Brainstorming About Software Development Difficulties

Qifan Lu (1822799; lqf96@uw.edu)

Context-Aware, Smarter API Searching and Code Completion

It is no surprise that most people spend a large portion of their time search for API usages rather than typing when they code. Usually, programmers either turn to API documentations for help, or they search on Stack Overflow for API usages and discussions. This process usually requires switching between multiple windows and relatively complex mouse or keyboard operations, which is time-consuming and sometimes frustrating.

Modern IDEs and plugin-enhanced editors usually provide auto-completion functionality and API documentation searching functionalities in order to improve coding efficiency and reduce typing. Unfortunately, none of them can help if a programmer knows his/her intention. Most auto-completion implementations sort candidates in alphabetical or frequent-based order, and does not take context information (such as nearby variable names, function calls or comments) into consideration. Documentation search might help programmers find the APIs they are looking for, but it still involves window switching and manual typing of APIs.

To provide a more coherent coding experience, we can propose a context-aware, smarter API searching and code completion plugin. Such a plugin will utilize a feature-rich source code and documentation dataset. During coding process the plugin will actively scan the context, extract features and determine the order of candidate APIs using machine learning. In case the programmer is not satisfied with the auto-completion result, he/she can enter extra hints in a pop-up search box. Preliminary researches and applications on neural auto-completion for E-Mails [1] and source code [2] have been emerging in the recent years; however, combining AST analysis result with hints from comment utterances remains an important research problem. Furthermore, efficient large-scale dataset training on servers and application and storage on client machines will be the prerequsite of widespread use of such plugins.

Testing Pull Requests and Merging Numerous Pull Requests for Large Projects

Large open-source projects can contain up to hundreds of thousands of testcases and receive up to hundreds of pull requests per day. Because of the huge amount of testcases it usually takes hours

for the testing process of the whole project to complete. The problem is exacerbated when there are many pull requests waiting to be tested, while the number of testing machines (usually CI containers or VMs) are limited. This can cause pull requests for these projects to queue up and severely affect the workload of testing and merging patches.

To combat these problems, large open-source projects usually use special workload to alleviate the burden on CI infrastructures. For example, the Rust project uses BORS, an automated integrator to handle testing of pull requests against Rust compilers. It supports setting priority of pull requests so that important pull requests are tested first. But perhaps the most important functionality of BORS is roll-up pull request [3], in which multiple pull requests (usually consist of small modifications) are merged and tested together. This greatly improves testing efficiency, but if the roll-up pull request failed one has to manually find out the individual pull request that causes the failure. And even with this improvement large open-source projects such as Rust compilers can still suffer from pending pull request problem.

For large open-source projects, the fundamental way to solve this problem is to introduce "incremental" testing. This can be done by analyzing variable, function, class or interface dependency of a particular test case. If none of the dependencies of a testcase changes then the testcase can be skipped; however, the soundness of skipping analysis can be a problem. Besides, the testcase may leverage invariants and heuristics of APIs to skip testcases so that modifications to basic components (e.g. Rust or LLVM data structures) do not trigger full-scale testing of the whole project. Finally, dependency analysis may help detemining the particular pull request that causes the failure of a roll-up pull request, so that it can be excluded and the roll-up pull request can be tested again.

Weak Typing to Gradual Typing or Strictly Typing Refactor

Strongly-typed programming languages were the mainstream when computers were slow, before dynamically-typed languages became popular due to their terse syntax and duck-typing nature. However, as open-source projects become larger, people start to favor strongly-typing program again because of its clear description of program interfaces and prevention of program misuses. Python introduces optional type annotations in Python 3.5, and many popular JavaScript projects are rewritten in TypeScript which is a gradually-typed superset of JavaScript.

The refactoring of weakly-typed programs to strongly-typed programs, also called program type annotation, can be tedious and error prone. Therefore, we look for tools that can automate this API typing process. For Python programs, Dropbox and Instagram have open-sourced two libraries (PyAnnotate [4] and MonkeyType [5]) that generates type annotations by running the program and collect type information at the runtime. In other words, both libraries only make use of dynamic analysis. For JavaScript, no library exists to automatically annotate JavaScript code and refactor them as TypeScript code.

One major drawback of PyAnnotate and MonkeyType is that they are only using dynamic analysis to collect type information, which, like all other dynamic analysis techniques, is unsound and may leave out possible types. Furthermore, these libraries cannot handle generic types and produce naive typings that need refinements from developers. One solution to these problems is to combine static analysis with dynamic analysis. We can first run the program and observe the possible types of variables from dynamic analysis, and then use these observed types to deduce the types of variables and parameters through abstract interpreter. We can also leverage the natural language utterances in JSDoc or Python DocStrings to deduce the type of parameters and return values, which has been explored by [6].

References

[1] Yonghui Wu. Smart Compose: Using Neural Networks to Help Write Emails. https://ai.googleblog.com/2018/05/smart-compose-using-neural-networks-to.html
[2] Neural Complete. https://github.com/kootenpv/neural_complete
[3] Add PR "rollups". https://github.com/graydon/bors/issues/34
[4] PyAnnotate. https://github.com/dropbox/pyannotate
[5] MonkeyType. https://github.com/Instagram/MonkeyType
[6] R. S. Malik, J. Patra, M. Pradel. NL2Type: Inferring JavaScript Function Types from Natural Language Information. https://software-lab.org/publications/icse2019_NL2Type.pdf

Remarks

It takes three hours to complete this homework.