Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Erlang kódok refaktorálása: Függvénykiemelés

Tóth Melinda

Konzulens: Dr. Horváth Zoltán

Budapest, 2008

Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK, Ericsson Hungary, ELTE CNL and OMAA-ÖAU 66öu2.

Eötvös Loránd University
Faculty of Informatics
Programming Languages and
Compilers Department

Erlang Refactoring: Extract Function

Melinda Tóth

Supervisor: Dr. Horváth Zoltán

Budapest, 2008

CONTENTS

Introduction 5

1. The tool structure 6

2. User Manual 9
- 2.1 Installation guide for Windows 9
  - 2.1.1 Erlang compiler 9
  - 2.1.2 Cygwin: 9
  - 2.1.3 Emacs 9
  - 2.1.4 Source files of the rector tool 11
- 2.2 Installation guide for Linux 11
- 2.3 Minimum requirements 12
  - 2.3.1 Hardware 12
  - 2.3.2 Software 12
- 2.4 Running the tool 13
- 2.5 Extract Function 14
  - 2.5.1 Conditions of applicability 15
  - 2.5.2 Extract function refactoring 15
  - 2.5.3 Flowchart diagram of the refactoring 16
- 2.6 Development interface 19

3. Development Manual 22
- 3.1 Module description - Functions 22
  - 3.1.1 Exported functions 22
  - 3.1.2 Local functions 24
- 3.2 Testing the function and results 41
- 3.3 Extract function test cases 42
  - 3.3.1 Extract sort/2 from gcd. 42
  - 3.3.2 Function name conflicts with another function defined in the same module. 43

3.3.3 Function name conflicts with an imported function. . . . 44
3.3.4 Extract dummy/2 from gcd. 45
3.3.5 Extract dummy/2 from gcd. 46
3.3.6 Extracting function from guard. 47
3.3.7 Extracting a block-expression. 48
3.3.8 Extracting a function from a macro definition. 49

4. Summary 50
4.1 Summary 50
4.2 Related work 50
4.3 Differences 50

List of Figures 52

Bibliography 53

INTRODUCTION

Refactoring [1] is the process of improving the design of a program without changing its external behaviour. Behaviour preservation guarantees that refactoring does not introduce (or remove) any bugs.

An average trend in every programming language is the emergence of claim refactoring tools. Such tools grant safe and fast restructuring and transforming of the code to extend the effectiveness of the programmers. Most of that tools has been concentrated on object-oriented languages, but with the increasing use of functional programming languages it become necessary to develop refactoring tools in this area too.

Our project group developed a prototype of refactoring toolset for Erlang programs. We store the Erlang source code in an SQL database. The source displayed in Emacs. The refactoring tool applicable throw an Emacs menu point. We use the standard Erlang parser, but it does not make it possible to preserve the original look of the source code, and we found other subtle problems as well. So we create a custom parser that works better for refactoring purposes, which involves some changes to the existing syntactical graph model, which is largely based on the standard Erlang syntax tree. So we developed a new, formal semantical graph model that is more suitable for refactoring. And in the new model we use another relational database, Mnesia to store the graph.

This thesis is part of this bigger project. It determine the preconditions, realise the extract function refactoring and the test results.

In the first chapter I give an overview of the tool structure. In the next two part I present the User and the Development Manual, including a description of the testing method.

1. THE REFACTORING TOOL STRUCTURE

The phrase “refactoring” stands for program transformations that preserve the meaning of programs [1]. This transformations are often oriented to improve the quality of program code, as make it more readable, reusable etc. Refactorings are used by the developers every day, as rename variable, extract function etc. In object-oriented word refactoring has already appeared in programming methodologies [12] and it is used heavily in the industry.

In old-fashioned programming environments, refactoring steps have been applied manually by the programmer. But it is very dangerous in many respect, because the semantic changes on the program is not checked. For example if the developer extract a sequence of expression to a function and introduce a new function name which clashes another existing name.

Refactoring in functional languages is not really wide-spread yet, but there are many researches on the topic. For the functional programmer, the only full-featured refactoring tool is HaRe [14, 13] for Haskell programs within the Emacs [3] and VIM [15] editors. A prototype of a refactoring tool for Clean is also available [4, 5].

Our project's goal is to develop a refactoring tool for functional programming language Erlang, which is usable in the industry. The prototype of the tool use the Standard Erlang Parser, but it does not make it possible to preserve the original look of the source code, and we found other subtle problems as well. So we have been created a custom parser that works better for refactoring purposes and it involved some changes to the existing syntactical graph model. So we developed a new, formal semantical graph model that is more suitable for refactoring.

In order to store and manipulate the syntax tree the refactoring tool uses a relational database. The previous version of the RefactorErl prototype tool stores the tree in MySQL which is not part of the Erlang system. The current version uses Mnesia. Mnesia is a distributed DataBase Management System (DBMS), appropriate for telecommunications applications and other Erlang applications which require continuous operation and exhibit soft real-time

properties.

The interface to the graph representation of an Erlang program is defined in two layers. The first layer is a generic semantic graph abstract data type that can store and efficiently retrieve the results of syntactic and semantic analysis. The second layer consists of a graph model of Erlang programs and a set of modules for analysing, transforming and restoring Erlang source code.

Information retrieval from the graph

Under developing a refactoring the most used action is the graph manipulation. In our model the high level information retrieval is supported by a query language.

This query language consists of *path expressions*:

```
path() = [PathElem]
PathElem = Tag | {Tag, Index} | {Tag, Filter} |
                 {intersect, Node, Tag}
Tag = atom() | {atom(), back}

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

For example:

```
path(Root, [{file, {fullpath, '==', 'c:/extract.erl'}}]),
```

The result would be a list containing one element: the source node of that file which full path is c:/extract.erl

My primal role in the project is the extract function refactoring.

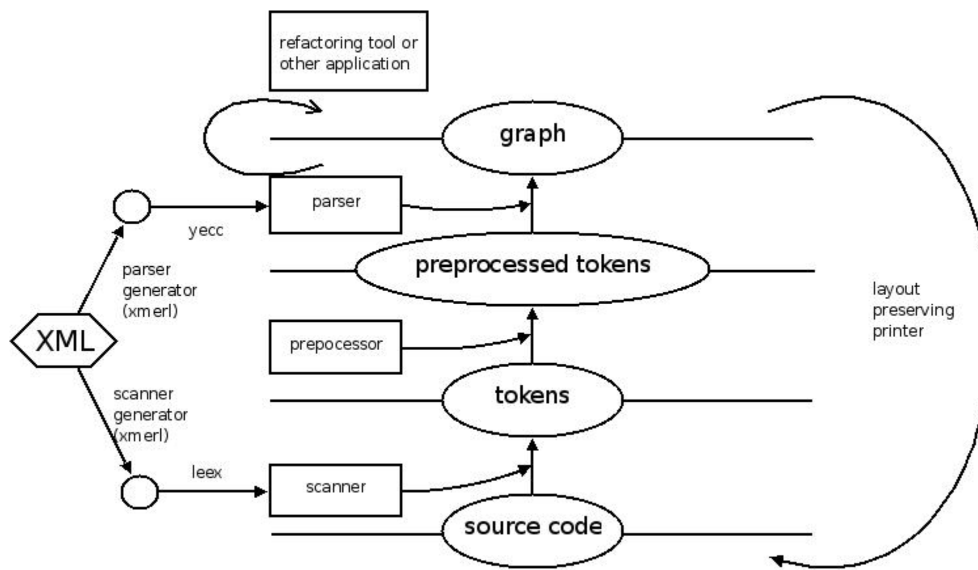


Fig. 1.1: The structure of the tool

2. USER MANUAL

2.1 Installation guide for Windows

There are four main component of the tool which need to install: Erlang compiler, Emacs, (optional) Cygwin and the source files of the refactor tool.

2.1.1 Erlang compiler

1. Download the file otp_win32_R12B-0.exe free from http://www.erlang.org/download.html.
2. Start the installation, choose or create a directory-name which does not contain space, for example: c:\refactorerl\erlang

2.1.2 Cygwin:

1. Download the file cygwin.zip free from http://cygwin.org/.
2. Download the Cygwin installer and install.

2.1.3 Emacs

1. Download the Emacs source from http://www.gnu.org/software/emacs/. Copy the source to an arbitrary directory (for example: c:\refactorerl\emacs).
2. Run (c:\emacs)\bin\addpm.exe
3. Start emacs choose the Save options from Options menu this will generate your emacs file
4. Copy this lines into your own .emacs file:

   (add-to-list 'load-path "c:/refactorerl/refactorerl/emacs")

(require 'refactorerl)

5. (Optional) Add this line below too, and then the refactorel server will start automatically, when you open an erlang file:

   (add-hook 'erlang-mode-hook 'refactorerl-mode)

6. Copy the .erlang.cookie file from the user's default directory (usually c:\Documents and Settings\Username) to the same place, where your .emacs file is.

7. Set the path as the pictures below show:

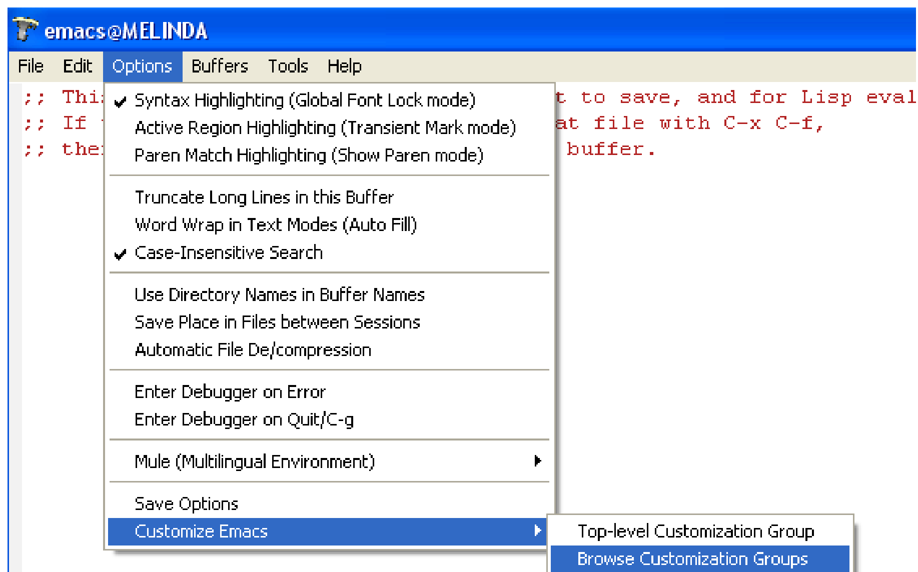


Fig. 2.1: The customize menu in emacs

Choose the Emacs Options menu point, and then Customize Emacs and Browse Customization Groups. From the appeared list choose the File Group. From File Group choose the Aoutomount Dir Prefix menu point and write the path to source of the refactor tool there.

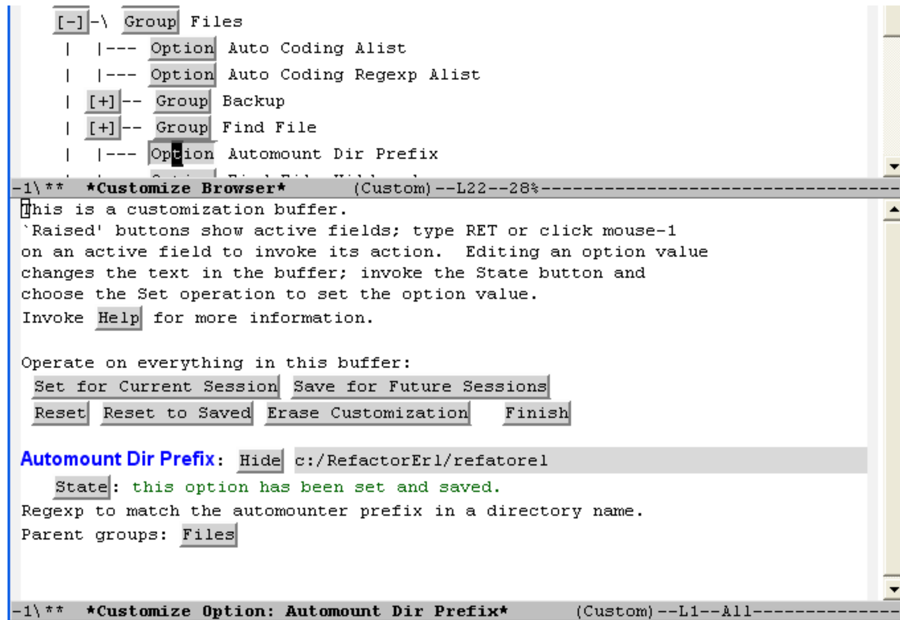


Fig. 2.2: Setting the path in emacs (the refactorerl directory include the source of the refactor tool)

2.1.4 Source files of the rector tool

1. Copy the source files to your dictionary for refactor modules (c:\refactorerl \refactorerl).

When using the Refactoring Tool numerous components have to be installed and configured manually, so our group developed and Installer program which makes this simpler. That installer developed for the prototype of Erlang refactoring tool, so it install the MySQL and MySQL/ODBC-connector, but that programs are not necessary for the new tool - it use Mnesia to store the semantic graph. After running that installer just the emacs costumization is necessary.

2.2 Installation guide for Linux

The components and the methods are similar as the installation in Windows system.

2.3 Minimum requirements

2.3.1 Hardware

250 MB free disk space for the components and 250 MB or more for the database

256 MB memory

2.3.2 Software

The four components and Windows XP/2000/2003/NT or Linux operating system.

2.4 Running the tool

The user has to start Emacs at first, and then open an Erlang file. If the user did not set in the .emacs file the automatic start, he has to set the refactorerl-mode(Alt+x refactorerl-mode) and start the refactorerl server. After that he can use the tool with commands: Alt+x refactorerl-'command'. Commands are:

1. quit - stops the refactor server
2. restart - restarts the refactor server
3. add-file - adds the current file to the graph
4. drop-file - drops the current file from the graph
5. debug-shell - open the debug shell
6. extract-function - extracts the selected part to a function, if the conditions are satisfied
7. draw-graph - asks a file name and draws the semantic graph to he file
8. update-status - updates the refactor server status

We have created a local keymap for Erlang Refactoring minor mode. That defines shortcut key:

1. Ctrl+c Ctrl+r Q : quit
2. Ctrl+c Ctrl+r R : restart
3. Ctrl+c Ctrl+r a : add-file
4. Ctrl+c Ctrl+r d : drop-file
5. Ctrl+c Ctrl+r e : debug-shell
6. Ctrl+c Ctrl+r G : extract-function
7. Ctrl+c Ctrl+r D : draw-graph
8. Ctrl+c Ctrl+r Ctrl+u : update-status

Before doing an extract function refactoring the user has to add the file to the graph and mark an expression, a subexpression or a sequence of expression. If it does not happen a warning will be appear that the file is not ready for refactoring.

2.5 Extract Function

An alternative of 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 expression(s): those variables which are bound outside of the expression(s), but the value of which is used by the expression(s). The extracted function will not be exported from the module.

In the example in Fig. 2.3 a function for sorting a pair (a two-tuple) of numbers is extracted from the tail-recursive definition of the gcd (greatest common divisor) function. The subexpression which is extracted into function sort/2 starts at the keyword `if`, and ends at the keyword `end`.

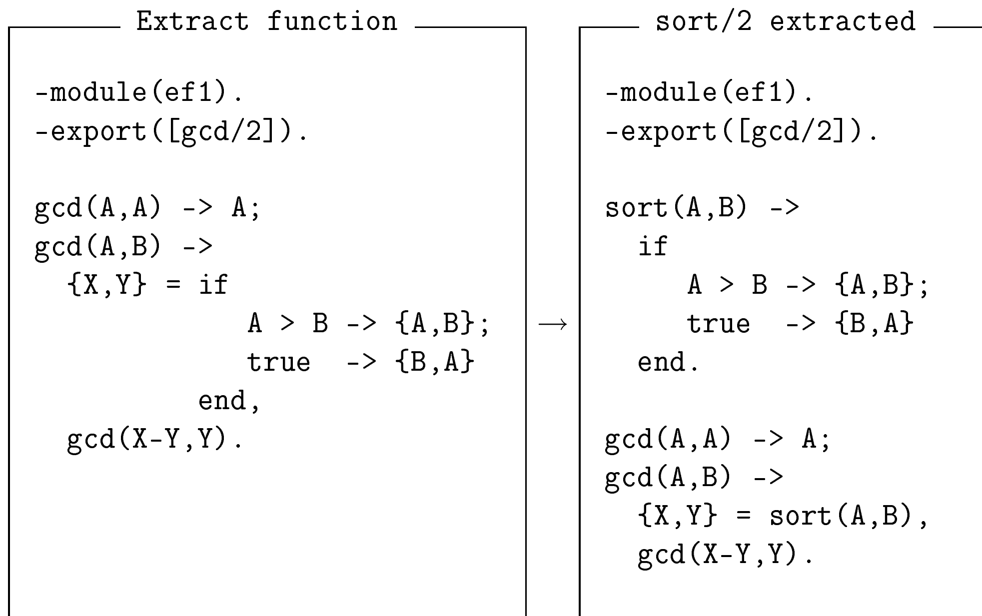


Fig. 2.3: Extract function `sort/2` from function `gcd/2`.

2.5.1 Conditions of applicability

1. 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.
2. The starting and ending positions should delimit a sequence of expressions.
3. Variables with possible binding occurrences in the selected sequence of expressions should not appear outside of the sequence of expressions.
4. The extracted sequence of expressions cannot be part of a guard sequence.
5. The extracted sequence of expressions cannot be part of a pattern.
6. The extracted sequence of expressions cannot be part of macro definition.
7. The extracted sequence of expressions cannot be a list generator.

2.5.2 Extract function refactoring

Most of the refactorings could distribute three part: collect necessary data, check the conditions of applicability and perform the modifications on the source.

Collect information

In this part of the refactoring the module collects the necessary informations: module identifiers, the selected expression(s) identifier(s), used variable, bounded variables and not bounded variable from the selected part. The module throws various terms when there is an error.

Check conditions

Using the collected data the function checks the conditions of applicability: the module exists in the graph, the selected part a legal function body, the expression do not appear a part of macro definitions, pattern, a list generator or guard expression, the given function name is a legal function name and do not conflict with another function in the module.

Perform refactoring

In the last part of the refactoring, the module extracts a selected sequence of expression to a function. It consist of two part. Create the new function definition and the the new application, and replace the selected part with the application and insert it to the graph. After that the refactoring will reconstruct the semantical information on the graph.

Parameters

The refactoring module path, the starting and ending positions of a subexpression, an expression or a sequence of expressions, and the name of the function to introduce.

2.5.3 Flowchart diagram of the refactoring

The following flowchart diagram (in Figures 2.4 and 2.5) shows the structure of the extract function (refac_extract_fun) module. The applied signs are:

- The begin and the end points are marked with green ovals.
- The error messages are marked with red ovals (the function ends with an error message).
- The decision points are marked with yellow diamonds.
- The red arrow texts are the results of the decisions.
- The blue italic arrow texts are the current branch, for example the node is an application or a clause.
- The functions of the tool are marked with white boxes.
- The functions of the tool — which inside algorithms are detailed — are marked with light blue boxes.

Fig. 2.4: The structure of the refac_extract_fun module

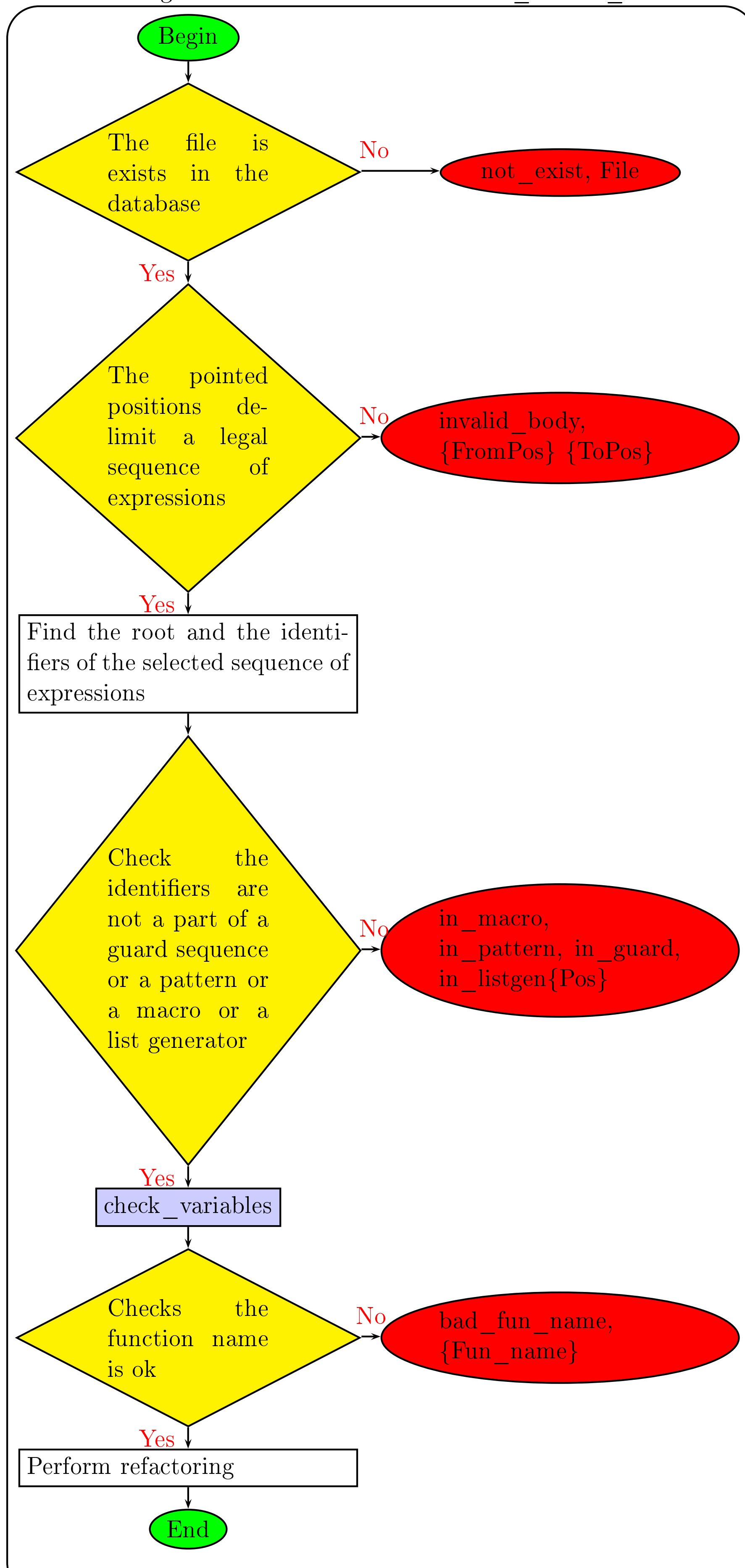
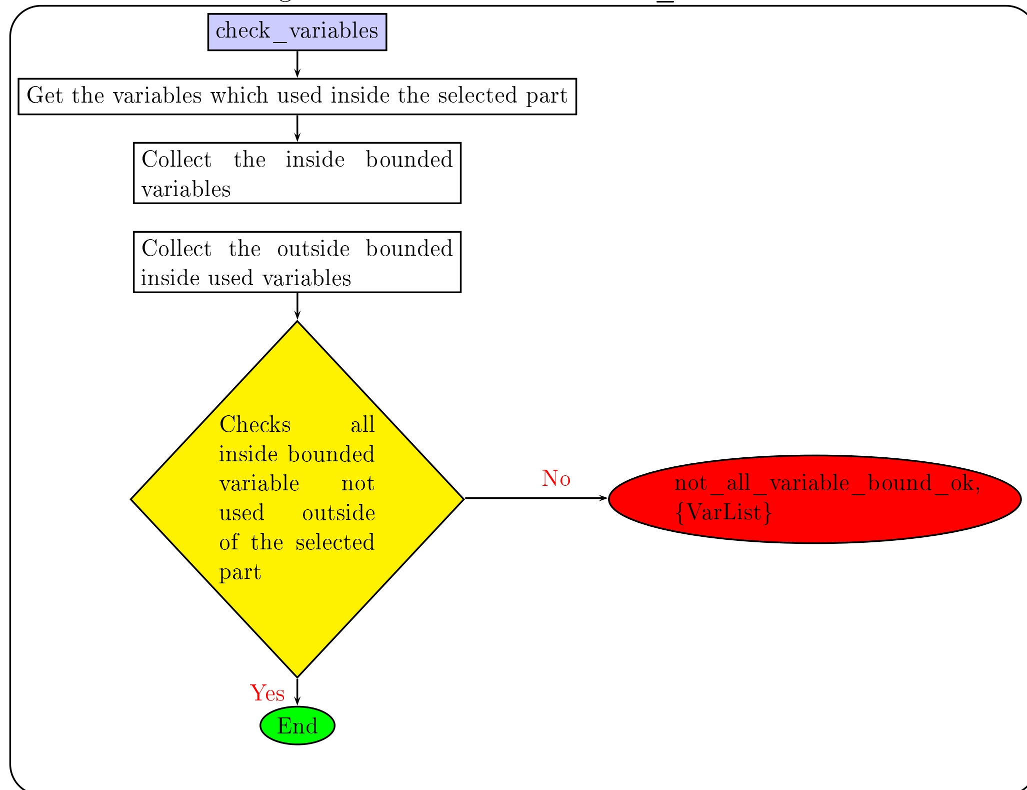


Fig. 2.5: The structure of check_variables



2.6 Development interface

The extract function refactoring is mostly used by Erlang programmers, who need safe and fast restructuring and transforming on his code.

The tool using Emacs because it seems most Erlang programmers use it. After the Emacs customization the tool provides facilities for on-line transformations.

The programmer should start the refactorerl server and add to the current file to the graph. After it the file is ready for refactoring. The user should mark the part which he want to extract in a function. Then calls the extract function with Ctrl-c Ctrl-r e. Emacs asks a function name. If the conditions are performed the extraction will be done.

Refactoring is the process of improving the design of a program without changing its external behaviour. That is the reason because we forbid some kind of transformation.

In the example in Fig. 2.6 extract the expression `W = X + Y + Z` is forbidden, because the variable W is bounded in the selected part, and its value is used outside of the selected part.

While extract the subexpression `X + Y + Z` is allowable.

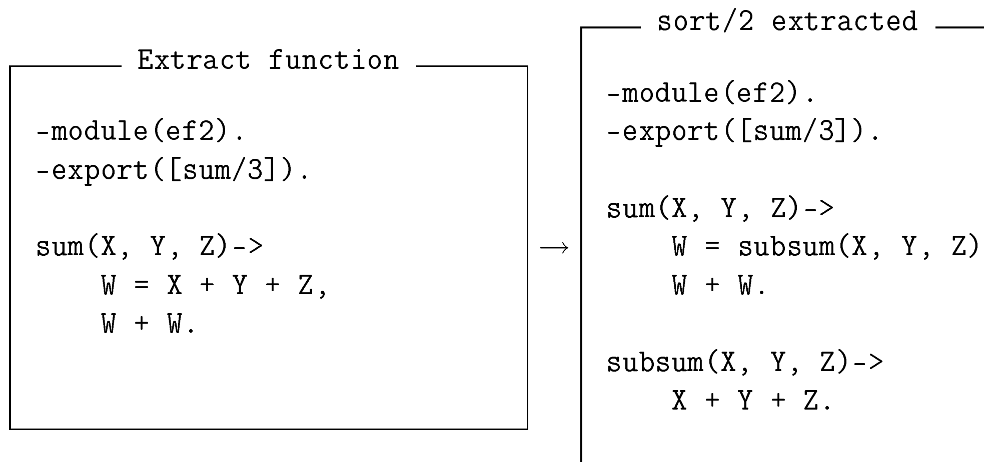


Fig. 2.6: Extract function `subsum/2` from function `sum/3`.

In the example in Fig. 2.6 extract the expression `W = X + Y + Z,` is forbidden too. The comma is a delimiter, it is not a part of the expression. In that case the extract said `not_legal_body` error message.

In the example in Fig. 2.7 a function for sorting a pair (a two-tuple) of numbers is extracted from the tail-recursive definition of the gcd (greatest common divisor) function. The expression which we would extracted into function dummy is the expression `A`. But in the module, there is a function called dummy with arity 1, so the extract will not be done.

Otherwise if the subexpression which we want extract into function dummy starts at the keyword `if`, and ends at the keyword `end`, the refactoring is performs, because the extracted function will have 2 parameter.

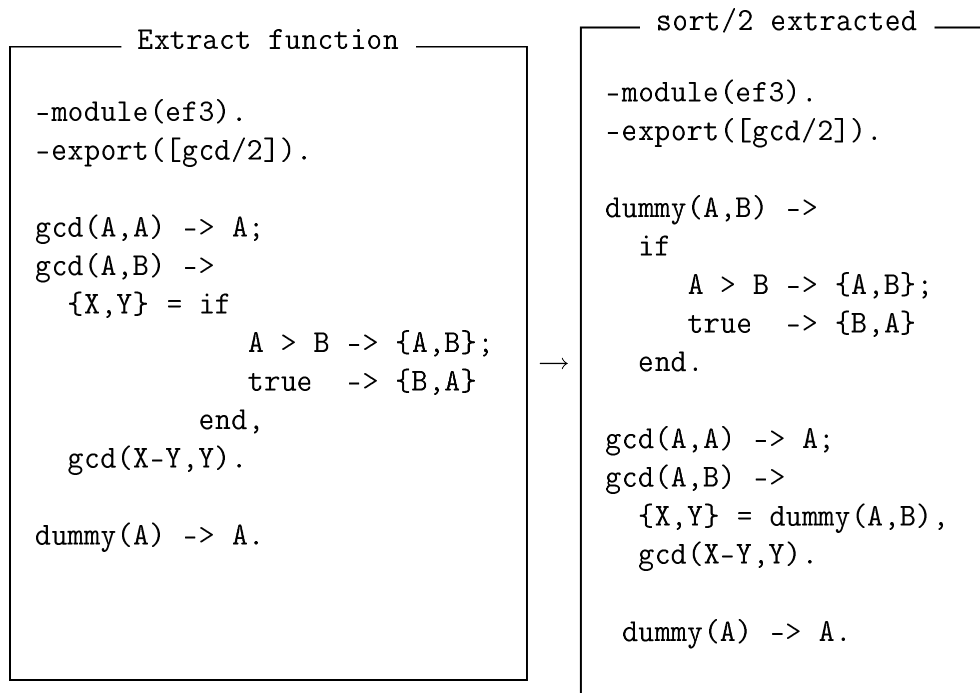


Fig. 2.7: Extract function `dummy/2` from function `gcd/2`.

However using the tool via Emacs is simple, it is not the only possibility. The tool can use by calling its exported function. The user has to compile the source of the tool, and after it he can use its exported functions. To compile the tools modules there is a simple support in the tool. The `build:build()` function prepare all necessary components.

Of course the first step should be start the refactorerl server and adds the file to the graph. For example:

If the user stands in the refactorerl directory, the command (via Con-

sol or CygWin): `erl -noshell -pz ebin -pz build -boot refactorerl -run refac_syntax_builder file filename` - starts the server and add the filename Erlang file to the graph. After it is possible to use the extract fun refactoring from command line.

The tool's operation can use on an Erlang Node too.

3. DEVELOPMENT MANUAL

3.1 Module description - Functions

The module extract a selected sequence of expression to a function, only if the conditions are satisfied, or throws various terms when there is an error.

3.1.1 Exported functions

extract_function/4 Performs the precondition checks and collects all the necessary information for the refactoring. It throws various terms when there is an error, with this structure: atom(), term(). If the atom is `ok`, the refactoring has been preformed without a problem.

- Parameter description:
  1. File: The path of the module.
  2. FromPos: The pointed first position in the editor.
  3. ToPos: The pointed last position in the editor.
  4. NewName: The name of the extract function.
- Implementation:
  - At first the function gets the node of the current module and the node of the current source; and checks that the module exists in the graph or not. If the module is not exists in the graph the function throws an error message in a tuple. The first element is `not_exists` atom, the second element the path of the file.
  - After it, the function searched the pointed tokens from the pointed positions; and the first and the last expression node from these tokens, and checks that the selected part delimits a sequence of

expression or not. If the selected part not a legal function body, the function throws an error message in a tuple. The first element is `not_legal_body` atom, the second element is the pairs of the pointed positions.

- After that the function checks that the identifiers are not a part of a guard sequence or a pattern or a macro or a list generator and throws `in_macro`, `in_pattern`, `in_guard` atoms, `in_listgen` if the check is fail.
- Using the first and last expression node we can get the selected expression node's list. If the first and last expressions are equal there is only one expression or just a subexpression in the selected part. The other case there are more top level expressions is the selected part.
- From the expression node(s) we collect the used variable nodes and its variable objects and sort it two part: bounded and not bounded variables in the selected part.
- After collecting the variable the function checks the conditions about function name. The number of the variables which used inside but bounded outside of the selected part become the arity of the function. The name and arity should not conflict with another function, either defined in the same module, or imported from another module and it should be a legal function name. The function throws `exists_error`, `import_error`, `not_function_name_error` atoms and the incorrect function name in an error message.
- If there is not any problem with the name, it follows with checks that the variables with possible binding occurrences in the selected sequence of expressions should not appear outside of the sequence of expressions. If not all variable bound ok, the function throws an error message in a tuple. The first element is `not_all_var_bound_ok` atom, the second element is a list of the outside used inside bounded variables.
- If the binding is ok, we can create the new function and the application. The name is the second parameter, the body is the selected sequence of expression, the parameters are the variables which used inside but bounded outside of the selected part, and the function does not have any guard sequence.

- After that the application removes the selected part, inserts the created application and function to the graph and does the necessary semantic reconstruction.

This function is used via Emacs to start the refactor step.

3.1.2 Local functions

get_module_and_source_node/1 Gets the module node and the source node from the path of the module.

- Parameter description:
  1. File: The path of he module
- Implementation:
  - At first the function calls the `?GRAPH:root()` function and gets the root node from the graph.
  - After it gets the source node of the graph. If she source does not exist in the graph the function throws a `not_exists` error message and the refactoring is finished.
  - From the source node gets the module node, and if it does not exist in the graph the function throws a `not_exists` error message and the refactoring is finished.
  - It returns with the source and the module node in a tuple.

The function is used in `extract_function/4`.

get_token_from_pos/2 Gets the token node form the position.

- Parameter description:
  1. FileName: The module name.
  2. Pos: The pointed position in the editor.
- Implementation:
  - At first it gets the token node from the position. If there is not any token on the given position, it throws a `bad_position` error message and the refactoring is finished.

- Otherwise it gets the path from the root to the token, and returns with the token node and the path in a tuple.

The function is used in `extract_function/4`.

get_first_and_last_expression/4 Gets the first and the last expression node.

- Parameter description:
  1. FromToken: The pointed first token.
  2. ToToken: The pointed last token.
  3. ResultList: The common part of the paths from the root to the pointed tokens.
  4. Path: Path from the root to the pointed first token.
- Implementation:
  - The function calls the `one_token/1` function, if the first element of the ResultList is `one_token` atom. Otherwise calls `more_token/3` function.
  - Finally it return with a tuple which contain four element: the pointed first and last expression node, an atom(Found) and the root of the common part of the paths.
  - The found atom has three value:
    1. `not_found`: the selected part not a legal function body
    2. `found_body`: the selected part is a sequence of expression
    3. `found_expr`: the selected part is an expression or a subexpression

The function is used in `extract_function/4`.

one_token/1 Gets the first and the last expression node if only one token is selected.

- Parameter description:
  1. Path: Path from the root to the pointed first token.

- Implementation:
  - It gets the pointed expression node from the path, which is the next to last element of the path.
  - If the element's first token is the pointed token, it returns with the expression node and the `found_expr` atom, else with the `not_found` atom.

The function is used in `get_first_and_last_expression/4`.

more_token/3 Gets the first and the last expression node if more tokens were selected.

- Parameter description:
  1. FromToken: The pointed first token.
  2. ToToken: The pointed last token.
  3. ResultList: The common part of the paths from the root to the pointed tokens.
- Implementation:
  - At first it gets the Top expression from the ResultList: the root of the common part of the paths.
  - If the first or last token is the first token of an expression, the function gets that nodes, else it will be an empty list.
  - If at least one of them are not the first token of an expression, it returns with the `not_found` atom.
  - Otherwise there are three possibilities:
    1. The top expression is a clause: in that case the selected part is a sequence of expression. The function returns with a tuple which contain four elements: the pointed first and last expression node, the `found_body` atom and the Top expression node.
    2. The top expression is an expression: in that case the selected part is a subexpression or a sequence of expression. The function returns with a tuple which contain four elements: the

pointed expression node, the `found_expr` atom and the Top expression node.
3. Else the function returns with `not_found` atom.

The function is used in `extract_function/4`.

get_expr_list/3 Gets the selected expressions nodes, and every top level expression node from the selected clause.

- Parameter description:
  1. FirstExpr: The pointed first expression's node.
  2. LastExpr: The pointed last expression's node.
  3. ClauseNode: The top expression node.
- Implementation:
  - If the first and last expression node are unequal, the selected part is a sequence of expression. In this case the Top expression node is the clause node which include the selected part.
  - If the first and last expression node are equal, the selected part is a subexpression or an expression. In this case the function gets the top level expression and after from that expression the clause node.
  - From the clause node the function gets the top level expression.
  - In the first case we should select the pointed expressions node from the top level expression nodes. The function uses the `lists:dropwhile` function. While the first expression do not found, the function delete the top level nodes from the list. After it reverse the list and while the last expression do not found, the function delete the top level nodes from the list. And then reverse the remaining list.
  - In the second case the expression list is the list which has one element, the first expression node.

The function is used in `extract_function/4`.

get_used_variable_node_and_object/2 Gets the used variables and it's variable object from the expressions.

- Parameter description:
  1. List: The selected expression(s) node list.
  2. Found: The Found atom.
- Implementation:
  - At first it test the value of the Found atom.
  - If it is `found_body`, then calls `get_used_variable_node_and_object_body/1` function, else calls `get_used_variable_node_and_object_subexp/1` function.
  - In both case the function return the selected part variable nodes and its variable objects in a tuple

The function is used in `extract_function/4`.

get_used_variable_node_and_object_subexp/1 Gets the used variables and it's variable object from a subexpression or an expression.

- Parameter description:
  1. List: The selected (sub)expression node list.
- Implementation:
  - If the expression node is a variable, then gets the variable object and returns with the expression node and the variable object.
  - Otherwise gets the variable from the (sub)expression calling the `get_var_from_sub/1` function.
  - After gets the variable objects.
  - Then the function returns with the variable node(s) and the variable object(s).

The function is used in `get_used_variable_node_and_object/2`.

get_var_from_sub/1 Gets the variables from a (sub)expression.

- Parameter description:
  1. List: Expression node list.
- Implementation:
  - If the List is empty it returns with an empty list.
  - Otherwise the function gets the variables from the head of the list, and after the tail of the list. It calls `get_var_from_clause/1` and `get_var_from_sub/1` function.
  - At last it returns the (sub)expression's variable node(s).

The function is used in `get_used_variable_node_and_object_subexpr/1`.

get_var_from_clause/1 Gets the variables from a clause.

- Parameter description:
  1. List: Expression node list.
- Implementation:
  - If the List is empty it returns with an empty list.
  - Otherwise the function gets the variables from the head of the list, and after the tail of the list. It calls `get_var_from_clause/1` and `get_varnode/1` function.
  - At last it returns the expression(s) variable node(s).

The function is used in `get_var_from_sub/1`.

get_used_variable_node_and_object_body/1 Gets the used variables and it's variable object from top level expressions.

- Parameter description:
  1. List: The selected expressions node list.
- Implementation:

  - At first the function calls the `get_varnode/1` function.
  - After gets the variable object from variable nodes.
  - At last it returns the variable node's list and the variable object's list in a tuple.

The function is used in `get_used_variable_node_and_object/2`.

get_varnode/1 Gets the variables from an expression list.

- Parameter description:
  1. List: Expression node list.
- Implementation:
  - If the List is empty it returns with an empty list.
  - Otherwise the function gets the variables from the head of the list, and after the tail of the list. It calls `get_varnode/1` function recursively.
  - The function returns the expression(s) variable nodes in a list.

The function is used in `get_used_variable_node_and_object_body/1` and in `get_var_from_clause/1`.

get_inside_bounded_variables/2

- Parameter description:
  1. VarNode: The used variable nodes.
  2. VarObject: The variable objects.
- Implementation:
  - At first the function gets the variable binding list form the used variable objects calling the `get_bindings/1` function.
  - After it gets the variable nodes which bounded inside of the selected part, and gets that variable's objects.
  - Then calls the `get_varref/1` function and collect the the variable objects references.

- After it selects the not bounded variable nodes.

The function is used in `extract_function/4`.

get_bindings/1 Gets the variable binding from the variable object.

- Parameter description:
  1. List: List of variable objects.
- Implementation:
  - If the List is empty it returns with an empty list.
  - Otherwise the function gets the variables from the head of the list, and after the tail of the list. It calls `get_bindings/1` function recursively.
  - The function returns the variable(s) binding nodes in a list.

The function is used in `get_inside_bounded_variables/2`.

get_varref/1 Gets the variable occurrence from the variable object.

- Parameter description:
  1. List: List of variable objects.
- Implementation:
  - If the List is empty it returns with an empty list.
  - Otherwise the function gets the variable occurrence from the head of the list, and after the tail of the list. It calls `get_varref/1` function recursively.
  - The function returns the variables occurrence nodes in a list.

The function is used in `get_inside_bounded_variables/2`.

get_varname/1 Gets the variable name.

- Parameter description:
  1. List: List of variable nodes.
- Implementation:
  - At first the function gets the variable object from the varnode and after gets the variable name from the object.
  - It returns with the variable names in a list.

The function is used in `extract_function/4`.

check_is_legal_body/4 The function checks if the source between the two position is a legal expression or not.

- Parameter description:
  - FromPos : The pointed first position in the editor.
  - ToPos : The pointed last position in the editor.
  - ModuleNode : The node of the module.
  - Found : The found atom.
- Implementation:
  - If the Found atom equals to `not_found`, then throws `not_legal_body` error message.

The function is used in `extract_function/4`.

check_is_legal_name/3 Checks that the function name is a legal function name, is not exists and is not imported in the module .

- Parameter description:
  1. ModuleNode: The current module node.
  2. NewName: The name of the new function.
  3. Arity: The arity of the new function.

- Implementation:
  - It calls `check_is_function_name/1`, `check_the_name_already_exists/3` and `check_the_name_is_imported/3` functions with the corresponding parameters.

The function is used in `extract_function/4`.

check_is_function_name/1 Checks if the new function name is acceptable as a function name.

- Parameter description:
  1. NewName: The name of the new function.
- Implementation:
  - If the function name does not begin with a lowercase then throws `not_function_name_error` error message.

The function is used in `check_is_legal_name/3`.

check_the_name_already_exists/3 Checks if introducing a new function would clash with the existing ones.

- Parameter description:
  1. ModuleNode: The current module node.
  2. NewName: The name of the new function.
  3. Arity: The arity of the new function.
- Implementation:
  - At first it gets the function object from the module.
  - Then collects in a list the function objects which have the same name and arity than the new function has.
  - If that list is not empty, then throws `exists_error` error message.

The function is used in `check_is_legal_name/3`.

check_the_name_is_imported/3 Checks if introducing a new function would clash with the importing ones.

- Parameter description:
  1. ModuleNode: The current module node.
  2. NewName: The name of the new function.
  3. Arity: The arity of the new function.
- Implementation:
  - At first it gets the imported function object from the module.
  - Then it collects in a list the function objects from the imported functions which have the same name and arity than the new function has.
  - If that list is not empty, then throws `import_error` error message.

The function is used in `check_is_legal_name/3`.

check_not_in_pattern_guard_macro_listgen/2 Checks the expression id are not a part of a guard sequence or a pattern or a macro or a list generator node.

- Parameter description:
  1. FirstExpr: The pointed first expression's node.
  2. LastExpr: The pointed last expression's node.
- Implementation:
  - If the first and last expression node are unequal, the function returns with `ok` atom.
  - Otherwise if the type of the expression is `pattern` then throws `in_pattern` error message.
  - If the expression is not a pattern, then the function analyze the kind of the expression. If it is `guard` or `list_gen`, it `in_guard` or `in_listgen` error message.
  - After it checks that the expression is a part of a macro application.If it is true, then throws `in_macro` error message.

The function is used in `extract_function/4`.

check_all_var_bound_ok/3 Checks that all variables with possible binding occurrence in the selected sequence of expression not appear outside of this sequence.

- Parameter description:
  1. AllSupExpr: The expression node list.
  2. UseVarObject: The variable object list.
  3. BoundVarNode: The bounded variable node list.
- Implementation:
  - At first it gets all used variable nodes and object from the AllSupExpr list calling `get_used_variable_node_and_object_body/1` function.
  - After it gets the outside used variables.
  - Then collect that outside used inside bounded variables into a list.
  - If that list is not empty than throws `not_all_var_bound_ok` error message.

The function is used in `extract_function/4`.

perform_refactoring/8 Removes the selected part, inserts the created application and function to the graph and does the necessary semantic reconstruction.

- Parameter description:
  1. Found: The Found atom.
  2. SourceNode: The node of the source.
  3. NewName: The created function name.
  4. NotBoundVarName: The not bounded variable names
  5. ExprList: The selected expression's node list.
  6. Top: The selected expression's clause.
  7. FromToken: The pointed first token.

8. ToToken: The pointed last token.

- Implementation:
  - Gets the function clause node.
  - Gets information from the previous and the next token.
  - Removes the selected expression node(s).
  - Then creates the parameter and the name for the application and for the function.
  - Then creates and insert the application and the function.
  - Updates the token structure.
  - At last it close the transaction and the analyze modules reconstruct the semantic information.

The function is used in `extract_function/4`.

remove_nodes/7 Removes the selected part from the graph.

- Parameter description:
  1. Found: The Found atom.
  2. Top: The selected expression's clause.
  3. ExprList: The selected expression's node list.
  4. FromToken: The pointed first token.
  5. ToToken: The pointed last token.
  6. FunLastToken:
  7. FunLastTokenNext:
- Implementation:
  - Removes the selected part from the original function.
  - Makes a note abut the previous and the next token.
  - Returns with that tokens.

The function is used in `perform_refactoring/8`.

create_parameters/1 Creates variables from name.

- Parameter description:
  1. NameList: The names of the variables.
- Implementation:
  - At first creates a variable node with the given name.
  - After creates a token and link it to the variable node.
  - Returns with the variable node list and the token node list in a tuple.

The function is used in `perform_refactoring/8`.

create_name/1 Creates an atom with the given function name.

- Parameter description:
  1. NewName:The new function name.
- Implementation:
  - Creates a name node and a name token with the given name.
  - After links the token to the node.
  - Retuns with the name node and name token in a tuple.

The function is used in `perform_refactoring/8`.

create_application/5 Creates the new application and inserts it to the graph.

- Parameter description:
  1. Found: The Found atom.
  2. Top: The selected expression(s) clause.
  3. PatternNodes: The application parameter nodes.
  4. NameNode: The application name node.
  5. Index: The index of the insert position in the graph.

- Implementation:
  - Creates the application node.
  - Inserts the application name node, the application pattern nodes to the graph..
  - Returns with the application node.

The function is used in `perform_refactoring/8`.

create_function/5 Creates the new function and inserts it to the graph.

- Parameter description:
  1. SourceNode: The current source node.
  2. FunPatternNodes: The function parameter nodes.
  3. FunNameNode: The function name node.
  4. ExprList: The selected expression node(s).
  5. Index: The index of the insert position in the graph.
- Implementation:
  - Creates the function clause and the function node.
  - Inserts the function name node, the function pattern nodes and the function body to the function clause.
  - Returns with the function node and the function clause node in a tuple.

The function is used in `perform_refactoring/8`.

token_update/13 Creates the tokens for the new function and the new application, and insert them to the graph.

- Parameter description:
  1. FromToken: The pointed first token.
  2. ToToken: The pointed last token.
  3. BeforeFromToken: The token before the FromToken.

4. AfterToToken: The token after the ToToken.
5. AppNameToken: The application name token.
6. FunNameToken: The function name token.
7. AppPatternTokens: The application pattern tokens.
8. FunPatternTokens: The function pattern tokens.
9. FunClause: The selected function clause.
10. Function: The created function node.
11. AppNode: The created application node.
12. FuncLastToken: The pointed function last token.
13. FuncLastNext: The token after the FunLastToken.

- Implementation:
  - At first links the application and function name token the corresponding place.
  - Then links the application and the function parameters calling the `insert_parameters_with_parenthesis/4` function.
  - Create and insert an arrow.
  - Insert the function clause to the graph.

The function is used in `perform_refactoring/8`.

insert_parameter_with_parenthesis/4 Insert parameters into the graph with parenthesis.

- Parameter description:
  1. NameToken: The created name token.
  2. PatternToken: The created pattern token nodes.
  3. Text: An atom, which will be the edge label.
  4. From: The application or the function node.
- Implementation:
  - Creates an open and a close parenthesis. Links them to From.

  - Links the NameToken to the open parenthesis.
  - Then calls the `insert_parameter/4` function and links its return value to the close parenthesis.

The function is used in `token_update/13`.

insert_parameters/4 Inserts the parameters to the application and the function definition.

- Parameter description:
  1. After: The last token.
  2. List: List of the pattern nodes.
  3. Text: An atom, which will be the edge label.
  4. From: The application or the function node.
- Implementation:
  - If the list is empty it returns with After token.
  - Otherwise it creates a comma, links it with the head of the list, and links it to From with Text label.
  - Links the After token to the head of the list with next label, then calls the `insert_parameters/4` function with the tail of the list and the After token will the head if the list.

The function is used in `token_update/13`.

error Handles error in refactorings. Currently it throws an exception, with the given parameters.

- Parameter description:
  1. Error: A two element tuple containing the type of the error, and a message to be able to make a better error message.
- Implementation:
  - Throws the error message.

The function is used in `get_module_and_source_node/1`,
`check_is_legal_body/4`,`check_is_function_name/1`,
`check_the_name_already_exists/3`,`check_the_name_is_imported/3`,
`check_all_var_bound_ok/3` and
`check_not_in_patter_guard_macro_listgen/2`.

3.2 Testing the function and results

Testing the tool via Emacs is fast and simple. I tested the extract function individual test cases. The test cases are produced by the other project members in order to able to test the refactor tool.

The format of test cases was designed to allow using the test data to do automatic testing.

We have to type of test case:

1. Accept: The transformation are allowed.
2. Deny: The transformation is not appropriate, so the extraction will not be done.

The tool work properly with the test cases and gave the expected result.

3.3 Extract function test cases

3.3.1 Extract sort/2 from gcd.

The name for the function to be created is “sort”. The refactoring is acceptable, the result is shown below.

```
-module(ef1).
-export([gcd/2]).




%tail-recursive definition of gcd

gcd(A,A) -> A;
gcd(A,B) -> {X,Y} = if
                        A > B -> {A,B};
                        true  -> {B,A}
                    end,
          gcd(X-Y,Y).
```

```
-module(ef1).
-export([gcd/2]).

sort(A,B) -> if
                 A > B -> {A,B};
                 true  -> {B,A}
             end.

%tail-recursive definition of gcd

gcd(A,A) -> A;
gcd(A,B) -> {X,Y} = sort(A,B),
          gcd(X-Y,Y).
```

Discussion. An if-expression is assigned to a variable. This expression is extracted into a function, and replaced with the invocation of that function. The if-expression does not bind any variables.

3.3.2 Function name conflicts with another function defined in the same module.

The name for the function to be created is “sort”. The refactoring violates the conditions, it is not acceptable.

```
-module(ef2).
-export([gcd/2]).

%tail-recursive definition of gcd

gcd(A,A) -> A;
gcd(A,B) -> {X,Y} = if
                        A > B -> {A,B};
                        true  -> {B,A}
                    end,
            gcd(X-Y,Y).

sort(A,B) -> if
                 A > B -> {A,B};
                 true  -> {B,A}
             end.
```

Discussion. Similar to ef1, but this time FunName (the name of the function to be produced by the refactoring) conflicts with another function, sort/2, defined in the same module. Another function name should be requested from the user, or a compensation (rename function) should be proposed.

3.3.3 *Function name conflicts with an imported function.*

The name for the function to be created is “sort”. The refactoring violates the conditions, it is not acceptable.

```
-module(ef3).
-import(sorting,[sort/2]).
-export([gcd/2]).

%tail-recursive definition of gcd

gcd(A,A) -> A;
gcd(A,B) -> {X,Y} = if
                        A > B -> {A,B};
                        true  -> {B,A}
                    end,
            gcd(X-Y,Y).
```

Discussion. Similar to ef2, but this time FunName (the name of the function to be produced by the refactoring) conflicts with an imported function, sorting:sort/2. Another function name should be requested from the user, or a compensation (remove import) should be proposed.

3.3.4 *Extract dummy/2 from gcd.*

The name for the function to be created is “dummy”. The refactoring is acceptable, the result is shown below.

```
-module(ef4).
-import(sorting,[dummy/1]).
-export([gcd/2]).




gcd(A,A) -> A;
gcd(A,B) -> {X,Y} = if
                        A > B -> {A,B};
                        true  -> {B,A}
                    end,
             gcd(X-Y,Y).
```

```
-module(ef4).
-import(sorting,[dummy/1]).
-export([gcd/2]).

dummy(A,B) -> if
                  A > B -> {A,B};
                  true  -> {B,A}
              end.

gcd(A,A) -> A;
gcd(A,B) -> {X,Y} = dummy(A,B),


             gcd(X-Y,Y).
```

Discussion. Similar to ef1, but this time the name of the new function is the same, as the name of an imported function. In this case this is not a problem, because the two functions have different arities, therefore they do not conflict.

3.3.5 *Extract dummy/2 from gcd.*

The name for the function to be created is “dummy”. The refactoring is acceptable, the result is shown below.

```
-module(ef5).
-export([gcd/2]).




gcd(A,A) -> A;
gcd(A,B) -> {X,Y} = if
                        A > B -> {A,B};
                        true  -> {B,A}
                    end,
            gcd(X-Y,Y).

dummy(X) -> X.
```

```
-module(ef5).
-export([gcd/2]).

dummy(A,B) -> if
                  A > B -> {A,B};
                  true  -> {B,A}
              end.


gcd(A,A) -> A;
gcd(A,B) -> {X,Y} = dummy(A,B),


            gcd(X-Y,Y).

dummy(X) -> X.
```

Discussion. Similar to ef1, but this time the name of the new function is the same, as the name of another function defined in the same module. Similarly to ef4, in this case this is not a problem, because the two functions have different arities, therefore they do not conflict. However, a warning should be given to he user, because this transformation might cause unintended overloading.

3.3.6 Extracting function from guard.

The name for the function to be created is “is_zero”. The refactoring violates the conditions, it is not acceptable.

```
-module(ef16).
-export([gcd/2]).

gcd(A,B) when B==0 -> A;
gcd(A,B) -> gcd(B,A rem B).
```

Discussion. Subexpressions occurring in a guard sequence cannot be extracted into a function. See the fourth condition of this refactoring.

3.3.7 *Extracting a block-expression.*

The name for the function to be created is “h”. The refactoring is acceptable, the result is shown below.

```
-module(ef20).
-export([f/1]).

f(X) -> Y = X*X,
        begin
          Z = X-1,
          Y*Z
        end.
```

```
-module(ef20).
-export([f/1]).

f(X) -> Y = X*X,
        h(X,Y).

h(X,Y) -> begin
            Z = X-1,
            Y*Z
          end.
```

Discussion. If a block-expression is extracted into a separate function, the begin-end statement becomes unnecessary. The current implementation of the refactoring tool need not eliminate the begin-end statement.

3.3.8 Extracting a function from a macro definition.

The name for the function to be created is “h”. The refactoring violates the conditions, it is not acceptable.

```
-module(ef21).
-export([f/1]).

-define(INC, -1).

f(X) -> X?INC.
```

Discussion. The sixth condition of this refactoring forbids the use of the transformation on a subexpression of a macro definition. This test case explains why: the transformation would replace “-1” with “h()”, where the definition of h/0 would be “h() -> -1”. After macro expansion the definition of f/0 would contain the incorrect “Xh()” expression.

4. SUMMARY

4.1 Summary

In this thesis I presented the description of extract function refactoring.

At first I gave a short overview of the refactoring in general, and the already existing refactor tools and our project main goal.

In the second chapter I wrote a short installation guide for Windows and Linux and the using process. After I specify the extract function refactoring, the preconditions of the transformation and then I presented the algorithm for refactoring using flowchart diagrams.

In the last chapter I wrote a detailed development manual about the program.

4.2 Related work

I have implemented the extract function refactoring in the prototype refactoring tool too.

4.3 Differences

But working in the new graph representation is simple and perspicuous. In the graph there is more direct semantical information (over the syntactical information) than it was in the previous representation. Some of them, like the variable binding, make the implementation more easier. Instead of the complicated and sometimes not efficient database queries we can write simple or complex path expression, which make the source more perspicuous.

Another good help in the work is the visualisation. There is a graph printer in the tool. Under the implementation and the continuous testing it provides possibility to detect an anomaly quickly, or to realise a missing part, or just to know the structure of the graph.

One of the most important features is the refactor-friendly structure of the graph model and the semantical information's automatic reconstruction after the refactor. That defend us to make semantical mistakes and its complicated correction process.

LIST OF FIGURES

1.1 The structure of the tool 8

2.1 The customize menu in emacs 10
2.2 Setting the path in emacs (the refactorerl directory include the source of the refactor tool) 11
2.3 Extract function `sort/2` from function `gcd/2`. 14
2.4 The structure of the refac_extract_fun module 17
2.5 The structure of check_variables 18
2.6 Extract function `subsum/2` from function `sum/3`. 19
2.7 Extract function `dummy/2` from function `gcd/2`. 20

BIBLIOGRAPHY

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

[2] Martin Fowler's refactoring site. http://www.refactoring.com/.

[3] GNU Emacs homepage. http://www.gnu.org/software/emacs/.

[4] R. Szabó-Nacsa, P. Diviánszky, and Z. Horváth. Prototype environment for refactoring Clean programs. In *The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, July 1–4*, 2004.

[5] P. Diviánszky, R. Szabó-Nacsa, and Z. Horváth. Refactoring via database representation. In L. Csőke, P. Olajos, P. Szigetváry, and T. Tómács, editors, *The Sixth International Conference on Applied Informatics (ICAI 2004), Eger, Hungary*, volume 1, page 129.

[6] J. Armstrong, R. Virding, M. Williams, and C. Wikstrom. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[7] J. Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, December 2003.

[8] J. Barklund and R. Virding. *Erlang Reference Manual*, 1999. Available from http://www.erlang.org/download/erl_spec47.ps.gz. 2007.06.01.

[9] Erlang homepage. http://www.erlang.org/.

[10] Li, H., Thompson, S.J., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T. *Refactoring Erlang Programs.* Accepted for 12th International Erlang/OTP User Conference,Stockholm, November 9-10, 2006.

[11] T. Nagy, A. Víg. *Storing Erlang source code in database.* Bachelor thesis, Faculty of Informatics, ELTE, Budapest, Hungary, February 2007.

[12] K. Beck. *Extreme Programming Explained.* Addison-Wesley, 2000. ISBN-0201-71091-9.

[13] H. Li. *Refactoring Haskell Programs.* PhD thesis, Computing Laboratory, University of Kent, Canterbury, United Kingdom, September 2006.

[14] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. *Haskell Workshop: Proceedings of the ACM SIGPLAN workshop on Haskell*, Uppsala, Sweden, 2003, pages 27–38.

[15] VIM Editor homepage. http://www.vim.org/. 2006.06.01.