*Proceedings of the 8th International Conference on Applied Informatics*
*Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 165–172.*

# Erlang Semantic Query Language*

**Lilla Hajós, Melinda Tóth, László Lövei**

Department of Programming Languages and Compilers,
Eötvös Loránd University, Budapest, Hungary
e-mail: {hajos,toth_m,lovei}@inf.elte.hu

**Abstract**

Under developing a software when somebody tries to fix bugs or to understand an existing program source, it could be very useful if somebody (something) can answer to some simple or difficult question about the source code. That questions are not about the syntax rather about the semantics of the program. For example, when somebody tries to fix a bug a very simple question could be useful: what is the value and where it is bound to a variable. In a huge code-base searching it manually is not straightforward especially in case of dynamically typed languages, like Erlang. This paper introduces a semantic query language to get information about Erlang programs and to help with software development. We achieve this goal by giving the developers the means to query information they need, that is too complicated or simply too time-consuming to get by conventional ways. This paper gives an overview of the language we designed for this purpose and some use cases.

## 1. Introduction

codes we may need to look up large amounts of information to complete our tasks. Most of the time, the information we need is simple. We try to search for positions where we access a record, for the calls of a function etc. In case of a simple code most of the time you do not need to write queries to get these kind of information. As soon as you need some combination of these simple queries, it can be substantially more complicated to get the expected results manually. And even when you have simple queries, it can be useful to see the collected results and to access it easily.

Moreover it is possible to solve more complicated problems by using simple queries. You can query for example the occurrences of functions that came from a given module. It is a simple query in the language, but if you have some functions imported it can be quite time-consuming to look up every such occurrence.

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

The paper is structured as follows. Section 2 briefly describes the RefactorErl tool. Section 3 and Section 4 describe the language which was designed to query information about Erlang programs and present some use cases. Section 5 gives related work, then Section 6 concludes the paper.

# 2. RefactorErl

The phrase "refactoring" (introduced by Fowler [3]) means a meaning preserving source code transformation, so while you change the program structure you do not alter its behaviour. RefactorErl [4] was built to refactor Erlang programs. Erlang [5] is a dynamically typed functional programming language, thus to gather all of the necessary information for a behaviour preserving transformation is not straightforward. Therefore a semantic program graph was built to store the Erlang source code and the information could computed by static analysis. The semantic graph is a three layered model. It has a lexical layer, which contains the token information about the source. The syntactic layer based on an abstract syntax tree. After scanning and parsing the program different semantic analyzers add the static semantic information to the graph. The resulted graph is a rooted, directed, labeled graph with typed nodes and edges. Querying semantic information about the graph is possible through *path expressions*: you should give the starting node of the query and the sequence of links you want to follow. These queries are good for querying information for those programmers who try to implement refactorings in the system, but they are strongly depend on the graph representation.

# 3. Semantic query language

RefactorErl stores a lot of syntactic and static semantic information about the program source, so based on that we call RefactorErl as a source code analyzer and transformer tool. We try to show the result of those analysis to the Erlang programmers, thus we have developed a Semantic Query Language to query information about the source code. The language is related to the semantic elements of the Erlang language and it does not require any knowledge about our representation (Thus a programmer can easily use it).

## 3.1. Language concepts

When we try to get information about programs, usually we are looking for files, functions, variables etc. or some properties of these. We designed our language to handle these semantical units.

**Entities** Entities correspond to the semantic units of Erlang. The result of a query written in the language is a set of entities. Each element of a set belongs to the same type. We have the following entity types defined: `file, function,`

`variable`, `macro`, `record`, `record field` and `expression`. Each entity type has a set of selectors and properties defined for them. You can query information about specific entities with the help of these.

**Selectors** Selectors are binary relations between entities. The entities belong to one of the seven entity types. A selector selects a set of entities that meet given requirements for each entity. For example you can select the functions defined in a given file. In that case the selection is a relation between files and functions.

**Properties** Another way to get information about entities is to query one of their properties. For example you can query about a function whether it is exported or not. The main use for properties is that filters can be built with their help.

**Filters** A filter is a boolean expression built mainly using properties. Using a filter means selecting the subset of entities the filter holds true for. For example you may be interested in all the exported functions of a given file, or the functions with 0 arity, or maybe a combination of these: the exported functions with 0 arity. In the example exported and arity are both properties of functions and by using them it is possible to build a filter to select the required subset of functions.

## 3.2. Syntax and semantics

The query language was designed to write semantic queries. As of now they can not be combined, so the description of the language is the equivalent of the description of the semantic queries themselves.

### 3.2.1. Semantic queries

```
semantic query ::= initial selection ['.'  query sequence]
   query sequence ::= query ['.'  query sequence]
   query ::= selection | iteration | closure | property query
```

A semantic query written in the language consists of an initial selection and a sequence of queries. The initial selection and the queries are each separated by a full stop. A query can be selection, iteration, closure or property query. Queries operate on entities.

### 3.2.2. Initial selection

```
initial selection ::= initial selector[ '[' filter ']' ]
```

An initial selection consists of an initial selector and optionally a filter. Initial selectors get the current file and position as their parameters and return a set of entities as result. The entities of the result belong to the same type, but the type can not always be determined in advance, it depends on the parameters. Almost

all of them begin with the character `@` to indicate that they depend on a position.

Examples:

- `@variable`[1] looks for a variable at the given position. If no variable can be found the result will be empty.
- `modules` gives the entities representing the modules loaded into the semantic program graph of RefactorErl.
- `@definition` gives the entity that fits the best at the given position.

### 3.2.3. Selection

```
selection ::= selector [ '[' filter ']' ]
```

A selection consists of a selector and optionally a filter. A selector is a binary relation between two sets of entities. The elements of a set belong to the same entity type.
$selector \subseteq entitytype_1 \times entitytype_2$
$entitytype_i \in \{file, function, variable, macro, record, recordfield, expression\}$

Examples:

- The file entity has a selector named records which has the return type record.
  $records \subseteq file \times record$
  You can query the records of a given file by writing the `"@file.records"` query.

### 3.2.4. Iteration

```
iteration ::= '{' query sequence '}' int [ '[' filter ']' ]
```

Iteration in the language means the repeated application of a query sequence. The queries are relations and a sequence of queries is a composition of these queries. Using iteration is possible if the domain and codomain of the query sequence is the same. The application is repeated exactly *int* times.

The result shown in this case is not only the result of the iteration but the partial results also, in the form of chains. This way it is possible to gain additional information about programs.

Examples:

- The result of the semantic query `"@function.{calls}3"` is the same set of entities as of `"@function.calls.calls.calls"`. The result shown in

[1] `@var` can also be used instead of `@varialble`. Abbreviations of the names of initial selectors, selectors and properties can be used where the abbreviations are unambiguous.

the first case will give more information: it gives the call chains with the maximum length of 3 starting from a given function.

### 3.2.5. Transitive closure

```
closure ::= '(' query sequence ')' int [ '[' filter ']' ]
   closure ::= '(' query sequence ')+' [ '[' filter ']' ]
```

Transitive closure in the language means the closure of a query sequence. The query sequence here is the same as in iteration, a binary relation with the same domain and codomain. Using the denotation $R$ for this relation the meaning of the transitive closure:
$R^0 = R$
$R^i = R^{i-1} \cup \{(r_1, r_3) | \exists r_2 \colon (r_1, r_2) \in R^{i-1} \wedge (r_2, r_3) \in R^{i-1}\}$
$R^+ = \bigcup_{i \in \mathbb{N}} R^i$.
The result shown after a transitive closure is the same as the result shown after iteration.

Examples:

- The result of the semantic query `"@function.(calls)3"` is the same set of entities as of `"@function.calls` $\cup$ `@function.calls.calls` $\cup$ `@function.calls.calls.calls"`. The results shown in the first case are the call chains with the maximum length of 3 starting from a given function.
- The result shown after the semantic query `"@function.(calls)+"` is the list of all possible call chains starting from a given function.

### 3.2.6. Property query

```
property query ::= property [ '[' filter ']' ]
```

A property query consists of a property and optionally a filter. Properties are functions that give the value of the property for an entity.
$property \colon entitytype \rightarrow valuetype$
$valuetype \in \{atom, string, int, bool\}$
The main purpose of properties is to filter sets of entities using them, but their values can be queried too. To query the value of a property you have to use the name of the property at the end of a semantic query. It is possible to use them somewhere else in the query sequence, but in that case they are simply skipped. Filters written after a property is applied to the working set of entities.

Examples:

- *The usual way to use a property query:* Query the value of the property *exported* for the *function*s of the given *file*: `"@file.functions.exported"`

- *Filters after properties:* The following semantic queries will give the same result: `"@fun.calls.arity[bif]"` and `" @fun.calls[bif].arity"`. The result of this query is the *arity* – number of arguments – of the *bif*s – built in functions – *call*ed from a given *function*.

- *Properties within the query sequence:* The semantic queries `"@file.path.functions"` and `"@file.functions"`, give the same result, because the property *path* is skipped. The queries give the local *function*s of the given *file*.

### 3.2.7. Filters

A filter is a boolean expression to select subsets of entities. After applying a filter, the result contains the elements of the original set where this boolean expression is true. Building filters is possible using *atoms*, *strings*, *integers*, *properties* and *embedded queries*.
The use of *strings* and *integers* is unambiguous, but the names of *properties* are *atoms*, so it is checked for each *atom* if they are *properties* or not.
*Embedded queries* can be used to query information about entities that is otherwise unavailable, that is it can not be expressed by the help of *properties*. For example we may need the functions with variables named *File*. This information can not be expressed with the help of properties. Without embedded queries it is only possible to query the variables named *File* and query the functions containing these variables after that, with the following query:
`"@file.functions.variables[name=="File"].function_definition"`
Embedded queries make it possible to use these kind of queries effectively, without the need to continue with the query directly. The continuation of the query is in the filter, used like a property with a boolean value. The value is considered true if the result of the query is not empty. For the previous example using the query `"@file.functions[.variables[name=="File"]]"` will give the desired results.

*Atoms*, *strings*, *integers* and *properties* can be used in comparisons. The language uses `/=`, `==`, `>=`, `=<`, `<` and `>`. The results of comparisons are the same as in Erlang.

The resulting expressions can be combined by `and`, `or`, and `not` operators, and parentheses can be used, too. The operator precedence for the filters is as follows:

| Operator precedence (decreasing) | |
|---|---|
| `not` | unary |
| `/=`, `==`, `>=`, `=<`, `<`, `>` | left associative |
| `and` | left associative |
| `or` | left associative |

# 4. Case studies

We have been developing this language to help software developers.

### 4.1. Looking up information

The simplest way of using the language is to speed up simple, but time-consuming processes. For example the looking up of informations.

`@record.references :` A list of references to a given record. – The summarized result can be more useful than looking up every such occurrence by hand.

`@def :` Goto definition. – A useful query widely used in other languages too.

`mods[name==io].functions.references :` References to the functions of the *io* module. – While debugging or testing new codes functions from the *io* module are often used. Queries like this one can help with the cleaning up of the code.

`@function.references :` References of a given function. – This query can be useful for looking up occurrences of deprecated functions.

### 4.2. Easier understanding of codes

The language can be used to help with the understanding of codes. While working with complicated codes collected results can give a better overview of the code as a whole. Using the language it is possible to comprehend the connection within the code easier.

`@file.functions :` The functions of a file. – The collected result of this query may give an easier overview of a module.

`@file.functions.calls.module :` The modules that use the functions of a given file. – This query may help understanding the dependencies between modules.

`@function.references :` The references to a given function. – To help understanding the given function.

`@function.(called_by)+ :` The call chains starting from a given function. – Information about the structure of a code.

## 5. Related work

Erlang has a cross reference tool called *Xref* which is similar to our semantic query language. *Xref* uses information extracted from BEAM files to analyze dependencies between functions, modules, applications and releases. Our language has a different approach. We work with the source codes, there is no need to compile the files to the analysis (however it is possible to load beam files to the semantic program files), but we don't work with either applications or releases. We can also query informations about functions and modules , but these are not the only semantic informations we can query about programs. In this respect we can get more information and the use of our language is quite simple too.

# 6. Conclusion

We developed a language with the needs of software developers in sight. Software development and maintenance requires an extensive knowledge of the semantics of codes. As keeping in mind all of the information is impossible, we need to look for it every time the need arises.

The semantic query language was designed to help software development by making semantic information about codes easily accessible. We can use the results of queries to:

- Speed up some simple, but time-consuming processes. For example we can look up semantical information about function, records etc. easily.
- Fix bugs. It is possible to write queries that give information that is otherwise quite difficult to gain.
- Understand the structures and dependencies within codes.

We also considered the simplicity of the syntax a goal, to make the use of it as simple as possible. This way learning requires only a little effort from the user.

# References

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

[2] World Wide Web Consortium: *XQuery 1.0: An XML Query Language,* W3C Recommendation 23 January 2007, `http://www.w3.org/TR/xquery/`

[3] Fowler, M. and Beck, K. and Brant, J. and Opdyke, W. and Roberts, D.:, Refactoring: Improving the Design of Existing Code, Addison-Wesley (1999)

[4] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Bozó, I., 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(2009) Sp. Issue

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