



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

Refactoring Erlang programs with regards to energy efficiency

Supervisors:

Melinda Tóth, István Bozó
Assistant professor

Author:

Gergely Nagy
Computer Science Bsc

Budapest, 2019

Témabejelentő

Contents

| | | |
|----------|--|----------|
| 1 | Introduction | 4 |
| 1.1 | Motivation | 4 |
| 1.2 | Background | 5 |
| 1.3 | GreenErl | 6 |
| 1.4 | Problem description | 6 |
| 2 | User documentation | 9 |
| 2.1 | System requirements | 9 |
| 2.2 | Installation guide | 10 |
| 2.2.1 | Installing GCC | 10 |
| 2.2.2 | Installing Erlang | 11 |
| 2.2.3 | Installing GraphViz | 11 |
| 2.2.4 | Building RefactorErl | 12 |
| 2.2.5 | Installing Emacs | 12 |
| 2.2.6 | Setting up Emacs and RefactorErl | 13 |

| | | |
|----------|--|-----------|
| 2.3 | Using the environment | 14 |
| 2.3.1 | Using Emacs with RefactorErl | 14 |
| 2.4 | Applying refactorings | 17 |
| 2.4.1 | Eliminating <code>proplists:get_value</code> calls | 17 |
| 2.4.2 | Transforming list to map | 18 |
| 2.4.3 | Eliminating higher-order functions | 22 |
| 2.4.4 | Undo | 27 |
| 3 | Developer documentation | 28 |
| 3.1 | Problem description and design considerations | 28 |
| 3.1.1 | Integration to RefactorErl | 29 |
| 3.1.2 | Eliminating <code>proplists:get_value</code> | 30 |
| 3.1.3 | Transforming list to map | 30 |
| 3.1.4 | Eliminating higher-order functions | 32 |
| 3.2 | Algorithmic solution | 32 |
| 3.2.1 | Eliminating <code>proplists:get_value</code> calls | 33 |
| 3.2.2 | Transforming a list to map | 34 |
| 3.2.3 | Eliminating higher-order function calls | 44 |
| 3.3 | Implementation | 53 |
| 3.3.1 | Eliminate <code>get_value</code> | 56 |

| | | |
|----------|--|------------|
| 3.3.2 | Transforming a list to map | 58 |
| 3.3.3 | Eliminating higher-order functions | 76 |
| 3.4 | Testing | 88 |
| 3.4.1 | Unit testing | 89 |
| 3.4.2 | Testing energy consumption | 98 |
| 4 | Summary and Conclusions | 105 |
| 4.1 | Further development possibilities | 106 |
| | Bibliography | 106 |

Chapter 1

Introduction

In this chapter, I am going to present the background and motivation for my work and also give a description of the problem I solved in this thesis.

As global warming and climate change are becoming more and more ubiquitous, it is more and more important in every walk of life to become more energy conscious. This applies to software design and software development as well.

1.1 Motivation

In our TDK thesis [1], we presented a tool (GreenErl) for measuring the energy consumption of Erlang programs. With the help of this tool, we conducted measurements and investigated the energy consumption of Erlang language elements. We paid special attention to the energy consumption of different data structures and functional language-specific elements, such as higher-order functions.

As a conclusion of these measurements, we identified some possible ways to refactor Erlang programs in order to make them more energy efficient. Refactorings are important since with the transformations legacy codes can be restructured to be more energy efficient. The goal is to help the semi-automatic transformation of Erlang code bases.

In this thesis, I am going to present an implementation of the refactorings based on energy consumption measurements.

1.2 Background

Erlang

Erlang [2] is a widely used functional programming language with built-in support for concurrency and distributed systems. It has a dynamic type system, and the language is fault tolerant by design [3]. It was designed to be used for large scale, distributed telecommunication systems.

In my work, I rely on the data structures provided by Erlang, and I also use higher-order functions heavily, so I would like to present these language elements.

Data structures. The most basic and widely used data structure are lists. Lists can be used to store binary tuples representing key-value pairs. These constructs are called proplists. For the same purpose, there are some special associative data structures available in the language. These are maps and dictionaries. Erlang/OTP provides some modules (`lists`, `proplist`, `maps`, `dict`) for accessing and manipulating these data structures.

Higher-order functions. Higher-order functions are an important element in most functional programming languages. In Erlang, the most common higher-order functions, such as `map` and `filter` are available for use with lists in the `lists` module.

RefactorErl

RefactorErl [4, 5] is a static code analysis tool developed since 2006 at the Department of Programming Languages and Compilers at Eötvös Loránd University, Faculty of Informatics [6]. The tool can be used to analyze and refactor existing Erlang code bases.

RefactorErl represents an Erlang program with a directed, rooted graph, with typed nodes and edges, called the Semantic Program Graph (SPG). This representation is based on the abstract syntax tree of the program, but also contains lexical and semantic information in order to help refactoring [7] and code comprehension.

RefactorErl provides a way for developers to implement their own refactorings by traversing and modifying the relevant subtrees of the program graph.

1.3 GreenErl

The GreenErl framework is a tool for measuring the energy consumption of Erlang programs. The tool is based on RAPL [8], a power consumption management tool created by Intel. The tool provides an interface to measure the energy consumed by Erlang functions and a way to visualize and analyze the results.

1.4 Problem description

In our TDK thesis [1] we used the GreenErl tool to measure the energy consumption of Erlang software. The results of these measurements were used to propose transformations on Erlang source codes that possibly decrease energy consumption.

My task was to analyze each refactoring, identify the pre- and postconditions and create the algorithm for the transformations. I also had to implement these refactorings using the RefactorErl tool. The main objective of each refactoring is described

below.

Eliminating `proplists:get_value` calls

Using the `proplists:get_value/2` or `proplists:get_value/3` was found to be less energy efficient than using `lists:keyfind/3`. These functions provide slightly different interfaces for the same basic operation of looking up an element in a proplist.

The task of this refactoring is to make it possible to get rid of a call to the `proplists:get_value` function without any large scale transformation of how the surrounding code behaves.

Transforming list to map

Another possible refactoring to reduce energy consumption is to use maps instead of proplists, where possible. Previously, we proposed a refactoring to transform a recursive function definition that takes a proplist as its parameter to use a map instead.

The task of this refactoring is to make it possible for a developer to transform a recursive function, without having to rewrite the entire function definition. With this refactoring we must preserve the semantics of the function, only the underlying data structure can be transformed.

Eliminating higher-order function calls

Higher-order functions are an essential part of functional programming languages, and as such also of Erlang. Despite this, it was found that they provide some overhead in energy consumption.

The goal of this refactoring is to create a way for developers to eliminate calls to the most common (`lists:map`, `lists:filter`) higher-order functions. This elimination can happen in different ways, such as using list comprehensions or creating a recursive function. I let the users decide which method is the most suitable in their use case.

Chapter 2

User documentation

This chapter contains the user documentation of the refactorings and a guide on how to install the RefactorErl tool and its dependencies.

2.1 System requirements

The following requirements contain the setup on which the program has been tested and is guaranteed to work correctly.

Hardware requirements for installing and running the software:

- At least 512 MB of RAM
- At least 300 MB of free disk space for RefactorErl and its dependencies
- At least 200 MB of free disk space for Emacs
- There are no special requirements for the CPU and GPU

Software requirements:

- GNU/Linux operating system (the software has been tested using Ubuntu 16.04 LTS)

- Erlang/OTP 21.0 or newer
- GCC 4.7.2 or newer compiler for building RefactorErl
- Emacs 24.5 or newer for using the refactorings
- Optional: GraphViz 2.30.1 or newer for visualizing the SPG with RefactorErl

2.2 Installation guide

This guide contains a detailed description on how to install the dependencies of my software and how to setup Emacs to use the refactorings. To install all dependencies root access is needed.

2.2.1 Installing GCC

The GCC compiler for C and C++ is needed for some components of Erlang and for building the RefactorErl tool. It is recommended to use the package manager of the used operating system. On Ubuntu 16.04 LTS the following command installs the latest version of GCC:

```
sudo apt install g++
```

If you do not have access to a package manager, G++ can also be installed from the official GCC website [9]. This website also gives detailed instructions on how to install the software after downloading it.

After a successful installation the version of GCC can be verified using the following command:

```
gcc -version
```

2.2.2 Installing Erlang

Erlang is needed to build RefactorErl and also to compile refactored Erlang programs.

It can be installed in two ways. The recommended way is to use the package manager of your operating system. The following command installs the latest available version of Erlang:

```
sudo apt-get install erlang
```

It is also possible to install Erlang by downloading it from the official Erlang website [10] and following the instructions on that page.

After a successful installation Erlang can be run using the `erl` command in the terminal. On startup Erlang should display the version of the installed software in the following form:

```
Erlang/OTP 21 [erts-10.2.4]
```

2.2.3 Installing GraphViz

This step is optional and is only required for RefactorErl to create the SVG files that show a visual representation of the semantic program graph.

The latest GraphViz software can be downloaded from the GraphViz website [11] and installed using the instructions on that website. Another way to install GraphViz is to use the package manager software. The following command installs the latest version of the software:

```
sudo apt-get install graphviz
```

After a successful installation the `dot -V` command displays the version of the installed software.

2.2.4 Building RefactorErl

The official release of RefactorErl can be downloaded from the RefactorErl website [6], but this version does not yet contain the refactorings described in this thesis.

To get the version of RefactorErl containing the refactorings defined in this work, one should unzip the `source.zip` file from the attached CD. After unzipping, a directory called `trunk` and a file named `.emacs` should appear. For building RefactorErl only the `trunk` directory is needed.

Copy the `trunk` folder to the directory in which RefactorErl needs to be placed. Open a terminal from the `trunk/tool` directory. To build the RefactorErl tool, execute the following command from the `tool` directory:

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

This command builds the tool, it should take a few minutes to complete. This step requires the Erlang and GCC dependencies to be installed already.

2.2.5 Installing Emacs

To install Emacs, the text editor which can be used to execute the refactorings, follow the instructions on the Emacs website [12].

Emacs can also be installed using the package manager software, the following the command installs the latest available version of Emacs:

```
sudo apt-get install emacs
```

After a successful installation, the Emacs GUI can be started by executing the `emacs` command in the terminal.

2.2.6 Setting up Emacs and RefactorErl

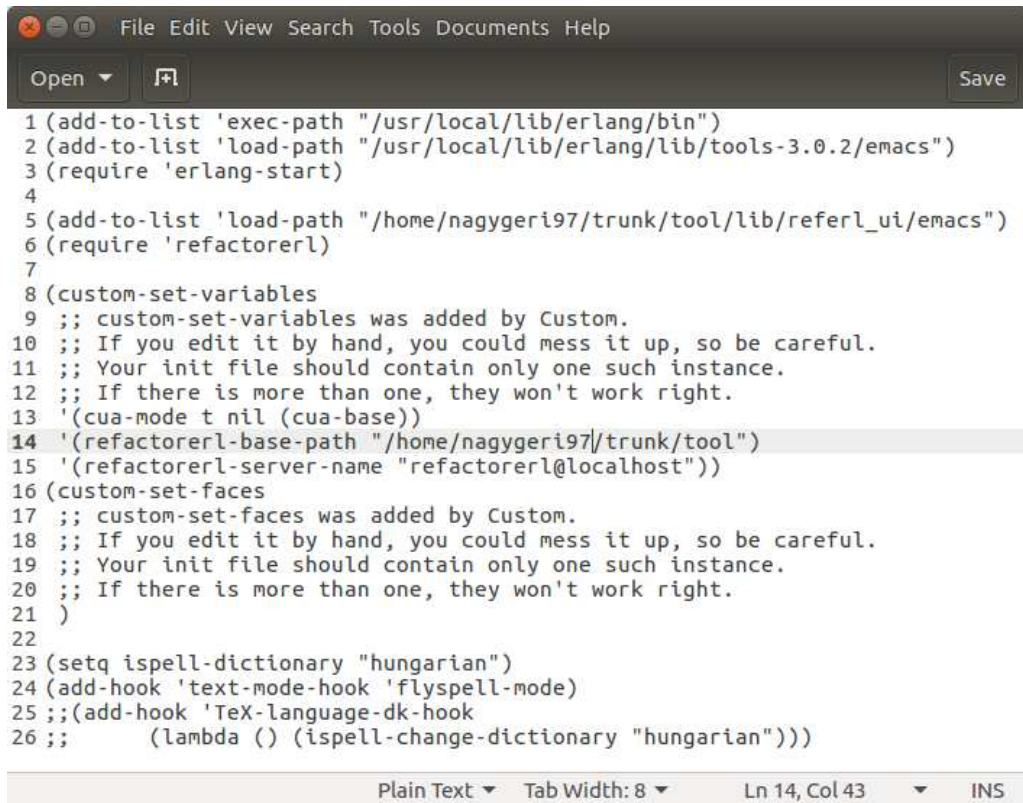
RefactorErl provides great support and integration with Emacs. It is possible to set Emacs up to execute refactorings by selecting the code to refactor and the drop down menu or keyboard shortcuts to perform the refactorings.

To make this integration possible, Emacs needs to be set up to use RefactorErl. This is achieved by the `.emacs` file. This file can be found when unzipping the `source.zip` file from the attached CD. Make sure that hidden files are shown as well, otherwise the `.emacs` file might not show up.

This `.emacs` file needs to be copied to the home directory of the user. An example of a correctly setup `.emacs` file is shown on Figure 2.1. Some editing of this file is required in order to work on the specific computer. In lines 5 and 14, the text `"PATH_TO_TRUNK"` should be replaced with the actual path to where RefactorErl is installed. If it is installed in the home directory, it should be replaced with `/home/username`, where `username` is the username of the actual user performing the installation.

In line 1 `"/usr/local/lib/erlang/bin"` should be replaced with the actual path to the Erlang binary file. For most standard Erlang installation this value should be correct, but make sure to check if it is actually the path to the Erlang binary.

In line 2 `"/usr/local/lib/erlang/lib/tools-3.0.2/emacs"` should be replaced with the path to the `emacs` folder of the Erlang lib. This is the folder that contains the `erlang.el` Lisp file. For most installations of Erlang this should also be correct, however the exact version number of `tools-x.y.z` may differ. Make sure that the correct path is specified in this line.



```
1 (add-to-list 'exec-path "/usr/local/lib/erlang/bin")
2 (add-to-list 'load-path "/usr/local/lib/erlang/lib/tools-3.0.2/emacs")
3 (require 'erlang-start)
4
5 (add-to-list 'load-path "/home/nagygeri97/trunk/tool/lib/referl_ui/emacs")
6 (require 'refactorerl)
7
8 (custom-set-variables
9  ;; custom-set-variables was added by Custom.
10 ;; If you edit it by hand, you could mess it up, so be careful.
11 ;; Your init file should contain only one such instance.
12 ;; If there is more than one, they won't work right.
13 '(cua-mode t nil (cua-base))
14 '(refactorerl-base-path "/home/nagygeri97/trunk/tool")
15 '(refactorerl-server-name "refactorerl@localhost"))
16 (custom-set-faces
17  ;; custom-set-faces was added by Custom.
18  ;; If you edit it by hand, you could mess it up, so be careful.
19  ;; Your init file should contain only one such instance.
20  ;; If there is more than one, they won't work right.
21 )
22
23 (setq ispell-dictionary "hungarian")
24 (add-hook 'text-mode-hook 'flyspell-mode)
25 ;;(add-hook 'TeX-language-dk-hook
26 ;;      (lambda () (ispell-change-dictionary "hungarian"))))
```

Figure 2.1: The .emacs used with RefactorErl placed in my home directory

2.3 Using the environment

This section contains a description on how to use RefactorErl with Emacs and how to make an existing Erlang source file ready for refactoring.

2.3.1 Using Emacs with RefactorErl

Start Emacs using from a terminal using the `emacs` command. To use RefactorErl with Emacs, it also needs to be started. From a separate terminal, move to the `trunk/tool` directory of RefactorErl. To start RefactorErl, use the following command:

```
bin/referl -db kcmuni
```

The `-db kcmuni` option specifies which database manager to use. The default database used is `mnesia` [13]. All refactorings work with Mnesia as well, but using

`kcmmini` (Kyoto Cabinet) [14] provides faster execution of refactorings. For further information about this option and any other options available, please refer to the documentation of `RefactorErl` [15].

Emacs shortcut notation. Emacs provides lots of keyboard shortcuts. These shortcuts use the `Ctrl` and `Alt` keys. To denote these notations, instead of, for example, `Ctrl + x` it is common to use the `C-x` notation. Similarly instead of `Alt + x`, the notation `M-x` is used.

Enabling RefactorErl mode. After opening an Erlang source file with Emacs, `RefactorErl` mode needs to be enabled. This can be activated using the keyboard shortcut `M-x` and typing the `refactorerl-mode` command. After this command a `Refactor` option will appear on the menu bar. This menu item is shown on Figure 2.2.

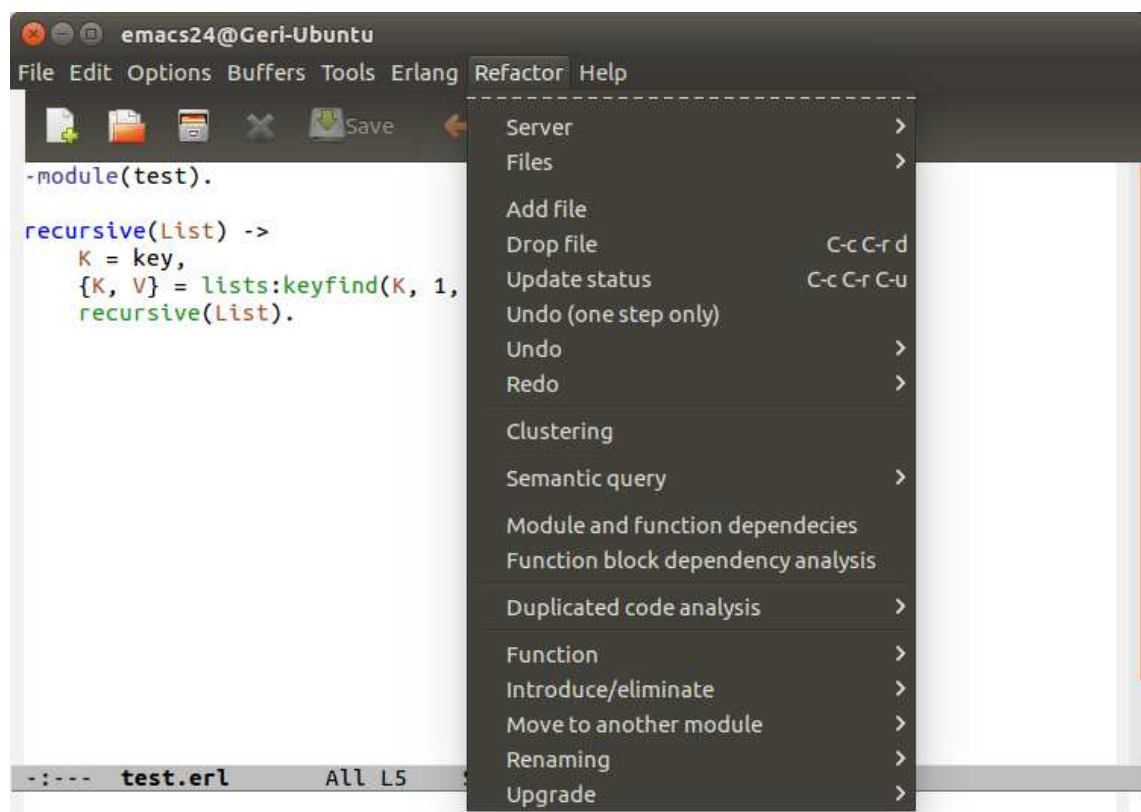
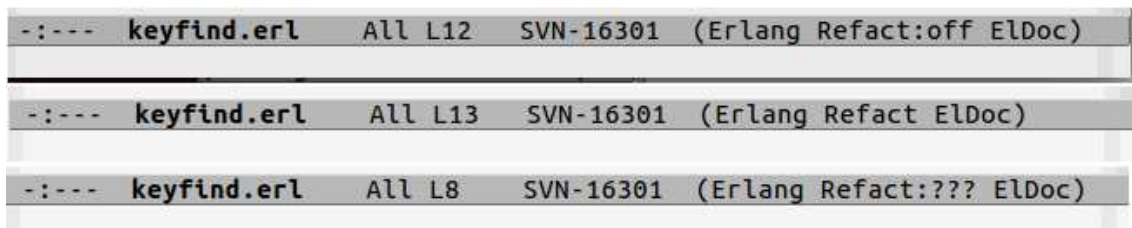


Figure 2.2: The menu item appearing in Emacs after `RefactorErl` mode is enabled.

Making a file ready for refactoring. When RefactorErl mode is enabled, a file does not automatically become ready for refactoring. The following steps need to be performed to make a file ready for refactoring. All steps can be done through the Refactor menu:

- Optionally use "Refactor > Files > Reset database" to clear all existing files from the database. This step ensures that the refactoring will not conflict with files already added to the database.
- "Refactor > Update status" to make sure the file is not yet added to the database
- "Refactor > Add file" to add the file to the database
- In some cases Emacs may not recognize that the file was added to the database. In this case use "Refactor > Update status" to make the file refactorable. Figure 2.3 shows how to check if this last step needs to be taken.



```
 -:--- keyfind.erl All L12 SVN-16301 (Erlang Refract:off ElDoc)
 -:--- keyfind.erl All L13 SVN-16301 (Erlang Refract ElDoc)
 -:--- keyfind.erl All L8 SVN-16301 (Erlang Refract:??? ElDoc)
```

Figure 2.3: This figure shows the information Emacs presents in different scenarios at the bottom of the window. The last part of the text, in parenthesis, is the difference between the cases. In the first case the file is not added to the database, in the second case the file is added to the database and in the last case Emacs does not know if the file has been added to the database or not. In the last case, the final, optional step of making a file ready for refactoring needs to be performed.

After these steps all applicable refactorings can be performed on the file.

2.4 Applying refactorings

This section contains a description for each proposed refactoring, that may help decrease energy consumption. For each refactoring the applicability and preconditions assumed by the refactoring are also listed. For the usage of all refactorings Emacs needs to be set up with RefactorErl.

This section only aims to help users of these refactorings to use the refactorings in the correct manner and understand how each refactoring can be invoked. For a detailed view of how code transformations work, and which constructs get transformed to which constructs, see the Developer Documentation in Chapter 3.

2.4.1 Eliminating `proplists:get_value` calls

This refactoring can be used to transform calls to the `proplists:get_value/2` and `proplists:get_value/3` functions to the more energy efficient `lists:keyfind/3` function. This transformation can be applied to any call to `get_value` function.

Preconditions. There are no preconditions to this refactoring. If the refactored call to `proplists:get_value` is not semantically correct the resulting refactored code will also be incorrect, but all semantically correct calls are transformed so that the resulting code emulates the behaviour of `get_value`.

Usage. To use this refactoring the following steps need to be executed:

- Position the cursor over the name of the `get_value` function call that needs to be refactored (Figure 2.4).
- Either use the Refactor menu to select "Refactor > Introduce/Eliminate > Eliminate `proplists:get_value`" or use the keyboard shortcut "C-e C-r e g" (Figure 2.5).

The end result of using this refactoring can be seen on Figure 2.6. Successful refactoring is denoted by the "RefactorErl: transformation done." message at the bottom of the screen. If any error happens during refactoring the error message will be shown at the same place as the message of success.

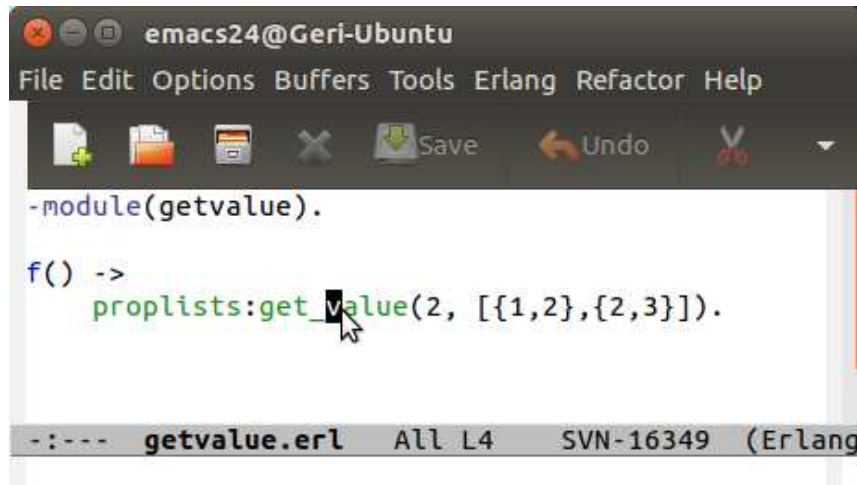


Figure 2.4: Step 1 of using the "Eliminate proplists:get_value" refactoring

2.4.2 Transforming list to map

The goal of this refactoring is to transform a recursive function, whose parameter is a proplist to instead take a map as a parameter.

The body of the function is limited to use the list in only certain function calls. These function calls are:

- o lists:keyfind/3
- o lists:keydelete/3
- o lists:keymember/3
- o lists:keystore/4
- o lists:keytake/3

In addition to these function calls, the list can also stand by its own at the end of a clause, meaning it is the value returned from the function.

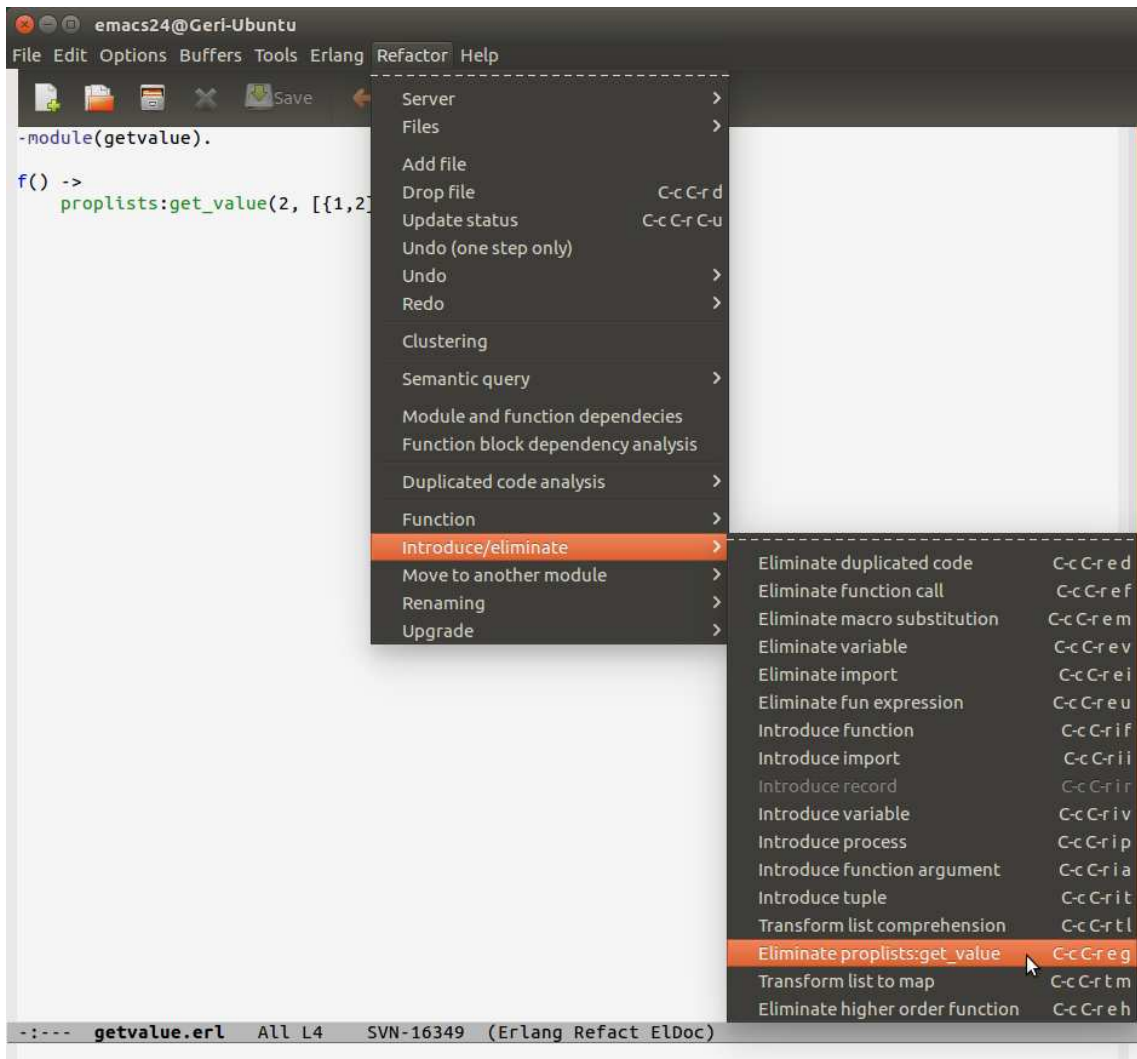


Figure 2.5: Step 2 of using the "Eliminate proplists:get_value" refactoring

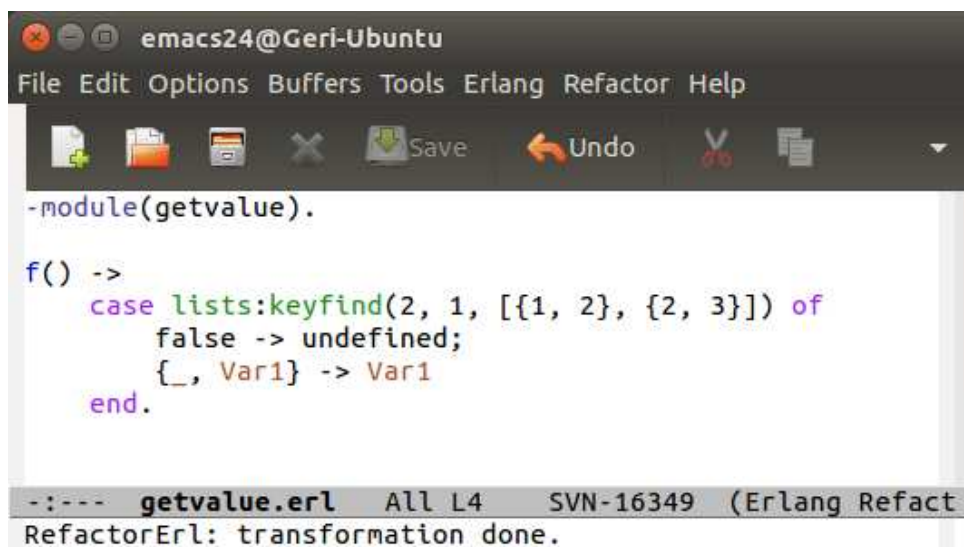


Figure 2.6: Result of using the "Eliminate proplists:get_value" refactoring

If the list is used in any other context, it will result in an error during refactoring.

All clauses of the recursive function must contain either a single variable name at the position of the list variable, or the empty list pattern matching expression (`[]`). Any other kind of pattern matching to the list will result in an error during refactoring.

Preconditions. This refactoring assumes several preconditions, some of which can be checked by the refactoring tool, but others cannot be checked because of the dynamic type system of Erlang.

The checked preconditions are:

- The function must be recursive. The tool can only check if a recursive call happens on any branch of the function definition, but for maximal gains in energy efficiency the function should contain the recursive call so that in most cases it gets called.
- The selected variable must be a parameter of the function.
- The list variable should only be used in the functions and contexts mentioned above.
- The functions mentioned above all need a parameter specifying which part of the tuple in the list denotes the key. This parameter can only be the integer literal '1'.
- The result of `lists:keystore/4` and `lists:keydelete/3` can only be bound to a variable, no pattern matching is allowed, so that all occurrences of the modified list can also be transformed.
- The result of `lists:keytake/3` can only be bound to a ternary tuple, so that the modified list can be transformed as well.

The refactoring relies on the user to make sure that all unchecked preconditions are met before refactoring. If some of the unchecked preconditions are not true,

the refactoring cannot guarantee that the resulting program will behave correctly. These unchecked preconditions are:

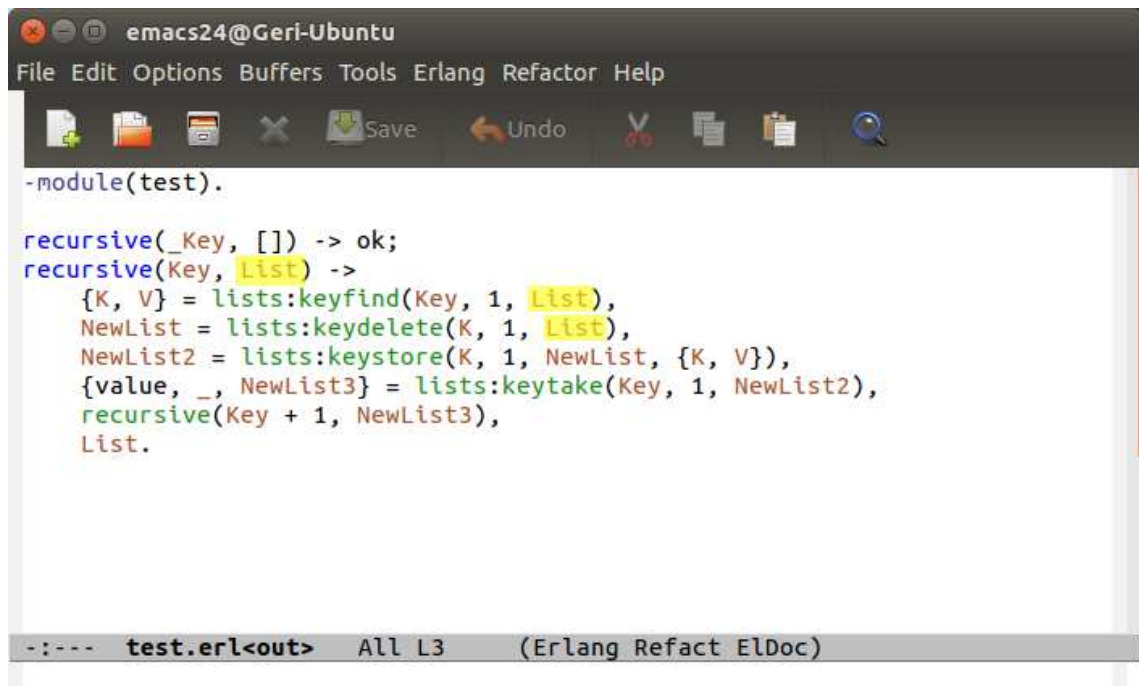
- The selected variable must be a list of binary tuples, whose first element is the key, while the second element of the tuples is the value belonging to that key.
- All keys in the proplist must be unique at all times, even at arbitrary points during the execution of the recursive function. This is needed because maps only support unique keys.
- The ordering of the elements in the list has no significance. The reason for this is that maps are unordered and any meaning in the ordering of the elements would get lost in the process of transformation.

Usage. The usage of this refactoring is mostly similar to how the previous refactoring can be used. It consists of the following steps:

- Position the cursor over the name of the variable that contains the list and needs to be transformed to a map. Any occurrence in any clause of the function is sufficient, even the parameter of the recursive function can be selected. Figure 2.7 shows all the possible instances of the variable that can be selected.
- Either use the Refactor menu to select "Refactor > Introduce/Eliminate > Transform list to map" or use the keyboard shortcut "C-e C-r t m" (Figure 2.8).

The correct result for refactoring the code shown on Figure 2.7 is shown on Figure 2.9.

Any error during the refactoring is a result of a checked precondition failing. In case of an error read the error message and make sure the referenced precondition is met.



```
emacs24@Geri-Ubuntu
File Edit Options Buffers Tools Erlang Refactor Help
Save Undo
-module(test).

recursive(_Key, []) -> ok;
recursive(Key, List) ->
  {K, V} = lists:keyfind(Key, 1, List),
  NewList = lists:keydelete(K, 1, List),
  NewList2 = lists:keystore(K, 1, NewList, {K, V}),
  {value, _, NewList3} = lists:keytake(Key, 1, NewList2),
  recursive(Key + 1, NewList3),
  List.
```

test.erl<out> All L3 (Erlang Refact EDoc)

Figure 2.7: All possible instances of the List variable that can be selected for the transformation to map to work correctly.

2.4.3 Eliminating higher-order functions

This refactoring provides a possibility to eliminate calls to the higher-order functions `lists:map/2` and `lists:filter/2`. With each execution of the refactoring a single call to either the `map` or the `filter` higher-order function gets eliminated. The call of the higher-order function can be replaced either with a list comprehensions or a call of a recursive function. The used method is the choice of the user. The refactoring supports eliminating higher-order function calls that use an implicit fun or a lambda passed as an argument to the function.

Preconditions. The only precondition for this refactoring is that the selected function should be a call to one of the higher-order functions `lists:map/2` or `lists:filter/2`. This is checked during the refactoring and an error is shown if anything else is selected.

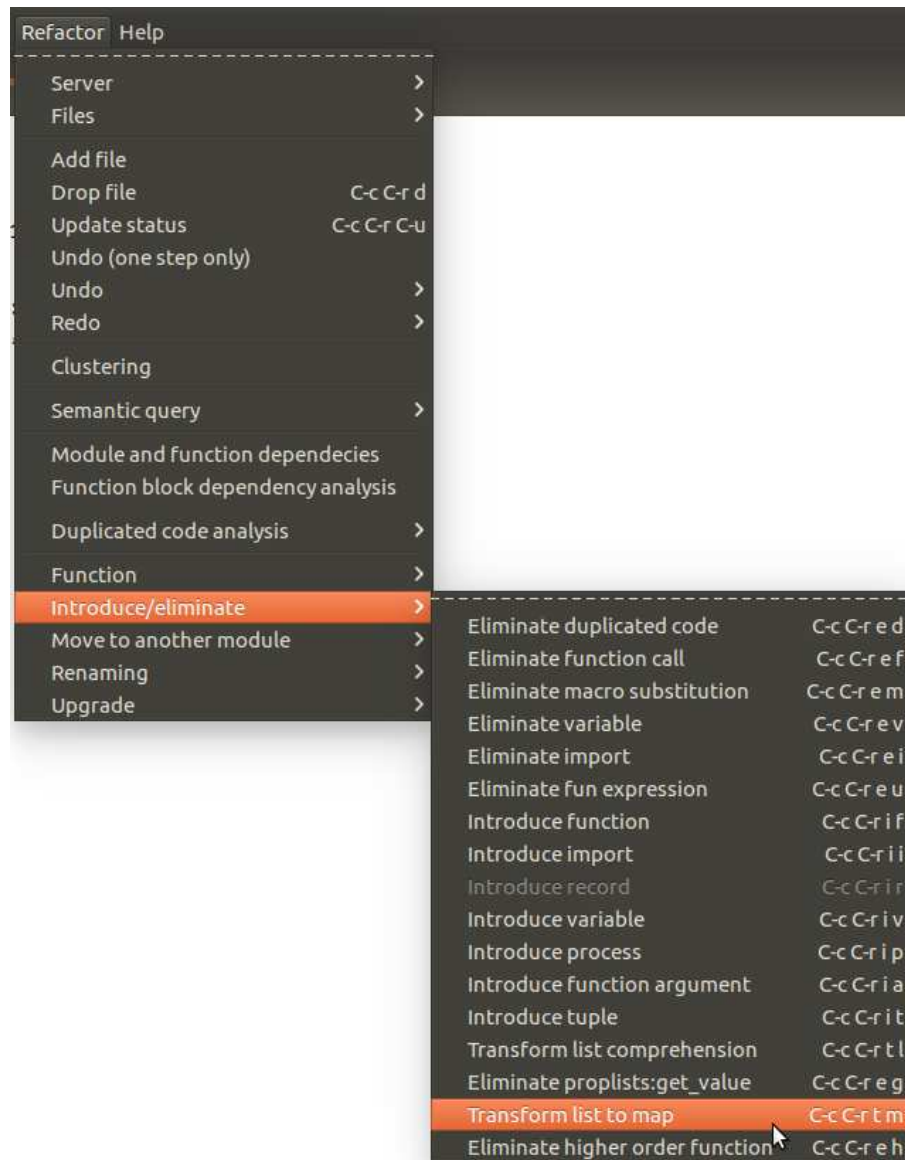
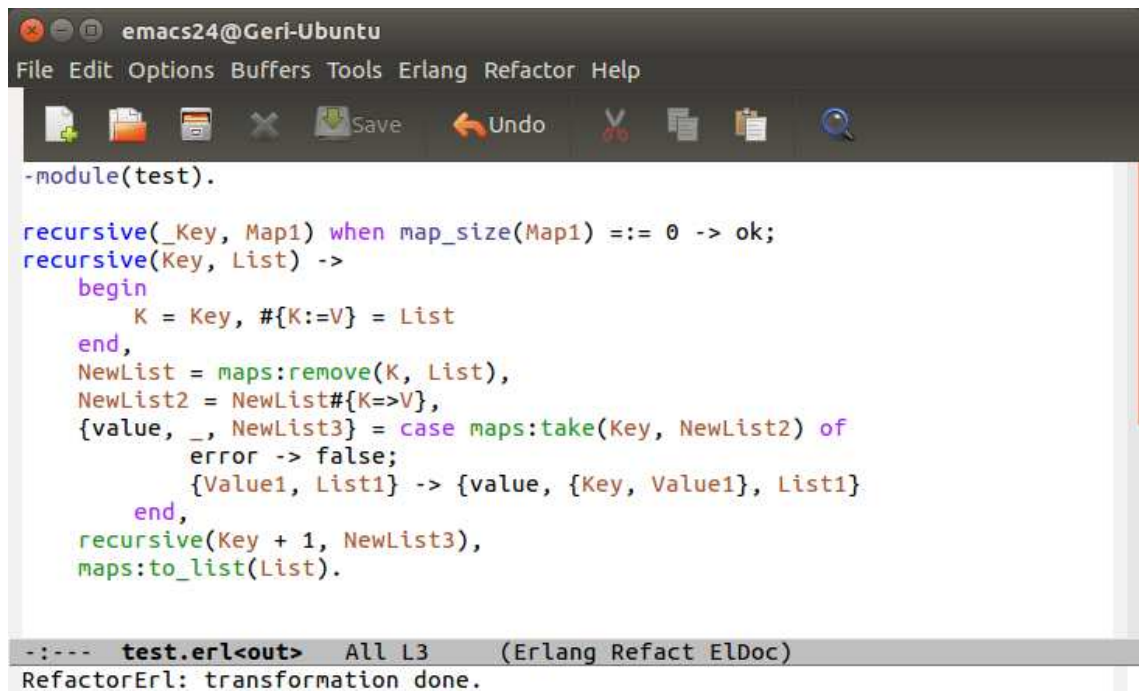


Figure 2.8: The menu option to perform the transformation from list to map.



```
-module(test).

recursive(_Key, Map1) when map_size(Map1) == 0 -> ok;
recursive(Key, List) ->
  begin
    K = Key, #{K:=V} = List
  end,
  NewList = maps:remove(K, List),
  NewList2 = NewList#{K=>V},
  {value, _, NewList3} = case maps:take(Key, NewList2) of
    error -> false;
    {Value1, List1} -> {value, {Key, Value1}, List1}
  end,
  recursive(Key + 1, NewList3),
  maps:to_list(List).
```

RefactorErl: transformation done.

Figure 2.9: The result of the "Transform list to map" refactoring.

Usage. The usage of this refactoring is similar to both of the refactorings shown before. The followings steps are needed:

- o Position the cursor over the name of the higher-order function call that needs to be refactored, similarly to previous refactorings.
- o From the menu bar, select "Refactor > Introduce/eliminate > Eliminate higher order function" or use the "C-e C-r e h" keyboard shortcut. (Figure 2.10)
- o If everything is correct, a dialog should appear, prompting the user to decide the mode of transformation. Three choices are available, one for using list comprehension, one for using recursion and one for letting the refactoring decide. This automatic decision by the refactoring is described in detail below. (Figure 2.11)
- o After clicking "Submit" the refactoring should complete, while after clicking "Cancel" the refactoring will be canceled and no change will be done to the source code.

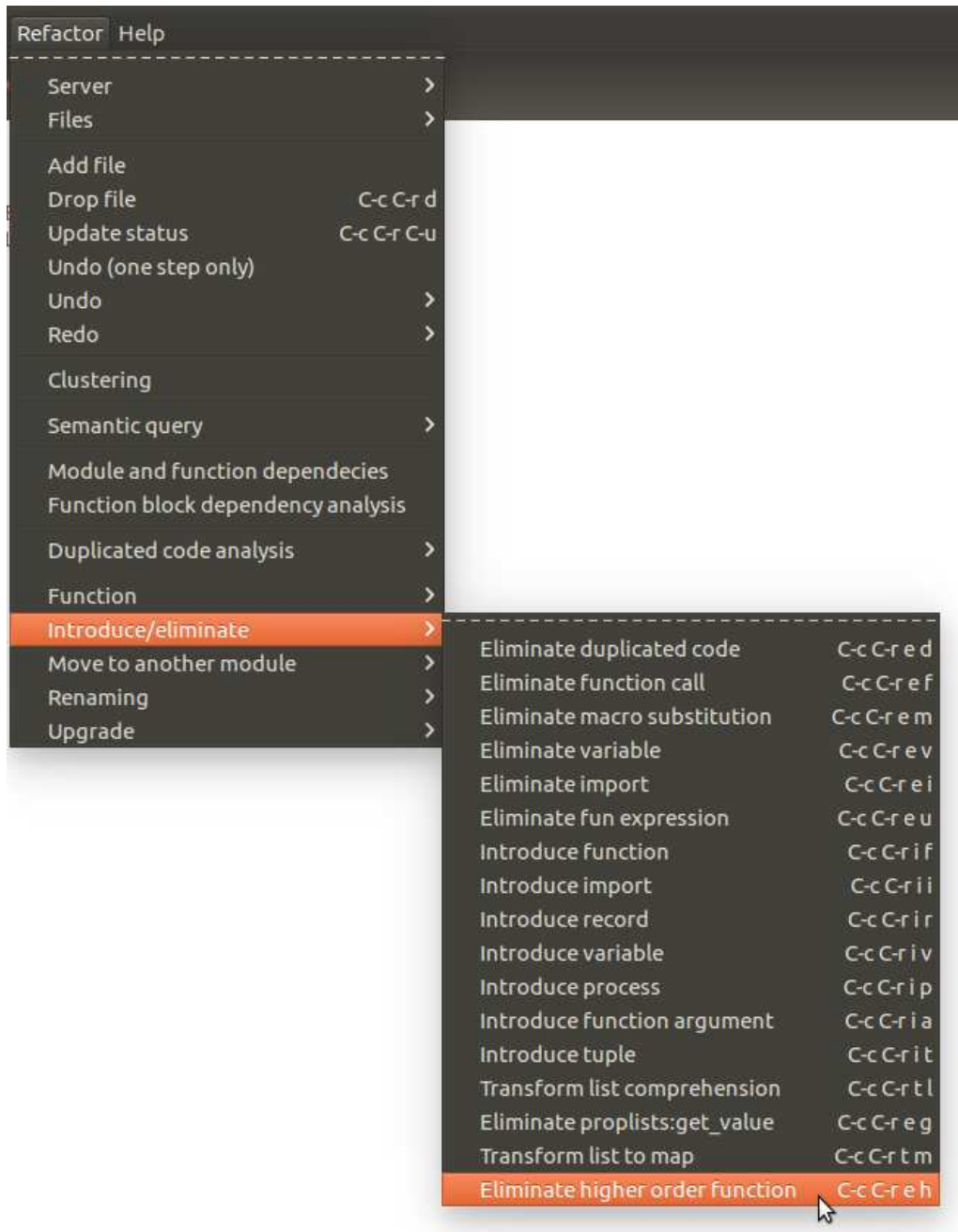


Figure 2.10: The menu item corresponding to the "Eliminate higher order function" refactoring

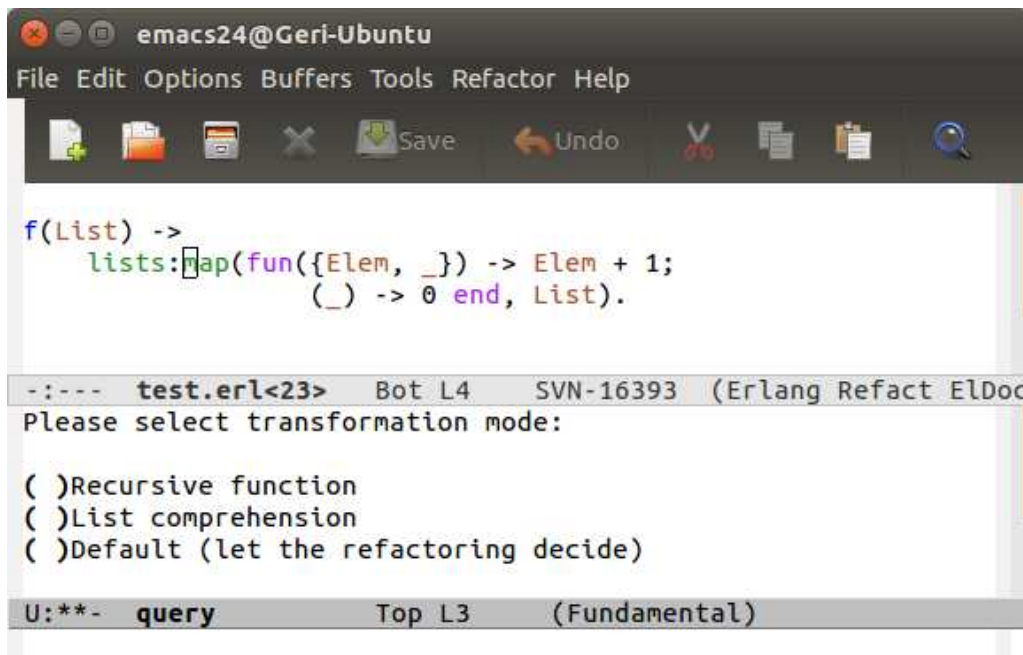


Figure 2.11: Dialog to decide the mode of eliminating higher-order function calls

The result of choosing to use a list comprehension is shown in Figure 2.12, while the result of using a new, recursive function definition for the elimination is shown on Figure 2.13.

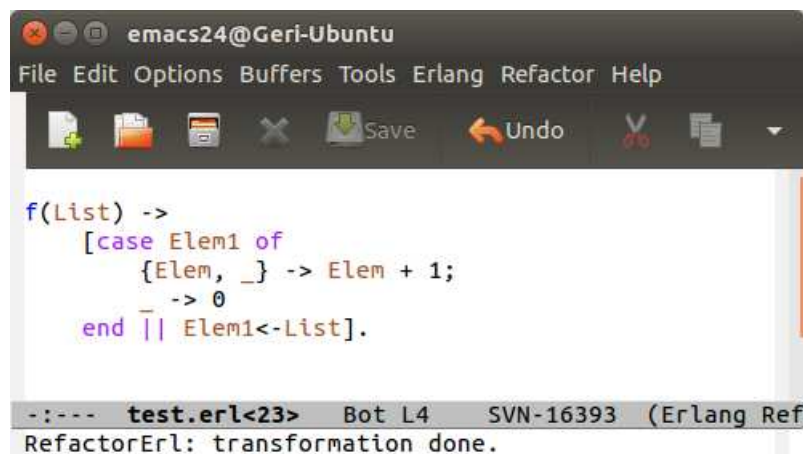
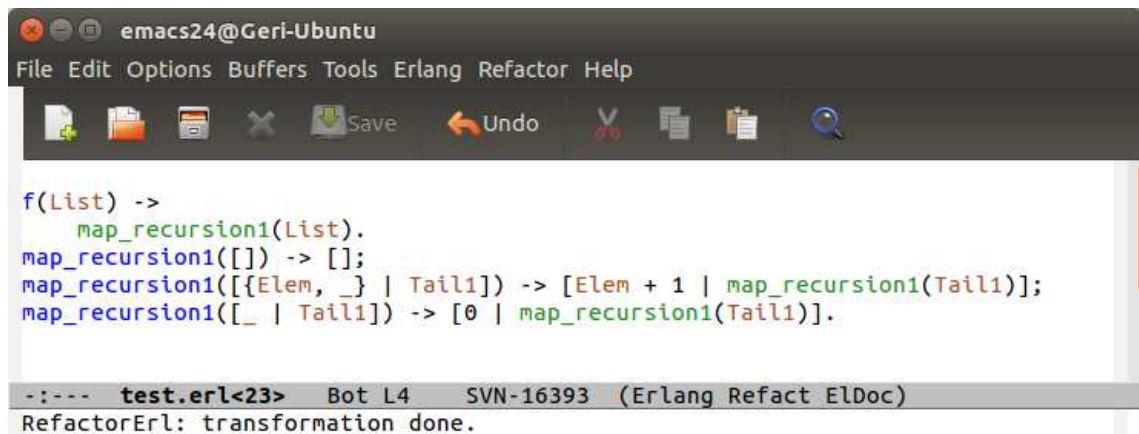


Figure 2.12: Result of using a list comprehension to eliminate a call to the higher-order function `lists:map/2`

Deciding the method of transformation. If the user does not want to decide which method to use for transformation, the algorithm offers some basic deductions to choose the more suitable form of transformation. Since using list comprehensions is sensible and maintainable if the amount of code written in them is minimal,

The image shows a screenshot of the Emacs24@Geri-Ubuntu window. The title bar reads "emacs24@Geri-Ubuntu". The menu bar includes "File Edit Options Buffers Tools Erlang Refactor Help". The toolbar contains icons for Save, Undo, and other standard Emacs actions. The main text area displays the following Erlang code:

```
f(List) ->  
    map_recursion1(List).  
map_recursion1([]) -> [];  
map_recursion1([{Elem, _} | Tail1]) -> [Elem + 1 | map_recursion1(Tail1)];  
map_recursion1([_ | Tail1]) -> [0 | map_recursion1(Tail1)].
```

The status bar at the bottom shows "-:-- test.erl<23> Bot L4 SVN-16393 (Erlang Refact ElDoc)" and a message "RefactorErl: transformation done."

Figure 2.13: Result of using recursion to eliminate a call to the higher-order function `lists:map/2`

the refactoring decides to use list comprehensions when an implicit fun is used, or when the body of the lambda only contains a single expression. All other cases would result in a complex list comprehension so a new, recursive function is defined instead.

2.4.4 Undo

RefactorErl provides a way to undo the most recent refactoring performed. This can be accessed from the menu under "Refactor > Undo (one step only)". This results in a popup question to confirm the undo¹.

¹In some cases performing an undo may result in the file not being ready for refactoring. In this case the current file either needs to be removed and added again, or the entire database needs to be reset for refactorings to work correctly again.

Chapter 3

Developer documentation

This chapter contains the developer documentation of the refactorings I implemented. The structure of this documentation is as follows:

- First, the problem description and design considerations are presented for each implemented refactoring.
- Next, the algorithmic solution to the problem is presented.
- After that, the implementation is shown in detail.
- Finally, the method of testing and used test cases are described.

3.1 Problem description and design considerations

The problem that this thesis presents a solution to is to refactor Erlang code bases so that the energy consumption of the software decreases. Previously there were three refactorings identified based on measurements, that can improve energy efficiency [1]. This section provides an overview of the scope of the additions to RefactorErl and also how and in what way my work is integrated to the tool. I also present the basic idea behind these refactorings, as well as the most important considerations and design decisions for implementing the transformations.

3.1.1 Integration to RefactorErl

The structure of RefactorErl is shown on Figure 3.1. To implement a refactoring, it must be embedded in the "Refactoring" layer. This layer builds on the existing graph queries to extract information from the Semantic Program Graph (SPG). Creating the new syntax tree elements and performing the transformations are handled by the respective components of RefactorErl. It is also possible to perform data flow analysis with the tool to help refactoring.

To make the refactorings available to users, the UI layer has to be modified to contain the newly defined transformations. In my work I used the UI capabilities of Emacs, which makes it possible for the refactorings to appear in the Refactor menu.

The design of each refactoring follows the capabilities and requirements of the tool. A more detailed description of how refactorings can be implemented using RefactorErl is presented in Section 3.3.

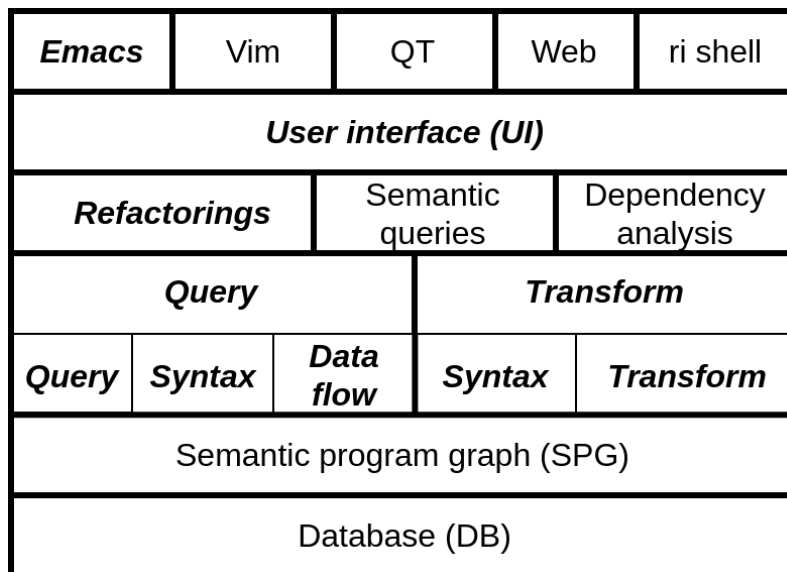


Figure 3.1: Structure of the RefactorErl tool with the used and modified components highlighted

3.1.2 Eliminating `proplists:get_value`

The goal of this refactoring is to replace calls to the `proplists:get_value` functions with calls to the more energy efficient `lists:keyfind/3` function.

Since the interface provided by the two functions is slightly different, the most important design decision in the case of this refactoring was to either transform the bindings of the result of `get_value` calls and make sure any subsequent code that uses the result is also transformed correctly, or at the place of the function call use a slightly more complex expression to emulate the value returned by `get_value` using `keyfind`. I chose the latter version, since the scope of changes would have been enormous using the former method, and it would have required heavy dataflow analysis. The second method limits the scope of the transformation to only the place of the function call and even though in some cases it results in more lines of code, it can better guarantee the correctness of the refactoring.

3.1.3 Transforming list to map

This refactoring aims to improve the energy efficiency of recursive functions that use a proplist (list of binary tuples) to store and manipulate key-value pairs. The transformation includes changing the underlying data structure from list to map and transforming all expressions that use a list to use maps instead.

Because of energy efficiency reasons it is important that the recursive function receives the proplist as a parameter, since then the conversion from list to map can be done on the calling side.

The design considerations in the case of this refactoring include selecting the expressions the list is allowed to appear in. For this selection I used the results presented in our TDK thesis [1]. The list is only allowed to be used in expressions where it makes sense to use a map (meaning that the expression does not rely on the ordering of the elements and has to do with using key-value pairs).

Of the functions that the `lists` library module provides the following can be transformed to use maps energy efficiently:

- `lists:keyfind/3`
- `lists:keydelete/3`
- `lists:keymember/3`
- `lists:keystore/4`
- `lists:keytake/3`

In the refactoring of these functions one possible solution was to use the functions in the `maps` module to simply emulate the behaviour of said functions, but this would have resulted in unmaintainable code. Instead, I chose to perform some context analysis and choose the most suitable alternative using maps. This means that depending on context, each call can be transformed in multiple ways.

The functions I considered for refactoring but decided not to transform them:

- `lists:keymap/3`
- `lists:keymerge/3`
- `lists:keysort/2`
- `lists:keysearch/3`
- `lists:keyreplace/4`

The reason for not refactoring these function calls is that some cannot be replaced with equivalent calls to functions on maps, because they rely on the ordering of the elements (`lists:keysort/2` and `lists:keymerge/3`), others are deprecated (`lists:keysearch/3`) while even others do not have a real, efficient alternative when using maps (`lists:keyreplace/4`). It would be possible to emulate the behaviour of `lists:keyreplace/4` by using maps, but the resulting code would be complex and inefficient.

3.1.4 Eliminating higher-order functions

The final refactoring I implemented has the goal to eliminate calls to some commonly used higher-order functions, specifically, the `lists:map/2` and `lists:filter/2` higher-order functions, because based on energy consumption measurements the energy efficiency can be increased by eliminating calls to these functions.

There are two methods by which these higher-order function calls can be eliminated. It is possible to create a list comprehension that behaves the same way the higher-order function would, or to introduce a new, specialized recursive function definition.

When designing this refactoring, I considered that the refactoring itself should choose the most suitable method of the two, but the final decision is that the user should decide, because they might have a better understanding of their code than a refactoring could have. However, the refactoring also provides an option for when the user cannot or does not want to decide on the method. In this case the refactoring uses some heuristics to decide which transformation to use.

Another consideration was that, concerning the use of implicit functions, if the definition of that function is available for the refactoring, should it be inlined where possible. The decision on this matter was that inlining would result in long, not readable, and most importantly, unmaintainable code, so the function definition should be kept separately when it is already given as a separate function.

3.2 Algorithmic solution

This section contains a description of the algorithms used to perform the desired transformations.

3.2.1 Eliminating `proplists:get_value` calls

To rewrite calls to `proplists:get_value` to calls to `lists:keyfind/3` I chose to emulate the behaviour of `get_value` by using `keyfind`. This means that even in cases when an error happens during the function call, the behaviour should be preserved.

The function `proplists:get_value/2` returns only the value corresponding to a key if the key exists, and returns the atom `undefined` otherwise [16]. On the other hand, `lists:keyfind/3` returns the tuple `{Key, Value}` if the key exists and the atom `false` otherwise [17].

The way to emulate the behaviour of `proplists:get_value/2` using the more efficient `lists:keyfind/3` is shown on Figure 3.2.

```
1 proplists:get_value(Key, List)
```

(a) Before

```
1 case lists:keyfind(Key, 1, List) of  
2   false -> undefined;  
3   {_, Var} -> Var  
4 end
```

(b) After

Figure 3.2: Refactoring for eliminating `proplists:get_value/2` calls.

There is also the `proplists:get_value/3` function, which takes one extra argument, which specifies the default value returned if the key does not exist in the list. This function can be refactored to be replaced by `lists:keyfind/3` in the same way, as shown in Figure 3.2, we only need to replace `undefined` with the specified default value.

3.2.2 Transforming a list to map

This refactoring is able to identify a recursive function, determine if the selected variable is in fact a parameter of that function, and if the variable is only used in a way that makes sense for us to use a map instead.

There are also some additional constraints, which cannot be checked because of the dynamic type system of Erlang. These are the following:

- The list has to be a proplist
- The first element of the binary tuples must be the key
- The order of the elements in the list must not matter
- Keys must be unique in the proplist

In the following paragraphs the algorithms for the transformation of calls to the functions mentioned in Section 3.1.3 are shown.

lists:keyfind/3. `lists:keyfind/3` returns the `{Key, Value}` tuple if the key is in the list, otherwise returns the atom `false`. Since the `#{Key := Value} = Map` pattern matching expression throws an exception if the key is not present in the map, this pattern matching can only be used in a limited number of cases, depending on context.

These contexts are when the result of the `keyfind` call is bound to a binary tuple, since if the key is not present, this would also result in a `badmatch` exception. In all other contexts, the `maps:find/2` function must be used to emulate the behaviour of `lists:keyfind/3`.

Besides, whether the statement is the last statement in the function clause also needs to be checked, since in that case the tuple, that `lists:keyfind/3` would have returned needs to be returned, thus `maps:find/2` must be used in all of these cases.

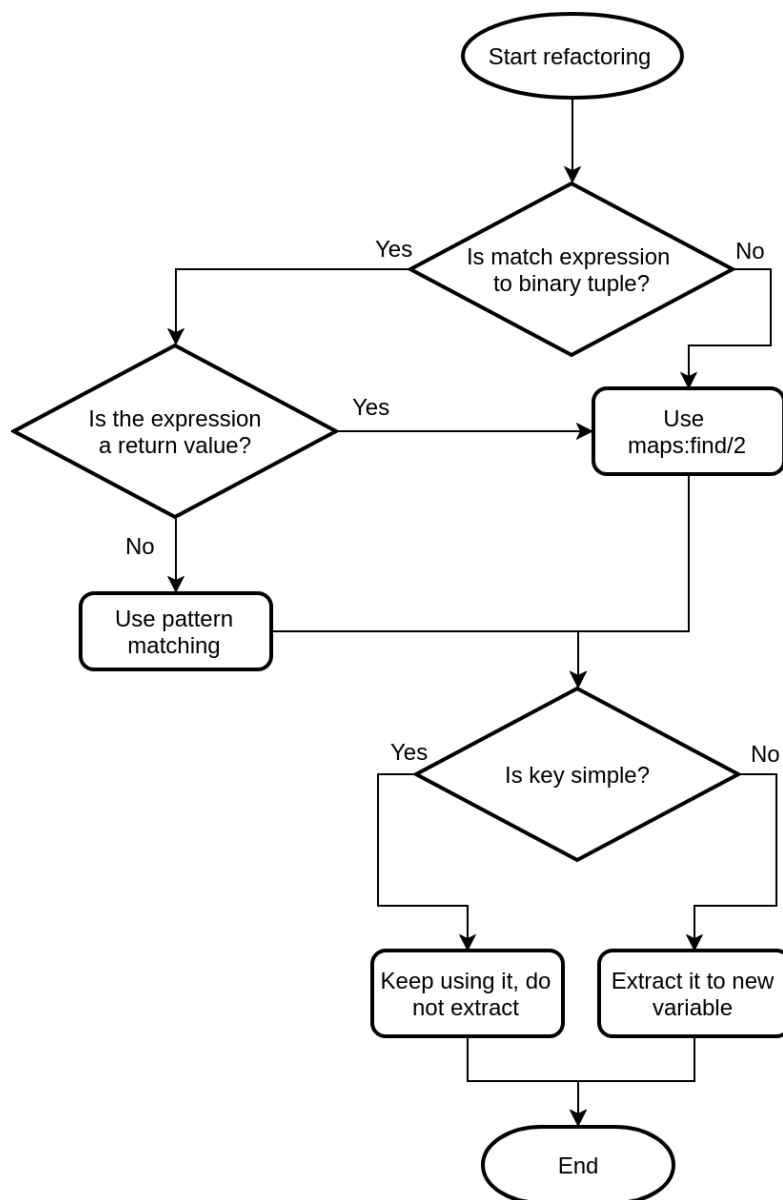


Figure 3.3: Flowchart of the decisions we need to make in order to transform a `lists:keyfind/3` call

```

1 % 1.) Transformed to pattern matching
2 {_, Value} = lists:keyfind(Key, 1, List)
3
4 % 2.) Transformed to maps:find/2
5 lists:keyfind(Key + 1, 1, List)

```

(a) Before

```

1 % 1.) Transformed to pattern matching
2 #{Key := Value} = List
3
4 % 2.) Transformed to maps:find/2
5 begin
6     Key1 = Key + 1,
7     case maps:find(Key1, List) of
8         error -> false;
9         {ok, Value} -> {Key1, Value}
10    end
11 end

```

(b) After

Figure 3.4: Refactoring for eliminating `lists:keyfind/3` calls.

Finally, it needs to be determined if the key is a simple key. A simple key is a key that it is a variable, integer literal or an atom. The goal of identifying simple keys is that the evaluation of simple keys is guaranteed to be a quick, lightweight operation without any side effects. There are other types which could be used as simple keys, but it would be complex to check the corresponding subtree. If the key is simple, it can simply be repeated whenever it is needed in the refactored version. However, if the key is complex, it needs to be extracted to a new variable, since the key can be a computationally intensive function call, or it can even have side effects when evaluated more than once.

In the case of multiple instructions, such as extracting the key to a new variable, the expressions need to be wrapped in a begin-end block, so that it can be treated as a single expression in all contexts. There are some contexts where this could be omitted, but it would require heavily context dependent transformations.

All of these decisions needed for the refactoring are presented with a flowchart on Figure 3.3. Some refactoring examples are also shown on Figure 3.4.

lists:keydelete/3. `lists:keydelete/3` returns a new proplist with the element corresponding to a given key removed. This is exactly the same functionality as with `maps:remove/2`, which returns a new map with the element belonging to the given key removed.

If the result of this expression is bound to a variable, all further occurrences of that variable have to be checked and transformed as well, since the bound variable is a map in the refactored version instead of a list. Because of the need for transformation in all expressions that use the result of this function call, the refactoring only allows the result of `keydelete` calls to be bound to a single variable, no pattern matching is allowed.

If this expression stands as the last statement in a clause, the result has to be transformed back to a list, in order to keep the return value of the function the same. This conversion is done using the `maps:to_list/1` function.

In the case of `keydelete` there is no need to extract the key to a new variable, even if it is not a simple key. The reason for this is that in every case, the key part of `keydelete` is only used once in the refactored version, so the behaviour stays the same, even for non-simple keys.

lists:keymember/3. Out of all the functions, `lists:keymember/3` is the easiest to transform, since it simply returns a boolean value denoting if the given key is present in the list or not. The `maps:is_key/2` function provides the exact same functionality, only with maps, so `keymember` calls simply need to be replaced with `is_key` calls.

There is no need to transform the return value in this case, since both functions return a boolean value with the same meaning. Also, similarly to `lists:keydelete/3`, the key does not need to be extracted to a variable.

lists:keystore/4. The `lists:keystore/4` function is used to update an already existing key-value pair, or to insert a new key-value pair to the list, if the key does

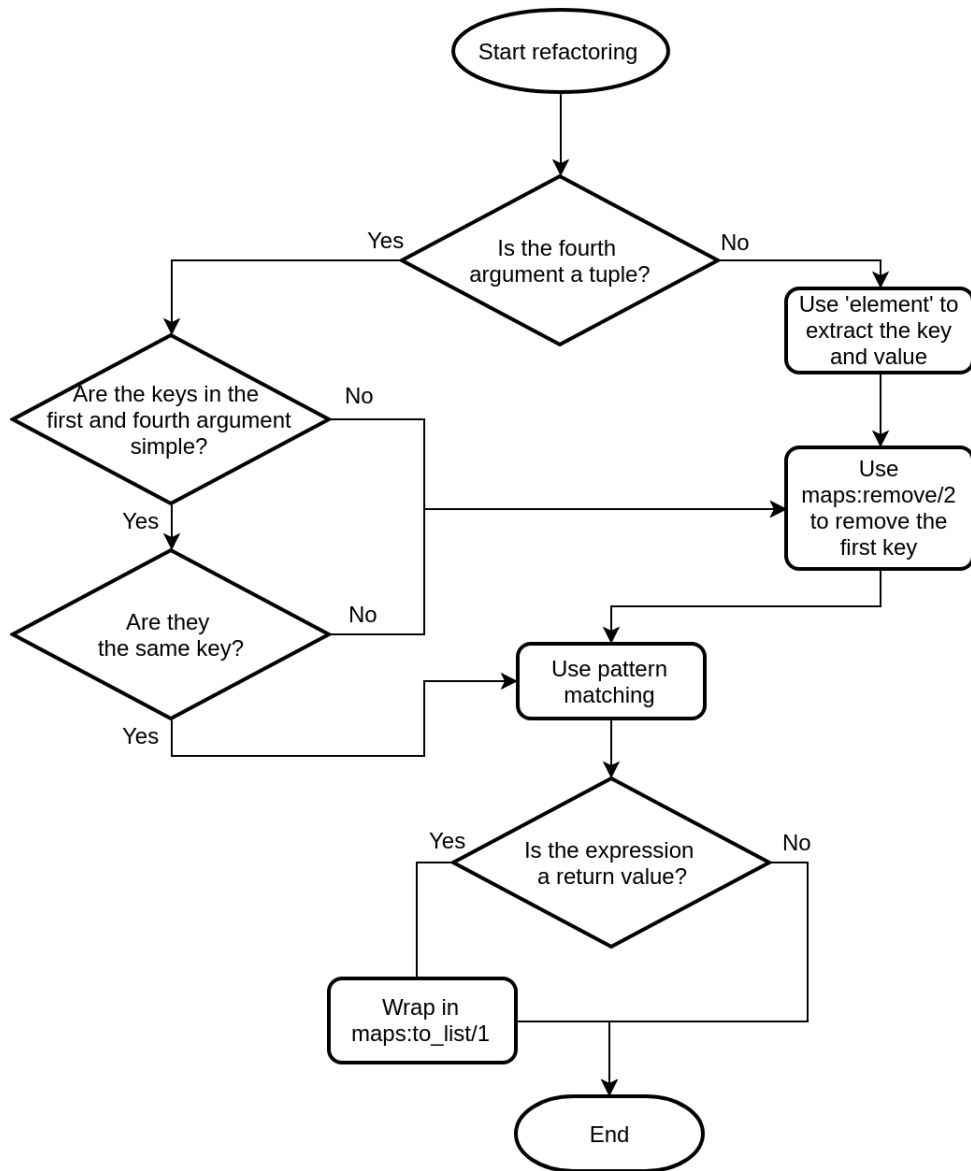


Figure 3.5: Flowchart of the decisions we need to make in order to transform a `lists:keystore/4` call


```

1 % 1.) Same key used
2 lists:keystore(Key, 1, List, {Key, Value})
3
4 % 2.) Variable given as fourth parameter
5 lists:keystore(Key, 1, List, Tuple)
6
7 % 3.) Complex fourth parameter given,
8 %     is also a return value
9 lists:keystore(Key, 1, List, get_tuple(P1, P2))

```

(a) Before

```

1 % 1.) Same key used
2 List#{Key => Value}
3
4 % 2.) Variable given as fourth parameter
5 (maps:remove(Key, List))#{element(1, Tuple) => element(2, Tuple)}
6
7 % 3.) Complex fourth parameter given,
8 %     is also a return value
9 begin
10     Tuple = get_tuple(P1, P2),
11     maps:to_list((maps:remove(Key, List))#{element(1, Tuple) =>
12                 element(2, Tuple)})
13 end

```

(b) After

Figure 3.6: Refactoring for eliminating `lists:keystore/4` calls.

not already exist. The fourth argument of this function is the actual key-value pair, that will replace the element with the key equal to the first argument of this function. This means the key can also be changed by this call.

Ideally, the `Map#{Key => Value}` syntax should be used, which updates or inserts a value belonging to a key. The `keystore` call can only be transformed to this pattern matching directly, when the two keys in the call are equal. Checking the equality of keys in general is difficult, so a constraint introduced in this step is the equality will only be checked in the case of simple keys. Here, again, a simple key is either a variable, an atom or an integer literal.

If at least one of the keys is not a simple key, they are treated as not equal. The value belonging to the first key must be first removed from the map using `maps:remove/2`. Only after this can the new element be inserted using the pattern matching syntax. Since the refactoring requires the precondition that keys are always unique in the list, it is not possible that the insertion after removal would overwrite an existing value. However, there is no way for this to be checked and the refactoring must rely on the user making sure the preconditions are true.

If the fourth parameter of `keystore` is not a tuple, but some other expression, the `element/2` function can be used to extract the key and value part of that expression. If it is not just a simple variable, it also needs to be extracted to a new variable to keep the same behaviour and avoid evaluating a complex computation twice, with possible side effects.

Finally, if the call to `keystore` is the last expression in the current function clause, the transformed function needs to be wrapped in a call to `maps:to_list/1` in order to keep the return value of the function a list.

All these necessary decisions that need to be made are shown on Figure 3.5. There are also some code examples shown on Figure 3.6.

Also, similarly to `lists:keydelete/3` if the result of this call is bound to a variable, all occurrences of which need to be transformed as well. Because of this no other

type of binding is supported by the refactoring.

```
1 lists:keytake(Key, 1, List)
```

(a) Before

```
1 case maps:take(Key, List) of
2   error -> false
3   {Value, NewList} -> {value, {Key, Value}, NewList}
4 end
```

(b) After

Figure 3.7: Refactoring for eliminating `lists:keytake/3` calls.

lists:keytake/3. The `lists:keytake/3` is used to remove an element from the list, while also getting the value belonging to the removed key. The function returns the `{value, {Key, Value}, List}` tuple if the key is present in the list and returns the atom `false` otherwise.

The behaviour of the `maps:take/2` is similar, but it returns only the `{Value, Map}` tuple if the key exists and the atom `error` otherwise.

A case expression can be used to emulate the behaviour of `keytake` with the `maps:take/2` function. Since the key has to be used more than once, it needs to be extracted to a new variable, if it is not a simple key.

A basic case for using `maps:take/2` instead of `lists:keytake/3` is shown on Figure 3.7.

Since the value of this expression contains the new map, all occurrences of any variable bound to this new map also has to be transformed. Because of this reason, if a binding happens using the result of the `keytake` function call, the refactoring only allows it to be a binding to a ternary tuple, whose third element is a variable.

If the return value of the whole recursive function is the value of this call to `keytake`, then the third element of the result tuple has to be transformed back to a list by wrapping it in a `maps:to_list/1` call.

Transforming pattern matching in parameter list. Since the transformed function is recursive, a common use case would be if there are multiple clauses, one of which contains a pattern matching to an empty list.

To transform this empty list pattern matching, a new variable needs to be introduced instead of the pattern. This will be the variable containing the map. The size of the map can be checked in the guard of the function clause by the `map_size/1` function. The guard has to check for maps with size 0.

An example of transforming a function clause containing an empty list pattern is shown on Figure 3.8.

```
1 recursive(_, []) -> ok;  
2 recursive(Key, List) ->  
3     NewList = lists:keydelete(Key, 1, List),  
4     recursive(Key + 1, NewList).
```

(a) Before

```
1 recursive(_, Map) when map_size(Map) == 0 -> ok;  
2 recursive(Key, List) ->  
3     NewList = maps:remove(Key, List),  
4     recursive(Key + 1, NewList).
```

(b) After

Figure 3.8: Transforming an empty list pattern matching expression to use maps instead.

Transforming outside calls to the recursive function. The final thing that needs to be done is to make sure that calls to the refactored recursive function are transformed so that the recursive function now gets a map as its actual parameter instead of a list. This can be achieved by wrapping the correct argument into a call to `maps:from_list/1`.

Finally, an example of transforming a whole recursive function, together with transforming the calls to that function, is shown on Figure 3.9.

```

1 -module( test ).
2
3 main( List ) ->
4     recursive( 0, List ).
5
6 recursive( 100, List ) -> List ;
7 recursive( Key, List ) ->
8     {_, Value} = lists:keyfind( Key, 1, List ),
9     NewList = lists:keydelete( Key, 1, List ),
10    lists:keymember( Key, 1, NewList ),
11    recursive( Key + 1, NewList ).

```

(a) Before

```

1 -module( test ).
2
3 main( List ) ->
4     recursive( 0, maps:from_list( List ) ).
5
6 recursive( 100, List ) -> maps:to_list( List );
7 recursive( Key, List ) ->
8     #{Key:=Value} = List ,
9     NewList = maps:remove( Key, List ),
10    maps:is_key( Key, NewList ),
11    recursive( Key + 1, NewList ).

```

(b) After

Figure 3.9: Refactoring an entire recursive function to use maps instead of lists.

3.2.3 Eliminating higher-order function calls

For eliminating higher-order functions eight similar algorithms are used, based on which higher-order function is eliminated (`lists:map/2` or `lists:filter/2`), if a list comprehension or recursion is used and whether the higher-order function call uses a lambda or an implicit fun.

Map to list comprehension

To create a list comprehension from a map function call, the mapping function call needs to be placed in the expression part of the list comprehension, the generator should traverse the whole list and there should be no filters.

Implicit fun-expressions When using implicit fun-expressions in a map function call, the function call can simply be moved to the list comprehension. This transformation is shown on Figure 3.10.

```
1 lists:map(fun increase/1, List)
```

(a) Before

```
1 [increase(X) || X <- List]
```

(b) After

Figure 3.10: Refactoring a map function call which uses an implicit fun, to a list comprehension.

Lambdas. When using lambdas, based on measurements of energy consumption, the fun expression must not be included in the body of the list comprehension as a fun expression, since that would increase energy consumption significantly. What needs to be done is to inline the definition of the lambda in the list comprehension.

First, investigate the inlining of lambdas with a single clause. These fun expressions can also be split in two groups. One where there is only a single expression in

the body of the fun (such as `fun(X) -> X + 1 end`), and the other where there may be many expressions (such as `fun(X) -> Y = 1, X + Y end`). Since only a single expression can be in the body of a list comprehension, if there are multiple expressions in the body of the lambda, they need to be wrapped in a begin-end block, so that they behave as if they were a single expression. With a single expression in the body, it can simply be inlined without wrapping it in a begin-end block.

It is also possible for lambdas to have multiple clauses and for the clauses to have guards, similarly to function definitions. For example the following fun expression has two clauses, one with a guard and one without: `fun(X) when X > 0 -> positive; (_) -> non_negative end`. If the fun expression has multiple clauses or uses guards, the previous inlining techniques do not work. What needs to be done instead is to create a case expression, where each clause will correspond to a branch in the lambda, the branches have the same patterns and same guards as the clauses and the bodies of each clause is copied to the corresponding branch.

Examples for all inlining techniques described above are shown on Figure 3.11.

Filter to list comprehension

Transforming a filter higher-order function call to a list comprehension is similar to how map higher-order function calls can be transformed. The main difference is that with filter the function call or the lambda has to be moved to the filter part of the list comprehension, while the body part should contain only the variable that is bound by the generator.

Implicit fun-expressions When using implicit fun-expressions in a filter function call, the function call can simply be placed in the filter part of the list comprehension. This transformation is shown on Figure 3.12.

Lambdas. Transforming the use of fun expressions to be used in a list comprehension is similar to how these lambdas could be transformed when eliminating a

```

1 % 1.) Single clause, single expression
2 lists:map(fun(X) -> X + 1 end, List)
3
4 % 2.) Single clause, multiple expressions
5 lists:map(fun(X) -> Y = 1, X + Y end, List)
6
7 % 3.) Multiple clauses, some with guards
8 lists:map(fun(X) when X > 0 -> positive;
9           (_) -> non_negative end, List)

```

(a) Before

```

1 % 1.) Single clause, single expression
2 [X + 1 || X <- List]
3
4 % 2.) Single clause, multiple expressions
5 [begin Y = 1, X + Y end || X <- List]
6
7 % 3.) Multiple clauses, some with guards
8 [case Elem of
9   X when X > 0 -> positive;
10  _ -> non_negative
11 end || Elem <- List]

```

(b) After

Figure 3.11: Refactoring a map function call which uses a lambda, to a list comprehension.

```

1 lists:filter(fun even/1, List)

```

(a) Before

```

1 [X || X <- List, even(X)]

```

(b) After

Figure 3.12: Refactoring a filter function call which uses an implicit fun, to a list comprehension.

map higher-order function. Distinction can be made based on whether the lambda has a single clause or multiple clauses, and also in the case of a single clause if it has a single expression in its body or not.

Based on this classification, similarly to map, the possibilities are to inline the single expression of the body, wrap the multiple expressions in a begin-end block or create a case expression in the case of multiple clauses.

One significant difference between eliminating filter and eliminating map is that with maps, the joker (`_`) pattern can appear as the variable bound by the generator, since only the patterns used in the lambda would be referred to by the inlined function definition. This is different with filter, since in the body of the list comprehension the actual element of the list has to be referred to, so the use of a joker pattern in the generator is not allowed. If a joker was used in the fun expression a new variable must be introduced, through which the element of the list can be accessed.

Some examples for eliminating the filter higher order function are shown on Figure 3.13.

Map to recursion

When transforming a map call to recursion, a new recursive function has to be defined, which is specialized for the exact case we are trying to transform. The higher-order function call should be replaced with a call to this recursive function.

The newly defined recursive function must contain at least two clauses. The first clause should be the default clause, which only accepts an empty list as its parameter and the result should also be an empty list. This clause describes the base case of the recursion. All subsequent clauses should implement the specialized functionality described by either an implicit fun-expression or a lambda. These clauses should perform a recursive call, such that the last expression in all of these clauses is of the form `[SomeValue | recursive(Tail)]`.

```

1 % 1.) Single clause, single expression
2 lists:filter(fun(X) -> X rem 2 == 0 end, List)
3
4 % 2.) Using a joker in the pattern of the fun
5 lists:filter(fun(_) -> Y > 2 end, List)
6
7 % 3.) Single clause, multiple expressions
8 lists:filter(fun(X) -> Y = 2, X rem Y == 0 end, List)
9
10 % 4.) Multiple clauses, some with guards
11 lists:filter(fun(X) when X > 0 -> true;
12             (_) -> false end, List)

```

(a) Before

```

1 % 1.) Single clause, single expression
2 [X || X <- List, X rem 2 == 0]
3
4 % 2.) Using a joker in the pattern of the fun
5 [Elem || Elem <- List, Y > 2]
6
7 % 3.) Single clause, multiple expressions
8 [X || X <- List, begin Y = 2, X rem Y == 0 end]
9
10 % 4.) Multiple clauses, some with guards
11 [Elem || Elem <- List, case Elem of
12   X when X > 0 -> true;
13   _ -> false
14 end]

```

(b) After

Figure 3.13: Refactoring a filter function call which uses a lambda, to a list comprehension.

Implicit fun-expressions. Similarly to list comprehensions, a map call which uses an implicit fun is the simplest case to create a recursive function for. Apart from the base case clause in the recursion only one other clause is needed, which will contain a single expression. This expression calls the implicit function with the head of the list as its actual parameter, and prepend it to the result of the recursive call using the tail of the list. This transformation is shown on Figure 3.14.

```
1 lists :map(fun increase/1, List)
```

(a) Before

```
1 % Function call replaced
2 map_recursion(List)
3
4 % New function defined
5 map_recursion([]) -> [];
6 map_recursion([Head | Tail]) ->
7   [increase(Head) | map_recursion(Tail)].
```

(b) After

Figure 3.14: Refactoring a map function call which uses an implicit fun, to a new recursive function.

Lambdas. To transform a map call using a fun expression, the recursive function should have a separate clause for each clause of the lambda. The expressions inside the clauses do not need to be transformed in any way, except for the very last expression in each clause. In this case the result of the last expression has to be prepended to the list returned by the recursive call.

In each clause, pattern matching is used to extract the head element of the list. The exact pattern the head is matched to is the same as the single parameter of the corresponding clause in the fun expression. All clauses in the fun expression have a single parameter, since the `map` higher-order function requires the functions passed to it as a parameter to be unary functions.

A fun expression created inside of the map call can use variables that have been bound outside of the scope of the lambda, but in the scope of the map call. This use

of external variables means that to move the clauses of the lambda to a recursive definition, all used external variables have to be added to the recursive function as parameters, so that when an expression uses those variables they will be in scope.

The next problem is that one clause of the lambda may use the external variable **X**, while another clause uses **X** as a pattern, and as such shadows the external variable. In the transformation these cases of shadowing have to be identified, and in the clauses where shadowing is happening, a new variable name must be assigned to the shadowed variable so that it can be forwarded in the recursive call at the end of the clause.

Examples ranging from simple fun expressions to more complex ones used inside a call to map are shown on Figures 3.15 and 3.16.

```
1 % 1.) Single clause, multiple expressions
2 lists:map(fun(X) -> Y = 1, X + Y end, List)
3 % 2.) Multiple clauses, some with guards
4 lists:map(fun(X) when X > 0 -> positive;
5           (_) -> non_negative end, List)
6 % 3.) Using external variables in on clause,
7 %     shadowing it in another.
8 %     Y is an external variable
9 lists:map(fun({X1, X2}) -> X = X1 * X2, X + Y;
10          (Y) -> Y + 1 end, List)
```

Figure 3.15: Before - Refactoring a map function call which uses a lambda, to a new recursive function.

Filter to recursion

Transforming a call to the filter function to use recursion instead is almost identical to how calls to the map higher-order function can be eliminated. The only significant difference being that the last expression in each non-base case clause should now be a case expression in the following form:

```
1 case SomeValue of
2   true -> [Head | recursive(Tail)];
3   false -> recursive(Tail)
4 end
```

```

1  % 1.) Single clause, multiple expressions
2  % Function call replaced
3  map_recursion(List)
4  % New function defined
5  map_recursion([]) -> [];
6  map_recursion([X | Tail]) ->
7      Y = 1,
8      [X + Y | map_recursion(Tail)].
9  % 2.) Multiple clauses, some with guards
10 % Function call replaced
11 map_recursion(List)
12 % New function defined
13 map_recursion([]) -> [];
14 map_recursion([X | Tail]) when X > 0 ->
15     [positive | map_recursion(Tail)];
16 map_recursion([_ | Tail]) ->
17     [non_negative | map_recursion(Tail)].
18 % 3.) Using external variables in on clause,
19 %     shadowing it in another.
20 %     Y is an external variable
21 % Function call replaced
22 map_recursion(List, Y)
23 % New function defined
24 map_recursion([], _) -> [];
25 map_recursion([X1, X2 | Tail], Y) ->
26     X = X1 * X2,
27     [X + Y | map_recursion(Tail, Y)];
28 map_recursion([Y | Tail], Y1) ->
29     [Y | map_recursion(Tail, Y1)].

```

Figure 3.16: After - Refactoring a map function call which uses a lambda, to a new recursive function.

This ensures that only the desired elements get filtered.

Implicit fun-expressions Similarly to how the use of implicit fun-expression in the case of maps could be transformed, only one additional clause of the recursive function needs to be created (besides the base case), which calls the implicit function and decides whether the current element needs to be prepended to the result of further recursive calls. One such refactoring is shown on Figure 3.17.

```
1 lists : filter (fun even/1, List)
```

(a) Before

```
1 % Function call replaced
2 filter_recursion(List)
3
4 % New function defined
5 filter_recursion([]) -> [];
6 filter_recursion([Head | Tail]) ->
7     case even(Head) of
8         true -> [Head | filter_recursion(Tail)];
9         false -> filter_recursein(Tail)
10     end.
```

(b) After

Figure 3.17: Refactoring a filter function call which uses an implicit fun, to a new recursive function.

Lambdas. Again, transforming filter higher-order function calls that use lambdas is quite similar to how it can be done with the map function. Only the last expression in each clause has to be modified to use the case expression shown above.

One additional thing that needs special attention, is that when creating the pattern matching to extract the head of the list, the head cannot be allowed to be matched to a joker pattern, since it needs to be referred to later on in the function definition. Thus if the pattern in the clause of the fun expression was either a joker or a more complex pattern matching expression containing a joker (such as `{A, _}`), a new variable has to be created so that in the body of the function the head of the list can be referred to.

Some examples showing how to eliminate the filter higher-order function are presented on Figures 3.18 and 3.19.

```
1 % 1.) Multiple clauses, some with guards and a joker
2 lists:filter(fun(X) when X > 0 -> X rem 2 == 0;
3             (_) -> false end, List)
4 % 2.) Using external variables in on clause,
5 %     shadowing it in another.
6 %     Joker appears inside a tuple. Y is an external variable
7 lists:filter(fun({X1, X2}) -> X1 == X2;
8             ({Y, _}) -> Y > 0 end, List)
```

Figure 3.18: Before - Refactoring a filter function call which uses a lambda, to a new recursive function.

3.3 Implementation

To implement the previously presented algorithms for the refactorings, a single module has to be created in RefactorErl. This module has to be placed in the `trunk/tool/lib/referl_user/src` directory of the tool.

The name of the newly defined module must contain the `reftr_` prefix, followed by the name of the refactoring. The three presented refactorings can be found in the following files:

- Eliminating `proplists:get_value` calls: `reftr_elim_get_value.erl`
- Transforming list to map: `reftr_transform_list.erl`
- Eliminating higher-order functions: `reftr_elim_hof.erl`

The exported functions in the module of a refactoring are specified by RefactorErl to be the following functions:

prepare/1. This function is the one performing the necessary steps for refactoring. It must gather all necessary information by executing queries, it must check all

```

1 % 1.) Multiple clauses, some with guards and a joker
2 filter_recursion(List) % Function call replaced
3 % New function defined
4 filter_recursion([]) -> [];
5 filter_recursion([X | Tail]) when X > 0 ->
6     case X rem 2 == 0 of
7         true -> [X | filter_recursion(Tail)];
8         false -> filter_recursion(Tail) end;
9 filter_recursion([Head | Tail]) ->
10     case true of
11         true -> [Head | filter_recursion(Tail)];
12         false -> filter_recursion(Tail) end.
13 % 2.) Using external variables in on clause,
14 %     shadowing it in another.
15 %     Joker appears inside a tuple. Y is an external variable
16 filter_recursion(List, Y) % Function call replaced
17 % New function defined
18 filter_recursion([]) -> [];
19 filter_recursion([X1, X2 | Tail], Y) ->
20     case X1 + X2 == Y of
21         true -> [X1, X2 | filter_recursion(Tail, Y)];
22         false -> filter_recursion(Tail, Y) end;
23 filter_recursion([Head = {Y, _} | Tail], Y1) ->
24     case Y > 0 of
25         true -> [Head | filter_recursion(Tail, Y1)];
26         false -> filter_recursion(Tail, Y1) end.

```

Figure 3.19: After - Refactoring a filter function call which uses a lambda, to a new recursive function.

possible constraints and it may interact with the user. The transformation of the program graph must be done by the functions returned by `prepare/1`.

- *Parameters:*

Args :: `proplist()` - The arguments of the refactoring, in the form of a list of key-value tuples. This contains all possible parameters of the refactoring, such as the name of the refactored file and the position at which the refactoring has to be performed.

- *Return value:*

A list of functions that need to be executed by `RefactorErl`. These functions perform the transformation on the semantic program graph, including creating the new syntactic nodes and edges and replacing subtrees of the graph.

The first function in the list takes no parameter, but all subsequent functions are unary, and the actual parameter will be the value returned by the previous function.

Each function is responsible for performing a step of the whole refactoring, after the execution of each function the program graph must be in a syntactically correct state, semantic correctness is only required at the end of the final function execution.

These returned functions should not throw any exceptions, but they can execute queries if needed. All checks should be performed before returning the functions, not inside of the returned functions.

error_text/2. The responsibility of this function is to create a user friendly description that gets displayed in the case that a check fails or any error gets thrown. This function handles `LocalErrors`.

- *Parameters:*

ErrorKey :: `atom()` - The key describing the type of the error. This key is used to give specialized error messages for every possible error.

ErrorInfo - Any additional info of why and where the error was thrown. This information is used to create more user friendly error messages. The type of **ErrorInfo** can be anything, but most commonly it is a list.

- *Return value:*

A list of strings that are concatenated to produce the error message.

These two functions are common and required for all refactorings. The following sections present the implementation details specific to each refactoring. These implementation details include a more specific view of what `prepare/1` does for each refactoring.

3.3.1 Eliminate `get_value`

This refactoring is implemented in the `reftr_elim_get_value` module.

prepare/1. The execution of this function consists of the following steps:

- Query for the 'application' node selected.
- Identify the function call and check if it is `proplists:get_value` with an arity of either 2 or 3.
- Extract the actual parameters of the function call.
- Create a new, unused variable name with the prefix "Var".
- Create a lambda that is returned in the list. This lambda constructs the syntactic nodes of the `lists:keyfind/3` call and uses copies of the extracted arguments. Finally, replace the subtree belonging to `get_value` with the newly constructed nodes.

- *Parameters:*

Args :: `proplist()` - The arguments for this refactoring must contain the `file` and `position` keys.

- *Return value:*

A list containing a single function that performs the transformation of the SPG.

get_closest_parent_of_type/2. Get the closest ancestor of a node in the SPG that is of a given type.

Check if the current `Node` is of the given type, if yes return it, otherwise perform a query to get the parent of that node and recursively calls itself.

- *Parameters:*

Node :: `node()` - The node whose parent has to be extracted.

Type :: `atom()` - The type of the desired parent node

- *Return value:*

The first node that is of the given type and is an ancestor of the initial parameter node in the SPG.

error_text/2. The possible `LocalErrors` thrown during this refactoring are the following:

- **bad_application:** The selected expression by the user has no ancestor of type `'application'`.

- **bad_function:** The selected application is not a call to `proplists:get_value` with an arity of 2 or 3.

- **ambiguous:** This error key is used, if queries that should only return a single node fail and either return multiple nodes or none at all. If the SPG is in a correct state, this error should never be thrown.

3.3.2 Transforming a list to map

This refactoring is implemented in the `reftr_transform_list` module.

prepare/1. The execution of this function consists of the following steps:

- Query for the binding of the selected variable and check if it is a parameter of the function.
- Query for the selected function and check if it is a recursive function.
- Identify all clauses of the function and create a list of functions that transform the parameters of each clause if needed, using `transform_pattern/1`.
- Identify all occurrences of the selected variable in the bodies of every function clause and create a list of functions to transform these references to the variable using `transform_variable/2`.
- Identify all calls to the recursive function and filter for the ones appearing outside of any clause of the function. These calls need to be transformed. Again, a list of functions is created for these transformations using `transform_outside_call/3`.
- Finally, all the lists of functions are concatenated in the order they were created. A lambda with zero parameters is prepended to the list, so that all functions created for transformation can be unary functions. This is needed, because only the first function of the list can have zero parameters, and the transformations have no information which function will become the first element of the list.

- *Parameters:*

`Args :: proplist()` - The arguments for this refactoring must contain the `file` and `position` keys.

- *Return value:*

A list containing all functions that perform the transformation of the SPG.

transform_pattern/1. Checks if the pattern node given as a parameter is either of type 'variable' or 'cons', throws `invalid_pattern` if it is anything else. If the 'cons' pattern does not belong to an empty list pattern a `cons_error` is thrown.

If the pattern is of type 'cons' a call to `transform_empty_list_cons/1` is made to create the transformation functions.

Otherwise, patterns of type 'variable' are not transformed, so an empty list is returned.

- *Parameters:*

`Pattern :: node()` - The node of the SPG containing the pattern corresponding to the formal parameter of a clause of the recursive function.

- *Return value:*

A list containing the functions needed for the transformation of the pattern node. This can be an empty list or a list containing two functions.

transform_empty_list_cons/1. Queries for the function clause node and creates two functions to execute the transformation of the empty list pattern.

The first function generates a new, unused variable name with the prefix "Map" and constructs the variable node, which replaces the empty list pattern. The variable name is passed on to the second function.

The other function adds the guard expression to the function clause, which performs the check for empty map. For this a function call is created to `map_size` using `create_fun_call/2`. This function call is inserted in an infix equality comparison expression, which is then added as a guard to the function clause.

- *Parameters:*

`Pattern :: node()` - The node of the SPG containing the pattern corresponding to the formal parameter of a clause of the recursive function.

- *Return value:*

A list containing the functions needed for the transformation of the empty list pattern.

transform_variable/2. Executes a query to get all referenced to the variables in its parameter. Then calls `transform_varref/2` to transform each individual reference.

- *Parameters:*

Vars :: [node()] - A list of nodes representing all the variables that need to be transformed in the recursive function. This needs to be a list, because different function clauses can bind different variables to the same positional argument.

Func :: node() - The node in the SPG representing the recursive function that is being refactored.

- *Return value:*

A list of transformations performing the changes in the SPG for all occurrences of the **Vars** given as a parameter.

transform_varref/2. Query the immediate parent of the **Varref** node. If this parent is a clause, than the list is present in the clause as a standalone expression, meaning it is not part of any function call or operation. In this case the function `transform_standalone_list/2` is used to create the functions for the transformation. Otherwise `transform_varref_par/3` is called to create the transformation.

- *Parameters:*

Varref :: node() - A node in the SPG representing an occurrence of the variable that is being refactored.

OriginalFun :: node() - The node in the SPG representing the recursive function that is being refactored.

- *Return value:*

A list of functions transforming the single occurrence given as the first argument.

transform_standalone_list/2. Using the `is_return_expr/2` function decides if the variable is the last expression in its clause. If not, no transformation is done, otherwise `transform_return_list/1` is called.

- *Parameters:*

`Varref :: node()` - A node in the SPG representing an occurrence of the standalone list that is being refactored.

`OriginalFun :: node()` - The node in the SPG representing the recursive function that is being refactored.

- *Return value:*

A list of functions transforming the standalone list. If the list is not a return expression an empty list is returned.

transform_return_list/1. Creates a function, that wraps the list in a call to `maps:to_list/1`

- *Parameters:*

`Varref :: node()` - A node in the SPG representing a standalone list, which is a return value of its clause.

- *Return value:*

A list containing the single function that performs the wrapping of the list in the `to_list` function call.

transform_varref_par/3. Checks whether the variable in its parameter appears in an 'arglist'. If not, a `bad_parent` error is thrown, otherwise the function `transform_varref_arglist/3` is used to transform the function call in which the variable appears.

- *Parameters:*

`Varref :: node()` - A node in the SPG belonging to a reference of the variable that is transformed.

`Parent :: node()` - The parent node of `Varref`.

`OriginalFun :: node()` - The node in the SPG representing the recursive function that is being refactored.

- *Return value:*

A list containing functions to transform a function call containing the variable.

transform_varref_arglist/3. Query the function node that is being called. If it is the same as the recursive function, no transformation is needed. Otherwise check if the function call is an allowed call. Allowed calls are:

- o `lists:keyfind/3`
- o `lists:keydelete/3`
- o `lists:keymember/3`
- o `lists:keystore/4`
- o `lists:keytake/3`

If the call is not allowed a `bad_fun_call` error is thrown, otherwise `transform_varref_function_call/5` is used to transform the specific function call.

- *Parameters:*

`Varref :: node()` - A node in the SPG belonging to a reference of the variable that is transformed.

`ArgList :: node()` - The parent node of `Varref`, which is representing the argument list of the function call, that the variable appears in.

`OriginalFun :: node()` - The node in the SPG representing the recursive function that is being refactored.

- *Return value:*

A list containing functions to transform a function call containing the variable.

transform_varref_function_call/5. Using pattern matching to the module name, function name and arity of the function, creates the specific functions for transforming a call to each allowed function.

Also performs some checks using `validate_n_arg/2` and `validate_list_arg/3`.

- *Parameters:*

`MFA :: {atom(), atom(), integer()}` - The module name, function name and arity of the function call being transformed.

`Varref :: node()` - A node in the SPG belonging to a reference of the variable that is transformed.

`Args :: [node()]` - A list of nodes representing the actual parameters of the function call.

`FunApp :: node()` - The node containing the function application.

`OriginalFun :: node()` - The node representing the recursive function that is currently being refactored.

- *Return value:*

A list containing functions to transform a function call containing the variable.

For all allowed function, the resulting list of functions created by the functions `{FUNCTION_NAME}_transform`, where `{FUNCTION_NAME}` is the name of the function call being transformed, such as `keyfind` or `keydelete`.

The following additional steps are performed for the allowed functions, before the respective transform function calls.

- `lists:keyfind/3`

Determine the context in which `keyfind` is used by calling the `determine_keyfind_branch/3` function.

- `lists:keydelete/3`

Transform all occurrences of the variable that is bound by the result of the function call, using the `transform_newly_bound_list/3` function.

Also check if the function call expression is a return value of its function clause, using the `is_expr_or_parent_return_expr/2` function.

- `lists:keymember/3`

This function requires no additional steps.

- `lists:keystore/4`

Transform all variables that get bound to the modified list, using the `transform_newly_bound_list/3` function.

Determine the context in which `keystore` is used by the function `determine_keystore_branch/2`.

Also check if the function call is a return expression in the clause (`is_expr_or_parent_return_expr/2`).

- `lists:keytake/3`

Transform all occurrences of the variable that gets bound to the resulting modified list using `transform_newly_bound_list/3`.

Also check if the function call is a return value, using the `is_expr_or_parent_return_expr/2` function.

transform_newly_bound_list/3. Checks if the result of a function call is bound to a variable, if yes calls `spec_transform_newly_bound_list/3`

- *Parameters:*

Branch :: `atom()` - The name of the function, whose result is transformed.

FunApp :: `node()` - The node containing the function application.

OriginalFun :: `node()` - The node representing the recursive function that is currently being refactored.

- *Return value:*

A list containing functions to transform all occurrences of any newly bound variable resulting from the `FunApp` function call.

spec_transform_newly_bound_list/3. Extracts the bound variable. If the result of `keytake` is being transformed checks if the bound expression is a ternary tuple, whose last element is a variable.

If the result of either `keydelete` or `keystore` is being transformed, only checks if the bound expression is a variable.

If the checks fail, an `unsupported_match` exception is thrown.

If the bound variable is successfully extracted, `transform_new_list_part/2` is used in all cases to create the transformation functions for all occurrences of the newly bound variable.

- *Parameters:*

Branch :: `atom()` - The name of the function, whose result is transformed.

Parent :: `node()` - The node containing the parent of the function application.

OriginalFun :: `node()` - The node representing the recursive function that is currently being refactored.

- *Return value:*

A list containing functions to all occurrences of any newly bound variable resulting from the function call.

transform_new_list_part/2. Extract the variable from the binding information of the newly bound variable and using `transform_variable/2` transforms all occurrences of it.

- *Parameters:*

`NewListPart` :: `node()` - The node where the binding of the new variable happens.

`OriginalFun` :: `node()` - The node representing the recursive function that is currently being refactored.

- *Return value:*

A list containing functions to transform all occurrences of the newly bound variable.

determine_keyfind_branch/2. Determines which algorithm should be used to transform a call to `keyfind`. This is achieved by analysing the type of the key argument and the context of the function call. For looking at the context the `classify_keyfind_tuple/2` function is used, while the 'type' of the key is determined by the `keytype/1` function.

- *Parameters:*

`FunApp` :: `node()` - The node containing the `keyfind` function application.

`Args` :: [`node()`] - A list of nodes representing the actual parameters of the `keyfind` function call.

`OriginalFun` :: `node()` - The node representing the recursive function that is currently being refactored.

- *Return value:*

An atom, based on the information extracted in the function the following results are possible:

- `match_joker`: if the result is bound to a tuple whose first element is the joker pattern.
- `match_same_key`: if the result is bound to a tuple whose first element can be shown to be the same key as the key in the argument list.
- `match_different_key`: if the result is bound to a tuple whose first element cannot be shown to be the same key as in the argument list.
- `simple_key`: no binding to tuple is happening and the key is 'simple'.
- `complex_key`: no binding to tuple is happening and the key is 'complex'.
- `simple_key_ret`: no matter if there is any binding, if the function call is part of the last expression in the clause, and the key is 'simple' this branch is selected.
- `complex_key_ret`: no matter if there is any binding, if the function call is part of the last expression in the clause, and the key is 'complex' this branch is selected.

classify_keyfind_tuple/2. Classifies the type of matching that is happening to the result of the `keyfind` call.

- *Parameters:*

`LHS :: node()` - The node that contains the expression that gets bound to the result of the `keyfind` call. This must be a tuple.

`KeyArg :: node()` - The argument node of the argument list of `keyfind`, which corresponds to the key.

- *Return value:*

An atom depending on the structure of the tuple.

- `match_joker`: the first element of the tuple is a joker pattern
- `match_same_key`: `KeyArg` and the first element of the tuple can be shown to be the same
- `match_different_key`: `KeyArg` and the first element of the tuple cannot be shown to be the same

keyfind_transform/3. Based on the branch that is given in the first argument, chooses which algorithm to use. Constructs the needed syntactic elements and replaces the original `keyfind` call with the newly created nodes.

- *Parameters:*

`Branch` :: `atom()` - The branch determining which transformation algorithm will be used.

`FunApp` :: `node()` - The node containing the `keyfind` function application that needs to be replaced.

`Args` :: `[node()]` - A list of nodes representing the actual parameters of the `keyfind` function call.

- *Return value:*

A list containing a single function that performs the transformation of the `keyfind` call.

keydelete_transform/3. Based on if the function call is a return value in its clause, creates the transformation function, which constructs the needed syntactic elements and replaces the original `keydelete` call with the newly created nodes.

- *Parameters:*

`IsReturn` :: `boolean()` - Specifies if the call to `keydelete` is returned from its clause.

FunApp :: **node()** - The node containing the **keydelete** function application that needs to be replaced.

Args :: [**node()**] - A list of nodes representing the actual parameters of the **keydelete** function call.

- *Return value:*

A list containing a single function that performs the transformation of the **keydelete** call.

keymember_transform/2. Creates the transformation function, which creates the needed syntactic elements and replaces the original **keymember** call with the newly created nodes.

- *Parameters:*

FunApp :: **node()** - The node containing the **keymember** function application that needs to be replaced.

Args :: [**node()**] - A list of nodes representing the actual parameters of the **keymember** function call.

- *Return value:*

A list containing a single function that performs the transformation of the **keymember** call.

determine_keystore_branch/2. Determines which algorithm should be used to transform a call to **keystore**. This is achieved by checking the actual parameters of the function.

- *Parameters:*

FirstKey :: **node()** - The node for the first argument of the function, which is the key part.

Tuple :: **node()** - The node for the fourth argument of the function, which is the new key-value tuple.

- *Return value:*

An atom, showing the relations between the first and fourth argument of the **keystore** function call:

- **same_keys**: it can be shown that the first and the fourth arguments contain the same key.
- **different_keys**: it cannot be shown that the first and the fourth arguments contain the same key, but the fourth argument is a tuple pattern.
- **no_tuple_simple**: the fourth argument is not a tuple pattern, but the first argument is a simple key.
- **no_tuple_complex**: the fourth argument is not a tuple pattern, and the first argument is a complex key.

keystore_transform/4. Chooses the algorithm based on the branch determined by **determine_keystore_branch/2** and whether the function call is a return value, creates a function that transforms the call to **keystore** by creating the new syntactic elements and replaces the original **keystore** call.

- *Parameters:*

Branch :: **atom()** - Specifies the relations between the first and fourth argument, as determined by **determine_keystore_branch/2**

IsReturn :: **boolean()** - Specifies if the function call is a return expression.

FunApp :: **node()** - The node containing the **keytake** function application that needs to be replaced.

Args :: [**node()**] - A list of nodes representing the actual parameters of the **keytake** function call.

- *Return value:*

A list containing a single function that performs the transformation of the `keystore` call.

keytake_transform/4. Chooses the transformation algorithm based on the type of key and whether the function call is a return value, creates a function that transforms the call to `keytake` by creating the new syntactic elements and replaces the original `keytake` call.

- *Parameters:*

`KeyType` :: `atom()` - Specifies if the key argument in the function call is a simple or complex key. The allowed values for this parameter are `simple_key` and `complex_key`.

`IsReturn` :: `boolean()` - Specifies if the function call is a return expression.

`FunApp` :: `node()` - The node containing the `keytake` function application that needs to be replaced.

`Args` :: `[node()]` - A list of nodes representing the actual parameters of the `keytake` function call.

- *Return value:*

A list containing a single function that performs the transformation of the `keytake` call.

transform_outside_call/3. Creates the function for transforming a single call to the refactored recursive function by wrapping the correct argument in a call to the `maps:from_list` function.

- *Parameters:*

`Parameter` :: `node()` - The node in the SPG belonging to the formal parameter of the function.

`Expr` :: `node()` - The node for the call to the recursive function.

`FunName` :: `atom()` - The name of the recursive function.

- *Return value:*

A list containing a single function that performs the transformation to a function call to the refactored recursive function.

is_match_expr/1. Determines if an expression is of type 'match_expression'

- *Parameters:*

Expr :: **node()** - The expression that needs to be checked.

- *Return value:*

A boolean value based on whether the expression is a 'match_expression'.

is_return_expr/2. Determines if an expression is a return value in a given function or not.

- *Parameters:*

Expr :: **node()** - The expression that needs to be checked if it is a return value.

Func :: **node()** - The node referring to the function that needs to be used for the checking.

- *Return value:*

A boolean value denoting if the expression is a return value in the function.

is_expr_or_parent_return_expr/2. Determines if an expression or its parent is a return value in a given function or not.

- *Parameters:*

Expr :: **node()** - The expression that needs to be checked if it or its parent is a return value.

Func :: **node()** - The node referring to the function that needs to be used for the checking.

- *Return value:*

A boolean value denoting if the expression or its parent is a return value in the function.

create_fun_call/2. Constructs the necessary syntactic elements for a function call.

- *Parameters:*

Function :: {**atom()**} | {**atom()**, **atom()**} - The function name or the module and function name that needs to be called.

Args :: [**node()**] - The nodes that represent the arguments of the function call.

- *Return value:*

The constructed function application node.

validate_n_arg/2. Validates the type and value of the argument that specifies which part of the tuple is the key. This can only be an integer 1. If validation fails a **bad_n_arg** error is thrown.

- *Parameters:*

NArg :: **node()** - The node belonging to the argument that determines which element of the tuple in a proplis denotes the key.

MFA :: {**atom()**, **atom()**, **integer()**} - The module name, function name and arity of the function call being validated.

- *Return value:*

No usable return value, if the function throws an error validation has failed, if no error is thrown validation has passed.

validate_list_arg/3. Validates the parameter of the function calls that contains the list. This parameter should refer to the same variable that is being transformed.

- *Parameters:*

ListArg :: **node()** - The node belonging to the variable referred to by the argument of the function call.

Varref :: **node()** - The node belonging to the variable being transformed.

MFA :: {**atom()**, **atom()**, **integer()**} - The module name, function name and arity of the function call being validated.

- *Return value:*

No usable return value, if the function throws an error validation has failed, if no error is thrown validation has passed.

keytype/1. Determines if an expression is a simple type or not. The following types are considered to be simple:

- atom
- integer
- variable

- *Parameters:*

KeyArg :: **node()** - The node representing the key argument of the function calls.

- *Return value:*

An the **simple_key** atom if the type of the node is any of the ones mentioned above, otherwise the **complex_key** atom is returned.

error_text/2. The possible `LocalErrors` thrown during this refactoring are the following:

- `non_recursive`: The function in which the variable is selected is not recursive.
- `non_parameter`: The selected variable is not a parameter of the function clause containing it.
- `parameter_type`: The selected variable is bound somewhere else, not in the parameter list.
- `bindings`: The selected variable is bound by multiple expressions.
- `ambiguous`: A query, that should only return a single value, instead returns either multiple values or none at all. Normally, if the SPG is in a correct state this error should never happen.
- `bad_parent`: The selected variable does not appear in an 'arglist', but instead some other kind of expression.
- `bad_fun_call`: The selected variable appears in the 'arglist' of a function that cannot be transformed.
- `bad_n_arg`: A call to the functions that can be transformed does not use the integer literal 1 as its second parameter.
- `bad_list_arg`: The argument, that contains the list is not the same as the selected variable.
- `unsupported_match`: In the case of `keystore`, `keydelete` and `keytake` only a limited type of matching is allowed.
- `cons_error`: Some clause of the recursive function uses list pattern matching other than the empty list pattern.
- `invalid_pattern`: Some clause of the function uses pattern matching, that is not the empty list pattern or a single variable.

3.3.3 Eliminating higher-order functions

This refactoring is implemented in the `reftr_elim_hof` module.

prepare/1. The execution of this function consists of the following steps:

- Query for the closest ancestor of the selected expression, which has type 'application', using the `get_closest_parent_of_type/2` function.
- Extract the information from the function application and validate that it is a higher-order function call that can be refactored, using the `validate_hof/1` function.
- Get the mode of transformation (list comprehension or recursion) from `Args`, or if it is not present ask the user to select the mode, using the `get_mode/1` function.
- Use the heuristic algorithm to select the correct mode if 'default' is provided.
- Finally, create the functions for transforming the higher-order function call using `transform_hof_call/6`.

- *Parameters:*

`Args :: proplist()` - The arguments for this refactoring must contain the `file` and `position` keys. An additional parameter can be added in `Args` to specify if a list comprehension or recursion should be used. This parameter should belong to the key `hof_mode` and can have a value of `recursive`, `list_comp` or `default`.

- *Return value:*

A list containing all functions that perform the transformation of the SPG.

get_closest_parent_of_type/2. Get the closest ancestor of a node in the SPG that is of a given type.

Check if the current `Node` is of the given type, if yes return it, otherwise perform a query to get the parent of that node and recursively calls itself.

- *Parameters:*

`Node` :: `node()` - The node whose parent has to be extracted.

`Type` :: `atom()` - The type of the desired parent node

- *Return value:*

The first node that is of the given type and is an ancestor of the initial parameter node in the SPG.

`validate_hof/1`. Checks whether the selected function application uses either `lists:map/2` or `lists:filter/2`. If it is anything else, a `bad_hof` error is thrown.

- *Parameters:*

`MFA` :: `{atom(), atom(), integer()}` - The module name, function name and arity of the selected function.

- *Return value:*

This function has no usable return value, if the validation fails it throws an error, otherwise, if nothing is thrown, the validation has passed.

`get_mode/1`. Extracts the selected mode of transformation from `Args`, or if it is not present uses `ask_hof_question/1` to interact with the user and get the necessary input.

- *Parameters:*

`Args` :: `proplist()` - The same arguments that get passed to `prepare/1`.

- *Return value:*

Returns an atom denoting the selected mode of transformation. It can be the following three values:

- `list_comp`
- `recursive`
- `default`

ask_hof_question/1. Asks the user to select an option of the three possible modes of transformation.

- *Parameters:*

`Text :: string()` - The text of the question that has to be asked from the user.

- *Return value:*

Returns an atom denoting the selected mode of transformation. It can be the following three values:

- `list_comp`
- `recursive`
- `default`

get_fun_type/1. Extracts the function parameter from a higher-order function call and determines whether a lambda or an implicit fun is used.

- *Parameters:*

`Expr :: node()` - The function application node of a higher-order function.

- *Return value:*

Either the atom `fun_expr` if a lambda is used or the atom `implicit_fun` if an implicit fun is used.

transform_hof_call/6. Creates the list of transformation functions using the relevant algorithm. The used algorithm is selected based on the function being refactored, the mode of transformation and whether a fun expression or an implicit fun is used.

- *Parameters:*

MFA :: {atom(), atom(), integer()} - The module name, function name and arity of the selected function call.

Mode :: atom() - The mode of transformation, which is either `list_comp` or `recursive`.

FunType :: atom() - The type of the function argument used in the higher-order function call. Either `fun_expr` or `implicit_fun`.

Expr :: node() - The function application node of a higher-order function.

Args :: [node()] - The nodes representing the arguments of the higher-order function.

File :: string() - The name and path to the file that is being refactored.

- *Return value:*

A list of functions that perform the transformation of the higher-order function call.

This function contains different clauses for every possible combination of function, transformation mode and function type used. The description of how each clause works and what they do is shown below.

- list comprehension, implicit fun

This clause extracts the implicit function argument and then creates the function to perform the transformation. The transformation function constructs the syntactic nodes for the list comprehension and a call to the implicit function. These nodes are used to replace the higher-order function call in the SPG.

The difference between the map and filter higher-order functions is that the implicit function call is placed at different locations when creating the nodes for the list comprehension.

- list comprehension, fun expression

The fun expression (lambda) gets extracted from the argument list and all the patterns, guards and bodies of the lambda get extracted using the `get_patterns_guards_bodies/1` function.

After that the type of the lambda is determined with regards to its clauses, bodies and patterns. For this the `determine_clause_type/1` function is used.

Finally, the transformation function is created by the function `map_list_comp_from_fun_expr/4` in the case of map and `filter_list_comp_from_fun_expr/4` in the case of filter.

- recursion, implicit fun

In this case the resulting list contains two functions. The first one is creating the nodes for the new recursive function definition and adding it to the file. The second function replaces the higher-order function with a call to the newly defined recursive function. The name of the new function is passed between the two functions.

- recursion, fun expression

In this clause, the transformation functions are created by a call to the higher-order function `transform_recursion_fun_expr/6`. The difference between the implementation for map and filter is the functions passed to the HOF. In the case of maps the following functions are used:

- `map_transform_pattern/2`
- `map_transform_last_body/5`

In the case of filter the following functions are used:

- `filter_transform_pattern/2`
- `filter_transform_last_body/5`

get_patterns_guards_bodies/1. Extracts the patterns, guards and bodies belonging to a fun expression.

- *Parameters:*

`FunExpr` :: `node()` - The node representing the fun expression.

- *Return value:*

A list of type `[[node()], [node()], [node()]]`, where each tuple corresponds to a clause of the fun expression and the first element of the tuple contains the patterns, the second the guards and the third the bodies belonging to the clause.

determine_clause_type/1. Determines the type of clauses belonging to a fun expression based on the number of clauses, number of bodies and the number and type of patterns.

- *Parameters:*

`PatternsGuardsBodies` :: `[[node()], [node()], [node()]]` -

The patterns, guards and bodies belonging to the clauses of a fun expression.

- *Return value:*

An atom denoting the determined type.

- `single_body_var`: a single clause with a single body and a single pattern of type 'variable'
- `single_body_joker`: a single clause with a single body and a single pattern of type 'joker'
- `multiple_bodies_var`: a single clause with multiple bodies and a single pattern of type 'variable'
- `multiple_bodies_joker`: a single clause with multiple bodies and a single pattern of type 'joker'
- `default`: all other cases

map_list_comp_from_fun_expr/4. Based on the type of the fun expression, as determined by `determine_clause_type/1` creates a transformation function that constructs a list comprehension either containing a single expression or multiple expressions grouped together by a block expression, and replaces the higher-order function call with the newly created list comprehension nodes.

Whether the fun expression uses the 'joker' pattern or not is not taken into account in the transformation.

- *Parameters:*

Branch :: `atom()` - The type of the fun expression.

Expr :: `node()` - The node corresponding to the higher-order function call.

PatternsGuardsBodies :: `[{[node()], [node()], [node()]}]` -

The patterns, guards and bodies belonging to the clauses of the fun expression.

List :: `node()` - The second parameter node of the higher-order function call.

- *Return value:*

A list containing a single transformation function.

filter_list_comp_from_fun_expr/4. Based on the type of the fun expression, as determined by `determine_clause_type/1` creates a transformation function that constructs a list comprehension either containing a single expression or multiple expressions grouped together by a block expression, and replaces the higher-order function call with the newly created list comprehension nodes.

Whether the fun expression uses the 'joker' pattern or not is taken into account and results in a different transformation function, since a 'joker' pattern cannot be used when creating a list comprehension to filter.

- *Parameters:*

Branch :: `atom()` - The type of the fun expression.

`Expr` `::` `node()` - The node corresponding to the higher-order function call.

`PatternsGuardsBodies` `::` `[{[node()], [node()], [node()]}]` -

The patterns, guards and bodies belonging to the clauses of the fun expression.

`List` `::` `node()` - The second parameter node of the higher-order function call.

- *Return value:*

A list containing a single transformation function.

`transform_recursion_fun_expr/6`. Gathers all variable references to outside variables from the fun expression using `outside_varrefs/1`.

After that creates a list containing two transformation functions. The first one creates the recursive function and inserts it in the module, the second one replaces the higher-order function call with the call to the recursive function.

When creating the recursive function definition, `transform_clause/5` is used to transform each clause of the fun expression.

- *Parameters:*

`Expr` `::` `node()` - The node referring to the higher-order function call.

`Args` `::` `[node()]` - The argument list of the higher-order function, as represented in the SPG.

`File` `::` `string()` - The name and path of the file which is being refactored.

`TransformPatternFun` `::` `fun((node(), [string()]) -> node())` -

A function that can be used to transform a pattern node based on all used variable names in the scope, to a new pattern node.

`TransformLastBodyFun` `::`

`fun((node(), string(), node(), string(), [string()]) -> node())` -

A function that can be used to transform the last expression in the body of a clause. The parameters of this function are the node that needs to be

transformed, the name of the new recursive function, the pattern that binds the head of the list, the variable name referring to the tail of the list, and a list of all the used names in the scope.

Prefix :: `string()` - The prefix of the name of the function that needs to be defined.

- *Return value:*

A list containing two transformation functions that replace the higher-order function call with a call to the newly defined recursive function.

outside_varrefs/1. Gathers all variables referenced, but not bound by a fun expression.

- *Parameters:*

FunExpr :: `node()` - A node referring to a fun expression.

- *Return value:*

A list of nodes, that are the nodes referring to the variables which are used in the fun expression, but are bound in the scope outside the fun expression.

transform_clause/5. This higher-order function performs the transformation of a single clause of a fun expression. The variable names in the recursive function are transformed so that shadowed outside variables are renamed and can be passed on in the recursive call. This is done using the `get_shadow_adjusted_name/3` function.

Transformation of the pattern matching which binds the head of the list is done using the function received in the parameters. The last expression in the body of the clause is also transformed using a function received in the arguments.

- *Parameters:*

Clause :: `node()` - The clause of the fun expression, which is currently being transformed.

FunName :: `string()` - The name of the newly defined recursive function.

OutsideVarNames :: `[string()]` - The names of the variables that are used in the clauses of the fun expression, but are bound outside of it.

TransformPatternFun :: `fun((node(), [string()]) -> node())` -

A function that can be used to transform a pattern node based on all used variable names in the scope, to a new pattern node.

TransformLastBodyFun ::

`fun((node(), string(), node(), string(), [string()]) -> node())` -

A function that can be used to transform the last expression in the body of a clause. The parameters of this function are the node that needs to be transformed, the name of the new recursive function, the pattern that binds the head of the list, the variable name referring to the tail of the list, and a list of all the used names in the scope.

- *Return value:*

The node of the newly transformed clause, that needs to be inserted in the module.

map_transform_pattern/2. Returns a copy of the pattern.

- *Parameters:*

Pattern :: `node()` - The pattern node that has to be transformed.

ShadowAdjustedNames :: `[string()]` - The names of the variables used in the fun expression, that are not bound by the lambda. This variable is unused in this function and is only present so that the same higher-order function can be used for both map and filter.

- *Return value:*

A copy of the **Pattern** node

map_transform_last_body/5. Performs the transformation of the last expression in the body of a clause. Creates the necessary nodes for constructing the result list and performing the recursive call.

- *Parameters:*

LastBody :: `node()` - The node representing the last expression in the body of a clause.

FunName :: `string()` - The node representing the last expression in the body of a clause.

Pattern :: `node()` - The node of the pattern binding the head of the list. This variable is unused in this function and is only present so that the same higher-order function can be used for both map and filter.

TailName :: `string()` - The name of the variable containing the tail of the list.

FunName :: `string()` - The node representing the last expression in the body of a clause.

ShadowAdjustedNames :: `[string()]` - The names of the variables used in the fun expression, that are not bound by the lambda.

- *Return value:*

The node of the transformed expression.

filter_transform_pattern/2. Transforms a pattern so that if it contains a joker anywhere, it will be also bound to a new, unused variable name.

- *Parameters:*

Pattern :: `node()` - The pattern node that has to be transformed.

ShadowAdjustedNames :: `[string()]` - The names of the variables used in the fun expression, that are not bound by the lambda.

- *Return value:*

The transformed version of the **Pattern** node

filter_transform_last_body/5. Performs the transformation of the last expression in the body of a clause. Creates the necessary nodes for constructing the result list and performing the recursive call.

- *Parameters:*

LastBody :: **node()** - The node representing the last expression in the body of a clause.

FunName :: **string()** - The node representing the last expression in the body of a clause.

Pattern :: **node()** - The node of the pattern binding the head of the list.

TailName :: **string()** - The name of the variable containing the tail of the list.

FunName :: **string()** - The node representing the last expression in the body of a clause.

ShadowAdjustedNames :: **[string()]** - The names of the variables used in the fun expression, that are not bound by the lambda.

- *Return value:*

The node of the transformed expression.

get_shadow_adjusted_name/3. Checks if a variable is shadowed in a clause, and if it is creates a new, unique name for it.

- *Parameters:*

Expr :: **node()** - The node of the expression that determines the scope which needs to be checked for used variable names.

OutsideVars :: **[string()]** - The names of the variables that get bound outside the fun expression and may be shadowed by the patterns in the current clause.

PatternVars :: **[string()]** - The names of the variables bound by the pattern in the current clause.

- *Return value:*

A list created from `OutsideVars`, where the name of each shadowed variable has been changed to a new, unique name.

error_text/2. The possible `LocalErrors` thrown during this refactoring are the following:

- `bad_application`: The selected expression is not a function application.
- `bad_hof`: The selected function application is not a call to either `lists:map/2` or `lists:filter/2`.
- `invalid_fun_arg`: The first argument of the higher-order function is neither a lambda nor an implicit fun.
- `pattern_num`: The arity of the fun expression is incorrect.
- `implicit_arity`: The arity of the implicit fun is incorrect.
- `ambiguous`: A query, that should only return a single value, instead returns either multiple values or none at all. Normally, if the SPG is in a correct state this error should never happen.

3.4 Testing

There are two ways the refactorings can be tested. The first is to create unit tests to make sure all refactorings work correctly and perform the desired transformation.

The second way is to check if the refactorings have achieved the goal they were designed to reach. The final goal of all the refactorings shown in this thesis was to improve the energy efficiency of Erlang software. This can be tested using the GreenErl energy measurement framework.

3.4.1 Unit testing

RefactorErl provides a way to easily implement and run unit tests. Besides showing the results of running a test in the shell, it also generates a report in the form of an HTML document after running the unit tests. This `result.html` file can be found in the `trunk/tool/testresult/` directory.

These tests are placed in the `trunk/test/refact` directory. For every refactoring a new folder has to be created, which contains all the unit tests for that particular transformation. The name of this new folder is the same as the name of the module containing the refactoring, without the `reftr_` prefix, for example in the case of my refactorings the directories are: `elim_get_value`, `transform_list` and `elim_hof`.

To run the unit tests belonging to a refactoring, open a terminal, start the RefactorErl shell and use the following command, where `REFACTOR_NAME` is the name of the module containing the refactoring, without the `reftr_` prefix.

```
reftest_refact:run([test,REFACTOR_NAME]).
```

The structure of a unit test

Each unit test is contained in its own directory. This directory contains the Erlang source file that has to be refactored by the test case. It also contains a file called `TEST`, which describes the test case and gives the arguments for the refactoring, such as the filename and position. These are the arguments that the refactoring is going to receive in its `Args` parameter. An example of this `TEST` file is shown on Figure 3.20. Finally, the directory of a test case also contains a folder named `'out'`. This folder contains an Erlang source file with the expected result of the refactoring.

Eliminate `proplists:get_value`

Test for this refactoring can be run from the RefactorErl shell using the following command:

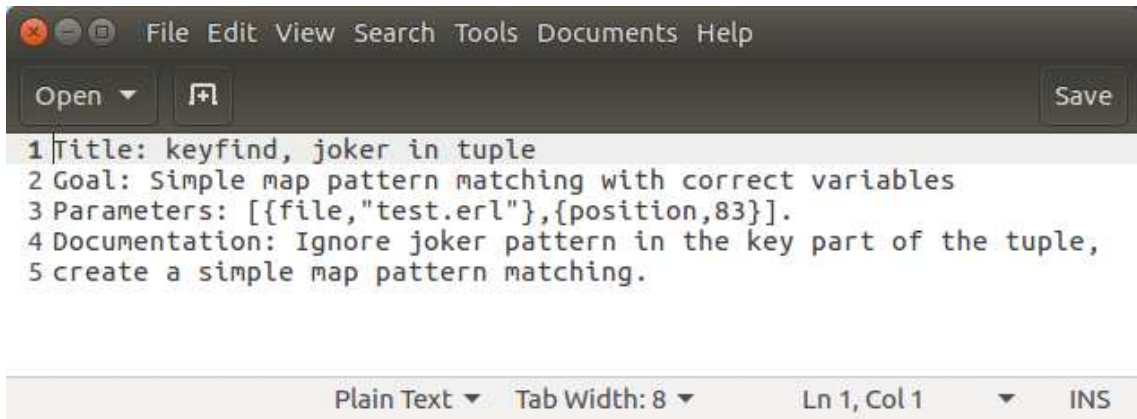


Figure 3.20: An example showing the unit test description used in RefactorErl

```
reftest_refact:run([test,elim_get_value]).
```

One test case is shown as an example on Figure 3.21. In this example the binary `proplists:get_value` is called, and it is the function call that is being selected for refactoring.

```

1 -module(getvalue).
2
3 f() ->
4   proplists:get_value(2, [{1,2},{2,3}]).

```

(a) Input

```

1 -module(getvalue).
2
3 f() ->
4   case lists:keyfind(2, 1, [{1, 2}, {2, 3}]) of
5     false -> undefined;
6     {_, Var1} -> Var1
7   end.

```

(b) Expected output

Figure 3.21: Input and expected output of a test case for the 'eliminate `get_value`' refactoring.

There are a total of 9 test cases created for this refactoring. The types of these test cases are the following:

- Refactor `proplists:get_value/2`, the binary version

- Refactor `proplists:get_value/3`, the ternary version
- The function name is the selected part
- The module name is the selected part
- The `proplist` module is imported
- Another function with the same name is created and it is called.

Running the unit tests results in a log in the RefactorErl shell (see Figure 3.22), which specifies how many test cases have passed and how many have failed and for what reason. An HTML summary of the testing is also created, which describes each test case, and gives the reason in case of a failure. Part of this HTML log is shown on Figure 3.23.

```

Terminal File Edit View Search Terminal Help
--==<##### elim_get_value/09 #####>
Title: own get_value defined
13 refr_elim_get_value
    Args: [{position,36},
           {file,"/home/nagygeri97/Documents/Egyetem/TKD/SVN/green_erlang/t
runk/tool/tmp/getvalue.erl"},
           {ask_missing,false}]
The test is acceptable. Diff test result:
No difference between source files!

Summary:
  All testcases: 9
Accepted testcases: 6
Errors in test input: 0
Errors in refactoring: 0
Errors in results: 0
Errors in graph: 3
Unsupported testcases: 0
Warnings: 0

Test(s) finished.
Testing took 2 seconds
Results: /home/nagygeri97/Documents/Egyetem/TKD/SVN/green_erlang/trunk/tool/test
result/result.html
ok
(refactorerl@localhost)13>

```

Figure 3.22: The output of running a test case, and the summary of all `elim_get_value` tests in the RefactorErl shell.

Note on failing test cases. There are in this test suite 3 cases, in which RefactorErl does not accept the result of the test case as correct. The reason for this is that when checking the output of a refactoring, RefactorErl not only checks the

Test log

Last running: 2019.05.29 16:10
Parameters of last running: [{test,elim_get_value}]

[Summary](#)

Test case name: 'elim_get_value/01'
Test case title: Funname selected, get_value/2

The test is acceptable. Diff test result:
Accepted. No difference between source files!

Test case name: 'elim_get_value/02'
Test case title: Modname selected, get_value/2

The test is acceptable. Diff test result:
Accepted. No difference between source files!

Test case name: 'elim_get_value/03'
Test case title: Funname selected, get_value/3

The test is acceptable. Diff test result:
Accepted. No difference between source files!

Test case name: 'elim_get_value/04'
Test case title: Modname selected, get_value/3

The test is acceptable. Diff test result:
Accepted. No difference between source files!

Figure 3.23: The beginning of the result.html file after running all elim_get_value test cases.

syntax tree of the program, but also checks the correctness and consistency of the semantical nodes and edges in the SPG. In my refactorings I only manipulate the syntax tree of the programs, and it is the responsibility of RefactorErl to insert the correct semantical information in the graph. In the case of the failing test cases, the output of the testing is the following:

```
Accepted. No difference between source files!  
BUT, the result graph or CFG graph is not consistent!
```

This means that the test case is failing not because the transformation is not working correctly on the level of the syntax tree, but because the insertion of semantical information to the graph is not correct.

Transforming list to map

To run the unit tests for this refactoring the following command can be used from the RefactorErl shell:

```
reftest_refact:run([test,transform_list]).
```

There are a total of 38 test cases for this refactoring. One of the more complex test cases is shown on Figure 3.24 as an example. This example shows a test case where the transformed variable is used in multiple function calls, an empty list pattern is used in one of the clauses and the return value of one of the clauses is the list itself.

There are test cases created for all of the following scenarios:

- The recursive function contains a single call to a function that is supported by the refactoring. These function calls are the following:
 - lists:keyfind/3
 - lists:keydelete/3
 - lists:keymember/3

```

1 -module( test ).
2
3 recursive( _Key, [] ) -> ok;
4 recursive( Key, List ) ->
5     {K, V} = lists:keyfind( Key, 1, List ),
6     NewList = lists:keydelete( K, 1, List ),
7     NewList2 = lists:keystore( K, 1, NewList, {K, V} ),
8     {value, _, NewList3} = lists:keytake( Key, 1, NewList2 ),
9     recursive( Key + 1, NewList3 ),
10    List .

```

(a) Input

```

1 -module( test ).
2
3 recursive( _Key, Map1 ) when map_size( Map1 ) == 0 -> ok;
4 recursive( Key, List ) ->
5     begin
6         K = Key, #{K:=V} = List
7     end,
8     NewList = maps:remove( K, List ),
9     NewList2 = NewList#{K=>V},
10    {value, _, NewList3} = case maps:take( Key, NewList2 ) of
11        error -> false;
12        {Value1, List1} -> {value, {Key, Value1}, List1}
13    end,
14    recursive( Key + 1, NewList3 ),
15    maps:to_list( List ).

```

(b) Expected output

Figure 3.24: Input and expected output of a test case for the 'transform list to map' refactoring.

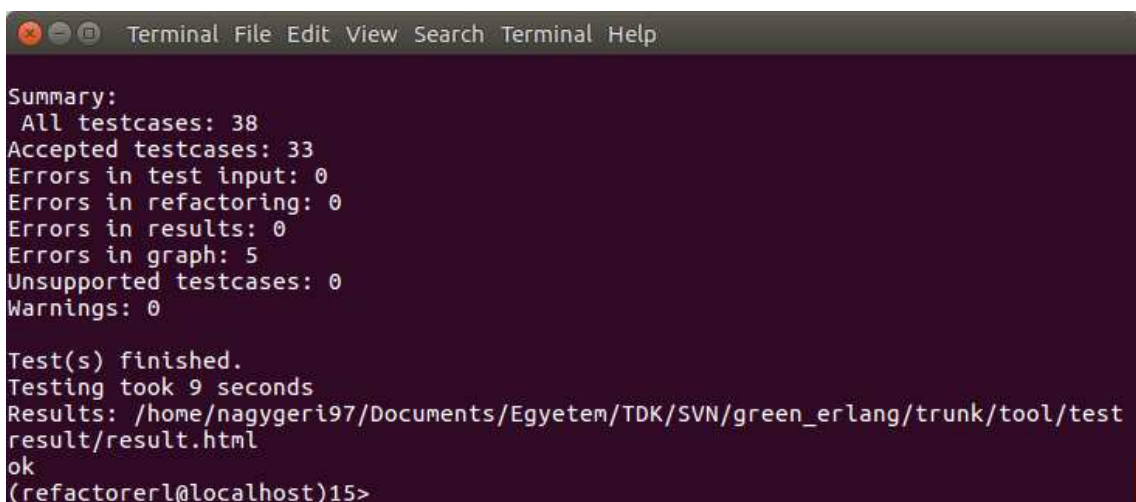
- lists:keystore/4
- lists:keytake/3

For each possibility there are multiple test cases so that every branch of the refactoring algorithm is tested by at least one test case. This includes, for example, binning the result of `lists:keyfind/3` to different constructions or not even using the return value of it.

Each function call is also tested when it is placed as the return value of the function.

- o Transforming the calls to the recursive function.
- o Using an empty list pattern in one of the clauses.
- o More complex test cases containing multiple expressions that need to be refactored, to test the different transformations working together.

The summary given by the RefactoErl shell when running these test cases is shown on Figure 3.25. Similarly to the previous refactoring, this test suite also has some failing test cases because of the reasons explained previously.



```
Terminal File Edit View Search Terminal Help
Summary:
  All testcases: 38
  Accepted testcases: 33
  Errors in test input: 0
  Errors in refactoring: 0
  Errors in results: 0
  Errors in graph: 5
  Unsupported testcases: 0
  Warnings: 0

Test(s) finished.
Testing took 9 seconds
Results: /home/nagygeri97/Documents/Egyetem/TDK/SVN/green_erlang/trunk/tool/test
result/result.html
ok
(refactorerl@localhost)15>
```

Figure 3.25: The summary of all `transform_list` tests in the RefactorErl shell.

Eliminating higher-order functions

The unit tests for this refactoring can be run from the RefactorErl shell using the following command:

```
reftest_refact:run([test,elim_hof]).
```

There are a total of 42 test cases in this test suite. One test case is shown on Figure 3.26 as an example. The example shows a `map` function call using a lambda with multiple clauses and using external variables, some of which are shadowed in certain clauses.

There are test cases to check the correct behaviour of every possible branch in the algorithm, specifically, the following scenarios are tested:

- Testing the transformations for both the `lists:map/2` and `lists:filter/2` higher-order functions.
- Testing the use of list comprehensions and recursion.
- Testing the use of lambdas and implicit fun expressions.
- In the case of lambdas:
 - Joker pattern used
 - Single clause or multiple clauses
 - Single or multiple expressions in the body of a clause
 - External variables used, sometimes shadowed.
- Checking the uniqueness of newly introduced variables.
- The new recursive function has a unique name.

The summary created by the RefactorErl shell after running all the test cases for this refactoring is shown on Figure 3.27.

```

1 -module(test).
2 -export([f/2]).
3
4 f(List, Y) ->
5     X = 2, Z = 3, A = 4,
6     lists:map(fun(Elem) when Elem == 42 ->
7         Z = X + Y,
8         Elem + Z,
9         Q = 2;
10        (X) -> X1 = X + Y;
11        ({X,Y}) -> oh end, List).

```

(a) Input

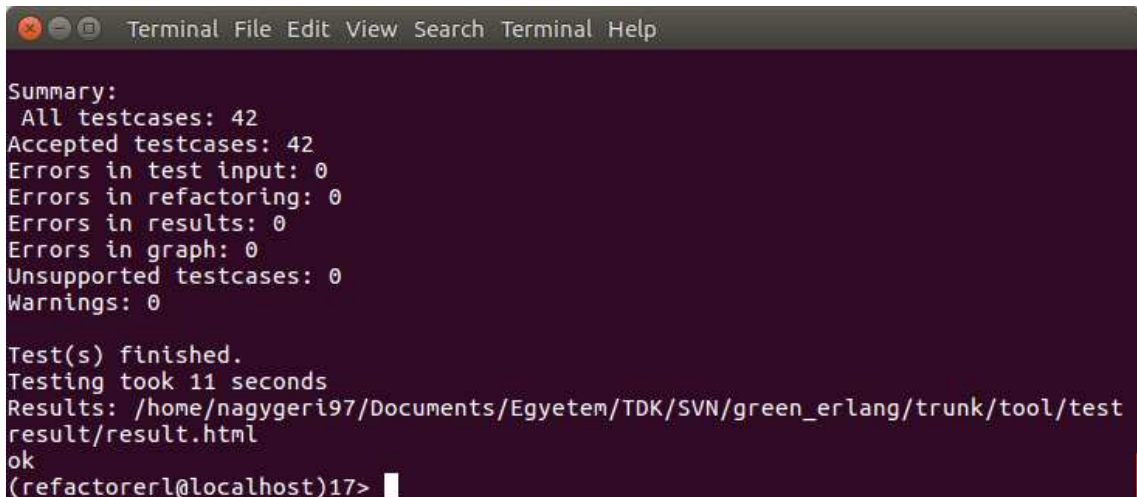
```

1 -module(test).
2 -export([f/2]).
3
4 f(List, Y) ->
5     X = 2, Z = 3, A = 4,
6     map_recursion1(List, X, Y, Z).
7
8 map_recursion1([], _, _, _) -> [];
9 map_recursion1([Elem | Tail1], X, Y, Z) when Elem == 42 ->
10    Z = X + Y,
11    Elem + Z,
12    [Q = 2 | map_recursion1(Tail1, X, Y, Z)];
13 map_recursion1([X | Tail1], X2, Y, Z) ->
14    [X1 = X + Y | map_recursion1(Tail1, X2, Y, Z)];
15 map_recursion1([X, Y | Tail1], X1, Y1, Z) ->
16    [oh | map_recursion1(Tail1, X1, Y1, Z)].

```

(b) Expected output

Figure 3.26: Input and expected output of a test case for the 'eliminate higher-order function' refactoring.

A terminal window with a dark background and light text. The window title is "Terminal File Edit View Search Terminal Help". The output shows a summary of test results for 'elim_hof' tests. The summary includes: All testcases: 42, Accepted testcases: 42, Errors in test input: 0, Errors in refactoring: 0, Errors in results: 0, Errors in graph: 0, Unsupported testcases: 0, Warnings: 0. Below the summary, it says "Test(s) finished.", "Testing took 11 seconds", and "Results: /home/nagygeri97/Documents/Egyetem/TDK/SVN/green_erlang/trunk/tool/test result/result.html". The prompt "(refactorerl@localhost)17>" is visible at the bottom.

```
Terminal File Edit View Search Terminal Help
Summary:
  All testcases: 42
  Accepted testcases: 42
  Errors in test input: 0
  Errors in refactoring: 0
  Errors in results: 0
  Errors in graph: 0
  Unsupported testcases: 0
  Warnings: 0

Test(s) finished.
Testing took 11 seconds
Results: /home/nagygeri97/Documents/Egyetem/TDK/SVN/green_erlang/trunk/tool/test
result/result.html
ok
(refactorerl@localhost)17>
```

Figure 3.27: The summary of all `elim_hof` tests in the RefactorErl shell.

3.4.2 Testing energy consumption

To test the effect of the refactorings on energy consumption, the GreenErl framework for energy measurement was used. For each refactotring, I created a use case, in which the refactoring can be applied. The energy consumption of the program was measured before and after refactoring.

Eliminating `proplists:get_value` calls

The program used to measure the effect of the 'eliminate `proplists:get_value`' refactoring is shown on Figure 3.28. This test program consists of a single recursive function which takes three parameters: a key, a proplist and an integer specifying how many times to perform the recursive call. All the function does is look up a value in a proplist belonging to a given key. The energy consumption of this function was measured with lists of length ranging from 10 000 to 10 000 000 and repeat count (N) 100. The key the program was looking for in the list was always at the exact middle of the data structure.

The after version of the test program was created using the refactoring I implemented in RefactorErl.

Figure 3.29 shows the energy consumed by the test program both before and after

```

1 f(_, _, 0) -> ok;
2 f(Key, List, N) ->
3   proplists:get_value(Key, List),
4   f(Key, List, N - 1).

```

(a) Before

```

1 f(_, _, 0) -> ok;
2 f(Key, List, N) ->
3   case lists:keyfind(Key, 1, List) of
4     false -> undefined;
5     {_, Var1} -> Var1
6   end,
7   f(Key, List, N - 1).

```

(b) After

Figure 3.28: Before and after versions of the test program using `proplists:get_value`.

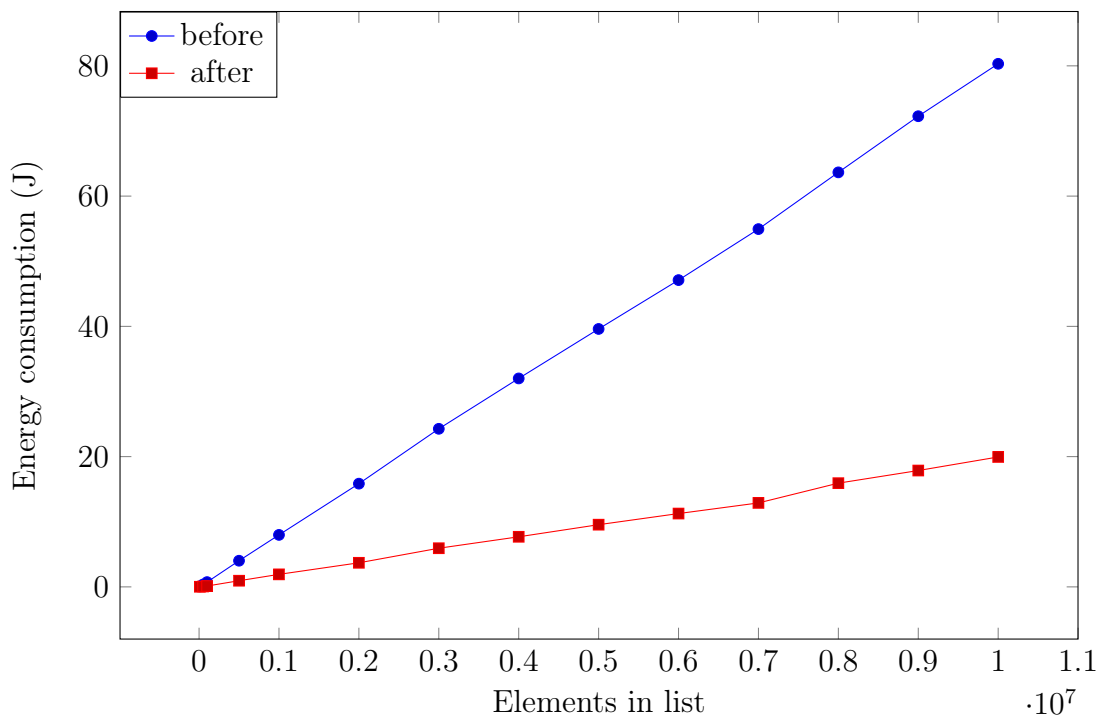


Figure 3.29: Energy consumption values of the before and after versions of the test program using `proplists:get_value`.

applying the refactoring. The after version consumes significantly less energy compared to before refactoring. This shows that applying this refactoring to a program can improve energy consumption values, as intended.

Transforming a list to map

```

1 main(List) -> recursive(List, 0).
2
3 recursive([], _) -> ok;
4 recursive([], _) -> ok;
5 recursive(List, Key) ->
6   {_, Value} = lists:keyfind(Key, 1, List),
7   NewList = lists:keystore(Key, 1, List, {Key, Value + 1}),
8   {value, _, NewList2} = lists:keytake(Key, 1, NewList),
9   recursive(NewList2, Key + 1).

```

(a) Before

```

1 main(List) -> recursive(maps:from_list(List), 0).
2
3 recursive(#{}, _) -> ok;
4 recursive(List, Key) ->
5   #{Key:=Value} = List,
6   NewList = List#{Key=>Value + 1},
7   {value, _, NewList2} = case maps:take(Key, NewList) of
8     error -> false;
9     {Value1, List1} -> {value, {Key, Value1}, List1}
10  end,
11  recursive(NewList2, Key + 1).

```

(b) After

Figure 3.30: Before and after versions of the test program for transforming lists to maps.

The program shown on Figure 3.30 was created to measure the effect of transforming a list to a map in a recursive function call. The after version of the test program was created using the 'transform list to map' refactoring.

The energy consumption of calls to the `main/1` function was measured using lists that contained from 100 to 20 000 key-value pairs. These proplists contained the tuples with keys ranging from 1 to the length of the list. The elements of the list were shuffled using the same seed every time to get comparable results for the before

and after versions.

The `recursive/2` function for a given key looks up the value for that key, modifies the list so that the value is incremented by one, then actually deletes the value belonging to the key, thus decreasing the size of the list. We repeat this until all elements of the list have been deleted. This will happen for sure, because the keys are ranging from 1 to the length of the list, first the key given to the recursive function is 0 and it gets incremented by one after each iteration.

All function calls inside the recursive function use the list parameter in a way that the refactoring is able to transform to an equivalent expression using maps. Note that in the after version the `main/1` function still gets a list as its parameter and only when the call to `recursive/2` happens does it convert the list to a map.

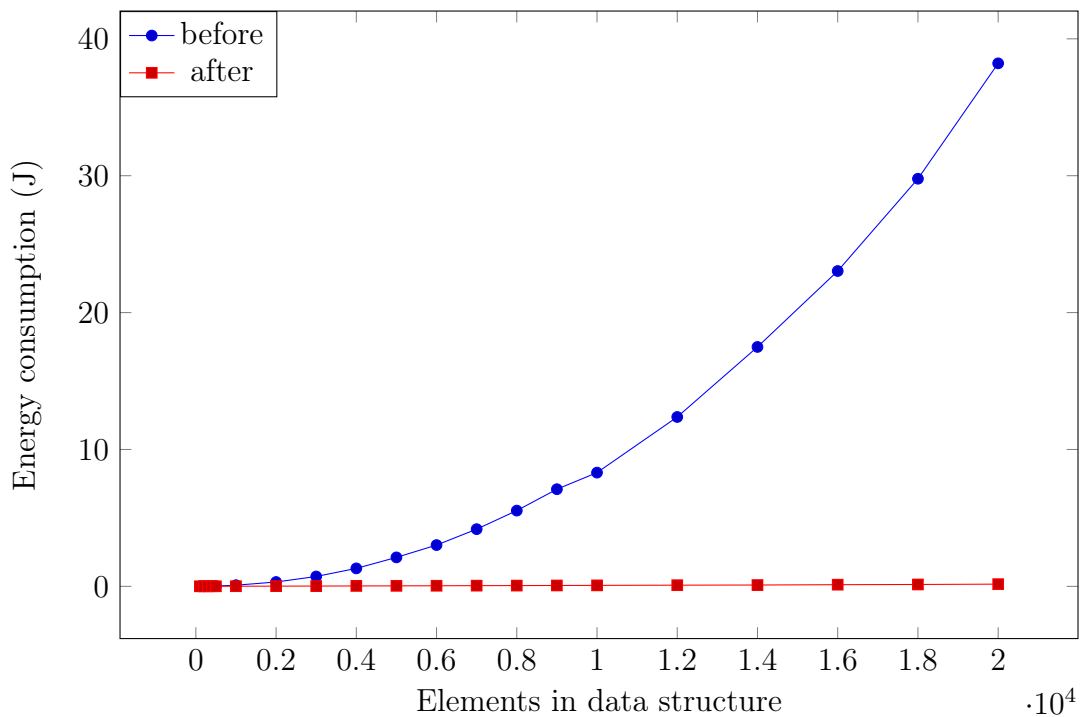


Figure 3.31: Energy consumption values of the before and after versions of our test program for transforming lists to maps.

The energy consumption values for the test program before and after refactoring are shown on Figure 3.31. This graph shows that using a map consumes a really small amount of energy compared to the energy consumption of the before version, using lists. This shows that transforming a list to a map is worth it with regards to

energy consumption and that are refactoring is working as intended.

Eliminating higher-order function calls

The test programs for measuring the effect of the 'eliminate higher-order functions' refactoring is shown on Figure 3.32. In this case there are two refactored versions. In one of the refactored versions only list comprehensions are used to eliminate the higher-order function calls. In the other one only recursion is used to eliminate higher-order functions.

The test program consists of three higher-order function calls, that operate on the lists created by the previous HOF call. Both map and filter are used, as well as implicit fun expressions and lambdas.

The energy consumption values of the before and both after versions are shown on Figure 3.33. This graph shows that the energy consumption improved in both cases. There is no significant difference in this case between refactoring using list comprehensions or using recursive functions. This decrease in energy consumption shows that both methods for eliminating higher-order functions are working as intended, and that these refactorings are beneficial for the energy consumption of Erlang programs.


```

1 do_stuff(X) ->
2   A = X, B = X*X, C = X*X*X, A + B + C.
3 even(X) -> X rem 2 == 0.
4 f(List) -> Y = 42,
5   NewList = lists:map(fun(X) -> case even(X) of
6     true -> Y;
7     false -> X end end, List),
8   NewList2 = lists:filter(fun(X) -> X /= Y end, NewList),
9   lists:map(fun do_stuff/1, NewList2).

```

(a) Before

```

1 f(List) -> Y = 42,
2   NewList = [case even(X) of
3     true -> Y;
4     false -> X
5   end || X<-List],
6   NewList2 = [X || X <- NewList, X /= Y],
7   [do_stuff(Elem1) || Elem1 <- NewList2].

```

(b) After - list comprehensions

```

1 f(List) -> Y = 42,
2   NewList = map_recursion1(List, Y),
3   NewList2 = filter_recursion1(NewList, Y),
4   map_recursion2(NewList2).
5 map_recursion1([], _) -> [];
6 map_recursion1([X | Tail1], Y) ->
7   [case even(X) of
8     true -> Y;
9     false -> X
10  end | map_recursion1(Tail1, Y)].
11 filter_recursion1([], _) -> [];
12 filter_recursion1([X | Tail1], Y) ->
13   case X /= Y of
14     true -> [X | filter_recursion1(Tail1, Y)];
15     false -> filter_recursion1(Tail1, Y)
16   end.
17 map_recursion2([]) -> [];
18 map_recursion2([Head | Tail]) ->
19   [do_stuff(Head) | map_recursion2(Tail)].

```

(c) After - recursion

Figure 3.32: Before and after versions of our test program for eliminating higher-order functions. Helper functions are only shown in the before part, but are defined in both of the after versions.

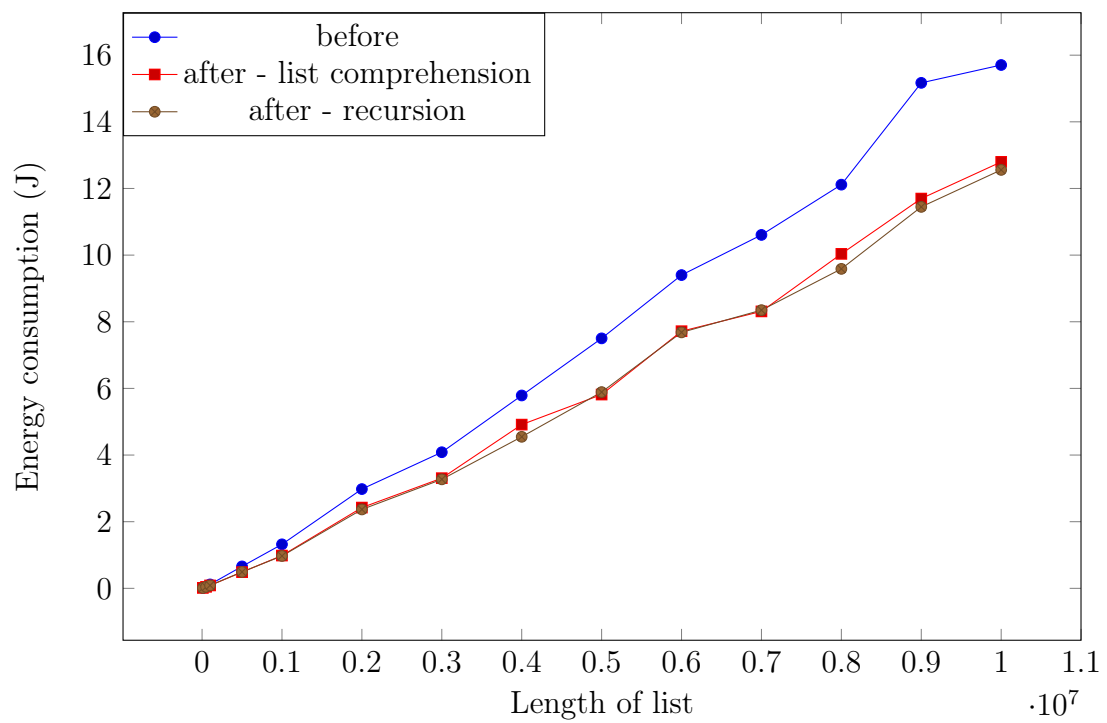


Figure 3.33: Energy consumption values of the before and after versions of our test program for transforming lists to maps.

Chapter 4

Summary and Conclusions

The problem I solved in this thesis is to refactor Erlang programs so that the energy consumption of the program may decrease. The motivation for these energy conscious refactorings was our previous TDK thesis [1] in which we conducted measurements regarding the energy consumption of Erlang programs, and based on the measurements identified some areas that could be worth refactoring.

The topics of the refactorings are the following:

- Eliminate calls to the `proplists:get_value` functions, and use the function `lists:keyfind/3` instead.
- Transform a list to a map data structure, when used in a recursive function.
- Eliminate calls to higher-order functions.

My task was to analyze the problem and create an algorithm for performing these transformations. For each refactoring I have created a detailed algorithm, describing how each transformation can be done and how the context in which a language construct is used influences the refactoring. I also identified the necessary preconditions for each refactoring.

Finally, I implemented the three refactorings using the static code analysis tool

RefactorErl [4]. These implementations have been tested, so that they work correctly. Also, the effect of each refactoring on energy consumption was measured, and I found that these refactorings can decrease the energy consumption of Erlang programs, so the initial goal has been achieved.

4.1 Further development possibilities

Further development possibilities include also creating a refactoring so that not only lists can be transformed to maps, but dictionaries as well. The reason for this is that measurements of energy consumption show that using a dictionary is more energy efficient than using a map.

Another area for possible future development is the elimination of higher-order functions. There are more higher-order functions included in the `lists` module, such as `lists:foldr/3` and `lists:all/2`. These could also be made part of my refactorings, so that calls to these higher-order functions can also be eliminated.

Bibliography

- [1] A.A. Meszaros, G. Nagy, I. Bozo, and M. Toth. Greenerl, measuring the energy consumption of erlang programs and energy conscious refactorings, May 2019. TDK thesis.
- [2] Ericsson AB. Erlang Programming Language. <http://www.erlang.org>. [Accessed: 2019.05.25.].
- [3] Fred Hebert. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 2013.
- [4] M. Tóth and I. Bozó. Static analysis of complex software systems implemented in erlang. Central European Functional Programming Summer School – Fourth Summer School, CEFPS 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.
- [5] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, ISBN 978-9949-23-178-2, pages 138–148, Tallin, Estonia, October 2011.
- [6] Eötvös Loránd University, ELTE-Soft Ltd., and Ericsson Hungary. Refactorerl. <http://plc.inf.elte.hu/erlang/>, 2015. [Accessed: 2019.05.24.].
- [7] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Thomas Nagy, Melinda Tóth, and Roland Király. Building a refactoring tool for erlang. In *Workshop on Advanced Software Development Tools and Techniques*, 05 2008.

- [8] Srinivas Pandruvada. RUNNING AVERAGE POWER LIMIT - RAPL. <https://01.org/blogs/2014/running-average-power-limit---rapl>, 2014. [Accessed: 2019.03.27.].
- [9] Free Software Foundation Inc. Gcc, the gnu compiler collection. <http://www.gnu.org/>, 2019. [Accessed: 2019.05.24.].
- [10] Ericsson AB. Erlang programming language. <https://www.erlang.org/>, 2019. [Accessed: 2019.05.24.].
- [11] Graphviz - graph visualization software. <https://www.graphviz.org/>, 2019. [Accessed: 2019.05.24.].
- [12] Free Software Foundation Inc. Gnu emacs. <https://www.gnu.org/software/emacs/download.html>, 2019. [Accessed: 2019.05.24.].
- [13] Erlang AB. Mnesia reference manual. <http://erlang.org/doc/apps/mnesia/>, 2019. [Accessed: 2019.05.24.].
- [14] FAL Labs. Kyoto cabinet: a straightforward implementation of dbm. <https://fallabs.com/kyotocabinet/>, 2012. [Accessed: 2019.05.24.].
- [15] Manual page of refactorerl. <http://pnyf.inf.elte.hu/trac/refactorerl/>, 2019. [Accessed: 2019.05.24.].
- [16] Erlang AB. proplists. <http://erlang.org/doc/man/proplists.html>, 2019. [Accessed: 2019.05.04.].
- [17] Erlang AB. lists. <http://erlang.org/doc/man/lists.html>, 2019. [Accessed: 2019.05.04.].