Specifying What to Search For

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912 USA
spr@cs.brown.edu

Abstract

In this position paper we look at the problem of letting the programmer specify what they want to search for. We discuss current approaches and their problems. We propose a semantics-based approach and describe the steps we have taken and the many open questions remaining.

1. Motivation

One of the first things a programmer should do when writing new code is to find existing, working code with the same functionality, and reuse as much of that code as possible. With the large amount of open-source code available and the fact that most applications are not completely novel, one could imagine that a significant amount of the code that is being written today has been written before in some form, and much of it is available in an open-source repository.

This is the type of motivation given for code search. The emphasis here is on enabling reuse, avoiding writing what has been written before, making effective use of open source software, speeding up development, and producing higher quality software systems.

Code search, if it is going to achieve these ends, has to be substantially different from traditional web search. In particular it has to be designed and implemented so that:

- It is easier to use the results returned from the search engine rather than creating the code from scratch. Reuse should be relatively simple.
- The results returned must do what the programmer wants them to do. They shouldn't be an approximation or just related.
- The programmer must be able to use the resultant code. This means that the code must conform to **all** the requirements for the potential application.
- Programmers must be able to find what they are looking for. It is much trickier to determine this by looking at the summary or even the code itself than it is for textual information.

Achieving these goals should be the aim of the code search community. This involves addressing several problems. The first is getting access to all the appropriate open source (and other) code. The second involves organizing this data set and providing an appropriate query mechanism. The last is to provide the appropriate user interface for code search.

The first two of these problems, while difficult, have proven tractable, as can be seen in the various existing solutions. The difficulties lie in the fact that most code is not designed primarily for human readability and other factors such as program structure can be important aspects of the search. The last problem however, attempting to make the code search interface meet the programmer's needs, is the one that I find the most interesting and the least tractable.

2. Current Solutions

Current interfaces to code search take three principle forms. The first and most prevalent approach utilizes keywords. Here one depends on matching the user's vocabulary with that of the original programmer. It also requires finding keywords that are unique enough to actually identify the code in mind. Keyword searches typically yield lots of unevaluated results. The problem with this is that programmers have to read each instance of returned code, attempt to understand what it does, and then determine if it meets their requirements.

My experience with keyword based code search is that it creates a lot of work for the programmer. For any simple piece of code, the effort required to analyze the returned results is more than the effort that would have been required to write the code in the first place. Also, it can be a significant amount of work converting the code into the programmer's target framework after finding an appropriate method or class. This makes reuse based on code search difficult and unappealing.

The second approach is to use information about program structure, for example method signatures or expected loop structures [1-3]. These can be combined with keyword search. While the additional information can be helpful, its applicability is relatively limited. When searching for a method, the program structure used in the algorithm is generally unknown or irrelevant to the search. Moreover signatures are only relevant if the code base being searched and the user's code share the same environment or if the programmer happens to be looking for a code snippet that is independent of their coding environment, a rarity in practice.

The third approach used today is to let the programmer specify test cases possibly accompanied by a set of keywords. When keywords are not provided, the test cases themselves can be used to derive keywords.

This approach is closer to what is needed for effective code search. Here the programmer is attempting to define what they are looking for, and define it in a way that the system might understand. Test cases can go beyond simple functionality and check additional code properties such as security and error handling. They also can define the context in which the code has to work.

There are several problems here. The first is that test cases can be difficult to write. In many cases the amount of code required for testing can exceed the amount of code being retrieved. When part of test-driven development, this might be an acceptable cost, but otherwise it can be a deterrent [4]. The second problem is that the problem might be one where test cases are difficult to define because the solution is not well defined. An example I've run across here is a method that determines what countries a news article is about, returning a probability vector where there is no real "correct" answer. The third problem is that, as the test cases and methods become more specific and constrain the set of possible solutions, the likelihood of finding any code that does exactly what the programmer wants becomes diminishingly small.

3. Examples

To make this analysis more concrete, consider some examples of code search I have attempted.

My first case was relatively simple. I needed to get the text from an HTML page using a Java class or method. The difficulty was that in my application white space in the text was relevant and I needed to understand one type of tag. Moreover, I wasn't particularly fussy about an interface; I could work equally well with a parser that works with callbacks or a parser that generated a tree of nodes.

Searching for the keywords "html parser": tends to yield lots of results, most of which are not relevant. Even when I found an implementation of a HTML parser this way, I still had to determine if that parser collapsed white space or not. Because the parsers are relatively sophisticated components, this involved significant work. Often, it seemed easier to do this dynamically, running the parser on a sample file and seeing what it returned. However, this required writing a framework that instantiated and used the parser first, and the framework was essentially different for each of the available parsers. This relates to the problem of using test cases in this situation; because the parsers have different interfaces, selecting a single test harness will essentially limit or eliminate what might be viable parsers from being considered. Once I select a particular callback framework or DOM model, it is very unlikely that I will find more than one parser and that parser is unlikely to do what I need with spaces.

In another example I needed to compute a topographical ordering for the set of nodes in a graph. The problem here is that I have my own graph structure. Code search finds lots of code that does topological sort. The results fall basically into two categories. First are those that are embedded in a graph class (or actually a whole hierarchy of graph classes). Of course the graph class here is significantly different than mine, and it looks like attempting to reuse the code is going to be more work than writing topological sort from scratch. Second are those that take sets of nodes and edges as arguments. Here, the problem was that the sets contain objects representing nodes or edges and these objects were quite different from the objects in my graph model. A further complication arose in understanding the behavior of the different algorithms when the graph was cyclic and there was no topological ordering.

Finally, I needed code that would find a least squares solution to a system of linear equations. Here code search helped to identify a relevant library that could be used almost directly. The real problem arose when we noted that we needed to add constraints so that all the resultant values were non-negative. Searching for this was not productive until it was pointed out that this was an instance of a quadratic programming problem. Searching for quadratic programming yielded solutions, but none were close to the desired specifications. To use them I had to write significant code that would convert the system of linear equations into a corresponding quadratic, set up the constraints, and then map the resultant values into the actual solution.

These examples only touch the surface of what programmers face when they attempt to use code search to facilitate reuse. In each case the programmer had a good idea of what he was searching for, however it was either difficult to specify or the results required significant work in order to be usable or both. Before code search becomes usable, these problems will need to be addressed.

4. Semantics-Based Search

In order for code search to be effective, the programmer needs to be able to specify what they are interested in finding. They need to state what they want the identified code to do functionally, where it has to fit in, and what constraints (e.g. performance, error handling, security, privacy, synchronization) they want to impose on it.

This information is precisely a description of the semantics of the code where semantics is taken in a broad sense. It might include a formal description of the behavior of the code (formal semantics), a high level description of the code in terms of keywords or text (informal semantics), pseudo code for the function (structural semantics), test cases (semi-formal semantics), security and privacy limitations, error handling (formal or informally), recovery information, performance requirements, synchronization requirements, the context where the code will be used, and the desired coding style and conventions.

The programmer generally knows a subset of this information and needs to convey it to the code search tool in some manner. The goal of a code search interface should then be to encourage the specification of as much of this information as possible.

But even if the programmer could be precise here, it would not be enough. As programmers become more precise as to what they want, the odds of identifying code that exactly matches their specifications approaches zero. Moreover, many of the problems, for example signatures and use of the programmer's environment or context, need to be addressed in order to correctly interpret other parts of the specifications such as test cases.

Thus, an effective code search tool has to automate much of the way that the programmer would want to use the result of the search. In particular it has to automatically refactor the located code to fit the programmer's environment. Examples of the transformations or refactorings that might be done here include: adapting the signature of the identified code to the programmer's specification, eliminating unnecessary functionality, isolating the desired functionality from the middle of an existing routine, bringing in additional classes and methods that are required to make the code functional, modifying the code to use existing support classes rather than their own, converting the coding style to meet the current project's requirements, adapting the code to fit into an existing class, converting a class-based implementation into a method-based one (or vice versa), generalizing or specializing parameter and return types, changing the way errors are reported, and adding or removing logging or debugging statements.

5. The S^6 Project

The goal of our research is to create an appropriate front end for code search that allows semantic specification of what to search for and automatically performs the appropriate transformations that programmers would otherwise have to do to make the identified code usable in their application.

Our approach to date shows that this might be an achievable goal, but that significant work still needs to be done [5]. Our front end concentrates on allowing the user to specify test cases quickly and effectively. It requires keywords as a starting point for the search. In addition, it allows contracts to be defined for each method and the code to be run in a restricted Java security context. A front end to the system is available at http://conifer.cs.brown.edu/s6. Source code for the system is available from the author.

While this is a start, much more needs to be done. Keywords are sometimes hard to find or select. Complex test cases, for example test cases that require the user to write code, are not supported by the front end. The Java security model is limited and does little about privacy concerns. Contracts are only checked when running test cases, not statically against the code. Performance can only be evaluated in terms of the performance on the test cases (which are generally too simple for this purpose), and then only as part of sorting the output. No information about the structure of the target code is used, nor is anything done about synchronization. There is only limited support currently for handling user context such as existing classes and methods.

Our code search engine also applies a suite of transformations to attempt to adapt the code to meet the programmers' requirements. While this is necessary to accommodate test cases, and has been quite effective in simpler cases, much still needs to be done. The current transformations do not take into account the target environment or attempt to map the implicit environment of the identified code to the target environment. This is where most of the work of adapting identified code actually occurs. In addition, it only has a limited set of type mappings and does not support many class-level transformations. While much of this is easy to add in theory, in practice it is difficult because of the exponential number of possible mappings that can arise.

6. Challenges

Based on our experiences, we can identify several challenges that need to be met before code search will be really practical, challenges that we hope will be taken up by the code search community.

The first is finding a practical approach to letting the programmer specify the semantics of what they are looking for. This approach has to handle of the complexities of real world problems and situations. The approach will have to be an amalgam, since no one specification method or technique will work for all (or even a majority of) cases.

The second is developing the underlying code representations and search mechanisms that will support such a front end. Keywords alone are not sufficient. Signatures or program structures are helpful, but have to be flexible enough to accommodate the possible transformations. One should be able to search not only based on the target code, but also on the desired and original contexts.

The third is finding a suitably broad set of transformations that mimic what programmers do when they adapt the result of code search to fit their applications, and then automating these transformations in a practical way. The main problem here is making this process intelligent, avoiding the potentially exponential number of results, and integrating transforms with the back end and the semantic specifications.

7. Acknowledgements

This work is supported by the National Science Foundation through grant CCR0613162.

8. References

1. Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," *Proc. OOPSLA 2006*, pp. 682-682 (October 2006).

2. Andrew Begel, "Codifier: a programmer-centric search user interface," *Workshop on Human-Coputer Interaction and Information Retrieval*, (October 2007).

3. Raphael Hoffmann and James Fogarty, "Assieme: finding and leveraging implicit references in a web search interface for programmers," *Proc. UIST 2007*, pp. 13-22 (October 2007).

4. Otavio Lemos, Sushil Bajracharya, Joel Ossher, Ricardo Morla, Paulo Masiero, Pierre Baldi, and Cristina Lopes, "CodeGenie: using test-cases to search and reuse source code," *ASE '07*, pp. 525-526 (November 2007).

5. Steven P. Reiss, "Semantics-based code search," *ICSE 2009*, (May 2009).