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

Melinda Tóth and Zoltán Horváth

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

November 16, 2010

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

Outline

1. RefactorErl
   - Framework
   - The tool
2. Erlang Semantic Query Language
   - Path expressions
   - Language elements
   - Syntax
   - Examples
3. Summary

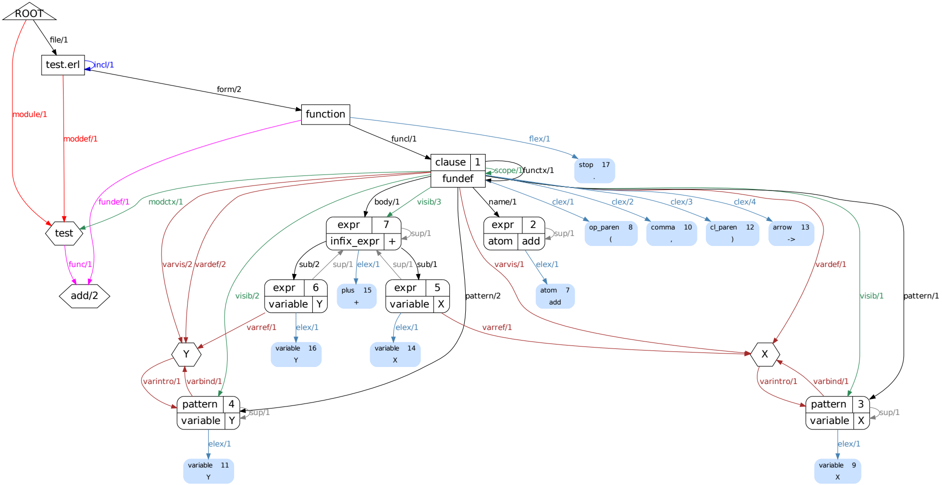
RefactorErl

- Semantic Program Graph:
  Lexical layer + AST + Semantic layer
- Efficient information retrieval
- Stored semantic information – Mnesia
- New semantic analyser framework
  - Incremental analysis
  - Modular structure
  - Asynchronous parallel execution
  - 7 times faster initial loading (Intel Core2 Quad, 2.4 GHz)
  - Side-effect analysis, data-flow analysis, dynamic function call analysis

The tool RefactorErl

- Platform for source code transformations
  - Rename
  - Move definition
  - Expression structure
  - Function interface
- Non-refactoring facilities – different analysis
  - Call graph visualisation
  - Dependency visualisation
  - Clustering
- Query Language
- UI: Emacs, Interactive/Scriptable Erlang shell interface, CLI, Web based

Example graph



Path expressions

- Support information gathering for refactoring
- Depend on the representation

```
path() = [PathElem]

  PathElem = Tag | {Tag, Index} | {Tag, Filter} |
             {intersect, node(), Tag}
  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       = '==' | '/=' | '=<' | '>=' | '<' | '>'
```

Path expression example

- List the functions defined in a module:

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

Path expression example

- List the functions defined in a module:

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

- Set of library modules

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

- Extended evaluation framework

Path expression example

- List the functions defined in a module:

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

- Set of library modules

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

- Extended evaluation framework

-
  ```
  exec(Module, reflib_module:functions())
  ```

Semantic query language

- A user-level query language for getting information about Erlang source code
- Language concepts:
  - Entities
  - Selectors
  - Properties
  - Filters
- Example:

```
mods[name==mymod].funs[name==myfun].calls
@file.funs[name==myfun].calls
```

- Custom query or predefined query

Syntax of the queries

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

Syntax of the 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 the 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

```
calc(...) ->
    A = ...,
    ...
    {A, ...}.

test(...)->
    Calc = calc(...),
    ...,
    {First, ... } = Calc,
     First.

run() ->
    some_value = test(...).
```

Semantic query examples

```
calc(...) ->
    A = ...,
    ...
    {A, ...}.

test(...)->
    Calc = calc(...),
    ...,
    {First, ... } = Calc,
     First.

run() ->
    some_value = test(...).
```

- Value of a variable
  `@expr.origin`

Semantic query examples

```
calc(...) ->
    A = ...,
    ...
    {A, ...}.

test(...)->
    Calc = calc(...),
    ...,
    {First, ... } = Calc,
     First.

run() ->
    some_value = test(...).
```

- Value of a variable
  `@expr.origin`
- Call chain
  `@fun.(called_by)+` or
  `@fun.(calls)+`
  `mods.funs[name==calc].(called_by)+`

Semantic query examples

```
calc(...) ->
    A = ...,
    ...
    {A, ...}.

test(...)->
    Calc = calc(...),
    ...,
    {First, ... } = Calc,
     First.

run() ->
    some_value = test(...).
```

- Value of a variable
  `@expr.origin`
- Call chain
  `@fun.(called_by)+` or
  `@fun.(calls)+`
  `mods.funs[name==calc].(called_by)+`
- Side effect
  `mods.funs.dirty`

Dynamic function calls

```
sum([]) ->
    0;
sum([H|T]) ->
    S = sum(T),
    H + S.

test1(List)->
    Fun = sum,
    test2(?MODULE, Fun, List).

test2(Mod,  Fun, List)->
    apply(Mod, Fun, [List]).
```

Dynamic function calls

```
sum([]) ->
    0;
sum([H|T]) ->
    S = sum(T),
    H + S.

test1(List)->
    Fun = sum,
    test2(?MODULE, Fun, List).

test2(Mod,  Fun, List)->
    apply(Mod, Fun, [List]).
```

- Function references

  `@fun.refs`

  `mods.funs[name==sum].refs`

Dynamic function calls

```
test1(List, ArgList)->
  Fun = sum,
  test2(?MODULE, Fun, [List]),
  test2(?MODULE, other, ArgList).

test2(Mod, Fun, Args)->
  apply(Mod, Fun, Args).
```

sum([]) -> ...

other(A) -> ...

other() -> ...

Dynamic function calls

```
test1(List, ArgList)->
  Fun = sum,
  test2(?MODULE, Fun, [List]),
  test2(?MODULE, other, ArgList).

test2(Mod, Fun, Args)->
  apply(Mod, Fun, Args).
```

sum([]) -> ...

other(A) -> ...

other() -> ...

- Dynamic function references
  `@expr.dynfun`

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]]
  ```

Checking coding conventions

- **Rule1**: A module should not contain more than 400 lines

```
mods[line_of_code > 400]
mods.funs[line_of_code > 20]
```

- **Rule2**: Use at most two levels of nesting – do not write deeply nested code

```
@file.funs[max_depth_of_cases > 2]
@file.max_depth_of_cases
mods[max_depth_of_cases > 2]
```

- **Rule3**: Use no more than 80 characters in 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

```
mods.funs[is_tail_recursive==non_tail_rec]
```

Embedded queries

Without:

```
mods.functions.
  variables[name=="File"]
    .fundef
```

With:

```
mods.functions
  [.variables[name=="File"]]
```

Other example:

```
mods.fun.refs
      [.sub[index==2 and type==tuple]
        .sub[index==1 and value==req_add]]
```

Summary and Future work

- RefactorErl: source code analyser and transformer tool
- Query language:
  - understand source code
  - debug information
  - maintenance
- Give a set of library functions for queries
- Extend the language (recursion, if, variables)

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