RefactorErl: a source code analyser and transformer tool [1]

Melinda Tóth

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University

RefactorErl team:
István Bozó, Dániel Horpácsi, Zoltán Horváth, Roland Király,
Róbert Kitlei, Tamás Kozsik, Judit Kőszegi

November 15, 2010

[1] Supported by KMOP-1.1.2-08/1-2008-0002, ELTE IKKK, and Ericsson Hungary

Outline

1. Introduction
   - History
   - Design goals
2. Architecture
   - Model
   - Implementation
3. Use cases
   - Refactoring
   - Analysis

History

- Original idea: SQL based refactoring (Clean)
- Research on Erlang refactoring (Ericsson Hungary)
- Experiments
  - MySQL, standard parser and pretty printer
  - Mnesia, custom parser, whitespace preservation
- Real-world applications for analysis

Design goals

1. Store semantic information instead of calculating each time
   - Efficient retrieval – graph model
   - Incremental analysis
2. Provide a platform for source code transformation
   - Generic solutions are preferred
   - Non-refactoring applications

Requirements

- Support of large code bases
- Language coverage
- Comment preservation
- Layout preservation (indentation)

Three-layered graph model

1. Lexical level
   - Tokens
   - Preprocessing
   - Comments, whitespace
2. Syntactic level
   - Abstract Syntax Tree
   - Files
3. Semantic level
   - Module, function, record, variable nodes
   - Links to definition and reference points

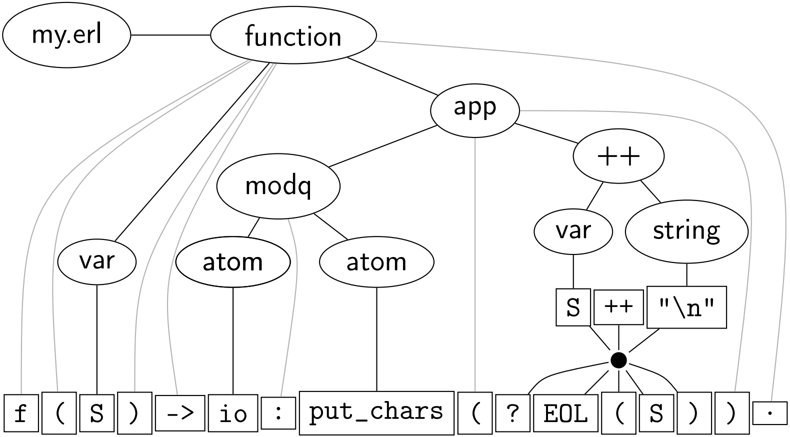
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```

```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```

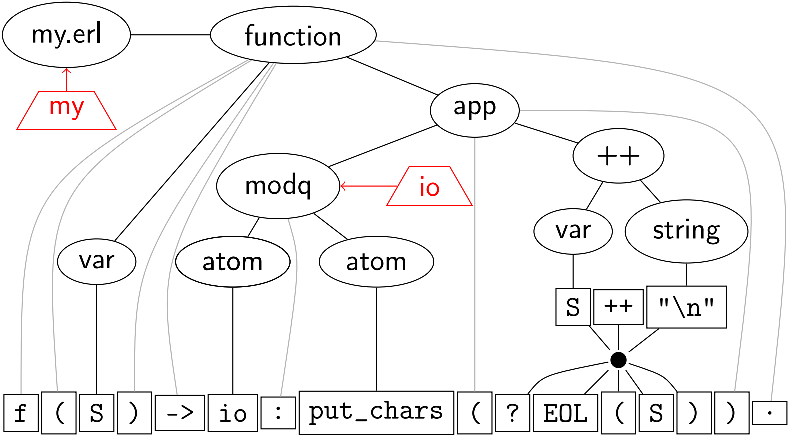
```
f ( S ) -> io : put_chars ( ? EOL ( S ) ) .
```

```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```

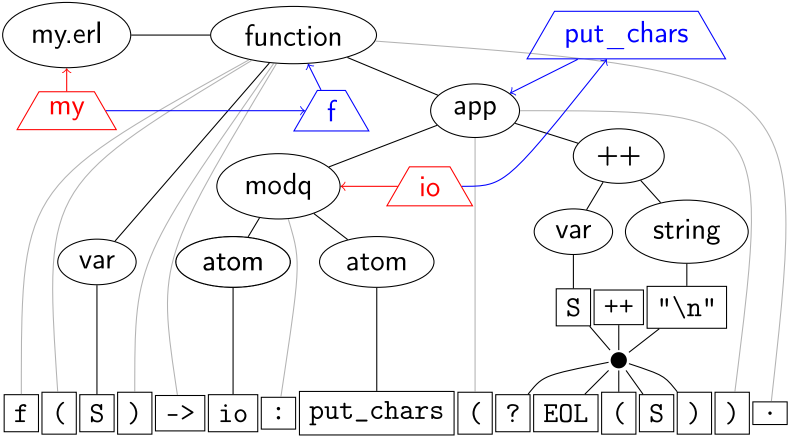
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



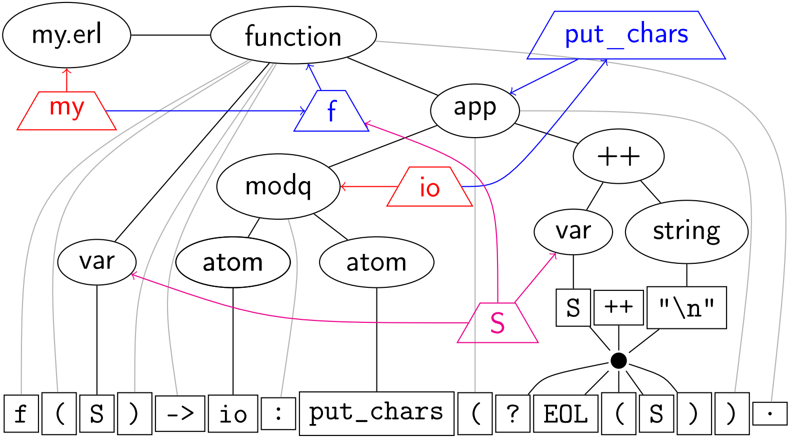
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



Graph storage and manipulation

- Nodes and edges are stored in Mnesia tables
  - Node attributes: token text, variable name, ...
  - Edge labels: subexpression, variable reference, ...
- Graph path: filtered edge label sequence
  - Edges are indexed by labels
  - Cost doesn't grow with code size
- Frequently used queries need only fixed length paths
- Syntax tree based transformations, automatic token handling
- Incremental semantic analysis is triggered by syntactic changes

Further details

- Extended syntax description
  - Defines the representation
  - Source for parser, lexer, and token updater
- Analyser framework
  - Extensible, modular structure
  - Asynchronous, incremental (7 times faster initial loading, Intel Core2 Quad, 2.4 GHz)
  - Contains side-effect analysis and pretty-printing
- Generic user interface support
  - GNU Emacs, XEmacs
  - Interactive and Scriptable Erlang shell interface (`ri & ris`), CLI, Web

Refactoring steps

Rename

- variable
- function
- record, record field
- macro
- module/header file

Function interface

- introduce function argument
- reorder parameters
- introduce tuple
- eliminate/introduce import

Move definition

- macro
- record
- function

Expression structure

- eliminate/introduce variable
- eliminate/introduce function
- eliminate macro application
- eliminate fun-expression

Data structure

- Introduce record
- Upgrade module interface

Clustering

- Code restructuring based on component relations
  - Function calls
  - Record and macro usage
- Module clustering
  - Split a large block of modules to more manageable parts
  - Involves splitting of header files
- Function clustering
  - Split a large module into smaller parts
  - Refactoring: move function

Semantic query language

- A language to get information about the Erlang source
- Language concepts:
  - Entities
  - Selectors
  - Properties
  - Filters
- Example:

  `mods[name==mymod].funs[name==myfun].calls`
- Custom query or predefined query

Syntax of semantic queries

- `semantic_query ::= initial_selection ['.' query_sequence]`

Syntax of semantic queries

- `semantic_query    ::= initial_selection ['.' query_sequence]`

- `initial_selection ::= initial_selector ['[' filter ']']`
- `query_sequence    ::= query ['.' query_sequence]`
- `query             ::= selection | iteration | closure |`
  `property_query`

Syntax of semantic queries

- `semantic_query    ::= initial_selection ['.' query_sequence]`

- `initial_selection ::= initial_selector ['[' filter ']']`
- `query_sequence    ::= query ['.' query_sequence]`
- `query             ::= selection | iteration | closure | property_query`

- `selection         ::= selector ['[' filter ']']`
- `iteration         ::= '{' query_sequence '}' int ['[' filter ']']`
- `closure           ::= '(' query_sequence ')' int ['[' filter ']'] '(' query_sequence ')+' ['[' filter ']']`
- `property_query    ::= property ['[' filter ']']`

Semantic query examples

- Value of a variable
  `@expr.origin`

Semantic query examples

- Value of a variable
  `@expr.origin`
- Call chain
  `@fun.(called_by)+`
  `@fun.(calls)+`

Semantic query examples

- Value of a variable
  `@expr.origin`
- Call chain
  `@fun.(called_by)+`
  `@fun.(calls)+`
- Side effect
  `@fun.dirty`

Semantic query examples

- Value of a variable
  `@expr.origin`
- Call chain
  `@fun.(called_by)+`
  `@fun.(calls)+`
- Side effect
  `@fun.dirty`
- Dynamic function calls
  `@fun.refs`
  `@fun.dyncall`
  `@fun.dyncalled_by`
  `@expr.dynfuns`

Semantic query examples

- Value of a variable
  `@expr.origin`
- Call chain
  `@fun.(called_by)+`
  `@fun.(calls)+`
- Side effect
  `@fun.dirty`
- Dynamic function calls
  `@fun.refs`
  `@fun.dyncall`
  `@fun.dyncalled_by`
  `@expr.dynfuns`

- Sent messages

```
mods.funs.exprs
  [.sub[index==1 and value==upd]
    and type==send_expr]
```

Semantic query examples

- Value of a variable
  `@expr.origin`
- Call chain
  `@fun.(called_by)+`
  `@fun.(calls)+`
- Side effect
  `@fun.dirty`
- Dynamic function calls
  `@fun.refs`
  `@fun.dyncall`
  `@fun.dyncalled_by`
  `@expr.dynfuns`
- Sent messages

```
mods.funs.exprs
  [.sub[index==1 and value==upd]
     and type==send_expr]
```

- Received messages

```
mods.funs.exprs[type==receive_expr]
  .sub[.sub[index==1 and value==upd]
     and type == tuple]
```

Identifying callback functions

```
request_add(...)->
  gen_server:call(Server, {req_add, {Phone, Name}}).

handle_call({req_add, {Phone, Name}}, From, LoopData) ->
  ...
```

Callback functions

- ```
  mods[name == gen_server].
   funs[name == call and arity == 2].
   refs[type == application].
   param[index == 2]
  ```
  ```

Identifying callback functions

```
request_add(...)->
  gen_server:call(Server, {req_add, {Phone, Name}}).

handle_call({req_add, {Phone, Name}}, From, LoopData) ->
  ...
```

Callback functions

- ```
  mods[name == gen_server].
   funs[name == call and arity == 2].
   refs[type == application].
   param[index == 2]
  ```
- ```
  mods[name == "CallBackMod"].
   funs[name == handle_call and arity == 3].
   args[index == 1]]
  ```

Metrics – Checking coding conventions

- Measure the structural complexity of Erlang source code
- `show max_depth_of_calling for module ('dyn')`
  `mods[name==dyn].max_depth_of_calling`
- Conventions:
  - **Rule1**: A module should not contain more then 400 lines
    `mods[line_of_code > 400]`
  - **Rule2**: Use at most two level of nesting, do not write deeply nested code
    `@file.funs[max_depth_of_cases > 2]`
    `mods[max_depth_of_cases > 2]`
  - **Rule3**: Use no more than 80 characters on a line
    `mods.funs[max_length_of_line > 80]`
  - **Rule4**: Use space after commas
    `mods.funs[no_space_after_comma > 0]`
  - **Rule5**: Every recursive function should be tail recursive
    `@file.funs.is_tail_recursive`

Further analysis

Module/function dependency analysis

- Available in console mode
- Checks cyclic dependencies among modules or functions
- Draws/prints dependencies and cyclic dependencies

Summary

- RefactorErl: source code analyser and transformer
- Refactoring: helps development
- Analysis: helps maintenance

http://plc.inf.elte.hu/erlang