Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 173–180.

Implementing Structural Complexity Metrics in Erlang[*]

Roland Király, Róbert Kitlei, Péter Tömösközi

Department of Programming Languages and Compilers,
Eötvös Loránd University, Budapest, Hungary
e-mail: kiralyroland@inf.elte.hu, kitlei@inf.elte.hu, tpeter@ektf.hu

Abstract

In this paper, we present a way to measure the structural complexity of distributed functional programs. Apart from introducing newly devised complexity metrics, it is desirable to make use of long standing structural complexity metrics [2] [1] [3] known for their usefulness in practice. However, most of these metrics are applied to procedural and OO programs [4] [5], which have different semantics and language constructs. We describe our approach to make some of these metrics applicable to functional programs. Also, we investigate how closely the above metrics follow changes in the source code made either by hand or by refactoring. Finally, we show that with a sufficiently rich set of metrics, it is possible to enhance the refactoring process further either by giving more elaborate feedback to the user on the effectiveness of a series of refactoring steps, or by automatically taking refactoring steps along the best choice offered by the measured metrics. We illustrate our achievements in Erlang [7] [6], using the back-end of the RefactorErl refactoring tool [8] [10] [9].

1. Introduction

Nowadays, with the increasing size and complexity of programs, testing and validating the code after changing it, takes an expanding part of developing.

The cost of posterior changes and modifications in the program code depends highly on the structural complexity of the original source code.

Measuring complexity is important, as it can indicate weaknesses of the program, or it can reveal, at an early phase, that testing is unattainable, or the cost of it is too high. In order to carry out measuring, we have introduced a new complexity metrics [2, 4, 5, 3] and also applied traditional, long standing practical methods, expanding these to suit the measuring of functional programs.

[*]Supported by KMOP-1.1.2-08/1-2008-0002 and Ericsson Hungary

Erlang [6] is a functional programming environment designed for building concurrent and distributed fault-tolerant systems with soft realtime characteristics.

RefactorErl [9] is a refactoring [11] tool that can store and recover the source code into and from the database to execute each refactor step. This system does not only automate systematic transformations on programs [12]. It can analyse the structure of the refactored program - based on the syntactic rules of the underlying programming language - and it can also collect and use semantical information about the source code.

2. Calculating metrics

A metric query language is incorporated into RefactorErl. These queries make measurements on the program text.

```
show number_of_funpath for module ('mod1','mod2') sum, avg
```

The queries are translated into lower level queries that traverse paths of the RefactorErl semantic graph. The semantic graph contains the abstract syntax tree of the program code (see in section 3), embellished with automatically collected semantic information. The metrics themselves are calculated based on the end nodes of the traversals.

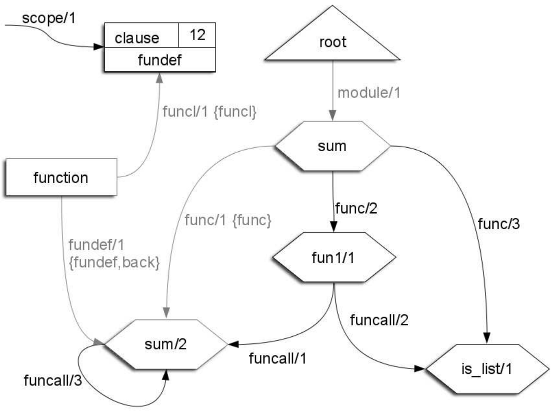

Figure 1: A subgraph of the program graph showing semantic nodes of modules and functions

The infrastructure developed to enquire complexity metrics can be separated into two layers. The lower layer contains the implementation of functions that make it possible to calculate individual metrics, and interface functions to access the functionalities from the outside.

Name of the metric	**Node type**
module_sum	module
line_of_code	module/function
char_of_code	module/function
number_of_fun/macro/record	module
included_files	module
imported_modules	module
number_of_funpath	module
function_calls_in/out	module
cohesion	module
function_sum	module/function
otp_used	module
max_depth_of_calling/cases	function/module
min_depth_of_calling/cases	function/module
number_of_funclauses	function/module
branches_of_recursion	function/module
mcCabe	module/function
calls_for_function	function
calls_from_function	function
number_of_funexpr/messpass	function/module
fun_return_points	function/module
average_size	module/function

Table 1: Metrics in the RefactorErl.

Metrics in the RefactorErl The first column of table 1 contains the function used for the implementation of the metric, the second column contains the type of the measured program construct (node).

Some measure in the table certain attributes of functionality, the levels of embedding, or the complexity of the control structures , but none are compound, complex metrics, however implemented metrics can be combined in any form, thus giving us the opportunity to create and try a more complex metric system both on the level of the query language and the interface functions.

3. Graph representation of the program text

The query of the metrics implemented in the RefactorErl system is made by wandering in the path defined in a semantic graph, constructed based on the program text, expanded with semantic information, along with calculating the metrics from the semantic information gathered in the path.

We can use the query language, developed for this purpose, in the RefactorErl system to defining graph paths on the RefactorErl Semantic Graph (picture 1). This language consists of path expressions, which lead to one or more graph nodes, or to a list of nodes. The system provides us the ability to enquire all the attributes that belong to a node from the node.

In the RefactorErl system the semantic graph is defined as an abstract data type that is capable of representing the syntactic and semantic structure of program text.

In the RefactorErl system the semantic graph is defined as an abstract data type that is capable of representing the syntactic and semantic structure of program text.

Syntactic and semantic program structure basically means attributes of program parts and many kinds of links between program parts. Such a structure can be represented by a directed, ordered, and labeled graph:

$$\mathcal{SG} = (N, A_N, A_V, A, T, E),$$

where

- N is the set of graph nodes, these will represent the nodes of the syntax tree and additional semantic nodes,
- A_N is a set of attribute names,
- A_V is a set of possible attribute values,
- $A : N \times A_N \to A_V$ is the node labeling partial function,
- T is a set of edge tags, and
- $E : N \times T \times \mathbb{N}_0 \to N$ is a partial function that describes labeled, ordered edges between the nodes.

The labeling function A provides attribute values for nodes. When a node n has an attribute a, the value of the attribute is $A(n, a)$.

Edges that are directed from a given node n and have a tag t have a fixed order. The endpoint of the i^{th} edge is given by $E(n, t, i)$. It is assumed that indexes start with 0 and are continuous.

High level information retrieval is supported by a query language that makes it easy to traverse graph structures with fixed depth. This query language consists of *path expressions* defined as follows:

$$\begin{aligned} P &= [PE_1, PE_2, \ldots, PE_k], \\ PE_i &= (t_i, d_i, f_i), \end{aligned}$$

that is, a path P is a list of path elements PE_i, where

- $t_i \in T$ is a link tag,
- $d_i \in \{F, B\}$ is a direction specifier (F stands for forward and B stands for backward), and
- $f_i : \mathbb{N}_0 \times N \to \mathbb{L}$ is a filtering function.

A path expression is usually evaluated using a single start node (although a list of start nodes could be used as well). The result of the evaluation is a list of nodes (link order is preserved for forward links).

The list gained this way is good for us to calculate complexity metrics that belong to the labeled node, from which we started the path expression.

In case of complexity metrics, nodes are nodes of modules and functions, they are also a list of them. In case of simpler size metrics it is enough to define the number of the list elements, but more complex metrics can be calculated with the analysis of the elements of the list as well, or with analyzing the data gained during the reading of the graph path.

4. Textual query language usage

Look at the example 2 to get a feel for how to use a path expressions to measure complexity metrics. The following simple code contains few functions written in Erlang.

```
-module(a).

 quicksort([H|T]) ->
    {Smaller_Ones,Larger_Ones} = split(H,T,{[],[]}),
    lists:append(quicksort(Smaller_Ones),
                       [H|quicksort(Larger_Ones)]);
 quicksort([]) -> [].

 split(Pivot, [H|T], {Acc_S, Acc_L}) ->
    if Pivot > H -> New_Acc = {[H|Acc_S], Acc_L};
       true      -> New_Acc = {Acc_S, [H|Acc_L]}
    end,
    split(Pivot,T,New_Acc);
 split(_,[],Acc) -> Acc.
```

Figure 2: Simple example

With the following query, we count the return points of the functions in the module.

```
show fun_return_points for function ({'a','quicksort',1},
                                      {'a','split',3}) sum
```

where

- `fun_return_points` is the function giving the measurment,

- `function` the type of nodes in the query,
- `({'a','quicksort',1},{'a','split',3})` List that contains the name of the modules in which the functions were defined, the name of the functions and theirs arities.

  Having such a textual interface to the queries is very convenient, as the programmer can access the queries on a very high level of abstraction.

5. Related Work

Several measurements used in imperative or OO programs [14, 13] that can be transported to functional programming languages are implemented in the RefactorErl system. These metrics can also be used for measuring functional languages, since they measure program constructs, which are used in most programming paradigms in a sufficiently similar manner. Such measurements include the relations and complexity of branching, functions and procedures [2, 3] (in case of OO programs: methods and their cohesion).

The AV metrics [5], measure the paradigms independently considering the data and their interrelations. AV metrics are implemented in Java.

There are some complexity measures for Java source code, which come integrated with the Eclipse development system [15].

Only a few implementations or embedded systems can be found for measuring functional source texts. There are functional complexity metrics, which measure some constructions of the functional languages [16, 17, 18], however, these either have not been implemented, or only prototype versions are available, which is not ready for general use. Their utility is proven by purely mathematical means,

A metrics implementation has also been developed for the F# language [19]. In this application [20, 21] examines the cardinality of program constructs found in F# source codes.

6. Conclusion

We have introduced a model for measuring functional programming languages, The implemented metrics are listed in Table 1. implemented several metrics, and measured their correlation. Most of the metrics are either quantity metrics, or simple complexity metrics.

By themselves, they are not very expressive: they measure certain attributes of functionality, the levels of embedding, or the complexity of control structures. However, these metrics can be combined in any desired order, forming more complex, compound metrics. This combination can be achieved both at the low-level interface and at the metrics query level.

We have presented a technology to measure structural complexity metrics in functional code. It is based on a program graph representation. We have expressed the metrics themselves in terms of basic notions of functional programming languages, which appear as semantic nodes in the program graph.

References

[1] Weyuker E. J., Evaluating software complexity mesures. *IEEE Trans. Software Engineering, vol.14, pp.1357-1365 1988.*

[2] McCabe T. J. A Complexity Measure, *IEE Trans. Software Engineering, SE-2(4), pp.308-320 (1976)*

[3] Piwowarsky, P. A Nesting Level Complexity Measure. *ACM Singplan Notices 17(9) pp.44-50 (1982)*

[4] Zoltán Porkoláb, Ádám Sipos, Norbert Pataki, Structural Complexity Metrics on SDL Programs. *Computer Science, CSCS 2006, Volume of extended abstracts, (2006)*

[5] Zoltán Porkoláb Programok Strukturális Bonyolultsági Mérőszámai. *PhD thesis Dr Tőke Pál, ELTE Hungary, (2002)*

[6] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víďż˝g, A., and Nagy, T.: Refactoring in Erlang, a Dynamic Functional Language *In Proceedings of the 1st Workshop on Refactoring Tools, pages 45–46, Berlin, Germany, extended abstract, poster (2007)*

[7] Erlang - Dynamic Functional Language
http://www.erlang.org

[8] Zoltán Horváth, Zoltán Csörnyei, Roland Király, Róbert Kitlei, Tamás Kozsik, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víďż˝g, Use cases *for refactoring in Erlang, To appear in Lecture Notes in Computer Science, (2008)*

[9] R. Kitlei, L. Lövei, M Tóth, Z. Horváth, T. Kozsik, T. Kozsik, R. Király, I. Bozó, Cs. Hoch, D. Horpácsi. Automated Syntax Manipulation in RefactorErl. *14th International Erlang/OTP User Conference. Stockholm, (2008)*

[10] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víďż˝g, A., Nagy, T., Tóth, M., and Király, R.: Building a refactoring tool for Erlang *In Workshop on Advanced Software Development Tools and Techniques, WASDETT 2008, (2008)*

[11] Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király, Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, Anikó Víg Use Cases for Refactoring in Erlang. *In Central European Functional Programming School, volume 5161/2008, Lecture Notes in Computer Science, pages 250–285, (2008)*

[12] T. Kozsik, Z. Horváth, L. Lövei, T. Nagy, Z. Csŕnyei, A. Víg, R. Király, M. Tóth, R. Kitlei. Refactoring Erlang programs. *CEFP'07, Kolozsvár (2007)*

[13] Fóthi Á., Nyéki-Gaizler J. On The Complexity of Object-Oriented Programs *in Proc. of the 3rd Symp. on Programming Languages and Software Tools Kaariku, Estonia, (1993)*

[14] Fóthi Á., Nyéki-Gaizler J., Porkoláb Z. The Structured Complexity of Object Oriented Programs *Computers and Mathematics with applications, (2002)*

[15] Eclipse Foundation
http://www.eclipse.org/

[16] Klaas van den Berg, Software Measurement and Functional Programming, *PhD Thesis University of Twente (1995)*

[17] Ryder, C. Software Measurement for Functional Programming, *PhD thesis, Computing Lab, University of Kent, Canterbury, UK 2004)*

[18] Ryder, C., Thompson, S. *Software Metrics: Measuring Haskell*, In Marko van Eekelen and Kevin Hammond, editors, Trends in Functional Programming (September 2005)

[19] Visual FSharp
http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/

[20] Diviánszky, P., Horváth, Z., Mészáros, M., and Páli, G.: Infrastructure for Analysis of F# *ELTE Innovation Day, February (2009)*

[21] Mészáros, M., Diviánszky, P., Góbi, A., Kovács, A., Leskó, D., Páli, G.: Detaching and Reconstructing the Documentary Structure of Source Code *8th International Conference on Applied Informatics, Eger, Hungary, (2010)*