Python Probabilistic Type Inference with Natural Language Support

(Artifact)

Zhaogui Xu[*], Xiangyu Zhang[†], Lin Chen[*], Kexin Pei[†], and Baowen Xu[*]

[*] State Key Laboratory of Novel Software Technology [†] Department of Computer Science
[*] Nanjing University, China [†] Purdue University, USA

zgxu@smail.nju.edu.cn {xyzhang, kpei}@cs.purdue.edu {lchen, bwxu}@nju.edu.cn

1. SCORECARD

1.1 Insightfulness

Python is increasingly popular in both academy and industry. According to IEEE Spectrum [1], Python ranks number 4 among all the mainstream programming languages in terms of popularity. However, since Python is a dynamic language, developers often suffer from the lack of type information during development, which leads to lots of type bugs encountered in real-world Python programs. Unfortunately, type inference for Python programs is highly challenging. According to our empirical study, over 40% function calls in Python projects are external functions calls. These external functions are usually very difficult to analyze due to the different programming languages or lack of source code. Most existing works conduct type inference by observing data flow from variables with known types to variables with unknown types. As a result, when encountering external function calls, they heavily rely on manual mocking of these external functions. Unfortunately, manually mocking is extremely expensive and impractical for real world Python projects. Besides, many Python projects serve as libraries which provide APIs to other downstream projects. These APIs are usually not invoked within the project. Since most existing works are forward analysis that proceed along the direction of control flow (e.g., abstract interpretation), they can hardly infer API parameter types.

We develop the first probabilistic type inference engine for Python programs. Unlike existing works, we leverage a probabilistic model to bridge various kinds of uncertain type hints in Python programs, which include attribute accesses, data flow, variable names and explicit type checks. Leveraging probabilistic inference, our tool allows the confidences of individual type hints to be propagated and aggregated across different variables. Eventually, our tool converges on the probabilities of the variable types. Through ranking the computed type probabilities, our tool can effectively filter out those unlikely types according to some given threshold.

1.2 Usefulness

Our tool is highly useful in practice. We have evaluated our tool on 18 real-world Python projects, and the results show that our technique can type 79.09% of the variables that cannot be typed by a state-of-art system. Our tool has the precision of 82.86%. In other words, developers can substantially benefit from our tool, e.g., avoiding type errors and getting hints in API usage. Since our tool infers types according to various type hints, it easily avoids tedious manual mocking of external functions. Moreover, our tool can infer types of API parameters even when these APIs are not invoked within the project. For a better study of our technique, we also prepare detailed studies of interesting cases on our website `https://sites.google.com/site/pyprobatyping/`. Our tool is also cost-effective. It can infer the types for most variables within one second. This suggests that our technique can be integrated in an IDE (in the future).

1.3 Usability

Is it easy to understand? Our paper has already presented a detailed description about our technique and evaluation. In the artifact package, we have prepared detailed description about how to use the tool and how to validate the results.

Is it accompanied by tutorial notes? We have prepared a pretty formatted README file in which we present a detailed tutorial for the users about the requirements, instructions and evaluations.

Is it easy to download, install, or execute? We put our tool, benchmarks, running scripts and documentation in a virtual machine image so that everything is set up properly to avoid tedious configurations and potential incompatibility issues. The image can be downloaded from our project website `https://sites.google.com/site/pyprobatyping/`. For a quick start, we have installed all the required environments in the VM image. To start the tool, the users only need to input several commands in the terminal. For easy validation, our tool also pretty prints the results.

2. ARTIFACT DESCRIPTION

We present our artifact in a VMWare image which can be started on any host with a VMWare Workstation (version 10) installed. In this section, we will give a detailed description about how to use the artifact.

Installation of VMWare Workstation. Our artifact image requires VMWare workstation 10, which can be downloaded from [2]. Note that the Linux version of VMWare Workstation is packaged in a "`*.bundle`" executable so that one can use the command "`sudo sh /path/to/<filename>.bundle`" to install it.

Loading the VM image. First of all, go to our website `https://sites.google.com/site/pyprobatyping/` to download our VM image. After decompressing it, start the installed VMware Workstation and go to menu `File>Open>Select the *.vmx file` to load the image.

Requirements for starting the image. The initial memory allocated for our VM image is 7GB, and the number of processors and cores is 2x2. If the host machine cannot survive it, please go to `VM>Settings>Hardware Tab` to reset the resources just before starting the VM. If the allocated memory is too low, we cannot ensure all the benchmarks will run successfully. In addition, the image requires at least 15GB free disk space on the host. To get a better performance, one can also give the image more resources (e.g., memory and processors).

Artifact contents. After loading the image, the HOME directory contains the following contents:

- A README file on the desktop (`$HOME/Desktop`) in the VM, including basic description of the working directory, execution requirements, instructions, evaluation and so on.
- A Python environment with our tool and the required libraries installed.
- A working directory (`$HOME/Current/NamingProject`), including benchmarks, data, test drivers and running scripts. The structure is listed as below:
  - `Benchmarks` folder, including all the benchmark source code.
  - `Data` folder, including all the dynamic collected data (e.g., variables and types) by our tracing tool.
  - `SData` folder, containing all the static data generated by PySonar2, which includes the statically inferred types of each variable and data flow between variables.
  - `MData` folder, including all the merged data of dynamically and statically collected. This folder is initial empty and will be dynamically filled by our tool.
  - `tests` folder, including all the running scripts, configurations and testing drivers.

We also provide some case studies on our website `https://sites.google.com/site/pyprobatyping/`.

Running the Tool. First of all, open a terminal, and then move to the working directory by command: `cd $HOME/Current/NamingProject/tests`. We provide two kinds of evaluation. One is to execute all the benchmarks by command: `./run.sh [-l=<N>] <HIGH> <ETA>`. Here, `[-l=<N>]` is an option to choose how many kinds of constraints (`N=1...4`) are considered with the order corresponding to Figure 12(d). Note that all kinds of constraints will be included if this option is omitted (i.e., `N=4` by default). `<HIGH>` and `<ETA>` represents the high probability threshold *HIGH* and the belief threshold η of the naming convention, respectively. For instance, to evaluate the results of Table 2, one just needs to input the command `./run.sh 0.95 0.7`. For details of these thresholds, please refer to Section 3 and 5.4 in our paper. It will take several hours to complete all the benchmarks, so another choice is to run each benchmark one by one by command: `python run.py test-XX-YY [-l=<N>] <HIGH> <ETA>` where `XX` and `YY` represents the category and project name, respectively. One can find them all as `test-XX-YY.py` in the working test directory (`$HOME/Current/NamingProject/tests`). Take the benchmark `httpbin` as an example, the corresponding command should be `python run.py test-httptools-httpbin 0.95 0.7`. We actually put all these information in the README file.

Evaluating the Results. The evaluation goal is to reproduce results shown in Table 2 and Figure 12. These are the main results of our experiment. To evaluate the results step by step, please follow the instructions in the Section [Evaluation] of README. Our tool will output all the results in the folder `log/<project-name>`. The logged files are listed as below:

- `raw-diff-same-summaries.txt` stores the overall failure percentage of PySonar2. Note that the results are based on the traced variables.
- `analysis-summaries-<N>-<HIGH>-<ETA>.txt` stores partial summarized results corresponding to Table 2 and Figure 12.
- `analysis-results-<N>-<HIGH>-<ETA>.txt` presents the detailed results of the inferred types associated with probabilities. Refer to README for the meaning of each record.

3. REFERENCES

[1] IEEE Spectrum http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

[2] VMWare Workstation https://my.vmware.com/web/vmware/info?slug=desktop_end_user_computing/vmware_workstation/10_0