Static Type Analysis for Python

Tiancong Dong, Lin Chen[*], Zhaogui Xu
State Key Laboratory for Novel Software Technology,
Nanjing University, Nanjing, China
dongtccs@gmail.com

Bin Yu
Communication Station,
Unit 95028, P.L.A,
Wuhan, China

Abstract—**Python is a kind of dynamic-typed language which provides flexibility but leaves the programmer without the benefits of static typing. This paper describes PType, a tool that works for static type annotation and inference for python. It could simulate the built-in modules, transform the Python source code to IR(Intermediate representation) which we design and annotate, infer and reduce the IR into the type system. PType could provide the explicit type information, detect the type errors at compile time and improve the efficiency of development and the accuracy of point-to analysis. By the evaluation of applying PType to a suite of benchmarks, we find that PType can annotate and infer the types with a good precision and coverage in accept time.**

Keywords—type analysis; type annotation; type inference; Python

I. Introduction

Dynamic languages like Python, Ruby, Perl are more and more popular. Compared with the traditional static languages like C/C++ and Java, they make programmers focus more on the element of programming rather than the definition of data types. Thus would produce clean code and increase the efficiency of development. However, with the dynamic type system, Python VM(Virtual Machine) would not perform type checking until the run time. This means that Python could neither detect type errors nor provide the type information at compile time. Python could not offer what methods or field of each variable owns at compile time. In addition, the absence of type information makes point-to analysis which is essential for program slicing hard.

In this paper, we make static analysis for Python. First we preprocess the source code to transform the complicated Python grammars to simplified ones. Second, we make type annotation and inference to deduce the program to the type system we construct. As there are lots of built-in modules, classes and functions integrated in Python interpreter without .py source code, we have to annotate these built-in libraries to provide their type information for analysis. With PType, we could detect type errors of web applications or some other Python programs. In addition, the work of Michael Gorbovitski [1] shows that with type information some alias pairs could be eliminated if type mismatch found, thus would increase the precision and efficiency of point-to analysis.

II. Python Type Language

In essence, making type annotation and inference equals to reducing Python programs to the type system. Though Python has one type system in "types" module, those types are just some type names without specific methods or field that the types own. Besides, when used, the "types" module has to be dynamic imported. So we design a kind of new type annotation language on basis of "types" module. The type system contains basic type, any type, user-defined type, self type, collection type, union type and the organization of these types.

Basic types, contains the following types:

- T_{none}, represents the type of undefined value. Each function should have a return value, if not, we suppose it returns none with the type T_{none}.
- T_{num}, is for the digit value including int, long, double and complex.
- T_{string}, works for the string value.
- T_{IO}, for input and output, represents standard input/output, files and devices.
- T_{env}, includes the context information that needed at run time. It is mainly used in annotation of built-in modules, classes and functions.

Any type, T_{any}, represents uncertain type. When a function is defined, its parameters` types are $T_{any.}$

User-defined types, include the types for user-defined objects like function and class.

- T_{func}. We define the type of function in form of function template. It contains function name, the list of parameters and their types, the return type.
- T_{method}, almost the same with T_{func}, has a binding class or class instance.
- T_{class}, is the type of class which contains class name, a list of parent classes, a list of static field(including method) names and their types.
- T_{module}, means type of module. It contains the classes, functions and variables defined in the module.

[*] Corresponding author. E-mail address: lchen@nju.edu.cn

Self type, similar to the "this" pointer in C language, refers to the current object. It usually works as the binding class of methods.

Collection type, is a suite of types for collection object like list, tuple, set, dict etc. The type in the collection contains the item`s type. For example, the T_{tuple}, includes the type of every item, T_{item}, in it.

Union type, as a whole, is a list of types. If some variable`s type is union type, that means its type includes all the types of items in the list.

III. PYTHON STATIC TYPE ANALYSIS

The process of static type analysis can be divided into the following steps. First, preprocess the source code and generate the AST(Abstract Syntax Tree). Second, convert the AST to IR. Third, make type annotation, and then perform type inference.

A. Preprocess

Python supports many new features, such as functional programming etc. which increase the difficulty of type analysis. Our idea is to transform these complex forms to the known form which is easy to process. So before performing type analysis, we preprocess the source code to simplify it.

Generator, in short, is a special function that behaves like an iterator. It is achieved through the yield keyword and usually used in a for loop. One benefit of generator is to save memory space, because instead of storing all the results, programmers could call the next method to get the next result one after another. The way we handle generator is transforming it to a list with all the results stored in it. For generator, the main concern is the return value that is the items in the list after transforming. So the transform is feasible.

List comprehension, is a syntactic construct available for creating a list based on existing lists. Our approach is to transform list comprehension to a procedure that is creating an empty list, traversing the existing lists, operating on each item and adding these items to the new list. In fact, we replace the list comprehension with several simple statements.

Exec statement, is used to execute statements is string or file. In Python, exec and eval, are pure dynamic. We could only handle the situation that the parameter is constant string. If so, we invoke the built-in compile function to compile the constant string to AST, and then add it to the original AST. The eval situation is similar. In addition, if it has explicit locals or globals parameters, we could get the scope of variables in the constant string.

Decorator, is a special function whose role is to wrap other functions or classes to change the behaviors of the wrapped objects. In fact, Python would package the wrapped functions and classes as parameters, and pass them to the wrap function.

```
@wrapper
def foo():
    pass
```

Here, the decorator actually equals to foo = wrapper(foo). So we replace the decorator with this statement.

B. Type Annotation

Here type annotation means annotating the object`s type when it is defined. For function, we add the type information to $T_{func}$ including function name, the list of parameters and their types, and the return type. We also maintain the types of all variables in the symbol table of current scope. Sometimes the return type or local variables` types have related with the parameters` types, then we would associate them. Function in Python is first-class object which could act as parameter and return value of other functions like ordinary variable, so the nested function is allowed. The so-called nested function is a kind of function with other functions defined in it. As we maintain the symbol table and the symbols` type, the type information of inside function would be found in the symbol table of outside function. If the function is multi-nested, the type of inside function would be stored in its direct outside function. One application of nested function is closure. We call the statements composed of function and the execution context of these statements closure. Each function has a field __globals__ which contains namespace where function is defined. In static analysis, we adopt the symbol table to indicate closure information. For class, the annotation includes annotating all the class`s methods and field, maintaining its collection of parent classes.

Built-in modules are widely used in Python. Through the statistics of Python standard library, we find that one-third of the built-in modules are cited more than 10 times, some of them even cited more than 100 times. There are 62 built-in modules with lots of classes, functions and constants. Because these built-in modules have no Python source code, we cannot analyze their type information directly. So we would simulate the built-in module with Python.

According to whether generating side effects, functions in built-in modules can be divided in two types. The one that generates side effects will create, modify or delete the object on the heap. The other type would just generate status information. We have two strategies to process the built-in modules. First one is fully functionally simulating and implementing the built-in modules with Python. It means that we can completely replace the original built-in modules with our version. The disadvantage of this approach is that it would take a huge workload. The other strategy is only providing the type information required for analysis. For example, for functions, we simply annotate the parameters` types and the return type. The advantage of this strategy is that it is relatively simple to implement, and still obtain the type information, while the drawback is that it is not precise enough, neither could handle the functions with side effects. If we need to make another kind of analysis, we have to re-simulate the built-in modules. So we have adopted a compromise solution that is annotating the type information for the built-in elements without side effects and fully functionally simulating for the ones with side effects.

Some functions in built-in modules will return the built-in types, such as the structure of the C language. In this situation, we define classes with the same name with built-in types. Some will collect the context information when executed or perform differently according to the context environment. These functions are usually used for debugging or have related with the operating system. So we define the type, T_{env}, to represent dynamic context environment. Some built-in functions could be invoked at any time without importing any other modules such as abs(). These functions are relatively simple to process, while the difficult of processing built-in elements lies on the built-in modules imported in the standard library which usually have more complex logic or create, quote, modify, delete other built-in elements. In order to apply the type information gotten from simulation, we encapsulate the built-in modules, classes, functions and constants into an abstract layer. Once built-in elements invoked, we redirect the call to the abstract layer, and analyze the module that we have simulated. Due to the extensive use of Python standard library, we annotate some commonest modules of the standard library in addition to built-in modules.

C. *Type Inference*

Shown in Figure 1, we adopt a type inference algorithm based on the constraint graph. First traverse the entire program, visit each statement, and then get the dependencies between types by constructing a constraint graph. Each object in the symbol table will have a collection that contains the object`s maybe types. Nodes in the constraint graph represent variables or expressions, while edges represent the constraint which could be obtained through data flow. When adding a new type in the variable`s type collection, the type would also be added to the collections of adjacent nodes. The algorithm in this paper is flow insensitive with a lower space complexity and relatively simple to implement compared with flow sensitive one. As it is a static analysis at compile time, we take a conservative plan that deduced type set at least contain the actual type collection. Through the evaluation below, we find that in most cases the differences between inference type collection and actual ones are less than 50%.

```
type_inference
{
    preprocess(source code)
    graph = initialize_constraint_graph()
    while(has_next_statement)
    {
        stmt = get_next_stmt()
        type = get_type_info(stmt)
        update(graph, type)
    }
}
update(graph, type)
{
    while(type_set_changed)
    {
        propagate_type_in_graph(type, graph)
    }
}
```

Fig. 1. Type inference algorithm

Python supports dynamic access to object`s attributes by getattr(obj, attr). For getattr, we support a limited, conservative approach that is the attr have to be a constant string. We would find and get the types with the name attr in obj`s maybe type collection.

IV. EVALUATION

We make two experiments to verify the effectiveness of the type analysis in the paper. In experiment I, we annotate the types of some modules from standard library to validate that PType could handle the built-in modules. In experiment II, we make type annotation and inference on a group of open source programs to evaluate the running time and precision .

For any variable, the Set_{infer} donates the type set that we get by making type analysis while the Set_{actual} is the variable`s actual type set. We define that the type analysis is accurate if $Set_{actual} \subset Set_{infer}$ and $|Set_{actual}|/|Set_{infer}| \geqslant 0.5$. $|S|$ denotes the size of set S. The $|Var|$ means the number of all the variables in the program. The $|Var_{accu}|$ means the number of variables whose type information we get by analysis is accurate.

The results of experiment I are shown in Table I. The first three columns list the modules` names, their sizes as computed by SLOCCount, the built-in modules imported. All the type analysis for these modules from standard library takes less than 20s with the precision greater than 85%. In standard library are mainly the definitions of classes, functions and variables with many functions` return types related with the types of parameters. So the type analysis for standard library is more accurate. Among the modules we select, the os module is a little special for it takes longer but with less lines of code. We find that this module imports many other built-in modules that have to be analyzed, so it takes much time.

Table II shows the results of experiment II. The first program is a benchmark program that is used to measure the performance of Python interpreter. The remaining six are popular open source programs from Github. The precision column shows that the majority of precision rate is greater than 80%. The analysis time is no more than 90s even some programs have more than 1k lines of code. We have to note that the time in Time column excludes the analysis time for the imported standard library modules, because the lib modules have been annotated before with the type information that we could directly use. In real programs, the dynamic features that would damage the accuracy and efficiency of static analysis are not widely applied, so the precision of type inference could be high.

TABLE I. EXPERIMENT I. RESULT

Module	**LOC**	**Time(s)**	**$\vert Var\vert$**	**$\vert Var_{accu}\vert$**	**precision**
ast	152	10.2	170	161	94%
timeit	174	9.8	126	109	86%
imputil	309	12.1	276	250	90%
os	421	18.5	358	315	88%
aifc	667	13.0	364	352	96%
threading	678	13.7	482	470	97%
zipfile	1081	17.7	703	674	95%

Some main reasons make the type analysis for some variables inaccurate. First, the uses of exec, getattr, setattr and delattr make it hard or even impossible to obtain the accurate attributes of some objects in static analysis. Second, some variables` values have related with dynamic environment or the input of programs. Third, the exception would change the program`s control flow, which make it inaccurate for type annotation. At last, the types in the type inference algorithm based on constraint graph are transitive, which makes the inaccuracy caused by the above reasons propagate to the adjacent node in the graph.

TABLE II. EXPERIMENT II. RESULT

Program	LOC	Time(s)	|Var|	|Var_{accu}|	precision
pystone-1.1	203	12.7	172	172	100%
lice-0.4	227	13.1	183	152	83%
schedule-0.2.1	312	11.9	139	121	87%
toro-0.5	631	34.7	302	248	82%
boom-0.9	738	41.4	549	434	79%
sure-1.2.0	1695	78.6	1356	1125	83%
sqlparse-0.1.11	2376	83.2	1478	1241	84%

V. Related Work

Some researchers have previously studied making type analysis for dynamic languages including Ruby, Javascript, and Python etc. For Ruby, Morrison [2] developed a type inference algorithm that has been integrated into some IDE of Ruby on Rails. Michael Furr, Jong-hoon An, Jeffrey S.Foster, Michael Hicks [3][4] designed RIL, a kind of Ruby intermediate representation and developed a tool named DRuby. They used it to make type analysis for Ruby programs and detect the type errors at compile time. There are some work [5][6] for Javascript as well. Brian Hackett and Shu-yu Guo [7] developed a hybrid type inference algorithm and apply it in the Firefox browser. For Python, the early work of type analysis focus on reducing the run-time type checking to improve performance, typical are the Starkiller [8][9] from Michael Salib and the inference for atomic type from Cannon [10]. J.Aycock [11] proposed aggressive type inference which could transform the program of Python to Perl. There are some proposals making changes to Python. RPython [12] is a statically-typed subset of Python which limits the dynamic features of Python. The work of Alex Holkner and James Harland [13] shows that the dynamic features are rarely used in real programs. TPython [14] is an expansion of Python adding static type declaration. It also modified the Python interpreter to support the added syntax. The father of Python, Guido van van Rosssum [15] once considered adding optional static type and permitting programmers to explicitly add type for the variable. But this would bring a series of problems like duck type. The type inference algorithm in this paper is a variation on standard techniques [16].

VI. Conclusion

We develop a tool, PType, to make type analysis including constructing a type system, preprocessing the source code to simply it, making type annotations for classes, functions and built-in modules, making type inference based on constraint graph. We applied the tool to a range of programs and standard library modules and found that it has a good precision in an acceptable time. With the type information, type errors would be detected at compile time.

Acknowledgment

This work is supported in part by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61272080, 61170071), and the Natural Science Foundation of Jiangsu Province (BK2011190).

References

[1] Michael Gorbovitski , Yanhong A. Liu , Scott D. Stoller , Tom Rothamel , Tuncay K. Tekle, Alias analysis for optimization of dynamic languages, Proceedings of the 6th symposium on Dynamic languages, October 18-18, 2010, Reno/Tahoe, Nevada, USA

[2] J. Morrison. Type Inference in Ruby. Google Summer of Code Project, 2006.

[3] Michael Furr , Jong-hoon (David) An , Jeffrey S. Foster , Michael Hicks, Static type inference for Ruby, Proceedings of the 2009 ACM symposium on Applied Computing, March 08-12, 2009, Honolulu, Hawaii

[4] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. The Ruby Intermediate Language. In Dynamic Language Symposium, 2009

[5] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In ECOOP, pages 428–452, 2005

[6] P. Thiemann. Towards a type system for analyzing javascript programs. In ESOP, pages 408–422, 2005

[7] Brian Hackett , Shu-yu Guo, Fast and precise hybrid type inference for JavaScript, Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, June 11-16, 2012, Beijing, China

[8] M. Salib. Starkiller: A Static Type Inferencer and Compiler for Python. Master's thesis, MIT, 2004.

[9] M. Salib. Faster than C: Static type inference with Starkiller. Master's thesis, MIT, 2004.

[10] B. Cannon. Localized Type Inference of Atomic Types in Python. Master's thesis, California Polytechnic State University, San Luis Obispo, 2005

[11] J. Aycock. Aggressive Type Inference. In Proceedings of the 8th International Python Conference, pages 11–20, 2000

[12] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. Rpython: Reconciling dynamically and statically typed oo languages. In DLS, 2007.

[13] Alex Holkner , James Harland, Evaluating the dynamic behaviour of Python applications, Proceedings of the Thirty-Second Australasian Conference on Computer Science, January 01-01, 2009, Wellington, New Zealand

[14] Yu C, Zhou TL, Zhou XY, Chen L, Xu BW. TPython: An Extension of Python. Computer & Digital Engineering. Pages 54-58, 2009(5)

[15] Guido van van Rossum. Adding Optional Static Typing to Python. https://www.artima.com/weblogs/viewpost.jsp?thread=85551

[16] A. Aiken, M. F¨ahndrich, J. S. Foster, and Z. Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In TIC, pages 78–96, 1998.