STATIC ANALYSIS BASED SUPPORT FOR PROGRAM COMPREHENSION IN ERLANG [1]

Melinda TÓTH, István BOZÓ, Judit KŐSZEGI, Zoltán HORVÁTH
Department of Programming Languages and Compilers Eötvös Loránd University, Faculty of Informatics,
Pázmány Péter st. 1/C, Budapest, Hungary, e-mail: {toth_m, bozo_i, kojqaai, hz}@inf.elte.hu

ABSTRACT

Program comprehension is important process in software maintenance, considering the lifetime of an industrial software. The first task for a developer is to understand the structure and the behaviour of the program without considering the type of the change - refactoring, bugfix - must be performed on the source code. Understanding and debugging the source code is not straightforward in case of a dynamically typed functional programming language, like Erlang. Thus RefactorErl supports code comprehension through a Semantic Query Language that helps the developers to query semantic relationships in their software product.

Keywords: *Erlang, program comprehension, query language*

1. INTRODUCTION

Under developing a software certain changes could be performed on the source code to improve its quality or to fix bugs, etc. Maintaining the software could be hard for the developer of the program and even harder for new project members. A tool which helps in program comprehension is valuable for everyone. Therefore our goal is to develop such tool for Erlang [11].

Our research was focused on refactoring [2] at first. Hence complex static analysis is necessary to guarantee a meaning preserving source code transformation, thus we have been developing RefactorErl [13] as a source code analyser and transformer tool. The result of the static analysis could be useful for developers who want to understand the behaviour of the program, or could simplify everyday programming tasks. Therefore we have decided to develop a query language [6] in RefactorErl to make the information about source code available for programmers.

The paper is structured as follows: Section 2 provides a description of our Erlang refactoring tool, RefactorErl; Section 3 introduces different levels of information query in RefactorErl; Section 4 describes examples how RefactorErl can support code comprehension; Section 5 describes the web based interface for using queries; Section 6 presents further source code analysis applications available in our refactoring tool. Finally, Section 7 describes related work, and Section 8 is a conclusion.

2. REFACTORERL

RefactorErl [3] is a source code analyser and transformer tool originally developed to support refactoring of Erlang programs. Refactoring [2] is a meaning preserving source code transformation, i.e. to modify the source code without altering its behaviour. Erlang [11] is a dynamically typed functional programming language which was designed to write high available, concurrent, distributed, fault tolerant systems with soft real-time characteristic, like telecommunication systems.

To guarantee a meaning preserving transformation in case of a dynamic language is not straightforward, and lots of static checks have to be satisfied. Therefore each refactoring is divided into two parts: checking side-conditions and transforming the source code. To validate the side-conditions different kinds of syntactic and static semantic information is required about the software. Therefore RefactorErl represents the source code in a Semantic Program Graph that provides effective information retrieval about the source code.

The *Semantic Program Graph – SPG* stores and manipulates the source code in a layered graph: it has a lexical, a syntactic and a semantic layer. The lexical layer contains the token information including whitespaces and comments. The syntactic layer in based on an the abstract syntax tree (AST). When the scanning and parsing process is completed and the AST is ready, different semantic analysers construct the semantic layer, e.g. the function call graph or the binding structure of variables etc. The semantic analysers are part of the modular, incremental, asynchronous analyser framework of the tool. This framework makes it possible to run the analysis in parallel on the syntactically independent parts of the program. The module, function, variable, record and context analysers form the base of the semantic layer and further complex analysis, such as side-effect analysis, data-flow analysis, dynamic function call analysis, etc.

RefactorErl is fully integrated with (X)Emacs and Eclipse, it has an interactive and a scriptable Erlang console interface (called *ri & ris*), it is also available through Vim or GVim and it has a Web based server interface to serve multiple users.

3. QUERYING SEMANTIC RELATIONSHIPS

As already mentioned, RefactorErl represents the source code in a Semantic Program Graph. This graph is a rooted, directed graph with typed nodes and labelled edges. Gathering information about the software is possible through graph traversing. You can query semantic information about the source code using a query language or draw the graph to find semantic relationships in your program.

[1]SUPPORTED BY KMOP-1.1.2-08/1-2008-0002 AND ERICSSON HUNGARY

```
path() = [PathElem]

PathElem = Tag | {Tag, Index} | {Tag, Filter}
Tag      = atom() | {atom(), back}
Index    = integer() | {integer(), integer()} | {integer(), last}
Filter   = {Filter, 'and', Filter} | {Filter, 'or', Filter} |
           {'not', Filter} | {Attrib, Op, term()}
Attrib   = atom()
Op       = '==' | '/=' | '=<' | '>=' | '<' | '>'
```

Fig. 1 The syntax of *path expression*

3.1. Path Expressions

RefactorErl introduces a low level query language (similar to XPath [15]) for retrieving static semantic and syntactic information about the source code. This language was originally designed to support information gathering for refactorings, but it is also useful in code comprehension to determine relationships in the source code.

The Semantic Program Graph can be traversed by evaluating a *path expression* starting from a given node. The structure of the *path expressions* and the filters are written according to the Erlang EDoc type specification syntax [9] on Figure 1. The type `path()` is a sequence of `PathElem`. A `PathElem` can be a graph edge label (*Tag*) or a graph edge label with filtering options ({*Tag, Index*} or {*Tag, Filter*}). The former case represents the labelled graph edges to follow during graph traversal, and the latter make it possible to select a subset of graph nodes during the graph traversal according to the given filtering options. It is possible to filter the result with syntactic or semantic information (`{Tag, Filter}`) and also with the indices of edges in the graph (`{Tag, Index}`). For instance, the pair `{esub, {6,8}}` denotes the sixth, seventh and eighth subexpressions of a graph node. The graph edges can be traversed both forward (`Tag = atom()`) and backward direction (`Tag = {atom(), 'back'}`).

The *path expressions* can be evaluated with the `path/2` function, which takes two arguments: a starting graph node `node()` and a `path()` to follow.

For example, the following function call queries the functions defined in a given module:

```
path(ModuleNode,
     [{form, {type, '==', func}}])
```

The disadvantage of the path expression is that it strongly depends on the graph representation of RefactorErl, and even the smallest change on the representation can affect the path expressions. For instance, if we change the label of an edge, then each path expression using that label should be updated too. Therefore we have defined a few library modules with functions which returns the path expressions, and a high level evaluation framework to execute queries. The main element of the framework is the function `exec`. Its basic behaviour is similar to the function `path`, it takes a starting node and a path to execute.

For example, the queries that are related to modules were defined in module `reflib_module`, and `function/0` defines the previously used query:

```
-module(reflib_module).
...
records()->
    [{form, {type, '==', record}}].

functions()->
    [{form, {type, '==', func}}].
```

You can query the functions defined in the given module `ModuleNode` by using the query execution framework and the library functions:

```
exec(ModuleNode,
     reflib_module:functions())
```

Using the `exec/2` function you can also evaluate a sequence of queries from the start node (with functions `seq/1` or `seq/2`) or evaluate more queries from the same start node (with functions `all/1, all/2, any/1, any/2`). For example, the following query returns both, the functions and records defined in the module `ModuleNode`:

```
exec(ModuleNode,
     all(reflib_module:functions(),
         reflib_module:records())
```

3.2. Erlang Semantic Query Language

When using *path expressions* to query information it is necessary to know the structure of the Semantic Program Graph. We can not presume that the developer has any precognition about our program representation, thus we have defined a user friendly *Erlang Semantic Query Language* [6] that does not assume any knowledge about the semantic graph. It operates with Erlang language entities, thus no prior knowledge is necessary in addition to the Erlang language.

The language was designed to provide help in the software development process. It uses a formalism close to the Erlang language concepts, thus a developer can easily learn and adopt it.

```
semantic_query    ::= initial_selection ['.' query_sequence]
query_sequence    ::= query ['.' query_sequence]
query             ::= selection | iteration | closure |
                      property_query
initial_selection ::= initial_selector ['[' filter ']']
selection         ::= selector ['[' filter ']']
iteration         ::= '{' query_sequence '}' int ['[' filter ']']
closure           ::= '(' query_sequence ')' int ['[' filter ']'] |
                      '(' query_sequence ')+' ['[' filter ']']
property_query    ::= property ['[' filter ']']
```

Fig. 2 The structure of the semantic queries

The language concepts are defined according to the semantic units and relationships of the Erlang language, e.g. functions and function calls, records and their usage, etc. The main elements of the language are the *entities*: module, function, variable, expression, record, macro, etc. Each entity has a set of *selectors* and *properties*. A **selector** is a binary relation between *entities*, it selects a set of *entities* that meet the given requirements. A **property** is a function, which describes some properties of an entity type. For example a function has a `name` and an `arity` *property*, and it has a `refs` *selector*, which refers to its references. Thus here the *selector* `refs` is a relation between functions and expressions, and the mentioned *properties* refer to one of the attributes of the entity. It is also possible to filter entities based on the properties. A **filter** is a boolean expression to select a subset of *entities*. We can build *filters* by using *properties* with boolean values, valid Erlang comparisons, logical operators or embedded queries.

There is a special *selector* type in the language called **initial_selector**, which is the starting point of a semantic query. The selectors of this type do not have an initial set of *entities* to work with, so they do not belong to specific *entity* types. The *initial_selectors* get the current file and position as their parameters. Almost all of them start with the character @ to indicate that they depend on a position. For example the *initial_selector @var* will look for a variable at the given position. If no variable can be found the result will be empty. Besides the position based *initial_selectors* there is another *initial_selector*: *mods*. This selector returns all of the modules that are loaded into the semantic program graph.

The formal syntax of the language is described on Figure 2. A *semantic query* is a sequence of *queries* starting with an *initial_selector* and an optional filter. Queries are separated with dots. A query is a

- selection (calculates the relationship with other entities based on selectors),
- iteration (iterates a query `n` times),
- closure (calculates the transitive closure of a query sequence) or
- property query (selects a property of an entity).

4. SUPPORTING PROGRAM COMPREHENSION

Understanding and maintaining a huge software system is a hard task, and it becomes more harder if the programming language gives opportunity to use dynamic constructs, like dynamic function calls. Therefore, a Semantic Query Language have been developed for RefactorErl to provide a tool for program comprehension of Erlang programs. This query language offers help for better understanding the relations among the program parts that are difficult, or almost impossible to detect manually.

In the rest of this section we give case studies about how to use the language to find specific information about the source code.

4.1. Call Chain

Assume that during the development the return value of a specific function could be changed. In this case we have to lookup for every application of this function and update the context according to the performed change. We can get these references by evaluating the *@fun.refs* query. But these modifications could spread through the call chain further and further. To calculate the backward call chain we can use the transitive closure on the function call graph. The following query identifies the backward call chain for the selected function: *@fun.(called_by)+*, where *@fun* means the selected function in the editor.

We can calculate the forward call chain with the query: *@fun.(calls)+*. The result will contain those functions that are called by the pointed function. During the debugging, when the selected function returns with bad return value, this query seems to be useful.

Calculating the full transitive closure can take a long time on a large code base. If a shorter chain also holds useful information, we can use the *@fun.(calls)n* query (where *n* denotes the maximum length of the call chain).

4.2. Finding Dynamic Function References

In Erlang, the most commonly used dynamic construct is the `apply/3` BIF (built-in function). That function takes three parameters: the name of the module, where the function is defined, the name of the function and a list, which contains the actual parameters of the function. `apply/3` evaluates the function with the given parameters. Understanding a program with `apply(Mod, Fun, ArgList)` dynamic function calls could be hard, in case we can not determine the exact value of the `Mod, Fun, ArgList` variables. Based on static data-flow analysis RefactorErl is able to determine dynamic function references. Therefore when the *@fun.calls* query is evaluated, the result also contains functions using `apply/3` to call *@fun*. However, it is also possible to query only the dynamic references using the *@fun.dynref, @fun.dyncall, @fun.dyncalled_by* queries.

In everyday programming task, when we want to determine what the program does at a certain point of the program, the *@expr.dynfun* query returns for us those functions, that are possibly called by the pointed `apply` expression.

Lets consider the following Erlang code part, that contains the definition of the `sum` function, which summarizes the elements of a list, and the definition of `test1` and `test2` functions, which call the `sum` function with a dynamic construct.

In the body of function `test1` it is easy to detect that the `apply` expression calls the `sum` function. In the body of `test2` it is not so easy to detect the called function manually, because it may be called outside the module (for example, from `mod2`). Pointing the cursor to the apply expression in function `test2` and then running the query *@expr.dynref* returns both `test/1` and `sum/2` – because of the function calls in `dup_sum/0`.

```
  -module(mod1).
  -export([sum/2, test1/1, test2/3]).

  sum([], Acc) ->
      Acc;
  sum([H|T], Acc) ->
      sum(T, Acc + H).

  test1(List)->
      Fun = sum,
      apply(mod1, Fun, [List, 0]).

  test2(Mod,Fun, ArgList)->
      apply(Mod, Fun, List).
-------------------------------------------
  -module(mod2).

  dup_sum()->
      mod1:test2(mod1, sum, [[1,2,3], 0]) ++
      mod1:test2(mod1, test1, [[1,2,3]]).
```

4.3. Identifying Spawned Processes

Erlang has built in support for concurrent program development. When asynchronous message passing and blocking message receiving expressions are often used in the source code, following the control and understanding the behaviour of the program are difficult.

Lets consider the following function body:

```
fun1()->
   ...
   receive
       {msg, B}  -> process(B);
       {info, A} -> print(A)
   end,
   ...
```

The execution of function `fun1/0` depends on the received message. If the message contains the message tag `msg`, function `process/1` is called. If the message contains the tag `info`, function `print/1` is called. To determine which message was sent, we have to find the process sending the message, so we have to find the `spawn/3` function call that started the execution of function `fun1/0`. A `spawn/3` call is similar to the `apply/3` call, only it executes the given function in a new process: `spawn(Mod, Fun, Args)`.

A more complex query should be used if we want to detect the function applications of the `spawn/3` spawning the function `fun1/0`:

```
mods[name = "erlang"]
  .funs[(name = spawn) and (arity = 3)]
    .refs[type = application]
      [.param[index = 2]
         .origin[(type = atom) and
                 (value = fun1)]]
```

This query first selects the `erlang` module, then selects the function from this module with the given name (`spawn`) and arity (3). Then *.refs* identifies the references of function `spawn/3` and filters out the function calls (applications) from the result. This filtering is necessary, because there are other types of function references, such as export or import list references to the function. The selector *.param* means that we want to query the parameters of an application. Specially, in this query, we need the second parameter of the application, so we select only the second parameter, which is the name of the called function. Then, by using the *origin* selector we can query the origin value of the parameter and filter out the atoms with value `fun1`. We determine the origin values of the expression *e* with data-flow analysis, thus the origin expressions are that expressions which value can flow into expression *e*. (The *query[.query]* notation stands for the embedded queries. This special filter is true if the result set of the embedded query is not empty.)

The presented semantic query identifies the `spawn/3` function calls. The return value of the function is the process identifier (Pid) of the newly created process. We have to use forward data-flow reaching (*@reach* from Pid) to select the expressions where Pid identifies the recipient of a message passing: `Pid ! SomeMsg`. We can filter the result to select only the sent messages using the following query: *@reach.top[type=send_expr].sub[index=2]*.

```
mods[name = "erlang"]
  .funs[(name = spawn) and (arity = 3)]
    .refs[type = application][.param[index = 2].origin[(type = atom) and (value = fun1)]]
      .reach.top.[type=send_expr].sub[index=2]
```

Fig. 3 Finding received messages

Since the result of the spawn/3 call is the process identifier, you can concatenate the two queries: Figure 3.

We should note here that executing this query could return no result because not the fun1/0 was spawned, but the function that calls function fun1/0:

```
parallel() ->
    spawn(mod, fun2, []).

fun2() ->
    ...
    fun1(),
    ...
```

In this case we have to query the revers call chain from fun1/0 using *@fun.(called_by)+*. In our example the result contains fun2/0, and we should replace the name fun1 with fun2 in the query on Figure 3.

4.4. Identify Callback Functions

The Erlang/OTP [12] framework provides a set of applications and standards designed to help the Erlang developers to build new applications. The gen_server behaviour of OTP was designed to provide the generic framework of a client-server application, such as starting the server, communicating with clients via message passing, stopping the server, etc. The specific part of the client-server application should be implemented in a, so-called, callback module. For instance, the callback module implements the specific functionality of the server: what messages are allowed to be send, what is the response of the server for a specific message, etc. The developer can use gen_server:call/3 or gen_server:cast/3 for synchronous and asynchronous messages sending to the server, and all of these messages have to be handled in a handle_call/4 or handle_cast/3 functions in the callback module implementing the specific behaviour and response of the server.

The query language is also applicable for detecting typical programming mistakes. One of them is when a programmer uses the gen_server behaviour and forgets to handle all of the messages or mistypes a message tag. The user can first query the messages handled in callback functions (2), then the messages used in gen_server calls (1) and compare them manually.

```
(1) mods[name = "gen_server"]
      .funs[(name = call) and (arity = 2)]
        .refs[type = application]
          .param[index = 3]
```

```
(2) mods[name = "CallBackMod"]
      .funs[(name=handle_call) and (arity=3)]
        .args[index = 1]
```

4.5. Bad Smell Detection

Static analysis and the semantic query language could help in bad smell detection, too.

For instance, the Semantic Query Language can be used for dead code detection. In most cases the dead code fragment is an unused function (1) or an unused variable (2):

```
(1) mods.funs[not .refs[type=application]]}
```

```
(2) mods.funs.vars[not .refs]
```

An other type of bad smells could be those server requests which are unnecessarily synchronous, and it could be rather asynchronous. That occurs in case of gen_server usage, when we use the gen_server:call/3 function and its return value is ignored. This is a synchronous server request, which waits for a response from the server side; sometimes it is necessary for synchronization, but in most cases the only reason is to obtain the result. You can use the following query to check whether the return value of the result is used anywhere:

```
mods[name=gen_server]
  .funs[(name=call) and (arity=3)]
    .refs.reach
```

The inefficient usage of the ++ operator can be a bad smell code, too. This operator when concatenates two lists, makes a copy of its left hand side operand. When this operator is repeated during list processing, and its left hand side operand is an ever growing list, this list will be copied again and again. This kind of suspicious code could be inefficient in case of long lists. The following query helps to find the suspicious lists, but the length of the left hand side list depend on the actual value of the list:

```
mods.funs
  .exprs[value='++']
    .sub[index=1][.origin[type=cons]]
```

4.6. Complexity Metrics

The cost of posterior changes and modifications in the program code depends highly on the structural complexity of the source code. Measuring complexity is important, as it can indicate the weaknesses of the program, or it can reveal, at an early phase, that testing is unattainable, or its cost is too high. Measuring complexity also helps us to give an estimation of the cost of servicing and changing the source code.

Numerous structural complexity metrics have been developed for Erlang in RefactorErl. A query language for calculating complexity metrics on the source code is embedded in the RefactorErl framework [4], but the structural complexity metrics are also available through the semantic query language. The metrics appear as *properties* for modules and functions. The usage of these metric properties is identical with the usage of the other properties, accordingly we can use a metric to query directly its value, furthermore we can build a filter expression with its help.

An important field of applicability of metrics is checking design rules. There are simple rules, such that

- **Rule1:** *No module should contain more then 1000 LOC* or
- **Rule2:** *Do not use more than 80 characters in a line*.

The queries *mods[line_of_code* $\geq$ *1000]* and *mods.funs[max_length_of_line* $\geq$ *80]* return the modules and functions breaking the previous rules.

There are some other more complex design rules to follow. To check these rules manually is s not straightforward, for instance:

- **Rule3:** *Every function is tail recursive*.

You can select the functions breaking this rule using the following query: *mods.funs[is_tail_recursive=non_tail_rec]*.

5. WEB BASED INTERFACE FOR SEMANTIC QUERIES

Besides the Emacs [10] integration, RefactorErl includes an interactive and a scriptable shell interface that allow the user to run and display semantic queries. In the latter interface, query results are able to be utilized in further computations and other queries. In addition, there is a quite different, third option to run semantic queries: the web-based interface, due to its server-client manner, allows many programmers to access semantic information simultaneously via web browser.

Technically, the refactoring tool along with a web server run on a dedicated server computer and they communicate each other. By this, a possibly large amount of common source code can be loaded into the tool running on a high-performance server and the semantic content becomes available to all the programmers. The centralised solution allows arbitrarily many clients to execute semantic queries on a shared code base. The web server passes the query requests to the tool, and the received results are sent back to the clients via HTTP.

The web based interface has many more benefits and implements additional functionality. For instance, an *expression editor* guides us in composing queries, by supplying with suggestions of possible continuations. For example, when examining dynamic connections of a given function and typing

```
mods[name==foo].funs[name==bar].dyn
```

the editor suggests us to continue either with *dynrefs*, *dyncalls*, or *dyncalled_by*. This autocomplete mechanism helps new RefactorErl users to use the language, and also for all developers to speed up query construction and avoids constructing wrong queries.

There is support for saving query strings as well as for storing and displaying results of previously run queries. Stored queries can be browsed and re-run. When one appoints a query for re-running, the tool checks whether the code has been modified since the time of save: if no changes have been made then the stored result is displayed, otherwise the query gets indeed executed. Stored queries are shared and are available for every client, however, logged in users are able to hide queries belonging to others and only browse own data. After the result of a query appears, one can select resulting entities to be displayed: the source code of the containing module is shown and the specific entity is highlighted.

Last but no least we have the opportunity to visualize the result of a frequently used query: *mods.funs.(calls)+*. Its result contains all the possible call chains of the program. A proper composition of these call chains describes the call graph of the program. The tool can construct a graph description from the result of the former query, which can be visualised on the web page. Cyclic call dependencies (marked with * in the textual result) are colour highlighted in the graph, making the call graph comprehension easier.

6. FURTHER SOURCE CODE ANALYSIS WITH REFACTORERL

Besides performing refactoring transformations and performing semantic queries the RefactorErl framework can be used to perform further analysis on the Erlang source code. Based on the source code representation and the additional semantic information in the RefactorErl graph model module and function clustering [7] can be performed.

6.1. Clustering

In large projects it is common that the structural complexity of the software becomes unmaintainable. It can be avoided by design decisions in the right time, otherwise design failures must be fixed with posterior corrections. Sometimes the code grows to the magnitude that no programmer is fully aware of the structure of the program code. Such code is hardly maintainable, therefore the entities must be grouped into smaller sets, called clusters, so that each cluster is small enough to be maintained effectively. The RefactorErl calculates a set of possible groups based on the semantic connections of the program entities and makes a suggestion for the user. If the user accepts the actual suggestion, using the functionality of the tool the user can perform the transformation.

6.2. Cyclic dependency analysis

Besides the call chain calculation and visualisation (Section 5) the tool provides a platform for dependency and cyclic dependency calculation among functions and modules. Module *lime* depends on module *plum* , if there is a function *pear* in *lime* and a function *apple* in *plum* where *pear* depends on *apple*. Function *pear* depends on function *apple*, if *pear* calls *apple*. The result of this dependency analysis can be visualised in a graph. The nodes of the graph are modules and function, the edges of the graphs are the dependencies. The cyclic dependency is a directed cycle in the graph. There are different levels of analysis: one results only module dependencies, the other results both module and function dependencies.

7. RELATED WORK

Software maintenance in different aspects of a software life-cycle is a well know topic. A number of papers deal with the problem of software evolution [5], but just a few from that with the presence of code comprehension [1].

Understanding and maintaining the source code written by other developers in large software project is nerve-racking, time consuming and seems almost impossible. This task can be speeded up by using a tool which performs static analysis on the source code and strain off only the relevant information for the user. Several current projects aim to develop a code understanding tool for different programming languages. These projects are related mainly with OOP programming languages or C, and most of them have been developing a commercial product. For instance, the tool Understand [14] is a source code comprehension and documentation tool for C/C++. It provides support for showing call hierarchies, call/callby trees and include/includedby trees and also for HTML documentation generation. CodeSurfer [8] is a similar commercial program-understanding tool for C/C++ that makes manual review of code easier and faster. It provides an interactive browser to show the result of different static analysis, such as data-flow analysis, control dependency analysis, impact analysis, etc.

8. CONCLUSION

Hence code comprehension has an influence on software maintenance a tool supporting code comprehension is useful in everyday life programming. In this paper we introduced the different layers for Erlang code representation, and then we focused on the semantic query language designed for querying different kinds of information about the Erlang programs. Then we presented industrial motivated problems and semantic queries to help in solving them and to show the usability of static source code analysis with RefactorErl.

These queries based on semantic relationship of Erlang programs, so they could result a more accurate and possibly smaller result set than a manually performed string pattern matching. At last but not at least, using queries is less time consuming than searching manually.

ACKNOWLEDGEMENT

The Semantic Query Language was designed by the RefactorErl group. We want to thank for Lászlo Lövei for the initial design ideas and Lilla Hajós for the current implementation.

REFERENCES

[1] ETZKORN, L. H. – BOWEN, L. L. – DAVIS, C. G.: *An Approach to Program Understanding by Natural Language Understanding*, Natural Language Engineering, vol. 5, no. 1, 1999, pp. 1–18.

[2] FOWLER, M. – BECK, K. – BRANT, J. – OPDYKE, W., – ROBERTS, D.: , Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[3] HORVÁTH, Z. – LÖVEI, L. – KOZSIK, T. – KITLEI, R. – NAGY, T. – VǴH, A. – TÓTH, M. – KIRÁLY, R.: *Modeling semantic knowledge in Erlang for refactoring*, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Cluj-Napoca, Romania, Studia Universitatis Babeş-Bolyai, Series Informatica, vol. 54, Sp. Issue, 2009.

[4] KIRÁLY, R. – KITLEI, R.: *Application of complexity metrics in functional languages*, In Proceedings of the 8th Joint Conference on Mathematics and Computer Science, Komrno, Slovakia, Jul. 2010.

[5] KOLLÁR, J. – PORUBÄN, J. – VÁCLAVÍK, P. – BANDÁKOVÁ, J. – FORGÁC, M.: *Adaptive Language Approach to Software Systems Evolution*, International Multiconference on Computer Science and Information Technology: 1st Workshop on Advances in Programming Languages (WAPL'07), Wisla, Poland, October 15–17, Polish Information Processing Society, 2007, 2, pp. 1081–1091.

[6] LÖVEI, L. – HAJÓS, L. – TÓTH, M.: *Erlang Semantic Query Language*, In Proceedings of the 8th International Conference on Applied Informatics, ICAI 2010, Eger, Hungary, Jan. 2010.

[7] LÖVEI, L. – HOCH, C. – KÖLLÖ, H. – NAGY, T. – NAGYNÉ-VÍG, A. – HORPÁCSI, D. – KITLEI, R. – KIRÁLY, R.: *Refactoring Module Structure*, In Proceedings of the 7th ACM SIGPLAN workshop on Erlang, pages 8389, Victoria, British Columbia, Canada, Sep. 2008.

[8] GRAMMATECH CodeSurfer Homepage, http://www.grammatech.com/products/codesurfer/ overview.html

[9] EDoc, http://www.erlang.org/documentation/doc-5.4.2.1/lib/edoc-0.1/doc/html/index.html

[10] GNU Emacs Homepage, http://www.gnu.org/ software/emacs/

[11] Erlang Homepage, http://www.erlang.org

[12] Erlang OTP documentation, http://www.erlang.org/doc

[13] RefactorErl, https://plc.inf.elte.hu/erlang

[14] Understand - Source Code Analysis & Metrics, http://www.scitools.com/ucpp.html

[15] World Wide Web Consortium: XML Path Language (XPath), Version 1.0. W3C Recommendation, Nov. 16, 1999. http://www.w3.org/TR/xpath/

Received June 13, 2011, accepted September 12, 2011

BIOGRAPHIES

Melinda Tóth is a second year PhD student. In 2009 she graduated (MSc) with distinction at the Faculty of Informatics at Eötvös Loránd University. She has been working with Erlang since 2007 with the RefactorErl project. Both her bachelor and master theses were based on Erlang and function related refactorings. Her PhD research field is about data and control flow graphs for functional languages, and impact analysis of refactorings.

István Bozó is a second year PhD student. He received his master's degree in Computer Science in 2009 from Eötvös Loránd University. Both his bachelor and master theses were focused on Erlang and function related refactorings. His PhD research field is impact analysis of Erlang programs and measuring the impact of refactoring source code transformations, and test case selection.

Judit Kőszegi is a first year PhD student at the Eötvös Loránd University. In 2010 she graduated (MSc) at the Faculty of Informatics, Eötvös Loránd University, Budapest. Her scientific research is focusing on static analysis and formal verification of programs written in functional languages, she plans to defend her PhD thesis in 2013.

Zoltán Horváth is Professor at, and Head of, the Department of Programming Languages and Compilers and Vice-Rector for International Affairs at Eötvös Loránd University in Budapest, Hungary. He defended his habilitation thesis in 2004; the title of his thesis was "Verification and Semantics of Mobile Code Written in a Functional Programming Language". Current topics researched under his supervision include language design, construction of programming language processing tools, formal methods and scheduling in grids. He has been supervising the RefactorErl project since 2005.