced by a mock implementation (does not have any side effects and returns the correct types)
- if it is not possible to determine the next steps of the Python program, the non-determinism will be used

In other words, we do not care about the values in the variables but we do care about the types of the variables. If giving a suitable advice to a programmer is not possible because of the given program's complexity, a non-deterministic advice will be used (in the section 3.2.4 explained). Because of simulating the Python interpreter, we have to copy the functionality of Python scope explained in the first chapter (section 1.3.1). The corner-stone of our implementation is composed of two main objects which represent the given source code: *Typez* and *Scope*.

3.2.1 Typez

The *Typez* is an object that represents all types within the *Type Inference*. Each instance of Typez has a reference on its current scope. The *Scope* (3.2.2) may contain the additional information or belonging methods/functions/attributes for a given *Typez* instance. This object supports the following methods:

- *resolve a symbol* - a function to look up a *Typez* instance corresponding to the given symbol with a possibility to choose a mode of resolving, either straight (search only in the scope of self) or a class mode that searches in the scope of self and cascades to a class type and its parents with respect to the *__class__* and *__bases__* attributes
- *resolve a class* - a function that answers *True* if and only if self is descendent of a class with a given name, otherwise it returns *None*
- resolve attributes - a function that returns a list of all possible attributes/functions for self (a code completion purpose)

The process of resolving a symbol has to follow the rules of the multiple inheritance in Python [14]. The only rule necessary to explain the semantics is the resolution rule used for the class attribute references. That is depth-first, left-to-right. Consider the following class definition:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Thus, if an attribute is not found in *DerivedClassName*, it is searched in *Base1*, then (recursively) in the base classes of *Base1*, and only if it is not found there, it is searched in *Base2*, and so on.

3.2.2 Scope

The *Scope* is an object based on a collection *dict* (a data structure dictionary). This object is responsible for the mapping symbols to the *Typez*. To be able to simulate the Python scoping (explained in the section 1.3.1), a reference to the encapsulated scope (the parent scope) is included. The *Scope* supports the main following function:

- *resolve a symbol* - a function to look up a *Typez* instance corresponding to the given symbol with a possible *straight* mode that searches only in self and a *cascade* mode that cascades to the parents of self

The result of the application of these classes may look as follows:

```
a=1+2 #What is a?
```

The variable *a* in our internal representation:

```
a=Typez(
kind : int
    node : AST.num
    value : default_value
    Scope (
     ....
        )
)
```

What does the *Type Inference* figure out? It knows that *a* is an integer and does not know the value because it uses the mocked implementation of the standard library (introduced in the section 3.2.5). For a better understanding, let's analyse the following instance:

```
class A(object):
    def __init__(self):
        self.name="example"
```

```
o=A()
o.a=5
o.b='Hello'
```

A variable *o* is represented as:

```
o=Typez(
    kind : obj
    node : None
    value : None
    Scope (
    'a' : Typez(kind : int, Node : AST.num, value : 5)
    'b' : Typez(kind : str, Node : AST.str, value : 'Hello')
    '__class__' : Typez(kind : class, Node : AST.ClassDef...
    )
)
```

It means that the variable *o* is some object and all required information like the dynamically added attributes, a type etc. are stored in the *Scope*.

3.2.3 Mechanism of symbolic execution

In this section, a mechanism of the symbolic execution is explained. As we have defined earlier in the section 3.1, the input for the *Type Inference* is an AST. A *externs* is a Python file which consists of all basic knowledge for our tool. There are defined all features of the standard library (in the perfect case).

In general, the *externs* is processed prior each execution in the same way as the input AST. The only difference is that the AST of the *externs* needs to be created and it is not retrieved as in the case of the input code. In this case, we have to do it on our own. The complete processing has the following steps:

1. proceed the *externs* by the AST Python module
2. the gained AST is analysed by the *Type Inference* (the process of learning the basic knowledge)

3. the AST of the given code (input) is analysed by the *Type Inference* (the process of checking the input code)

4. supplying the results - a list of errors, warnings

The review above is a brief introduction to the data flow in our solution but it does not describe the core of the symbolic execution. This type of the execution is used in the steps 2 and 3. How does the *Type Inference* analyse an abstract syntactic tree? For a better idea of our concept it is useful to get familiar with the AST. Consider a Python program consisting only of one simple statement:

```
a = 1
```

The string representation of the AST module's result is:

```
Module(body=[
    Assign(targets=[
        Name(id='a', ctx=Store()),
      ], value=Num(n=1)),
  ])
```

A more complex instance of a Python program, consisting of a class definition extended by two decorators (*dec1*, *dec2*) and three basis, is :

```
@dec1
@dec2
class foo(base1, base2, metaclass=meta):
    pass
```

The AST string representation according to the code above is as follows:

```
Module(body=[
    ClassDef(name='foo', bases=[
        Name(id='base1', ctx=Load()),
        Name(id='base2', ctx=Load()),
      ], keyword=
        keyword(arg='metaclass', value=Name(id='meta', ctx=Load())),
      ], starargs=None, kwargs=None, body=[
        Pass(),
      ], decorator_list=[
        Name(id='dec1', ctx=Load()),
        Name(id='dec2', ctx=Load()),
      ]),
  ])
```

The functionality of the AST module is described in chapter 1 (section1.3.2). For our purposes the examples above should provide the possibility for the readers to understand the semantic and describing power of the AST. The mechanism of the symbolic execution works as follows:

1. starts by processing the AST node *Module*
2. the *Type Inference* processes each AST node in the body of the *Module*
3. the *Type Inference* recursively processes the given AST node (depth-first-search)

For the purposes of a recursive analysis an object *Parser* was created.

3.2.4 Parser

The object *Parser* is able to proceed the majority of the AST nodes using already the presented objects *Typez* (3.2.1) and *Scope* (3.2.2). A function for almost each kind of the AST node is implemented. These functions are so called handlers for one specific kind of the AST node. During the recursive analysis, each AST node is proceeded by the proper handler. A handler performs only the minimum amount of the work that belongs to a specific type of the AST node. Afterwards the remaining work is delegated to the nested nodes and if needed the answers from the nested nodes are also collected and evaluated in the parent node. More precisely, the whole idea is to dispatch the work to the *exec_<ast_node_name>*functions, the *exec_name* functions are doing only minimum amount of a job to cover the execution of the specific piece of the code correctly, recursively using the *eval_code* to evaluate the values of their child nodes. Except the large number of handlers, *Parser* implements these following methods:

- *eval_code* - is the main method of the *Parser* that executes a code under the certain node within the given scope. It returns a return value of the code.
- *eval_in_root* - is the same method as the above, but automatically takes the scope as the root scope, which parent is an extern scope (the extern scope represents the *externs*, the highest possible scope).
- *warn* - is a method for creating the error reports consisting of an error message, a certain AST node and a symbol of error.

There is not enough time and space to present all the ideas hidden behind certain handlers but the most interesting were chosen. In the following, the details of a certain AST node processing is described.

Processing of collections

In Python, there are four basic collections: dictionary (*dict*), *set*, *list* and *tuple*. In our solution, the same idea is used for the first three of them. When an AST node is of kind *ast.List*, *ast.Set* or *ast.Dict* proceeded, a new appropriate object *Typez* including a type information of the collection is created. How do we represent all the data within the data structure? There are two possible ideas:

1. to store all elements
2. to store only one representative of all elements

The advantage of the first approach is obvious. It would be possible to simulate all operations on the collections correctly despite the fact that the *Type inference* is interested only in the types of instances (objects). On the other hand, a dangerous program would cause the great issues in this kind of approach. The risks are for instance an infinite loop, in which the element is appended to the list or using a big amount of a memory only because of the analysis. In our solution, the second idea is preferable, because only the type of one element is known. In addition, it is not a usual case that the programmer loads the collections with the different types of elements (except *tuple*, discussed later). Therefore for the collections *set* and *list* only a representative (object *Typez*) is stored. In case of the *dict*, we have to store two types, the *Typez* for key and value.

Why is it not a good idea to implement a *tuple* in the same way? The reason is the wrong assumption about loading the collections with elements of the same type. It is quite common to use a *tuple* like this:

```
def findKeyByValue(value,data):
    for k,v in data.items():
        if v==value
            return (k,v)
    return None

d = {1:'item1', 2:'item2', 3:'item3'}
res_pair = findKeyValue('item2',d)
```

As we can see, a usage of a dictionary almost every time requires a *tuple* loaded with the different types of elements. Therefore, in our implementation the size of a *tuple* is set to the magic number 5 for a reason of being able to analyse correctly at least the simple usage of the pairs, the triples, etc.

Processing of class and function definitions

When a class or a function is defined, the *Type Inference* needs to remember the name of a new class or function. When a Python program is executed by the Python interpreter, a definition of a function is just stored under its name in a scope. On the other hand, a definition of a class is not only stored under its name, but it is also executed (so the interpreter knows what functions are implemented in the certain class). For this reason, the *Type Inference* does not analyse the definitions of the functions till the certain function is really called in the given Python program. Thanks to the evaluating of the class definition it is possible to store the docstring of the class in the object *Typez* according to the class definition. That is not a necessity, but a nice feature of our tool for the code completion.

Processing of if condition

An *if* condition is a very strong tool. Our possibilities of the evaluating condition are limited by the knowledge of the variable's type. In this programming construction we want to determine if the body of a condition is in the runtime executed or the *else* branch is executed. Let's have a look at our motivating example:

```
def makeDecision(x):
    if x>0:
        return 'positive number'
    else:
        return 0

result = makeDecision(5)
```

In other cases, it may be sufficient to evaluate both branches. But especially in this example, running function *makeDecision* with another argument would change the type of the variable *result*. Therefore, the following rules for evaluating the condition were defined:

1. if the condition is simple (the values of arguments are known), then evaluate the appropriate branch.
2. if the condition is too complex, throw a coin with value one on the top and zero on the bottom. If the value one is on the top, evaluate the body of *if* branch, otherwise evaluate *else* branch.
3. The *Type inference* knows always to evaluate the expressions correctly in following form: *isinstance(x,y)*, *x is None*. In other cases the coin may be used.

The ability to evaluate *isinstance* and *is None* as exactly as the Python interpreter, rises from our correct information on a type competence. Throwing a coin is nothing else than introducing the nondeterminism to our solution. This clearly means that our analysis for the same input AST may report various sets of errors every time during the run.

Encapsulating Parser

Because of the nondeterminism, there is a need to make reports of errors more deterministic. It does not look user friendly, when two analysis on the same input report the various errors. In spite of the fact that we are not able to remove the randomness completely, a suitable solution for this issue is proposed. The *Parser* is encapsulated by a new object *FinalParser*.

The *FinalParser* is an object that runs the instance of the *Parser* with the same inputs many times. The count of iterations is defined in advance. Actually, the count is configured by a user. It depends on the user preference between speed and exactness. In the following text, the number of iterations is denoted as X for a better readability. When the *FinalParser* is finished, there are X possibly different sets of the problems generated by the *Parser*. At first, a *Warning* as a new kind of error is introduced. In our terms since now, *Warning* is an error with a lower priority. It may be an error or just a programmer is more clever than the *Type inference*. The first poor attempt to distinguish the errors that arise always and rarely was following. When an error arises in every set of the problems then it is reported as an *Error*. Otherwise as a *Warning*. This approach leads us into the issues. Let's consider the following instance:

```
x = ... #x is a number, value is unknown for Type inference
if x>50:
    print(y)
else:
    print(z)
```

As a result, the *FinalParser* would return two warnings, because of undefined variables y and z. In the evaluation of the condition, the nondeterminism was used, because the value of variable x was lost during the analysis. Therefore the tossing of a coin decides what branch will be evaluated. As a result, the two obvious errors are reported as the warnings only because of the nondeterminism. The mistake is that we do not require the visiting of a certain place of the code/AST . Therefore even if the error is reported every time when the *Type Inference* is checking the certain place in the code/AST, this error is reclassified as a *Warning*. For this reason, the final algorithm is based on monitoring of the visited AST nodes. After each iteration, the following steps are executed:

1. if the current error was not reported in the previous iterations and the AST node has been already visited in the previous iterations, then this error is reclassified as a *Warning* and added to the list of warnings if not already in the list.
2. if the current error was not reported in the previous iterations and the AST node has not been already visited in the previous iterations, then this error remains as an error and the AST node is marked as visited.

This algorithm helps us to eliminate a changing of the set of errors and warnings. Of course, the possibility that some error just vanishes is still here but with the appropriate number of iterations it is reasonably small. The only weakness is that an error may be reclassified to a warning or vice versa, for the same input.

After each running of the *Parser's* instance, the current state of the scope is stored for the purpose of the code completion. The code completion requires an ability to stop the analysis in a certain part of AST. The reason is demonstrated in the following example:

```
1 class Demo:
2     def __init__(self):
3         self.x = 5
4 d = Demo()
5 d.
6 d.y = 'test'
7 d.
```

In the context of the code completion, it is a big difference whether the code completion on the target instance *d* is called before the line *6* or after it. The breakpoint defines a line number before which the execution has to stop. Therefore the state of the scopes is equal to a given place in the input code. Afterwards in each of the *X* scopes, the certain target of the code completion is resolved and results are united into one list of the options without a multiple occurrence.

The object *FinalParser* encapsulates all the functionality of our tool. The installation and manual guide is in the part Appendix of this thesis. The number of the iterations is set to the default value 20.

3.2.5 Standard library

As we have seen in the section 2.1.2 of chapter 2, the standard library of Python has a wide range of the features. In our project we denote the built-in functions as the *externs*. In the *externs*, everything is placed what needs to be implicitly defined. As it was declared, our implementation is a mock one. A short sample of working with the numbers (generally) in the mocking notation:

```
class num(object):
    def __add__(self,x):
        if isinstance(x,num):
            return TypeContainer(num)
        else:
            return TypeError(x)

    def __div__(self,x):
        if isinstance(x,num):
            return TypeContainer(num)
        else:
            return TypeError(x)
```

The mocking class *num* is created. In our solution, this object represents all numbers and it handles the operations called on them. In this example, the *num* class is very limited. Only the operations *__add__* and *__div__* are defined. What will happen if the operation *__mul__* is invoked in the input code? Our tool will report an error based on the non-existing function in the context of the class *num*. On the other hand, consider a case in which a defined operation is called correctly. In this specific instance, the argument *x* is validated whether it belongs to the right class type. If yes, a special class *TypeContainer* is a container that encapsulates the return type of the given function. If not, a special class *TypeError* contains a source of the mistake and later an error report will be generated in the *Parser* (3.2.4). With this approach, the *Type Inference* is taught how to proceed the statements from the standard library. More specifically, we teach the *Type Inference* what types do return the functions and what arguments are of the appropriate types. The appropriate types are those which Python interpreter does not evaluate in the body of the given function as an error. As it is written in the example above, the *num* can be added only together with another *num*. Otherwise a type error is reported because of argument *x*.

3.3 Summary of the implementation

To summarize this chapter, we proposed the idea of our solution and briefly presented the parts of the implementation. We have to admit, that some kinds of the AST nodes are not implemented in an absolutely precise way in the sense that the processing of the certain nodes could be more exact (closer to the behaviour of the Python interpreter). Furthermore, there is still an ongoing process of implementing more and more features from the standard library of Python.

4 Results

This chapter is dedicated to the results of our tool. We present the examples of using our solution and afterwards the *Type Inference* is compared with the related projects analysed in chapter 2.

4.1 Demonstrating the functionality

The *Type Inference* supports a wide range of the constructions in Python and is able to detect the various types of errors. In various cases a detection of an undefined variable (name) is not of the same difficulty. As we have seen in chapter 2, for example in some cases, the solutions were able to report a certain kind of error but on the other hand, no error was reported in a more complex example. Therefore the concrete samples are not bounded to the concrete demonstration of the functionality. The aim is not only to show what errors are able to be detected by the *Type Inference*, but also what does the *Type Inference* know about the type of the variables. This is possible thanks to our approach of resolving the symbols described in the part 3.2.3. For this reason, the various short code snippets are shown and the result of our tool is discussed. During programming of our solution, a large number of tests was created and they describe the functionality very precisely. Therefore, the code snippets are inspired by them.

At first, we will summarize the main functionality of our project. The main functionality is meant in the field of the error reporting. In our opinion, the main contribution of our work is the ability to infer types of the variables. The following kinds of error are supported:

- Name is not defined (undefined variable)
- Object does not support indexing
- Need more than 'x' values to unpack
- Too many values to unpack
- Object is not iterable
- Iterator : does not support __next__ method
- Bad number of arguments
- Unexpected keyword argument
- Multiple arguments for keyword argument
- Unsupported operand type
- Non-existent function (class)
- Exceptions must derive from object *Exception*
- Non-existent attribute
- Unhandled raise of exception

In the next part a few code snippets are presented as a demonstration of the error detection and the type inference. The first example is focused on the non-existent classes:

```
1 class A:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5 class B:
6     pass
```

```
 7
 8  a = A(1,2)
 9  b = B()
10  c = C()
11  d = D()
```

Our solution reports an error on the lines *10* and *11* because of the non-existent function. In addition, the variables *c* and *d* are referenced in our solution as *any* type. Therefore any further work with them would not raise an error. In the following example we will take a look at the non-existent attributes:

```
 1  class A:
 2      def __init__(self, x, y):
 3          self.x = x
 4          self.y = y
 5  a = A(3,4)
 6  a.x = a.z
 7  a.z = a.x
 8  a.w = a.w
 9  a.y = a.z
10  a.t = t
11  t = a.u
```

The *Type inference* reports the following errors based mainly on the non-existent attribute: on the line *6* the non-existent attribute *z*, on the line *8* the non-existent attribute *w*, on the line *10* the name *t* is not defined and on the line *11* the non-existent attribute *u*. The next example refers to working with the functions as attributes (class closure):

```
1  class A:
2      def __init__(self, x):
3          self.x = x
4      def get_x(self):
5          return self.x
6  a = A('test')
7  getx = a.get_x
```

```
8  getx_class1 = A.get_x
9  getx_class2 = A.get_x
10 x1 = getx()
11 x2 = getx_class1()
12 x3 = getx_class2(a)
```

Two different cases of a storing method in a variable were properly evaluated in our solution. For this reason, one error was reported on the line *11* because of a missing instance of *A* in the function call. In the next code snippet, it is shown that the *Type Inference* is able to proceed an inheritance in the proper way.

```
class A:
    def __init__(self):
        pass
    def get_x(self):
        return self.x
class B(A):
    def __init__(self):
        pass
    def get_y(self):
        return self.y
b = B()
b.x = 'john'
b.y = 4
x = b.get_x()
y = b.get_y()
```

The *Type Inference* is able to resolve the methods *get_x*, *get_y* for instance *b* of the class *B*. Moreover, it knows that *x* is of the type string and *y* of the type number. The next example is demonstrating the proceeding of the arguments in a function call. Three types of the passing arguments are considered: positionals, keywords and defaults.

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

```
 4         self.y = y
 5         self.z = z
 6 a = A("test", z = 1, y = True)
 7 x = a.x
 8 y = a.y
 9 z = a.z
10 b = A("test")
11 x1 = b.x
12 y1 = b.y
13 z1 = b.z
14 c = A("test", z = 1)
15 x2 = c.x
16 y2 = c.y
17 z2 = c.z
18 d = A(z = 1, b = 2, y = 3)
19 e = A(1, z = 1, x = True)
```

As we can see in the example above, the class *A* has a constructor with two optional arguments. Five different instances (*a*,*b*,*c*,*d*,*e*) of the class *A* were created with various callings of the constructor. The creation of instance *a* on the line *6* is done by one positional argument and two keyword arguments. No error is reported and our solution disposes of an information on the variables *x* (type string), *y* (type bool) and *z* (type number). The instance *b* is created by calling the constructor of the class *A* only with one positional argument. That is a correct statement because of the two optional arguments and therefore the variable *x1* is a string, *y1* is a number and *z1* is a bool. The instance *c* is a result of the combination of the positional and keyword arguments. Our solution determined also the correct types of the variables *x2*, *y2* and *z2*. On the line *18*, an error was reported correctly because of the unexpected keyword argument *b*. On the line *19* an error arised from the reason of the multiple arguments for the argument *x*.

To sum up, the functionality of the *Type Inferencer* was briefly introduced. Our solution has more features in the sense of the error detection. For the purpose of meeting all the functionalities, we recommend to look into the source code of our project (located in Appendix). The best documentation is the source code. There is possible to find the test file, in which it is easy to deduct all the features. In the conclusion of this chapter, our project is compared with the related solutions.

4.2 Comparison with existing solutions

This section is dedicated to the comparison of our solution with the already existing projects. In the section above, we demonstrated that our ideas and the implementation really works in the practice. In the following text we want to show a contribution of the *Type Inference* to a current state in the field of the study. When comparing many tools, a process of marking the results is needed. To compare this, a benchmarking is described bellow and afterwards the behaviour of all chosen solutions is measured and the results presented.

4.2.1 Benchmarking

The goal of a benchmarking is to rate fairly each of the chosen projects when analysing a given input code. Our assumption is that we are able to define correctly the expected errors in the short samples of a Python program. Afterwards, the following rules are applied:

- the number of the expected errors is the maximum number of the points for a sample
- if a tool reports an error on the right line with a suitable reason, then it gets 1 point
- if a tool reports an error in the clearly correct part of the code, then it looses 1 point
- if a tool reports a warning (of lower importance than error) in the code that is not explicitly harmful, then it looses 0.5 point
- if a tool reports a warning (of lower importance than error) instead of an error, then it gets only 0.5 point

Such an approach guarantees us to choose the most exact tool that is able to detect a smelling code in the programs. In the following part the results of the 'competition' are summarized.

4.2.2 Results and samples

In our comparison every chosen project analysed the given input code. Due to the benchmarking we are able to mark each analysis and to evaluate global ranking of tested solutions at the end. For this purpose we have designed 8 code snippets consisting of the various constructions. The aim was to cover a big range of the possible cases in Python.

The source code of the samples and the concrete results of a certain tool are possible to find in the Appendix. The explanation should not be necessary because a lot of the constructions were already analysed. The results are as follows:

Inputs - max points	Pyflakes	Pylint	Pychecker	Pydev	TypeInf
sample1 - 13p	1	5	1	1	13
sample2 - 4p	1	1	1	1	3
sample3 - 2p	0	1	1	0	2
sample4 - 1p	0	0	1	0	1
sample5 - 2p	0	1	1	0	1
sample6 - 1,5p	0	0	1	0	1
sample7 - 3p	0	2	1	0	2,5
sample8 - 2p	0	1	0	0	2
sum/28.5b	2	11	7	2	25,5

Figure 4.1: Table of results

As we can see, our solution in the field of the error analysis defeated all other chosen projects. We are very proud of this result and we hope that our solution will be released for the public use in the near future. The only limitation in writing these samples for this chapter was choosing the already implemented functions/objects from the standard library. The reasons are mentioned in the section 4.3 of this work.

The disadvantage of our solution is that it does not provide such a wide range of the functionality like e.g. Pylint. This means that the best practise would be to use also Pylint for the purposes of controlling the code style together with our project. This would provide a complex analysis of the Python programs with the suitable advices and it would be especially applicable for the beginners in the Python programming.

4.3 Possibilities of improvement

Our results are very encouraging and motivating for the further work. On the other hand, it is our responsibility to showthe weak places of our solution and to state the possible aims for further improvements. Currently, our solution is mainly limited by the *externs*. In the *externs*, only a subset of the standard library is implemented. That means that the *Type Inference* does not cover all the features of the standard library in the sense of knowing the names and the return types of the functions. Therefore, in the current state our project may generate the false errors. On the other hand, for the purposes of our thesis we would like to emphasise that building the suitable mock version of the builtins is a long time run. That is not because of our incorrect approach, but it requires a lot of the manual work that is definitely doable. For instance, Pylint (2.1.2) has various problems with processing of all the features from the standard library. Please take into consideration that this tool has been existing for more than 10 years.

The second goal for the future plans is to add more features. There are still few kinds of the AST nodes, in which the process of executing could be improved. Nowadays, our solution is not able to proceed the imports correctly. This upgrade would increase the applicability of our solution in the complex Python programs. Despite the fact that the *Type Inference* is missing this kind of the functionality, it is already a strong tool prepared for the meaningful usage. We hope that we will be able to publish our project *pynfer* in few months.

Conclusion

In our thesis, the problem of the inferencing types in dynamically typed languages was were dealt. A tool that is able to detect the possible errors in the Python programs was proposed and implemented. The term of the symbolic execution was introduced as a core idea of our solution.

The symbolic execution is a trade-off between the static and the dynamic approach in the type inference (for more details see section 1.1). We combined the safety of the static analysis with the power and the accuracy of the dynamic analysis. As a result, we got a powerful approach with a big potential. The demonstration of the correctness of our ideas was made by the implementation of this solution in Python.

At this point, it is suitable to summarize the results of this thesis. Our tool provides a wide range of the error detection in the given source code. In the last chapter, the specific functionality is demonstrated on the various code samples. Moreover, the result of our thesis is compared with the already existing projects that were analysed in chapter 2. This comparison showed the power of our tool and in the discipline of the error detection beated the other chosen solutions. Therefore, we would like to release our project in the near future.

The future plans are clearly defined in the last chapter. The main goal is to be able to handle as many features of the Python standard library as possible. As we explained in chapter 2, it is doable, but it requires a lot of time. Even though the various versions of the solutions were being released in the last ten years they are not able to proceed all the standard library. It would be very interesting to take a closer look at the following question. How could be this process of learning the features from the standard library improved?

Bibliography

[1] L. Zak, "Integration of text editor with code-analysing tool," Master's thesis, University of Comenius, Bratislava, Slovakia, 2014.

[2] Python, `http://www.python.org/`

[3] M. Summerfield, *Programming in Python 3: A Complete Introduction to the Python Language*. Pearson Education, 2009.

[4] M. Lutz, *Learning Python*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 5 ed., 2013.

[5] T. Kluyver, "Green tree snakes - the missing python ast docs," 2012, `http://greentreesnakes.readthedocs.org/`

[6] PyDev, `http://www.pydev.org/`

[7] PyDev, `http://www.pydev.blogspot.com/`

[8] Pylint, `http://www.pylint.org/`

[9] Logilab, `http://www.logilab.org/project/pylint`

[10] Logilab, `http://www.logilab.org/blogentry/78354`

[11] Pyflakes, `http://www.pypi.python.org/pypi/pyflakes/0.8.1`

[12] PyChecker, `http://www.pychecker.sourceforge.net/`

[13] PyChecker repository, `http://www.pychecker.sourceforge.net/`

[14] Python, "Multiple inheritance." `http://www.docs.python.org/release/1.5/tut/node66.html`

[15] Python3 documentation, `http://docs.python.org/3/`

Appendix

The zipped folder *appendix.zip* is located on the enclosed CD. Inside the the folder are located two folders named as *pynfer* and *samples*. For the purposes of demonstrating the functionality only of our part (*Type Inference*), we prepared a special approach. Otherwise, we recommend to try out the whole solution located in the GitHub repository. There are also all necessary instructions for the installation in combination with editor Vim.

Please follow the next steps for the installation of the *Type Inference*:

1. Make sure that Python 3 is installed on the current computer. It is possible to check it by typing *Python3* in the console. The additional information on installation is possible to get on the website of Python [2].

2. Unzip the folder named *appendix.zip* and copy the nested folder *pynfer* to the arbitrary location.

3. Run the Python file *RunAnalysis.py* located in the project */pynfer/tool/*. The program *RunAnalysis.py* takes as an argument the Python file that has to be checked. The second argument is optional and it determines the count of the iterations that our tool has to perform. It should be the trade-off between the speed and the accuracy (more precisely in the section 3.2). The default value is set to 20. The command for the terminal may look as follows:

   ```
   python3 RunAnalysis.py sample.py 10
   python3 RunAnalysis.py sample.py
   ```

   The result printed in the terminal may look as follows:

   ```
   python3 RunAnalysis.py sample.py
   Following errors were detected:
   Line 10 - E - Non-existent attribute - x
   Line 15 - W - Unsupported operand type - y
   ```

   If the input code is not syntactically valid, the result is following:

```
python3 RunAnalysis.py sample.py
sample.py contains syntactical errors!
```

In the folder *samples*, there are examples used in comparison with the related solutions in the section 4.2. For each involved solution and each code snippet, there is one file that describes exactly the behaviour. For instance, files named *sample1_pylint.py* and *sample1_pyflakes.py* describe the behaviour of two different tools on the same input code. These files can be also used as the testing examples in the instructions above.