Call Graph and Data Flow Analysis of a Dynamic Functional Language[1]

Tamás Nagy, Zoltán Horváth, László Lövei, Melinda Tóth and Anikó Nagyné Víg

Refactoring is about improving the design of existing program code: making changes to the source code which preserves the meaning of the program in order to improve non-functional characteristics of the code like readability or maintainability.

In refactoring, the highest amount of work is usually the precondition checking, which makes sure that the refactoring does not change the behaviour of the system. Compared to precondition checking an uncomplicated transformation is almost straightforward. To check whether the preconditions are met, the type system of the language can provide very useful information. Our primary research areas are those functional languages which do not have a static type system. Our aim is to find static analysis techniques which can provide enough information to check whether preconditions are met. Two examples of such techniques are call graph and data flow analysis.

Call graph analysis aims to give the exact function dependency relations in a given amount of code. This means that the wider the scope of the analysis the more accurate the result will be, but the smaller scope's call graph can never be invalidated by the new data, just new edges can appear.

Data flow graph on the other hand aims to give back the flow of data through the functions [6]. In other words it shows how parameters, global variables are used, passed around the system.

In Erlang to have an accurate call graph data flow analysis has to be done as well. This is the result of the highly dynamic nature of the language, and because the way functions are treated. For example it is possible to call functions from data we receive from different parts of the system using the `erlang:apply` built-in function. The source code of such data does not clearly show that it will be used in a function call. In fact in different parts of the system it can be used for a different purpose. This is because functions are identified by `atoms`. `Atoms` can be created dynamically, for example with the `erlang:list_to_atom` function.

There are a subset of function calls which do not need data flow analysis. These are the static function calls. Where every element of the function call are known at compile time.

The dynamic calls – where the called function is not known at compile time – can be further categorized based on how much information is present at compile time. Naturally the less information given the harder the analysis is. There are some edge cases where the analysis is impossible. In these cases the analysis' aim is to limit the possible functions which the call could refer to.

The static analysis aims to create the function call graph of the static calls. This analysis is straightforward if we use the results of the semantical analysis which is incorporated in our refactoring system [3].

Collecting the function calls, then sorting based on which function they are in and which function they call, is essentially the work that has to be done to create the call graph for the static calls.

While building the static call graph is a relatively lightweight job, the dynamic call graph building takes significantly more time and resources. Of course the dynamic analysis possibly adds more data to the graph resulting in a more accurate call graph.

The analysis method is based on the Observer design pattern; which means we have entities (variables, atom, tuples etc.) that are loosely connected to each other. During the analysis new connections and entities are created as well. Connection represents the dependency between certain entities.

[1]Supported by ELTE IKKK, Ericsson Hungary

When an entity finds out more information about itself (for example a variable about its possible values) it sends this information to the other entities which are connected to it. Because connections can be created to an entity after its analysis is completed, the entity is not deleted after its analysis is finished.

Entities are modeled with Erlang processes, and data propagation with message sending. This approach creates the opportunity of parallel computation, because there is no strong order to how the entities should be processed. There is one further advantage which is recomputation after changes happen. If there are changes in the underlying code, the graph can be adjusted by creating and deleting entities and edges. In other words there is no need to recompute the whole graph.

To start the analysis we need initial entities which will be further analysed, these are the dynamic calls unknown values. For example the values of the `erlang:apply/3` function's parameters. Further analysis is done by investigating entity types and surroundings. This could result in new edges and new entities which need to be further analysed. When the value of the initial entities is found out with the analysis it is made available through the `call_graph` applications interface in the same way as the static call graph data.

In general the result of the call graph and data flow can be widely used by the precondition checks of the different refactorings [1, 2, 4, 3, 5].

By creating a different interface to retrieve the existing data of the static analysis, the data collection part for automatic detection of refactoring opportunities can be easier. Of course this interface will be used to retrieve the dynamic function call data as well. By providing a common interface for the two different data we further ease data collection complexity.

References

[1] C# Refactory homepage. `http://www.xtreme-simplicity.net/`.

[2] Eclipse Project homepage. `http://www.eclipse.org/`.

[3] Erlang Official homepage. `http://erlang.org/`.

[4] H. Li, S. Thompson, and C. Reinke. The Haskell Refactorer, HaRe, and its API. *Electronic Notes in Theoretical Computer Science*, 141(4):29–34, 2005.

[5] R. Szabó-Nacsa, P. Diviánszky, and Z. Horváth. Prototype environment for refactoring Clean programs. In *The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Vol. of extended abstracts*, page 113, Szeged, Hungary, July 2004. (full paper: `http://aszt.inf.elte.hu/~fun_ver/`,10 pages).

[6] G. Naumovich. Using the observer design pattern for implementation of data flow analyses. *SIGSOFT Softw. Eng. Notes*, 28(1):61–68, 2003.