RefactorErl 0.9.12.01 Manual

Welcome to RefactorErl, the static source code analyser and refactoring tool for Erlang programs! This document will guide you through the basic usage of the software.

Contents

1 Installation and configuration **4**
- 1.1 Required software 4
- 1.2 Compilation 4
- 1.3 User interfaces 5

2 Using RefactorErl from the console **6**
- 2.1 General usage 6
- 2.2 Module and function dependencies. 12
- 2.3 Interface layers 18
- 2.4 Server management command list 19
- 2.5 Running dynamic call analysis 20
- 2.6 Transformation command list 21
- 2.7 Scriptable RefactorErl interface 22

3 Using RefactorErl in the command line **26**

4 Web interface **28**
- 4.1 Installation 28
- 4.2 Start up 28
- 4.3 Shutting down 29
- 4.4 Logging in 29
- 4.5 Semantic queries 29
- 4.6 Database operations and environmental nodes 33
- 4.7 Errors 34
- 4.8 Dependency examinations 35
- 4.9 Logging out 36

5 Old web interface for semantic queries **36**
- 5.1 Installation 36
- 5.2 Starting and stopping the web server 36
- 5.3 Usage 38

6 Querying semantic and syntactic information **39**
6.1 Semantic queries . . . 39
6.2 Metric queries embedded into semantic queries . . . 55
6.3 Metric queries . . . 57
6.4 Using metric queries from different interfaces . . . 62
6.5 Metric analyser mode . . . 62

7 Basic usage in Emacs/XEmacs **66**
7.1 Configuration in XEmacs/Emacs. . . . 66
7.2 RefactorErl mode . . . 67
7.3 The “Refactor” menu . . . 67

8 Using refactorings **72**
8.1 Rename function . . . 72
8.2 Rename header . . . 73
8.3 Rename macro . . . 74
8.4 Rename module . . . 75
8.5 Rename record . . . 77
8.6 Rename record field . . . 78
8.7 Rename variable . . . 79
8.8 Move function . . . 79
8.9 Move macro . . . 82
8.10 Move record . . . 83
8.11 Eliminate function call . . . 85
8.12 Eliminate macro substitution . . . 87
8.13 Eliminate fun expression . . . 88
8.14 Eliminate variable . . . 89
8.15 Introduce function . . . 90
8.16 Introduce import . . . 92
8.17 Introduce function argument . . . 93
8.18 Introduce record . . . 96
8.19 Introduce tuple . . . 97
8.20 Introduce variable . . . 99
8.21 Transform list comprehension . . . 100
8.22 Reorder function parameters . . . 103
8.23 Generate function specification . . . 104
8.24 Upgrade interface: regexp→re . . . 105

9 Clustering **107**
9.1 The Emacs interface for clustering . . . 107
9.2 Console interface of clustering . . . 108

10 Using RefactorErl in VIM **110**
10.1 Installation . . . 110
10.2 Menu and command structure . . . 110
10.3 Position based refactorings . . . 113
10.4 Interaction . . . 114
10.5 Semantic queries . . . 114

11 Using RefactorErl in Eclipse **115**
11.1 Installation . . . 115
11.2 Database management . . . 117
11.3 Executing refactoring transformations . . . 117
11.4 Executing semantic queries . . . 119
11.5 Clustering . . . 120

12 Supporting undo/redo mechanism in Emacs **121**
12.1 Installation . . . 121
12.2 Usage . . . 121
12.3 Examples of behaviour . . . 122

1 Installation and configuration

In RefactorErl you can use two database backend. One uses the ordinary MNESIA database, and the other one uses a C++ database storage model, and that makes the tool much more faster. You can choose at startup which database backend to use. Later we refer to the C++ database version as NIF, because it uses the Erlang NIF library.

1.1 Required software

- Erlang/OTP R14B03 (or newer) is required to compile and run RefactorErl.
- While not required, having a standard `make` makes the installation of the tool much easier.
- The current version of RefactorErl is available from http://plc.inf.elte.hu/erlang/dl/.

And if you want to use the NIF database, then there is an additional requirement:

- g++ 4.3.6 (or newer).

Under Unix based systems you can install gcc, which contains g++ compiler. Under Windows you can have g++ compiler, if you install MinGW. You can download it from http://www.mingw.org/. It is **recommended** that your PATH environment contains g++.

1.2 Compilation

Unpack the source package. In the following, we refer to the unpacked `refactorerl-0.9.12.01_Mnesia` or `refactorerl-0.9.12.01_NIF` directory as the root directory of the tool.

Compilation using make. If you have a standard `make` tool installed, compilation is as easy as running

```
make
```

in the root directory of the tool. Running this command produces the documentation as well.

Compilation needs access to the Erlang commands `erlc` and `erl`.

Direct compilation. If you don't have `make`, you can compile RefactorErl by entering the following command in the root directory:

```
bin/referl -build tool
```

An additional parameter is needed if your `PATH` environment does not contain the Erlang compiler.

```
bin/referl -build tool -erl PATH_TO_ERL
```

If you do not want to use the NIF database backend, then you can use the `-no_nif` flag, and build the tool like this:

```
bin/referl -build tool -no_nif
```

So in this case the C++ code will not be compiled.

1.3 User interfaces

RefactorErl has several interfaces: Erlang console (Sections 2 and 2.7), CLI (Section 3), Web interface (Section 4), (X)Emacs (Section 7), (G)Vim (Section 10) and Eclipse (Section 11).

2 Using RefactorErl from the console

This section introduces the console usage of RefactorErl, as implemented in the module `ri`, which stands for RefactorErl Interface.

2.1 General usage

Starting the tool. First, build the tool as described in Section 1. Then, starting from the base directory of RefactorErl, run

```
bin/referl
```

under Linux or

```
bin/referl.bat
```

under Windows.

Note that the tool starts with the MNESIA database backend by default, and if you want to use NIF, then you have to start the tool like this:

```
bin/referl -db nif
```

For further parameters, use the `-help` option.

-erl PATH	Path to the Erlang Executable to use
-base PATH	Path to the RefactorErl base directory
-sname NAME	Short name of the Erlang node
-name NAME	Full name of the Erlang node
-srvname NAME	Name of the Erlang server node to connect
-server	Start in server mode (no shell is started)
-client	Start in client mode (no server is started)
-g++ PATH	Path of the g++ compiler to use
-build TARGET	Build TARGET (e.g. tool, doc, clean)
-no_nif	C++ (NIF) code will not be compiled (use with `-build tool`)
-emacs	Start as an Emacs client
-yaws	Start with yaws web server
-yaws_path PATH	Path to the Yaws ebin directory
-yaws_name NAME	Set yaws server name
-yaws_port PORT	Set yaws port
-yaws_listen IP	Set yaws IP
-nitrogen	Start with Nitrogen
-browser_root	Set the file browser root directory
-images_dir	Set root directory, where generated images will be written
-db [mnesia|nif]	The database engine to use. Default is mnesia.
-help	Print this help text

Table 1: `bin/referl` options

If the environment variable `REFERL_DIR` is set, the tool will use this directory as the base Mnesia directory (similar to invoking `erl -mnesia dir '$REFERL_DIR'` from the command line). Also, the tool will save other files (e.g. log files) in this directory.

Getting help. General help can be acquired in `ri` by

```
ri:help().
```

or even shorter as

```
ri:h().
```

This function lists several topics, on which further help is available as

```
ri:h(Topic).
```

If the name of a function is known, specific help can be acquired by adding an `_h` postfix to the name. For example, help for the function `add` is available as

```
ri:add_h().
```

Compiling the tool. The tool can be recompiled by invoking

```
ri:build().
```

This function can also take a list of build parameters. This feature is mostly used through development.

Note that this function tries to compile the C++ files as well, if these were not compiled before. So if you want to prevent this, then you have to specify the `no_nif` additional parameter to the function. So in this case the recompiling looks like this:

```
ri:build(no_nif).
```

Managing files. You can add files to the RefactorErl database by calling the `add` function with either a filename as a string or a module name as an atom. Note that in the latter case, "ri" defaults to the current working directory (which you may work around by including a path in your singe-quoted atom). If you specify a directory instead of a regular filename, then it will be recursively traversed. You may just as well give a list of atoms or strings to add more files at once. All of the following example commands would add the same file:

- `cd(dir)`, `ri:add(modname).`
- `ri:add('dir/modname').`
- `ri:add(['dir/modname']).`

- `ri:add("dir/modname.erl").`
- `ri:add("/current/dir/modname.erl").`

The module displays the progression of loading.

Removing files from the database is similarly easy and also recursive, except for one difference. As the system by the time you want to remove a module must know the exact location of the said, you need not restrict yourself to dropping a module relative to the current directory, but must in exchange use real module names that do not contain path delimiters. The following will equally work:

- `ri:drop(modname).`
- `ri:drop([modname]).`
- `ri:drop("dir/modname.erl").`
- `ri:drop("/current/dir/modname.erl").`

Modules can be loaded as applications, but the base of your library has to be set before:

```
ri:addenv(appbase, "path/to/my/applib").
```

You can check the already given application base directories:

```
ri:envs().
```

Let's see an example:

```
(refactorerl@localhost)18> ri:envs().
output = original
appbase = "/usr/local/lib/erlang/lib"

(refactorerl@localhost)19> ri:add(usr, synatx_tools).
Application synatx_tools not found under usr
not_found

(refactorerl@localhost)20> ri:add(usr, syntax_tools).
Adding: /usr/local/lib/erlang/lib/syntax_tools-1.6.7.1/src
...
```

You can also set include directories to your include files using:

```
ri:addenv(include, "path/to/my/include").
```

It is possible to delete the defined environment variables:

```
ri:delenv(include).
```

Loaded files can be saved using

```
ri:save(Filename).
```

For convenience, both the filenames and the directory names can be given as atoms as well as strings.

The list of loaded files can be obtained by calling

```
ri:ls().
```

This call also displays the status of the loaded files (`error` or `no_error`).

If the module `m` is loaded,

```
ri:ls(m).
```

will give information about the functions, records and macros in the file.

The contents of a file can be listed by

```
ri:cat(m).
```

Loading BEAM files. Usually, Erlang source files (having the extension `.erl`) are loaded into RefactorErl. In addition, RefactorErl is also capable of loading compiled `.beam` files.

```
ri:add("compiled.beam").
```

Note that this feature is applicable only to those `.beam` files that were compiled with the `debug_info` option. Also note that the resulting file will be pretty printed by RefactorErl.

Using transformations. Transformations can be called using their abbreviated names, and the list of required parameters. These commands are listed in Section 2.6.

There is another way to call a transormation. This way let the user to choose: user want to specify all of arguments or not. There are lots of cases when the user can not specify all of the required arguments. In this case the tool can help the user with interactions. The tool ask questions and the user has to answer it to specify the missing arguments. The interactions also work if there is some problem with the given arguments. For implemented interaction details, see the transformation descriptions.

The following example shows how an interaction looks like:

The example uses the Rename function transformation. This transformation needs a function to be specified. In this example we do not give any argument, so the tool need to ask for. First it asks for a module to know where it should search for functions. Then it asks the user to select a function. If user has to answer with 'yes' or 'no'. Then if the user specified a function, the tool asks for a new name. If everything works fine (no illegal parameter given) the transformation will rename the specified function to the given new name.

```
(refactorerl@localhost)3> ri:transform(rename_fun, [{}]).
Please answer the following question (blank to abort).
 Please type in the module name:: alma
Please answer the following questions (blank to abort).
 Please specify a function:
 Type in the answer for this question as a raw Erlang term:
 [{text,"f7/0"},{format,radio},{default,false}]
 -> yes
 Type in the answer for this question as a raw Erlang term:
 [{text,"f9/2"},{format,radio},{default,false}]
 -> no
 Type in the answer for this question as a raw Erlang term:
 [{text,"f6/2"},{format,radio},{default,false}]
 -> no
 Type in the answer for this question as a raw Erlang term:
 [{text,"f6/1"},{format,radio},{default,false}]
 -> no
 Type in the answer for this question as a raw Erlang term:
 [{text,"f5/2"},{format,radio},{default,false}]
 -> no
Please answer the following question (blank to abort).
 Please type in a function name:: newfun
modified /home/csoki/Work/referl/alma.erl
result
(refactorerl@localhost)4>
```

Figure 1: Interaction example

Manipulating the graph. You can reset RefactorErl by invoking

```
ri:reset().
```

This will remove all loaded files. This function should be called if the graph gets corrupted.

You can add a checkpoint using

```
ri:backup().
```

If the transformations you have performed are not satisfactory, you can go back to the previous checkpoint using

```
ri:undo().
```

If you use NIF, then it is little different:
You can create backups with `ri:backup/0` or `ri:backup/1` and you can load these backups with `ri:restore/1`.

When execute a transformation a backup will be created, which name differs from the ordinary backups, and the ri:undo/0 function will restore that.

Inspecting the graph. You can draw the semantic representation graph of RefactorErl by calling

```
ri:graph().
```

This function produces a `.dot` file (by default, `graph.dot`, although this can be customised), which can be transformed to several visual formats using Graphviz. One of these transformations is available from RefactorErl for convenience:

```
ri:svg().
```

The representation can be filtered:

```
ri:svg(OutFile, Filter).
```

where `Filter` is one of the following:

- `all`: default, all edges except environmental ones are shown
- `syn`: only syntactic edges are shown
- `sem`: only semantic edges are shown
- `lex`: only lexical edges are shown
- `all_env`: all edges are shown, no filtering
- `ctx`: context related edges are shown
- `not_lex`: all edges except lexical ones are shown
- `dataflow`: dataflow related edges are shown
- a list of the above: shows the union of the designated subgraphs

Using queries. Queries 6 can be invoked by either

```
ri:q(Query).
```

or

```
ri:q(Module, Regexp, Query).
```

The former is applicable when a query starts generally, such as

```
ri:q("mods.funs.name").
```

For those queries that begin from a selected position (these queries start with "@" when used from Emacs), the second variant is required. As the console cannot mark a position, the first and the second component indicate the starting point for the query. The following example shows how to get all the variables used in the body of the function `f/2` from the module `m`.

```
ri:q(m, "f\\\\(X, Y\\\\)", "@fun.var").
```

Additional options can be given to a semantic query in a proplist as the last argument. The following arguments are currently recognized:

- `{out,FileName}` - write the textual output of a query to a file
- `linenum` - prepends match sites with file and line number information similar to `grep -n`.

The following example outputs all defined functions with line numbers to a file named `result.txt`.

```
ri:q("mods.funs",[linenum,{out,"result.txt"}]).
```

2.2 Module and function dependencies.

We say that a module A is dependent on another module B ($A \rightarrow B$) if there is at least one function call from A to B. A cyclic dependency appears, when B is also dependent directly ($B \rightarrow A$) or indirectly (e.g. $B \rightarrow C \rightarrow A$) from A. Note that it is possible to have a cyclic dependency among the modules while having no cyclic dependencies among the functions. For example, a function call from `A:foo` to `B:foo`, and from `B:foo2` to `A:foo2` implies a cyclic dependency on the module level.

On the other hand, if one wants to have a deeper analysis and pays more attention to the concerning functions, a function level query should be done. In our previous example, no function level cycle appears, unless `A:foo` calls `B:foo`, and `B:foo` calls `A:foo`.

The following examinations can be done considering dependencies:

- Checking whether there are cycles, if so, listing them out
- Printing out the cycles, meaning the modules/functions will not be represented by their proper graph node, but with their names (for instance, instead of `{'$gn', module,3}`, module `test` will be printed, if the graph node stands for that exact module).
- Checking for cycle from one or more nodes as starting points
- Drawing the dependency graph
- Drawing the dependency graph from a starting node
- Drawing the cyclic part of the dependency graph if one exists (you can also give a cyclic node as a starting node)

Dependency analysis can be done in two ways: directly with their proper modules or by the means of `ri` interface. To call the desired query, the user should give a proplist, stating the different requirements. The two interface functions are:

- `ri:draw_dep/1` - for drawing
- `ri:print_dep/1` - for listing, printing out to the standard output

The options and the keys for the functions are:

- `{level, Level}| Level = mod | func`
  Stating the level of the query, module or function level.
- `{type, Type} Type = all | cycles`
  The investigation should be done on the whole graph/table, or just on the cycle part (if it exists). When listing out the cycles, "all" gives back the result in their graph node form, while "cycles" returns with their proper names (see examples above).

- `{gnode, Node | NodeList::lists(Node)},`
  `Node = node() | Name,`
  `Name = Module::atom() | Function::string()`
  Specify a node or nodes (given in a list) as a starting point for the analysis. In module level, the name should be given as an atom, while in the case of function it has to be a string ("Module:Function/Arity"). In both cases the node/nodes can be given as graph nodes as well.

- `{dot, Dot::string()}`
  The user can stipulate his own name and absolute path or the generated .dot file. Unless it is a new absolute path, the .dot file will be placed into the `./dep_files` directory. Only available in the case of `draw_dep`, new feature.

Examples:

```
ri:draw_dep([{level, mod}, {gnode, erlang}]).
ri:draw_dep([{level, mod}, {gnode, erlang},
    {dot, "/home/working/dot/test.dot" }]).
ri:draw_dep([{level, func}, {gnode, "lists:hd/1"}]).
ri:draw_dep([{level, mod},{gnode, {'$gn', module, 4}}]).
ri:draw_dep([{type, cycles}, {level, func},
     {gnode, {'$gn', func, 36}}]).
```

Representation on module and function levels Let's take an example to explain the meaning of the representation of dependency graphs. A `ri:draw_dep([{level, func}, {type, all}])` call was made, which generated a Graphviz dot file. Also `ri:anal_dyn()` was run, which is a dynamic analyser. Due to its work the call graph changes a bit, new types of nodes and edges are introduced. The result can be seen in Figure 2:

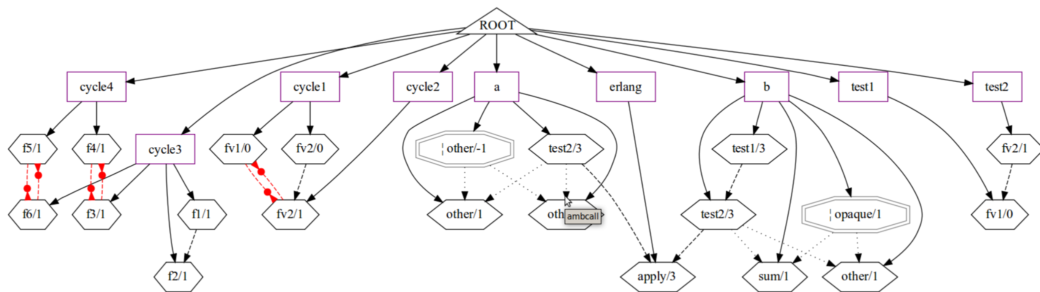


Figure 2: An example of the representation of function level dependencies.

Explanation of Figure 2:

- ROOT triangle - no actual purpose, functioning as a starting point, only appears in the representation

- Rectangle/box nodes (eg.: cycle1, a, test2) - representing modules (colour: deep purple)
- Hexagon nodes (eg.: f1/1, apply/3, test2/2) - representing functions (colour: black)
- Double octagon nodes - representing opaque nodes (colour: black, label colour: gray)
- Solid, continuous edge, normal arrowhead - normal edge indicating the modules from the root, and the modules and their definitions of functions (colour: black)
- Dashed edge, normal arrowhead - indicates that a function calls another function (funcall) (colour:black)
- Dashed edge, special arrowhead - indicated a function call, but also that it is a cyclic edge (colour: red)
- Double octagon nodes - representing opaque, -1 arity nodes (colour: black, label colour: gray)
- Dotted edge, normal arrowhead - indicates an ambcall, dyncall, may_be edge (colour:black)

Every node and edge have tooltips. In the case of nodes it shows the adequate graph node, while considering edges it depicts the type of function call (`funcall, may_be, ambcall, dyncall`). Static calls are labelled by `funcall`.

Warning! We note here that tooltips may not be shown under certain browsers (for example Mozilla Firefox 7.0.1). If this happens, please try another browser.

Functionblock examination. Function blocks are groups/clusters of modules mainly implementing some functionality inside an application. We also seek dependencies between them, which is conceptually similar to dependencies between modules: a function block FB1 is dependent on a function block FB2 if a module from FB1 is dependent on one from FB2.

A function block can be defined with its containing modules: given as a list, or defining them with the containing directory, or defining them by a regular expression.

The function `ri:fb_relations/1` is used for the function block dependency analysis. The argument, which is again a proplist of this function determines the exact examination type.
The Options are the following:

- `{command, Command}`
```
Command = get_rel | is_rel | check_cycle
        | draw | draw_cycle
```

- `get_rel` - Displays the relationship between the given function block list. The result is a tuplelist where a tuple represents a relation.
- `is_rel` - Decides whether there is a connection between the two given function blocks.
- `check_cycle` - Checks for cycles in the dependencies between the given function block list. Unless list is given, checks among every function block list.
- `draw` - Prints out the entire graph or creates a subgraph drawing from the given function block list. Output file is `fb_relations.dot`.
- `draw_cycle` - Prints out a subgraph which contains the cycle(s). Unless cycles exist, prints out the entire graph. Output file is `fb_rel_cycles.dot`.

- `{fb_list, List}`
```
List = [string()] |
          [{Basename::string(), [Function block::atom()]}]
```
  Chosen function block lists for further examinations. If no list given, then it takes every function block list, which means that every absolute path defines a function block.

- `{other, Other}`
```
 Other = bool()
```
  The `Other` parameter stands whether the category "Other" would be taken into consideration or not (`true/false`). The default value is `true`.

Examples:

```
ri:fb_relations([{command, check_cycle}]).
ri:fb_relations([{command, draw_cycle}]).
ri:fb_relations([{command, is_rel},
                 {fb_list,["path_to_dir/subdir",
                  "path_to_dir/subdir/subsubdir"]}]).
ri:fb_relations([{command, is_rel},
                 {fb_list,{"path_to_dir", [1, 2]}}]).
```

Optional *Other* category Let's think about function blocks in the first sense as mentioned in the beginning, so every directory with its absolute path gives a different function block. The Other category is a special collector name for those modules which cannot be divided into any function block. Practically this covers those modules which do not have directories (for example, usually erlang). This category can be taken into consideration as a false function block, so a new option was introduced for eliminating this category. With this the dependencies with *Other* category are skipped, the tool takes into consideration only the real connections.

Representation of functionblocks Representing the function block relationships is very similar to the previously introduced module and function visualisation. It could be useful to make the standard output messages also available,

because function blocks are represented with numbers (the long and complex directory structures can make the generated image more tangled). Moreover, as a tooltip, the proper function block name is provided.

```
(refactorerl@localhost)10> ri:fb_relations([{command, draw}]).
Earlier results deleted (except .dot files).
Building dependency table...
Creating
 "/home/RefactorErl/tools/new/tool/dep_files/fb_relations.dot" file...

-----------------------------------------------------------
Function block 1 is "/home/RefactorErl/test/cyclic/cycles"
Function block 2 is "/home/RefactorErl/test/regexp/common/serv1/ebin"
Function block 3 is "Other"
Function block 4 is "/home/RefactorErl/test/error"
Function block 5 is "/home/RefactorErl/test/cyclic/no_cycle"
Function block 6 is "/home/RefactorErl/test/opaque"
Function block 7 is
"/home/RefactorErl/test/regexp/common/serv2/lib/ebin"
-------------------------------------------------------------
```

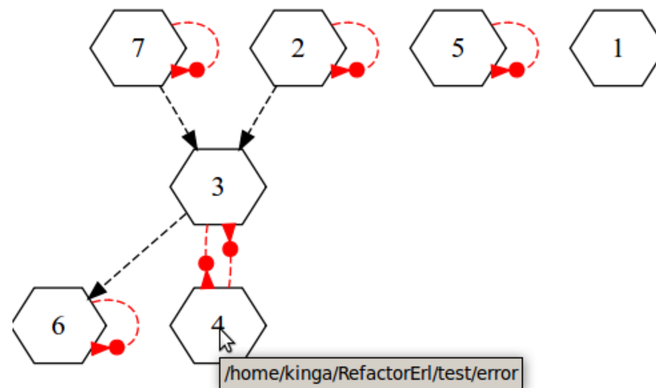


Figure 3: An example of the representation of fb level dependencies.

Explanation Figure 3:

- Hexagon nodes (eg.: cycle1, erlang, test2) - representing function blocks as number (colour: black)
- Dotted edge, normal arrowhead - indicates that a fb calls another fb (colour:black)
- Dotted edge, special arrowhead - cyclic edge (colour: red)

Function blocks can be filtered by the means of regular expressions using the function `ri:fb_regexp/1` and its parameters are the following:

- `{type, Type}`
  `Type = list | get_rel | cycle | draw`
  - `list` Prints out every function block which matches the basic regular expression.
  - `get_rel` Decides whether there is a connection between the two given function blocks.
  - `cycle` Checks for cycles in the dependencies between the given function block list.
  - `draw` Prints out the entire graph or creates a subgraph drawing from the given function block list. Output file is `fb_relations.dot` or can be user defined with the `dot` key.
- `{regexp, Value}`
  `Value = File::string() | [RegExp::string()]`

  If this option (tuple) is not given, the program works with a basic regular expression. The basic rule: *<function block>/common/<service>/ebin* or *<function block>/common/<service>/lib/ebin.* and regular expression saved for this:
  `^(/)[0-9a-zA-Z_./]+/common/[0-9a-zA-Z_.]+/(lib/)?(ebin)$`.
  - `Value` - If the regular expression is given in a file then every single regexp has to be defined in a separate line and must follow the Perl syntax and semantics as the *"http://www.erlang.org/doc/man/re.html"* erlang module resembles so. However, the user can give the regular expressions in a list as well. If there is an error with a regular expression in the file or in the list, it prints out the regexp, the error specification, and the position. The most usual regexp is ".*" (the Perl syntax does not allow simply "*", because this symbols means possible unlimited repetition of characters declared before it, and there are no characters declared before it)

Examples:

- `ri:fb_regexp([{type, draw}, {dot, test.dot}]).`
- `ri:fb_regexp([{type, list}, {regexp, "regexp"}]).`
- `ri:fb_regexp([{type, list}, {regexp, "^/home/[a-z./]+}]).`

User defined function blocks One can make his own function block in the following three ways:

- Giving the exact modules (with their name) which should be in one function block.

- Regular expressions covering the structure of the directories.
- By regular expressions covering the structure of the directories and the structure of the name of the files.

Example:

```
refusr_fb_regexp:re([{type, list}, {fb,
          [[cycle1, cycle2],
          "/home/user/[a-zA-z]*",
          "/home/user/[a-zA-z./]*/.*_ui.erl"]}]).
```

2.3 Interface layers

Interface layers and additional relations can define and can check wether in this architecture there are function calls, that insult the layer hierarchy or not.

Defining interface layers We can define the hierarchy of the interface layers with a list. The first element of the list is at the bottom of the layer hierarchy and the last element of the list is at the top of it. This means, that by default every layer can call functions from its own layer and the layer immediately below. The list contains tuples. Every tuple defines an interface layer: the first element of the tuple is a label with the name of the label; the second element is a list, that contains the modules, which from the layer is built. There are four ways to specify this list with:

- Name of the modules: `[{il1,[one1,one2]},{il2,[two1]}]`.
- Module nodes: `[{il1,[{'gn',module,2},{'gn',module,4}]}, {il2,[{'gn',module,6}]}]`.
- List of regexps: `[{il1,["^(/home/user)/[a-zA-Z0-9_/]+(/layer1)$"]}, {il2,["(/layer2)$","^(/home/user/layer2/src)$"]}]`.
- File, that contain regexps: `[{il1,["layer1"]}, {il2,["layer2"]}]`. In this example `layer1` and `layer2` are files, which contain regexps. When working with files in the list have to be only one file name.

If regular expressions are used to define interface layers, we can mix the two ways of specification: `[{il1,["^(/home/user)/[a-zA-Z0-9_/]+(/layer1)$"]}, {il2,["layer2"]}]`.

Define additional relations Between two layers we can define a relation, which allows function calls from the first to the second layer. The definition of these additional relations available with a list, that contains this pairs in tuples. We can refer to interface layers with their names. Suppose, that we want to allow function calls from `il1` to `il2` as well as from `il2` to `il4` layer: `[{il1,il2},{il2,il4}]`.

Examples:

- `ri:check_layered_arch([{il1,["^(/home/user/layers/layer1)$"]}, {il2,["^(/home/user/layers/layer2)$"]}, {il3,["regexp3"]}],[]).`
- `ri:show_layered_arch([{il1,["^(/home/user/layers/layer1)$"]}, {il2,["^(/home/user/layers/layer2)$"]}, {il3,["regexp3"]}],[]).`
- `ri:show_layered_arch([{il1,["^(/home/user/layers/layer1)$"]}, {il2,["^(/home/user/layers/layer2)$"]}, {il3,["regexp3"]}], [{il1,il3}]).`

2.4 Server management command list

Here's the list of supported server management commands:

- `add(FDML)` - add a module, file, directory or a list of these to the database
- `drop(FDML)` - drop a module from the database
- `ls()` - list files that are in the database
- `backup()` - update the backup (checkpoint)
- `undo()` - undo the transformation (rollback, only one step)
- `clean()` - clean backups (delete all checkpoints)
- `reset()` - reset the database to an empty state, but valid schema
- `graph(Target)` - assume no options and call one of the next two
- `graph(Atom,Options)` - assume ".dot" extension and call the one below
- `graph(File,Options)` - draw the graph with the given options
- `svg()` - draw the graph to `graph.svg` and call Graphviz
- `svg(File)`
- `svg(File, Options)`

The additional/modified commands, that you can use, if you use the NIF database engine:

- `backup()` - creates a backup
- `backup(CommitLog)` - creates a backup as `ri:backup/0`, but here the user can attach a commit log to the backup file
- `ls_backups()` - returns a lists of backups, that has been created before with `ri:backup/0` or `ri:backup/1`

- `backup_info(Backup)` - returns information about the given backup
- `restore(Backup)` - restores the given backup
- `create_graph(Name)` - creates a graph with the given name
- `rename_graph(OldName, NewName)` - renames a graph that has the given OldName, with the given NewName
- `ls_graphs()` - returns a list of the created graphs
- `actual_graph()` - returns the actual graph's name
- `load_graph(Name)` - loads the given graph
- `delete_graph(Name)` - removes the given graph
- `delete_all_graphs()` - removes all graphs

2.5 Running dynamic call analysis

Dynamic function calls are either dynamic MFA calls (those MFA calls whose callee is given by means of non-literal expressions) or apply calls (calls to the *erlang:apply/3* built-in function). These constructs are rather difficult to be statically analysed, since their callee is determined just at execution-time. In many cases, we are able to find potential callees, however, this kind of analysis would be inefficient to be performed incrementally. Therefore dynamic call analysis has to be requested by the user.

In order to analyse dynamic call constructs, load all the libraries and files you would like to work with. Then in the RefactorErl shell, type:

```
ri:anal_dyn().
```

The dynamic call analysis starts and informs you about the entire process and its progress. Firstly, all function calls are gathered from the database. Secondly, dynamic ones are filtered and passed to the next step, where a parallel algorithm tries to identify the callee of each call. Finally, the results are merged and stored into the database.

For instance, executed on *Erlang stdlib 1.17.4*:

```
24012 function calls found in the database.
Looking for dynamic calls...
541 function calls seem to be dynamic.
Looking up dynamic calls... (  533/  541)
Identification of 8 dynamic calls timed out.

533 dynamic calls identified in the database.
Analysing dynamic calls... (  533/  533)

Analysis completed.
```

As you can see in this example, there may be function calls that are not identifiable in a reasonable time (currently the timeout is set to 5 seconds, and there were 8 calls whose analysis has been ignored). This is due to the complexity of some data-flow path analysis. Users may increase the timeout on their own risk in order to make the analysis process able to identify all the callees.

Please note that most database modifications (e.g. loading or refactoring a module) may invalidate the results of this dynamic call analysis. Currently this invalidation step is not automatically done, so the user has to invoke it. To clean the dynamic calls (and every the corresponding entry) from the database, simply call:

```
ri:clean_dyn().
```

2.6 Transformation command list

Here's the list of supported transformations:

- `elimvar(In, Pos)` - eliminate variable
- `extfun (In, Range)` - extract the selected expressions into a function
- `expfun (In, Pos)` - expand implicit funexpression to function call
- `genfun (In, Range, NewVarName)` - generalize function with a new argument
- `inlfun (In, Pos)` - inline function at application
- `inlmac (In, Pos)` - inline macro at use
- `intrec (In, Range, NewRec, [RecFldName1, RecFldName2, ...]))` - introduce record instead of tuple
- `merge  (In, Range, NewVarName)` - merge common subexpressions into a new variable
- `movfun (From, ToMod, [{FunName,Arity}|_])` - move function
- `movrec (From, To, [RecName|_])` - move record
- `movmac (From, To, [MacName|_])` - move macro
- `reorder(In, {FunName,Arity}, [ArgIdx|_])` - reorder function arguments
- `renfld (In, RecName, RecFldName, NewRecFldName)` - rename record field
- `renfun (In, {FunName,Arity}, NewFunName)` - rename function

- `renhrl (FromFile, ToFile)` - rename header file
- `renrec (In, RecName, NewRecName)` - rename record
- `renmac (In, MacName, NewMacName)` - rename macro
- `renmod (From, ToMod)` - rename module
- `renvar (In, Range, NewVarName)` - rename variable
- `tupfun (In, Range)` - change function arguments into tuple
- `upregex()` - upgrade regexp from "regexp" module to "re" module usage
- `genspec(File, {FuncName, Arity})` - generate function spec

In the above list, the parameters can have the following types:

- `In`, `From`, `To`: filename as string or module name as atom
- `FromFile`, `ToFile`, `MacName`: strings
- `ToMod`, `RecName`, `RecFldName`, `FunName`: atoms
- `Arity`, `ArgIdx`: integers
- `Pos`: integer, a character position in the file
- `Range`: a pair of positions

2.7 Scriptable RefactorErl interface

ris is similar to ri, with the following basic ideas:

- results are always returned via the function return value
- no mandantory standard output
- arguments very regular - semantic queries for almost everything
- you can also input a semantic query via atoms instead of strings to ease escaping.
- operations are composable (i.e., continue one where another has left off) - queries and refactorings can go back and forth
- you can perform a series of batch refactorings in a single step by selecting multiple entities at once

Function returns

An encapsulated query_results() type. ris:unpack/1 ris:desugar/1

If a query chain yields an empty list at some point, the remaining part will also yield an empty list.

Exceptions are thrown to indicate abnormal operation (internal error, illegal operations, connection errors, etc.) or denied transformation (unmet preconditions). The exact format is {Severity, {Code::{_,_,_}, Message::string()}, where Severity is 'abort' for denied operations, and 'error' for fatal errors.

Usage examples

2.7.0.1 Adding files

```
Added = ris:add_byname("mymods.erl").

ReAdded = ris:add('mods[name~"mymodu.*"]').
```

2.7.0.2 Dropping files

```
ris:drop('mods[name ~ "mymo.*"]').
```

2.7.0.3 Refactorings

```
RenamedVars = ris:rename("mods.fun.var[name=="Colour"]", "Color").

MovedFuns = ris:move("mods[name ~ "^referl_.*"].fun[exported==false]",
                     fun(F)->
                         ris:qstr([F,".mod.name"])++"_util"
                     end).

Moved = ris:move("mods[name==mod1].fun[name==fun1 and arity==3]",
                 mod2).

NewFun = ris:extract("mods[name==module1]
                        .fun[name==f and arity==0]
                           .expr[last==false]",
                     mynewfun).

NewVar = ris:intvar("mods[name==module1]
                        .fun[name==f and arity==0]
                            .expr[index==1].esub[class/=pattern]",
                    "Varname").

NewRec = ris:intrec("mods[name==module1]
                        .fun[name==f and arity==0]
                            .expr[index==1].esub[class/=pattern]",
```

```
                    {newrec,[field1,field2]}).

NewFun = ris:reorder("mod[name==mod1].fun[name==fun1,arity==3]",
                     [3,2,1]).

NewFun = ris:tupfun("mod[name==mod1].fun[name==fun1,arity==3]",
                    {1,2}).


ris:generalize("mod[name==china].fun[name==sum].var[name=="A"]").


ris:eliminate("mod[name==china].fun[name==sum].var[name=="A"]").

ris:inline("mod[name==china].fun[name==sum]
               .expr[type/=pattern and index==1]").
```

Operators

The result of the queries can be combined with the following set operators:

- Intersection – The following example takes the intersection of the files included by the two modules.

```
ris:q({"mods[name==mod1].includes", intersect,
       "mods[name==mod2].includes"}).
```

- Union - similar to the above (use key 'union')

- Substraction - similar to the above (use key 'minus')

- Range – Ranges of expressions can be selected which denotes a list of continuous expressions between two syntactical siblings. An empty semantic query denotes the beginning or end of a block when used as the initial or final limit respectively. In this example, the expression range starts from the (first) match expression that contains "Var1" in the pattern side up to the end of the syntactical block.

```
NewFun = ris:extract({"mods[name=mod1]
                           .fun[name==f and arity==0]
                              .expr[type==match_expr]
                                   .esub[class==pattern and
                                         type==variable and
                                         .var[name=="Var1"]]",
                      range, ""}, mynewfun).
```

- Sequence – Queries can be sequenced to continue a query from where another has left off. This example first adds the module from file 'mymodule.erl'. The add call returns the entities loaded. A semantic query aggregate of a list works by executing the first query (or in this case, specifying a starting entity), and then running the next query in the chain (in this case getting the name of files included by the add call).

```
ris:q([ris:add_byname("mymodule.erl"),".includes.name"]).
```

- another sequence example – The following example shows an example for composition. It first renames all functions whose name is 'duck' to 'quack'. It then appends the suffix '_quacker' to the name of all functions which call these quacks. The new name is automatically converted to an atom.

```
New = ris:rename('mods.fun[name==duck]',quack),
Callers = ris:q([New,".callers"]),
[ris:rename(Fun,atom_to_list(Name)++"_quacker") ||
 Fun <- Callers,
 Name <- ris:q([Fun,".name"])].
```

```
New = ris:rename('mods.fun[name==duck]',quack),
Callers = ris:q([New,".callers"]),
ris:rename(Callers,
           fun(Fun)->
               ris:qstr([Fun,".name"]) ++ "_quacker"
           end).
```

2.7.0.4 Textual display Use ris:show/1 to stringify entities. ris:show/2 does the same while accepting additional options already known for ri:q/3. Use the respective ris:print/1 and ris:print/2 functions for screen and file output.

```
ris:print(ris:q("mods.fun")).
```

The following gives the same result set, but written to the given file and annotated with line numbers. (Note that you could also manually write the output of ris:show/1 to a file.)

```
ris:print(ris:q("mods.fun"),
          [{out,"funs.txt"}, linenum]).
```

3 Using RefactorErl in the command line

This section introduces the Command Line Interface (CLI) of RefactorErl.

We already described the Erlang shell interface of the tool, which proves to be pretty useful in most of the cases. However, we realised that sometimes one may need to access the tool from outside the Erlang shell. The RefactorErl CLI provides a lightweight interface that can execute Erlang commands inside RefactorErl. More precisely, it invokes Erlang functions that are specified by command line arguments.

We note that the current version is a prototype, only discovering opportunities and usefulness of this kind of interface. Notice that the CLI is applicable directly by hand as well as indirectly by code editors. Latter makes RefactorErl callable from editors that do not support Erlang shell.

Installation

If you have successfully downloaded and compiled RefactorErl, you have nothing else to install, since the CLI is written as an EScript.

Before the first usage, you should redefine two functions defined in the RefactorErl CLI:

- referlpath() defines the directory path to the tool ("." by default)
- referlnode() defines the node name of the tool ("refactorerl@localhost" by default)

Also, you should make the 'RefactorErl' EScript executable. You may use a command like

```
chmod +x RefactorErl
```

Usage

Using the CLI is pretty straightforward. You can simply pass the following data to the script as command line arguments:

- The name of the module the invoked function is located in
- The name of the invoked function
- The function arguments

For example,

```
/path/to/refactorerl/bin/RefactorErl mod fun arg1 arg2 "arg3"
```

will invoke `mod:fun(arg1, arg2, arg3)` inside RefactorErl.
Note that Erlang terms passed to the CLI should be enclosed by double quote marks.

Handling the RefactorErl server First of all, if the RefactorErl server is not running, you should start it up to be able to communicate with. Type

```
RefactorErl start
```

to start up. If the server is already running, you get a message of it.
Also, you can stop RefactorErl. If the server was not running, you will be informed.

```
RefactorErl stop
```

Examples

Resetting database:

```
RefactorErl reset
RefactorErl ri reset
```

Adding files:

```
RefactorErl ri add "/path/to/module.erl"
```

Listing database contents:

```
RefactorErl ri ls
```

Renaming function `mod:f/1` to `g`:

```
RefactorErl ri renfun mod "{f, 1}" g
```

4 Web interface

The web based interface has many benefits and implements additional functionality. The main features of the web interface are the ability to run semantic queries- both of global queries and queries starting with @ are supported, the query construct assistant, the query storage and the visualisation of the query result, possibility to see running queries and abort them if necessary, database operations, ability to mark files with error forms, dependency examinations.

JavaScript must be enabled in the browser to be able to use the interface!

4.1 Installation

To be able to use the web based interface we need to have an already working Yaws webserver. Required version is 1.89. Help for the installation: http://yaws.hyber.org/yaws.pdf (Chapter 2).

4.2 Start up

We can start the interface either with `referl` script with `-nitrogen` switch, or from RefactorErl shell. In both cases we get a default configuration with server name `localhost`, port `8001`, and IP `127.0.0.1`.

4.2.1 Starting up with `referl` script

Parameters to be configured:

- `-yaws_path YPATH`: The absolute location of your Yaws ebin directory.
- `-yaws_listen YLISTEN`: Valid IP address, which Yaws will listen to.
- `-yaws_name YNAME`: Valid domain name, which Yaws will be bound to.
- `-yaws_port YPORT`: Valid port number, which Yaws will be bound to.
- `-browser_root BROOT`: The web based interface allows database operations. The root directory for those operations can be set, by giving the path of root directory in this parameter.
- `-images_dir IDIR`: Path of the directory where the generated images, which are the visualisation of the results of dependency examinations, will be written.

Usage of switches is optional, except the `-nitrogen` switch.

Example:

```
bin/referl -nitrogen
           -yaws_path /Users/V/yaws-1.89/ebin
           -yaws_listen 127.0.0.1
```

```
-yaws_port 8000
-yaws_name localhost
-browser_root /Users/V/erlang
-images_dir /Users/V/graph_images
```

4.2.2 Starting up from RefactorErl shell

We have 2 functions for starting the interface : `ri:start_nitrogen/0` and `ri:start_nitrogen/1`. If the 0 arity function is used, the interface starts up with default configuration. If the 1 arity function is use the start up can be configured using a prop-list. Available properties are the same as described in the previous section. Usage of switches is optional.
Example:

```
ri:start_nitrogen([{yaws_path,"/Users/V/yaws-1.89/ebin"},
      {yaws_listen,"127.0.0.1"}, {yaws_name, localhost},
      {yaws_port,"8000"}, {browser_root,"/Users/V/erlang"},
      {images_dir,"/Users/V/graph_images"}]).
```

4.3 Shutting down

It is important to log out, before shutting down the interface, because the log out process will delete the dynamic generated images, which belong to the user. If the interface have been started up from RefactorErl shell, then `ri:stop_nitrogen().` can be called to shut down the interface.

4.4 Logging in

To log in one must open a browser (recommended: Mozilla Firefox) and enter the URL defined by the configuration (the default is `http://localhost:8001/`) after the web server had been started. Usage of services are allowed only to authorized people. First, you have to log in with a username (passwords are not supported yet). The browser will be redirected to `queries` page.

4.5 Semantic queries

This service is available under "Queries" menu.

Constructing semantic queries The query construct assistant is located at the top-left corner of the page. While typing into the text-box, the interface offers possible continuations for the actual, uncompleted sub-term. The offered option can be chosen from a drop-down list. This auto-complete mechanism helps new RefactorErl users to use the language, and also all developers to speed up query construction and to avoid constructing wrong queries.

Pressing the "Run" button evaluates the query and displays the result (Figure 5).

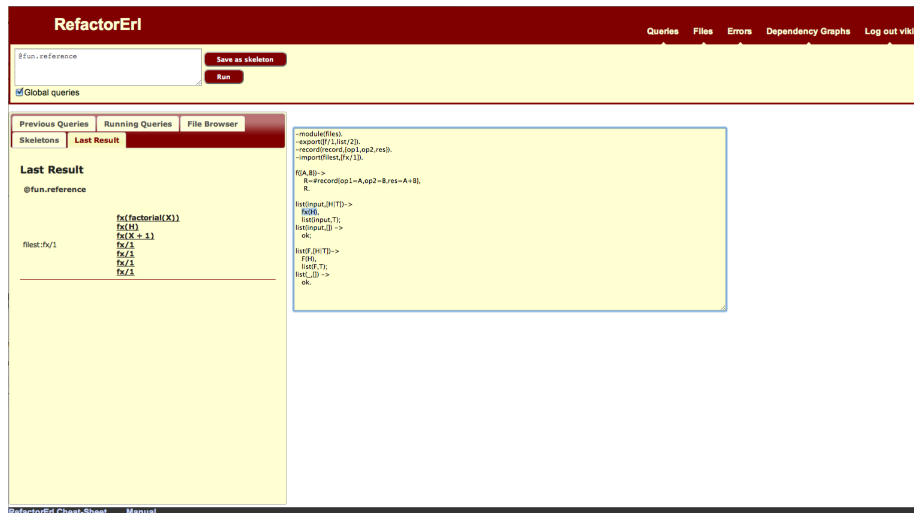


Figure 4: Web interface

Position based queries A file browser is placed at the left side of 'Queries' page at 'File browser' tab. One of the previously loaded files can be opened in *text view* from the file browser panel, by clicking on the magnifying glass icon after selecting a file. By selecting a region or pointing at a position in the text- box, where the contents of the file are loaded in, one can specify the exact position as starting point. Also the result of the previous query can still be reused as a starting point.

Node based queries Previously loaded files can also be opened in *links view* from the file browser panel, by clicking the link icon after selecting a file. In this view queries can be started from a specific node. You can select a node by clicking on it (only text in bold can be selected). The selected node will be marked with red color.

After selecting a node you can either:

- Run pre-defined queries for that node from the top-right corner of the screen.
- Run previously made queries for that node type by selecting one from the *Choose previous query* dropdown box.
- Run a query from the query construct assistant text-box, which will start from that node.

The first result of the query will be shown (if there was any). Results can be browsed with the appearing Prev and Next buttons. The total number of results as well as the currently shown result's number are displayed under those buttons.

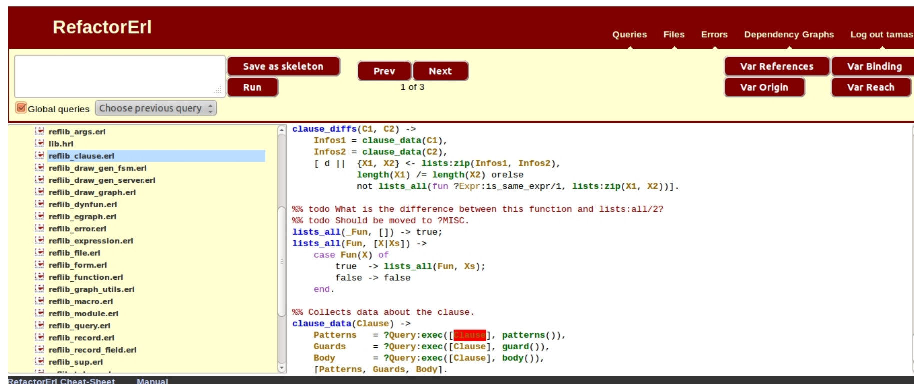


Figure 5: The links view

NOTE: This feature is still under development, and does not support every query. Please use the Position based queries when you need more explicit results.

Global queries If the *Global queries* checkbox is checked, all queries run from the query construct assistant text-box will work as in *text view* (and not starting from the selected node). This setting is only important when browsing a file in *links view*, and a node is currently selected.

Alias-able queries We made queries alias-able, cause of named queries can be identified easier, and are more readable for humans, too.

After a query has been successfully run first, it appears at the left side of 'Queries' page at 'Previous Queries' tab.

By clicking the corresponding 'E' button, a text-field appears, which contains the query string. By replacing the query string with the name, what is wanted to assign to the query, and clicking the 'Save' button the query will be named. After a name has been assigned to the query, the name appears at 'Previous Queries' tab instead of the query string. If the name of the query should be changed, the same mechanism should be done as described above.

The query string can be shown as an information element by clicking the '?' button.

Previous Queries For your convenience, the interface stores previously executed queries and their results, which are listed at the left side of the page at "Previous Queries" tab. One can choose to list only the queries which belong to the current user, or all the queries stored in the system.

When one of the queries located in the list is clicked on, the interface first examines if the database has changed since that given query was last run. If there are no changes, the stored result is displayed. In the other case, the query is re-executed, the stored result is updated, and the new result is displayed.

Users have possibility to delete their own queries from the list by clicking on the corresponding "X" icon, or to run queries which belong to other users by clicking on the query string.

By clicking on the corresponding "?" icon, the query string and previously added comments are always shown. If query is one of the queries with @ the starting file and position are also shown.

By clicking on the corresponding "E" icon, one can assign or can reassign a name to the query. This mechanism helps users to identify queries much easier.

Users can add comments, either on their own queries, either on the queries of the others, too. By clicking on the corresponding "C" icon, one can modify the previously added comment. By default, the comment field is empty.

Running Queries The list of currently running queries is displayed at the left side of the page at "Running Queries" tab. The list is updated in every second. A running query,which has been started by the current user, can be aborted by pressing the corresponding `X` icon. The current user, who started the query, is notified about the success of the abort.

Displaying the result of queries Results are displayed at the left side of the page at "Last Result" tab in a table. The source of the entries of the result's can be visualised by clicking on the given entry in the detail. If the source is loaded to the database, it will be shown in a text-box located to the right of the table. The part of the source code responsible for the entry will be highlighted in the text-box.

Skeleton of queries Large amount of queries are similar to each-other. The difference usually originates from the actual value of their filter parameter or from the used sub-query. Examples are shown below.

```
mods[name = a].funs[(name = f) and (arity = 2)].refs
mods[name = b].funs[(name = g) and (arity = 3)].refs
```

or

```
mods.funs[arity = 2]
mods.funs[exported and is_tail_rec]
```

A new abstract level is introduced, where those queries are not only similar to each-other, but those queries do equal to each-other, we called it Skeleton. Examples are shown below.

```
mods[name = $ModName$].funs[name = $FunName$ and
                             arity = $Arity$].refs
mods.funs[$FunSubQuery$]
```

Usage of the skeleton The service of the skeletons is available under 'Queries' menu. A skeleton can be constructed by typing its body, which observes the rules of the valid skeleton, into the query construct assistant, than

'Save as skeleton' button should be pressed. By pressing the 'Save as skeleton' button, a dialogue box appears, where the wanted name of the constructed skeleton should be typed. By clicking the 'Save' button, which is placed in the dialogue box, the skeleton will be saved. The constructed skeleton appears in the list of available skeletons, which is located at the 'Skeletons' tab, whether the save was success. If any error occurs during the save, an error message will be shown.

A previously saved skeleton can be evaluated by calling as a function with the actual values of the parameters. A valid actual parameter can contain nearly anything, only the `'` character is needed to avoid, because the `'` character is the delimiter of the value of an actual parameter. Auto-complete does not only offer the possible endings, but also does offer the joint skeletons.

A valid skeleton call is shown below:

```
Name = skeleton_name

Body = mods.funs[$FunSubQuery$]

skeleton_name(' (arity>0) and (name like s) ').
```

Previously saved skeletons are listed at 'Skeletons' tab.

By clicking the name of the skeleton, a valid 'skeleton call' will be placed in the query construct assistant, where the actual parameters should be written by replacing _ character with the corresponding parameter value.

By clicking the `?` icon, the body of the skeleton and the owner of the skeleton will be shown.

After a successful evaluation of a skeleton, the generated semantic query string and its result are saved in the 'Previous queries' list, and the result of the semantic query will be shown in the right side of the page.

Only the owner of the skeleton can edit its body, or can delete it.

By clicking the corresponding 'E' icon, the body of the skeleton will be placed into the query construct assistant. After the necessary changes had been made then the 'Update skeleton' button has been pressed, the body of the skeleton will be updated.

By clicking the corresponding 'X' icon, the skeleton will be deleted.

4.6 Database operations and environmental nodes

This service is available under "Files" menu. The service works correctly only, if the process has the appropriate right for files and directories. The file browser panel is located at the left side of the page. Files which are located on server, or which had been loaded into the database can be browsed.

Browsing files on the server This mode can be reached by selecting "Browse server" from the drop-down located at the top of the browser. The root directory of the browser is an optional configuration parameter. Possible values are:

- If the `browser_root` parameter is set during start up, the given value is the root directory.
- If the `browser_root` parameter is not set during start up, but the database had contained files before start up, the root elements are the directories of these files.
- If the `browser_root` parameter is not set during start up, and the database had not contained files before start up, then RefactorErl's lib directory will be the root directory.

In this mode, directories can be listed, the contents of files can be shown, and a selected directory or file can be added to database. The selected directory or file is shown by a blue background. Status messages are displayed during the addition of the file to database, and also after the progress has finished.

Browsing loaded files This mode can be reach by selecting "Browse loaded files" from the drop-down located at the top of the browser. In this mode, directories can be listed, contents files can be shown, a selected file or directory can be reloaded to the database, and a selected file can be dropped from the database. Status messages are displayed during the reloading/dropping process of the file to/from the database, and also after the progress has finished.

Dropping a file means only that is drop from the database, so the file is not deleted from file system.

Environmental nodes 'Appbase' environmental nodes can be listed, deleted or set also under Files menu.

Adding/deleting files After selecting a certain file from the browser tab, you can perform database operations, such as adding or dropping it from the database. The progress of the performed operation is shown on a progress bar.

4.7 Errors

This service is available under "Errors" menu.

If database contains file(s) with error form(s), the list of errors will be displayed in a table. A row in the table is equivalent to one error. File name and error message are shown in the table. The place of the error in the source code can be visualised by clicking on the file name. The file is loaded into the text-box, and the place of the error will be highlighted.

If the database does not contain any files with error forms, "No error" message is displayed in the page.

4.8 Dependency examinations

Analyse in function or in module level Dependency examinations can be performed on module or function level on the whole graph or the cyclic sub-graph. A self-defined module or function can be set as the starting node of the examination by the user. This service is available under "Dependency graph" menu at "Function or Module level" tab.

Possible configurations of the examination are:

- configuring the level of the examination: can be set to `module` or `function` by selecting the required option from the "Level" drop-down.
- configuring the type of the examination:
  - `None`: In this case, the type of examination is not given, this mode is used when the user wants to set the starting node of the examination.
  - `Whole graph`: Dependency examinations are performed on the whole graph.
  - `Cyclic sub-graph`: Dependency examinations are performed only on the set of cyclic sub-graphs.
- configuring the starting node of the examination:
  - `module`: the user has to start typing the name of the module in the text-field located under the "Starting node (module)" label. While typing into the text-field, the interface offers possible endings for the string, based on the names of the modules which are loaded to the database. The offered endings can be chosen from a drop-down list.
  - `function`: the user has to start typing the name of the function in the text-field located under "Starting node (function)" label. The name of the function should be given in the following form: `module_name:function_name/arity`. The interface offers possible endings, as described in the upper section.

After the examination has been configured, the "Generate graph" button can be pressed to start it. The result is available in two different graphic formats. By clicking "Generated graph in .dot" link the result is sent to the browser in .dot format. By clicking "See generated graph in .svg" the result is displayed in a new window in the browser.

Analyse in functionblock level Hence dependency analyse can be done not only in function or in module level, but also in functionblock level. This service is available under Dependency graphs menu at 'Functionblock' tab.

Predefined functionblocks are the loaded directories, but a functionblock can be defined by user, too, by typing the needed Perl-like regular expression into the textfield, than by pressing the 'Add' button.

The available functionblocks are placed in the 'Functionblock' labelled textbox, where from the subjects of the analyse can be dragged than can be dropped into

the 'Subjects' labelled textbox. After the 'Subjects' list has been configured, the 'Generate dependency graph' button should be pressed. The result of the current analysis can be seen in svg format, or can be downloaded in dot format from the result panel.

An application, called Graphviz, is needed to generate the result in svg format.

All images generated during examination, are placed in the directory, which has been set during start up by `images_dir` switch. If `images_dir` parameter has not been set, all images are placed in current working the directory.

4.9 Logging out

If the interface is no longer needed, the user can log out by clicking the "Log out `username`" menu. During the logout process the interface clears the users session, generated files which originate from users examinations, and redirects the browser to the login page. The interface deletes neither the queries executed by the user, nor their results, and keeps the state of the database, too.

5 Old web interface for semantic queries

This interface is deprecated, so use the new interface instead of it. The new interface is described at Chapter 4.
We have a possibility to start the tool, load some code into the RefactorErl database, afterwards start a yaws webserver instance by using the `ri` console interface or `referl` script with special options. This webserver then provides a web interface for running semantic queries on the database. A screenshot from the interface is shown in Figure: 6.

5.1 Installation

Yaws does not come with the standard Erlang/OTP, so first of all, you have to download (from http://yaws.hyber.org/download) and install it. Recommended version: 1.88 or higher.

Help for the installation: http://yaws.hyber.org/yaws.pdf (Chapter 2).

5.2 Starting and stopping the web server

Within the RefactorErl, yaws runs in an embedded mode.

You can start the web server either with `referl` script `-yaws` option, or from RefactorErl shell by calling `ri:start_yaws/0/1`. In both cases we get a default configuration with server name `refactorErl`, port `8001` and IP `0.0.0.0`. We can modify server name, port and IP, and we can define the location of our yaws ebin directory.

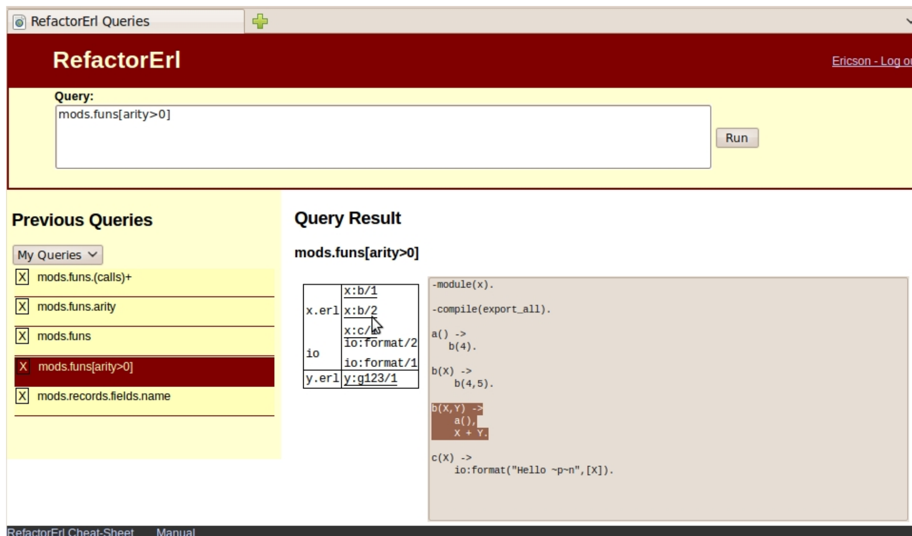


Figure 6: Web interface for semantic queries

Configuration with referl script If we simply start `referl` with `-yaws` option, yaws will be searched in our `erlang/lib` library and will be started with the default configuration.

Options:

- Add your yaws path to the code path: `-yaws_path PATH`
- Change default server name: `-yaws_name NAME`
- Change default port: `-yaws_port PORT`
- Change default IP: `-yaws_listen IP`

Example:

```
referl -yaws -yaws_name "localhost" -yaws_port 8000
       -yaws_path "/home/somebody/yaws-1.88/ebin"
       -yaws_listen 127.0.0.1
```

Configuration with RefactorErl shell If we call `ri:start_yaws()` it will search yaws in our `erlang/lib` library and will start with the default configuration.

By calling `ri:start_yaws/1` we have the same options as with the script options. We can pass a proplist as parameter with `yaws_path`, `yaws_name`, `yaws_port` and `yaws_listen` keys (of course, any of the property can be missing).

Example:

```
ri:start_yaws([{yaws_path,"/home/somebody/yaws-1.88/ebin"},
               {yaws_port,8000},
               {yaws_listen,{127,0,0,1}},
               {yaws_name,"localhost"}
             ]).
```

5.3 Usage

After the web server has been started, open a browser (recommended: Mozilla Firefox) and enter the URL defined by the configuration (the default is `http://localhost:8001/`).

First, you have to login with a username (passwords are not yet supported).

Executing queries Type your query in the input text field and press the `Run` button. The query is added to the list of previous queries, the result of the query appears in the `"Result"` section. Underlined entries are clickable: after a click the Erlang file belonging to the entry appears, and the entry's code part highlights.

Only queries with `modules` or `files` initial selectors are allowed currently.

Selecting a previous query By using a drop-down list you can decide to see either all the previously executed queries or only queries executed in your user profile.

After selecting a previous query in the history its result appears the same way as seen in the case of running new queries. If the database has not been changed since the last execution of the query, its previously stored result appears, otherwise the query is re-executed and then displayed.

Deleting a previous query When we choose to browse our own queries (not all of the queries), we can simply delete a previous query by clicking `X` sign before the choosen query.

6 Querying semantic and syntactic information

6.1 Semantic queries

A semantic query language was designed to query semantic information about Erlang programs.

When you are using Emacs or other GUI-s of RefactorErl use the “Semantic query” menu you can run your own semantic queries by selecting the “Run query” (or using the `C-c C-r s q` shortcut in Emacs) and typing your own semantic queries. It is possible to run queries in `ri` using functions `ri:q/n`. The web interface of the tool not only provide the opportunity to run queries but helps you constructing them.

6.1.1 Language elements

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.

Semantic queries 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.

```
semantic_query ::= initial_selector ('[' filter ']')* ('.' query)*
                       ['.'property] |
                   initial_selector ('[' filter ']')* ('.' query)*
                       ['.'property ':' statistics]

query          ::= selector ('[' filter ']')* |
                   iteration ('[' filter ']')* |
                   closure ('[' filter ']')*

iteration      ::= '{' query ('.' query)* '}' int
closure        ::= '(' query ('.' query)* ')' ( int | '+' )

filter         ::= exp100
exp100         ::= exp200 ['or' exp100]
exp200         ::= exp300 ['and' exp200]
exp300         ::= exp400 [comparator exp400]
exp400         ::= ['not'] exp_max
exp_max        ::= '(' exp100 ')' | ('.' query)+ |
                   atom | int | string
```

Initial selectors 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` 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.

Selectors 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.

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

Transitive closure 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:
$R^0 = R$
$R^i = R^{i-1} \cup \{(r_1, r_3) | \exists r_2 \colon (r_1, r_2) \in R^{i-1} \wedge (r_2, r_3) \in R^{i-1}\}$
$R^+ = \bigcup_{i \in \mathbf{N}} R^i$.
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 ∪ @function.calls.calls ∪ @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.

Property query and statistics Properties are functions that give the value of the property for an entity.
$property \colon entitytype \to valuetype$
$valuetype \in \{atom, string, int, bool\}$

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.

Example:

- To query the value of the property *exported* for the *function*s of the given *file*: `"@file.functions.exported"`

For properties with numeric values statistics are also available. Using these for the results of metric queries can give more information than a simple list of values.

Example:

- To query the average length of the *function*s of the given *file*: `"@file.functions.line_of_code.average"`

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

6.1.2 Entity details

This section provides a reference guide to the elements of the query language. Every entity type is explained together with its selectors and properties.

File entity A file entity corresponds to an Erlang source file. This entity represents the module concept of Erlang, as modules are mapped to files, but header files also fall into this category.

Initial selectors. File entities can be referred at the start of a query by two selectors: `files` yields every file loaded into the refactoring database, and `@file` yields the current file.

Selectors. The following selectors can be used on file entities. Note that every selector can be used on either module or header files, but some of them will give no results for headers (e.g. a header file can't export functions).

`funs` (function): returns the set of function entities defined in the file.

`records` (record): returns every record entity defined in the file.

`macros` (macro): returns the macro entities introduced in the file.

`includes` (file): returns all the file entities included in the file, either directly with a `-include` directive, or indirectly through an include chain.

`included_by` (file): using this selector on a particular file entity gives all the file entities that include the previous file.

`imports` (function): returns the set of function entities that are referred with an import directive in the file.

`exports` (function): returns the set of functions exported in the file.

Properties. These properties are defined for file entities:

`module` (boolean): gives true if the file is a module.

`header` (boolean): returns true if the file is a header (.hrl).

`name` (atom, string): gives the name of the file. In case of modules it does not contain the `.erl` extension to make it easier to work with module names.

`filename` (string): returns the file name as a string with the `.erl` extension.

`dir` (string): returns the directory containing the file.

`path` (string): returns the absolute path of the file (the directory and the complete file name).

Function entity Function entities correspond to Erlang functions. Function entities come from two sources: they are either defined in a source file that is loaded into the database, or they are referred in the loaded code, but their definition is not known. In the latter case, some information about the function is not available.

Initial selector. At the start of a query, the `@fun` selector can be used to refer to the current function (either the function called by the selected expression, or the function being defined at the current selection).

Selectors. The following selectors can be used on function entities:

`refs` (expression): returns every expression that refers to the function. These references can be function applications, import/export directives and implicit function expressions.

`dynref` (expression): returns every expression that refers to the function through a dynamic function call (e.g. an `apply` call, for details see Section 2.5). We note here, that you should run the dynamic call analysis at first (`ri:anal_dyn()`), otherwise the result will be empty.

`calls` (function): returns the set of function entities called in the body of the function.

`dyncalls` (function): returns the set of functions called in body of the function dynamically (for details, see Section 2.5).

`called_by` (function): returns every function that refers to the specific function, either by application or implicit call.

`dyn_calledby` (function): returns the set of functions that calls the given function dynamically (for details, see Section 2.5).

`args` (expression): gives the function arguments as a list of expression entities.

`body` (expression): returns the top-level expressions of each clause body.

`exprs` (expression): returns the top-level argument, guard, and body expressions of each clause.

`vars` (expression): returns the set of variable entities defined (binded) in the bodies of the function.

`file` (file): returns the file entity that the given function is defined in.

Properties. The following properties are defined on function entities:

`name` (atom, string): gives the name of the function.

`exported` (bool): returns true if the function is exported.

`arity` (int): gives the arity (number of arguments).

`bif` (bool): returns true if the function is an auto-imported built-in function.

`pure` (bool): returns true if the function is free of side-effects. We have to mention here, that those functions that are not loaded into the database of RefactorErl (for example, library functions) are considered as impure. If you need a more precise analysis, you should add the affected files to the database of RefactorErl. The knowledge about the BIF-s is built into the tool.

`dirty` (bool): returns true if the function has side-effect.

`defined` (bool): returns true if the definition of the function is loaded into the refactoring software.

`module` (atom): returns the name of the containing module.

`spec` (string): returns the specification of the function (If it is not available in the source code, RefactorErl calculates it.)

Variable entity Variable entities hold all the information about variables, such as binding expressions, scopes etc.

Initial selector. To start a query with a variable entity, use the `@var` selector to refer to the selected variable.

Selectors. The following selectors can be used on variable entities:

`refs` (expression): returns the set of expression entities that refer to the variable.

`bindings` (expression): returns the top-level expression that binds the expression. It is always a pattern expression, e.g. an argument of the function, of a pattern in a case construct. If the binding expression is ambiguous, every possible binding expression is returned.

`fundef` (function): gives the function entity which contains the binding (definition) of the variable.

Properties. There is only one property for a variable:

`name` (atom, string): returns the name (identifier) of the variable.

Expression entity Expressions are the basic syntactic elements of Erlang, and they are represented by expression entities. This is the only syntactic entity in the query language, it is mainly useful to express relations that cannot be calculated by selectors of other entities. Patterns and guards are also represented as expressions, however, there are syntactic units which are not expressions and cannot be handled in this query language, like clauses and module attributes.

Initial selector. A query can be started from the currently selected expression using the `@expr` selector.

Selectors. The following selectors work on expression entities. Note that an expression is always treated as a single unit with its subexpressions, even in case expressions with more clauses like `if` or `case`.

`fundef` (function): returns the function entity which contains the expression.

`funs` (function): returns every function entity referred directly in the expression.

`vars` (variable): returns every variable entity referred in the expression.

`records` (record): returns every record entity referred in the expression.

`macros` (macro): returns every macro entity referred in the expression.

`sub` (expression): returns the set of expressions that are nested in the expression in any depth. This is mainly useful together with the *type* property.

`top` (expression): returns the expression's top-level expression.

`file` (file): returns the file entity where the expression is located.

`origin` (expression): this is an experimental selector, which returns the origin of the expression's value determined by data flow analysis.

`reach` (expression): this is an experimental selector, which returns the places where the expression's value is copied, determined by data flow analysis.

Properties. These properties are defined for expression entities. Note that some of them depends of the context of the expression, which makes possible selecting tail calls or indexing function arguments.

`type` (atom): returns the type of the expression. The type can be one of the following:

- application: `func(Arg1, ...)`
- implicit_fun: `fun func/1`
- `parenthesis`: `(Expr)`
- `tuple`: `{A, B, C}`

- `binary`: `<<3.14/float>>`
- `binary_field`: `3.14/float` in the previous binary
- `record_access`: `Rec#name.fld`
- `record_expr`: `#name{fld=Val}`
- `record_update`: `Rec#name{fld=Val}`
- `record_index`: `#name.fld`
- `match_expr`: `Pattern = Expr`
- `send_expr`: `Pid ! Expr`
- `'bnot'`: `bnot Expr` (there are other prefix operators like this)
- `'+'`: `A + B` (there are other infix operators like this)
- `cons`:

  ```
  [1, 2, 3 | Tail]
  ```

- `list`: `1, 2, 3` in the previous list
- `nil`: `[]`
- `atom`, `char`, `float`, `integer`, `string`: constants
- `variable`: `Var`
- `catch_expr`: `catch Expr`
- `list_comp`:

  ```
  [A || A <- L, A > 0]
  ```

- `list_gen`: `A <- L` in the previous list comprehension
- `filter`: `A > 0` in the previous list comprehension
- `bin_comp`:

  ```
  {<< <<C/utf8>> || <<C:16>> <= Str >>}
  ```

- `binary_gen`: `<<C:16>> <= Str` in the previous binary comprehension
- `block_expr`: `begin ... end`
- `if_expr`: `if ... end`
- `fun_expr`: `fun () -> ... end`
- `receive_expr`: `receive ... end`
- `case_expr`: `case Expr of ... end`
- `try_expr`: `try ... end`

`value` : returns the value of the expression. It is either the value of a constant expression or an operator of an operator-based expression.

`class` (atom): returns the class of the expression: it either can be `expr` (stands for single expressions), `pattern`, or `guard`.

`last` (bool): returns true if the expression is the last expression of the clause.

`index` (int): returns the index of the expression in the containing expression list.

`tailcall` (bool): returns true if the expression is a tail call. Tail call expressions are function applications, which are the last expressions of their containing clause.

Record entity Records are represented with record entities, which provide access to record information.

Initial selector. To start a query on the currently selected record, use the `@rec` selector.

Selectors. These are the selectors for record entities:

`refs` (expression): returns the set of expression entities that refer to the record either by field access, record update or field index.

`fields` (field): returns the list of fields of a particular record.

`file` (file): returns the file entity that the record is defined in.

Properties. There is only one record property:

`name` (atom, string): gives the record's name.

Record field entity The record field entity provides access to each record's field-specific information.

Initial selector. Record field-related queries can be started with the `@recfield` selector to get the current record field.

Selectors. These are the record field selectors:

`refs` (expression): returns every expression that refers to the given field either by field access, index, or record update.

`record` (record): returns the record which has the particular field.

Properties. Record fields also have only one property:

`name` (atom, string): returns the name of the field.

Macro entity Preprocessor macro directives can be queried using the macro entity.

Initial selector. The `@macro` selector can be used to start a query with the currently selected macro.

Selectors. The following selectors are available for macros:

`refs` (expression): returns every expression that refers to the given macro with the form `?Macro...`.

`file` (file): returns the file entity that the macro is defined in.

Properties. Macro entities have the following properties:

`name` (Atom, string): returns the identifier of the macro.

`arity` (int): returns the arity of a given parametric macro. If the macro is a constant, the arity is 0.

`const` (bool): returns true if the given macro is a constant. Note that there are parametric macros with no parameters, these are treated as non-constant macros.

6.1.3 Examples

Basic queries As you can read in the introduction, in this language we build difficult queries from lot of very simple queries. Here are some examples for simple ones:

`@fun.refs` : returns a list of expressions which call the pointed function.

`@file.funs.calls` : returns all function calls from current module group by the module's own functions.

`@file.funs[arity==3]` : returns all functions which have 3 arguments.

Advanced queries Let's see some useful queries:

`@file.funs.vars[name=="Expl"]` : returns all functions which have a variable named "Expl". It useful when we want to know which functions use variables with same name.

`mods[name=="io"].funs[name==format].refs` : returns all io:format calls, this query is very useful when you have finished your software, and you want to find all debug messages.

`@expr.origin` : for example we stand in a variable, and run this query, we get information about the variable gets its value from where. This functionality uses data-flow analysis.

`@fun.refs.origin` : returns information about the function gets its return value from where and how its calculated.

6.1.4 Entity details

In this subsection, we list the names of initial selectors, selectors and properties and their possible abbreviations and synonyms.

Initial selectors	
Name	*Synonyms*
`@function`	`@fun`
`@variable`	`@var`
`@record`	`@rec`
`@recfield`	`@field`
`@macro`	-
`@expression`	`@expr`
`@module`	`@mod`
`modules`	`mods`
`@file`	-
`files`	-
`@definition`	`@def`

File entity			
Selectors		**Properties**	
Name	*Synonyms*	*Name*	*Synonyms*
`function`	`functions, fun, funs`	`module`	`is_module, mod, is_mod`
`record`	`records, rec, recs`	`header`	`is_header`
`macro`	`macros`	`name`	-
`includes`	-	`directory`	`dir`
`included_by`	-	`path`	-
`imports`	-		
`exports`	-		

Function entity			
Selectors		**Properties**	
Name	*Synonyms*	*Name*	*Synonyms*
references	refs, ref, reference	exported	-
calls	-	name	-
called_by	-	arity	-
arguments	args	bif	-
body	-	pure	-
expressions	exprs, expr, expression	defined	-
variables	vars, var, variable	module	mod
file	-	dirty	-
dynamic_ calls	dynref, dynrefs	spec	-
dynamic_ calls	dyncall, dyncalls		
dynamic_ called_ by	dyncalled_by		

Variable entity			
Selectors		**Properties**	
Name	*Synonyms*	*Name*	*Synonyms*
references	refs, ref, reference	name	-
bindings	-		
fundef	-		

Record entity			
Selectors		**Properties**	
Name	*Synonyms*	*Name*	*Synonyms*
references	refs, ref, reference	name	-
fields	-		
file	-		

Record field entity			
Selectors		**Properties**	
Name	*Synonyms*	*Name*	*Synonyms*
`references`	`refs, ref, reference`	`name`	-
`record`	`rec`		
`file`	-		

Macro entity			
Selectors		**Properties**	
Name	*Synonyms*	*Name*	*Synonyms*
`references`	`refs, ref, reference`	`name`	-
`file`	-	`arity`	-
		`const`	-

Expression entity			
Selectors		**Properties**	
Name	*Synonyms*	*Name*	*Synonyms*
`fundef`	-	`type`	-
`functions`	`function, fun, funs`	`value`	`val`
`variables`	`vars, var, variable`	`class`	-
`records`	`record, rec, recs`	`last`	`is_last`
`macro`	`macros`	`index`	-
`subexpression`	`sub, esub, subexpr`	`tailcall`	`is_tailcall`
`parameter`	`param`	`has_side_effect`	`dirty`
`top_expression`	`top, top_expr`		
`file`	-		
`dynamic_functions`	`dynfun, dynfuns`		

Metrics for files (as properties)	
Name	*Synonyms*
module_sum	mod_sum
line_of_code	loc
char_of_code	choc
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_funpaths, num_of_funpath, num_of_funpaths
function_calls_in	fun_calls_in
function_calls_out	fun_calls_out
cohesion	coh
otp_used	otp
max_application_depth	max_app_depth
max_depth_of_calling	max_depth_calling, max_depth_of_call, max_depth_call
min_depth_of_calling	min_depth_calling, min_depth_of_call, min_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_funexpr	num_of_funexpr
number_of_messpass	num_of_messpass
fun_return_points	fun_return_point, function_return_points, function_return_point
max_length_of_line	-
average_length_of_line	avg_length_of_line
no_space_after_comma	-

Table 2: List of metrics for modules

Metrics for functions (as properties)	
Name	*Synonyms*
line_of_code	loc
char_of_code	choc
function_sum	fun_sum
max_application_depth	max_app_depth
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
calls_for_function	calls_for_fun, call_for_function, call_for_fun
calls_from_function	calls_from_fun, call_from_function, call_from_fun
number_of_funexpr	num_of_funexpr
number_of_messpass	num_of_messpass
fun_return_points	fun_return_point, function_return_points, function_return_point
max_length_of_line	-
average_length_of_line	avg_length_of_line
no_space_after_comma	-
is_tail_recursive	-

Table 3: List of metrics for functions

6.1.5 Statistics

In this subsection, we list the names of selectors and their possible abbreviations and synonyms.

Statistics	
Name	*Synonyms*
`minimum`	`min`
`maximum`	`max`
`sum`	`-`
`mean`	`average, avg`
`median`	`med`
`variance`	`var`
`standard_deviation`	`sd`

6.2 Metric queries embedded into semantic queries

6.2.1 Metrics as semantic query properties

In RefactorErl, metrics can be applied to modules or to functions. Modules are equivalent to `file` entities in the semantic query language, and functions are equivalent to `function` entities. We can say that a metric is a kind of property belongs to a `file` or `function` entity, so we can simply add the proper metrics to the properties of entities.

Motivation: checking coding conventions with metrics Usually we have some coding conventions applied to our modules or functions. With our extended semantic query language we can check these conventions, and filter improper modules or functions.

We have studied the *Effects of software design rules on refactoring* in an earlier report (*Erlang Refactoring: New Refactoring Steps, 31.05.2007.*). That report focused on the relation with refactoring, but also listed a number of "Programming Rules and Conventions". Most of that design rules refer to programming style, revision information, legibility and comments. The currently implemented metrics do not work with revision or comment information, but that information is present in the semantic program graph, so the implementation is possible.

Hereinafter we present some design rules from the mentioned report and some metrics to check these rules.

Rule1: A module should not contain more then 400 lines. When we would like to filter modules containing more than 400 effective lines of code, we have to load our modules to RefactorErl system, and enter the following query:

```
modules[line_of_code > 400]
```

In the result we will find our too long modules.

Rule2: A function should not contain more then 15 to 20 lines. When we would like to check, which functions do not fulfil this convention in our modules loaded into the RefactorErl database, we use the following query:

```
modules.funs[line_of_code > 20]
```

Rule3: Use at most two level of nesting, do not write deeply nested code. It is achieved by dividing the code into shorter functions. With one of our metrics we can count the nesting level of case expressions, so we can filter functions with more than two maximum depth of cases. In this example, we would like to get the result just from our actual module.

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

If we just would like to know, whether all of the functions fulfil this convention or not, we can simply query the maximum nesting level of cases in the whole module. If this value is more than two, there is at least one function containing deeply nested cases.

```
@file.max_depth_of_cases
```

At least, let's filter modules containing functions with too deeply nested cases.

```
mods[max_depth_of_cases > 2]
```

Rule4: Use no more than 80 characters on a line. We can filter all of the functions, which contains lines with more than 80 characters with the following query:

```
mods.funs[max_length_of_line > 80]
```

Rule5: Use space after commas. We have a metric which returns with the number of cases when we do not fulfil this convention. When a modul or a function breaks this rule, the result of the metric will be more, then 0.

Filter functions containing at least one case when whitespace misses after a comma:

```
mods.funs[no_space_after_comma > 0]
```

Rule6: Every recursive function should tail recursive. Tail recursion means that we have no recursive call (either direct or indirect) in our function, just in the last expression. Filter functions that recursive, but not tail recursive:

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

Short description of the metrics You can find the list of metrics can be used as properties in semantic queries in tables 2 and 3. The tables give the original names and synonyms of the metrics.

6.3 Metric queries

A metric query language was designed to query some metric information about Erlang programs.

6.3.1 Defined metrics

module_sum - The domain of the query is a module. The sum of the chosen complexity structure metrics measured on the modules functions. The proper metrics adjusted in a list can be implemented in the desired number and order.

line_of_code - The domain of the query is a module or a function. The number of the lines of part of the text, function, or module. The number of empty lines is not included in the sum. As the number of lines can be measured on more functions, or modules and the system is capable of returning the sum of these, the number of lines of the whole loaded program text can be enquired.

char_of_code - The domain of the query is a module or a function. The number of characters in a program script. This metric is capable of measuring both the codes of functions and modules and with the help of aggregating functions we can enquire the total and average number of characters in a cluster, or in the whole source text.

number_of_fun - The domain of the query is a module. This metric gives the number of functions implemented in the concrete module, but it does not contain the number of non-defined functions in the module.

number_of_macros - The domain of the query is a module. This metric gives the number of defined macros in the concrete module, or modules. It is also possible to enquire the number of implemented macros in a module.

number_of_records - The domain of the query is a module. This metric gives the number of defined records in a module. It is also possible to enquire the number of implemented records in a module.

included_files - The domain of the query is a module. This metric gives the number of visible header files in a module.

imported_modules - The domain of the query is a module. This metric gives the number of imported modules used in a concrete module. The metric does not contain the number of qualified calls (calls that have the following form: `module:function`).

number_of_funpath - The domain of the query is a module. The total number of function paths in a 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. It is very similar to the metric called cohesion.

function_calls_in - The domain of the query is a module. Gives the number of function calls into a module from other modules. It can not be implemented to measure a concrete function. For that we use the `calls_for/1` function.

function_calls_out - The domain of the query is a module. Gives the number of every function call from a module towards other modules. It can not be implemented to measure a concrete function. For that we use the `calls_from/1` function.

cohesion - The domain of the query is a module. The number of call-paths of functions that call each other. By callpath 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.

function_sum - The domain of the query is a function. The sum calculated from the functions complexity metrics that characterizes the complexity of the function. It can be calculated using various metrics together. We can define metrics that are necessary to calculate the metrics constituting the sum (with enumeration in the `referl_metrics` module).

max_depth_of_calling - The domain of the query is a module or a function. The length of function call-paths, namely the path with the maximum depth. It gives the depth of non-recursive calls. Recursive calls are provided by `depth_of_recursion/1` function. The depth of calling in the following example is 3.

```
...
f([A|B], Acc) ->
   Acc0 = exec(A, Acc),
   f(B, Acc0);
f([], Acc0)->
    Acc0.

exec(A, Acc)->
   io:format("~w",[A]),
   A + Acc.
...
```

max_depth_of_cases - The domain of the query is a module or a function. Gives the maximum of case control structures embedded in *case* of a concrete function (how deeply are the case control structures embedded). 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. However, the following embedding does not increase the sum.

```
...
A = case B of
       1 -> 2;
       2 -> ok
    end
...
```

max_depth_of_structs - The domain of the query is a module or a function. Gives the maximum of structures embedded in function. (how deeply are the *block*, *case*, *fun*, *if*, *receive*, *try* control structures embedded) In case of a module it measures the same regarding all the functions in the module.

number_of_funclauses - The domain of the query is a module or a function. Gives the number of a functions clauses. Counts all distinct branches, but does not add the functions having the same name, but different arity, to the sum. The number of funclauses in the following example is 2.

```
...
f(Fun, [H|Tail])->
  Fun(H),
  f(Tail);

f(_, [])->
  ok.

f(A, B)->
 A + B.
...
```

branches_of_recursion - The domain of the query is a module or a function. Gives the number of a certain function's branches, how many times a function calls itself, and not the number of clauses it has besides definition. The branches of recursion in the following example is 2.

```
quicksort([H|T]) ->
    {Smaller_Ones,Larger_Ones} = split(H,T,{[],[]}),
    lists:append( quicksort(Smaller_Ones),
                  [H | quicksort(Larger_Ones)]
                );
quicksort([]) -> [].

split(Pivot, [H|T], {Acc_S, Acc_L}) ->
    if Pivot > H -> New_Acc = { [H|Acc_S] , Acc_L };
```

```
        true      -> New_Acc = { Acc_S , [H|Acc_L] }
    end,
    split(Pivot,T,New_Acc);
split(_,[],Acc) -> Acc.
```

calls_for_function - The domain of the query is a function. This metric gives the number of calls for a concrete function. It is not equivalent with the number of other functions calling the function, because all of these other functions can refer to the measured one more than once.

calls_from_function - The domain of the query is a function. This metric gives the number of calls from a certain function, namely how many times does a function refer to another one (the result includes recursive calls as well).

number_of_funexpr - The domain of the query is a module or a function. Gives the number of function expressions in a module. It does not measure the call of function expressions, only their initiation. In the next example the number of the funexpr is 1.

```
...
F = fun(A) -> A + 1 end,
F(1),
F2 = fun a/1,
...
```

number_of_messpass - The domain of the query is a module or a function. In case of functions it measures the number of code snippets implementing messages from a function, while in case of modules it measures the total number of messages in all of the modules functions.

fun_return_points - The domain of the query is a module or a function. The metric gives the number of the functions possible return points (or the functions of the given module).

average_size - The domain of the query is a module or a function. The average value of the given complexity metrics (e.g. Average *branches of recursion* calculated from the functions of the given module).

max_length_of_line - The domain of the query is a module or a function. It gives the length of the longest line of the given module or function.

average_length_of_line - The domain of the query is a module or a function. It gives the average length of the lines within the given module or function.

no_space_after_comma - The domain of the query is a module or a function. It gives the number of cases when there are not any whitespaces after a comma or a semicolon in the given module's or function's text.

is_tail_recursive - The domain of the query is a function. It returns with 1, if the given function is tail recursive; with 0, if it is recursive, but not tail recursive; and -1 if it is not a recursive function (direct and indirect recursions are also examined). If we use this metric from the semanctic query language (see section 6.2), the result is converted to `tail_rec`, `non_tail_rec` or `non_rec` atom.

mcCabe - McCabe cyclomatic complexity metric. Available for modules and functions. We define it based on the control flow graph of the functions with the number of different execution paths of a function, namely the number of different outputs of the function.

otp_used - Gives the number of OTP callback modules used in modules.

6.3.2 Aggregations, filters on query results

We can extend our queries with filters. A filter can be formatting and aggregating function or the definition of the structure of the result.

The possible aggregation filters are listed below:

max - maximum on the result list

tolist - default return value of the query

totext - string format of the result

fmaxname - maximum with the name of the node

avg - average on the result list

min - minimum of the result list

sum - sum of the result list

6.3.3 Examples

Simple query With the following query we count the number of functions of the modules given in the list.

```
show number_of_fun for module ('a','b')
```

where

- A *number of fun* Number of fun a function giving the number of functions,
- A *module* Module the type of node in the query,

- A $('a','b')$ contains the names of modules in which we calculate the metrics. In case the type of the node was defined as function the list must contain the following elements: The name of the module, in which the function was defined, the name of the function and its arity. In this case the list can have more than one element. The next list `{'test','f',1}` defines a function which is defined in the `test` module. Its name is `f`, and its arity is `1`.

Advanced query In the next example we would like to define the number of recursive calls of two functions defined in the `a` module, the number of branches on which the particular function calls itself, and we sum up the two results with the help of the `sum` aggregating function.

```
show branches_of_recursion for
                 function ({'a','f',1},{'a','g',0}) sum
```

At the end of queries we can place filters which filter the results that are received at the output, or which are aggregating functions which change the result of the query.

6.4 Using metric queries from different interfaces

Using one of the integrated GUI of RefactorErl you can use the metric queries by selecting the Semantic query menu. For example, in Emacs, selecting the Run metric query from the submenu or using the C-c C-r m q shortcut you can type your own metric queries in the minibuffer and run it. From the console interface, you can use the `ri:metric/1` function.

6.5 Metric analyser mode

The metric analyser mode has to be manually enabled by selecting `Start metrics` from the `Server` item in the `RefactorErl` menu in the Emacs interface of RefactorErl. Clicking `Start metrics` from the same menu turns the mode off. When Metrics mode is on, the status indicator displays `Erlang Refact Metrics` or a similar message. When metrics mode is turned on, RefactorErl initializes the internal metrics representation by creating the necessary tables, and loading them with the available module and function nodes. It also calculates the initial values of the metrics.

The metrics mode of RefactorErl can also be enabled by invoking `ri:metricmode(on)` from the command line. The mode can be turned off by calling `ri:metricmode(off)`, and its current status can be queried by `ri:metricmode(show)`.

The limits of the metrics can currently be configured by editing the file `metricmod.defs`. Figure 7 shows an example of the current format of the file.

The file contains two Erlang terms, one for the module level metrics and another for the function level metrics. The metrics analyser system has built-in defaults; any options given here override the defaults. For a given metric, the lower and upper limits can be given, e.g. the limits on the lines of code in the module are overridden in this file so that they are considered correct only if they are between 100 and 1000.

```
{module_metrics,
    [{line_of_code,{100,1000}},
     {char_of_code,{100,60000}},
     {number_of_fun,{0,10}},
     ...]}.
{function_metrics,
    [{line_of_code,{0,20}},
     {char_of_code,{0,600}},
     {function_sum,{0,infty}},
     ...]}.
```

Figure 7: Contents of metricmod.defs

When in metric analyser mode, the Emacs interface of RefactorErl displays the values of changed metrics after a transformation is invoked. The analyser opens up the `Metrics Result` buffer, and shows the expected and the current values of the measured metrics for all modules that have at least one metric that is out of bounds as shown in Figure 8.

Values of metrics outside the user defined limits ("bad code smells") can also be queried manually using `Show bad smells` in the `Semantic query` menu. Figure 9 shows the output of this query, which is quite similar to Figure 8; the main difference is that Figure 9 shows all bad metric values, while the other displays only those that are affected by the transformation.

The RefactorErl console also supports querying bad smells in the code. After calling `ri:metricmode(on)`, the user can call `ri:metricmode(show)`, which returns a term that describes the modules and functions where the metrics have values outside the user defined limits. Figure 10 shows an example output of this call; for example, the module `a` has 5 lines of code, which does not fit the arbitrary range 10..20.. The mode can be turned off by calling
`ri:metricmode(off)`

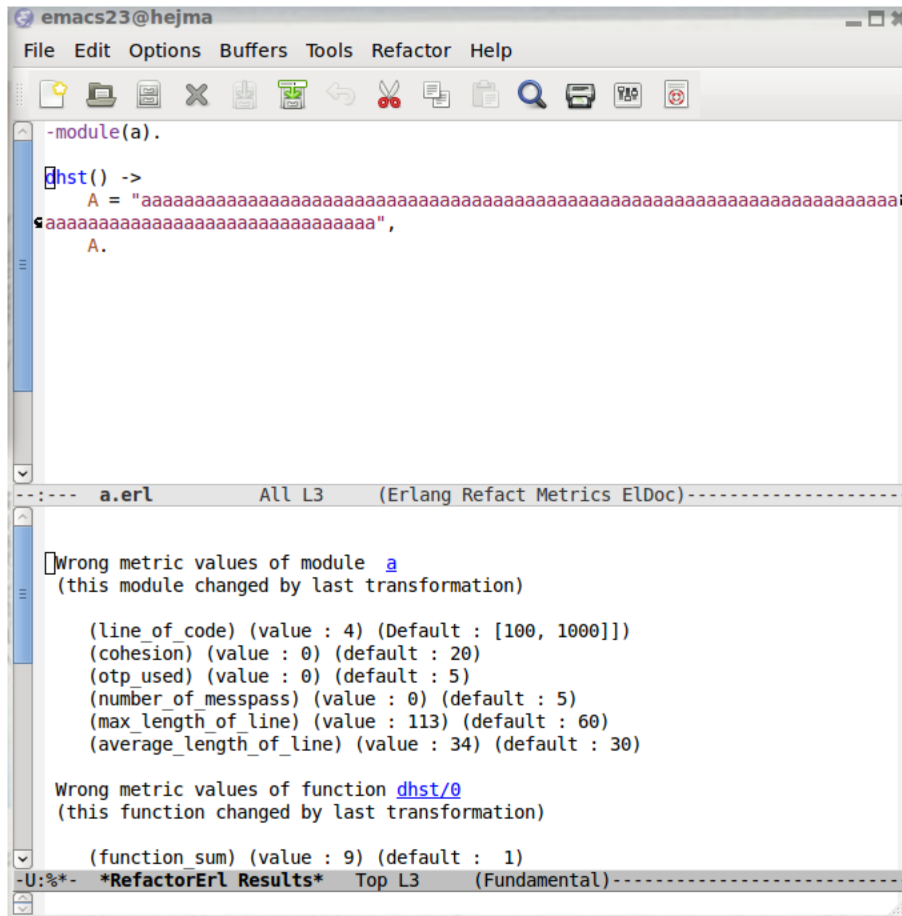


Figure 8: Bad metric values after a transformation

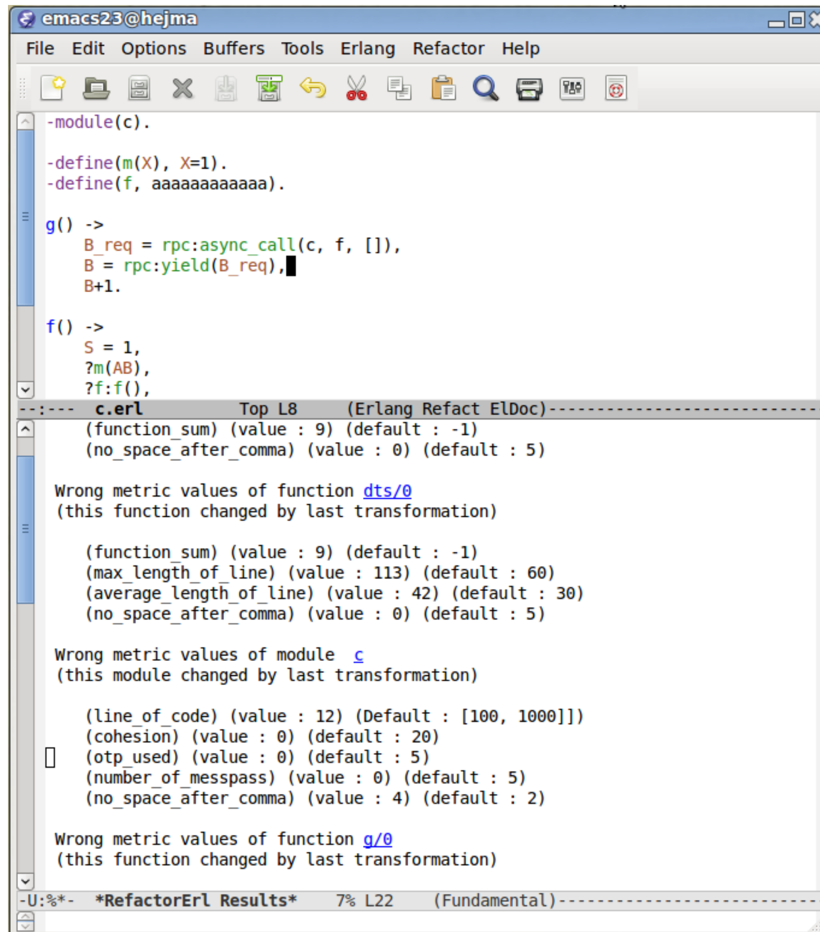


Figure 9: Bad metric values in all loaded code

```
[{module, a,
    [{line_of_code, 5, {10, 20}},
     {char_of_code, 100000, {100,60000}},
     {number_of_fun, 42, {0,10}},
     ...]},
 {function, f, 1,
    [{line_of_code, 21, {0,20}},
     {char_of_code, 601, {0,600}},
     ...]}].
```

Figure 10: Console bad smell output

7 Basic usage in Emacs/XEmacs

The GNU Emacs and the XEmacs text editors are supported user interfaces. At least version 22 is required for Emacs and version 21 is required for XEmacs.

Refactoring is performed by an Erlang server process, which stores analyzed source files in a database. This server is started automatically by the (X)Emacs interface, and it should be stopped manually before leaving (X)Emacs. Only those files that are loaded into the database are subjects to refactoring.

7.1 Configuration in XEmacs/Emacs.

Loading RefactorErl. First of all, you need to load the RefactorErl library into XEmacs or into Emacs. You can do this by placing the following two lines into your `.xemacs/init.el` file in the former or into your `.emacs` file in the latter case (you can copy&paste it from the `README.TXT` file from the RefactorErl package):

```
(add-to-list 'load-path
             "/path-to-root-dir/lib/referl_ui/emacs")
(require 'refactorerl)
```

If you don't know where your `.emacs` file is, the easiest way to locate it is to start Emacs and type `C-x C-f ~/.emacs` – this always opens the right file. Similar in XEmacs type `C-x C-f ~/.xemacs/init.el`

If you don't know what `C-x C-f` is, then you should read the (X)Emacs tutorial to get more familiar with the environment – you can access it by starting (X)Emacs and pressing the keys `Ctrl+h`, and then `t`.

Basic settings. Before you can run RefactorErl, you must make some settings through the Emacs customization system.
Type `M-x customize-group`, then enter `refactorerl` as the name of the group. Here you can change the following settings:

1. `Refactorerl Base Path`: you must enter the full path to the root directory of the tool. Strictly speaking, this is the only required customization, every other setting has a sensible default.

2. `Refactorerl Erlang Runtime`: if your `PATH` setting doesn't contain the Erlang commands, then you should enter the full path to the `erl` command here.

3. `Refactorerl Data Dir`: you can change the location of the database here.

4. `Refactorerl Server Type`: if you want to experiment with the command line interface of the tool, you should set this to "Managed server with shell access". This feature requires the standard Erlang editing mode.

After setting "Refactorerl Base Path" (and maybe "Erlang Runtime" –

```
(add-to-list 'exec-path "/path_to_erlang/bin")
(add-to-list 'load-path "/path_to_erlang/lib/tools-2.6.2/emacs")
(require 'erlang-start)
```

), you should be able to use the tool.

7.2 RefactorErl mode

Refactoring functionality is provided through an (X)Emacs minor mode[1] that can be turned on using the command `M-x refactorerl-mode`. When this mode is active, there is a status display in the mode line that shows one of the following values:

- `Refact` means RefactorErl mode is active, and the file can be refactored.
- `Refact:off` means RefactorErl mode is active, but the file is not in the database, so it cannot be refactored.
- `Refact:err` means the file has some errors.
- `Refact:???` means that RefactorErl mode is active, but there is no information available about the file. Normally this is shown only during processing, use the "update status" command if it doesn't go away.

When RefactorErl mode is active, functionality can be accessed through either the "Refactor" menu or keyboard shortcuts. Key sequences start with `C-c C-r`, and a complete list is given by `C-c C-r C-h`.

As a standard (X)Emacs feature, help on RefactorErl keyboard shortcuts is available by typing `C-h k` followed by the keyboard command itself. For example, `C-h k C-c C-r r f` provides help on the *rename function* refactoring. The same works with menus: selecting a menu item after typing `C-h k` gives help on the menu item.

7.3 The "Refactor" menu

RefactorErl provides a user-friendly interface in Emacs and in XEmacs. A snapshot from the Emacs menu is shown in Figure: 11.

7.3.1 Stopping and starting the server

While working with Erlang files the sever process of the RefactorErl can be stopped or can be restarted if it is required. The refactoring server can be-stopped by selecting the "Stop server" menu item from the "Refactor" menu or typing the shortcut `C-c C-r Q`. The RefactorErl server can be started (or restarted) by selecting the menu item "Start server" from the Refactor menu or typing the `C-c C-r R` shortcut.

[1] A minor mode is an optional Emacs feature that can be turned on or off.

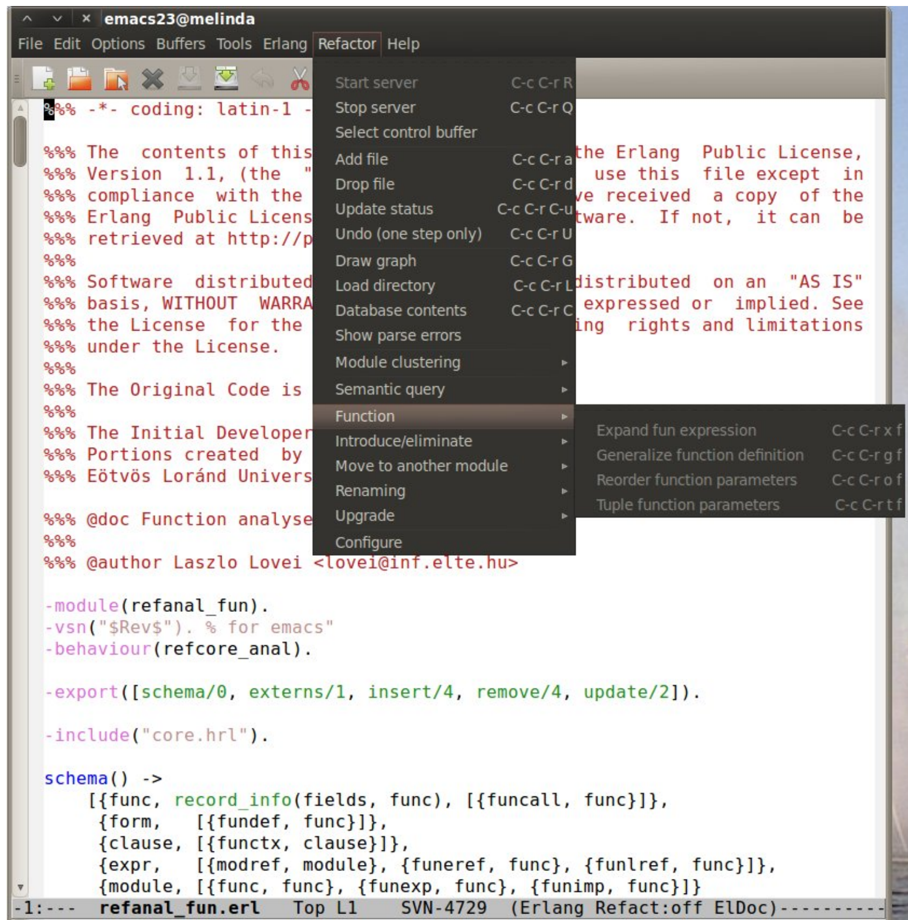


Figure 11: RefactorErl in Emacs

7.3.2 Control buffer.

Enabling `refactorerl-mode` in Emacs automatically starts the “RefactorErl” control buffer. This buffer contains some RefactorErl related functionality and settings.

Configuration.

- Application directories - you can add or remove application directories, it requires a full path to the selected application directory
- Include directories - you can add or remove include libraries with header files, it requires a full path to the selected include directory

- Output directory - you can change the output directory, where to save the file(s) after the transformation

Reset database. With this item you can reset the content of the entire database. It removes every loaded module, header file from the database and deletes the previously set include directories too.

Show Files. Selecting this item lists the content of the database. Every row corresponds to a loaded file and contains a hyperlink to the actual file. Selecting a hyperlink opens the corresponding file in a buffer.

Show parse errors. See in 7.3.5.

Clear buffer. Resets the control buffer to the last saved state. If the there are some changes that have not been saved, these changes will be lost after clearing the buffer.

7.3.3 Selecting files for refactoring.

A single file can be loaded into the database by opening it, enabling RefactorErl mode for it, and using the "Add file" command (in the menu, or by the keys `C-c C-r a`). The "Drop file" command (`C-c C-r d`) removes the file from the database. When a file is modified and saved, it is automatically updated by removing and re-adding its changed forms.

The whole contents of a directory can also be loaded without going through the files one-by-one, using the "Load directory" command in the menu (or `C-c C-r L`). The actual contents of the database are shown by the command "Database contents" (`C-c C-r C`), the list is shown in the "RefactorErl File List" buffer with links to the files themselves and buttons to remove them from the database.

7.3.4 Undo.

Undo is supported for all refactorings except for renaming a module. In order to use it, execute "Undo (one step only)" (`C-c C-r U`). This restores the files to their state before the last refactoring step. This restoration includes dropping all the changes that you may have done to these files. You have been warned.

For renaming a module, the easiest way to undo the transformation is to do the inverse transformation: rename the module to its previous name.

7.3.5 Show parse errors.

Selecting the "Show parse errors" menu item from the "Refactor" menu lists the parse errors in the database which arose while adding files to the database (parse errors for files where the indicated status is "error"). Every parse error is described in a separate row and gives information about the file which contains

the actual parse error, the location in the file and context information about the error. The parse errors are listed in the “RefactorErl File List” buffer with links to the location of the error.

7.3.6 Draw graph.

There is interface to visualize the content of the database. Choosing the “Draw graph” or typing the `C-c C-r G` shortcut the interface asks for a file name with extension `.dot` where to create the description of the database content. The tool creates the description of the graph to the given path and file, using Graphviz a graphic visualization of the graph can be created. Note that the resulting graph may be too large for Graphviz to handle.

7.3.7 Semantic Queries

You can use the “Semantic Query” menu item to run semantic queries or you can use the shortcut `C-c C-r s q`. For more details about the queries see Section 6.

7.3.8 Clustering

For details about the clustering see Section 9.

7.3.9 Refactorings

You can use the menu or shortcuts to call a refactoring. For details about the refactorings see Section 8. The following refactorings are available in version 0.9.12.01 of RefactorErl.

Transformation name	**Selection**	**Shortcut**
Rename function	Def, call	`C-c C-r r f`
Rename header	In module	`C-c C-r r h`
Rename macro	Def, use	`C-c C-r r c`
Rename module	In module	`C-c C-r r m`
Rename record field	Def	`C-c C-r r r f`
Rename record	Def	`C-c C-r r r d`
Rename variable	Def, use	`C-c C-r r v`
Move function	Pop-up	`C-c C-r m f`
Move macro	Pop-up	`C-c C-r m m`
Move record	Pop-up	`C-c C-r m r`
Eliminate function call	Function call	`C-c C-r e f`
Eliminate fun expression	Fun expression	`C-c C-r e u`
Eliminate variable	Occurrence	`C-c C-r e v`
Eliminate macro substitution	Substitution	`C-c C-r e m`
Introduce function	Expression, body	`C-c C-r i f`
Introduce import	Qualifier	`C-c C-r i i`
Introduce function argument	Expression	`C-c C-r i a`
Introduce record	Tuple	`C-c C-r i r`
Introduce tuple	Def, call	`C-c C-r i t`
Introduce variable	Expression	`C-c C-r i v`
Transform list comprehension	Expression	`C-c C-r t l`
Reorder function parameters	Def, call	`C-c C-r o f`
Upgrade interface: regexp	Anywhere	`C-c C-r u i r`

8 Using refactorings

Note that in the current output of the tool, the layout of the code parts changed by the refactorings are pretty printed. This does not affect the layout of those code parts that are left untouched.

8.1 Rename function

The name is an important property of a function. While its actual value does not influence the semantics of the program, it is a key point in the readability of the code, and because Erlang programs mainly consist of functions, their names play an often underestimated role in software development and support. It is worth the effort to try to choose them well, and to correct misleading names even when they are user throughout the code. This transformation helps in the latter case.

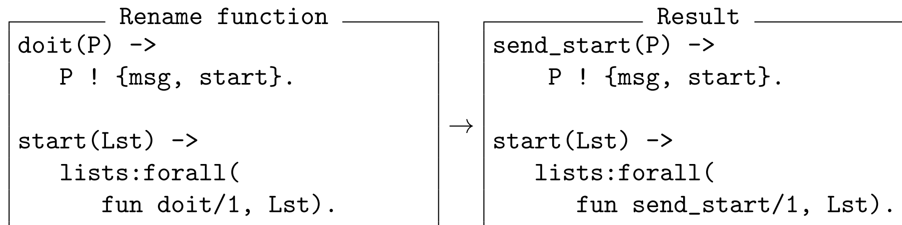


Figure 12: A simple function renaming.

8.1.1 Usage in Emacs

- Add the file that contains the function to the database (`C-c C-r a`).
- Position the cursor over the name of the function in any clause of the function definition, any application of the function or the function in an export list.
- Call the refactoring from the menu or with `C-c C-r x r f`.
- Type the new function name.

8.1.2 Side conditions

- There must be no function with the given name and the same arity as the function to be renamed among the functions in the module, the functions imported in the module, and the auto-imported BIFs.
- There must be no function with the given name and the same arity as the function to be renamed among the local and imported functions in the modules that import the function to be renamed.

- If the user does not specify a function to be renamed or the specified function does not exist, the transformation starts an interaction to ask the user to specify one. The user has to select a function from a radio group.

8.1.3 Transformation steps and compensations

1. The name label of the function is changed at every branch of the definition to the new one.
2. In every static call to the function, the old function name is changed to the new one.
3. Every implicit function expression is modified to contain the new function name instead of the old one.
4. If the function is exported from the module, the old name is removed from the export list and the new name is put in it.
5. If the function is imported in an other module, the import list is changed in that module to contain the new name instead of the old one.

8.2 Rename header

The rename header refactoring renames the header file to the given new name, and makes changes in those files in which it is referred to.

If the new name of the header file contains a path and this path is not equal to the original one, the transformation moves the header file to its new place and renames it.

In the example in Fig. 13 the `header1.hrl` is renamed to `newname`. The file and all of its references are also renamed to `newname`.

8.2.1 Usage in Emacs

- Add the header file to the database (`C-c C-r a`).
- Position the cursor into a header file.
- Call the refactoring from the menu or with `C-c C-r x r h`.
- Type the new name of the header file into a status line.

Side conditions

- The type of the file has to be a header file. If the pointed file is a module, the transformation will fail.
- The directory must not contain a file having the same name as new name given. If it contains, the transformation starts an interaction to ask for a new name.

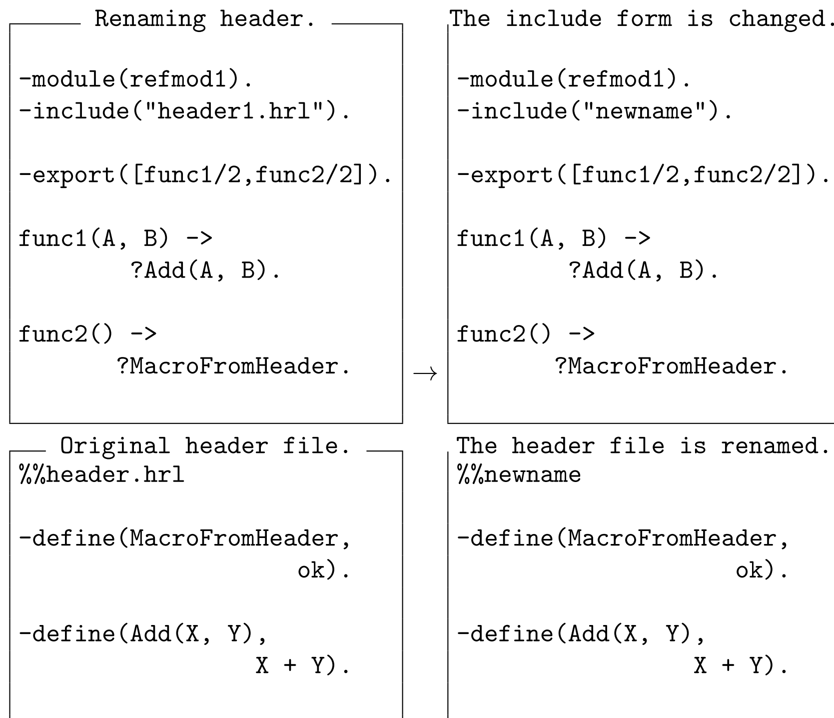


Figure 13: Renaming header file `header1.hrl` to `newname`.

Transformation steps and compensations

1. Rename the header file name to the new name on the graph.
2. Rename the references to the header file in the include forms. (Actually, the include form will be deleted and recreated with a new path and file name).
3. Rename or move and rename the file to the new name.

8.3 Rename macro

The "rename macro" transformation renames a macro and all of its occurrences in either modules and header files. The condition of the renaming is that there is no name conflict with another record in the file containing the macro, in any of its includes or anywhere it has been included at.

8.3.1 Usage in Emacs

- Add the file to the database (`C-c C-r a`).
- Position the cursor over any macro definition or application.

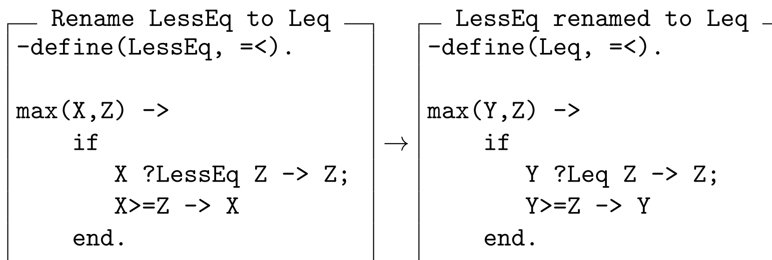


Figure 14: Renaming macro `LessEq` to `Leq`

- Call the refactoring from the menu or with `C-c C-r x r c`.
- Type the new macro name.

8.3.2 Side conditions

- No macro already exist with the same new name in either
  - in a file hosting definition or usage of the macro,
  - in files included by the said,
  - in files that include the said
- If one of the above conditions fails, the transformation starts an interaction to ask for a new macro name.
- If the user does not specify a macro or the specified macro does not exist, the transformation starts an interaction to ask for a macro.

8.3.3 Transformation steps and compensations

1. The macro name is replaced with the new name at both definition and all usage sites

8.4 Rename module

The rename module refactoring renames a module with the given new name. Renames the file and make changes in other file where this module is referenced.

In the example in Fig. 15 the `mod1` module is renamed to `newmod`. The file is also renamed to `newmod.erl`.

8.4.1 Usage in Emacs

- Add the file that contains the module to the database (`C-c C-r a`).
- Position the cursor over the name in the module attribute.
- Call the refactoring from the menu or with `C-c C-r x r m`.

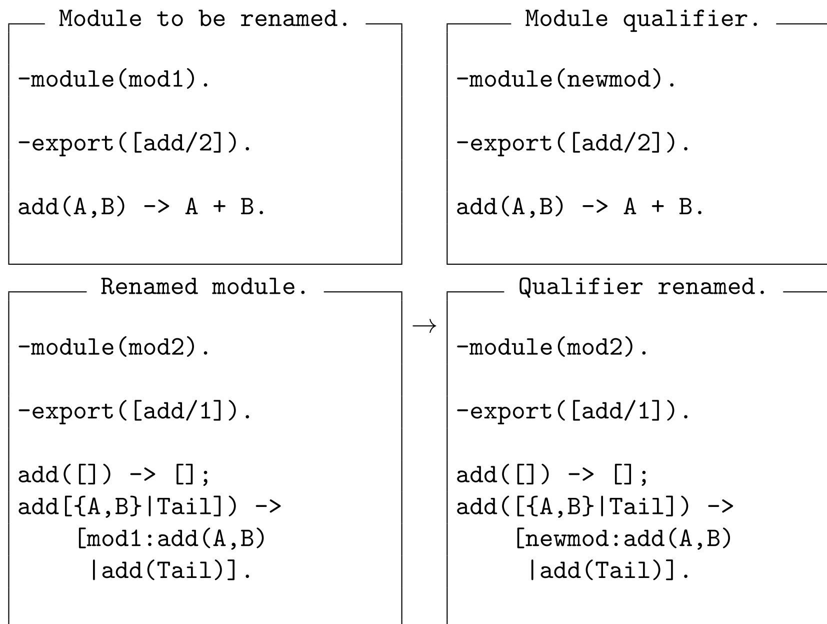


Figure 15: Renaming module `mod1` to `newmod`.

- Type the new module name.

8.4.2 Side conditions

- The given new name should be a legal file name.
- There must not exist another module with the given new name in the graph.
- There must not exist another file with the given new name in the directory of the module to be renamed.
- If one of the above conditions fails, the transformation starts an interaction to ask for a new module name.

8.4.3 Transformation steps and compensations

1. Rename the current module name to the new name.
2. Rename the collected module qualifiers to the given new name.
3. Rename the references to the module in the import lists.
4. Rename the file to the new name.

8.5 Rename record

This refactoring renames records in modules or header files. After the transformation, the old name will be replaced by the new name in the record definition and in every reference to the given record (e.g. record field access or field update expressions). The condition of the renaming is that there is no name conflict with another record in the file (which contains the given record), in its includes, or where it is included (the latter is only possible when we are renaming in a header file).

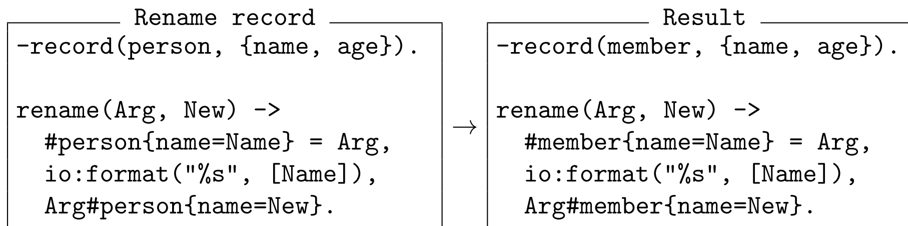


Figure 16: Renaming record “person” to “member”

8.5.1 Usage in Emacs

- Add the file that contains the record definition to the database (`C-c C-r a`).
- Position the cursor over the name in the record definition.
- Call the refactoring from the menu or with `C-c C-r x r r d`.
- Type the new record name.

8.5.2 Side conditions

- There must be no record with the new name
  - in the file that contains the record,
  - in files which are included by this file,
  - in files which include this file.
- If one of the above conditions fails, the transformation starts an interaction to ask for a new record name.
- If the user does not specify a record, the transformation starts an interaction to ask the user to specify a record.

8.5.3 Transformation steps and compensations

1. The record name is changed to the new name in the definition of the record and in every record expression that refers the record.

8.6 Rename record field

The rename record field transformation supports renaming the specified field of a record. The name of the field is replaced in the definition form and in every referrer expression and token to the given new name. To the field belongs a semantic field object linked to the record, it stores the field name. The data of this object is updated too.

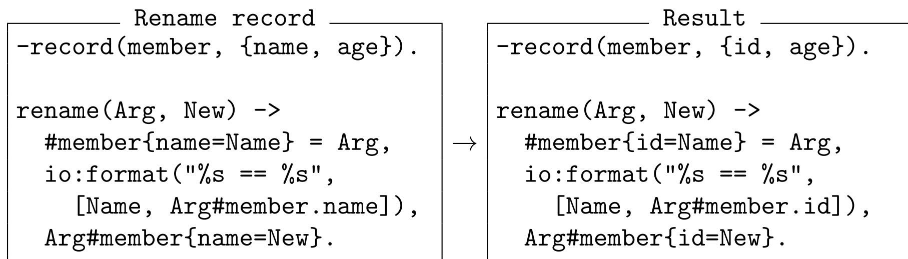


Figure 17: Renaming field `name` to `id`

8.6.1 Usage in Emacs

- Add the file that contains the record definition to the database (`C-c C-r a`).
- Position the cursor over the field name in the record definition.
- Call the refactoring from the menu or with `C-c C-r x r f`.
- Type the new field name.

8.6.2 Side conditions

- The record must have no field with the same name as the given new field name. If it has, the transformation starts an interaction to ask for a new record field name.
- If the user does not specify a record field, then the transformation starts an interaction to ask the user to specify one.

8.6.3 Transformation steps and compensations

1. The field name is changed to the new name in the definition of the record and in every record expression that refers the field.

8.7 Rename variable

The "rename variable" transformation renames a variable and all of its occurrences. The only semantic information it requires is the scope and visibility of the variables.

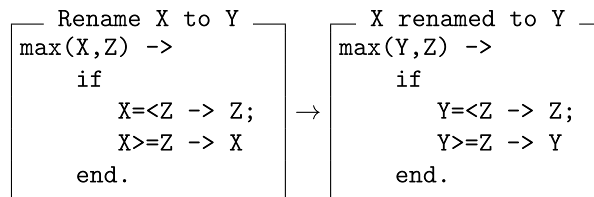


Figure 18: Renaming variable `X` to `Y`

8.7.1 Usage in Emacs

- Add the file that contains the variable to the database (`C-c C-r a`).
- Position the cursor over any instance of the variable.
- Call the refactoring from the menu or with `C-c C-r x r v`.
- Type the new variable name.

8.7.2 Side conditions

- The new variable name does not exist in the scope of the variable, either as a defined variable or as a visible variable. If it exists then the transformation starts an interaction to ask for a new variable name.
- If the user does not specify a variable, the transformation starts an interaction to ask for a variable. It gives a list of variables which can be reached from the selected function clause.

8.7.3 Transformation steps and compensations

1. Replace every occurrence of the variable with the new name. In case of variable shadowing, other variables with the same name are not modified.

8.8 Move function

Modules contain functions which consist of one or more clauses. Moving functions between modules is a possible refactoring step. This transformation must move each clauses of the given functions from the source module to the target module and have to compensate every reference of functions and of entities used by the functions.

The move function refactoring is useful in module clustering, there it is needed to split the library modules into parts that are determined by the clusters.

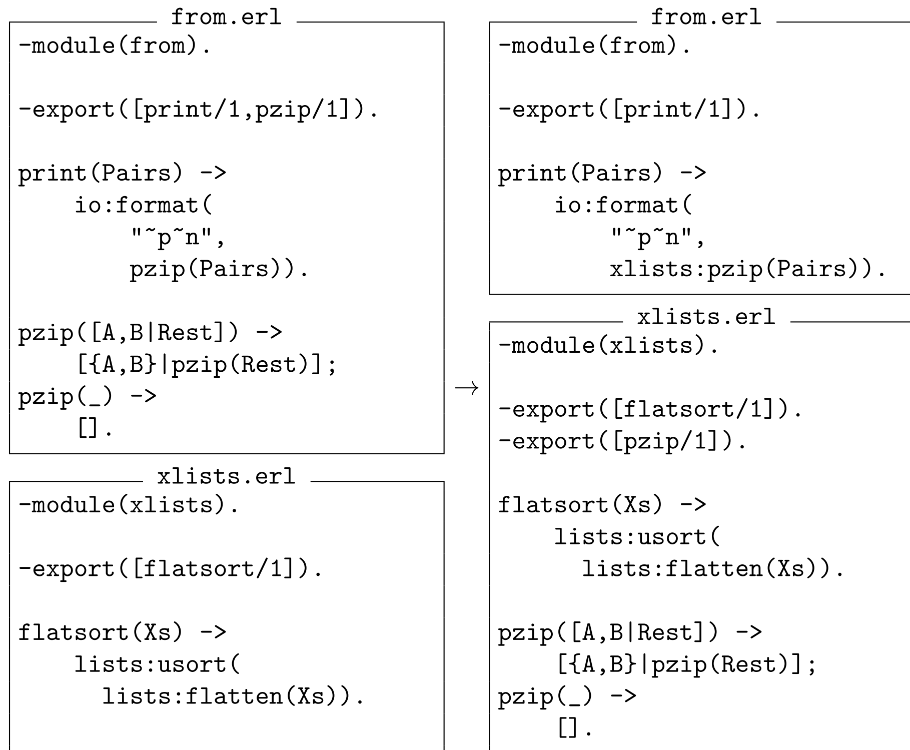


Figure 19: Move function example

8.8.1 Usage in Emacs

- Add the source and the destination file to the database (see section 7.3.3)
- Call the refactoring from the menu or with `C-c C-r m f`
- Fill out the form that pops up
  - Type the name of the target module
  - Select the functions to be moved
- Start the transformation with the "Move" button

8.8.2 Side conditions

- The names of the selected functions should not conflict with other functions in the target module, neither with those imported from another

module (overloading). Furthermore, the name should be a legal function name in all modules.

- If the user do not select functions to be moved, the transformation starts an interaction. The tool gives a checkbox list to the user to select the functions to be moved.
- Macro name conflicts must not occur in the target module, that is, macro names used in the functions must refer to the same macro definition in the source and in the target module. This applies to macros used in these macros too.
- Record name conflicts must not occur in the target module, that is, record names used in the functions must refer to the same record definition in the source and in the target module.

8.8.3 Transformation steps and compensations

1. In the refactoring step the functions to be moved have to be marked either at the definition or in the export list. A list has to be created from the function name and arity pairs. Duplicity should be avoided and only real function names and arities should occur in the list.
2. The new place of the functions, or the target module, has to be asked from the user. If there is no such module in the tool database, it has to be loaded.
3. If the transformation does not disobey the rules, the functions have to be deleted from their original places together with all their clauses.
4. The moved functions have to be placed to the end of the new module.
5. Functions have to be deleted if they appear in the export list of the original module. (If they were exported, they have to be exported in their new place, too.)
6. The functions, which are called in the moved function but remain in the original module, have to be put in an export list in the original module.
7. If the functions to be moved are called in other functions from the original module, they have to be exported in the new module and the calls in the original module have to be changed to qualified calls.
8. If the moved functions are referred to by qualified names, the module names have to be changed to the new module name.
9. After the transformation the module names in the import lists have to be changed to the name of the target module.
10. The moved function in the target module has to be deleted from the import list.

11. Records and macros used in the moved function have to be made visible in the target module, either by moving them into header files (or including the header file if the definition is already in one), or copying their definition.

8.9 Move macro

This transformation moves macro definitions between two files. Source and target files can be either modules or header files, the conditions are slightly different in every case. The goal of the transformation is to make the moved macro definitions available in every place where they are used.

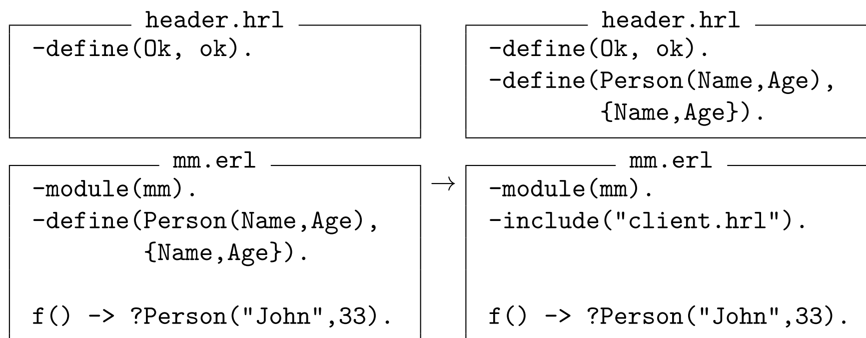


Figure 20: Move macro example

8.9.1 Usage in Emacs

- Add the source and the destination file to the database (see section 7.3.3)
- Call the refactoring from the menu or with `C-c C-r m m`
- Fill out the form that pops up
  - Type the name of the target module
  - Select the macros to be moved
- Start the transformation with the "Move" button

8.9.2 Side conditions

- The names of the macros to be moved must not clash with existing macro names in none of:
  - the target file,
  - the target's included files
  - files where the target is included

- If the user does not specify the macros to be moved, the transformation starts an interaction to ask the user to specify macros. The user has to select the macros to be moved from a checkbox list.
- Moving macros from a header to a module is only allowed if there exist no other module that both includes the header and uses some of the macros to be moved.
- An include form can only by introduced when it does not cause inconsistency at the place of inclusion.

8.9.3 Transformation steps and compensations

1. The macro definitions are removed from the source file.
2. If the target is a header file that does not exist, the file is created.
3. The macro definitions are placed at the end of the target header file, or before the first function of the target module file.
4. If a macro is moved into a header file, then every module that uses the record is changed to include the target header file. This is not an issue when the target is a module file.

8.10 Move record

This transformation moves record definitions between two files. Source and target files can be either modules or header files, the conditions are slightly different in every case. The goal of the transformation is to make the moved record definitions available in every place where they are used.

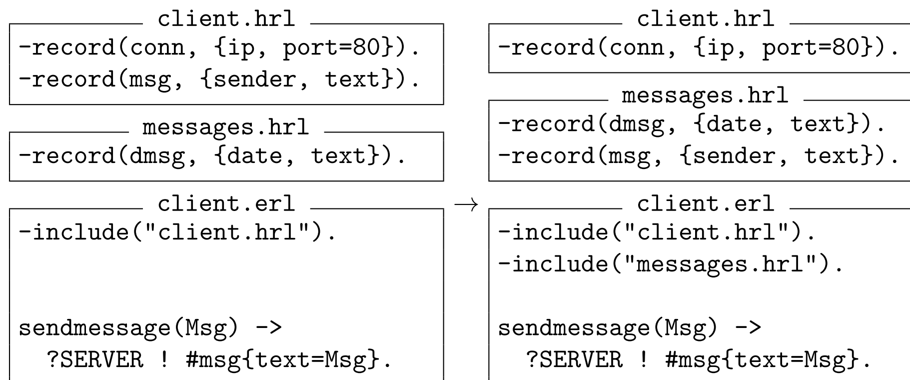


Figure 21: Move record example

8.10.1 Usage in Emacs

- Add the source and the destination file to the database (see section 7.3.3)
- Call the refactoring from the menu or with `C-c C-r m r`
- Fill out the form that pops up
  - Type the name of the target module
  - Select the records to be moved
- Start the transformation with the "Move" button

8.10.2 Side conditions

- The names of the records to be moved do not clash with existing record names
  - in the target file,
  - in files that are included in target and
  - in files where the target file is included.
- If the user does not specify the records to be moved, the transformation starts an interaction to ask the user to specify records. The user has to select the records to be moved from a checkbox list.
- Moving records from a header file to a module file is only permitted if no other modules include the header file and use some of the records to be moved.
- If a file inclusion has to be introduced during the transformation, this inclusion must not cause inconsistency at the place of the inclusion.

8.10.3 Transformation steps and compensations

1. The record definitions are removed from the source file.
2. If the target is a header file that does not exist, the file is created.
3. The record definitions are placed at the end of the target header file, or before the first function of the target module file.
4. If a record is moved into a header file, then every module that uses the record is changed to include the target header file. This is not an issue when the target is a module file.

8.11 Eliminate function call

The eliminate function call refactoring step substitutes the selected application with the corresponding function body and executes compensations. The function may consist of one or more function clauses and may have guard expression(s), inline can handle these cases.

In the example in Fig. 22 the `sort/1` application is inlined. The application is replaced with a case expression.

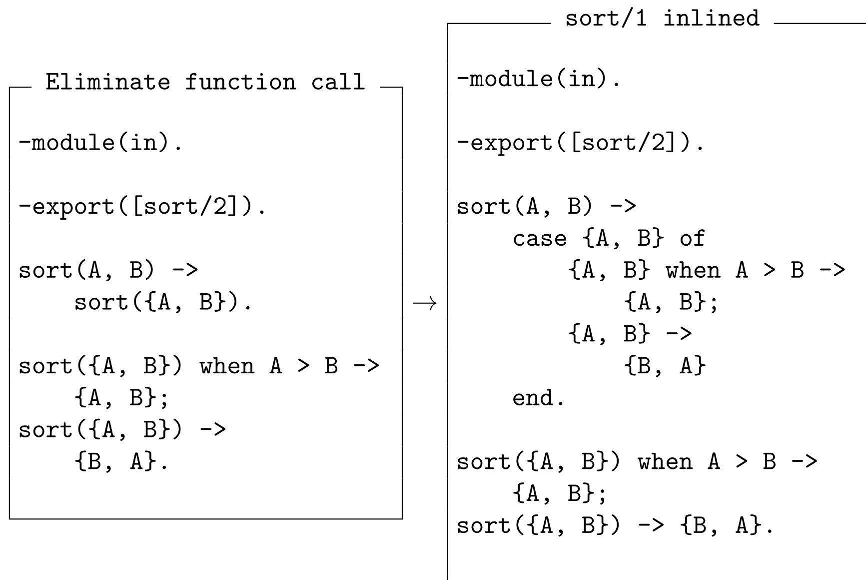


Figure 22: Eliminate function application of `sort/1`.

8.11.1 Usage in Emacs

- Add the file that contains the function application to be inlined to the database (`C-c C-r a`).
- Position the cursor over a function application.
- Call the refactoring from the menu or with `C-c C-r e f`.

8.11.2 Side conditions

- Applying the inline function must not cause variable name conflicts. A variable name conflict arises when the same variable name is used in the body of the function clause and in the scope of the selected function application, except the variables which are bound in the formal parameters where the structure of the formal and the actual parameters are equivalent.

- If the function is defined in other module:
  - the function do not contain local (not exported) function applications in its body.
  - macro name conflicts must not occur in the current module, that is, macro names used in the functions must refer to the same macro definition in the current and in the definition module. This applies to macros used in these macros too.
  - record name conflicts must not occur in the current module, that is, record names used in the functions must refer to the same record definition in the current and in the definition module.
- If the user does not specify a function application or the specified function does not exist, the transformation starts an interaction to ask the user to specify one. The user has to select a function from a radio group.

8.11.3 Transformation steps and compensations

1. Find the corresponding function definition and copy the function clause(s) and create (if it is needed) a corresponding structure from the expressions of the body(ies), the guard expressions (if there is any) and from the patterns of the function clause(s).
2. Where the actual and formal parameters are structurally equivalent, create variable name pairs and rename the corresponding variables in the copied body.
3. Where the formal and structural parameters are not equivalent, create a match expression from these parameters. The left hand side is a tuple from the formal parameters, and the right hand side is a tuple from the actual parameters.
4. If the function consists of
   - one clause and does not have guard expression and the body of the function contains only one expression and match expressions should not be created from the parameters, replace the application with this single expression.
   - one clause and does not have guard expression and the body of the function contains more than one expression and the parent expression of the selected application is a clause, replace the application with the sequence of the expressions from the body of the function clause extended with the created match expression.
   - one clause and does not have guard expression and the body of the function contains more than one expression and the selected application is a subexpression:

    - create a begin-end block from the sequence of the expressions from the body of the function clause extended with the created match expression
    - replace the application with this begin-end block.
  - more than one clause, or it has guard expressions (or both) or variables appear multiple times with the same name in a pattern list:
    - create a case expression from the function clause body(ies) expression(s), guard and pattern expressions.
    - replace the application with this case expression.

5. If the definition of the function is in another module
  - qualify the applications in the copied body which call exported functions from the defining module.
  - qualify the applications in the copied body which call imported functions.
  - copy or import record and macro definitions to the current module which are used in the copied body(ies).

8.12 Eliminate macro substitution

The eliminate macro substitution refactoring step substitutes a selected macro application with the corresponding macro body and takes care of necessary compensations.

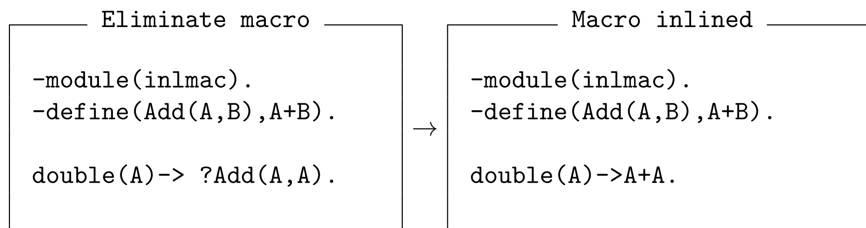


Figure 23: Eliminate macro application `?Add(A,A)`.

8.12.1 Usage in Emacs

- Add the source file to the database (`C-c C-r a`).
- Position the cursor over a macro application.
- Call the refactoring from the menu or with `C-c C-r e m`.

8.12.2 Side conditions

- The selected macro must not contain stringification or another macro in its definition.
- The selection must not be inside a macro definition.
- If the selection is not specify a macro usage the transformation starts an interaction to let the user specify one. It gives a list with the possible macro usages.

8.12.3 Transformation steps and compensations

1. All the side conditions must met.
2. The following must be gathered that are necessary to fully finish the transformation:
    - Edges to be deleted: those between 'subst' and both its children and the macro definition; ones connecting all intermediate 'token' nodes with 'subst', the containing expression and the original lexical tokens; and the one between the 'subst' node and its arguments.
    - Nodes to be deleted: those lexical children of the 'subst' nodes parameter child which do not participate in the final solution (commas), children of the 'subst' node, the 'subst' node itself and the intermediate 'token' nodes.
    - Edges to be rewired: those edges that originally connected a non-shared lexical node created specifically for the given substitution.
    - Nodes to be created: those nodes which must be cloned from the macro definition because they are shared between all usages.

8.13 Eliminate fun expression

The "eliminate fun expression" transformation expands an implicit fun expression into an explicit one. This transformation can be done separately, but sometimes it is needed by other transformations as a compensation step. One such example is the module-qualifier correction in the "move function" refactoring. When an implicit fun expression need module qualifying, the result will be like this: `fun module:function/2`. This form is not supported in older Erlang versions. But when we expand the fun expression, a simple function application appear instead of the short form (`fun function/2` equals to `fun(V1, V2) -> function(V1, V2) end`, and it will become able to module-qualifying.

When we want to modify the function application or just don't want to use this syntactical sugar (the implicit form of fun expression), we can use this transformation to expand the expression.

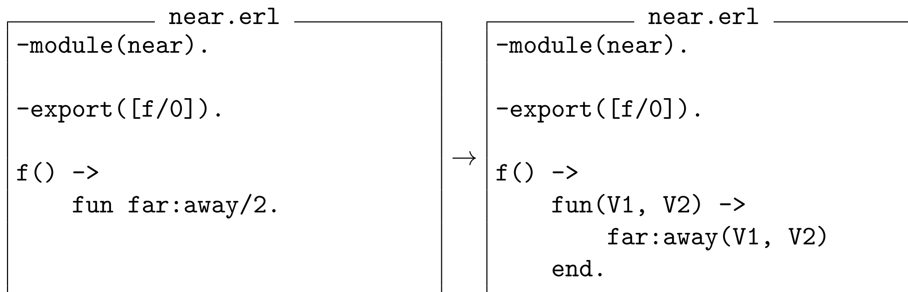


Figure 24: Eliminating the fun expression `fun far:away/2`

8.13.1 Usage in Emacs

- Add the file that contains the function application to be expanded to the database (`C-c C-r a`).
- Position the cursor over the name of the function to be expanded.
- Call the refactoring from the menu or with `C-c C-r e u`.

8.13.2 Side conditions

- The selected expression should be an implicit fun expression or part/subexpression of an implicit fun expression

8.13.3 Transformation steps and compensations

1. If the implicit fun expression is found, the new syntax structure is created and the old expression is replaced with the new one.

8.14 Eliminate variable

In this refactoring, all instances of a variable are replaced with its bound value. Those instances of the variable where the value of the instance is not used can be dropped.

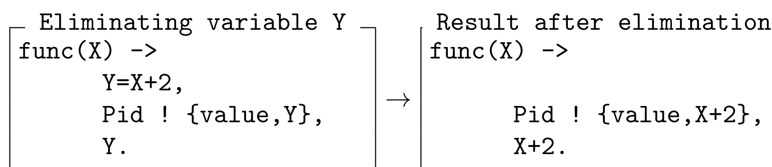


Figure 25: Replacing all instances of variable Y with its value.

8.14.1 Usage in Emacs

- Add the file that contains the variable to be eliminated to the database (`C-c C-r a`).
- Position the cursor over any instance of the variable.
- Call the refactoring from the menu or with `C-c C-r e v`

8.14.2 Side conditions

- The variable has exactly one binding occurrence on the left hand side of a pattern matching expression, and not a part of a compound pattern.
- The expression bound to the variable has no side effects.
- Every variable of the expression is visible (that is, not shadowed) at every occurrence of the variable to be eliminated.
- If the selection is not specify a variable but it is inside a function clause, the tool gives a list to the user to select a variable. The list contains the reachable variables in the given function clause.

8.14.3 Transformation steps and compensations

1. Every occurrence of the variable is substituted with the expression bound to it at its binding occurrence, with parentheses around the.
2. If the result of the match expression that binds the variable is discarded, the whole match expression is removed. Otherwise, the match expression is replaced with its right hand side.

8.15 Introduce function

A function definition might contain an expression or a sequence of expressions which can be considered as a logical unit, hence a function definition can be created from it. The extracted function is lifted to the module level, and it is parametrized with the free variables of the original expressions: those variables which are bound outside of the expressions, but the value of which is used by the expressions.

8.15.1 Usage in Emacs

- Add the file that contains the function to be extracted to the database (`C-c C-r a`).
- Mark the exact part of the source you want to extract.
- Call the refactoring from the menu or with `C-c C-r i f`.
- Type the new function name.

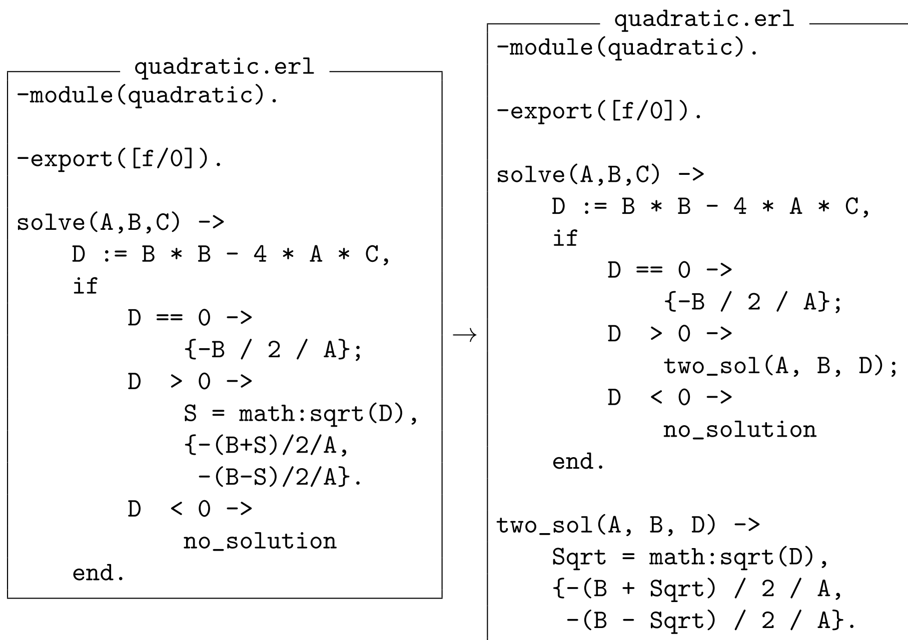


Figure 26: Introducing the new function `two_sol`

8.15.2 Side conditions

- The name of the new function should not conflict with another function, either defined in the same module, or imported from another module (overloading). Furthermore, the name should be a legal function name.
- If one of the above conditions fails, the transformation starts an interaction to ask for a new function name.
- The starting and ending positions should delimit a sequence of expressions.
- Variables with possible binding occurrences in the selected sequence of expressions should not appear outside of the sequence of expressions.
- The extracted sequence of expressions cannot be part of a guard sequence.
- The extracted sequence of expressions cannot be part of a pattern.
- The extracted sequence of expressions cannot be part of macro definition.

8.15.3 Transformation steps and compensations

1. Collect all variables that the selected sequence of expressions depends on.
2. Collect variables from the selected variables in step 1, which has binding occurrence out of the selected part of the module.

3. Add a new function definition to the current module with a single alternative. The name of the function is an argument to the refactoring. The formal parameter list consists of the variables selected in step 2.
4. Replace the selected sequence of expressions with a function call expression, where the name of the function is given as an argument to the refactoring, and the actual parameter list consists of the variables selected in step 2.
5. The order of the variables must be the same in steps 3 and 4.
6. If the selected expression is a block-expression, eliminate the begin-end keywords from the expression in the body of the created new function.

8.16 Introduce import

This refactoring imports the functions of the selected module that are used in the current file and removes the module qualifiers from the function calls of this module.

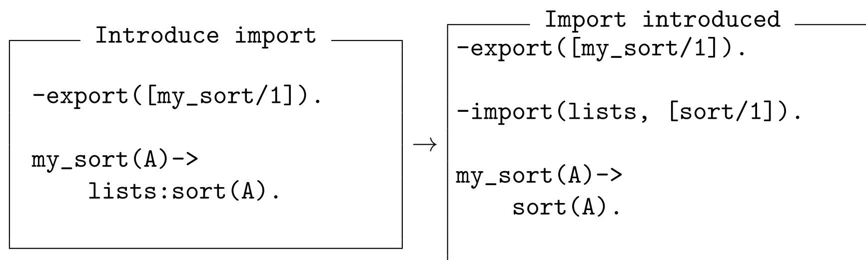


Figure 27: Import the `sort/1` function from the `lists` module.

8.16.1 Usage in Emacs

- Add the source file to the database (`C-c C-r a`).
- Select the module qualifier of a function application.
- Call the refactoring from the menu or with `C-c C-r i i`.

8.16.2 Side conditions

- No local function of the file has the same name and arity as the functions of the module that are used or imported in the file.
- No imported function in the file has the same name and arity as functions of the module that are used in the file.

- If there is a problem with the given module name which functions will be imported, the transformation asks for a new module. It gives a list to the user to specify a module. The list contains the modules from where the source module has already imported functions.

8.16.3 Transformation steps and compensations

1. In case there is no import list of the module in the file a new import list containing the functions of the module that are used in the file is added to the file.

2. In case there is only one import list of the module in the file the rest of the functions used in the file are added to this list.

3. In case there is more then one import list of the module, the contents of this list will be merged in one, and the rest of the functions used in the file are added to this list.

4. The module qualifiers of the module are removed from the corresponding functions.

8.17 Introduce function argument

This refactoring generalizes a function definition by selecting an expression (or an expression sequence), and makes this the value of a new parameter added to the definition of the function. The actual parameter at every call site becomes the selected part with the corresponding compensation.

The generalized function will not be exported from the module. If the original function is exported from the module, a new function definition with the original arity will be created which calls the generalized function with the corresponding arguments.

In the example in Fig. 28 a function for double the elements of a list is generalized by selecting the subexpression `2`. The subexpression is replaced with the variable `N` and the subexpression is added to the argument-list of function call in the body of `f(Z)` function.

8.17.1 Usage in Emacs

- Add the file that contains the function to the database (`C-c C-r a`).
- Select the expression along which the generalization should be done.
- Call the refactoring from the menu or with `C-c C-r i a`.
- Type the name for the new function argument.

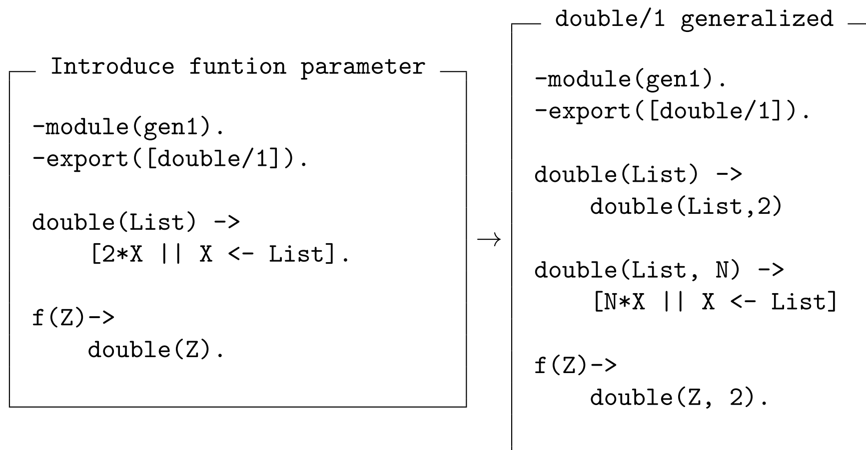


Figure 28: Introduce a new parameter to function `double/1`.

8.17.2 Side conditions

- The name of the function with its arity increased by one should not conflict with another function, either defined in the same module, imported from another module, or being an auto-imported built-in function.
- The new variable name does not exist in the scope of the selected expression(s) and must be a legal variable name. If the new variable name does not keep these conditions, the transformation starts an interaction to ask for a new variable name.
- The starting and ending positions should delimit an expression or a sequence of expressions.
- The selected expressions do not bind variables that are used outside the selection.
- Variable names bound by the expressions do not exist in the scopes of the generalized function calls.
- The expressions to generalize are not patterns and they do not call their containing function.
- If the selection is part of a list comprehension, it must be a single expression and must not be the generator of the comprehension.
- The extracted sequence of expressions are not part of a macro definition, and are not part of macro application parameters.

8.17.3 Transformation steps and compensations

1. If the selected expression does not contain any variables:
   - Give an extra argument (a simple variable) to the function. The name of this argument is the name given as a parameter of the transformation.
   - Replace the expression with a variable expression.
   - Add the selected expression to the argument list of every call of the function.
2. If the selected expression contains variable(s) or contains more than one expression or has a side-effect:
   - Add a new argument (a simple variable) to the function definition with the given variable name.
   - Replace the expression with an application. The name of the application is the given variable name, the arguments of the application are the contained variables.
   - Add a fun expression to the argument list of every call of the function. The parameters of the fun expression are the contained variables and the body is the selected expression.
3. If the selected expression is part of a guard expression:
   - Give an extra argument (a simple variable) to the function. The name of this argument is the name given as a parameter of the transformation.
   - Replace the guard expression with a variable expression with that new name.
   - Add the selected expression to the argument list of the function calls.
   - If the selected expression contain formal parameter, replace this with the actual parameter.
   - If the selected guard is a conjunction or a disjunction, create an `andalso` or an `orelse` expression and add this to the argument list of the function call.
4. If the generalized function is recursive, than instead of add the selection to the argument list of the function calls in the body, add the new parameter to the argument list.
5. If the generalized function is exported from the module, add a new function definition to the current module with the same parameters and guard. The body of this function is a call of the generalized function.
6. If the selection contains macro application/record expression move the definition of the macro/record before the first call of function.

8.18 Introduce record

Given a tuple skeleton, this transformation converts it to a record expression and inserts compensating record expressions or record update expression at the needed places.

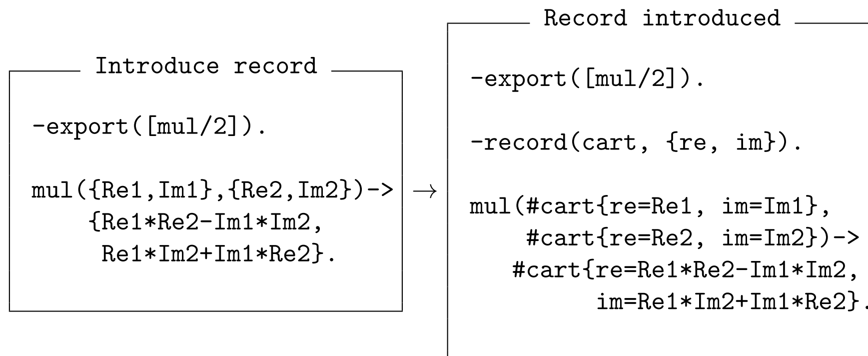


Figure 29: Introduce record `cart` with fields `re` and `im`.

8.18.1 Usage in Emacs

- Add the source file to the database (`C-c C-r a`).
- Select a tuple skeleton.
- Call the refactoring from the menu or with `C-c C-r i r`.
- Type the name of the record and the names of the fields.

8.18.2 Side conditions

- The name of the record we introduce should not conflict with another record. Furthermore, the name and field names should be a legal record name. If the name or the field name is not legal, the transformation starts an interaction to ask for a new name.
- The starting and ending positions should delimit a tuple skeleton.
- The transformed tuple cannot be embedded in a list comprehension, list or another tuple.
- The given field names number should match the number of the tuple's elements.
- The selected tuple cannot be a parameter of a function expression.
- If the selected tuple is a function parameter, there must not be an implicit reference to the function.

8.18.3 Transformation steps and compensations

1. The refactoring finds every tuple in the function pattern, which has the same type.
2. The transformation checks every function clause to find those, which parameter contains at least one same typed tuple.
3. The refactoring collects every function calls, which calls the collected function clause.
4. The refactoring finds all function calls in the collected function clauses, where the parameter contains at least one same typed tuple.
5. The transformation collects the return parameter, if it is a same typed tuple.
6. The refactoring finds every function calls in the collected function clauses, which parameter contains at least one same typed tuple. The transformation finds that function and the collection starts again from the first step.
7. If the record didn't exist before, its definition is constructed.
8. The collected tuples in the function patterns are replaced to record expressions. If a function clause contains function calls, the affected record gets bound with a variable name.
9. The return value is transformed to a record expression, if it was collected.
10. The unused variabled (in the record expression) are elliminated.
11. Those function calls parameters, which calls a collected function, are transformed to record expression.
12. If a collected function's return parameter is a same typed tuple, and the calling place is match expression, the left side is transformed to a record expression too.

8.19 Introduce tuple

In this transformation, consecutive arguments of a function are contracted into a tuple. This transformation addresses the formal parameter list of all the clauses in the function definition as well as the actual parameter list in each perceptible (viz. by static analysis) call of the function. The transformation affects more than one module if the function is exported.

The example in Fig. 30 illustrates the operation of the transformation on a function with a single clause. Both the definition of `step/2` and its application in `gcd/2` are changed.

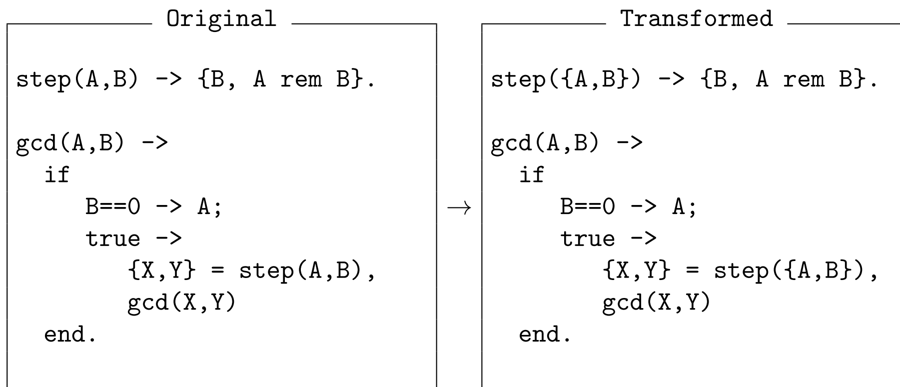


Figure 30: Tupling the two arguments of function `step`.

8.19.1 Usage in Emacs

- Add the file that contains the function whose arguments are to be tupled to the database (`C-c C-r a`).
- Mark the arguments to be tupled in the function definition.
- Call the refactoring from the menu or with `C-c C-r i t`.

8.19.2 Side conditions

- The function must be declared at the top level of a module, not a function expression.
- If the number of parameters that should be contracted into tuple is greater than one the arity of function will be changed. In this case the function with new arity should not conflict with other functions.
  - If the function is not exported, it should not conflict with other functions defined in the same module or imported from other modules.
  - If the function is exported, then besides the requirement above, for all modules where it is imported, it should not conflict with functions defined in those modules or imported by those modules.

8.19.3 Transformation steps and compensations

1. Change the formal parameter list in every clause of the function: contract the formal arguments into a tuple pattern from the first to the last argument that should be contracted.
2. If the function is exported from the module, then the export list has to be modified: the arity of the function is updated with new arity.

3. If the function is exported and another module imports it, then the arity must be adjusted in the corresponding import list of that module.

4. Implicit function references are turned into fun expressions that call the function, and the result is handled as any other function call.

5. For every application of the function, modify the actual parameter list by contracting the actual arguments into a tuple from the first to the last argument that should be contracted.

8.20 Introduce variable

During the introduce variable transformation, a new match expression is created that binds the selected expression to the variable that the user has given as input, and all instances of the expression is changed to the variable.

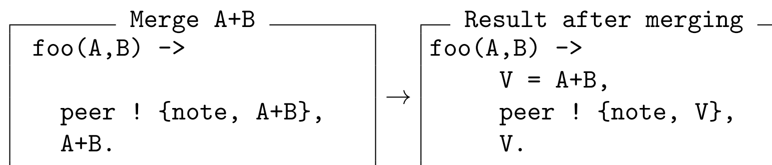


Figure 31: Example of Introduce variable transformation.

Merge expression duplicates refactoring is executed in the function (Figure 31), A+B expression in the Message-Passing (peer ! note,A,B) is stored in V variable, then all instances of the expression are changed to V variable.

8.20.1 Usage in Emacs

- Add the file that contains the expression to be merged to the database (`C-c C-r a`).
- Mark the expression whose duplicates are to be merged.
- Call the refactoring from the menu or with `C-c C-r i v`.
- Type the new variable name.

8.20.2 Side conditions

- The expression cannot be substituted if any of its sub-expressions have side effects.
- The transformation cannot be executed if the expression is in the head of a list comprehension, in a pattern or in a guard expression.
- If the expression occurs in a generator expression, it should not contain variables that are bound by generator patterns.

- The given variable name should not already exist in the given scope in order to avoid name clashes.
- If the given variable name is not legal then the transformation starts an interaction to ask for a new variable name.

8.20.3 Transformation steps and compensations

1. Determine the insertion point. The insertion point is the first possible location within a body where all of the variables of the expression have already received their binding. If the selected expression contains no variables, the insertion point may be in any containing scope; currently the innermost scope is chosen. If the selected expression contains at least one variable, the insertion point is in the outermost scope that contains all variables.
2. Insert a new match expression to the insertion point that binds the expression to the new variable. If there is an expression to be replaced at this position, the match expression should replace it.
3. Substitute all other instances of the expression to the new variable. Note that not all expressions whose structure is identical to the expression are instances of the expression: all variables have to have the same binding, and therefore the same scope. These substitutions should remove surrounding parentheses from instances.

8.21 Transform list comprehension

Turn `lists:map`, `lists:foreach` and `lists:filter` calls into list comprehension syntax, or do it backwards.

The two main cases of the transformation:

- `lists:map/2` or `lists:filter/2` or `lists:foreach/2` to list comprehension
  - The transformation is applied only if `lists:map/2` or `lists:filter/2` or `lists:foreach/2` is selected
  - If the first parameter is an explicit or an implicit fun expression the arity of the function must be equal to 2
  - If the first parameter is not a fun it is not checked what it really is
  - It is not checked wheter the second parameter really a list is
- list comprehension to `lists:map/2` and/or `lists:filter/2`
  - The transformation does not supports list comprehensions that contain more than one list generator
  - The result is a lists:filter/2 or a lists:map/2 or a composition of a lists:filter/2 and a lists:map/2

– The transformation does not optimise according to unused variables

– It is not checked wheter the generator really a list is

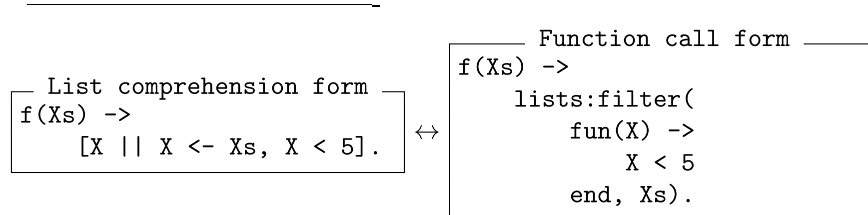


Figure 32: Transform between two equivalent forms of list filtering `im`.

8.21.1 Usage in Emacs

- Add the source file to the database (`C-c C-r a`).
- Select the construct to be transformed (a list comprehension or a call to `lists:map/2`, `lists:filter/2` or `lists:foreach/2`).
- Call the refactoring from the menu or with `C-c C-r t l`.

8.21.2 Side conditions

- `lists:map/2` or `lists:filter/2` or `lists:foreach/2` to list comprehension
  - The selection should contain an application of one of the relevant functions
  - If the first parameter is an implicit function or an explicit function, the arity has to be 1.
- list comprehension to `lists:map/2` and/or `lists:filter/2`
  - The list comprehension can have only one generator; transforming more complex comprehensions would likely complicate the code even more.

8.21.3 Transformation steps and compensations

- `lists:map/2` or `lists:foreach/2` to list comprehension
  - If the first parameter is an implicit function, a new variable is created in the list comprehension and the implicit function is called with this variable.
  - If the first parameter is an explicit function

    * If it has only one clause without pattern matching and guards
        - If it has only one body the body is copied into the head of the list comprehension.
        - If it has more then one body, a `begin..end` expression is created.
    * If it has more then one clause or one clause with pattern matching or with guards, a case expression is created in the head of the list comprehension. The application of the case is a new variable, and the branches of the case are the clauses of the function.
  - If the first parameter is anything else, an application is created for it in the head of the list comprehension.

- `lists:filter/2` to list comprehension
  - If the first parameter is an implicit function, a new variable is created in the list comprehension and the implicit function is called with this variable.
  - If the first parameter is an explicit function
    * If the function has one clause and the function has no pattern matching and no guards and the body returns a boolean value
        - If the function has only one body then it is inserted into the list comprehension as filter(s).
        - If the function has more then one body, then the bodies are inserted into the list comprehension in a begin..end structure.
    * If the function has two clauses and the second body has no pattern matching and no guards and returns constantly false
        - If the first clause has pattern matching, it is used in the generator.
        - If the first clause has guards, they are are inserted into the list comprehension as filters.
        - If the first clause has only one body which returns boolean value, it is inserted into the list comprehension as filter(s).
        - If the first clause has more then one body, then the bodies are inserted into the list comprehension in a begin..end structure.
    * If the function has more then two clauses it is transformed into a case structure.
  - If the first parameter is everything else it is inserted into the list comprehension as an application.

- list comprehension to `lists:map/2` and/or `lists:filter/2`
  - Building the `lists:filter/2`
    * The filters are inserted a function expressions body.

  * If the generator has pattern matching then the created function expression will have two clauses where the first clause has the pattern matching the second clause is constantly false.
  * If no filters and no pattern matching occurs then no `lists:filter/2` is created.
- Building the `lists:map/2`
  * If the head and the pattern of the list comprehension are equals, no `lists:map/2` is created.
  * Else a fun expression is created with the list comprehensions pattern and the list comprehensions head as body.

8.22 Reorder function parameters

The order of a function's arguments is a small, aesthetic aspect of a program. Swapping arguments can improve the readability of the program, and it can be used as a preparation for another refactoring, eg. to create a tuple from arguments that aren't next to each other.

The idea is illustrated by the following simple example (see Fig. 33), where a function's three arguments are reversed. To maintain the meaning of the program, every call of the function must be modified: the order of expressions that provide actual parameters must be reversed too.

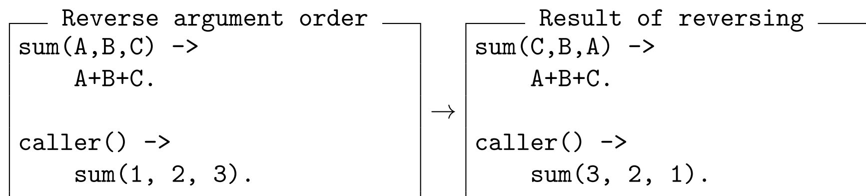


Figure 33: Simple argument reordering.

This refactoring can be carried out in almost every case without any problems, only dynamic function calls put limits to its applicability. Within the bounds of static analysis, every parameter reordering can be compensated at the place of function calls.

8.22.1 Usage in Emacs

- Add the file that contains the function to be reordered to the database (`C-c C-r a`).
- Position the cursor over the name of the function in any clause of the function definition.
- Call the refactoring from the menu or with `C-c C-r f o`.
- Type the new order of the function arguments.

8.22.2 Side conditions

- When a function application has an argument with side effects, the transformation may only be carried out after a warning that the order of side effects most probably will change, which may change the way the program works.
- When the given order is not legal, meaning it does not contain all of argument indices, the transformation starts an interaction to ask for a new order.

8.22.3 Transformation steps and compensations

1. Change the order of patterns in every clause's parameter list in the function according to the given new order.
2. For every static call of the function, change the order of the expressions that provide the actual parameters to the call according to the given order (obviously in all modules).
3. Every implicit function expression is expanded to the corresponding explicit expression which contains a static call to the function; this function call is then updated as described in the previous case.
4. For every call of the function that provides the arguments as a list, insert a compensating function expression that changes the order of the elements in the list according to the given new order.

8.23 Generate function specification

The "generate function specification" transformation calculates the specification of the selected function and insert the `-spec` directive in front of the function.

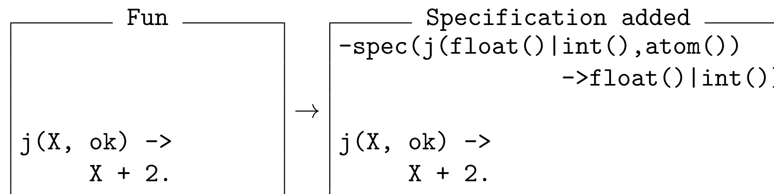


Figure 34: Inserting specification for function `f`

8.23.1 Usage in Emacs

- Add the file that contains the variable to the database (`C-c C-r a`).
- Position the cursor over the function definition..
- Call the refactoring from the menu or with `C-c C-r x g s`.

8.23.2 Side conditions

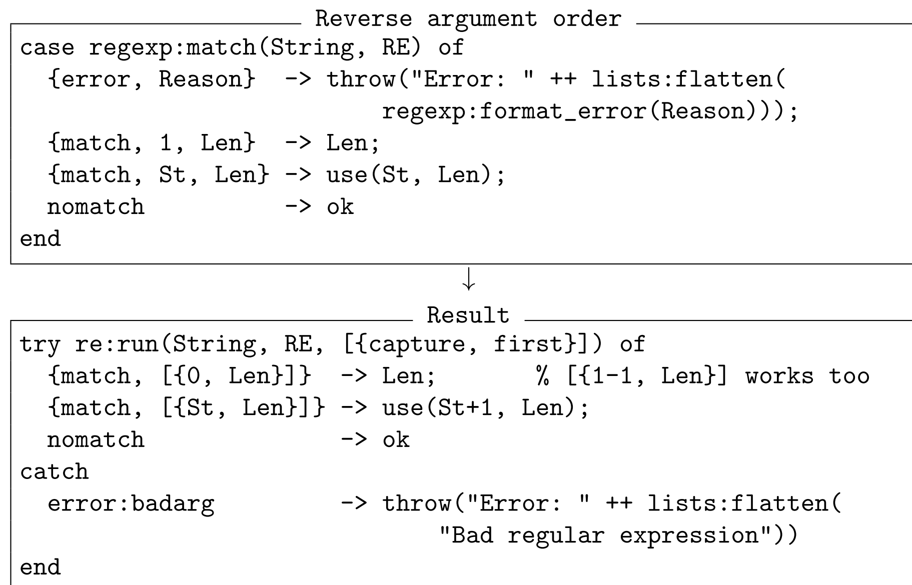
There is no side condition to use this transformation.

8.23.3 Transformation steps and compensations

1. Calculates the specification of the function.
2. Inserts the specification.

8.24 Upgrade interface: regexp→re

This transformation upgrades the calls of the old regular expression module `regexp` to the new `re` module. The transformation is based on a generic interface upgrade module, which is invoked with the right structural and semantic change descriptors.



Reverse argument order

```
case regexp:match(String, RE) of
  {error, Reason}  -> throw("Error: " ++ lists:flatten(
                          regexp:format_error(Reason)));
  {match, 1, Len}  -> Len;
  {match, St, Len} -> use(St, Len);
  nomatch          -> ok
end
```

↓

Result

```
try re:run(String, RE, [{capture, first}]) of
  {match, [{0, Len}]}  -> Len;      % [{1-1, Len}] works too
  {match, [{St, Len}]} -> use(St+1, Len);
  nomatch              -> ok
catch
  error:badarg         -> throw("Error: " ++ lists:flatten(
                              "Bad regular expression"))
end
```

Figure 35: An example of upgrading regexp interface.

8.24.1 Usage in Emacs

- Add the file that to the database (`C-c C-r a`).
- Call the refactoring from the menu or with `C-c C-r u i`.

8.24.2 Side conditions

- Patterns belong to applications to be updated should match at least one of the patterns written in the change descriptors.

8.24.3 Transformation steps and compensations

1. Updating the function application:
   - Modifying the called function's identifier
     `regexp:match ->re:run`
   - Matching the old function arguments and creating the new argument list
     `(String, RE) -> (String, RE, [capture, first])`
2. Finding the patterns to be updated and performing on them the matching and the replacing. Furthermore, applying the correction functions on the variable bindings or/and usages.
   Pattern: `match, 1, Len -> match, [0, Len]`
   Expression: `use(St, Len) -> use(St+1, Len)`
3. Turning the case expression to a try expression and moving error pattern to the catch block (`error:badarg` is moved to a catch block after we changed case expression to try expression).

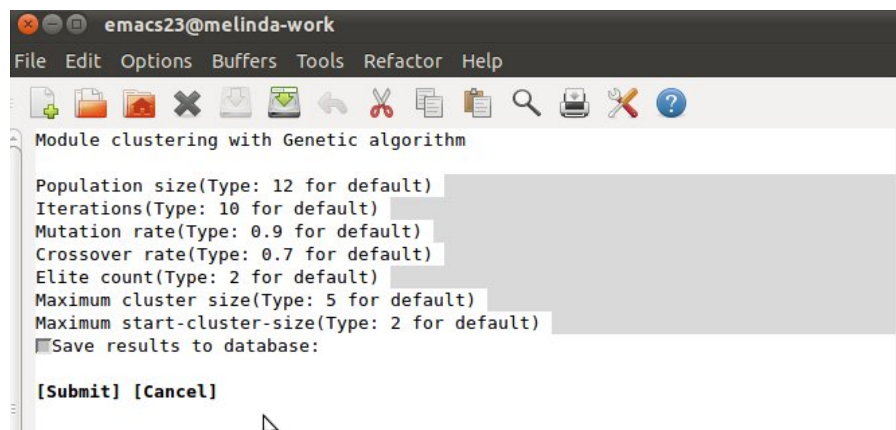
9 Clustering

9.1 The Emacs interface for clustering

There is a user interface to run clustering in Emacs. In order to use the Refactoring Tool for running clustering algorithms you should choose the type of clustering from the "RefactorErl Menu":

- Agglomerative module clustering
- Genetic module clustering
- Function clustering

After choosing the corresponding clustering type you have to fill out the parameter form and then press "Submit". The result of the clustering appears in a new buffer.



The following list contains the available options of the agglomerative algorithm: (These options can change dynamically)

- Transform function - (undefined or zero_one)
- Distance function - (call_sum or weight)
- Antigravity - (0.0 ..1.0)
- Merge function - (smart)

The list of the genetic algorithm options is shown below.

- Population size - number of chromosomes in an iteration (12)
- Iterations - number of iterations in the algorithm (10)
- Mutation rate - probability of mutation (0.9)

- Crossover rate - probability of crossovers (0.7)
- Elite count - number of chromosomes that survive the generation (2)
- Maximum cluster size - number of clusters allowed (5)
- Maximum start-cluster-size - number of clusters at startup (2)

If the storing option was checked, the result is saved. The mnesia table named cl_ui contains three lists. The first one is the list of the options of the algorithm. The second one is a list of the fitness numbers and the last one is a result of the clustering.

9.2 Console interface of clustering

You can use the clustering algorithm from `ri` by calling the `ri:cluster/0` function. You have to choose between agglomerative a genetic clustering at first and then between function and module clustering. Based on your choice `ri` will ask all of the parameters necessary for the clustering

```
ri:cluster()

Please choose an item from the list (blank to abort).

Please select an algorithm for clustering:
 1. Agglomerative
 2. Genetic
 type the index of your choice: 1
Please choose an item from the list (blank to abort).

Please select an entity type for clustering:
 1. function
 2. module
 type the index of your choice: 2
Please answer the following questions (blank to abort).
 Module clustering with Agglomerative algorithm
 Modules to skip(Type: none for default) none
 Functions to skip(Type: none for default) none
 Transform function(Select: [none,zero_one]) zero_one
 Distance function(Select: [call_sum,weight]) weight
 Antigravity(Type: 0.5 for default) 0.5
 Merge Function(Type: smart for default) smart
 Save results to database: (y/n) -> n
```

The result is:

```
See the direct information feed below:
 Clustering results:
 [[erl_syntax_lib,erl_syntax,igor,erl_tidy,epp_dodger,erl_recomment,
  erl_prettypr,prettypr,erl_comment_scan]]
 [[erl_syntax_lib,erl_syntax,igor,erl_tidy,epp_dodger,erl_recomment,
  erl_prettypr,prettypr],
 ...
 Fitness Numbers:
 [1.0,0.9473684210526315,0.918918918918919,0.8823529411764706,
 0.6896551724137931,0.5384615384615384,0.45,0.25]
```

10 Using RefactorErl in VIM

This section introduces the Vim plugin for using RefactorErl.

The command interface provides the use of `ReactorErl` in `VIM`.

In `VIM`, a `VIM plugin` has to be made to create a user interface which contains the menus of `RefactorErl` as well as those commands, with the help of which the refactorings can be run and the files loaded in the system can be handled.

10.1 Installation

To install the plugin, you can copy it to the vim plugin directory. For global installation, the plugin directory is usually located in `/usr/share/vim/vim`*Version*`/plugin`, for local install copy the plugin to *HOME*`/.vim/plugin` directory.

Then you should modify the `refpath` variable in the script to the installation directory of RefactorErl.

Since VIM interface is based on CLI it should be properly configured. For proper configuration of CLI, see the corresponding section.

Since the VIM interface does not cover all features of the tool, it is a good idea to use another interface, e.g. the Erlang shell interface (`ri`). After starting the refactorerl shell using the `bin/referl` command, the different interfaces can connect to the shell and all of them are usable simultaneously. But once the tool is started within VIM or from CLI you will be unable to connect to the tool with `ri`.

10.2 Menu and command structure

VIM can be used both in command line and in graphical environment. The VIM plugin of RefactorErl supports both of them. In the graphical front-end (Fig. 36) every command can be reached from the RefactorErl menu. The features are grouped to be easily found. These groups and the most important tasks can be found in Table 4, and the second column of the table shows the corresponding command to be used in non-graphical mode.

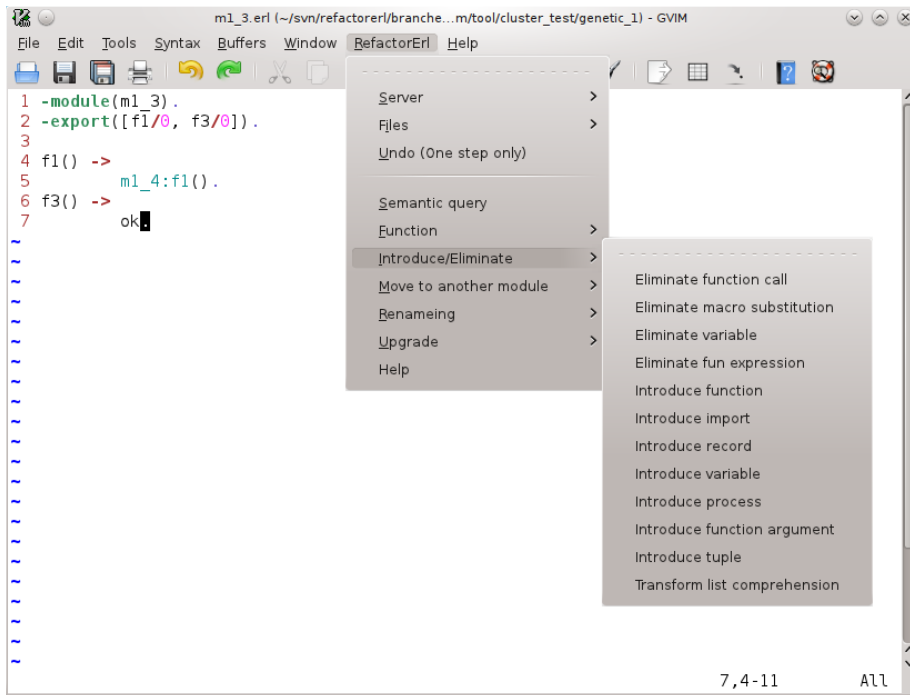


Figure 36: The GVim front-end of the tool

Menu item	Command
Server	—
Files	—
Undo (One step only)	`Refundo`
Semantic query	`Refsq`
Function	—
Introduce/Eliminate	—
Move to another module	—
Renaming	—
Upgrade	—
Help	`Refh`

Table 4: The top level groups and menu items in GVim menu.

Server submenu The server can be started and stopped by the menus and commands shown in Table 5. If the RefactorErl cannot be started an error message is shown in command buffer.

Menu item	Command
Server	—
→ Start	`RefSta`
→ Stop	`RefSto`

Table 5: Menu items in Server submenu in GVim.

Files submenu Features dealing with files can be found in Table 6. *Draw graph* makes a graphical representation of the database by creating a *dot* file.

Menu item	Command
Files	—
→ Add file	`Refa`
→ Drop file	`Refd`
→ Show database	`Refls`
→ Draw Graph	`Refgraph`

Table 6: Menu items in Files submenu in GVim.

Function submenu This submenu contains only the *Reorder function parameter* refactoring, this menu group has been created for future use. Menu item and the command can be found in Table 7.

Menu item	Command
Function	—
→ Reorder function parameter	`Refof`

Table 7: Menu items in Function submenu in GVim.

Introduce/Eliminate submenu In this submenu (shown in Table 8.) the refactorings concerning introduction and elimination of certain language constructs can be found.

Move to another module submenu In this menu item the refactorings concerning moving can be found. These are shown in Table 9. The modules changed by the refactoring are not opened.

Upgrade submenu The upgrade regexp refactoring can be found in the Upgrade submenu (shown in Table 10). This menu has been created for future use, too.

Menu item	Command
Introduce/Eliminate	—
→ Eliminate function call	`Refef`
→ Eliminate macro substitution	`Refem`
→ Eliminate variable	`Refev`
→ Eliminate fun expression	`Refeu`
→ Introduce function	`Refif`
→ Introduce import	`Refii`
→ Introduce record	`Refir`
→ Introduce variable	`Refiv`
→ Introduce process	`Reffp`
→ Introduce function argument	`Refia`
→ Introduce tuple	`Reftf`
→ Transform list comprehension	`Reftl`

Table 8: Menu items in Introduce/Eliminate submenu in GVim.

Menu item	Command
Move to another module	—
→ Move function	`Refmf`
→ Move macro	`Refmm`
→ Move record	`Refmr`

Table 9: Menu items in Move to another module submenu in GVim.

Menu item	Command
Upgrade	—
→ Upgrade regexp interface	`Refuir`

Table 10: Menu items in Upgrade submenu in GVim.

Renaming submenu The universal renamer and the type specific renamer refactorings are shown in Table 11.

10.3 Position based refactorings

Refactorings need either a single position or a range as a subject. In VIM, a range can be selected only in *Visual mode*, but you cannot execute a single statement while *Visual mode* is active, so you have to save the boundaries of the selection by pressing F2. Then you can run the range based refactorings. Position based refactorings are not affected by any saved ranges, they determine the actual cursor position every time a refactoring is started.

Menu item	Command
Renaming	—
→ Universal renamer	`Refr`
→ Rename function	`Refrf`
→ Rename header	`Refrh`
→ Rename macro	`Refrc`
→ Rename module	`Refrm`
→ Rename record	`Refrrd`
→ Rename record field	`Refrrf`
→ Rename variable	`Refrv`

Table 11: Menu items in Renaming submenu in GVim.

10.4 Interaction

If any of the given argument is missing or incorrect the interface provides user interaction to collect required data. To cancel an ongoing interaction you should use a blank answer.

10.5 Semantic queries

Semantic queries are fully supported in VIM interface (shown in Table 4). VIM does not support links to different sources, but the path and the location are listed after all entity in the result. This filepath can be copied and pasted after an open command (`:e` *`+line filepath`*).

11 Using RefactorErl in Eclipse

RefactorErl is accessible through several user interfaces, including a collection of shell commands as well as editor integrations. As we intend to support every popular development environment present for Erlang, we are going to cover not only Emacs and XEmacs, but Eclipse as well.

ErlIDE is an Eclipse-based development environment, which aims to assist programming of large scale Erlang applications. In order to integrate RefactorErl into Eclipse, we are engaged in creating an Eclipse plugin based on ErlIDE, which can enable access to all the functionality provided by RefactorErl, whilst utilising ErlIDE features as well. A prototype of the Eclipse plugin is already available.

11.1 Installation

11.1.1 Software requirements

- The current version of **RefactorErl** is available at `http://plc.inf.elte.hu/erlang/dl/`.
- The plugin is being developed and tested in **Eclipse 3.5.2**. (It likely works with newer versions as well.)
- This user interface is built upon **ErlIDE**, therefore it is required to have it with Eclipse. Inasmuch as the main structure of ErlIDE has been changed in its version `v0.11.6.201107010651`, it is necessary to get the latest version from `http://erlide.org/`.
- Finally, you can download **the plugin** itself from the RefactorErl homepage.

11.1.2 Deployment

To install this plugin you need to download the source code from the home page of the RefactorErl (`http://plc.inf.elte.hu/erlang/dl`). In order to generate the binaryes of the plugin you can compile it using `ant` or the compiler of Eclipse. Finally you have to copy the generated jar files into your Eclipse plugin directory. In Eclipse 3.5.x:

```
<user home>/.eclipse/org.eclipse.platform_3.5.0_155965261/plugin
```

11.1.3 Compiling the plugin in Eclipse

1. Select the location of the extension's source code as your current workspace.
2. Import the source code directories as a project.
   (a) In Package explorer tab (default on the left side) click the *Import* button from the popup menu.

   (b) Select *General/Existing Projects into workspace.*
   (c) You need to select your workspace's root.
   (d) If everything is correct you will see an `refactorerl.ui.core` entry beneath *Projects* label in the list.

3. Open `plugin.xml` file from the Package explorer tab. After click the *Overview* button.

4. Click the *Export wizard* button. A new window pops up, where the output directory has to be adjusted.

5. The generated `jar` has to be copied to the plugin directory (see: *Deployment* section). Finally (re)start Eclipse development environment.

11.1.4 Compile plugin using Ant

The source package contains a `build.xml` file, where these 2 properties have to be adjusted:

eclipse_home: the installation directory of Eclipse (default in Debian based systems: `/usr/lib/eclipse`).

eclipse_plugin: the user's local Eclipse configuration directory, where the installed plugins are too. (In Eclipse 3.5.x:
/home/<user>/.eclipse/org.eclipse.platform_3.5.0_155965261).

If the plugin is compiled using ant, the RefactorErl's Eclipse plugin immediately can be used after (re)started the development environment. To do that, just execute the beneath command:

```
ant jar
```

11.1.5 Configuration

The plugin, in order to be able to access RefactorErl, has to be set up before the first usage. For the configuration dialog, go to *Window/Preferences/RefactorErl* (Figure 37).

Start background process automatically If set, the plugin launches a new instance of the tool, otherwise an already running RefactorErl process will be used.

RefactorErl directory The installation directory of the tool.

Working directory A temporary directory, used to store the configuration files of RefactorErl. It is assumed to be a writable directory.

Waiting time This parameter represents the waiting time (in seconds) after starting RefactorErl process (has to be set to greater than 1). The slower access is possible to Erlang nodes, the greater value is suggested.

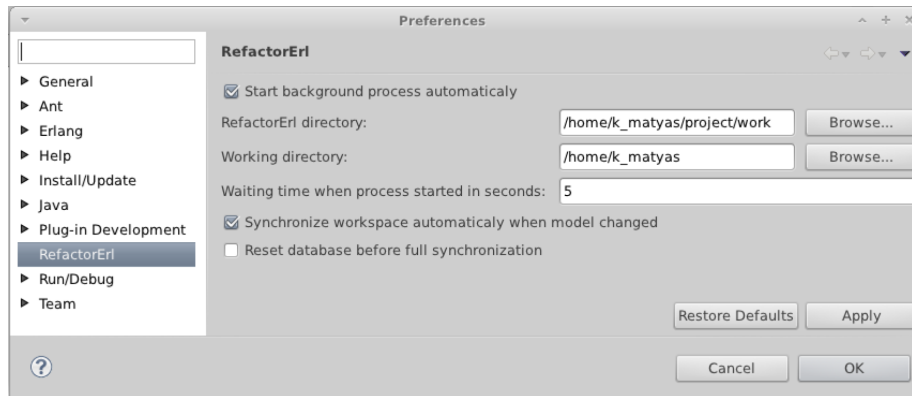


Figure 37: Preferences window

Synchronize workspace automatically then model changed If a source file has been changed, the extension automatically updates its inner representation within the tool.

Reset database before full synchronization Resets the database and erases its contents in the case of a full synchronisation. The use of this feature is recommended for RefactorErl developers only.

11.2 Database management

The Eclipse extension supports every database management functionality that is available in the RefactorErl tool. These can be executed from the top menu:

- Add file,
- Drop file,
- Reset database,
- Load directory,
- Undo (one step only).

You can get a list of the database contents by clicking on *RefactorErl/Files/Database contents*. A new tab, called *Database content* pops up (Figure 38), showing the modules having been loaded into RefactorErl. Also, you can select and drop modules from the tool database, and you can open files for editing.

11.3 Executing refactoring transformations

The Eclipse extension supports all the 24 refactorings that are currently available in RefactorErl. In order to execute a transformation, the user has to select

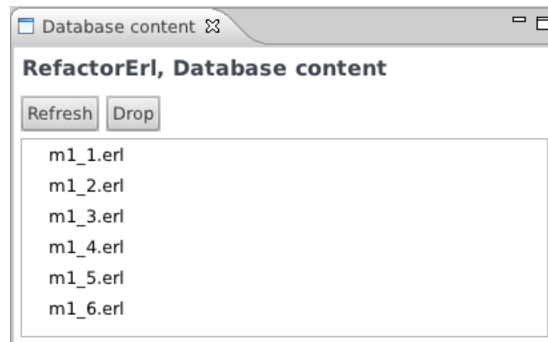


Figure 38: Database contents window

it from the menu and the plugin gives a general interface for customising the transformation.

- The most refactorings require not only a selected range in the source code, but need some additional parameters (e.g. a new function name). At the beginning of the transformation, an input window pops up (Figure 39) where the user can adjust each parameter of the transformation.

- In the case of invalid inputs (e.g. the function name is already in use) the tool gives the opportunity to correct the input (Figure 40) or to cancel the transformation.

- If the transformation is performable, the estimated result appears in a window (Figure 41), where the user can commit or cancel the modifications.

- If a refactoring is cancelled or a fatal error occurs during a refactoring, a window pops up (Figure 42), where the cause of the failure is described.

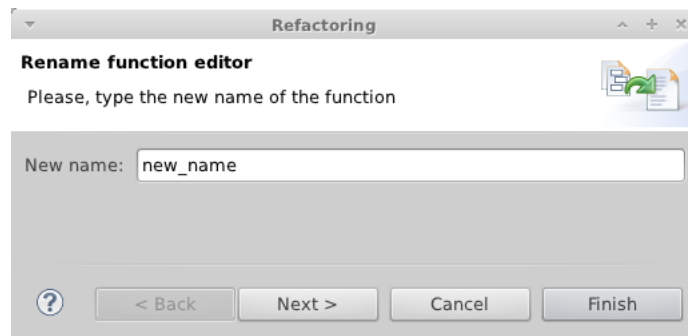


Figure 39: Rename function refactoring

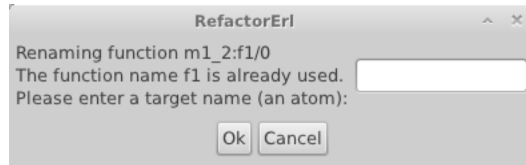


Figure 40: Refactoring interaction window

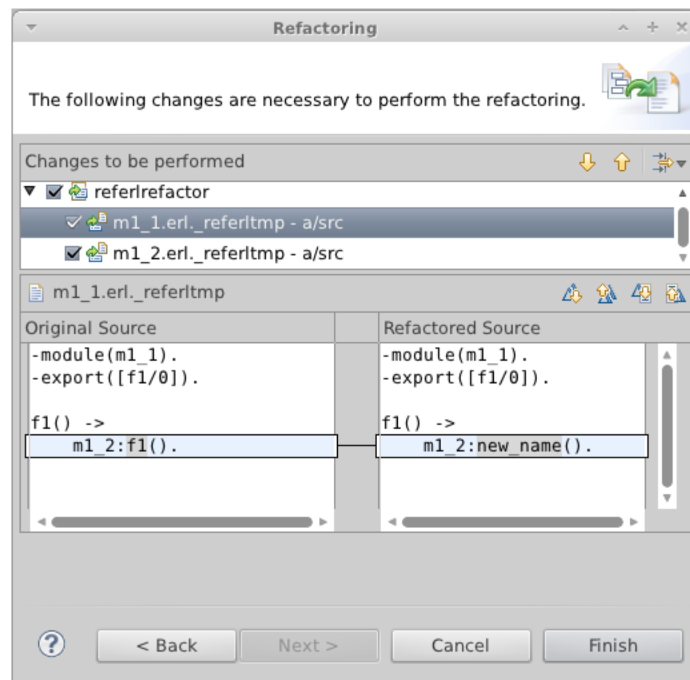


Figure 41: Refactoring result window

11.4 Executing semantic queries

RefactorErl gives support for querying various syntactic and semantic information through our semantic query language. We have created an interface for executing semantic queries within Eclipse, which works very similarly as in Emacs.

The interface dialog is shown on Figure 43). To open such a query tab, just click on *RefactorErl/Semantic query*. It gives an input field for entering queries and also draws a list showing query results in tree structure. If a leaf gets selected, the plugin opens the corresponding file and highlights the queried entity.

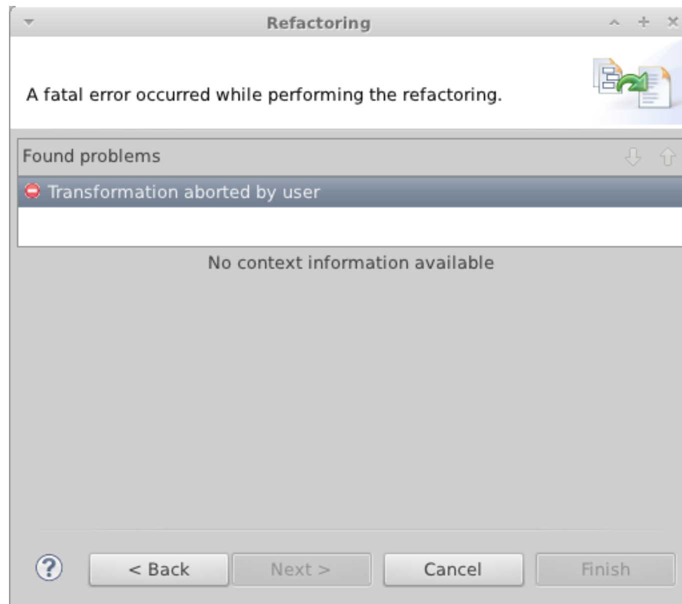


Figure 42: Aborted refactoring window

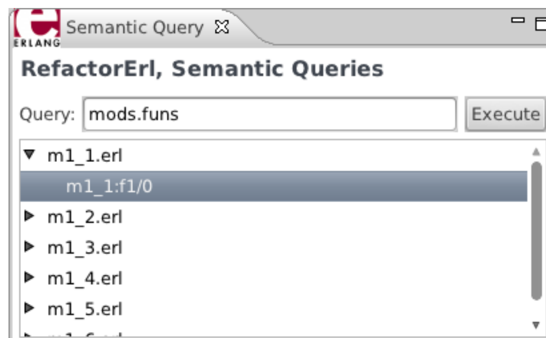


Figure 43: Semantic query window

11.5 Clustering

The RefactorErl tool provides a feature for reorganising the structure of the software, called clustering. It determines semantic dependencies among the modules and functions and makes suggestion to the developer how to repartition the software: blocks of modules, headers etc. As the RefactorErl tool, the Eclipse plugin supports the clustering on different entities like functions or modules. Although the module clustering can be executed using both genetic and agglomerative algorithms, the function clustering supports only the former one.

To execute a clustering algorithm, you should select either of them from the top menu (e.g: *RefactorErl/Module/Genetic*). Every clustering method has the same input window, where you can adjust the parameters of the algorithm. (It's the usage and the required parameters of the different methods are detailed in Section 9.) When the calculation has been finished, a new window pops up with the results of the clustering algorithm.

12 Supporting undo/redo mechanism in Emacs

RefactorErl had an undo operation for refactorings, but with this method could you lose some modifications. Namely if the you edit the text after a refactoring, then undo the changes, all changes since the last refactoring were lost. This undo was one step only, but it could be extended to more steps. However, if we extend this undo, at refactoring undo there can arise conflicts among changes of editing and the transformations - and also among two transformations -, but refactoring undo should handle these conflicts, and it must be as complete as possible. In this section we introduce a new undo/redo mechanism for the tool integrated in Emacs.

12.1 Installation

The new undo mechanism is disabled by default, to enable it, we need additional software components. This extended undo is implemented by a stand alone Haskell program, so we need ghci Haskell Compiler and cabal packaging system, that makes easier the installation of the required packages, modules. The Haskell compiler can be downloaded from the website http://www.haskell.org/ghc/download. Cabal is available at http://www.haskell.org/cabal/download.html or on Linux systems with cabal-install package. The files of this component can be found in lib/referl_ui directory within the tool under emacs-undo. In this directory you have to run

```
make
```

for compiling the source. After installing these components the *load-path* in Emacs has to be set to the new emacs-undo directory instead of emacs.

12.2 Usage

The undo (redo) function is available in the Refactor menu with Undo (Redo) menu item (see Figure 44). Calling the undo operation the program undoes the last change in the current file. Case refactoring the modifications will be rejected in every affected file, but in some cases can conflits arise. When you edit an affected file, or make one another refactoring, the system checks, if the second change affects the refactored area in the file. If the changes are overlapped, the undo reverts also this change, otherwise only the first one, and you get a merged state of the file. At more complex changes can also more changes get reversed. In both cases the changes can be redone until a newer modification.

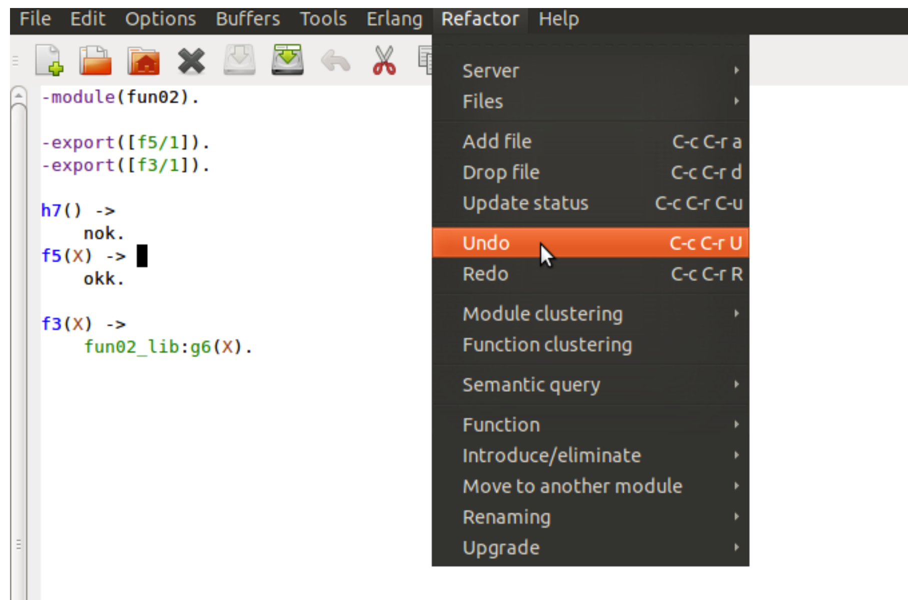


Figure 44: undo

12.3 Examples of behaviour

We give two examples, for showing the behaviour of the merging algorithm of the conflicts.

Example 1 (Figure 45)

1. we move the *pzip* function from module *from* to module *xlists*
2. we edit the module *xlists* after refactoring in area, that is not affected by refactoring (*flatsort*)
3. then we undo the refactoring in the module *from*: merge is successful, the edited text is unchanged

Example 2 (Figure 46)

1. we move the *pzip* function from module *from* to module *xlists*
2. we edit the module *xlists* after refactoring in area, that is affected by refactoring (*pzip*)
3. then we undo the refactoring in the module *from*: conflict arises, the change by editing is rejected

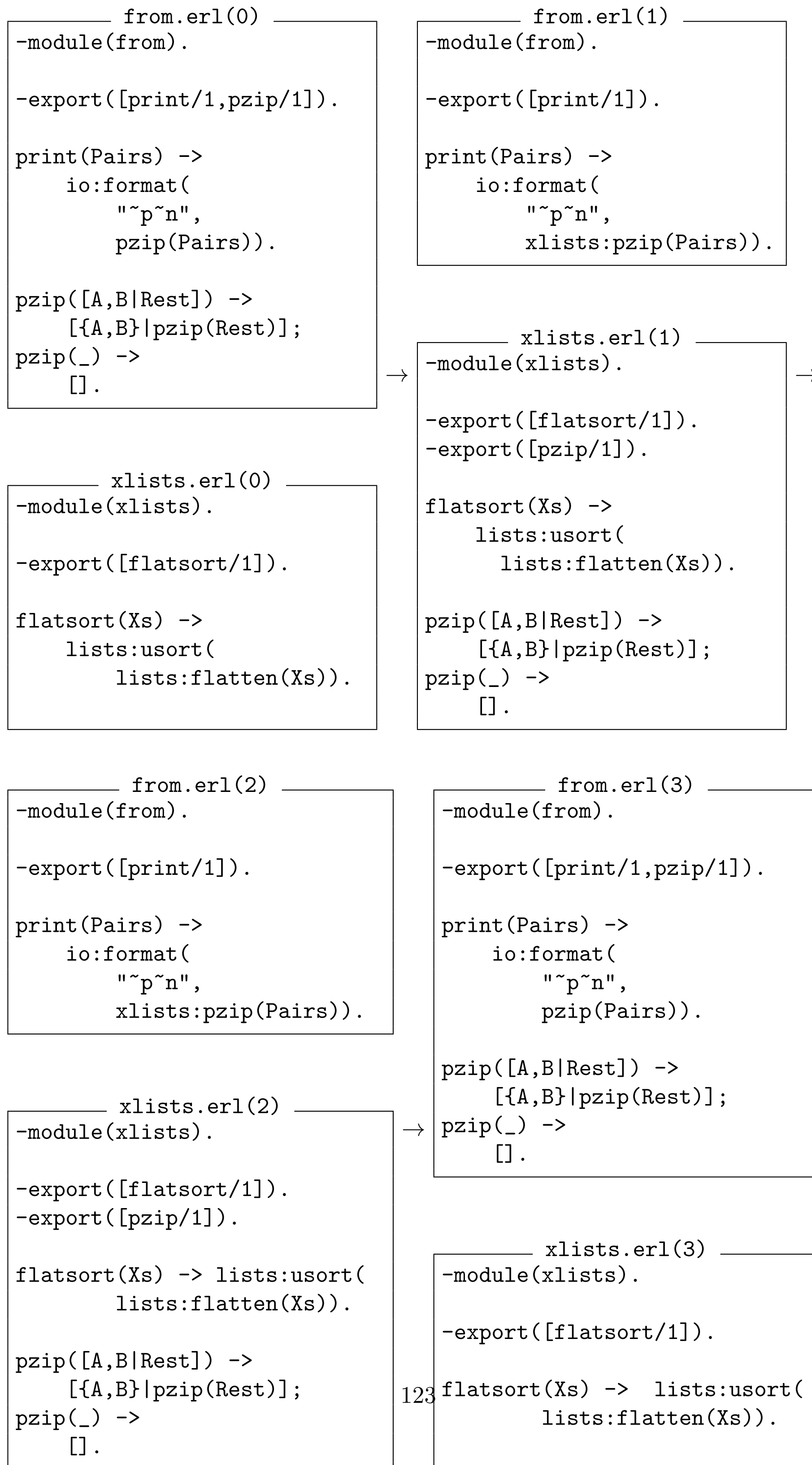


Figure 45: Moving function, editing (merge)

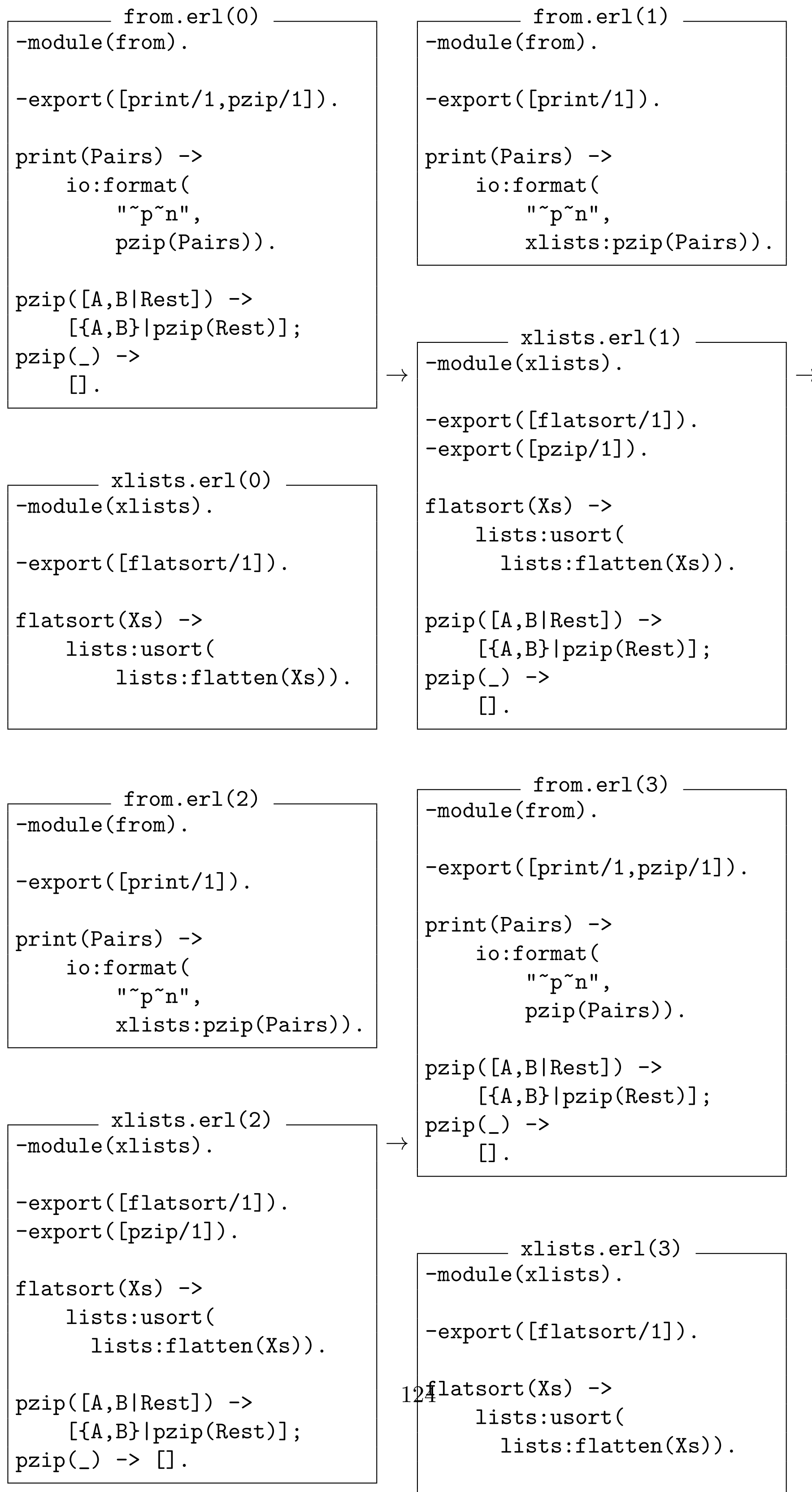


Figure 46: Moving function, editing (conflict)