Based on the paper "Semantic Code Browsing" by Isabel García-Contreras José F. Morales Manuel V. Hermenegildo

SEMANTIC CODE BROWSING - A REPORT AND IMPLEMENTATION

Jakub Wlodek
Department of Computer Science
Stony Brook University
Stony Brook, NY, 11783

https://github.com/jwlodek/Semantic-Code-Browsing

December 18, 2019

ABSTRACT

In most modern code browsing and code analysis approaches, the use of keywords in comments, variables, and function names often means that deriving meaning from code automatically is dependant on the quality of its documentation, and yet code with strong documentation is often the one least likely to require analysis. The paper "Semantic Code Browsing" by García-Contreras et. al. attempts to present a solution to these issues in the form of analyzing code meaning semantically rather than by using the naive approach with keyword matching, or even regular expression matching. In this report, I will explain their methodology and results, and discuss how I used Semantic Code Browsing to try to extract the meaning of a program written in several languages into a common one, and then implement an SQL-like query language for interacting with the Semantically Analyzed Program.

***K*eywords** Semantic Analysis · Code Browsing · Implementation

1 Introduction

In this section I will give a brief overview of the target paper, what it aims to accomplish, and a quick rundown of what I plan to implement from the paper, as well as what I would wish to extend and add to García-Contreras et. al.'s research.

1.1 What is Code Browsing?

Code browsing refers to the process of searching through source code for code relating to particular feature or implementation. This process can be as simple as finding sections of code that contain memory allocation, to identifying where in code the left side of a binary tree is accessed. Traditionally, this process has relied on variable names and in-source documentation. Most code browsing applications today search for keyword matches or some basic variable or function names. As you may imagine, this approach has some core limitations: if code is not documented correctly (such as comments copy pasted with functions that are then modified), then the code browser will return incorrect or inconsistent results. Additionally, if documentation is lacking, or function and variable names are not verbose enough, the modern code browser will not be able to assign any meaning to the attached code, simply skipping it, giving us the possibility of missing correct results. Another common approach to code browsing that allows for more robust solutions adds the use of regular expression based matching over simple keyword finding (see: Google Code Search), though even these solutions will break down if the documentation surrounding the code is not sufficient.

1.2 Semantic Code Browsing

In response to the limitations described in section 1.1 inherent in modern code browsing applications, García-Contreras et. al. chose to construct a new approach to analyzing code meaning, relying on the semantic structure of code to deduce its meaning rather than keywords or regular expressions. As a consequence of using code structure to deduce meaning, similarly to how Prolog variables can be matched at run time, variable or function names do not affect the accuracy of the browser, and comments are ignored entirely. For example, to a Semantic code browser the statements:

```
//Increment the counter by 1
item_counter = item_counter + 1
```

and

```
x = x + 1
```

are identical, despite one having descriptive documentation and variable names, and the other having a generic variable name. A semantic code browser can identify both statements as an increment by one on a generic variable X. Then, based on other context that comes either before or after the statement, it can assign some form of meaning to X.

1.3 What does the paper accomplish?

In their paper García-Contreras et. al. create a simple prototype program for performing Semantic Code Browsing in Prolog. the input and output formats are fairly limited, but the demo does show the benefits of their approach compared to other modern solutions. The paper uses a Prolog-like syntax for queries into a particular module, and relies on a combination of a find target and assertions defined by the user to match against existing predicates.

1.4 Extending Semantic Code Browsing

In this paper, I demonstrate creating a similar simple implementation as in García-Contreras et. al.'s paper, as well as some extensions to the idea.

1.4.1 Other Languages

The first extension I would make to the demo is to include support for an additional languages - namely C and Python. The program should be able to identify similar statements that accomplish the same thing in multiple languages, and represent this meaning in the same format internally. It should be simple to extend the resulting project with other languages, and the entirety of the internal data structure should be based on inheritance between more general program representation classes.

1.4.2 SQL-Like Query Syntax

Rather than limit the Semantic-Code-Browsing to Prolog program syntax, I would like to extend the implementation to allow for an SQL-like query language that can be used to ask questions about loaded programs. This language will be accessible through a custom shell that supports loading programs, displaying information, and processing queries.

2 Implementation Theory

In this section I will discuss some of the decisions I made regarding the implementation of Semantic Code Browsing internally, as well as some of the reasoning behind how the extensions to the project were constructed.

2.1 The Program Representation

Taking a page out of the structure of compilers and their AST representation of programs, I decided to model each loaded program internally as an instance of a *ProgramRepresentation* object. For each programming language, a subclass of this one will be created to house all language-unique features, such as predicates in Prolog, or lambda functions in Python. Each instance of such a representation will be associated with a set of programmatic terms, which will represent common blocks in the program. In C, for example, these would be Functions. Each term that can be found in the program is parsed into a subclass of the general *Term* class, and this includes functions, predicates, loops, conditionals, etc.

The instances of *Term* that are parsed from the program are assigned to their parent, creating a tree-like structure, with the top node being the *ProgramRepresentation*, followed by *Function* or *Predicate* objects, and then the rest of the terms. Each *Term* object and any that inherit from it will include some basic information that will be used during the Semantic Code Browsing queries. Semantic information - not just the term names but the type, usage and location will be stored. Once the *ProgramRepresentation* is generated from a given program, it is passed on to the Query Shell for user analysis.

2.2 Program Parsing

Similarly to how the *ProgramRepresentation* will be handled as a series of sub-classes for each language, the *ProgramParser* will provide some common methods that are used by all languages, but requires several methods to be implemented in sub-classes for each language. Depending on the file extension of the program given to the parser, an instance of the appropriate subclass will be created, and the parser will generate an instance of the appropriate *ProgramRepresentation*.

Once the parser generates a representation for the given input program, to the front-end query shell, the program should be largely language-agnostic, barring differences in program philosophy such as the differences between a logic programming langugae and an imperative one. Any constructs that can be labelled as common (ex. Conditionals) should be treated similarly.

2.3 The Query Engine

The next step above the `ProgramParser` is the `QueryEngine`. This is a class that handles all of the query processing for the system. It takes as an input the string of the query, and then converts it into an `SBCQuery` object, which stores all of the assertions passed in, as well as search item type, and search function/predicate arity (if necessary). The assertions and arity are then compared against the program representation for the program that is currently loaded.

Any instances matching the assertions as well as the search type and arity are placed into a `SCBQueryResult` object and piped back into the shell.

2.4 The SCB Query Shell

The SCB Query Shell is the top level shell used to query loaded semantic representations of programs. The desire is for it to have similar syntax to queries constructed for typical relational databases like MySQL. The hope is that if such a similar syntax exists, such a Semantic Code Browser could prove a useful tool particularly for searching through large, poorly-commented codebases.

The shell will also have support for a few non-query commands, such as printing program or specific function/predicate info, loading new programs, and others.

3 Implementation Details

In this section I will describe some of the changes made during the implementation phase of the project as well as describe how, in the current implementation, query format is handled.

3.1 Parser Limitations

Parsing programs into a common data structure format is a focus of compilers, that often use Abstract Syntax Trees (ASTs) for their program interpretation. The ability for a parser to achieve near perfect coverage of a language standard is a challenging endeavour, and as a result, I focused my efforts on creating a parser that could glean important program information in a quick pass. This also made implementing the parser much easier when adding new languages, though it meant that certain aspects of programs would go unaccounted for. This is probably one of the areas that can see the most improvement in future work.

3.2 Query Format

The format that queries entered into the query shell take is as follows:

- Each query begins with the `find` keyword.
- This is then followed by a search type, typically `function` or `predicate` with an optional arity marker.

- . Then, optionally, assertions can be added by appending the `where` keyword followed by assertions of the following types. Each assertion starts with a keyword and colon, followed by certain search terms. Assertions can be glued together using `or` and `and` keywords.
  1. `inputs:INPUT_1_TYPE,INPUT_2_TYPE...`
  2. `bodycontains:SEARCHTYPE (SEARCHTYPE: function, loop, conditional)`
  3. `returns:RETURN_TYPE`
- All queries end with a period.

Below are some examples of valid queries on an input program:

```
SCB Query > find function/2 where inputs:int,char* or returns:void.
SCB Query > find predicate.
SCB Query > find predicate where bodycontains:function and inputs:function/3,var.
SCB Query > find function where bodycontains:loop and returns:int*.
```

4 Results

In order to test my Semantic Code Browser I wrote several very simple programs in Prolog and in C, and I also found a larger C program that I had written some time ago for CSE 320. I tested loading these into the SCB Query Shell, and performing some queries of different kinds, with some results displayed below.

4.1 Example 1: Simple Prolog Program

My first example had me load the following program:

```
parent(a, b).
male(a).
father(X, Y) :- parent(X, Y), male(X).
```

I then performed a general program info query, which listed all of the predicates detected in the program. The `parent` and `male` predicates are described as terms of type atom, and the `father` predicate takes terms of type variable, and has a body consisting of 2 predicate calls.

```
PS > py .\browse_code.py .\examples\simple.pl
+-------------------------------------------------------------+
| Semantic-Code-Browsing                                      |
|-------------------------------------------------------------|
| SCB Query Shell v0.0.1                                      |
| Prolog Single-File Program - .\examples\simple.pl           |
| Jakub Wlodek - CSE 505 Final Project                        |
+-------------------------------------------------------------+

Welcome to the Semantic Code Browsing Query Shell. Please enter a
valid query for the given program, or see the documentation
and/or examples for instructions on constructing valid queries.

SCB Query > program info.

Prolog Representation w/ 3 predicates
Prolog Program represented as series of predicates.
Predicates:
Predicate Name: parent
Term name: parent
Function of arity 2
Terms:
- Variable Name: a, Expected Type: atom
```

```
- Variable Name: b, Expected Type: atom
Predicate Name: male
Term name: male
Function of arity 1
Terms:
- Variable Name: a, Expected Type: atom
Predicate Name: father
Term name: father
Function of arity 2
Terms:
- Variable Name: X, Expected Type: var
- Variable Name: Y, Expected Type: var
Body of predicate:
- Term name: parent
Function of arity 2
Terms:
- Variable Name: X, Expected Type: var
- Variable Name: Y, Expected Type: var
- Term name: male
Function of arity 1
Terms:
- Variable Name: X, Expected Type: var
```

Next, I queried the program for predicates of arity 2, that take take in an 2 atoms as inputs:

```
SCB Query > find predicate/2 where inputs:atom,atom.

Searching for predicate/2 that satisfy assertions:
 - Assertion: inputs -> ['atom', 'atom']

Query: find predicate/2 where inputs:atom,atom.

 > Predicate Name: parent Predicate arity: 2

1 Matching result(s) found.
```

As expected, the `parent` predicate was the only one that matched.

4.2 Example 2: Mid-Sized C Program

In the next example, I loaded the mid-size C program I had taken from my CSE 320 coursework, which represented some school information as a linked list. I first queried it for all loaded functions (at the top level of the program representation):

```
SCB Query > find function.

Searching for function that satisfy assertions:
No arity specified of target function or predicate.

Query: find function.

 > Function Name: check_id_repeat Function arity: 2
 > Function Name: search_by_id Function arity: 2
 > Function Name: search_by_name Function arity: 2
 > Function Name: search_by_major Function arity: 2
 > Function Name: fix_names Function arity: 1
 > Function Name: fix_majors Function arity: 1
 > Function Name: print_student Function arity: 3
```

```
 > Function Name: print_all_students Function arity: 3
 > Function Name: free_list Function arity: 1

9 Matching result(s) found.
```

The 9 functions in the example were found, and their arity listed. Note that despite the fact that in the C language, the discussion of function arity is rarely brought up, for my implementation that concept is used across all languages for the benefit of code reuse. Because the internal program representation is based largely on the same set of classes regardless of program input language, the program semantics are sometimes labelled something that traditionally fits a different language, as is the case with arity.

Next, assume that in our school system program, we need to find the functions that return student IDs, or the functions that return student records. We know that IDs are stored as `int` and records are in `struct student_records`. Thus, we construct a query for functions that return these:

```
SCB Query > find function where returns:int or returns:struct student_records*.

Searching for function that satisfy assertions:
 - Assertion: returns -> ['int']
 - Assertion: returns -> ['struct student_records*']
No arity specified of target function or predicate.

Query: find function where returns:int or returns:struct student_records*.

 > Function Name: check_id_repeat Function arity: 2 Returns: int
 > Function Name: search_by_id Function arity: 2 Returns: struct student_records*

2 Matching result(s) found.
```

In this case I simply construct two assertions, both on the function's return type, and I `or` them together. In a more practical application, I can use the describe command for a given function to list more information about it, say the `check_id_repeat`:

```
SCB Query > describe check_id_repeat.

Function Name: check_id_repeat
Term name: check_id_repeat
Function of arity 2
Terms:
- Variable Name: id, Expected Type: int
- Variable Name: list, Expected Type: struct student_list*
Body of function:
- Conditional of type if

1 Matching result(s) found.
```

4.3 Result Limitations

While the results of the Semantic Code Browsing implementation are largely positive, certain limitaions remain that could possibly resolved with some future work.

Due to the simplicity of the current `ProgramParser`, not all programs will be parsed correctly into program representations, as not all language features are supported. Features that cannot be identified by the parser are simply ignored in the representation which can lead to some unfortunate confusion, as a function known to be in the program is not query-able. As will be discussed in the Conclusions section, it may be interesting to see if it would be possible to link the Query Engine and frontend shell to a compiler front end, where the compiler pre-processes programs into an AST, and the query engine uses this instead of the internal parser for the Semantic Browsing.

Another potential improvement for the implementation is several more levels of polish; bugs, particularly ones having to do with illegal program parsing, are too common for the current implementation to be used for more than research, however, the initial results demonstrate how this approach could be useful for large scale program analysis without the aid of in-code documentation.

5 Conclusion and Future Work

In this report I have demonstrated the development of a Semantic Code Browsing shell with SQL-like syntax for queries that allows users to quickly search for functions and predicates in their programs given certain assertions. The system is modular enough to the point that implementing support for different languages is fairly straightforward, through the use of class inheritance and polymorphism.

There are numerous improvements to the system that future work could yield particularly in the flexibility of the querying system. Due to time constraints, I was unable to implement the full breadth of query-able values, which could ultimately allow for queries that search throughout the body of a program for particular code snippets and more.

While the internal representation was designed to be modular, as is the case with compilers (where there needs to be a different front end for most every language), certain language specific features needed to be hastily added to the parser and representation in order to make the code browsing work, at the cost of program complexity and readability. Linking the SCB Query Shell with the front end of a compiler, and using the actual AST that is guaranteed near perfect language parsing would likely also be a beneficial assignment, and would give far more information about a program than the fairly rudimentary parser included here.

References

[1] Isabel Garcia-Contreras and José F. Morales and Manuel V. Hermenegildo Semantic Code Browsing and Analysis In *DBLP:journals/corr/Garcia-Contreras16 Mon, 13 Aug 2018*