Beyond Autocomplete: Automatic Function Definition

Kyle I. Murray and Jeffrey P. Bigham
Department of Computer Science
University of Rochester
Rochester, NY 14627
kyle.murray@rochester.edu, jbigham@cs.rochester.edu

***Abstract*—Programmers have used autocomplete to reduce the cognitive overhead of remembering exhaustive lists of APIs for years. Autocomplete has a primary and obvious point of failure: when a programmer expects a certain method or function name to exist and it does not, the autocompletion list simply stops displaying results and disappears. We describe *automatic function definition* (AFD), which can succeed where autocomplete fails. It is a novel way to reduce the impact of threadbare libraries, increase the coding speed of primary programming tasks, and distribute work among different types of programmers and automatic tools. Instead of seeing an empty list, users can instead perform automatic function definition, which uses several sources to define the function that the user intended to use. We present three complementary techniques for defining functions based on the information about the function that a user provides while writing code as usual: code search, fellow programmers, and the crowd. Finally, we discuss our implementation of this work in progress and plans for evaluation.**

I. Introduction

Autocomplete is a common feature of integrated development environments. Traditionally, however, it has only been used to list names that have already been defined in a referenced library or that exist elsewhere in a given project. While it is true that many programming tasks involve frequent references to existing code, the reason for writing a new program or script in the first place is almost always to define something that does not already exist or that is not readily available to the programmer who is writing this new program; this is her *novel task*.

While coding, a programmer is likely to encounter a failure of autocomplete when some method that he expected to exist in a library he was using does not actually exist. It could be that another language's library has the method or that he thought that such a useful method *ought* to exist on the class and therefore assumed that it *was*. At this point, there are only a few ways to deal with the lack of code. First, the programmer can write what would have been the body of the missing code inline with the rest of the code for that project. Alternatively, he can often use a language's dynamic extensibility features to define the method, either globally or modularly, himself. Finally, the programmer can decide to skip this portion of the code for the moment and proceed to complete the actual novel task without ceding his valuable time and concentration to distractions.

All of the above approaches for dealing with the presence of an unexpectedly missing piece of functionality have flaws that impose immediate cognitive costs on the programmer. *Automatic function definition* (AFD) avoids these situational impairments by allowing the programmer to delegate the task and immediately continue working on the intricacies of the unique task that the programmer expected to be working on from the start. While she continues, the missing functions that she chose to have defined are being filled out in parallel from the various definition sources that we describe next.

II. Definition Sources

A. Code Search

A huge amount of code exists on the web, which has lead to the development of tools such as Sourcerer [1] and Google Code Search[1] to crawl it all. These tools can take as little as a keyword and a source language, but they also support finely-detailed search based on code structure, such as Sourcerer's *CodeRank*. These existing systems suggest that code search alone would be able to cover many situations in which a programmer might want to have function automatically defined. Some languages even have community-built repositories of commonly used functions that are missing from the standard libraries; these are ideal corpora to crawl for use with AFD.

B. Other Programmers

Large projects often rely on the ability of the senior programmers and managers to effectively delegate portions of the projects to teams of other programmers. AFD benefits from the availability of these other programmers by being able to delegate a task to someone else within seconds of the original programmer realizing that the task exists; it presents delegated tasks in one of the most prevalent modular forms: the function. Large projects are not the only programming situations in which task delegation is practical today. Services such as oDesk[2] allow even a single, self-employed programmer to subcontract other programmers by the hour. We imagine situations in which a single programmer subcontracts another programmer for a few hours during a time crunch near an important deadline and has the subcontractor handle a stream of delegated function definition tasks.

C. Crowdsourcing

A burgeoning solution to problems that cannot be solved completely with AI but that can sometimes be too expensive or

[1] http://google.com/codesearch
[2] http://odesk.com

slow to complete with dedicated, expert humans is the crowd [2]. The aforementioned function definition sources fall into the AI and dedicated human categories, respectively, and may therefore have the propensity to occasionally fall victim to the same problems that crowdsourcing was designed to solve. Interestingly, the crowd can be instructed to write code *or* search for code. This helps to alleviate concerns that crowd workers would not be experts enough to write certain code themselves, or that they would be as limited by existing code as regular code search.

III. Related Work

Collabode [3] provides an interface and environment for programmers to delegate tasks to collaborators and for collaborators to submit results in a process that Goldman *et al.* call micro-outsourcing. Tasks are specified by opening an interface where users type natural language instructions to collaborators, and the scope of the task is also specified in natural language. This differs from AFD in that the primary programmer is taken away from her own programming task in order to write instructions for someone else, and because only close collaborators are used as sources of code.

In CodeGenie [4], a user first writes unit tests to validate possible code that he wants CodeGenie to find. One these tests are written, CodeGenie uses Sourcerer [1] to find relevant code that passes the unit tests. Again, the most notable difference is the requisite time that the primary programmer must spend diverted from a task in order to write the code for the unit tests. The approach also relies on a single method of finding source code, code search, instead of using several. Calcite [5] provides assistance for using unfamiliar APIs. Mooty *et al.* introduce *placeholder methods*, which are method names that show up in the code completion combobox, but that do not actually exist. The text beside the placeholder method name documents, in natural language, how the "desired functionality" indicated by the placeholder method could be implemented. This text is crowdsourced. No *code* definitions are given for these placeholders, differentiating them from our AFD approach.

IV. Implementation

We implemented a testbed for AFD in a prototype, web-based JavaScript development environment. Code analysis is performed using Dimitrios Vardoulakis' implementation of type inference for JavaScript, based on his and Olin Shiver's CFA2 algorithm [6], which provides a type-annotated abstract syntax tree of a given source string. We use the tree and inferred types to provide standard autocomplete and to determine the proper placement of automatically-inserted definitions in source files. Automatic function definition is triggered by the user an option from the typical autocomplete combobox. One entry in the combobox has the function of initiating automatic definition and is present even when no other suggestions are left. We have not yet implemented the submission interface for other programmers or crowd workers, or the code search crawler.

V. Future Work

Automatic function definition is a work in progress, so significant future work remains to be done. Primarily, an interface and search mechanism that submit results back to the existing programming environment will be created. The human interface will likely use the same editor and submit completed function definitions back to an existing server that communicates with the primary programmer's editor. Past experience with Amazon's Mechanical Turk service for recruiting crowd workers [2] makes that service a likely choice for crowdsourcing testing, although there may be crowds more suited for the specific tasks of programming or finding existing code on the internet.

A formal evaluation of the system with JavaScript programmers is planned. It will compare the different definition sources against each other and against an environment without any AFD at all. Users will be given sufficient instruction and training on both types of environments. Metrics to which we will pay close attention include completion time, code accuracy, and users' opinions and thoughts on the system itself.

VI. Conclusion

We have presented *automatic function definition*, a novel way to transparently increase the apparent utility of code libraries, decrease the overall time spent coding for an individual, and distribute work among different types of programmers and automatic tools. We described three ways of sourcing function definitions automatically, and implemented a programming environment that is ready to accept definitions once they are provided. As work in progress, we have plans to build mechanisms to provide definitions to our environment and plans to evaluate the utility of our new system with programmers.

References

[1] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. In *Proc. of the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 681-682. 2006.

[2] J. P. Bigham, C. Jayant, H. Ji, G. Little, A. Miller, R. C. Miller, R. Miller, A. Tatarowicz, B. White, S. White, and T. Yeh. VizWiz: Nearly Real-time Answers to Visual Questions. In *Proc. of the 23rd Annual ACM Symposium on User Interfaces Software and Technology*, UIST '10, 333-342. 2010

[3] M. Goldman, G. Little, and R. C. Miller. Collabode: Collaborative Coding in the Browser. In *Proc. of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, 65-68, 2011.

[4] A. L. Lemos, S. K. Bajracharya, J. Ossher. CodeGenie: a Tool for Test-Driven Source Code Search. In *Proc. of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, 917-918. 2007.

[5] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers. Calcite: Completing Code Completion for Constructors using Crowds. *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '10, 15-22. 2010.

[6] D. Vardoulakis and O. Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. In *Proc. of the 19th European Symposium on Programming*, ESOP '10, 570-589. 2010.