The SourceGraph Program

Ivan Lazar Miljenovic

School of Mathematics and Physics
The University of Queensland,
Queensland, Australia
Ivan.Miljenovic@gmail.com

Abstract

As software has increased in size and complexity, a range of tools has been developed to assist programmers in analysing the structure of their code. One of the key concepts used for such analysis is the concept of a *call graph*, which is used to represent which entities in a code base call other entities. However, most tools which use call graphs are limited to either visualisation for documentation purposes (such as Doxygen) or for dynamic analysis to find areas to optimise in the software using profiling tools such as gprof.

SourceGraph is a new tool which takes a different approach to software analysis using call graphs, for projects written in Haskell. It creates a static call graph directly from the source code and then uses it to perform static analysis using graph-theoretic techniques with the aim of helping the programmer understand how the different parts of their program interact with each other. Whilst still a work in progress, it can already be used to find possible problems in the code base such as unreachable areas and cycles or cliques in the function calls as well as other useful information. SourceGraph thus provides programmers not only with various ways of visualising their software, but helps them to understand what their code is doing and how to improve it.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; G.2.3 [*Discrete Mathematics*]: Applications

General Terms Algorithms, Documentation, Languages

Keywords software analysis, static analysis, call graph, graph theory, Haskell

1. Introduction

Call graphs have been used since the 1970s [15] to represent and model the relationships between different entities in software. However, usage of call graphs is usually limited to either visual representation of the software or minimal analysis using profiling information to improve software efficiency.

This paper describes a new tool *SourceGraph* that takes a different approach: it utilises call graphs to statically analyse software written in the programming language *Haskell* using graph-theoretic techniques. Rather than trying to make code faster, its purpose is to assist the programmer in improving the overall quality of their software by helping them visualise and understand how the entities within their software interact.

PEPM'10 January 18–19, 2010, Madrid, Spain

Reprinted from PEPM'10, Proceedings of the 2010 ACM SIGPLAN Workshop on partial Evaluation and Program Manipulation, January 18–19, 2010, Madrid, Spain, pp. 151–154.

1.1 Similar Work

Various approaches have already been taken in considering how to analyse or understand software from graph and visualisation perspectives including the seminal work by Ball and Eick [1] (which explicitly uses non-graph techniques for visualising software, stating that they are "often too busy and cluttered to interpret"). Furthermore, several other tools exist specifically for analysing Haskell code such as HaSlicer [14] and Programatica [6]. Whilst these tools may use call graphs for slicing Haskell software (which is the main goal of HaSlicer), SourceGraph uses solely call graphs for various forms of software analysis (note that it does not yet have any slicing analysis). Another major advantage SourceGraph has over these is that it is a fully functional (albeit currently limited in some ways; see Section 5) utility that works with current versions of libraries and compilers: Programatica is no longer actively maintained, and HaSlicer never seemed to have moved beyond the proof-of-concept stage.

The only similar work that can be found in the literature is by Narayan, Gopinath and Sridhar [10]. Rather than using call graphs to analyse a specific project however, their focus is to find overlying topological similarities between call graphs from projects written in different programming languages. Furthermore, the call graphs for their OCaml and Haskell projects were generated based upon compiler output rather than directly from the sources, and as such might not be fully indicative of the code as written (e.g. no consideration of type classes, etc.).

2. Call Graphs

A call graph is a directed multigraph (that is, a directed graph which allows multiple edges between vertices) with the set of vertices V consisting of the entities within the software and a multiset of directed entity relationships E. Traditionally, the entities are the functions/methods/routines (depending on the terminology of the language being used) within the source code, and a relationship (e_1, e_2) indicates that entity e_1 calls entity e_2 (that is, the source code of e_1 has at least one reference to e_2). The relationships can be obtained using one of two methods:

Statically: Parse the source code to obtain all possible relationships.

Dynamically: Use profiling information to obtain the actual relationships used in a sample usage of the application.

For this paper, we will only be considering static call graphs, as SourceGraph is used to analyse *all* entity relationships, not just those utilised in a single run of a program.

3. SourceGraph

The SourceGraph program originally came about as a sample application of the *Graphalyze* library [9]. It uses the *haskell-src-exts* [3] library to parse Haskell code and then Graphalyze to convert the parsed code into a call graph and analyse it before producing an analysis report.

3.1 Haskell Call Graphs

Haskell [11] is a purely functional programming language (with basic entities referred to as *functions*; to simplify matters, we shall also use this term to refer to constants). Haskell projects are usually split up into several logical modules, with one module located in each file. There are however certain aspects of Haskell syntax that are *not* "basic entities" and thus require special consideration:

- Type Classes [16] provide a means of *ad-hoc* polymorphism. All data types that are part of a type class must provide implementations of a set of specified class *methods* (where a methods is function that is a valid operation on that type class). When we make a data type part of a class, we say that we are making that data type an *instance* of that type class. Type classes are also able to define default declarations of some or all methods that a particular instance is able to selectively use or override (for example, in the `Eq` type class, both equality and inequality have default declarations that are the negation of the other; instances of `Eq` are then able to only override one of these and utilise the default declaration of the other).
- Haskell's data constructors are also functions, distinguished from "normal" functions by starting with an upper-case character. Data types in Haskell can also have more than one constructor.
- Data types in Haskell may have *selector functions* defined for specific constructors if using record syntax. These provide labelled fields, with the functions themselves serving as predefined accessor functions and a method of changing that particular field in a data structure. As long as they have the same type, multiple constructors of a data type can have the same named field.

As such, SourceGraph makes the following adaptations to "standard" call graphs. These adaptations are necessary to avoid giving the impression that many call graphs were composed of several connected components rather than just one (as many class functions are articulation points in the call graph) and to include as many entities and relationships from the source code as possible into the call graph (early versions of SourceGraph did not take any of these into account, which resulted in sub-optimal call graphs and analysis results).

3.1.1 Classified Entity Types

Entities in a call graph can be one of six different types. These are distinguished from each other to make the different types of entities more visible in the call graphs, and to group them together if necessary (see Section 3.1.2).

1. A class method;
2. A default declaration of a class method;
3. An implementation of a class method within a class instance;
4. A data constructor;
5. A selector function;
6. A "normal" function.

3.1.2 Categorised Entity Types

For cases where it is advisable to treat type classes, data structures, etc. as a single unit, there are three further entity types. Such a case is the alternate module grouping analysis in Section 4; as all methods for a type class must be defined together in the same module, they are combined into a "type class" entity.

1. A type class (including the default declarations);
2. A data type (including constructors and selector functions);
3. An instance of a class for a particular data type.

3.1.3 Classified Entity Relationships

Rather than just recording relationships between entities, they are categorised based upon what the relationship from e_1 to e_2 is. This makes it easier to distinguish multiple relationships between several entities; for example, the instance of a method for a particular data type may use the same method for other types (for example, equality on lists requires the equality method for the list items).

1. e_1 is a default declaration of the method e_2;
2. e_1 is an instance of the e_2 method for a particular type;
3. e_1 is a selector function for the constructor e_2;
4. e_1 is a function that calls e_2.

3.1.4 External Entities

Call graphs typically only consider entities from either the file or the entire code base that is being examined. However, this may have the effect of artificially splitting up a Haskell call graph into several connected components due to the high usage of type classes. As such, SourceGraph attempts to determine which external entities are class methods and includes them as "virtual" entities to improve the coverage of the call graph.

3.2 Treatment of Class Methods

With these specific entity types, there remains the question of how to treat relationships involving a function calling a class method. The approach taken by SourceGraph is that each instance of a class function (whether for a default instance or an instance for a specific data type) has a virtual function created which represents that particular instance; this helps to distinguish between relationships from an instance of a class function which class function it is instantiating and which other class functions it may be calling within its definition. Whenever a function calls a class function, it is the actual class function entity that is represented in the call graph rather than one of the virtual instantiating functions.

4. Analyses Performed by SourceGraph

SourceGraph analyses software on three levels:

1. The entire code base;
2. The imports between modules (that is, which Haskell source files require other Haskell source files);
3. Each module on its own.

For each level, various analysis algorithms are applied. The current algorithms in the latest release publicly available at time of writing (0.5.5.0) that SourceGraph applies to each level can be found in Table 1; it is planned to include several more in the near future. Note that for most of these analyses (e.g. connected components), the analysis results are only shown if they are "interesting" (that is, they show extra information: if there is only one connected component, then there is no reason to include it within tha analysis report). A brief description of each algorithm and how they may be useful to programmers follows.

Algorithm	Entire Code Base	Module Imports	Per-Module
Graph Visualisation	✔	✔	✔
Graph Core Visualisation	✔		✔
Module Visualisation	✔		
Export Analysis	✔	✔	✔
Connected Components	✔	✔	✔
Clique Finding	✔[a]		✔
Cycle Finding		✔	
Clique-less Cycle Finding	✔[a]		✔
Chain Finding	✔[a]	✔	✔

[a] When analysing the entire code base, only cross-module cliques, cycles and chains are considered.

Table 1: Algorithms used to analyse Haskell code

Graph Visualisation: When visualising the call graph, the programmer is aided by the use of various colours and symbols to indicate what different parts of the graph denote: for example, class functions are differentiated from instance functions by different symbols, and exported entities are shaded using a different colour from non-exported entities. Furthermore, class, instance and data entities are clustered together appropriately.

Graph Core Visualisation: The *core* of the call graph is obtained by repeatedly removing root and leaf nodes. This helps visualise the "centre" of the call graph where the entities interact with each other the most.

Module Visualisation: Entities are clustered together into the modules they are defined in and visualised. An alternative module clustering constructed using the *Chinese Whispers* algorithm [2] is also provided. Note that to avoid splitting up class, instance and data declarations (which *must* be defined together), the relevant entities are compressed down into the special entity types listed in Section 3.1. The Chinese Whispers algorithm tends to over-cluster the results, but has the advantage over most other clustering algorithms in that it requires no external information [4].

Export Analysis: Compare the entities actually exported with those that are roots of the call graph. The entities that are roots but not exported are unreachable areas of the code base, and can be safely removed (assuming, of course, that they are not used in an alternate library or executable that SourceGraph is not aware of).

Connected Components: If a module/library contains more than one component, then it can be safely split up into separate modules/libraries. There are times that this is not advisable, however, such as for utility modules which contain various functions needed throughout the code base (e.g. a module of common mathematics routines).

Cliques and Cycles: Whilst Haskell's functional nature does lead programmers to more readily make use of recursion and co-recursion, if this is taken to an extreme level (e.g. five functions in a cycle or clique) then this could indicate a possible logic/understanding problem in the code. For modules, *any* cycle leads to possible problems (most Haskell implementations provide a mechanism where two modules can recursively import each other, which often requires additional instructions to the compiler).

Chains: A chain is used to indicate a sequence of function calls between entities where these function calls are the only ones present for the (interior) entities. More formally, a chain is a maximal path $e_1, e_2, \ldots, e_n$ where for each interior entity e_i (with $1 < i < n$), e_{i-1} is the only entity that calls e_i and e_i is the only entity that calls e_{i+1}. Note that the first condition must also hold for e_n and the second condition for e_1. These chains indicate entities that can be safely collapsed together into one big entity (exported entities are not considered for interior entities when constructing chains).

4.1 Sample Results

It is unfortunately not possible to include a sample analysis report produced by SourceGraph in this paper due to size constraints: the generated reports are typically quite long and any non-trivial visualisation is too large to include without suffering degradation of image quality. Sample reports are available online at `http://code.haskell.org/~ivanm/Sample_SourceGraph/SampleReports.html`.

5. Known Limitations of SourceGraph

There are several known problems/limitations currently in SourceGraph. These can be classified as those dealing with parsing the actual source code and those dealing with the actual usage of SourceGraph.

Parsing Problems

The de-facto implementation of Haskell is the Glorious Glasgow Haskell Compiler (GHC) [13], which comes with a number of extensions to the Haskell98 standard [12]. For the most part these are taken into account when parsing the source code; the following are extensions not yet included in the call graph (as there is no obvious way of doing so):

- Code that requires pre-processing (usually using the C Pre-Processor);
- HaRP (embedded regular expressions in Haskell code);
- Haskell Source with XML (embedded XML inside Haskell code);
- Template Haskell (compile-time meta-programming for Haskell);
- Data Family instances;
- Foreign Function Interface imports and exports.

There are several other extension-related parsing problems of a lesser nature. For the most part, these extensions are not considered when constructing the call graph as it has not yet been determined *how* to include them as they behave (and are parsed) differently from standard Haskell expressions/extensions (and the author is not as familiar with how they operate). This might be alleviated somewhat by using GHC's internal parser, but this is not as extensible and is tied to the version of GHC used (as a comparison, SourceGraph has successfully compiled with no changes using at least three major releases of haskell-src-exts as its API is relatively stable).

Furthermore, SourceGraph has trouble dealing with entities imported from external modules; in particular with class functions: it will attempt to classify imported entities (see Section 3.1), but if one module uses an external entity as a class function (by having a data structure instantiate it) and another treats it as an ordinary function, then SourceGraph will treat these as two separate entities. This can be alleviated somewhat if it is explicitly imported as a class, but some heuristics are still involved (as there is no way to tell by examining the import statement whether a class is being imported or just the record functions from a data structure).

Usage Problems

SourceGraph is currently extremely limited in terms of what options it will accept: it will currently only accept a root project

file (see Section 6 for more information) and analyses all Haskell source files found (whether they are needed or not). Furthermore, the generated report is in one monolithic HTML file (though Graphalyze's reporting facility does provide the ability to create other types of documents such as LaTeX).

Furthermore, the call graphs produced are at times either too large (in terms of number of entities, relationships and clusters involved) or break some fundamental assumption that GraphViz accepts, which results in SourceGraph not being able to visualise those call graphs. Even when GraphViz is able to cope with the graph, the resultant visualisations can be difficult to understand due to the number of relationships involved.

6. Obtaining and Using SourceGraph

SourceGraph is available under the GNU General Public License version 3 (or, at your discretion, later). The source code is available from the standard Haskell code repository *HackageDB* at `http://hackage.haskell.org/package/SourceGraph`. That page includes usage instructions and links to sample SourceGraph reports and the live software repository. As well as the relevant Haskell libraries that it uses, SourceGraph requires the GraphViz [5] suite of utilities to be installed so as to be able to generate the visualisations.

SourceGraph can currently analyse packages based upon either a Cabal [8] project file, or else an overall module. The former is preferred, as greater information is available from the Cabal file (such as the project name and all used modules) that must be inferred from an overall module. Whichever method is used, pass that file to SourceGraph:

```
$ SourceGraph path/to/project.cabal
```

or:

```
$ SourceGraph path/to/Project.hs
```

SourceGraph will then (if able to) create a `SourceGraph/` directory within the `path/to/` directory containing the analysis report in HTML format and a sub-directory with all of the visualisations. The visualisations are generated in PNG format, and link to larger SVG versions.

7. Conclusion and Future Work

SourceGraph introduces another approach of analysing software useful for programmers interested in how the entities within their code base interact. Whilst the set of analyses that SourceGraph applies to call graphs is rather limited at this time, there are several analysis algorithms that are planned to be introduced in the near future as well as considering alternate forms of visualisating the call graphs to make them easier to comprehend. Furthermore, those currently included still provide various starting points useful for manual refactoring of the source code as well as ways of visualising the call graph from different perspectives. In future, the report-based format for communicating the analysis results to the user may be complemented with an interactive interface to make it easier to manipulate and comprehend the visualisations.

Whilst SourceGraph currently only parses Haskell code, the overall concept can be extended to other languages. It is even possible to add this capability within SourceGraph itself thanks to projects such as Benedikt Huber's *Language.C* [7] library, which provides similar functionality as haskell-src-exts but for code written in C99.

Acknowledgments

With thanks to Professor Peter Adams for assistance in writing this paper, and the Haskell community for the various libraries used and assistance in writing and testing SourceGraph.

References

[1] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996. ISSN 0018-9162. doi: 10.1109/2.488299.

[2] C. Biemann. Chinese Whispers - an Efficient Graph Clustering Algorithm and its Application to Natural Language Processing Problems. In *Proceedings of the HLT-NAACL-06 Workshop on Textgraphs-06*, New York, USA, 2006.

[3] N. Broberg. haskell-src-exts: Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer. `http://hackage.haskell.org/package/haskell-src-exts`, 2009. accessed 5 October, 2009.

[4] G. Gan, C. Ma, and J. Wu. *Data Clustering: Theory, Algorithms, and Applications*. ASA-SIAM Series on Statistics and Applied Probability. SIAM, 2007. ISBN 0898716233.

[5] E. R. Gansner and S. C. North. An Open Graph Visualization System and Its Applications. *Software - Practice and Experience*, 30:1203–1233, 1999. `http://graphviz.org`.

[6] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the programatica toolset. In *High Confidence Software and Systems Conference, HCSS04*, 2004.

[7] B. Huber. Language.C - a C99 library for Haskell. `http://www.sivity.net/projects/language.c`, 2009. accessed 5 October, 2009.

[8] I. Jones. The Haskell Cabal: a common architecture for building applications and libraries. In M. van Eekelen, editor, *6th Symposium on Trends in Functional Programming*, pages 340–354, 2005. `http://www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc/24num.pdf`.

[9] I. L. Miljenovic. Graph theoretic analysis of relationships within discrete data. Mathematics honours thesis, University of Queensland, November 2008. `http://ivanmiljenovic.files.wordpress.com/2008/11/honoursthesis.pdf`.

[10] G. Narayan, K. Gopinath, and V. Sridhar. Structure and interpretation of computer programs. In *Theoretical Aspects of Software Engineering, 2008. TASE '08. 2nd IFIP/IEEE International Symposium on*, pages 73–80, June 2008. doi: 10.1109/TASE.2008.40.

[11] S. Peyton Jones, L. Augustsson, D. Barton, B. Boutel, W. B. J. Fasel, K. Hammond, R. Hinze, P. Hudak, J. Hughes, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. Also available at `http://www.haskell.org/definition/` and through Cambridge Press.

[12] S. Peyton Jones et al. GHC Language Features. `http://www.haskell.org/ghc/docs/latest/html/users_guide/ghc-language-features.html`, 2009. accessed 5 October, 2009.

[13] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993. `http://haskell.org/ghc/`.

[14] N. F. Rodrigues and L. S. Barbosa. Component identification through program slicing. In *In Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS*, pages 291–304. Elsevier, 2005.

[15] B. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, SE-5(3):216–226, May 1979. ISSN 0098-5589.

[16] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75283.