Annales Mathematicae et Informaticae
38 (2011) pp. 59–74
http://ami.ektf.hu

Metrics based optimization of functional source code*

[a]**Roland Király**, [b]**Róbert Kitlei**

[a]Institute of Mathematics and Informatics, Eszterházy Károly College
e-mail: kiraly.roland@inf.elte.hu

[b]Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers
e-mail: kitlei@elte.hu

Submitted November 22, 2011 Accepted December 21, 2011

Abstract

In order to control software development, usually a set of criteria is fixed, among other things defining limits for the size of modules and functions, guiding layout principles etc. These criteria are not always observed, especially if the criteria are specified after pieces of the code are already written – that is, handling legacy code.

In this paper, we describe a method how the code base can semi automatically improved to conform more to the development criteria. We define a usage of a query language with which the user can employ our software complexity metrics to identify the out-of-line code parts, and select a transformation strategy that are automatically used by the tool to improve the identified parts.

Keywords: erlang, refactoring, structural complexity metrics, metric, functional language

1. Introduction

Measuring metrics in order to assist software development is not a new idea. In his seminal paper, Thomas J. McCabe [10] reasoned about the importance of source code measurement. He was investigating how programs can be modularised in order to decrease the costs of testing.

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

The application of complexity and other kinds of metrics yield interesting results if we use them to measure large projects. Currently, we have measured the development of *RefactorErl* [1, 2] itself, comparing various versions. Since *RefactorErl* underwent a big change about one year ago, gaining a new layer with well-designed interfaces, and some refactorings were greatly simplified with much tighter connections to the interface modules, it was expected that the values of some metrics would change substantially.

Table 1 shows the measurement results before and after the changes in the analyzed code body. The rows labelled *max* show the three largest values.

RefactorErl	Before upgrade	After upgrade
Effective line of code (sum)	14518	32366
Effective line of code (avg)	308.89	425.868
Effective line of code (max)	812/701/745	8041/1022/770
Number of functions (sum)	1329	2648
Number of functions (avg)	17.038	21.7
Number of functions (max)	85/70/49	551/91/64
Max depth of cases (avg)	1.7	1.6
Max depth of cases (max)	4	4
Branches of recursion (sum)	201	750
Branches of recursion (avg)	6.931	18.75
Branches of recursion (max)	20/18/15	211/43/35
Num. of function clauses (sum)	1725	6778
Num. of function clauses (avg)	36.70	89.18
Num. of function clauses (max)	139/133/53	3251/303/165
Number of fun-expression (sum)	185	271
Number of fun-expression (avg)	5.138	4.839
Number of fun-expression (max)	20/18/17	31/26/22
Number of funpath (sum)	3911	10854
Number of funpath (avg)	83.21	142.81
Number of funpath (max)	261/224/196	3752/399/333

Table 1: Values of metrics in two versions of RefactorErl

Refactorings make use of the new interface layer, which abstracts away a lot of code. Since these parts of the code are removed from the code of the refactorings, the modules of the refactorings have become smaller in size and complexity. Conversely, the number of connections between these modules and the query and interface modules have increased.

The complexity of the code has decreased in many parts, increased in other parts, but it is clear from the results that the complexity of the source code has increased. Another factor that indicates an increase in complexity is that loading the tool using takes an order of magnitude longer than before.

On the other hand, if we observe which metrics have increased and which ones

have decreased, we can realize a correlation between the change in complexity metrics and the rise of load time, namely the time spent on syntactic and semantic analysis.

Measurements of this type can help solve several optimization problems, which are connected to the duration of loading and that of analysis.

Knowing the relation among modules, the chains of function calls and the depth of call chains help us show which sets of modules can be considered as clusters, or which parts of the code are heavily connected. In earlier work this type of measurements has been used to cluster modules and to carry out changes related to clustering, but it has not been made available yet as a distinct metric.

Regarding measurements we can realize that certain metrics often change together. For example, if the *Number of functions* rises, then the *number of funpath*, *Number of funclauses* and the *Cohesion* of modules and the *Cohesion* between modules follows, and this has an influence on load time, by increasing the depth and complexity of the syntax tree and the amount of the related semantic information.

We can observe that the changes in the source code have brought significant changes in the software metrics at several points. Some of the changes have improved the values of the metrics, and some metrics have changed for the worse. For large code bases, it is very hard to estimate the effect of individual changes in the source code on the overall quality of the code without proper tool support. An appropriate tool, on the other hand, indicates the current complexity of the source code and potential error sources. Such feedback helps the software designer to decide whether the development process is progressing as desired.

For large programs, it is imperative to be able to restructure the program so that it becomes clearer, easier to maintain and to test. In order to achieve these goals, it would be advantageous if the aforementioned tool would also support code restructuring.

In the rest of the paper, can we find answers to the following questions.

1. Is it possible to track the changes in the complexity of the code when transformations are made, and detect any problems that may arise?

2. We are seeking a method to automatically or semi-automatically improve the source code based on the measured complexity.

3. How can we use the measured values of the metrics to enhance the process of software development?

As an answer to these questions, we have implemented a system that can measure the structural complexity of Erlang programs.

The rest of the paper is structured as follows. In Section 2, we discuss how complexity can be measured in functional languages. In Section 3, we describe our representation and how we measure and store the values of the metrics. In Section 4, we give an extension to our previous metrics query language with which it is possible to run automated transformations based on the measured values of

the metrics. This section contains the main contribution of the paper. Section 5 discusses related works and Section 6 concludes the paper.

2. Measuring functional languages

Several metrics developed for measuring imperative and object oriented languages can readily be applied to measuring functional languages. This is possible because there are similarities in several constructs when regarded with a degree of abstraction. As an example, a similar aspect of a library, a namespace, a class and a module is that they all can be regarded as collections of functions. If the chosen metric does not take the distinctive properties of these constructs into account (variables, method overrides, dynamic binding, visibility etc.), then it can be applied to these apparently diverse constructs. Some other properties of functional languages which bear such adaptable similarities to features in imperative languages are: nesting levels (blocks, control structures), function relations (call graph, data flow, control flow), inheritance versus cohesion, and simple cardinality metrics (number of arguments).

Functional programming languages contain several constructs and properties which are generally not present in imperative languages, thus require special attention during adaptation:

- list comprehensions,
- expression evaluation laziness, lack of destructive assignment,
- lack of loop construct, which evokes heavy use of either
  - tail recursion, or
  - higher order functions,
- referential transparency of pure functions,
- pattern matching,
- currying.

While these features raise the expressive power of functional languages, most of the existing complexity metrics require some changes before they become applicable to functional languages. So far, we have been successful in converting the metrics that we have encountered.

In addition to adapting existing metrics, we have introduced metrics that are well suited in general and Erlang in particular. We would like to point out the following findings.

- *Branches of recursion* measures the number of different cases where a function calls itself. This metric can be applied to non-functional languages as well, yet we did not see it defined elsewhere.

- Several cardinality measures, such as the number of *fun expressions*, and *message passing constructs.*
- The number of different *return points of a function.*
- We can measure metrics on a single clause of a function.
- We have extended metrics to take *higher order functions* into account, for example, how many times a fun expression is called. Due to the dynamic nature of Erlang, runtime function calls are hard to inspect, and we still have to improve this aspect of this feature.
- We are planning to investigate message passing further, which will enable us to make our metrics more precise.
- We are planning to measure OTP (Open Telecom Platform) [3] behaviours, which will uncover currently hidden function calls.

2.1. Short description of the metrics

Here we present a short overview about our implemented metrics. Hereinafter there is an enumeration of metrics which can be used as property in the extended query language. In the tables 2, 3 and 4 we can find the original name of the metric and its synonyms (one can use either the original name or any of the synonyms), afterwards we can find their short definitions.

Metrics for functions and modules	
Name	*Synonyms*
line_of_code	loc
char_of_code	coc
max_depth_of_calling	max_depth_calling, max_depth_of_call, max_depth_call
max_depth_of_cases	max_depth_cases
number_of_funclauses	num_of_funclauses, number_of_funclaus, num_of_funclaus
branches_of_recursion	branches_of_rec, branch_of_recursion, branch_of_rec
mcCabe	mccabe
number_of_messpass	-
fun_return_points	fun_return_point, function_return_points, function_return_point

Table 2: List of metrics for modules and functions

Effective Line of code The number of lines in the text of the module's or the function's source code excluding the empty lines.

Characters of the code The number of characters in the text of the module's or the function's source code.

Max depth of calling The length of function call-paths, namely the path with the maximum depth. It gives the depth of non-recursive calls. Recursive calls are covered by *depth_ of_ recursion/1* function.

Max depth of cases Gives the maximum of case control structures nested in *case* of the concrete function (how deeply are the case control structures nested). In case of a module it measures the same regarding all the functions in the module. Measuring does not break in *case* of *case* expressions, namely when the *case* is not embedded into a *case* structure.

Number of funclauses The number of the given function's function clauses (which have the same name and same arity). In case of module it counts all of the function clauses in the given module.

Branches of recursion Gives the number the given function's branches i.e., how many times a function calls itself, and not the number of clauses it has besides definition.

McCabe McCabe cyclomatic complexity metric. We define the complexity metric in a control flow graph with the number of defined basic edges, namely the number of outputs a function can have disregarding the number of function outputs functions within the function can have. Functions called each count as one possible output.

The sum of the results measured on the given module's functions is the same as the sum measured on the module itself. This metric was developed to measure procedural programs, but it can be used to measure the text of functional programs as well. (in case of functional programs we measure functions).

Number of funexpr The number of function expressions in the given function or module. (It does not count the call of function expressions, only their creation.)

Number of message passings In case of functions it counts the number of code snippets implementing messages from a function, while in case of modules it counts the total number of messages in all of the functions of the given module.

Function return points The number of the given function's possible return points. In case of module it is the sum of its function return points.

Calls for the function The number of calls for the given function. (It is not equivalent with the number of other functions calling the given function, because all of these other functions can refer to the measured one more than once.)

Metrics only for functions	
Name	*Synonyms*
calls_for_function	calls_for_fun, call_for_function, call_for_fun
calls_from_function	calls_from_fun, call_from_function, call_from_fun
function_sum	fun_sum

Table 3: List of metrics only for functions

Calls from the function The number of calls from a certain function, namely how many times a function refers to another one. The result includes recursive calls as well.

Function sum The sum calculated from the function's complexity metrics that characterises the complexity of the function. It is calculated using various metrics together.

Metrics only for modules	
Name	*Synonyms*
number_of_fun	num_of_fun, num_of_functions, number_of_functions
number_of_macros	num_of_macros, num_of_macr
number_of_records	num_of_records, num_of_rec
included_files	inc_files
imported_modules	imp_modules, imported_mod, imp_mod
number_of_funpath	number_of_funpathes, num_of_funpath, num_of_funpathes
function_calls_out	fun_calls_out
cohesion	coh
otp_used	otp
min_depth_of_calling	min_depth_calling, min_depth_of_call, min_depth_call
module_sum	mod_sum

Table 4: List of metrics only for modules

Number of functions The number of functions implemented in the module, excluding the non-defined functions.

Number of macros The number of defined macros in the module.

Number of records The number of defined records in the module.

Number of included files The number of visible header files in the module.

Imported modules The number of imported modules used in the given module. The metric does does not take into account the number of qualified calls (calls that have the following form: *module:function*).

Number of funpath The total number of function paths in the given module. The metric, besides the number of internal function links, also contains the number of external paths, or the number of paths that lead outward from the module.

Function calls into the module The number of function calls into the given module from other modules.

Function calls from the module The number of function calls from the given module towards other modules.

Cohesion of the module The number of call-paths of functions that call each other in the module. By call-path we mean that an *f1* function calls *f2* (e.g. `f1()->f2().`). If *f2* also calls *f1*, then the two calls still count as one callpath.

Max depth of calling The maximum depth of function call chains within the given module. It gives the depth of non-recursive calls.

Module sum The sum of *function_sum* for all functions in the given module.

3. Program graph representation

In [4], we have introduced an extensible architecture in which the definition and acquisition of important attributes of the source code can be conveniently formulated. When the source code is loaded, it is parsed into an abstract syntax tree, which is then turned into a program graph by adding static semantic nodes and edges. These semantic nodes and edges describe the important attributes and connections of the source code: the call graph, the statically analysable properties of dynamic constructs, the data flow necessary to track the spreading of values. The semantic nodes currently comprise all information that is necessary for the calculation of metrics, but the architecture is extensible: new semantic constructs can be added easily.

The program graph, for our purposes, contains syntactic and semantic nodes and edges. The abstract syntax tree built upon the represented source code forms a subgraph of the program graph. In addition to this subgraph, the program graph also contains nodes that describe semantical information, such as the binding structure of variables. The edges of the program graph are directed, labelled, and for each node, the outgoing edges having the same label are ordered.

Low level query language. Information retrieval is supported by a low level query language that makes it easy to traverse graph structures. This low level query language consists of fixed length *path expressions*, which run starting from a single node, can traverse edges in a forward or backward direction, and can filter the resulting nodes in each step based on their contents.

Metrics are calculated by running several queries that collect syntactic and semantic constructs, and then evaluating the information content of the resulting nodes.

Summing it up, the calculation of complexity metrics takes place in three steps:

1. We construct the program graph of the source code. As we measure several metrics on the same program graph, the program graph is already available. During the static analysis of the source code, we construct the Abstract Syntax Tree of the code, then we expand it from all the semantic information gained with all of the static analyses. If we already have the semantic graph at hand, then the process only takes two steps.

2. We execute the path expression that is appropriate for the metric. The result of the path expressions defined on the constructed graph will be a list of nodes. In most cases, the characteristic complexity metric can be calculated from the result.

3. We calculate the value of the metric. For some metrics, this step is simply the calculation of the cardinality of the resulting list (e.g. number of functions), whereas for other metrics, filtering has to be done (e.g. internal cohesion of the module). The result of this expression is a list of all the function nodes, which are available on the defined graph path. The length of the list gives the total number of function paths. The result contains all of the function calls within the module and the for and from calls. If we wanted to measure regarding the internal cohesion of the module, then we would have to filter the result.

Caching calculated values of metrics As metrics have to be recalculated each time the code is changed, it is desirable to make this process as fast as possible. Fortunately, most metrics can be calculated incrementally, if we store the measured values in the associated *module* or *function* semantic node. This way, only the values of those metrics have to be adjusted that are affected by the change in the code.

Number of function clauses

```
show number_of_funclauses for module('exampmod')
```

Figure 1: Query language example

Function calls into the module

```
show function_calls_out for module('exampmod')
```

Figure 2: Query for modules

Example Erlang module

```
-module(exampmod).

abs(X) when X >= 0 -> X;
abs(X)             -> -X.

sign(0)            -> 0;
sign(X) when X > 0 -> 1;
sign(X)            -> -1.

manhattan(Xs, Ys) ->
 Pairs = lists:zip(Xs, Ys),
   List =
     [abs(X-Y) || {X, Y} <- Pairs],
   lists:sum(List).
```

Figure 3: An example module in Erlang

Textual query language Figure 1 and Figure 2 show two metrics queries. The former shows the number of all function clauses in the module; for the module *exampmod*, whose code can be seen in Figure 3, this value is 6, as the function *abs* has two clauses, *sign* has three and *manhattan* has one. The latter shows the number of calls of external functions in the module; for *exampmod*, this value is 2, since calls to *lists:sum* and *lists:zip* (functions of the module *lists*) are included, but the call to *abs*, a local function, is not.

In Section 4.1, we give an extension to this query language that enables the user to write transformations based on the measured metric values. Batches of such queries can be stored as scripts that automatically improve the source code when executed.

4. Metrics driven transformations

Most of the metrics can be associated to a node in the program graph so that the value of the metric can be calculated using only the syntax subtree below the node. We store the current values of metrics in the associated node, which serves two purposes when the code is transformed. Firstly, since most of the metrics are

compositional, we can use the stored values as caches, and only recalculate the parts that have changed, thereby making the calculation of the new values faster. Secondly, we can compare the old and the new values of the metrics, and we can make the necessary arrangements if the code is changed in an undesired way:

1. We can leave the transformation of the code to the user. This task is time consuming and error prone, especially if the code base is large, difficult, or unknown to the user. In this case, RefactorErl can help the user by displaying the values of metrics measured on the current code, and warns the user if a value goes beyond a specified limit.

2. The user may use the semi-automatical transformations of RefactorErl to improve the code. With this option, the user regains control of the process of transformation: he chooses what gets transformed and in what way. Using RefactorErl ensures that the code is transformed in all necessary places, and that the resulting code is syntactically valid, and semantically equivalent to the original.

3. As the main contribution of the paper, we introduce a new approach: metrics driven automatic code optimization. We elaborate it in the following section.

4.1. Metrics driven automatic code optimization

We introduce an extension to our query language in which the transformation engine of RefactorErl can be instructed to improve the source code based on the calculated metrics. The grammar of the original query language is shown in Figure 4.

Optimization query language Figure 5 shows the grammar of the optimization extension language. In Figures 4 and 5, `Id`, `Ids`, `ArRel`, `LogCon` `Var` and `Int` stand for an identifier, a list of identifiers, an arithmetic relation (e.g. $<$), a logical connector (e.g. *and*), a variable and an integer, respectively. The extension language is quite straightforward, describing which modules are to be transformed (*optimize*), which transformations are to be used (*using*), where the transformations are to be attempted (*where*), and at most how many steps are to be attempted (*limit*). In the *where* clause, the identifiers indicate a metric; variables may only be used if the query is part of a script, and the variable is bound to a value of a metric.

$$\begin{array}{rcl} \textbf{MetricQuery} & \rightarrow & \textbf{Show Loc} \\ \textbf{Show} & \rightarrow & \textbf{show}\ Id \\ \textbf{Loc} & \rightarrow & \textbf{module}\ Id \mid \textbf{function}\ Id \end{array}$$

Figure 4: Slightly abridged grammar of the metrics query language

$$
\begin{array}{rcl}
\textbf{Query} & \rightarrow & \textbf{MetricQuery} \mid \textbf{OptQuery} \\
\textbf{OptQuery} & \rightarrow & \textbf{Opti Trs Where Limit} \\
\textbf{Opti} & \rightarrow & \textbf{optimize all} \mid \textbf{optimize}\ Ids \\
\textbf{Trs} & \rightarrow & \textbf{using}\ Ids \\
\textbf{Where} & \rightarrow & \textbf{where Cond} \\
\textbf{Cond} & \rightarrow & \textbf{Expr}\ ArRel\ \textbf{Expr} \\
 & \mid & \textbf{Cond}\ LogCon\ \textbf{Cond} \\
\textbf{Expr} & \rightarrow & Id \mid Var \mid \textbf{MetricQuery} \\
\textbf{Limit} & \rightarrow & \textbf{limit}\ Int
\end{array}
$$

Figure 5: Grammar of the metrics query language with optimization

Metrics driven transformation example The first code snippet in Figure 7 shows a function that contains too deeply nested *case* expressions. Figure 6 shows the script we are going to use to instruct the engine to improve the code.

The script consists of two steps. The first step calculates the maximum level of case nesting in module *not_present* (not appearing in this paper); let this value to be 1. This value is assigned to the variable *P1*. The second step starts the transformation engine, which tries to decrease the number of nodes in module *to_refactor* where the condition holds. Since the *number_of_functions* is only one, the significant part of the condition selects those nodes where *max_depth_of_cases* is larger than one. In the original code, the function *f* contains a case construct of depth 3, which is then refactored using the *introduce function* transformation (*introduce_fun*). The transformation takes the body of the innermost case construct, and extracts it to a new function *f0*.[1]

As we have not reached the step limit, the condition is reevaluated: the *number_of_functions* has grown to 2, and the *max_depth_of_cases* is decreased to 2. Since this value is still over the desired value, a similar transformation step is applied as depicted in Figure 7. This is the last transformation step: we have reached the step limit. Incidentally, we have also eliminated all nodes where the condition of the query would hold.

Since the transformation engine executes the script without external help, it might transform the code in an inferior way to an expert. If the result of the script execution does not turn out to be desirable, the user want have to revert the code to the state before the execution of the script. It is also possible to revert only some steps that the script took.

[1]The name of the function is generated. If the name of the function is not to the liking of the user, it can be changed using another available transformation later.

```
P1 = show
       max_depth_of_cases
     for
       module not_present
optimize
  module to_refactor
where
  max_depth_of_cases > P1
  and
  number_of_functions < 10
using
  introduce_fun
limit 2
```

Figure 6: Metric query language example code

5. Related work

Several IDEs for object oriented languages (e.g. Eclipse [7], NetBeans, IntelliJ Idea) provide both metrics and refactorings, however, we are unaware that the two areas are connected in any of them.

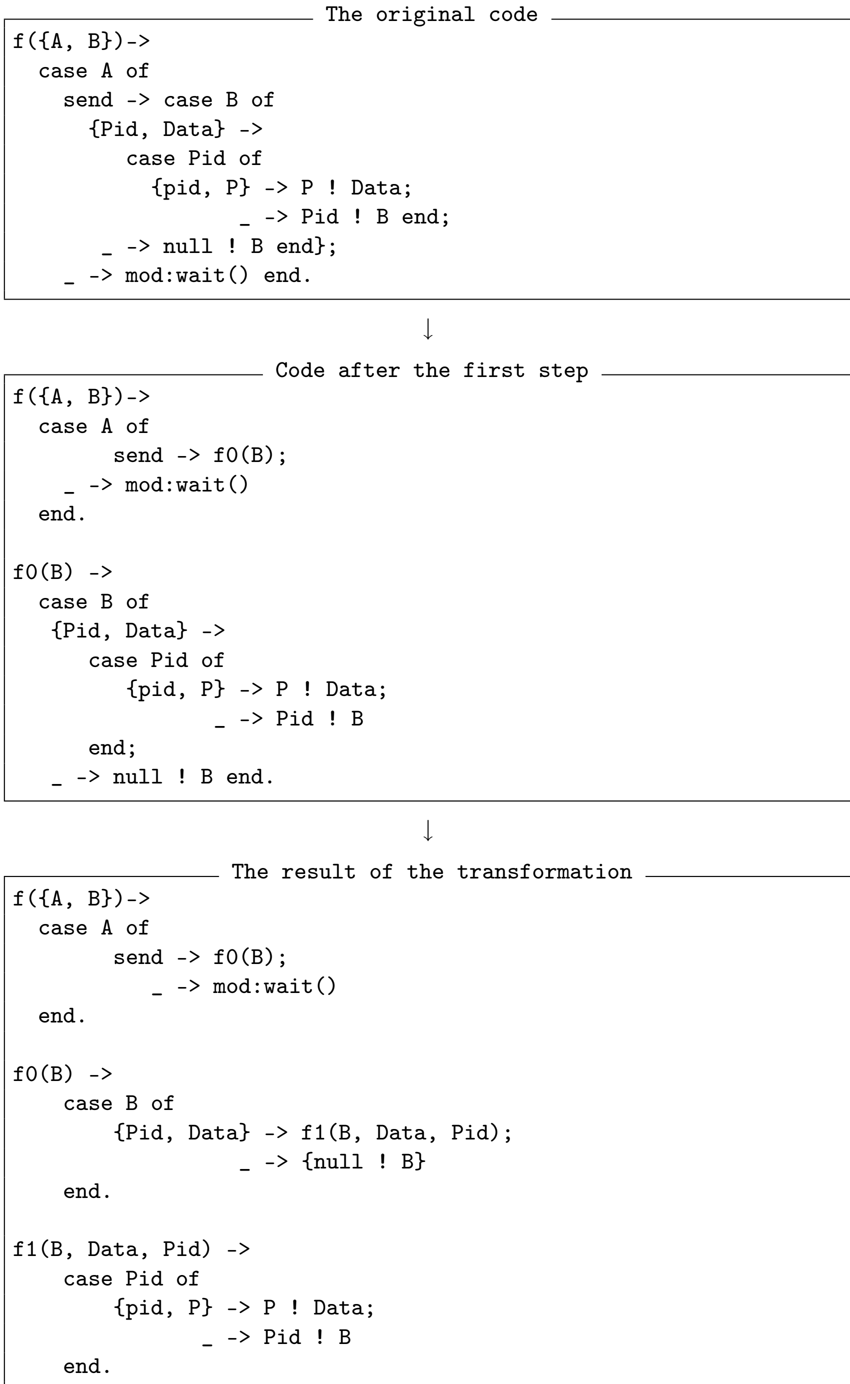
Simon, Steinbrückner and Lewerentz [5] have created a tool that visualizes several metrics based on Java and C++ code, thereby helping the user to make decisions about transforming his code. They show that well chosen metrics can support the decision of the user before he confirms a refactoring.

The goal of the project *Crocodile* [6] is to provide concepts and tools for an effective usage of quantitative product measurement to support and facilitate design and code reviews particularly for object oriented programs and reusable frameworks. While Crocodile is useful as a measurement tool, it also can interactively assist the programmer in executing transformation steps.

Tidier [8, 9] is a software tool that makes a series of fully automated code transformations which improve the performance, quality and/or structure of the code. Tidier uses simple semantics preserving transformations with an emphasis on easy validability. The transformations are universally applicable, and do not rely on metrics for guidance.

6. Conclusion and future work

In this paper, we have presented a way to improve software by applying automated transformations based on complexity metrics. We have defined a query language which makes the transformations accessible to the end user, and we have imple-



The original code

```
f({A, B})->
  case A of
    send -> case B of
      {Pid, Data} ->
         case Pid of
           {pid, P} -> P ! Data;
                  _ -> Pid ! B end;
       _ -> null ! B end};
    _ -> mod:wait() end.
```

↓

Code after the first step

```
f({A, B})->
  case A of
        send -> f0(B);
    _ -> mod:wait()
  end.

f0(B) ->
  case B of
   {Pid, Data} ->
      case Pid of
         {pid, P} -> P ! Data;
                _ -> Pid ! B
      end;
   _ -> null ! B end.
```

↓

The result of the transformation

```
f({A, B})->
  case A of
        send -> f0(B);
           _ -> mod:wait()
  end.

f0(B) ->
    case B of
        {Pid, Data} -> f1(B, Data, Pid);
                  _ -> {null ! B}
    end.

f1(B, Data, Pid) ->
    case Pid of
        {pid, P} -> P ! Data;
               _ -> Pid ! B
    end.
```

Figure 7: Two steps of automatic source code transformation

mented an engine that improves the source code by executing scripts written in the query language.

We have implemented the method described in the paper as part of RefactorErl, an Erlang analyser and transformation tool. The back end of the tool builds the program graph representation discussed in Section 3.

Using the information in the graph, we collect the values of the metrics and we store them in the corresponding nodes of the graph. When the graph is changed, these values are updated incrementally. The values are shown in the user interface, and are also queryable. The user may define a limit for a metric or a combination of metrics; when the code is measured to be outside this limit, the user is alerted. Analysis of how such limits should be defined may constitute a promising new line of research.

In addition to the calculation of the metrics values, we have implemented the metrics driven code optimization discussed in Section 4. We have extended the previous metrics query language as seen in Figure 5 to support query based automatic code optimization, and we have implemented the engine that runs the transformations according to the query. The engine is also capable of running scripts that contain batches of queries. If unsatisfied with the result, the user can fully or partially revoke these transformations.

References

[1] 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)*

[2] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, 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)*

[3] Erlang - Dynamic Functional Language, `http://www.erlang.org`

[4] Kitlei, R., Lövei, L., Nagy, T., Víg, A., Horváth, Z., and Csörnyei, Z. *Generic syntactic analyser: ParsErl*, In Proceedings of the 13th International Erlang/OTP User Conference, EUC 2007, Stockholm, Sweden, November 2007

[5] Frank Simon, Frank Steinbrückner, Claus Lewerentz *Metrics based refactoring* IEEE Computer Society Press 2001 30–38.

[6] Claus Lewerentz, Frank Simon *A Product Metrics Tool Integrated into a Software Development Environment* Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543 Springer-Verlag 256–257.

[7] Eclipse Foundation, `http://www.eclipse.org/`

[8] Thanassis Avgerinos, Konstantinos F. Sagonas *Cleaning up Erlang code is a dirty job but somebody's gotta do it.* Erlang Workshop 2009: 1–10.

[9] Konstantinos F. Sagonas, Thanassis Avgerinos *Automatic refactoring of Erlang programs.* PPDP '09 Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming 2009: 13–24.

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