Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Erlang refaktor eszköz

Nagy Tamás

Társszerző: Víg Anikó

Konzulens: Dr. Horváth Zoltán

Budapest, 2007

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

Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Erlang refaktor eszköz

Víg Anikó

Társszerző: Nagy Tamás

Konzulens: Dr. Horváth Zoltán

Budapest, 2007

Eötvös Loránd University
Faculty of Informatics
Programming Languages and
Compilers Department

Erlang refactor tool

Tamás Nagy and Anikó Víg

Supervisor: Dr. Horváth Zoltán

Budapest, 2007

CONTENTS

INTRODUCTION

An average trend in every programming language is the emergence of claim refactoring tools. Such tools grant safe and fast restructuring and transforming of the code to improve the effectiveness of the programmers.

The use of functional programming languages are also growing in the industry. There is a big need for refactoring tools in this area too. Our project group's goal is to plan and implement an application for the Erlang language. We choose to use a database to store the source and the semantical information based on the work in the Clean language [7]. This idea is used only in our university (ELTE), and we ported it to Erlang and also extended it to store the syntactical and semantical information. The main differences of the two implementations are the editor: the Clean refactorer has a syntactical based editor; and the languages: both of them is functional, but Erlang is a mostly dynamic language and variables do not have static types. However, we are using Emacs because it seems most Erlang programmers use this as their editor of choice. We are working together with the University of Kent in the United Kingdom to make a common tool, where an other Erlang refactoring tool is being produced, but using a different approach [17]: they are using annotated abstract syntax trees instead of a relational database. Both approaches use original ideas and appears to be correct as the results of the tests show. In the future we hope to conduct a survey to find the advantages and disadvantages of each approach; it would be interesting to observe which refactoring works best under the different approaches. Although we have some common refactorings (rename function, rename variable) the remaining three refactorings are designed and implemented just with the database approach.

Our role in the project is to plan the database, which stores the source code; the algorithm for the storing and recovering the source code to/from the database; the methods of implementation for the refactorings, based on the already written test cases. We searched and chose the components of the refactoring environment to build an effective and usable system. After

it we implemented and tested the program with more than 200 test cases. The size of the general modules, which needed for every refactor step, are approximately 15000 lines of Erlang source code, and the size of each current refactor steps are 300-500 lines.

The theme of the thesis is the five working refactorings; where the definition, analysis of the problem and for the two most difficult refactorings (reorder function arguments and eliminate variable) efficiency analysis are given. It shows the various stages of the work from the planning to testing phase with descriptions of the tool and the implementation.

In the first chapter we give a short overview of the existing works in the refactor theme, our basic ideas and the structure of our tool, and the representation of the database. We present how optimise the use of the tables the database for storing and recovering the source. The next part contains the main theoretical result, the new ideas, definitions and conditions of the refactorings and the properties of the Erlang language. In the next chapter we describe the carrying out: the algorithms and the implementation of the refactorings. In the last chapter we give efficiency analysis for two refactorings.We present what problems occur during the analysis, and how we succeed to correct them.

Aniko Vig is responsible for the tuple function arguments and the reorder function arguments refactorings. Tamas Nagy is responsible for the rename function, rename variable and the eliminate variable refactorings. It means the design of the algorithms, the implementation and also the efficiency analysis of them.

We marked in the contents, who wrote or edited the current parts. In the first chapter, the main base of the text was already written during previous publications and reports of the project group, we just updated, completed and edited it. For example we added a new figure for the structure of the tool. The main parts of the thesis, the second are the third chapter is our own product.

1. INTRODUCTION TO THE REFACTOR TOOL

1.1 Refactor

The term "refactoring" is for program transformation that preserves the meaning of the program [1]. Such transformations are often applied in order to improve the quality of program code: make it more readable, satisfy coding conventions, prepare it for further development, etc. Simple refactorings are used by developers almost every day; they rename variables, introduce new arguments to functions, or create new functions from duplicated code. The object-oriented paradigm is especially well suited for refactoring-supported programming. In this area refactoring has already appeared in programming methodologies [9] and it is used heavily in the industry.

In old-fashioned programming environments, refactorings have been applied manually by the programmer. This requires the application of systematic changes on the program text, which is hard to accomplish. It is also very error-prone, even for simple transformations like renaming a variable. If the standard search&replace function, that is found in every text editor, is used, false replacements are likely to occur. Semantical analysis is required to correctly identify the occurrences of a variable, which is not provided by a simple editor. Furthermore, the renaming of a variable should be avoided if its new name conflicts with another variable.

However, it *is* possible to perform most refactorings in an automated way with an appropriate tool—a tool that is aware of the syntactic and semantical rules of the programming language in use. Such tools exist for many programming languages, see in Fowler's work [2], and modern programming environments often incorporate such capabilities, for example Eclipse [10, 11]. Refactoring in functional languages is not really wide-spread yet, but there are many ongoing researches on the topic. For the functional programmer, the only full-featured refactoring tool is HaRe [3, 18], which provides refactoring capabilities for Haskell programs within the Emacs [4] and VIM [5] editors. A prototype of a refactoring tool for Clean is also available [6, 7].

The work presented in this document has an approach similar to that of that prototype tool.

The goal of our project is to create a refactoring tool for the functional programming language Erlang (and for the Erlang/OTP environment). The focus is on the method of extracting static, semantic information from programs written in a rather dynamic language. This information is the key for ensuring the safety of refactorings. A transformation is considered safe if it does not change the meaning of the program. The refactoring tool should help the programmer prevent unsafe transformations. The hardest part in designing refactorings is to formulate the conditions to ensure that the refactoring preserves the behaviour of the program. When the programmer requests a refactoring step to be performed on a program, the refactoring tool has to check this condition. If the transformation proves to be unsafe, the tool must refuse it, or offer "compensation" steps to make it safe. If the check succeeds, refactoring can take place, but even in that case the tool might issue a warning message (or an interactive tool might ask for confirmation interactively) if the transformation were likely to reduce the quality (e.g. the readability) of the code.

A novelty of the presented approach is that the strong dynamic nature of Erlang programs is handled by static analysis, and that programs are represented, stored and manipulated in a relational database. This feature makes it possible to express refactorings in a fairly compact and comprehensible way.

1.1.1 Refactoring in Erlang

Erlang/OTP [8] is a functional programming environment developed by Ericsson, designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics (like telecommunication systems). The core Erlang language consists of simple functional constructs extended with message passing to handle concurrency, and OTP is a set of design principles and libraries that supports building fault-tolerant systems [12]. The language has a very strong dynamic nature that partly comes from concurrency and partly from dynamic language features.

From the refactoring point of view, the most important characteristic of a programming language is the extent of semantical information available by static analysis. (Static code analysis is the analysis of computer software that is performed without actually executing programs.) As Erlang is a functional

language, most language constructs can be analysed easily: side effects are restricted to message passing and built-in functions (Built-in functions, or BIFs, are functions that are implemented in the runtime system.), variables are assigned a value only once, and the code is organised into modules with explicit interface definitions and static export and import lists. An unusual feature (at least in a functional language) is that variables are not statically typed, they can have a value of any data type.

On the other hand, the remaining few constructs offer a real challenge to static analysis. An example of this is matching corresponding message send and receive instructions. A destination of a message can be a process ID or a registered name, which are bound to function code at runtime. Data flow analysis might help in discovering these relations, but it is beyond the scope of this project.

Another kind of problem is the possibility of running dynamically created code. This functionality is clearly out of the scope of a static refactoring tool, but there are other constructs similar to this that are widely used and should be covered, like the `spawn` function that starts the evaluation of a function in a new process (and the function name and arguments might be constructed at runtime), or the `apply` function that calls a function or the `hibernate` function that puts the calling process into a wait state until a message is sent to it (with the same runtime-related problems). The normal function call syntax has some runtime features too: variables are allowed instead of static module or function names.

It is very important to exactly define what part of the language is covered by our refactoring tool. Due to the simple syntax and relatively small number of constructs in Erlang, full syntactical coverage of the language is feasible, which is a key point in real life usability. However, semantical coverage seems to be achievable only to a lesser extent. A third aspect besides the syntax and semantics of the core language is library coverage: `hibernate`, `spawn` and `apply` could be handled just like any other function call, but special support for them seems useful. OTP libraries fall in the same category.

Example refactorings

Refactoring is a programming technique for improving the design of a program without changing its behaviour. Refactoring may precede a program modification or extension, preparing the program for the modification, or may be used after finishing the work in order to bring the program into a

more improved shape. The transformations of refactoring can be used for optimisation as well. In this case the programmer writes a basic specification of the code, and improves its performance by applying refactorings.

Typical refactorings are:

- **Rename variable:** Find every occurrence of the variable (i.e. the variables with the same name in the scope) and replace every occurrence with the new name.
- **Rename function:** This refactoring finds the definition and every place of a call for a given function and substitutes it with a new name.
- **Rename module:** Find every occurrence of the module name and replace them with the new name. In Erlang the module name and the file name has to be identical. In this refactor step the filename of the current module has to be changed.
- **Reorder function arguments:** Change the order of arguments in the same way at the definition and every place of call for a given function.
- **Merge subexpression duplicates** All instances of the same subexpressions are stored in a variable defined by the user, then all instances of the original subexpression are changed to the variable.
- **Eliminate variable** All instances of a variable are replaced with its bound value in that region where the variable is visible. The variable can be removed where its value is not used.
- **Tupling function arguments:** Change the structure of some function arguments at the definition or at some place of call for a given function by grouping some arguments into one tuple argument; every occurrence of the argument list will be changed.
- **Extract function:** An alternative of a function definition might contain a sequence of expressions which can be considered as a logical unit, hence a function definition can be created from it. The extracted function is lifted to the module level, and it is parameterised with the variables that the expressions depend on. The sequence of expressions is replaced with a function call expression.

And some more complex ones: **specialisation of functions**, **generalisation of functions**, **fusion of functions**, **modification of data structures (for example tuple to record)**.

Creating a refactoring

In order to implement safe transformations for Erlang programs we need to check the preconditions of the refactorings and make compensations (if necessary) to ensure the correctness of the transformations. An appropriate method for syntactic and a static semantic analysis is needed to be performed prior to the desired transformations.

Refactoring is encouraged to be performed incrementally in small steps making small changes at once. After each step the programmer must be sure that the behaviour of the code, at least from the "black box" point of view, has not been changed. It is developing an automated interactive refactoring tool to carry out some of these tedious steps safely.

We are going to develop such an interactive environment, where one can incrementally carry out programmer-guided, meaning-preserving, program transformations in functional languages.

Syntactical and semantical information is to be stored in a set of related abstract syntax and semantic tables (a relational database).

To ensure correct transformations, preconditions of transformations will be checked with the help of the database tables. In some cases when the preconditions do not hold, we will interactively offer compensation transformations which the programmer can accept or refuse. The modification is immediately recorded into the database, so the programmer can use the modified program.

The following steps will show how the *tupling function arguments* refactoring was implemented:

- Define the refactor step and its behaviour (rules and test cases)
- Make the decisions between the possible alternative solutions
- Find the preconditions for the refactoring to get a safe refactor tool
- Design the interface toward the user (warnings, error messages)
- Code and test

In the following, we try to analyse this refactoring with examples. After an overview of the refactor tool's structure we represent the carrying out of the refactor step.

1.1.2 Refactoring using a database

Traditionally programs are stored and maintained in a textual format, but still have a structure. During project development, programmers work with a set of files stored in different directories of a file system (or a network of file systems), maintaining them via different file management utilities. Program transformations could be expressed and refactoring could be performed on programs in a more straightforward way if one gave programs a more sophisticated structure and provide a more sophisticated "manager program". An adequate tool for storing and maintaining information is a database manager. The approach presented here, similarly the Clean refactorer [6, 7], is to represent programs in relational databases in order to facilitate refactoring.

AST and database

The syntactic rules of a programming language describe how to represent programs written in that language as (abstract syntax) trees. An **abstract syntax tree (AST)** contains information about the structure of the program code, but many relations are not represented directly in it. The semantical rules of the programming language can be supported by the extension of ASTs with additional information. In the future we will refer to this as **AAST**: the additional information are annotated to the nodes of the syntax tree. For example, to rename a variable, one needs to find every occurrence of it.

An approach that is based merely on ASTs might be inefficient and hard to implement, because finding the occurrences of a variable requires the traversal of the AST. A more helpful approach would be to store direct information about variable occurrences. A possible way of accessing every occurrence of a variable easily is to link these occurrences to a central point, e.g. to the first occurrence in the AST. The resulting data structure is not a tree anymore, but rather a graph, which represents the semantic information too. Our approach is to represent such a graph as a set of relations in a relational database, and use SQL to manipulate it.

We decided to use SQL instead of Mnesia, the embedded database of

the Erlang language, in SQL we have much wider possibilities, the graph connections can be represented more effectively. For example Mnesia, can not handle joining tables together.

The database approach needs more time and effort on database designing and the migration of information from abstract Erlang syntax trees to the database, but the tool can be faster when the refactoring needs less traverse (database queries are more effective than tree part traversals). The second reducing factor is that it tries to avoid reconstruction of the database between two consecutive refactorings by incrementally updating the database so as to keep the stored syntactic and semantic information up-to-date, it maybe worth the effort. At this stage, it is hard to say which approach is better, for more details see [17].

The database representation

In the relational database representation, there are three kind of tables: tables that store the AST, tables that store semantical information and tables which contains collected informations (name, position, node_type) to improve the efficiency of the tool. The syntax-related tables correspond to the "node types" of the abstract syntax of Erlang as introduced in the Erlang parser. Semantical information, such as scope and visibility of functions and variables, is separated in an extensible group of tables. Adding a new feature to the refactoring tool requires the implementation of an additional semantic analysis and the construction of some tables storing the collected semantical information. It is possible to store semantical information of different levels of abstraction in the same database and to support both low-level and high-level transformations.

As an example consider the code in Figure 1.1. This is one clause of a function that computes the greatest common divisor of two numbers, the whole module and its AST is presented in Figure 1.1.2. Each node of the abstract syntax tree is given a unique identifier. These identifiers are written as subscripts in the code and in the figures (the AST of the code in Figure 1.1 is given in Figures 1.5 and 1.6). Every module has it own module identifier too.

The database representation of the AST is illustrated in Table 1.1. The table names "clause", "name", "infix_expr" and "application" refer to the corresponding syntactic categories. Without addressing any further technical details, one can observe that each table relates parent nodes of the corre-

$$\texttt{gcd}_{30}(\texttt{N}_{15},\texttt{M}_{16})\ \texttt{when}\ \texttt{N}_{17}\ \texttt{>=}_{18}\ \texttt{M}_{19}\ \rightarrow\ \texttt{gcd}_{23}(\texttt{N}_{24}\ -_{15}\ \texttt{M}_{26},\ \texttt{M}_{28});$$

Fig. 1.1: Source code of the example function clause.

sponding type with their child nodes.

The price for the separation of tables containing syntactic information from tables containing semantical information is an increased redundancy in the database. For example, the "names" table stores the variable name for each occurrence of the same variable, but it makes the queries more fast and effective. In order to make information retrieval faster, a auxiliary table, "node_type" was introduced. This table binds the identifier of each parent node to the table corresponding to its type.

Semantical information storing in the database approach

The source code and the ASTs of the module used as an example in Figure 1.1.2 is presented below. The tree is split into multiple parts for easier reading. The figures show the result of the Erlang parser, extended with the database identifiers as subscripts.

Greatest Common Divisor

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

gcd(N, N) ->
   N;

gcd(N, M) when N >= M ->
   gcd(N - M, M);

gcd(N, M) ->
   gcd(N, M - N).
```

Fig. 1.2: A module containing and exporting a single function.

Semantical information about Erlang programs are stored in tables such as "var_visib", "fun_visib", "scope", "scope_visib" and "fun_def". The ta-

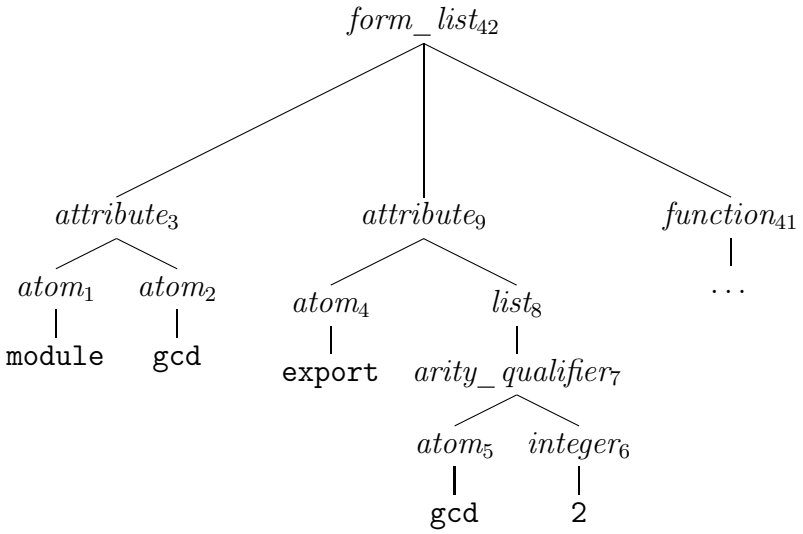


Fig. 1.3: The AST of `gcd` (Part 1)

ble "var_visib" stores visibility information on variables, namely which occurrences of a variable name identify the same variable. This table has two columns: "occurrence" and "first_occurrence". The former is the identifier of a variable occurrence, and the latter is the identifier of the first occurrence of the same variable. The "var_visib" table contains the following pairs regarding the code in Figure 1.1: (15,15), (17,15), (24,15), (16,16), (19,16), (26,16), and (28,16). The table "fun_visib" stores similar information for function calls, and "fun_def" maintains the arity and the defining clauses of functions. The "scope" table contains the scope of the nodes, what is the most inner scope they are in. The "scope_visib" table stores the hierarchy of the scopes. For the definitions see 1.2.1, 1.2.1.

The rename transformations are supported with an another table, "forbidden_names", which describes the names that are not allowed to be used for variables (and for functions). This table contains the reserved words in Erlang, names of the built-in functions, and also user-specified forbidden names.

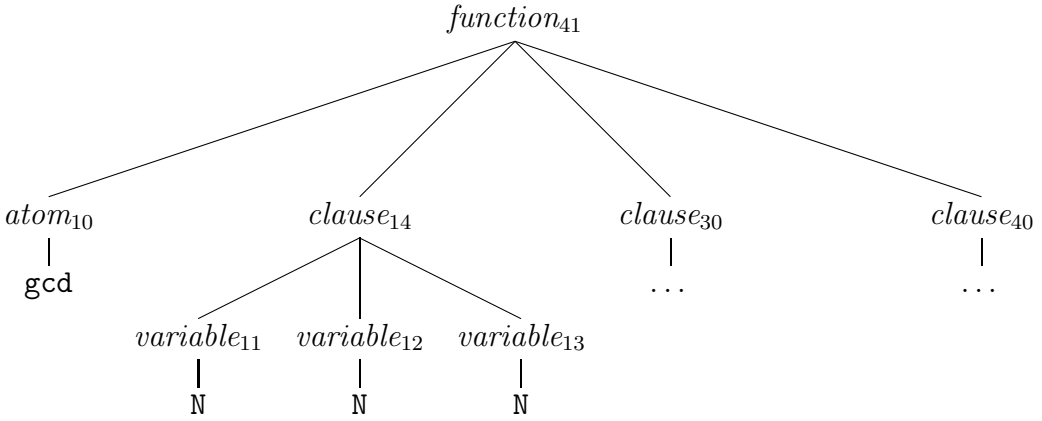


Fig. 1.4: The AST of `gcd` (Part 2)

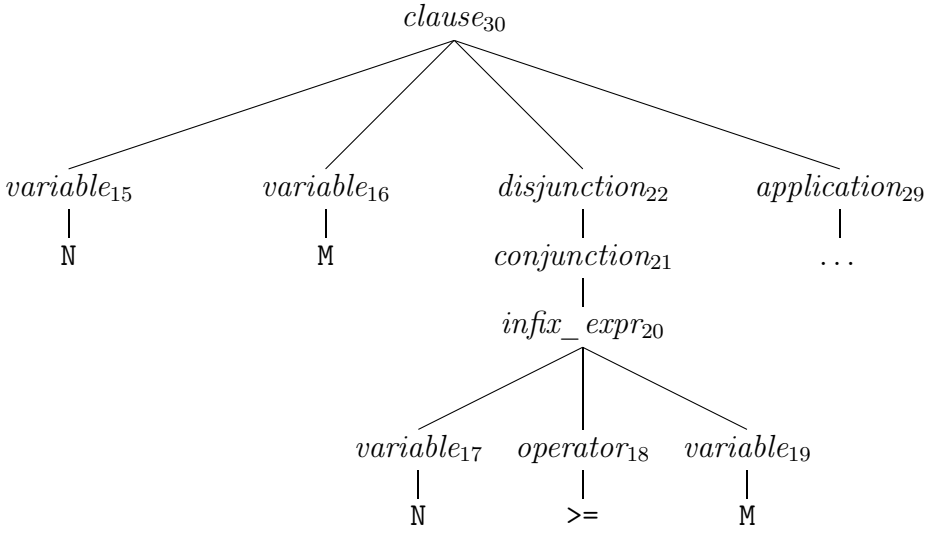


Fig. 1.5: The AST of `gcd` (Part 3)

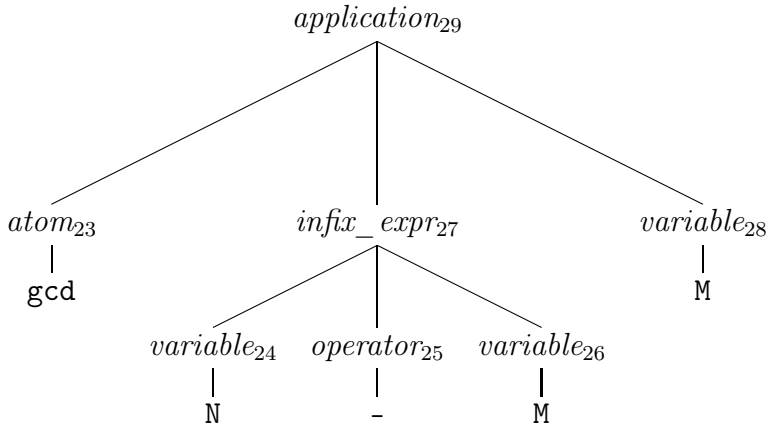


Fig. 1.6: The AST of `gcd` (Part 4)

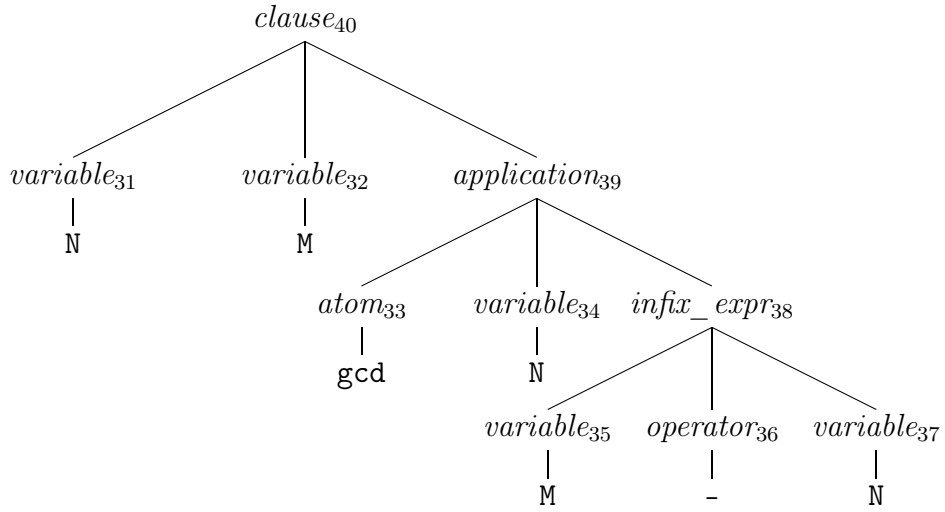


Fig. 1.7: The AST of `gcd` (Part 5)

information in the AST	database equivalent	
	table name	record in that table
1st parameter of clause 30 is node 15	clause	30, 0, 1, 15
The name of variable 15 is `N`	name	15, "`N`"
2nd parameter of clause 30 is node 16	clause	30, 0, 2, 16
Clause 30 has a guard, node 22	clause	30, 1, 1, 22
The left and right operands and the operator of the infix expression 20 are nodes 17, 19 and 18, respectively	infix_expr	20, 17, 18, 19
The body of clause 30 is node 29	clause	30, 2, 1, 29
Application 29 applies node 23	application	29, 0, 23
The content of atom 23 is `gcd`	name	23, "`gcd`"
1st param. of application 29 is node 27	application	29, 1, 27

Tab. 1.1: The representation of the code in Figure 1.1 in the database.

1.1.3 The Refactoring Tool

Design principles for the user interface

In order to provide a convenient environment for program developers, refactoring tool should be merged with other software development tools (for example an editor, a compiler, a debugger, a project manager, etc.). This section highlights an interesting aspect of how the integration of our Erlang refactoring tool with a programmer's editor will be achieved.

The tool will be interactive; it will be started within the programmer's editor. At startup it will analyse the program code being edited, and will create a database from it—or update an existing database with the modules that will have been modified since the previous refactoring session.

The structure of the tool

Help environments:

1. **Emacs** [4]:

   Emacs is an extensible, customisable, self-documenting real-time display editor.

   We use Emacs to display and edit the erlang source code; from an extended Erlang menu the users can build up the connection to the Erlang node and start the steps (initialise database, storing the source into the database and recover it) by choosing the correct menu item (call the suitable Erlang function through Distel)

2. **Distel** [14]:

   Distel extends Emacs Lisp with Erlang-style processes and message passing, and the Erlang distribution protocol.

   With this we can write Emacs Lisp processes and have them communicate with normal Erlang processes in real nodes. This makes it easy to write convenient Emacs user-interfaces to Erlang programs.

3. **Erlang/OTP** [16]:

   A complete development environment for concurrent programming Erlang is a general-purpose concurrent programming language and runtime system. Erlang was released by Ericsson as open-source to ensure its independence from a single vendor and to increase awareness of the language. Distribution of the language together with libraries and a real-time distributed database (Mnesia) is known as the Open Telecom Platform, or OTP.

   Users connect to a running Erlang node, where we can parse and compile the source code and execute the functions of the tool.

4. **MySQL** [15] (or any other SQL server):

   MySQL is an open source relational database management system (RDBMS) that uses a Structured Query Language (SQL) (the most popular language for adding, accessing, and processing data in a database). Because it is open source, anyone can download MySQL and tailor it to their needs in accordance with the general public license.

MySQL is noted mainly for its speed, reliability, and flexibility. Most agree, however, that it works best when managing content and not executing transactions.

We use MySQL to store the source code in a database. The database is based on the syntax tree of the source code.

The modules of the (refactoring) tool The refactoring tool is a group of Erlang modules to store and recover the source code into and from the database, and modules containing each refactoring. This thesis does not contain the refactor modules, we just described them to give an overview from the whole refactorer. Some files (for example distel.el) are modified in the environments to generate the refactor menu in Emacs, and message handling (for example throwing warnings to the user) in Emacs through Distel.

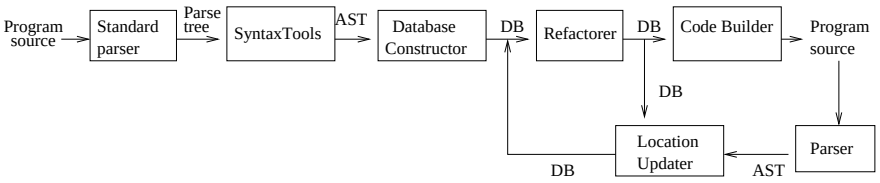


Fig. 1.8: The Implementation Architecture

Figure 1.8 summaries the implementation architecture of this approach.

The refactoring updates the database (which represents the AST and the semantic information), but the position information might no longer reflect the actual positions in the program source. In order to keep the position information up-to-date, we build up the updated syntax tree from the database and use the pretty-printer to refresh the code, then the position information is updated by a simultaneous traversal of the syntax tree represented in the database, and the AST is generated by parsing the refreshed code.

Modified existing files

- **ELisp files**

  1. **Distel.el**: We modified this file to add our tool to the refactor menu.

2. **Refaktor.el**: We made this file to generate the refactor menu in Emacs, and this file is our interface between the Emacs and the refactoring tool (Erlang function callings, giving warnings and message handling).

- **Erlang files** The modifications in these files are the added column numbers to the position information.

  1. **epp_dodger1.erl**: The Erlang preprocessor substitutes the macro definitions, so we are using `epp_dodger` instead of `epp`.
  2. **epp_recomment1.erl**
  3. **erl_scan1.erl**

The new erlang modules to the tool The structure of the modules can be seen in the 1.9 figure. The modules of the refactorer can be sorted into the following groups:

Interface module: This group contains only the `d_client` module. This module's functions are called from `distel.el` to start executing the refactor steps, the storing and rebuilding of the code to/from the database or initialising the database. Each function of the module call functions in the other modules which solve the tasks.

Connection modules: The tool has two possible connection type toward the database (native mysql and odbc). The `refactor_db` module handles the calls towards the database with a genserver behaviour. The connection type can be choosed. The `refac_superv` module is the supervisor module of the connection server.

Database modules: This group manages the database operations: initialisation, putting source code with semantic information to the database and restore the source code. The database modules: `db_init`, `into_db`, `out_from_db`.

AST manipulation in the database: This group represents the AST operations directly in the database. The modules: `create_nodes`, `delete_nodes`, `erl_syntax_db`

Common refactor modules: This group contains modules which are used in several refactorings. The `refactor` module contains the frequently used SQL queries to collect datas from the database (for example get the identifier of the function, if its name and arity is known). The `refac_checks` module contains the common functions for precondition checkings. The `refac_common` module contains the common functions during the execution of the refactorings.

Refactor modules: Each module in this group is responsible for one refactoring. The refactor modules:

- `refac_ren_var` Rename variable (1.2.2),
- `refac_ren_fun` Rename function (1.2.3),
- `refac_reorder_funpar` Reorder function arguments (1.2.5),
- `refac_tuple_funpar` Tupling function arguments (1.2.4),
- `refac_var_elim` Eliminate variable (1.2.6).

1.2 Syntactical and Semantical Analysis

1.2.1 Semantics of functions and variables

Scope and visibility of variables

Most of the refactorings are concerned about variables in some way. Erlang programs mainly consist of expressions, and expressions can use and define variables almost anywhere. The meaning of variables in an expression depends on its context, so the relation of the variables and the context of the expression must be maintained during a refactoring, otherwise the meaning of the modified program text would differ from the original.

The relation of variables and expressions is defined in terms of visibility rules. An expression can only use visible variables, and the visibility of variables begins with their creation, often in an expression. Every refactoring that works with variables or expressions must be able to determine the exact visibility region of every variable.

The exact semantical rules defining variable visibility are given in [13], using input and output contexts for every language construct, which is hard to follow and not really helpful in defining the conditions of a refactoring.

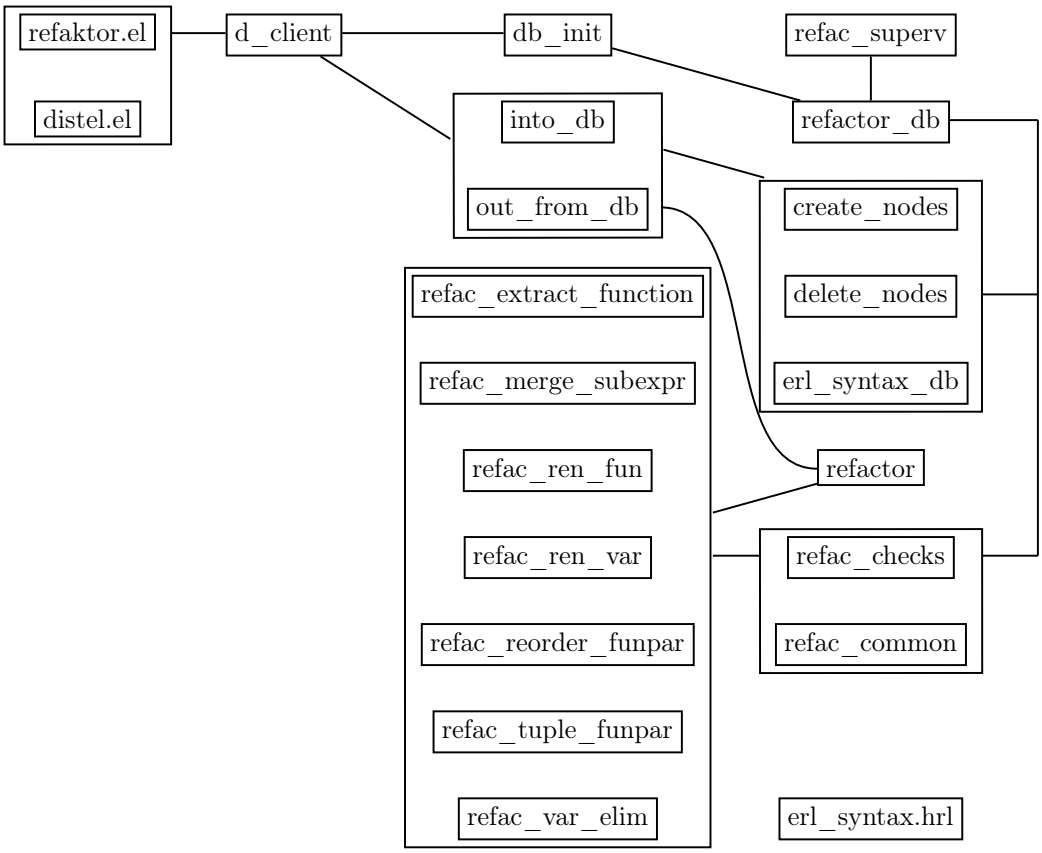


Fig. 1.9: The structure of the modules

We have created a more compact definition which is suitable for verifying refactoring conditions.

In Erlang, variables have a name[1] and a value bound to them (which never changes during the life of the variable). For our purposes, we define the *scope* of a variable as a non-contiguous region of the program text where the value of the variable is bound to its name, and a variable is *visible* in a region where its name can be used to refer to the variable.

Rules for variable scoping The scope of a variable is always limited to a function clause, a list comprehension or an element of a tuple or list. There are no global variables: every variable is *local* to some *scope delimiter*, namely to a function clause (of either a declared function or an explicit fun-expression), or to a list comprehension, or to an element of a tuple or list. Scope delimiters can be nested:

- the body of a function clause (of a declared function or of an explicit fun-expression) might contain a clause of an explicit fun-expression;
- the body of a function clause (of a declared function or of an explicit fun-expression) might contain a list comprehension;
- the body of a function clause (of a declared function or of an explicit fun-expression) might contain an element of a tuple or list
- a list comprehension might contain a clause of an explicit fun-expression;
- a list comprehension might contain a list comprehension;
- a list comprehension might contain an element of a tuple or list (typically, a generator of a list comprehension contains a list);
- an element of a tuple or list might contain a clause of an explicit fun-expression;
- an element of a tuple or list might contain a list comprehension;
- an element of a tuple or list might contain an element of a tuple or list.

[1] Variable names always begin with a capital letter or an underscore, the latter meaning an ignored value.

The outermost scope delimiters are the clauses of declared functions: they are never nested in other scope delimiters, and every other scope delimiter is nested (directly or indirectly) in a clause of a declared function.

A variable occurrence is *direct* in a scope delimiter, if it is an occurrence in that scope delimiter, but not an occurrence in some enclosed scope delimiter. A variable occurs directly in a scope delimiter, if it has at least one direct occurrence in that scope delimiter.

The scope of a variable is a non-contiguous region of the outermost scope delimiter where the variable directly occurs. (There is exactly one such outermost scope delimiter. Given a scope delimiter with no direct occurrences of a certain variable name, but with two or more independent nested scope delimiters with direct occurrences of that variable name, the variable name refers to two or more independent variables.) The scope of a variable is given as a set of expressions.

Variables are created by the pattern matching mechanism. Pattern matching is used in

- heads of function clauses (of declared functions or of fun-expressions),
- pattern match expressions,
- `case`, `receive` and `try` constructs and
- generators of list comprehension expressions.

The same variable might be defined by multiple pattern matchings, for example, in different branches of a branching expression. Therefore the scope of such a variable begins at multiple places of the program text: this is why the scope is a non-contiguous region of the program text.

```
h(X) -> if
            X >  0 -> Z = X,  Y = 1;
            X <  0 -> Z = -X, Y - -1;
            X == 0 -> Z = 0, Y = 0
        end,
        {Y,Z}.
```

The expressions "`Z = X`", "`Z = -X`" and "`Z = 0`" are not part of the (non-contiguous) scope of Y.

Expression B is to the right of expression A in expression E if A and B are directly in the same scope delimiter, and there exist expressions C,

D_1, D_2, ..., D_n such that A is a subexpression of C, and the expression "$C, D_1, D_2, \ldots, D_n, B$" is a subexpression of E. Note that "$C, D_1, D_2, \ldots, D_n, B$" can be a sequence of expressions (in a Block), a sequence of patterns (in a composite pattern), a sequence of Qualifiers in a list comprehension or a sequence of guards.

First we define *the scope of a pattern matching occurrence of a variable*. If a variable occurs in a pattern matching, then the scope of this pattern matching contains every expression to the right of the variable occurrence. Furthermore, if the pattern matching is

- in the head of a function clause, then the guards and the body of the clause is also part of the scope;
- in a list comprehension expression (the pattern of a generator), then the qualifiers to the right of the generator and the body of the list comprehension are also part of the scope;
- in a pattern match expression, then the right-hand side of that expression, and all expressions to the right of the pattern match expression are also part of the scope;
- in a branch of a "case", "receive" or "try" expression, then that branch of the expression and all the expressions to the right of the concerned ("case", "receive" or "try") expression are also part of the scope.

The *scope of a variable* is defined in the following way. Take the (existent and unique) outermost scope delimiter in which the variable directly occurs. The scope of the variable is the union of the scopes of the pattern matching occurrences of that variable that directly occur in that scope delimiter.

Rules for variable visibility The rules describing the locality of variables are the following.

- Variable names occurring in the formal argument list of a function clause denote variables local to that function clause. A variable name occurring directly in a function clause but not occurring in the formal argument list of the function clause denotes a variable local to that function clause if and only if the function clause is not enclosed in the scope of a variable with that same name. (Note that the function clause is enclosed in the scope of a variable if and only if the function clause is a

clause of an explicit fun-expression and either (1) the variable is a local variable of the enclosing scope delimiter, and the enclosed explicit fun-expression is within its—non-contiguous—scope, or (2) the enclosing scope delimiter itself is also enclosed in the scope of the variable.)

- Variable names occurring in the Pattern of a generator of a list comprehension denote variables local to the list comprehension. A variable name occurring directly in a list comprehension, but not in the Pattern of a generator of the list comprehension denotes a variable local to the list comprehension if and only if the list comprehension is not enclosed in the scope of a variable with that name.

- A variable name directly occurring in an element of a tuple or list denotes a variable local to that element of tuple or list if and only if that tuple or list is not enclosed in the scope of a variable with that name.

These rules reveal that occurrences of variable names in formal argument lists and patterns of generators of list comprehensions introduce variables that might *shadow variables* of the enclosing scope delimiter.

A *variable is visible* within its scope where none of the following limitations apply:

- A nested scope delimiter may introduce a new variable with the same name. The scope of such variables are excluded from the visibility region of the concerned variable (shawoding).

- Variables created in a `catch` or `try` expression are unsafe to be used outside that expression, so they are visible only inside the innermost enclosing `catch` expression.

- Variables that are created inside a branch of a branching expression (those are: `if`, `case` and `receive`), but are not bound a value in every branch, are unsafe to be used outside that expression, so they are not visible outside the expression.

- Variables created in the timeout expression of a `receive` construct's `after` branch, but are not bound in the body of that branch, are not considered to be bound in that branch at all.[2]

[2] Note that this behaviour does not comply with [13], and may be a compiler bug.

Binding variables The different occurrences of a variable are classified by [13] as "binding occurrences" and "non-binding occurrences". The binding occurrences are those occurrences where the variable is bound to its (not unique, but final) value.

```
f(X) -> if
           X >  0 -> Y = 1;
           X <  0 -> Y - -1;
           X == 0 -> Y = 0
        end,
        {Y,X}.
```

The first three occurrences of Y are binding occurrences, binding three different, but final values to variable Y. The last occurrence is non-binding.

Note that the same syntactic form (pattern match) can refer to a binding or a non-binding occurrence of a variable. The second occurrence of X in `d/1` is clearly a non-binding occurrence.

```
d(X) -> receive
          {X,Y} -> Y;
          _     -> 0
        end.
```

It is not possible to decide which are the binding occurrences of a variable: this depends on the specific Erlang implementation:

```
g() -> (X=1) + (X=1).
```

One of the occurrences of X is a binding occurrence, and the other occurrence is a non-binding occurrence; but it is unspecified (left to the Erlang implementation) which is the binding occurrence.

We introduce the concept of *possible binding occurrence*: in function `g/1`, both occurrences of X are possible binding occurrences. If a variable occurs in a pattern matching, then this occurrence is a possible binding occurrence, if the scope of this occurrence is not part of the scope of another pattern matching occurrence of the same variable.

Refactoring functions

Function visibility is much simpler than variable visibility: a named function is either exported, in this case it is visible from every module, or not

exported, and then it is visible only in the defining module. There are no function name hiding, embedded functions or any other kinds of complication. Here the real problem comes from the already mentioned dynamic language constructs. While a variable can only be used statically, a function name can be constructed dynamically, and functions can be called using built-in functions.

In the following, a compact description of functions in Erlang is given, then the relation of functions and refactoring is analysed.

Functions in Erlang Functions are always defined in modules, and they are identified by three components: the module name, the function name and the arity of the function (i.e. the number of formal arguments). The module and function names are so-called *atoms*, which are string-like data terms usually used as labels throughout the code. Two or more functions with the same name but different arities are permitted; we will use the name *overloaded functions* in such cases.

Static function calls use the name of the function and supply the list of arguments to be passed. It is important to note that the function name can be any expression that results in an atom; usually the name is given explicitly, but it is possible to use variables or even other function calls to compute the name of the function. When there is no function with the resulting name and arity (the latter is computed from the length of the parameter list), a runtime error is signalled. A compile time check is only performed when the function call uses an explicit name; in this case a call to a non-existing function results in a compilation error.

Functions are exported using the `export` module attribute. An export list contains function names together with arities (the syntax is `function/arity`), and defines the interface of the module. Outside the defining module an exported function can be accessed by supplying the module name with the function name (the syntax is `module:function(args)`). The module name is an expression again, that must result in an atom. Module names passed as parameters are used throughout the OTP to access callback modules. Although OTP code contains a lot of module names supplied as variables, this is not usual in normal application code.

Another language feature is the import list, which makes it possible to omit the module name from an external function call. The `import` module attribute is interpreted at compile time (just like `export`). It contains a

module name and a list of function names with arities.

There is one more possibility to call a function, using one of the built-in functions that result in a function call. These BIFs usually take three parameters: the module name, the function name and the arguments as a list. The same problem arises with the function and module names as with the function call expression, and there is a new one with the argument list: even when the module and function names are explicitly given, the argument list can be created dynamically, and the length of the list determines the arity of the function to be called. Built-in functions like that are `apply`, `spawn` (and its many variants), and `erlang:hibernate`.

Finally, there is one more construct that refers to a function: its name is *implicit function expression*. It requires an explicit function name (or module and function name) and an arity, so there is no problem with its static analysis.

Scope of function-related refactorings Many of the function-related refactorings rely on finding every place of call for a given function. Obviously, it is impossible to statically determine the place of every dynamic call, but there are semi-dynamically constructed calls that are possible to find. To establish the exact scope of the refactoring tool, here we categorise the function-related constructs based on the level of support for them.

Fully supported constructs. Static function calls with explicitly given function (or module and function) names and arguments, implicit function expressions and members of import and export lists are fully supported. Considering only these constructs, it is possible to determine the exact list of references to any given function.

Constructs with limited support. Function calls using the built-in functions mentioned above, when used with explicit module and function names, can be found by static analysis, and the only missing information to identify the function called by them is the function arity. This can be handled depending on the situation:

- When the referred function has no overloaded variants, the reference is unambiguous, but there can be problems with the parameter list which is constructed at runtime.

- When there are functions with the same name but different arities, the reference cannot be resolved by static analysis. This situation makes some refactorings (e.g. rename function) impossible, but the situation can be detected and the refactoring can be denied. In other cases, there are no new problems compared to the previous case (e.g. in reorder function arguments).
- There is one special case, when the argument list is given as a static list skeleton. This case is essentially the same as a normal function call, it just needs a bit more work to recognise.

Unsupported cases with a possible solution. There may be occurrences of function names in the program code that can not be classified as a reference to the function (remember that a function name is an atom which can be used in any role in the code). A type checker may help to get more information on which occurrence of an atom is used as a function name and which is not; this should be the subject of further studies. Right now we can only give a warning message on a possible occurrence of a function name in this case, because transformations in this case possibly alter the semantics of the program.

Inherently dynamic constructs. When a module or function name is created at runtime, a static analyser cannot get useful information on which function will be called at that point. Examples for this are module and function names read from the user, a file, or a database, and names that are passed as parameters or messages (or constructed from such data). These constructs are so hopeless to analyse that even a warning message is impractical for them (if the code contains one such construct, every function-related transformation would give a warning that has no usable information).

1.2.2 Analysis of the rename variable refactoring

Overview

This is one of the most simple refactorings. Its goal is obvious from the name, and it is one of the refactorings that can be supported by a tool without restrictions. The only semantical information that is necessary for it is the scope and visibility of the variables, which have been defined in 1.2.1.

Parameters

- The position of one of the instances of the variable, a pair of positive integers (line and column).
- The new name for the variable as a string; it must obey the syntactic restrictions for variables (i.e. begins with a capital letter or an underscore, contains only alphanumeric characters or underscores.)

Conditions of applicability

1. The new variable name is not visible at any occurrence of the variable.

Rules of transformation

1. Find every occurrence of the variable (i.e. the variables with the same name in the visibility range of the variable).
2. Replace every occurrence with the new name.

Discussion

The following are expected from a tool that changes the name of a variable:

- modify every occurrence of the variable,
- do not modify anything else in the source (other variables with the same name and other occurrences of the name are left intact) and
- do not allow to use a new name that changes the way the program works (especially, do not allow introduction of compilation errors).

The scoping rules exactly define which are the occurrences of a variable: every occurrence of the variable's name where the variable is visible. The definition ensures that no two variables with the same name are visible at the same place, so the first two requirements are fulfilled by this approach.

The third requirement can be violated by breaking the rule of disjoint visibilities, that is, re-using an existing variable name inside its visibility area. The refactoring is forbidden in this case by the condition.

When a renaming violates the condition, a compensation step usually can help to solve the situation, as illustrated in Figure 1.10. It is even feasible to offer such a compensation automatically.

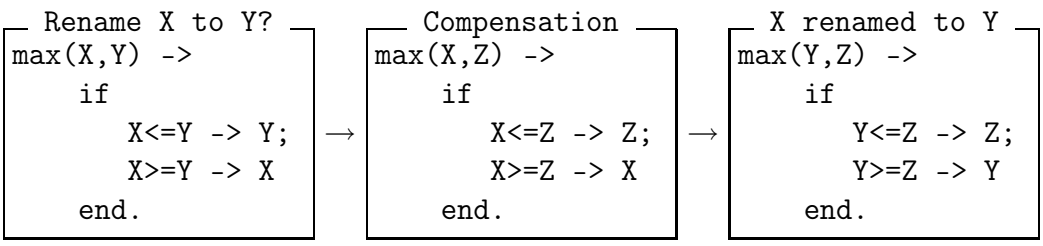


Fig. 1.10: The use of a compensation step when renaming variable `X` to `Y`

```
Count the occurrences of value V in list L
multiplicity(V,L) ->
       length(lists:filter( fun (X) -> X==V end, L )).
```

Fig. 1.11: Example of variable shadowing: `X` can be renamed to `L` without semantical change, but it is misleading.

There are situations when renaming a variable does not change the way the program works, but it does influence the readability of the code—possibly in a negative way. Renaming a variable can introduce shadowing, as illustrated in the example in Figure 1.11.

If `L`, the second argument of function `multiplicity`, is renamed to `X`, or `X`, the argument of the local function expression is renamed to `L`, then the argument of the local function shadows the second argument of the enclosing function within the body of the local function. Although the menaing of the function remains unchanged, the readability of the resulting code is much worse than the original. The condition actually forbids this renaming, but it is possible to accept these cases after a warning message.

1.2.3 Analysis of the rename function refactoring

Overview

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

Rename function

```
doit(P) ->
   P ! {msg, start}.

start(Lst) ->
   lists:forall(
      fun doit/1, Lst).
```

→

Result

```
send_start(P) ->
    P ! {msg, start}.

start(Lst) ->
   lists:forall(
      fun send_start/1, Lst).
```

Fig. 1.12: A simple function renaming.

Changing a function's name seems trivial, like our motivating example in Figure 1.12. However, as we have seen in Section 1.2.1, there are many constructs in Erlang that deal with functions at runtime and have a dynamic nature, and these make life hard for a static analyser. With respect to these dynamic constructs, we can only outline the possibilities of our automatic tool, and show "the line that's drawn between the good and the bad", define which constructs are supported and which are not. This is done using the terms of Section 1.2.1.

Parameters

- A position in the Erlang file that contains the function to be renamed, which points to anywhere inside the function.
- The new name for the function, which is an arbitrary string (altough developers are encouraged to pick simple atoms as function names).

Conditions of applicability

1. There must be no functions with the given name and the same arity as the function to be renamed in the module.
2. When there are multiple overloaded versions of the function, there must be no call to the function that creates the list of arguments dynamically.
3. When there are existing functions with the new name but arities different from the old function, and there are calls to these functions with dynamically created argument lists, a warning message must be given about possibly introducing an unwanted function call.

4. If the name of the function occurs anywhere as an atom unqualified as a function call, a warning message must be given about the possibility of leaving an occurrence of the name unchanged.

Rules of transformation

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

Discussion

In this refactoring, problems arise because of dynamic function references. These can (or cannot) be handled depending on the situation:

- When the function to be renamed has no variants with the same name but different arities, function calls using `apply` and `spawn` and static function names are essentially the same as static function calls, changing the name of the function will not influence these constructs.
- When there are functions with the same name but different arities, renaming one of these functions will inevitably change the semantics: calls to functions with different names cannot be handled by one `apply` call. Our proposed solution is to deny this kind of renaming, and provide a variation of the refactoring that renames all of the functions with the same name. This often meets a good programming style where functions with the same name do the same thing, so they should be renamed together.

Function names are atoms in Erlang, so the name of a function can be an arbitrary string provided in atom syntax (ie. put in single quotes) in the source code. However, it is hard to use functions with strange names, so the tool should give a warning if a function name is given that cannot be expressed using normal atom syntax.

1.2.4 Analysis of the tuple function arguments refactoring

In this transformation some consecutive arguments of a function are extracted into a tuple. This transformation addresses the formal parameter list of all the alternatives in the function definition, as well as the actual parameter list in each perceptible (viz. by static analysis) call of the function. The transformation can affect more than one module if the function is exported. If the length of the tuple is greater than one, the arity of the function will also change too.

The example in Figure 1.13 illustrates the operation of the transformation on a function with a single alternative. Both the definition of `gcd/2` (both branches) and its occurrence in the export list are changed.

Parameters

The position of the first formal argument to be included into the tuple (Position) and the number of formal arguments to be included into the tuple (Number). The Position is given as a pair of positive integers (line and column number), and the Number is a positive integer.

Conditions of applicability of the transformation

After the general rules, we create the preconditions of the current refactoring, using the following specification:

1. The position given must be a formal argument of a function definition, say, argument number n.
2. The function must be a declared function, not a fun-expression.
3. The conditions for being able to find the call sites of a function can be found in Section 1.2.1. This refactoring requires that call sites correspond either to "Fully supported constructs" or to the third case of "Constructs with limited support".

```
-module(gcd).

-export([gcd/2]).

gcd(A,B) when (A > 0)
         and (B > 0) ->
   gcd_impl({A,B});
gcd(A,B) ->
   throw({values_given_to_
    gcd_must_be_positive}).

gcd_impl({A,A}) -> A;
gcd_impl({A,B})
          when A > B ->
   gcd_impl({A-B,B});
gcd_impl({A,B}) ->
   gcd_impl({A,B-A}).
```

→

Transformed

```
-module(gcd).

-export([gcd/1]).

gcd({A,B}) when (A > 0)
           and (B > 0) ->
   gcd_impl({A,B});
gcd({A,B}) ->
 throw({values_given_to_
  gcd_must_be_positive}).

gcd_impl({A,A}) -> A;
gcd_impl({A,B})
          when A > B ->
   gcd_impl({A-B,B});
gcd_impl({A,B}) ->
   gcd_impl({A,B-A}).
```

Fig. 1.13: Tupling the two arguments of function `gcd`.

4. The Number given must not be too large: Number+n should be at most the arity of the function.

5. The new arity of the function is less than the original arity -1, if the tuple length is greater than one. The resulting function should not conflict with another if its name is overloaded.

   - If the function is not exported, it should not conflict with other functions defined in the same module or imported from other modules.
   - If the function is exported, then apart from the requirement above, for all modules where it is imported, it should not conflict with functions defined in those modules or imported by those modules.

Rules of the transformation

1. Collect all alternatives of the function definition.
2. Decide which formal parameter is referred by position.
3. For every alternative, change the formal parameter list: extract the formal arguments from argument number n to argument number n+Number into a tuple pattern.

   If the function has implicit function calls it needs special attention to create the new code parts.
4. If the function is exported from the module, then the export list has to be modified: the arity of the function decrements by (Number-1).
5. Find all call sites of the function. If the function is exported, every module must be scanned.
6. If the function is exported and another module imports it, then the arity must be adjusted in the corresponding import list of that module.
7. For every call site, modify the actual parameter list by extracting actual arguments from argument number n to argument number n+Number-1 into a tuple. The way to do this depends on the nature of the function call expression.

   Fully supported constructs: the argument list is explicit in the AST.

   Constructs with limited support: the argument list is given as a static list skeleton; this list expression should be reorganized into a shorter list.

Possible compensations to satisfy the preconditions

In the case of overloading conflicts the "rename function" refactoring can be used as a compensation, first rename the problematic function, and after it can be done the tupling.

Conclusion

- This refactoring is used to contract multiple formal arguments of a function into a tuple; it cannot be used to change the type (inner structure) of one of the arguments. If a formal argument is a composite pattern, a tuple, for example, some elements of that tuple cannot be contracted into a nested tuple by this transformation. The first source of problems is those function definitions with multiple alternatives, where the formal parameter lists of the alternatives contain the same formal argument with more details or less details on its structure. The second, maybe even more subtle source of problems, is the different structure of actual arguments at the call sites of the function, in this case type analysis can help. There are similar problems with the inverse transformation of "tuple function arguments".

- The selection of the first formal argument to be included in the tuple is allowed in any of the alternatives of the function definition.

- Position selection in "fun-expression" is not supported by this transformation, because it can not be identify all call sites of the function, since all the actual parameter lists should be altered.

1.2.5 Analysis of the reorder function arguments refactoring

Overview

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

The idea is illustrated by the following simple example (see Figure 1.14), where a function's three arguments are reversed. To maintain the meaning of the program, every call of the function must be modified: the order of expressions that provide actual parameters must be reversed too.

This refactoring can be carried out in almoest every case without problems, only dynamic function calls put limits to its applicability. Within the bounds of static analysis, every parameter reordering can be compensated at the place of function calls. Places of function calls are determined according to Section 1.2.1.

Reverse argument order

```
sum(A,B,C) ->
    A+B+C.

caller() ->
    sum(1, 2, 3).
```

→

Result of reversing

```
sum(C,B,A) ->
    A+B+C.

caller() ->
    sum(3, 2, 1).
```

Fig. 1.14: Simple argument reordering.

Parameters

- A position in an Erlang module that provides the function that is to be modified; the position can be in any caluse of the function.
- The new order of the function parameters, a list of integer numbers that specify which parameter goes where. For example, the list "3,2,1" means that the third parameter will be the first, the first parameter will be the last and the second parameter remains in the same place.

Conditions of applicability

1. When a function call uses expression with side effects to determine the actual parameters, the transformation may only be carried out after a warning that the order of side effects most probably will change, which may change the way the program works.
2. When atoms that are same as the function name are used in the code, a warning message should be given about that with respect to unqualified usage of the function name as advised in Section 1.2.1.

Rules of transformation

1. Change the order of patterns in every clause's parameter list in the function according to the given new order.
2. For every static call of the function, change the order of the expressions that provide the actual parameters to the call according to the given order.

3. Every implicit function expression is converted to the corresponding explicit expression which contains a static call to the function; this function call is then updated as described in the previous case.

4. For every call of the function that provides the arguments as a list, insert a compensating function expression that changes the order of the elements in the list according to the given new order (see below).

Discussion

It is easy to update a static function call according to this refactoring, but what is the case with the more dynamic constructs? When a built-in function is used for the function call and parameters are passed in a dynamically created list, it seems hard to change the order of the list members. Fortunately, we can solve this problem using functional features of Erlang. This transformation can be described by explicit functions which operate on lists: pattern matching is used to select every element of the list, and the function simply constructs a new list with the selected elements in the desired order (see Figure 1.15. for an example). The problem of different functions with different arities is easily solved with this approach: the generated function expression make no changes to lists with lengths that differ from the number of arguments in the transformed function, so dynamic calls to functions with other arities are not affected.

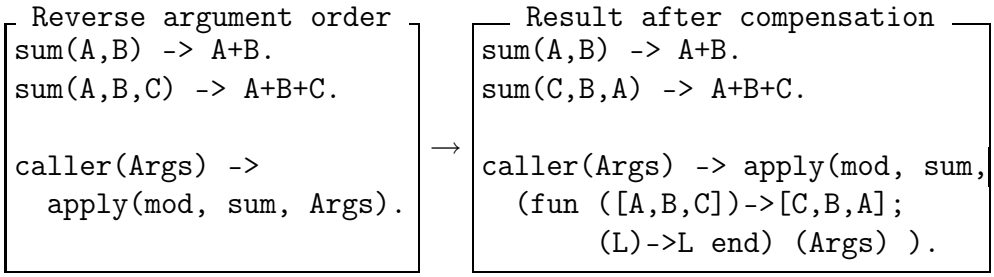


Fig. 1.15: Reorder arguments of a function that is called semi-dynamically.

1.2.6 Analysis of the eliminate variable refactoring

Overview

The next refactoring is the eliminate variable transformation. In this refactoring all instances of a variable are replaced with its bound value in that region where the variable is visible. The variable can be left out where its value is not used.

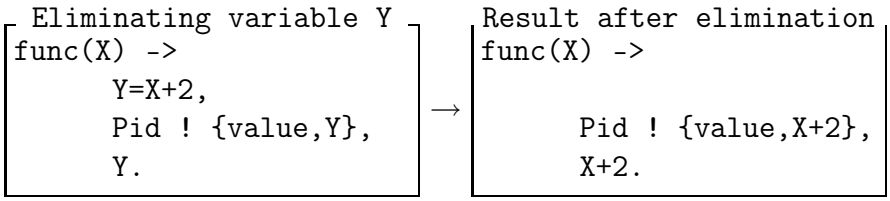


Fig. 1.16: Replacing all instances of variable Y with its value.

In Figure 1.16, In the place where Y is bound, Y variable is not used, therefore it can be left out in the first line. The further occurrences of variable V are changed to the original expression, which was stored in Y before the elimination.

There are situations when the variable cannot not be eliminated, for example, when the value stored in the variable is not known, or when the variable is created in multiple branches of a case, receive or if construct.

The steps of refactoring can cause compilation errors, when the variables we want to elimiate are in the apply or spawn BIFs (Built In Functions).

Parameters

- A position in the Erlang file that contains the function to be modified, which marks an occurrence of the variable to be eliminated.

Conditions of applicability

The variable can only be eliminated, if:

- It has exactly one binding occurrence on the left hand side of a pattern matching expression, and not a part of a compound pattern.
- The expression bound to the variable has no side effects.

- Every variable of the expression is visible (that is, not shadowed) at every occurrence of the variable to be eliminated. (see also the definition of visibility in 1.2.1.)

Rules of transformation

- Remove every occurrence of the variable that is not used. A variable is used if it is part of an expression whose result is not discarded.
- Substitute every occurrence of the variable with the expression bound to it at its bindig occurrence, with parentheses around the expression.
- Remove the bound expression from the place of definition together with the equal sign of the pattern matching expression.

Discussion

A frequently occurring problem during elimination is when a new expression requires brackets because of the precedency of its operations otherwise its meaning would change. This does not always cause a syntactic error, but can change the execution of the program.

The expression often becomes too complex after eliminating, which makes the code hardly legible.

Variable in a list

```
f(X) ->
    [A,B] = foo(),
    A.
```

Fig. 1.17: Eliminating the variable in a compound pattern is not realizable.

If the binding contains a list or tuple, and is bound with the resultion of a function (where the function gives a list or tuple). In this case the variable in the list cannot be eliminated unambiguosly, because the resultion will be known only after the evaluation of the function (see Figure 1.14).

In if, case primitives and in the branches of receive the first occurrence of the variable cannot be always told with certainty. If the binding exists only in certain branches, or the required branch is not executed, the varieble cannot be defined.

In many cases the above problems can be solved by compensational steps (by applying other refactorings).

2. DESIGN OF THE REFACTOR ALGORITHMS

2.1 Algorithm of the reorder function arguments and the eliminate variable refactorings

The following flowchart diagrams show the structure of the two module. The applied signs are:

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

Fig. 2.1: The structure of the refac_reorder_funpar module (Part 1)

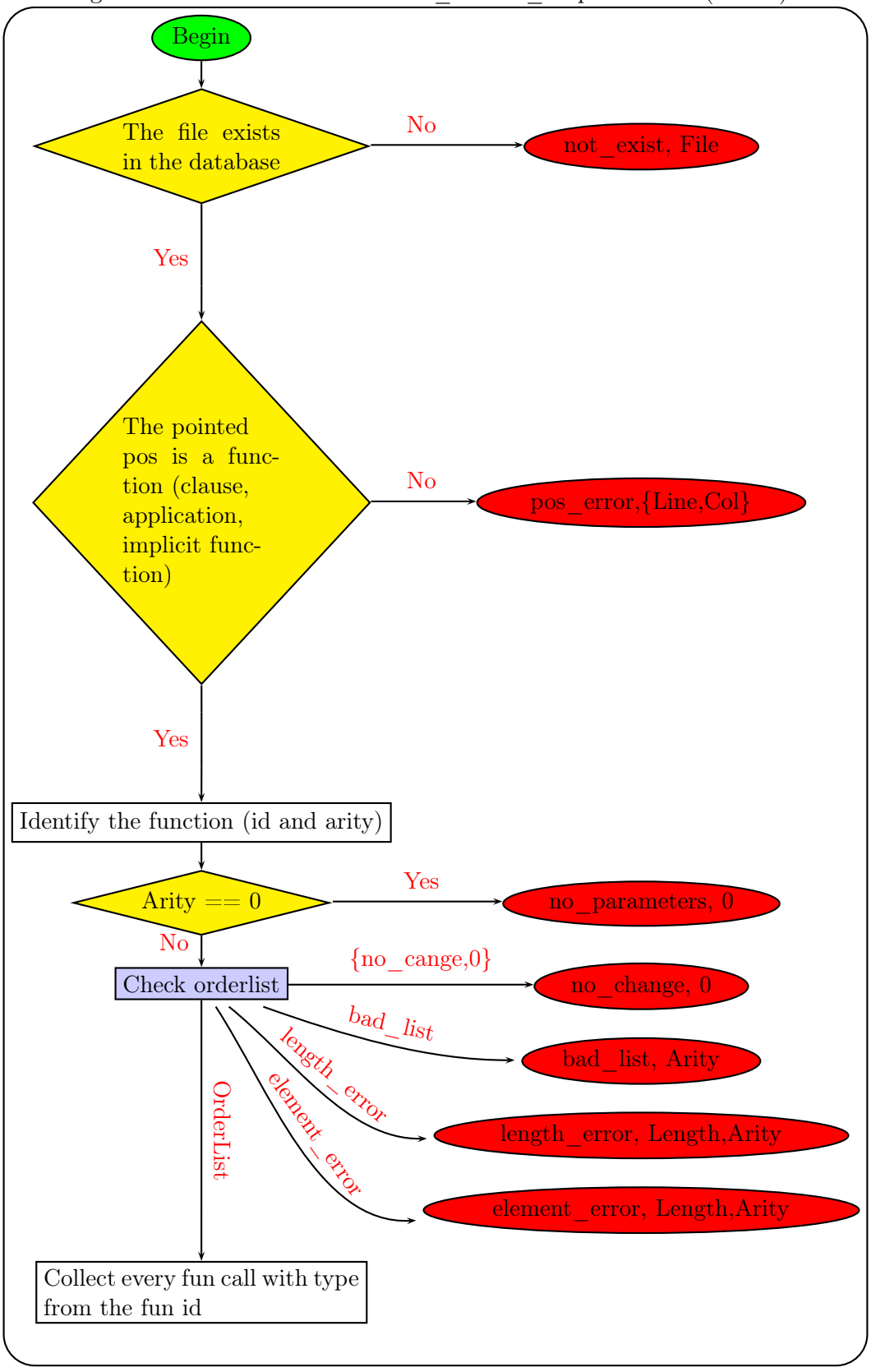


Fig. 2.2: The structure of the refac_reorder_funpar module (Part 2)

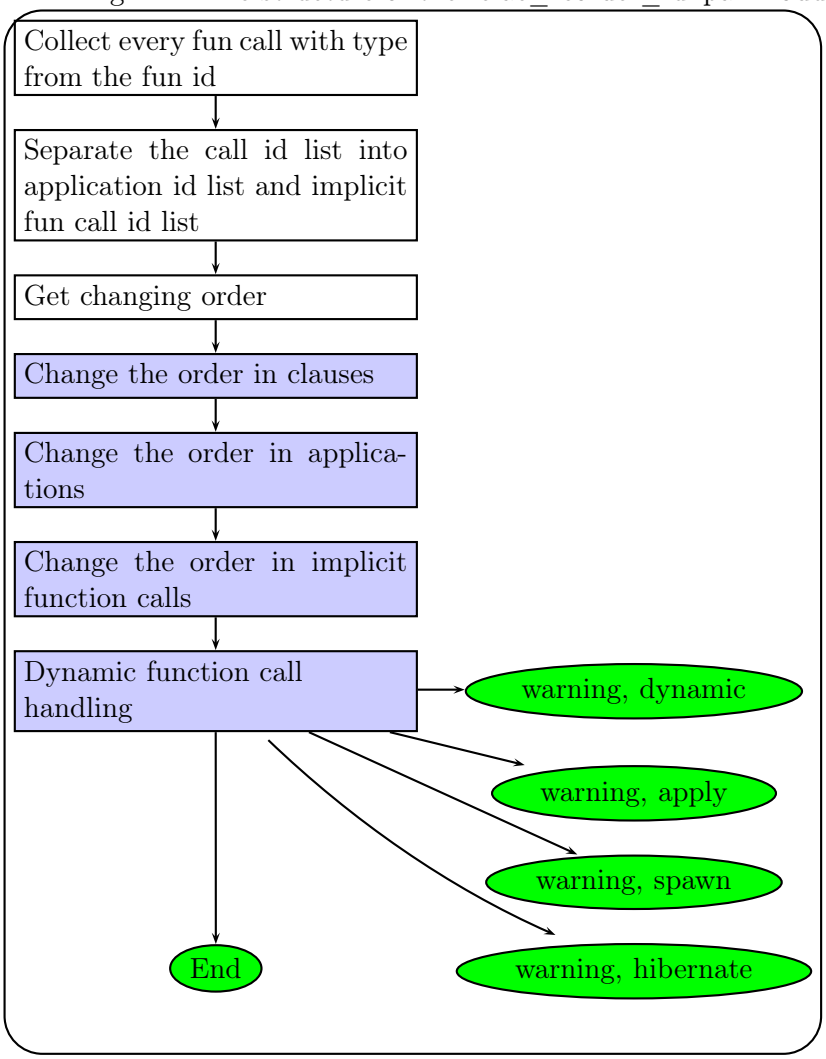


Fig. 2.3: Check orderlist

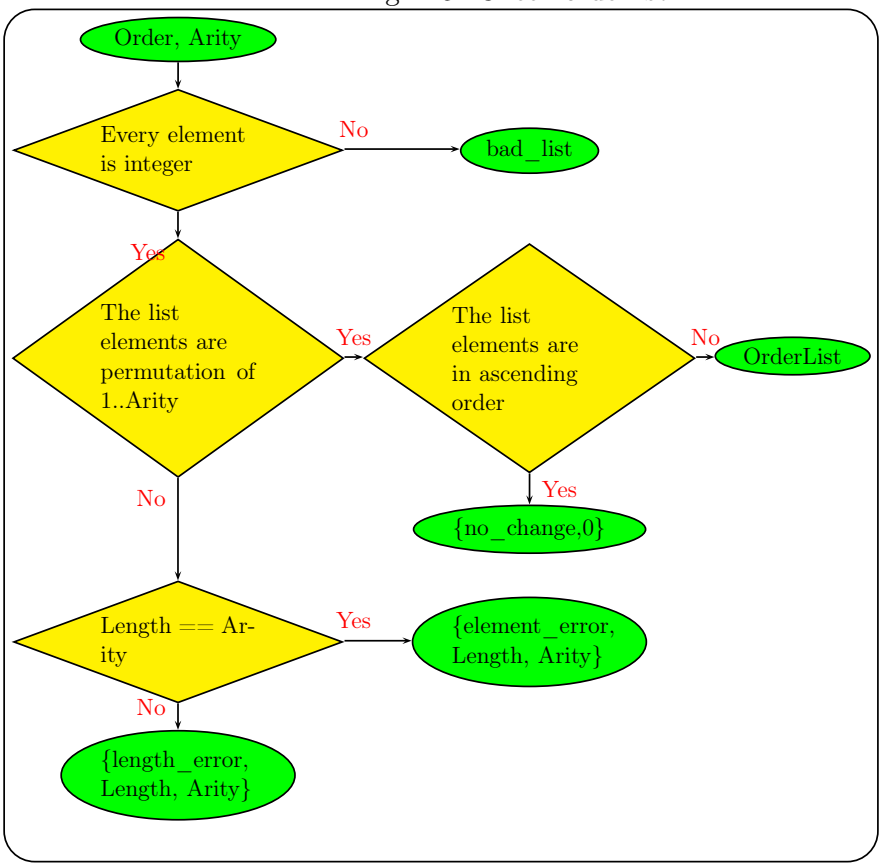


Fig. 2.4: Change the order in clauses

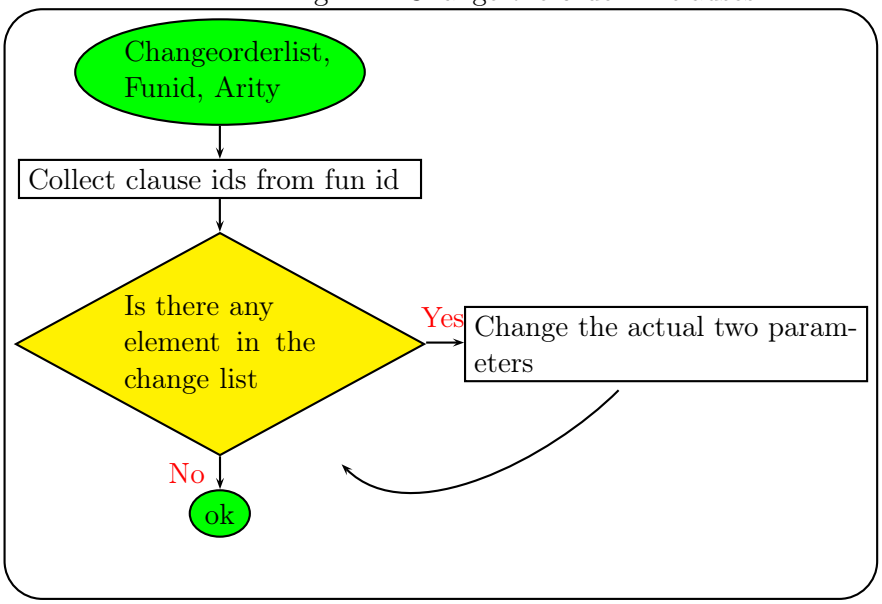


Fig. 2.5: Change the order in applications

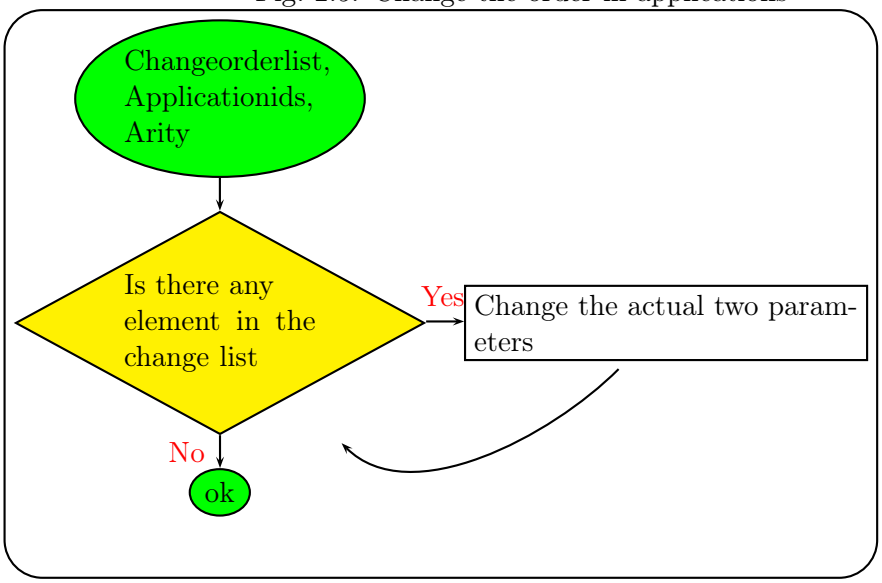


Fig. 2.6: Change the order in implicit fun calls

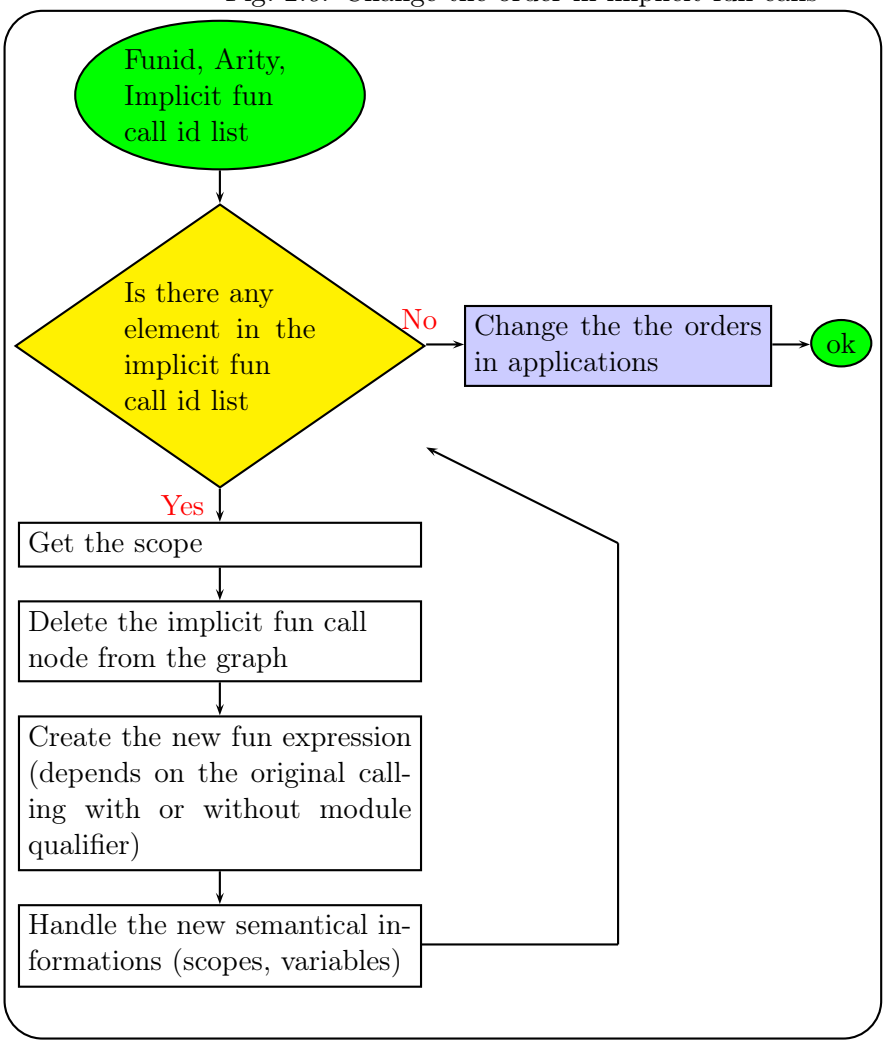


Fig. 2.7: Dynamic handling

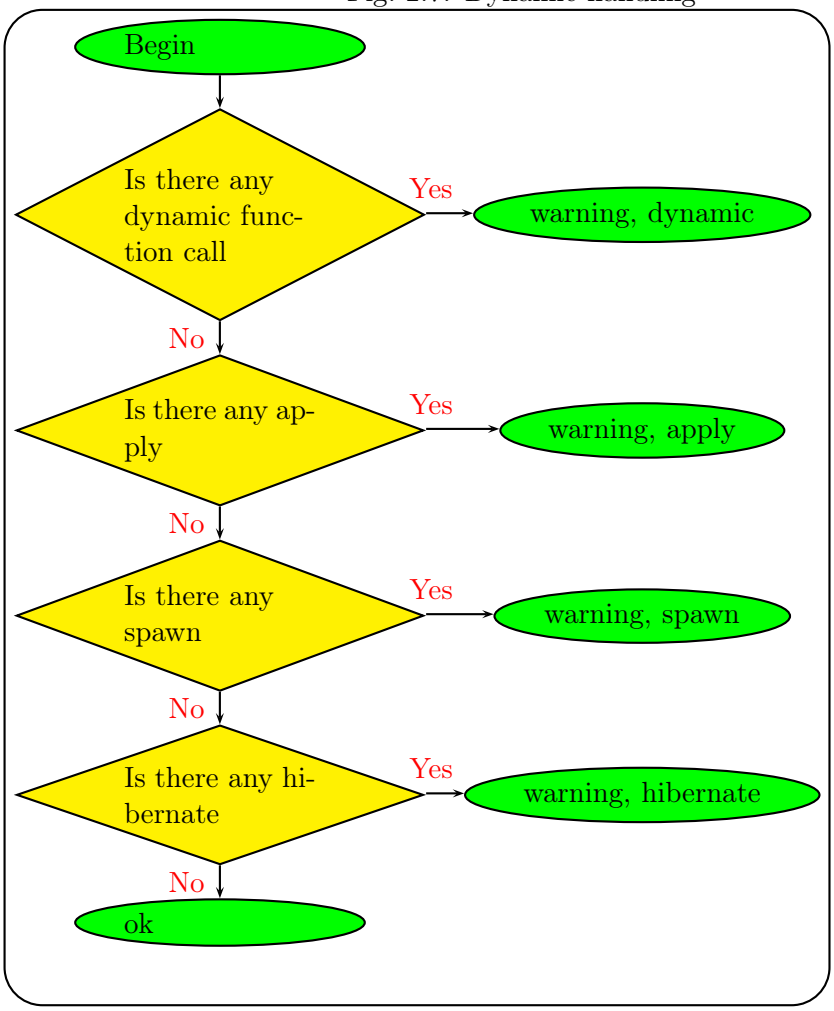


Fig. 2.8: The structure of the refac_var_elim module

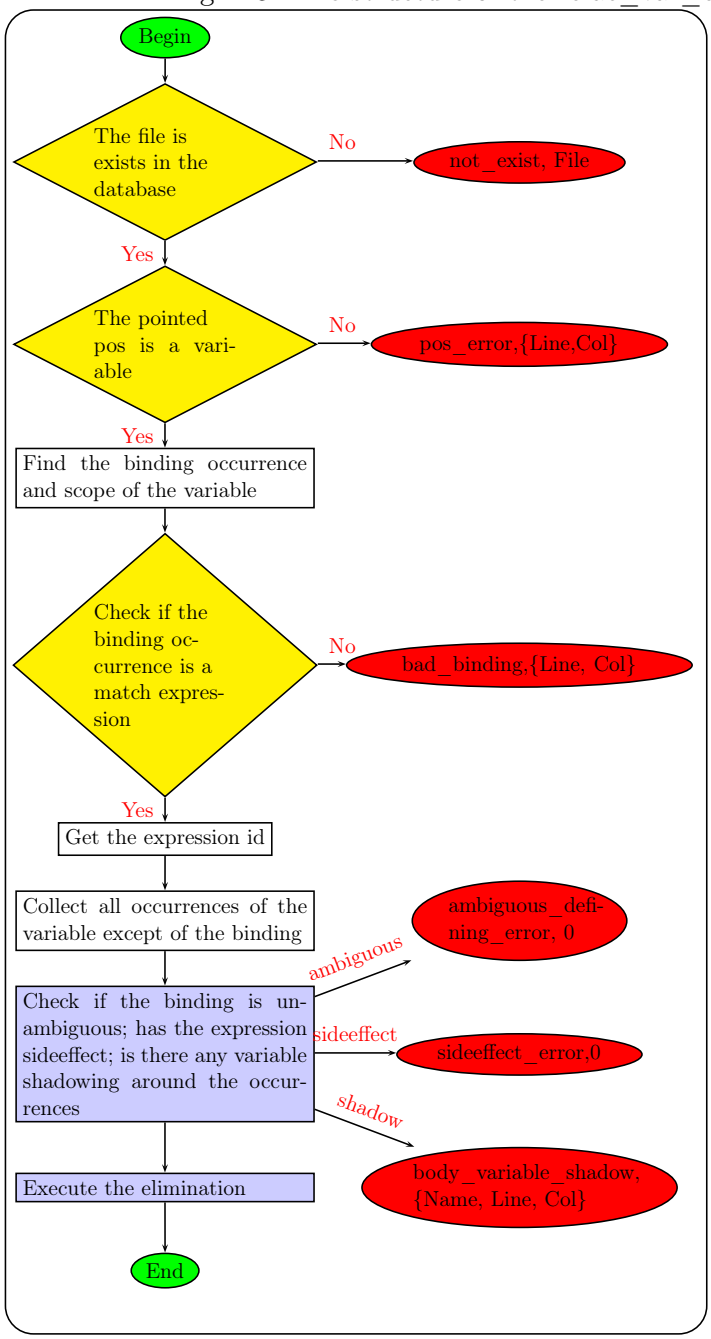


Fig. 2.9: Check if the binding is unambiguous; has the expression sideeffect; is there any variable shadowing around the occurrences

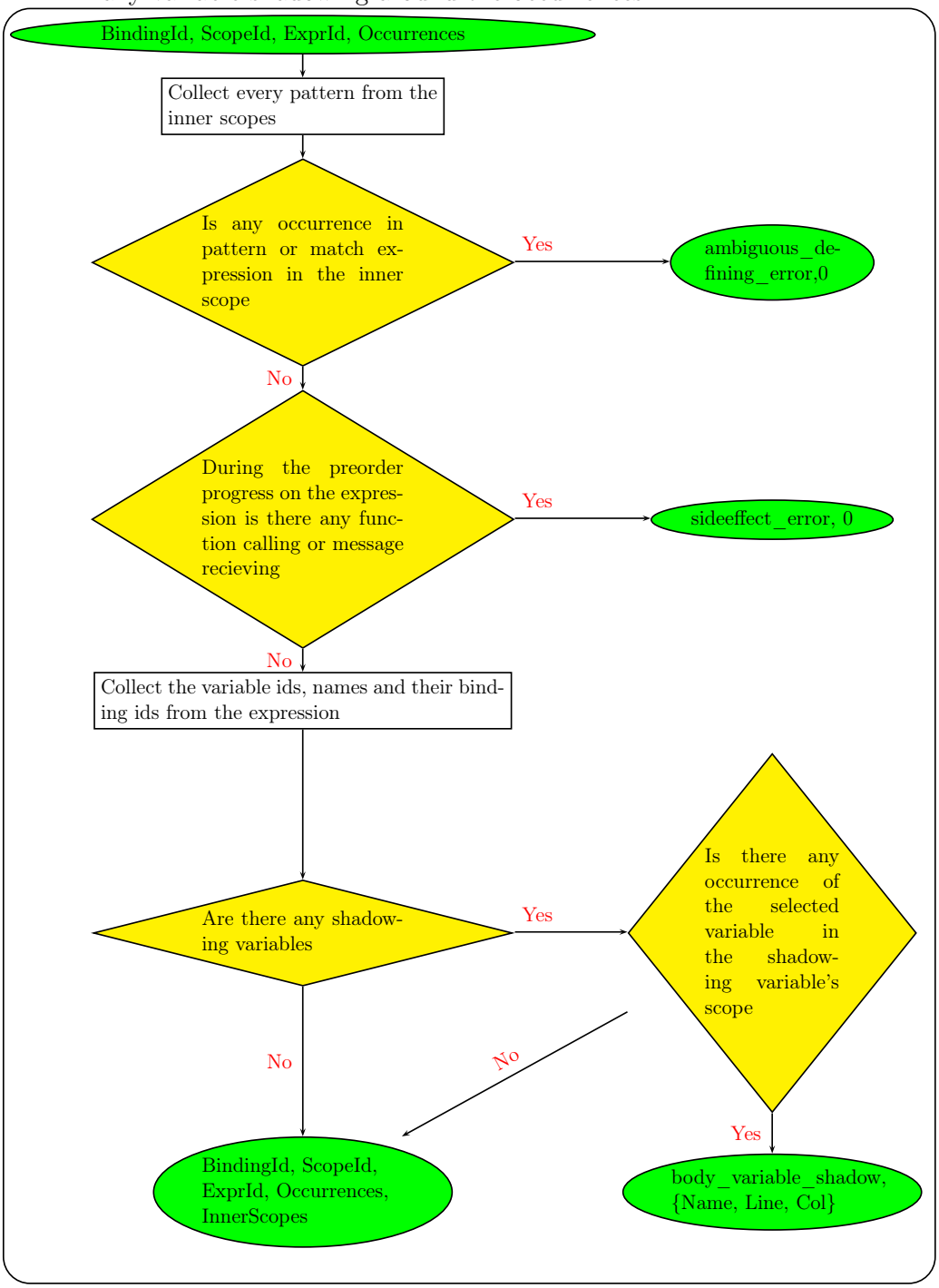


Fig. 2.10: Execute the elimination

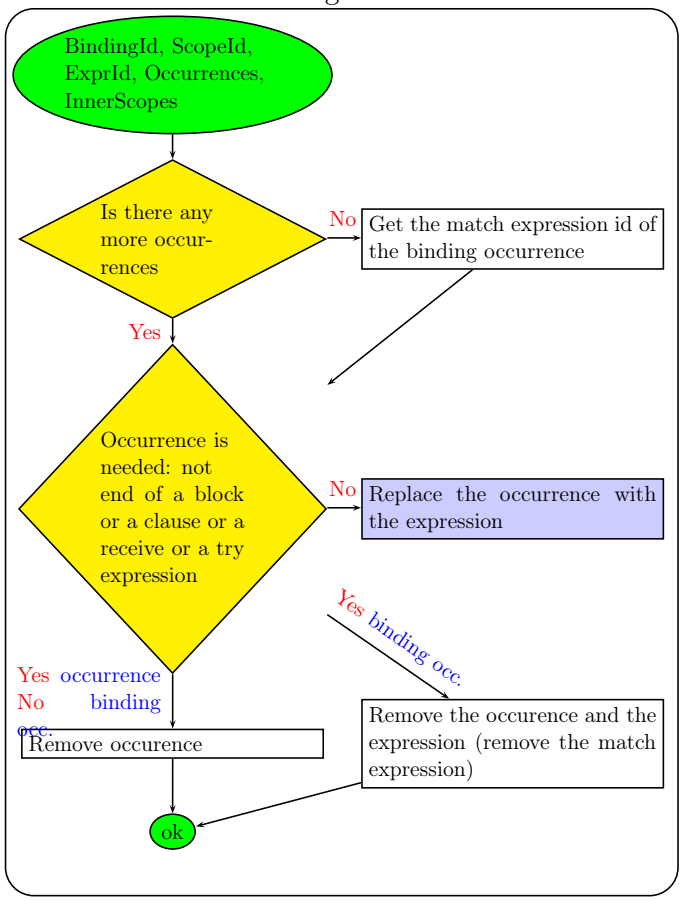
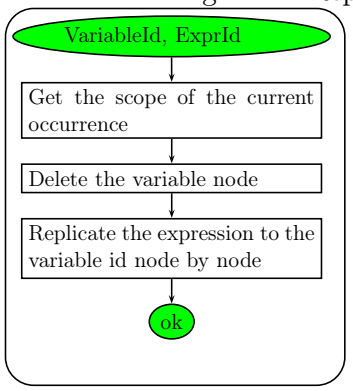


Fig. 2.11: Replace the occurrence with the expression



2.2 Implementation of the refactorings

2.2.1 Module connections

Every refactoring in the refactorer has its own module. The exported function of the refactoring is called from the `d_client` module, which is the interface module between the Distel and Emacs parts (for details see 1.1.3). The module of the refactoring uses the common refactoring modules, which contain general usable functions for every refactoring.

The exported function in the refactorings is always one function, usually with the same name as the refactoring (exception is the tuple function arguments, where the exported function is the `untuple_or_tuple_funpar/5` function). The refactoring uses functions from

- the `refactor` module to execute some general queries,
- the `refac_common` module to execute some general functionality,
- the `refac_checks` module to execute some general check functions.

For example, to get the identifier of the function which is given by the name and the arity. The other modules are the `create_nodes` and `delete_nodes` modules, which are used to delete the unnecessary nodes, and create the new syntax tree nodes (or parts) directly in the database. This is done for example in the refactorings with function arguments, when at least one of the occurrences of the function call is an implicit function call expression.

2.2.2 The applied methods and algorithms during the implementation

Storing the source code in the database

We used the preorder and postorder traversal method in order to extract the necessary information from the AST. The first traversal is a postorder traversal, because to be able to put the nodes into the database we have to give all the nodes a unique id. If the node is not a leaf, all the siblings have to have an unique id to be able to put the node in the database. Using the postorder traversal, this can be done in one run, because the postorder method first processes the leaf nodes and after it the parent nodes and so on, and a processing does not just puts the node into the database, but it associates a unique id to the node.

As we had shown in Section 1.1.2 the AST is not enough to decide that a refactoring is acceptable or not, we need also static and semantic informations. To retrieve this information we decided to use the preorder traversal, and found it easier to get the variable binding and the scoping information (For details see Section 1.1.2). Because the functions calls do not rely on the structure of the database. For example, it is impossible to shadow a function, or create embedded functions (it is only possible to create function expressions). We used the already existing informations in the database to compute the function call graph. This part of the database is recomputed when a new module is added to the database. The other parts related to an older module in the database doesn't change, because the function calls are the only one element of the language which can create connections between modules. The message sending can create connections between processes, and that could easily mean, but not necessarily, connections between modules (if the server and the client side is implemented in different modules). But to find the matching message sending and receiving parts in different code parts is really hard, and maybe impossible by using just static methods. Because of these reasons this kind of connection between modules is currently not supported.

SQL queries

The tool stores the static and semantic information of the source code in a SQL database (for details see Section 1.1.2). An ODBC connection provides the data flow between the database and the Erlang node. We tried to reduce the numbers of the queries by using more complex ones instead of more simple ones to improve the speed, because the message sending is expensive.

We made a genserver for handling the SQL queries, it hides the representation. At the moment it can work with odbc and native mysql connections, but it is easy to add other special binary connections. The form for an SQL query is calling the `refactor_db:sql_query` function with the string of the query as parameter. The return value of the function is the result of the query, a list of tuples. In this refactoring we are using mostly `select` command, but sometimes `update`, `insert` and `delete` commands. We are creating the query string by concatenating the parts with the ++ operator. There are some special conversions needed for the parameters in the following cases:

1. When the parameter is integer then `integer_to_list/1` function

makes the conversion;

2. When the parameter is a string then `io_lib:write_string/1` function makes the conversion;

3. When the parameter is a tuple in a list and only the first element is important then the conversion is `element(1,hd(MId))`.

The **select** command is used to get information from the database. The used variations of select command in the module:

- **One type** of information is needed with a simple **select** command from **one table**.

  For example: We need the module identifier that belongs to the current path. The `MId` variable will store the module identifier, the name of the table is `module` and the `File` variable contains the path. The result variable is a list in this case:

```
[{MId}] =
    refactor_db:select(
          "select mid from module where"
          " path=" ++ io_lib:write_string(File) ++";")
```

- **One type** of information is needed with a **select max** command from **one table**.

  For example: We need the stored position of the element which is selected by the user. The stored position is always the first character's position, but the user can choose any character of an element. The stored position will be the `CCol` variable, `pos` is the name of the table, `MId` is the current module identifier, `Line` and `Col` are the pointed position's elements. The plus one is needed in the column, because Emacs's first position is 0, but the parser's first position is 1. The result is the value of inside the list and tuple:

```
[{CCol}] =
    refactor_db:select(
          "select max(col) from  pos where mid="
          ++ integer_to_list(element(1,hd(MId)))
          ++ " and  line=" ++ integer_to_list(Line)
```

```
        ++ " and col<=" ++ integer_to_list(Col+1)
        ++ ";" )
```

- **One type** of information is needed with a simple **select distinct** command from **one tables**.

  For example: We need all module identifiers (but only once) in a list, where the current function occurs (the current function is the target function in the function calling expression). The `UsedLocations` variable will store the module identifiers list, the name of the table is `fun_call` and the `MId` variable contains the module identifier, and `FunId` is the identifier of the current (target) function.

  The code:

```
UsedLocations =
    refactor\_db:select(
          "select distinct mid from fun_call where tmid="
          ++ integer_to_list(MId) ++ " and target = "
          ++ integer_to_list(FunId) ++ " ;")
```

- **One type** of information is needed with a simple **select** command from **joined tables**.

- **More type** of information is needed with a simple **select** command from **one table**.

- **More type** of informations are needed with a simple **select** command from **joined tables** and between the conditions is an `embedded select` query.

  For example: We need the identifier and the node type of the current element. The result list of two element tuples will be stored in the `Ids` variable. The names of used tables are `node_type` and `pos`. `MId` is the current module identifier, `Line` and `CCol` are the position's elements. The node types are the same as in the original AST (see section 1.1.2) but are marked with integer numbers. 1 means application node, 12 means clause node:

```
Ids =
    refactor_db:select(
          "select pos.id,type from node_type, pos where"
```

```
" pos.mid=node_type.mid and pos.mid="
++ integer_to_list(MId)
++ " and node_type.id=pos.id and line="
++ integer_to_list(Line)
++ " and (type=12 or type=1) and col= "
" (select max(col) from node_type,pos where"
" pos.mid=node_type.mid and pos.mid="
++ integer_to_list(MId)
++ " and node_type.id=pos.id and line="
++ integer_to_list(Line) ++ " and col<="
++ integer_to_list(CCol)
++ " and ( type=12 or type=1 ));")
```

The **update** command is used to store the changes in the code during the refactoring directly in the database.

For example: Refreshing the arity value in the function information, when the function is exported and the arity changes (the length of the tuple is greater than 1). The actual table is fun_visib; the changing column is the argument; the new value is the NewArity. The MId is the module and the FunId is the function identifier. The pos has to be 0, because it means, that value of the argument column stores the arity of the function, if the pos is not 0, then this row belongs to a clause of the function:

```
refactor_db:update(
      "update fun_visib set argument="
      ++ integer_to_list(NewArity)
      ++ " where mid=" ++ integer_to_list(MId) ++
      " and id=" ++ integer_to_list(FunId) ++ " and pos=0;")
```

The **delete** command is used to delete information, elements from the code and database.

For example: Deleting the arguments of the clause, which will become superfluous, because they will be contracted to a new one during the refactor step. The name of the table is `clause`. The `MId` is the current module and the `ClauseId` is the current clause identifier. The `From` and `To` are the numbers of the start and end parameters of the tuple:

```
refactor_db:delete(
      "delete from clause where mid=" ++ integer_to_list(MId)
      ++ " and id=" ++ integer_to_list(ClauseId)
      ++ " and qualifier=0 and pos>" ++ integer_to_list(From)
      ++ " and pos<=" ++ integer_to_list(To) ++ " ;")
```

The **insert** command is used to add new information, elements to the database.

Solutions for branches

Every branch is decision point in the program. We used two possible solutions:

1. The first is **pattern matching**: every possible result is a new function branch. The checking of the conditions are usually pattern matching in a function parameter. For example if the program behaviour is not the same in different node types then the function will have separate clauses. Each clause in the argument list is a pattern matching to the node type. This solution is marked by (blue) italic style on the arrows in the flowchart diagram of the module (See in Section 2.1).

2. In the second one the decision is inside a **case structure**. The condition is between the `case` and `of` keywords. The condition can be an equality test or a function with result value. The possible values of the condition result will be the patterns of the branches. The cases are often layered in each other. This solution is marked with yellow diamonds in the flowchart diagram of the module (See in Section 2.1). The condition (with words) is in the diamond. The possible values are on the get-out arrows with red characters. An example source code can be seen in Figure 2 , where the second case structure is embedded, the first condition is an equality test, the second condition is a function.

Operation with lists (`map`, `flatten (my_flatten)`, `filter`)

One of the remarkable data structures is the list in the refactoring, because the return value of a SQL query is a list. We are using the following list algorithms in the refactorings (the main part of these algorithms are already written in the `lists` Erlang module):

Figure 2: **Case structures**

```
case Tree == Id of
        true ->
            true;
        false ->
            case erl_syntax_db:subtrees(MId, Tree) of
                [] ->
                    false;
                List ->
                    lists:map(fun(Elements)->
                         lists:map(fun(Element)->
                           preorder(MId, Element, Id)
                                    end, Elements)
                              end, List)
            end
end.
```

1. **Any**: to check if a condition is true in at least one element of the list. The result is a boolean value.

   For example we check with this function if the new function name with the arity causes a name clash in the import list of the module:

   ```
   case lists:any(
             fun({_,FN,A}) ->
                      (FN==FunName) and (A==NewArity)
             end,
              ImportedFunctions) of
   ```

2. **Delete**: to delete an element from the list. The result is a list.

   For example we are using this procedure to delete the current module identifier from the module identifier list, where the current function occurs. The result list will be the list of the outer modules, where the function is used (probably imported):

   ```
   UsedOuterLocations=lists:delete({MId}, UsedLocations)
   ```

3. **Filter**: to create a new list (part of the original) with the elements which fulfil the conditions. The result is a list.

   For example we sort the function calls with this function to two separate list, which belongs to an application (see "1" in the code) or implicit function (see "24" in the code):

```
ApplicationIds =
    lists:filter(
          fun ({1, _MId, _Id}) -> true;
              ({24, _MId, _Id}) -> false end, CallIds),
ImplicitFunCallIds =
    lists:filter(
          fun ({24, _MId, _Id}) -> true;
              ({1, _MId, _Id}) -> false end, CallIds)
```

4. **Flatten**: to eliminate the embedded lists in any depth. The element of the embedded list will be the elements of the new list.

5. **My_flatten**: to eliminate the embedded lists in one depth. We implemented this algorithm to improve the efficiency of the code.

6. **Map**: to execute a fun expression over every element of a list. The result is the new list.

   For example we get the list of the imported functions instead of the original import list identifier for every element of the list. The result is lists in the list:

```
lists:map(
    fun ({MId2, MId2ImportListIds}) ->
          refactor:get_imported_functions_and_ids(
                    MId2, MId2ImportListIds) end,
     ImportListIds)
```

7. **Member**: to check if an element is a member of the list. The result is a boolean value.

   For example we apply this function to check if the current function is imported in a module (part of the imported function list of the module). As this example shows we often use this function in a case structure:

```
case lists:member(
       {refactor:get_module_name(MId), FunName, OldArity},
        ImportedFunctions) of
```

Creating and deleting parse tree parts directly in the database

We need this operation for example in the tupling function arguments, and in the reorder function arguments, when one of the function call is implicit one. In the eliminate refactoring we need to delete a variable occurrence and replace it with the expression. This means some tree parts replicating directly in the database.

For example in the tupling function arguments refactoring the following algorithm is needed: If the tuple length is greater than 1, and every precondition is passed, then the arity of the function will reduce. It means that changes are needed in the definition and every function call of the function. These can be application, clause or implicit function expressions.

In the application and clause nodes some parameter will be converted to a tuple. It means in practice:

1. Collect the identifiers of the converted parameters to a list,
2. Create a new tuple node with the list of the contracted parameters as elements,
3. Update in the database the first affected parameter identifier to the identifier of the tuple in the parameter list (argument in the clause or application table),
4. Delete the other affected parameters,
5. Update the parameter positions in the same table,
6. Initialise the position and the scope of the new tuple node.

If the expression is an implicit function call it is more difficult, because we have to create unique variables and new tree parts (not just relocate an existing tree part, as above). We have to delete the implicit function calling expression and instead of it we have to create a function expression with the original parameters which calls the current function (application) with the tupled parameters. The new variables are the original parameters in the correct number. In practice it is the following:

1. Delete the implicit function node,
2. Create the new unique variables,
3. Create an atom node to the function name,
4. Create an application to the function calling,
5. Connect the new application to the original function,
6. Create a clause node (this will be the element of the function expression),
7. Create a function expression,
8. Initialise the positions and the scopes of the new variables,
9. Create a new scope for the new clause,
10. Connect the new variables together.

The new algorithm for the function call storing

The function call and its definition connecting's first algorithm was really inefficient. For every new module it threw away the previously known connections. After that it collected all the function calls and function definitions and stored the found connections.

The second algorithm introduces a new table fun_cache which is responsible for caching all the known function calls, and definitions. This way if we want to put a new module into the database we only have to look up the needed function calls and definitions from this table, store the found connections, and add the new modules informations to the cache table.

3. EFFICIENCY ANALYSIS ON TWO REFACTORINGS

In this chapter we try to discuss the following problems:

- Are the sql queries fast and efficient enough?
- Is the structure of the database optimised?
- Speed analysis of the tool parts -> the critical point is the number of the sql queries
- Native mysql or odbc module should be the connection towards the database?
- Experimental analysis of the refactorings on different input files

We tested the tool on more than 200 individual test cases (consists of one module each), which are produced by the other project members in order to be able to test the refactor tool. The tool worked properly with all of the test cases.

We tested the tool on a big project (approximately 90 modules and 1 MB Erlang source code) to test the tool on a multi module system.

We measured the speed of the tool on a system with 1.66Ghz Intel Pentium processor, 1024 MB DDR2 RAM and Windows XP operating system.

We will show in this chapter how we made the tool more efficient – beginning from the smallest stones (sql queries) toward the whole system.

We have to measure not only the time and efficiency of the two refactorings, but the time to store the source into the database, and reload from there. If the user would like to use any refactoring, he/she has to put it into the database first. After the source is once in the database several refactorings can be applied on it after each other. While the user does not edit anything in the source by hand, it is no need to reparse and store the code again.

3.1 SQL queries

We already presented our typical queries grouped by functionality in the Subsection 2.2.2.

The execution of every query consists of the following steps:

- Connecting: (3)
- Sending query to server: (2)
- Parsing query: (2)
- Executing the current command: (depends on the query type). For example inserting row: (1 × size of row) and inserting indexes: (1 × number of indexes)
- Closing: (1)

In the parentheses the numbers indicate the approximate proportions. This list shows, that for execution a query there are constant quite remarkable time cost: the connecting, sending and closing parts. We tried to reduce this cost by writing more complex queries instead of several short queries. In the insert and update commands this option is the only way to speed up the tool. If we send as much rows as possible in a list together in one query to be inserted or updated we succeed to optimise this query types.

For example the first query can substitute the other code parts, and the tool can be faster by three connection and closing times:

The complex query:

```
ExportListElementIds = refactor_db:select(
   "select list.element from form_list,node_type,attribute_,"
   ++ "list where node_type.mid = form_list.mid and "
   ++ "attribute_.mid = list.mid and list.mid = form_list.mid"
   ++ " and list.mid = " ++ integer_to_list(MId) ++ "and "
   ++ "node_type.id = form\_list.form and node_type.type=4"
   ++ "and attribute_.id = form and list.id = argument"
   ++ " and attribute_.pos = 1 and attribute_.id in"
   ++ " (select attribute_.id from attribute_, name where "
   ++ "attribute_.mid = name.mid and name.mid = "
```

```
    ++ integer_to_list(MId) ++ " and attribute_.argument = "
    ++ "name.id and name =\"export\");" )
```

The original queries:

```
[{Export}] = refactor_db:select(
   "select id from name where name=\"export\";"),

[{ExportId}] = refactor_db:select(
   "select id from attribute_ where argument="
   ++ integer_to_list(Export) ++ ";"),}

[{ListId}] = refactor_db:select(
  "select argument from attribute_ where pos=1 and id= "
  ++ integer_to_list(ExportId) ++";"),}

ElementIds = refactor_db:select(
  "select element from list where id= "
  ++ integer_to_list(ListId) ++";")}
```

3.1.1 Explain plans for the select queries

We get a new problem with using complex queries: are the new queries optimised? In MySQL a similar option is available to optimise the select queries, as in Oracle, just the output is different: a table instead of some data in tree structure.

When preceding a `SELECT` statement with the keyword `EXPLAIN`, MySQL displays information from the optimizer about the query execution plan. That is, MySQL explains how it would process the `SELECT`, including information about how tables are joined and in which order.

The explain plan table Each output row from the table provides information about one table, and each row contains the following columns:

- **id** A sequential number of the SELECT within the query.

- **select-type** The type of the select: it is *simple*, or complex. In the second case the most outermost and the embedded queries are marked with different words (*primary, subquery, union, derivered* etc.).
- **table** The name of the table which the output row refers to.
- **type** The join type. The different join types are listed here, ordered from the best type to the worst:
  1. *system*: The table has only one row (= system table).
  2. *const*: The table has at most one matching row, which is read at the start of the query. Because there is only one row, values from the column in this row can be regarded as constants by the rest of the optimizer. const tables are very fast because they are read only once. *const* is used when comparing all parts of a PRIMARY KEY or UNIQUE index to constant values.
  3. *eq_ref*: One row is read from this table for each combination of rows from the previous tables. Other than the system and const types, this is the best possible join type. It is used when all parts of an index are used by the join and the index is a PRIMARY KEY or UNIQUE index. *eq_ref* can be used for indexed columns that are compared using the = operator. The comparison value can be a constant or an expression that uses columns from tables that are read before this table.
  4. *ref*: All rows with matching index values are read from this table for each combination of rows from the previous tables. *ref* is used if the join uses only a leftmost prefix of the key or if the key is not a PRIMARY KEY or UNIQUE index (in other words, if the join cannot select a single row based on the key value). If the key that is used matches only a few rows, this is a good join type.
  5. *ref_or_null*: This join type is like *ref*, but with the addition that MySQL does an extra search for rows that contain NULL values. This join type optimisation is used most often in resolving subqueries.
  6. *index_merge*: This join type indicates that the Index Merge optimisation is used. In this case, the **key** column in the output row contains a list of indexes used, and **key_len** contains a list of the longest key parts for the indexes used.

7. *unique_subquery*: This type replaces ref for some IN subqueries of the following form: `value IN ( SELECT primary_key FROM single_ table WHERE some_expr)` *unique_subquery* is just an index lookup function that replaces the subquery completely for better efficiency.
8. *index_subquery*: This join type is similar to *unique_subquery*. It replaces IN subqueries, but it works for non-unique indexes in subqueries of the following form: `value IN (SELECT key_column FROM single_table WHERE some_expr)`
9. *range*: Only rows that are in a given range are retrieved, using an index to select the rows. The **key** column in the output row indicates which index is used. The **key_len** contains the longest key part that was used. The **ref** column is NULL for this type. *range* can be used when a key column is compared to a constant using any of the =, <>, >, >=, <, <=, IS NULL, <=>, BETWEEN, or IN operators.
10. *index*: This join type is the same as *ALL*, except that only the index tree is scanned. This usually is faster than *ALL* because the index file usually is smaller than the data file. MySQL can use this join type when the query uses only columns that are part of a single index.
11. *ALL*: A full table scan is done for each combination of rows from the previous tables. This is normally not good if the table is the first table not marked const, and usually very bad in all other cases.

- **possible_keys** It indicates which indexes MySQL can choose from use to find the rows in this table. This column is totally independent of the order of the tables.
- **keys** It indicates the key (index) that MySQL actually decided to use.
- **key_length** It indicates the length of the key that MySQL decided to use.
- **ref** It shows which columns or constants are compared to the index named in the key column to select rows from the table.

- **rows** It indicates the number of rows MySQL believes it must examine to execute the query.
- **Extra** It contains additional information about how MySQL resolves the query.

For example we would like to show some MySQL explain plans. At the time of the measuring the whole bigger test project was in the database.

- A select distinct query from one table with where options. The description of the query can be seen in Section 2.2.2. The query and its original result can be seen before the explain plan of it in the Figure 3.1.

  In the table we can see the already known information, that is a simple select query from one (the fun_call) table. In this example we can see the plus informations, that the query searches are not based on key columns, so the type is *index*, not *const*. The only possibility to make faster this query is to add the `tmid` and `target` columns to the primary keys, but in that case all the columns in the table would be keys. It is not efficient from the cache size point of view, and we also loose the idea of the keys. The other interesting information in the **row** column, which gives some information of the current table size.

```
mysql> select distinct mid from fun_call where tmid=59 and target=6981;
+-----+
| mid |
+-----+
|  59 |
|  73 |
+-----+
2 rows in set (0.00 sec)

mysql> explain select distinct mid from fun_call where tmid=59 and target=6981;
+----+-------------+----------+-------+---------------+---------+---------+------+------+-------------+
| id | select_type | table    | type  | possible_keys | key     | key_len | ref  | rows | Extra       |
+----+-------------+----------+-------+---------------+---------+---------+------+------+-------------+
|  1 | SIMPLE      | fun_call | index | NULL          | PRIMARY | 8       | NULL | 3511 | Using where |
+----+-------------+----------+-------+---------------+---------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

Fig. 3.1: Result and explain plan of a select distinct query from one table

- A select query from two joined tables with where options. The description of the query can be seen in Section 2.2.2. The query and its original result can be seen before the explain plan of it in the Figure 3.2.

The table shows, that this query as fast as possible, because it contains only *const* type selects from each tables.

```
mysql> select name from function, name where name.mid=function.mid
    -> and pos=0 and clause=name.id and function.mid=59 and function.id=6981;
+---------+
| name    |
+---------+
| do_recv |
+---------+
1 row in set (0.00 sec)

mysql> explain select name from function, name where name.mid=function.mid
    -> and pos=0 and clause=name.id and function.mid=59 and function.id=6981;
+----+-------------+----------+-------+---------------+---------+---------+-------------------+------+-------+
| id | select_type | table    | type  | possible_keys | key     | key_len | ref               | rows | Extra |
+----+-------------+----------+-------+---------------+---------+---------+-------------------+------+-------+
|  1 | SIMPLE      | function | const | PRIMARY       | PRIMARY | 12      | const,const,const |    1 |       |
|  1 | SIMPLE      | name     | const | PRIMARY       | PRIMARY | 8       | const,const       |    1 |       |
+----+-------------+----------+-------+---------------+---------+---------+-------------------+------+-------+
2 rows in set (0.00 sec)
```

Fig. 3.2: Result and explain plan of a select query from joined tables

- A complex select query from four joined tables and embedded select with where and in options. The query gives back the identifiers of the export list(s) elements from a module. The query and its original result can be seen before the explain plan of it in the Figure 3.3.

  The explain's result shows, that this query is as optimal as it can, because there are only *ref* or *eq_ ref* select types, We can not ameliorate it until every select would be *eq_ ref*, because we are working also with not key datas, and it can allow only the *ref* type. It can be seen, that the query has two select a primary and an embedded dependent subquery, which is joined from the `attribute_` and `name` tables and it is dependent on the outer query.

  We succeed to realise with this analysis a really unefficient query in the tool. The details and the solution can be seen in the Section 3.5.1.

3.1.2 Optimising the delete queries

To delete all rows from a table, `TRUNCATE TABLE table_name` is faster than `DELETE FROM`, but the Truncate operations are not transaction-safe; an error occurs when attempting one in the course of an active transaction or active table lock. The fastest option is for delete everything from a table is to drop the whole table, and create it again. In this solution we don't need to traverse the lines in the table, and delete them one by one. So we decided to use this `DROP` and `CREATE` order to initialise the database.

```
mysql> select list.element from form_list, node_type, attribute_, list
    -> where node_type.mid=form_list.mid and attribute_.mid=list.mid and
    -> list.mid=form_list.mid and list.mid=59 and node_type.id=form_list.form
    -> and node_type.type=4 and attribute_.id=form and list.id=argument and
    -> attribute_.pos=1 and attribute_.id in (select attribute_.id from
    -> attribute_, name where attribute_.mid=name.mid and name.mid=59 and
    -> attribute_.argument=name.id and name="export");
+---------+
| element |
+---------+
|      26 |
|      29 |
|      32 |
+---------+
3 rows in set (0.00 sec)

mysql> explain select list.element from form_list, node_type, attribute_, list
    -> where node_type.mid=form_list.mid and attribute_.mid=list.mid and
    -> list.mid=form_list.mid and list.mid=59 and node_type.id=form_list.form
    -> and node_type.type=4 and attribute_.id=form and list.id=argument and
    -> attribute_.pos=1 and attribute_.id in (select attribute_.id from
    -> attribute_, name where attribute_.mid=name.mid and name.mid=59 and
    -> attribute_.argument=name.id and name="export");
+----+--------------------+------------+--------+---------------+---------+---------+---------------------------------+------+-------------+
| id | select_type        | table      | type   | possible_keys | key     | key_len | ref                             | rows | Extra       |
+----+--------------------+------------+--------+---------------+---------+---------+---------------------------------+------+-------------+
|  1 | PRIMARY            | attribute_ | ref    | PRIMARY       | PRIMARY | 4       | const                           |   19 | Using where |
|  1 | PRIMARY            | list       | ref    | PRIMARY       | PRIMARY | 8       | const,parse.attribute_.argument |    1 |             |
|  1 | PRIMARY            | node_type  | eq_ref | PRIMARY       | PRIMARY | 8       | const,parse.attribute_.id       |    1 | Using where |
|  1 | PRIMARY            | form_list  | ref    | PRIMARY       | PRIMARY | 4       | const                           |  233 | Using where |
|  2 | DEPENDENT SUBQUERY | attribute_ | ref    | PRIMARY       | PRIMARY | 8       | const,func                      |    1 | Using where |
|  2 | DEPENDENT SUBQUERY | name       | eq_ref | PRIMARY       | PRIMARY | 8       | const,parse.attribute_.argument |    1 | Using where |
+----+--------------------+------------+--------+---------------+---------+---------+---------------------------------+------+-------------+
6 rows in set (0.00 sec)
```

Fig. 3.3: Result and explain plan of a complex select query from joined tables

3.2 Database structure

The structure of the database can be seen in the Tables 3.1, 3.2, 3.3 and 3.4 . The main ideas and explanations of the database are in the section 1.1.2.

3.2.1 Database tables

The tabular contains the tables of the database. The primary keys are marked with bold characters. The table names are marked with italic characters which are our own tables, not nodes of the syntax tree. If there are no type after a variable it is an integer, in every other case the type states between parentheses after the name.

We made 58 tables. If it is possible, we are working everywhere with the unique identifiers of the elements. The erlang abstract syntax tree has originally 50 types (syntactical category). We made one table for one syntactical category in 44 cases. The remaining 6 categories are always leaves in the tree, so we made some simplifies:

- If the type has only a name attribute, for example the atom and variable categories, we just store them in the `name` table. We could avoid the redundancy, that some table would be totally part of an other one. We store also the identifier's type and the position in separate tables..

Tab. 3.1: Structure of the database (Part 1)

Name of the table	**Col 1**	**Col 2**	**Col 3**	**Col 4**	**Col 5**
application	**mid**	**id**	pos	argument	
arity_qualifier	**mid**	**id**	body	arity	
attribute_	**mid**	**id**	pos	argument	
binary_	**mid**	**id**	**pos**	field	
binary_field	**mid**	**id**	**pos**	argu-ment	
block_expr	**mid**	**id**	**pos**	body	
case_expr	**mid**	**id**	**pos**	argument	
catch_expr	**mid**	**id**	expres-sion		
char_	**mid**	**id**	value		
class_qualifier	**mid**	**id**	class	body	
clause	**mid**	**id**	**pos**	**qualifier**	argument
comment	**mid**	**id**	**pos**	argument (varchar 255)	
cond_expr	**mid**	**id**	**pos**	clause	
conjunction	**mid**	**id**	**pos**	argument	
disjunction	**mid**	**id**	**pos**	argument	
float_	**mid**	**id**	value (float)		
forbidden_names	type	**forbid-den_name** (vchar 255)			
form_list	**mid**	**id**	**pos**	form	

Tab. 3.2: Structure of the database (Part 2)

Name of the table	**Col 1**	**Col 2**	**Col 3**	**Col 4**	**Col 5**	**Col 6**
fun_cache	**mid**	**id**	module (varchar 255)	fun (varchar 255)	arity	type
fun_call	**mid**	**id**	tmid	target		
fun_expr	**mid**	**id**	**pos**	clause		
fun_visib	**mid**	**id**	**pos**	argument		
function	**mid**	**id**	**pos**	clause		
generator	**mid**	**id**	pattern	body		
id_count	**mid**	formlid	num			
if_expr	**mid**	**id**	**pos**	clause		
implicit_fun	**mid**	**id**	name_id			
infix_expr	**mid**	**id**	lft	oper	rght	
integer_	**mid**	**id**	value (varchar 255)			
list	**mid**	**id**	**pos**	element		
list_comp	**mid**	**id**	**pos**	argument		
macro	**mid**	**id**	**pos**	argument		
match_expr	**mid**	**id**	pattern	body		
module	**mid**	path (vchar 255)				
module_qualifier	**mid**	**id**	module	body		
name	**mid**	**id**	name (vchar 255)			

Tab. 3.3: Structure of the database (Part 3)

Name of the table	**Col 1**	**Col 2**	**Col 3**	**Col 4**	**Col 5**
node_type	**mid**	**id**	type		
parentheses	**mid**	**id**	body		
pos	**mid**	**id**	line	col	
precomment	**mid**	**id**	**pos**	**qualifier**	argument (vchar 255)
prefix_expr	**mid**	**id**	operator	argument	
postcomment	**mid**	**id**	**pos**	**qualifier**	argument (vchar 255)
qualifier_name	**mid**	**id**	**pos**	segment	
query_expr	**mid**	**id**	body		
receive_expr	**mid**	**id**	**pos**	**qualifier**	argument
record_access	**mid**	**id**	argument	type	field
record_expr	**mid**	**id**	**pos**	argument	
record_field	**mid**	**id**	name_id	value	
record_index_expr	**mid**	**id**	type	field	
rule	**mid**	**id**	**pos**	argument	
scope	**mid**	**id**	scope		
scope_visib	**mid**	**id**	target		
size_qualifier	**mid**	**id**	body	size	
string	**mid**	**id**	value (vchar 5000)		
text	**mid**	**id**	value (vchar 5000)		

Tab. 3.4: Structure of the database (Part 4)

Name of the table	**Col 1**	**Col 2**	**Col 3**	**Col 4**	**Col 5**
try_expr	**mid**	**id**	**pos**	**qualifier**	argument
tuple	**mid**	**id**	**pos**	element	
var_visib	**mid**	**id**	target		

- If the type does not contain any attributes, we don't need to store anything, just the identifier's type and the position. These categories are the `nil`, `underscore` and `eof_marker`.
- The `error_marker` and `warning_marker` types are impossible to have in a tree. At the beginning there are checks concerning these, and if there any in the tree, the refactor tool gives a message to the user, that the source contained errors. These types indicate syntactical errors in the code therefore it can not be refactored safely. Considering this it is straightforward, that we do not need these types in the database.

We have 14 tables, which are general tables, or contain semantical informations of the code. We can sort them to the following categories:

- *Semantical information* storing tables:
  - `fun_call`: it connects the function definition with the usage for every function call.
  - `fun_visib`: it stores the clauses and the arity of the functions.
  - `scope`: it stores the most inner scope of every element.
  - `scope_visib`: it stores the hierarchy of the scopes.
  - `var_visib`: it stores the first occurrence for every variable.
- *Speed and efficiency* raising tables:
  - `fun_cache`: this table is the newest, it is in the database since we wrote the new algorithm to the function call storing as part of the source storing into the database. The difference of the speed can be seen by comparing of 3.7 and 3.8 charts. The table

contains also semantical informations, like the arity and the type of the function call, but the advantage of using this table in the efficiency can allow to it to be in this category. It is the only table in the database, where the module and the function name are stored in string format not in the name table, so we does not need to join at least two tables to connect the identifier of the function and the arity to the name.

  - `node_type`: it stores the syntactical category of every element. From the efficiency point of view, it is faster to search in one table by primary keys (*const* select type), than search 44 tables for it (it is faster in average 22 times, if we do not calculate the extra rows in the 44 tables).

- *Collector* tables:

  - `module`: it contains the full path of every module.
  - `name`: it contains every name in string format, which are in the stored source.
  - `pos`: it contains the position information of every node.
  - `precomment` and `postcomment`: they are syntactical categories, but they are not part of the syntax tree, but we need to store them to be able to recover the original source.

- Tables for special *refactor precondition check*s:

  - `forbidden_names`: This table contains every forbidden name for the renaming refactorings (BIFs, reserved words, and user defined ones).

- Tables for helping the *implementation*:

  - `id_count`: it stores the biggest identifier in every module. It helps to find unique identifier for the new elements.

In the *speed and efficiency* category of the tables sometime contain redundant information, what can be found in the other tables, but here they are collected for a different reason and into a different structure. The gain on the efficiency with using these tables makes it worth to store the redundant information, as it is detailed in each table description.

3.3 Speed analysis of the tool parts

3.3.1 Point charts for the comparison of the different possibilities when storing and recovering the source

We will present the measure result of the bigger project, when we put it into the database and recover it from there. We are working with odbc connection here, because we chose to use this connection type. The details of the decision can be found in Section 3.4. The charts for the native mysql connection can be seen there, in the Figures 3.9 and 3.10.

Every chart contains the whole time, the put into the database time and the recovering of the source from the database. We will live with the simplification that according the design rules for Erlang the maximal file should not be more than 500 lines. It is approximately 17000 bytes. We will show the results of the files, which were below this limit, except in the Figure 3.4. The tool normally uses the odbc connection (see 3.4), but we measured everything with the native mysql connection too. The other comparison is between the old and the new version of the code storing. We had to write a new algorithm for the function call storing because the old one was very slow, when more than one module were in the database. The old results can be seen in the Figure 3.7, and the new results in the Figures 3.8. It can be seen, that the old one was almost constant four time slower for every module as the average value of the new version. The problem appears just in the putting into the database part, as it can be seen in the three datarow. The problem was, that the algorithm always calculated all of the function calls again. After there was a big module put into the database, for every new module it took almost the same time to put the code into the database as the big ones. This resulted some unacceptable running times.

In the new version we are using a new algorithm and a new table (fun_cache), for the details see 2.2.2. The algorithm is linear for the whole project (figure 3.8), as it was with the individual modules (figure 3.6). The maximal time is approximately the same.

We choose the file size as one of the axles, because in average we assumed, that it is directly proportional to the complexity of the code. It can be seen from the charts, that is not exactly true, because the values have quite big variance due to the different complexity between approximately the same size modules. We will detail this influence in Section 3.5.

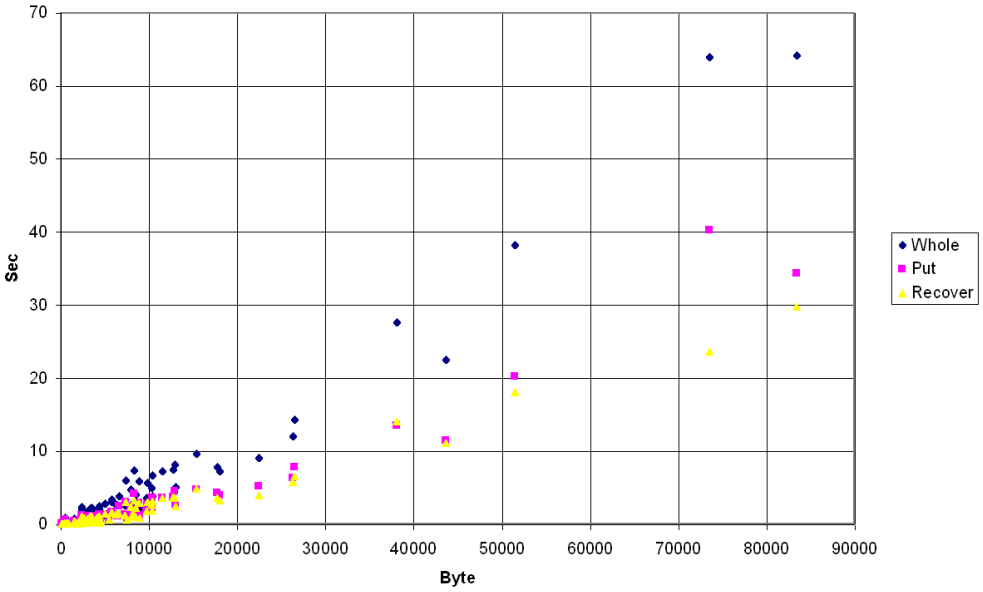


Fig. 3.4: The storing and recovering times of the whole project with the old algorithm, when the database is initialised after every module. The file size is up to 90000 bytes instead of the recommended 17000.

3.3.2 Results of the time profiling analysis to the different functionalities

In Erlang there are different tools for profiling execution time and function call numbers. We chose the `fprof` module, because it can give both of this informations. The module is used to profile a program to find out how the execution time is used. Trace to file is used to minimize runtime performance impact. It presents both own time i.e how much time a function has used for its own execution, and accumulated time i.e including called functions and also the number of the function callings.

The three main steps of the profiling:

1. Tracing: The trace contains entries for function calls, returns to function, process scheduling, other process related (spawn, etc) events, and garbage collection. All trace entries are time stamped. Using this tool, we does not need to use special compilation, because the `fprof` deletes all previous tracing, and places the trace flags and erase them when the process is ready.

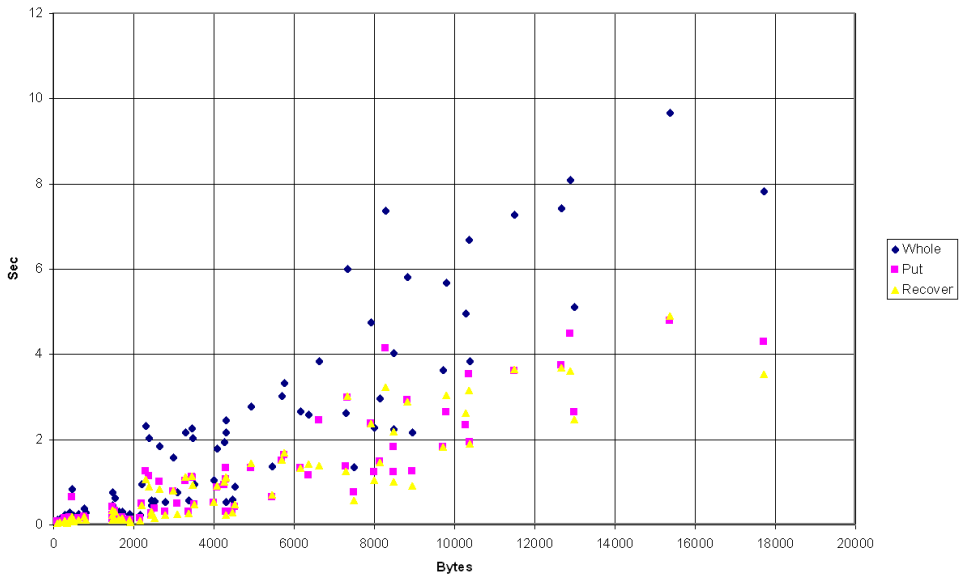


Fig. 3.5: The storing and recovering times of the project with the old algorithm, when the database is initialised after every module. The file size is maximum 17000 bytes.

2. Profiling: The program reads the trace file, creates and calculates the execution call stack, and calculates the raw profile datas from the stack and the timestamps. The result is in the server state, and it can be written to file.

3. Analysing: The program sorts, and filters the raw profile datas and writes the result to the console or into a file. The result can be parsed by the Erlang parser.

In every tracing tool, the effect of the analysis to the execution time is one of the most important question. This program tries to minimize the runtime performance degradation by using trace to file. The impact is the biggest, when the original program uses the file system a lot.

In our case the exact runtimes are not important, because we already presented the main trends, in this subsection we just try to analyse the proportion of different parts' runtime, and compare to the connection types, comparing their runtime usage structure on same inputs.

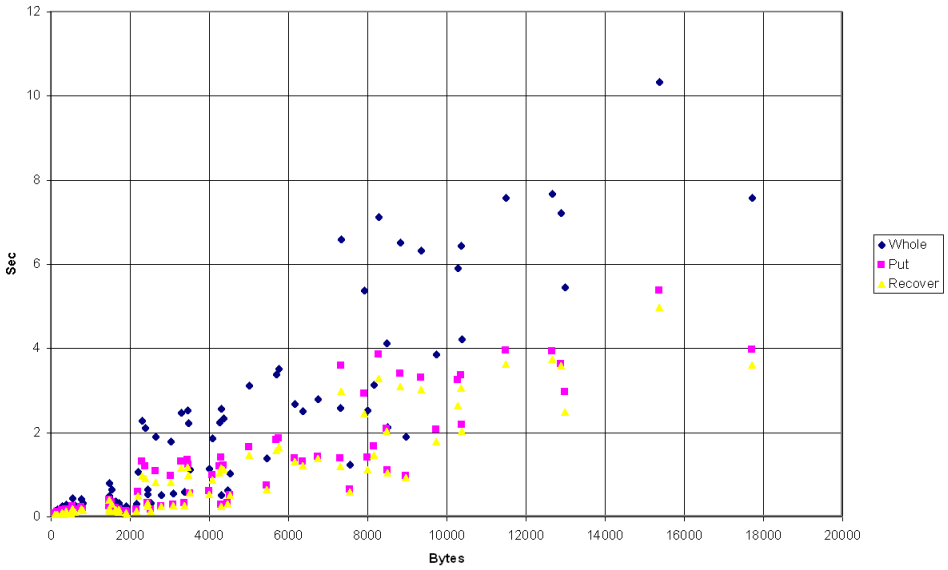


Fig. 3.6: The storing and recovering times of the project with the new algorithm, when the database is initialised after every module. The file size is maximum 17000 bytes.

An introduction example of the profile result

There is an example of the analysis result by analysing the following refactoring:

We would like to eliminate the `S` variable from the body of the `triangle` function as it can see below.

var_elim.erl original

```
-module(var_elim).

-export([triangle/3]).

triangle(A, B, C) ->
    S = (A + B + C) / 2,
    math:sqrt(S * (S - A) * (S - B) * (S - C)).
```

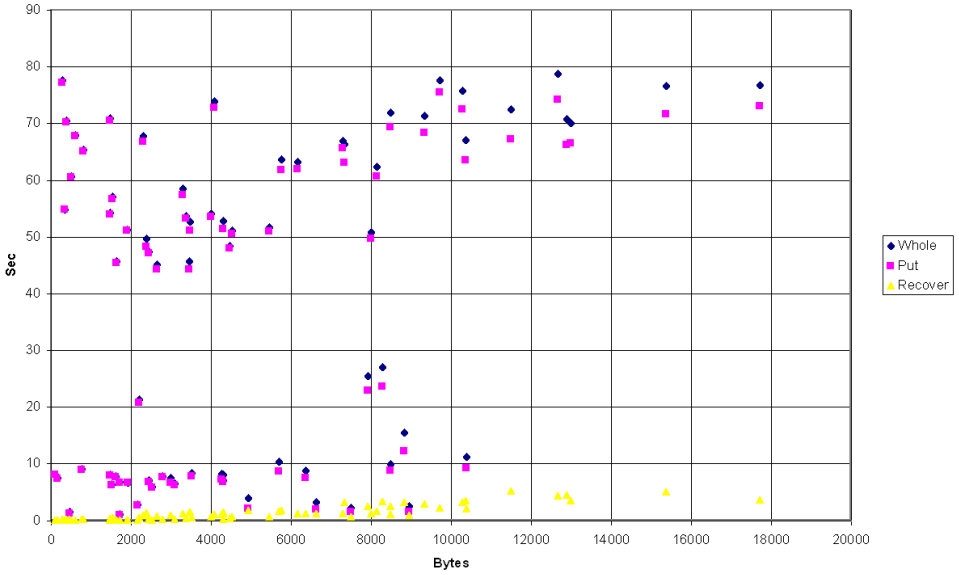


Fig. 3.7: The storing and recovering times of the project with the old algorithm, when the database is initialised just at the beginning. The file size is maximum 17000 bytes.

var_elim.erl after the elimination

```
-module(var_elim).

-export([triangle/3]).

triangle(A, B, C) ->
    math:sqrt((A + B + C) / 2 * ((A + B + C) / 2 - A) *
                ((A + B + C) / 2 - B)
                * ((A + B + C) / 2 - C)).
```

The analysis file starts with the specification of the applied options. We were manipulating with the `sort` option, because if we use the `acc` state, then we could see the main functional parts' time usage comparatively in the beginning. But if we choose the `own` state, then we could see the most time demanding functions in the beginning.

All time values in the printout are in milliseconds.

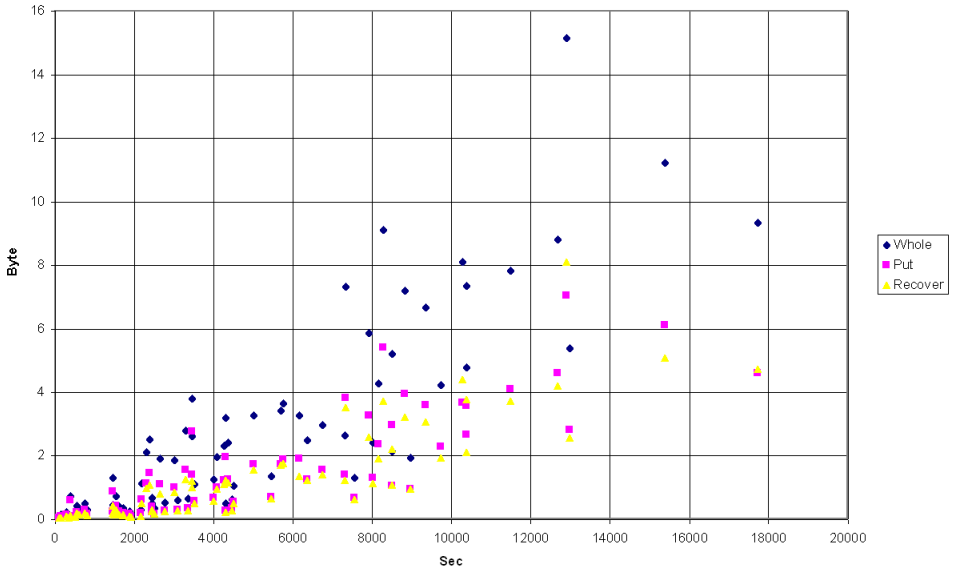


Fig. 3.8: The storing and recovering times of the project with the new algorithm, when the database is initialised just at the beginning. The file size is maximum 17000 bytes.

The next separate row contains the summarized result of the whole analysis (begins with totals):

- The `CNT` column contains the total number of function calls that was found during the analysis.
- The `ACC` column is the total time of the trace from first timestamp to last.
- The `OWN` column is the sum of the execution time in functions found in the trace, not including called functions.

In the following parts are the summarized datas of each process. Every line begins with the process identifier. The `ACC` column is undefined because summing the ACC times of all calls in the process makes no sense - we would get something like the ACC value from totals above multiplied by the average depth of the call stack.

In the main part of the analysis consists of one paragraph per called function. In this example we just cut the first four paragraphs. The function

marked with '%' is the one the paragraph concerns. For example in the first paragraph `{fprof ,apply_start_stop,4}` (fprof:apply_start_stop/4) function is the marked. Above the marked function are the calling functions - those that has called the marked, and below are those called by the marked function.

The columns of the measuring results are:

- The `CNT` column contains the number of times the function has been called.
- The `ACC` column is the time spent in the function including called functions.
- The `OWN` column is the time spent in the function not including called functions.

The rows for the calling functions contain statistics for the marked function with the constraint that only the occasions when a call was made from the row's function to the marked function are accounted for. The row for the marked function simply contains the sum of all calling rows. The rows for the called functions contains statistics for the row's function with the constraint that only the occasions when a call was made from the marked to the row's function are accounted for.

Eliminate variable (odbc)

```
%% Analysis results:
{  analysis_options,
 [{callers, true},
  {sort, acc},
  {totals, false},
  {details, true}]}.

%                                          CNT       ACC       OWN
[{ totals,                                51333,  797.742,  230.724}].  %%%


%                                          CNT       ACC       OWN
[{ "<0.36.0>",                            51063,undefined,  230.409}].   %%

{[{undefined,                                0,  797.742,    0.002}],
 { {fprof,apply_start_stop,4},               0,  797.742,    0.002},     %
 [{{d_client,var_elim,3},                    1,  797.740,    0.001},
  {suspend,                                  1,    0.000,    0.000}]}.

{[{{fprof,apply_start_stop,4},               1,  797.740,    0.001}],
 { {d_client,var_elim,3},                    1,  797.740,    0.001},     %
 [{{d_client,do_refactoring,2},              1,  797.739,    0.004}]}.
```

Eliminate variable (odbc) Part 2

```
{[{{d_client,var_elim,3},                          1,  797.739,    0.004},
  {{into_db,set_positions,1},                      1,    0.000,    0.001}],
 { {d_client,do_refactoring,2},                    2,  797.739,    0.005},     %
 [{{d_client,'-var_elim/3-fun-0-',3},              1,  452.992,    0.001},
  {{out_from_db,create_code,2},                    1,  225.115,    0.010},
  {{into_db,set_positions,1},                      1,  119.619,    0.003},
  {{into_db,simultaneous_visiting,2},              1,   90.683,    0.005},
  {{d_client,init,0},                              1,    0.009,    0.002}]}.

{[{{refactor_db,select,1},                       972,  444.416,    1.949},
  {{refactor_db,insert,1},                       215,   90.666,    0.430},
  {{refactor_db,delete,1},                       116,   60.091,    0.239},
  {{refactor_db,update,1},                       122,   30.956,    0.244},
  {{refactor_db,commit,0},                         2,   15.767,    0.004}],
 { {gen_server,call,2},                         1427,  641.896,    2.866},     %
 [{{gen,call,3},                                1427,  638.944,    1.430},
  {suspend,                                        74,    0.074,    0.000},
  {garbage_collect,                                12,    0.012,    0.012}]}.
```

3.3.3 The analysis of the profiling results of the tool

We will show in this section just the most interesting parts of the result. We will discuss the result of the eliminate variable, and reorder function parameter refactoring measuring with both odbc and mysql connection.

We always keep at least the marked line of the first paragraphs, and we cut out just the remaining ones. In every analysis the first function calling is the `fprof:apply_start_stop/4` function which calls the needed refactor function.

Reorder function parameters

We will analyse the reorder function parameters refactoring through an example. The original file is the same as the above example. We would like to reverse the parameters in the `triangle` function. We need to give the 3 2 1 sequence as the new parameter order at the beginning.

var_elim.erl original

```
-module(var_elim).

-export([triangle/3]).

triangle(A, B, C) ->
    S = (A + B + C) / 2,
    math:sqrt(S * (S - A) * (S - B) * (S - C)).
```

var_elim.erl after reordering

```
-module(var_elim).

-export([triangle/3]).

triangle(C, B, A) ->
    S = (A + B + C) / 2,
    math:sqrt(S * (S - A) * (S - B) * (S - C)).
```

It is a really interesting comparison, that a minimal output changing (only two characters) in a really small source hides 23 600 function calls. The tool has to analyse the whole source, gathering the needed informations, checking the preconditions, execute the changing, and recover the new source from the database.

It can be seen from the third paragraph, that to execute the refactoring (`reorder_funpar/4`) is approximately 6%, and the remaining time is needed to build up the current source from the database (`create_code/2`). We think that it is a good result, because all the checking, and the performing of the refactoring is much faster, than the traverse of the source in the database, and the write it out into a file.

From the paragraph of the main refactor function (`reorder_funpar`) the following scales can be calculated:

- 96% of performing the reordering.
- 4% of gathering the information from the database for the precondition checking and the performing. The main part here is the warnings function (before this algorithm upgrade (details in 3.5.1) it was 55%), which collects the possible dynamic function calls from the whole code, which are not supported in this version. Later, when this options will be supported, this collecting will move to the precondition checking part.
- almost 0% of making the precondition checks.

This proportions can be very various, for example, if the reordering affects only one clause of one function, but in the module are a lot of other functions, with a lot of function calls. The checking and gathering could be more complex, resulting higher runtime percent.

Eliminate variable (odbc) sorted by acc

```
%                                              CNT       ACC       OWN
[{ totals,                                   23574,  266.302,  104.632}].  %%%

{[{undefined,                                    0,  266.302,    0.002}],
 { {fprof,apply_start_stop,4},                   0,  266.302,    0.002},     %
 [{{d_client,reorder_funpar,4},                  1,  266.300,    0.001},
  {suspend,                                      1,    0.000,    0.000}]}.

{[{{fprof,apply_start_stop,4},                   1,  266.300,    0.001}],
 { {d_client,reorder_funpar,4},                  1,  266.300,    0.001},     %
 [{{d_client,do_refactoring,2},                  1,  266.299,    0.004}]}.

{[{{d_client,reorder_funpar,4},                  1,  266.299,    0.004},
  {{into_db,set_positions,1},                    1,    0.000,    0.001}],
 { {d_client,do_refactoring,2},                  2,  266.299,    0.005},     %
 [{{out_from_db,create_code,2},                  1,  173.098,    0.010},
  {{into_db,set_positions,1},                    1,   76.963,    0.003},
  {{into_db,simultaneous_visiting,2},            1,   60.522,    0.005},
  {{d_client,'-reorder_funpar/4-fun-0-',4},      1,   16.210,    0.001},
  {{d_client,init,0},                            1,    0.024,    0.002}]}.

...
{[{{refactor_db,select,1},                     346,  120.163,    0.693},
  {{refactor_db,update,1},                      51,   27.436,    0.102},
  {{refactor_db,commit,0},                       2,   15.390,    0.004},
  {{refactor_db,set_autocommit,1},               1,    0.014,    0.002}],
 { {gen_server,call,2},                        400,  163.003,    0.801},     %
 [{{gen,call,3},                               400,  162.183,   14.254},
  {suspend,                                     18,    0.018,    0.000},
  {garbage_collect,                              1,    0.001,    0.001}]}.

...
 { suspend,                                   1498,  161.674,    0.000},     %

...
{[{{d_client,'-reorder_funpar/4-fun-0-',4},      1,   16.209,    0.011}],
 { {refac_reorder_funpar,reorder_funpar,4},      1,   16.209,    0.011},     %
 [{{refac_reorder_funpar,perform_refactoring,6},  1,   15.535,    0.005},
  {{refac_common,warnings,0},                    1,    0.232,    0.004},
  {{refac_common,get_module_id,1},               1,    0.158,    0.002},
  {{refac_common,get_id_from_pos,4},             1,    0.076,    0.003},
  {{refac_common,get_fun_calls,2},               1,    0.048,    0.007},
  {{refac_reorder_funpar,get_changing_order,2},  1,    0.041,    0.002},
  {{refac_common,find_the_function,2},           1,    0.032,    0.002},
  {{refactor,get_arity_from_fun_id,2},           1,    0.030,    0.008},
  {{refac_reorder_funpar,produce_list,1},        1,    0.025,    0.001},
  {{refac_checks,check_orderList,2},             1,    0.021,    0.004}]}.
```

An other really interesting measuring data, that the refactorer spent the 61% of the runtime by executing sql queries.

Eliminate the variable

The example refactoring is the same as in the Subsection 3.3.2. It can be seen, that for executing a quite short and simple refactoring, the tool is making more than 50000 function call, but it's still below one second. It is much more than the reorder refactoring to the same example function, because in this case we have to replicate code parts (expressions) several times, and delete the old variables. All these functionality work directly on the database, which cause a lot of new function calls.

It can be seen from the third paragraph, that to execute the refactoring (`var_elim/3`) is approximately 55 %, and the remaining time is needed to build up the current source from the database (`create_code/2, set_position/1, simultaneous_visiting/2`). We think that it is a good result, because all the checking, and the performing of the refactoring (including new code parts generating) is just a little bit more, than the traverse of the source in the database, and write it out into a file.

Eliminate variable (odbc) sorted by acc

```
%                                          CNT       ACC       OWN
[{ totals,                                51333,  797.742,  230.724}].  %%%


 { {fprof,apply_start_stop,4},                0,  797.742,    0.002},     %

{[{{fprof,apply_start_stop,4},                1,  797.740,    0.001}],
 { {d_client,var_elim,3},                     1,  797.740,    0.001},     %
 [{{d_client,do_refactoring,2},               1,  797.739,    0.004}]}.

{[{{d_client,var_elim,3},                     1,  797.739,    0.004},
  {{into_db,set_positions,1},                 1,    0.000,    0.001}],
 { {d_client,do_refactoring,2},               2,  797.739,    0.005},     %
 [{{d_client,'-var_elim/3-fun-0-',3},         1,  452.992,    0.001},
  {{out_from_db,create_code,2},               1,  225.115,    0.010},
  {{into_db,set_positions,1},                 1,  119.619,    0.003},
  {{into_db,simultaneous_visiting,2},         1,   90.683,    0.005},
  {{d_client,init,0},                         1,    0.009,    0.002}]}.

{[{{refactor_db,select,1},                  972,  444.416,    1.949},
  {{refactor_db,insert,1},                  215,   90.666,    0.430},
  {{refactor_db,delete,1},                  116,   60.091,    0.239},
  {{refactor_db,update,1},                  122,   30.956,    0.244},
  {{refactor_db,commit,0},                    2,   15.767,    0.004}],
 { {gen_server,call,2},                    1427,  641.896,    2.866},     %
 [{{gen,call,3},                           1427,  638.944,    1.430},
  {suspend,                                  74,    0.074,    0.000},
  {garbage_collect,                          12,    0.012,    0.012}]}.
```

From the paragraph of the main refactor function (`eliminate_variable`) the following scales can be calculated:

- 7% of gathering the information from the database for the precondition checking and the performing.
- 4% of making the precondition checks.
- 89% of performing the elimination.

These proportions can be very various, for example, if the eliminating affects only one occurrence of one variable, but in the same function are a lot of other variables. The checking and gathering could be more time consuming.

Eliminate variable (odbc) sorted by acc

```
%                                                CNT       ACC       OWN
{[{{d_client,'-var_elim/3-fun-0-',3},              1,  452.991,    0.014}],
 { {refac_var_elim,eliminate_variable,3},          1,  452.991,    0.014},     %
 [{{refac_var_elim,perform_refactoring,5},         1,  404.805,    0.003},
  {{refac_common,local_preorder_var,2},            1,   16.923,    0.003},
  {{refac_checks,check_if_body_doesnt_have_sideffects,2},   1,   16.056,    0.004},
  {{refactor,get_name_from_name_id,2},             1,   13.709,    0.008},
  {{refac_checks,check_if_binding_is_unambiguous,3},   1,    0.945,    0.007},
  {{refac_checks,check_are_match_body_variables_shadowed,4},   1,    0.178,    0.002},
  {{refac_common,get_module_id,1},                 1,    0.160,    0.002},
  {{refac_common,get_id_from_pos,4},               1,    0.074,    0.003},
  {{refac_common,get_inner_scope,2},               1,    0.034,    0.001},
  {{refactor,get_every_occurrence_of_a_var_from_id,2},   1,    0.031,    0.008},
  {{refac_common,get_patternIdBody,2},             1,    0.031,    0.002},
  {{refactor,get_var_bind_occ_and_scope_from_id,2},   1,    0.030,    0.008},
  {{lists,delete,2},                               1,    0.001,    0.001}]}.
```

After the tool parsed and stored the source into database, every procedure is done inside the database. This means, that the tool has to call and execute a lot of sql queries. This can be read from both part of the analysis. We used the gen_server behaviour ([8], [12]) to create a module, which hides the actually used connection type from the other parts of the system. It supports every kind of sql query, with both connection type. The call/2 function calls is the innermost function, which sorts the different query types. The refactorer spent 70-80 % of the whole time with executing sql queries. It is a really important question from the efficiency point of view, if the executions of the sql queries are fast enough. We presented two possible solutions, the details can be read in the Section 3.4.

It can be seen, that most of the queries are select, so it was important to analyse the speed of this kind of queries. The details can be found in the Section 3.1.1. The slowest query type is the `commit`. If we are calling `commit`

after every query, then the connection opening and closing will appear two times for every query. We tried to minimise the number of the `commits` with switching off the `autocommit` option. For example in this refactoring we just call two times the `commit`, instead of more than 1000, as the number of the other queries. For example here is only two, which is 0.2% of the number of the queries, but in the runtime it means approximately 8%.

Eliminate variable (odbc) sorted by own

```
%                                          CNT       ACC       OWN
[{ totals,                                51333,  812.006,  285.888}].  %%%

{[{{refactor_db,select,1},                  972,  374.824,   29.775},
  {{refactor_db,insert,1},                  215,   89.377,    0.430},
  {{refactor_db,update,1},                  122,   58.873,    0.244},
  {{refactor_db,delete,1},                  116,    1.716,    0.239},
  {{refactor_db,commit,0},                    2,   44.895,    0.004}],
 { {gen_server,call,2},                    1427,  569.685,   30.692},     %
 [{{gen,call,3},                           1427,  538.908,    1.430},
  {garbage_collect,                          12,    0.012,    0.012},
  {suspend,                                  73,    0.073,    0.000}]}.

 { {erlang,'++',2},                        6024,   29.906,   29.690},     %

 { {prettypr,rewrite,2},                   3553,   43.907,   17.262},     %

 { {lists,map,2},                           236,  327.270,   15.375},     %
```

Main conclusion for every commands in the refactorer

It can be seen from every measures, that the main part of the runtime is executing sql queries.

The result in different commands:

- In the storing and recovering the source it is 50-60%. In this commands we are using really big non database dependent codeparts (we have to parse and traverse the original code to build up the abstract syntax tree, which we can store in the database.)
- In the eliminate variable refactoring it is 70-80%.
- In the reorder function parameters refactoring it is approximately 60-65%.

If we want to make our tool reasonably fast, we had to find the optimal solution to handle these calls.

3.4 Native mysql contra odbc module

As the results of the Section 3.3.2 shows, the critical point of the runtime in the tool is the connection speed toward the sql server.

We tried two possible solutions:

1. ODBC driver
2. Native mysql module

We will present the point charts to compare the two option. The basis of the measurement is the same bigger project. The point charts for the ODBC connection are already presented before, in the Figures 3.6 and 3.8.

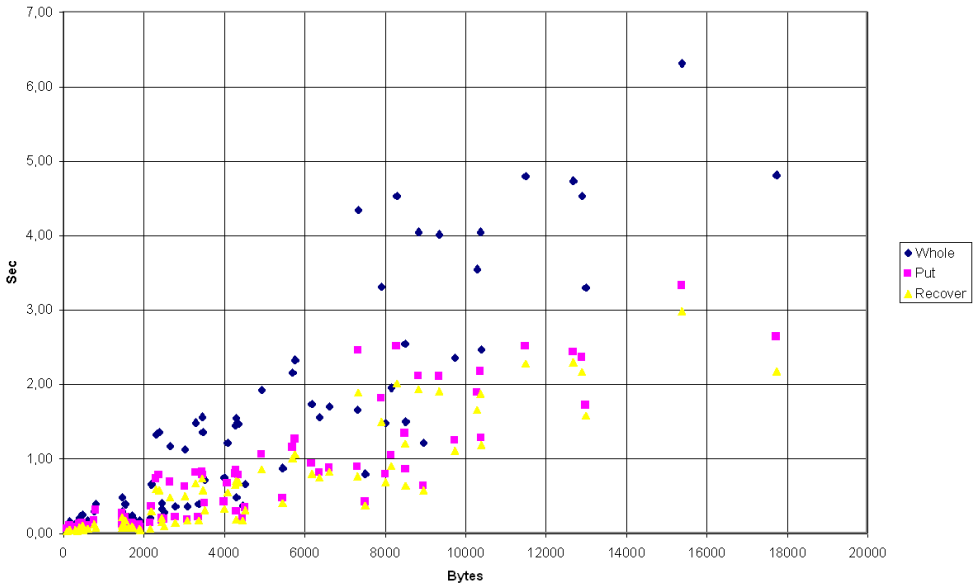


Fig. 3.9: The storing and recovering times of the project with the new algorithm, when the database is initialised after every module using mysql connection

We wrote the tool originally with the ODBC connection. If we are using the native mysql connection we have to convert the results of the queries, because they are not the same types. It is known, that the string conversions are not the best solution, but we needed them to convert the result. Even with this plus conversions the tool with native mysql connection is approximately 1,5 - 2 times faster as the odbc connection. The different rates of the

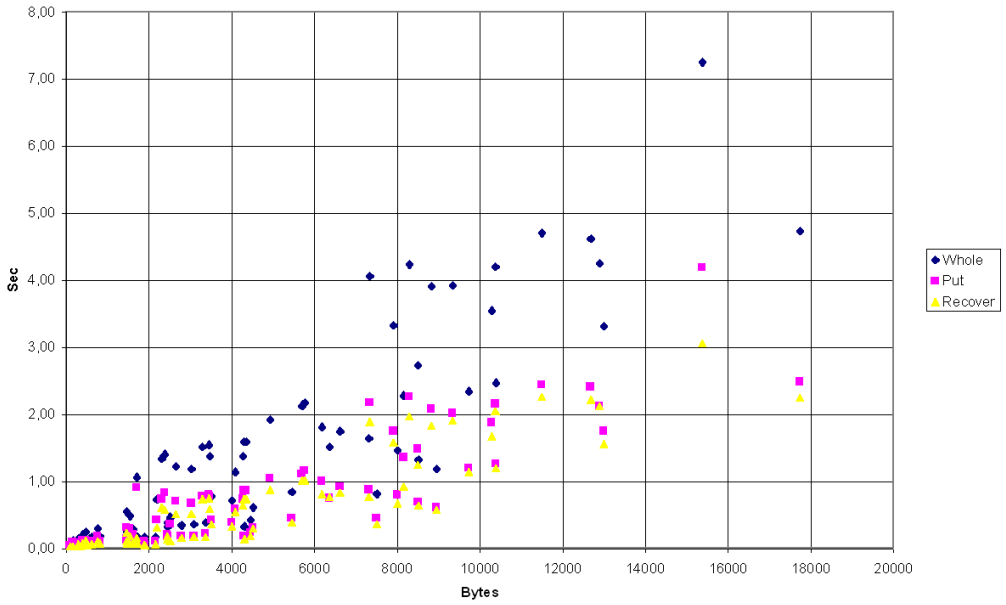


Fig. 3.10: The storing and recovering times of the project with the new algorithm, when the database is initialised just at the beginning using mysql connection

runtime structure can be read from the trace analysis of the same refactoring with the different connections.

- 60% the sql query calls of the runtime with native mysql in the eliminate variable refactoring instead of 70-80% with the odbc connection.
- 61% the sql query calls of the runtime with native mysql in the eliminate variable refactoring which almost the same as with the odbc connection.
- In the storing and recovering of the sorce, the changing is smaller it reduced to 48-58% from 50-60%. The reason: in these parts the tool uses the traverses of the original syntax tree in almost the 50% percent of the runtime, and during this the tool does not call sql queries.

The comparison of some basic technical details of the two connections can be seen in the Table 3.5.

Description	ODBC	Native Mysql
Usable database	Any SQL with ODBC	MySQL
Number of started programs	4 Win, 5 Linux	1
Average in-out time of an individual module in Windows	2,38 sec	1,49 sec
Average in-out time of a module in a project in Windows	2,66 sec	1,50 sec
Runtime relation to the file size	linear	linear
Average time for eliminate variable in Windows	3,42	2,48
Average time for reorder function parameters in Windows	3,56	2,01

Tab. 3.5: Comparison of the ODBC and native mysql connections

3.4.1 Platform dependent decision of the usage of the connections

As a summary we can say, that the mysql connection is faster, but much more specific, than the ODBC connection.

In Linux we have serious problems with the ODBC driver, because it takes more than 1 minutes (60 sec) to put a small file into the database (it is approximately 100 times slower, than the native mysql), and this is not acceptable. We recommend to use the native mysql for Linux users, even it is mean to loose the possibility to use other databases (Oracle for example).

In Windows system the difference is just 1,5 - 2 times, it is not very significant. We decided to use the tool with odbc connection not to loose the possibility of using Oracle (or any other) database.

These are just recommendations. The user always can choose (at every starting of the tool), which connection he/she would like to use. Technically it can be done in by creating a configuration file to the application. The user does not need to update and translate the source of the tool.

3.5 Experimental analysis

We will analyse the two refactorings in this chapter. In an experimental analysis is important to determine the best case and the worst case of the

input. According to the programming rules, we will choose the maximal file size for 500 lines. We examine separately the effect of the file size, and the complexity of the source to the runtime and the number of the sql queries. We count also the number of the sql queries, not only the runtime, because it shows the complexity of the executed refactorings. Every information gathering, precondition check, refactoring performing is made directly in the database by executing sql queries.

3.5.1 Experimental analysis of the reorder function arguments refactoring

The best case for the reordering refactoring is when there are no calls for the target function, and the function has only one clause.

The worst case is for the reordering point of view, when the target function has a lot of function calls, even implicit calls, where new code parts are needed to generate.

Effect of the file size in the best case

We created 43 test file. The first one contains only one function with one clause without any function calls. In the other test files we increased the number of the functions, but we don't added any function calls, just increased the length of the file by adding new functions. In the first 20 files we are increasing the number of the functions by 1, but after it by 10 until we reach the 500 lines limit.

In every test file the target function is the test/2 function. The refactoring modifies only the parameter order of the first function.

Example codes for testing:

The shortest file

```
-module(test1).

-export([test/2]).

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

The 3rd file

```
-module(test3).

-export([test/2]).

test(A, B) -> A + B.

test_1(A, B) -> A + B.

test_2(A, B) -> A + B.
```

Result Table

Functions	ODBC	MySQL	SQL	Functions	ODBC	MySQL	SQL
1	0,125	0,094	162	40	1,204	0,672	2814
2	0,140	0,125	230	50	1,515	0,797	3494
3	0,157	0,156	298	60	1,797	1,047	4174
4	0,187	0,141	366	70	2,047	1,140	4854
5	0,219	0,156	434	80	2,312	1,625	5534
6	0,250	0,156	502	90	2,516	1,453	6214
7	0,281	0,204	570	100	2,891	1,641	6894
8	0,281	0,203	638	110	3,156	1,797	7574
9	0,313	0,218	706	120	3,421	1,922	8254
10	0,344	0,266	774	130	3,610	2,031	8934
11	0,390	0,250	842	140	3,937	2,359	9614
12	0,422	0,250	910	150	4,250	2,469	10294
13	0,422	0,281	978	160	4,735	2,641	10974
14	0,484	0,282	1046	170	4,640	2,672	11654
15	0,469	0,343	1114	180	5,172	2,890	12334
16	0,594	0,328	1182	190	5,406	2,860	13014
17	0,562	0,329	1250	200	5,563	3,093	13694
18	0,579	0,359	1318	210	5,984	3,328	14374
19	0,609	0,359	1386	220	6,188	3,438	15054
20	0,641	0,360	1454	230	6,579	3,578	15734
30	0,921	0,531	2134	240	6,734	3,641	16414
				250	6,922	3,984	17094

The result of the measures can be seen in the Figure 3.11.

The dependency is linear between the length of the file and the runtime. The gradient is quite big, but the runtimes remain in the acceptable area. The slowing down is quite big due to the much more precondition checks, while the transformation is the same in every test file.

The little fractions could be there due to the different database cachings.

The MySQL connection is more effective, because the gradient is lower than with the ODBC connection.

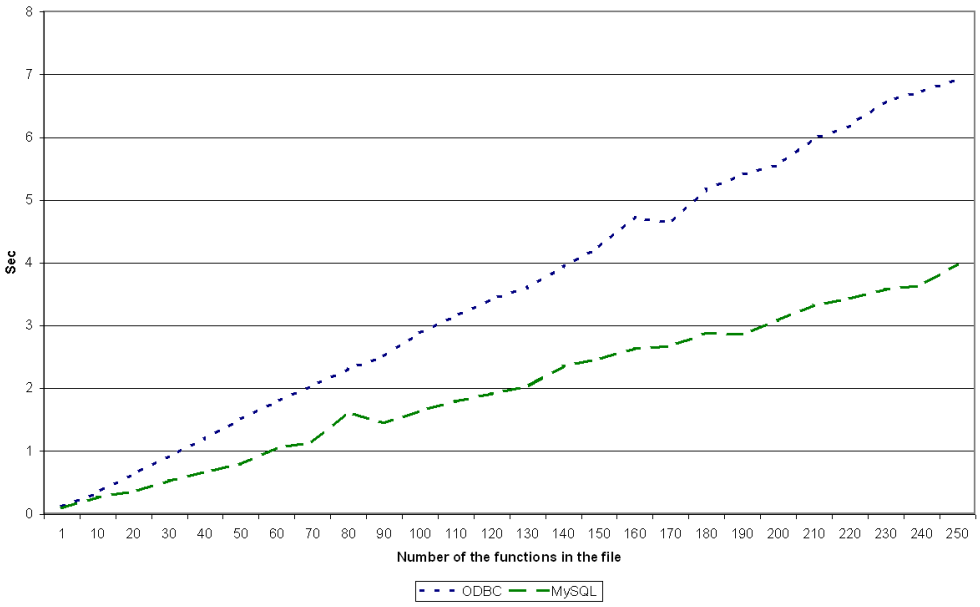


Fig. 3.11: Runtimes of the reorder function arguments refactoring by increasing file size with the best case

Effect of the function call number from the code complexity with constant file size

We created 43 test file. All files contain 250 functions. The first one contains only one function call of target function. In the other test files we increased the number of the function calls (every function contains 0 or 1 function calls), every function calls referred to the first function, in which we would like to reorder the parameters. In the first 20 files we are increasing the number of the function calls by 1, but after it by 10 until we reach the 500 lines limit.

In every test file the target function is the test/2 function. The refactoring modifies only the parameter order of the first function.

Example code for testing (every file contains 250 functions, we only show the first three in this examples)

The first file

```
-module(test1).

-export([test/2]).

test(A, B) -> A + B, test(A, B).

test_1(A, B) -> A + B.

test_2(A, B) -> A + B.
```

The 2nd file

```
-module(test2).

-export([test/2]).

test(A, B) -> A + B, test(A, B).

test_1(A, B) -> A + B, test(A, B).

test_2(A, B) -> A + B.
```

Result Table

Calls	ODBC	MySQL	SQL	Calls	ODBC	MySQL	SQL
1	6,922	4,063	17130	40	7,141	4,172	18261
2	6,765	3,953	17159	50	7,438	4,328	18551
3	6,860	3,922	17188	60	7,437	4,344	18841
4	6,687	4,000	17217	70	7,718	4,422	19131
5	6,922	4,062	17246	80	7,750	4,500	19421
6	6,844	4,031	17275	90	7,922	4,671	19711
7	6,703	4,016	17304	100	8,016	4,594	20001
8	6,750	4,078	17333	110	7,922	4,719	20291
9	6,875	4,125	17362	120	8,062	4,844	20581
10	6,891	4,031	17391	130	8,360	4,687	20871
11	6,859	3,985	17420	140	8,531	4,797	21161
12	6,922	4,000	17449	150	8,484	5,094	21451
13	6,906	4,032	17478	160	8,625	4,890	21741
14	7,000	3,906	17507	170	8,688	5,157	22031
15	6,985	4,094	17536	180	8,797	5,078	22321
16	6,968	4,078	17565	190	8,843	5,250	22611
17	7,016	4,015	17594	200	8,859	5,359	22901
18	7,000	4,032	17623	210	9,204	5,547	23191
19	6,937	4,140	17652	220	9,250	5,281	23481
20	6,954	4,157	17681	230	9,515	5,547	23771
30	7,156	4,094	17971	240	9,672	5,438	24061
				250	9,672	5,656	24351

The result of the measures can be seen in the Figure 3.12.
The dependency is linear between the increasing number of the function

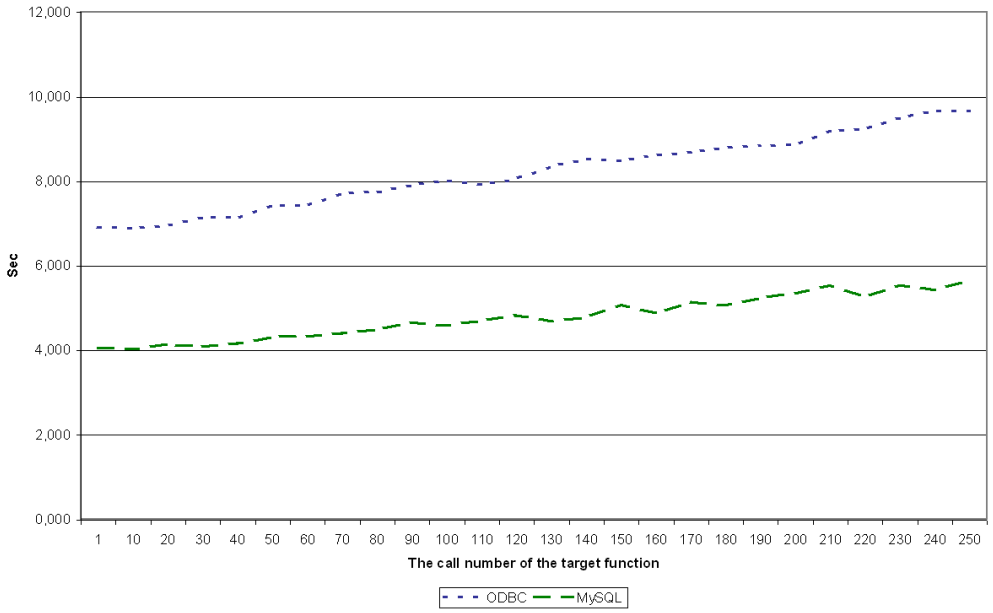


Fig. 3.12: Runtimes of the reorder function arguments refactoring by increasing the function call number with 250 functions in the files

calls and the runtime with approximately the same file size (the function number is constant). The gradient is small, the regression is close to the constant.

The two connection types give almost the same achievements, the regression lines are almost parallel.

It can be seen from this two measures, that increasing the function calls does not increase the runtime as much as the increasing of the functions. The runtime depends more on the precondition checks as on the performing the refactoring.

Effect the number of the implicit function calls from the code complexity with constant file size

We created 43 test file. All files contain 250 functions with one function call in each. The first one contains only one implicit function call of target function. In the other test files we increased the number of the implicit function calls (every function contains 0 or 1 implicit function calls), every

function calls referred to the first function, in which we would like to reorder the parameters. In the first 20 files we are increasing the number of the implicit function calls by 1, but after it by 10 until we reach the 500 lines limit.

In every test file the target function is the test/2 function. The refactoring modifies only the parameter order of the first function, but in every implicit function call it has to generate new codeparts.

We could not measure it with our original algorithm to give warnings when dynamic function calls are in the code (apply, hibernate, spawn). The refactorer returned with timeout error when all of the worst case files were in the database. The explain of the select showed, that there was 10042 row reading in the application table, and in each of the three subqueries 75675 row reading in the name table. The query was extremely slow, because it needs approximately 10^{18} row analysis.

With the new select query the number of the needed row readings are 37357. We succeeded to reduce to approximately 10^4 the row analysis.

Example code for testing (every file contains 250 functions, we only show the first three in this examples)

The shortest file

```
-module(test1).

-export([test/2]).

test(A, B) -> A + B, apply(fun test/2, [B, A]).

test_1(A, B) -> A + B, test(B, A).

test_2(A, B) -> A + B, test(B, A).
```

The 2nd file

```
-module(test2).

-export([test/2]).

test(A, B) -> A + B, apply(fun test/2, [B, A]).

test_1(A, B) -> A + B, apply(fun test/2, [B, A]).

test_2(A, B) -> A + B, apply(fun test/2, [B, A]).
```

The result of the first file

```
-module(test1).

-export([test/1]).

test(B, A) ->
    A + B,
    apply(fun (RefacVar_1, RefacVar_2) ->
                  test(RefacVar_2, RefacVar_1)
          end,
          [B, A]).

test_1(A, B) -> A + B, test(B, A).

test_2(A, B) -> A + B, test(B, A).
```

The length of the file is increasing after the execution of the reordering. The length of last result file is 1753 line instead of 500 line original.

Result Table

Implicit	ODBC	MySQL	SQL	Implicit	ODBC	MySQL	SQL
1	9,922	6,265	24496	40	12,469	7,547	30112
2	9,938	6,188	24640	50	13,266	7,875	31552
3	10,234	6,062	24784	60	13,828	8,703	32992
4	10,188	6,063	24928	70	14,468	8,859	34432
5	10,328	6,203	25072	80	15,094	9,266	35872
6	9,953	5,984	25216	90	15,735	9,609	37312
7	10,125	6,188	25360	100	16,203	10,156	38752
8	10,422	6,422	25504	110	18,000	10,329	40192
9	10,422	6,031	25648	120	17,734	10,921	41632
10	10,531	6,469	25792	130	18,438	11,469	43072
11	10,640	6,187	25936	140	19,171	11,828	44512
12	10,657	6,516	26080	150	19,719	12,235	45952
13	11,203	6,500	26224	160	19,984	12,375	47392
14	10,953	6,453	26368	170	21,031	12,890	48832
15	11,062	6,406	26512	180	22,250	13,328	50272
16	10,938	6,516	26656	190	23,907	13,766	51712
17	11,062	6,484	26800	200	23,250	14,094	53152
18	11,094	6,578	26944	210	24,218	14,640	54592
19	11,110	6,672	27088	220	24,782	15,813	56032
20	11,218	6,735	27232	230	25,485	15,531	57472
30	11,797	7,031	28672	240	25,672	15,860	58912
				250	26,953	16,234	60349

The result of the measures can be seen in the Figure 3.13.
The relation is linear between the increasing number of the implicit func-

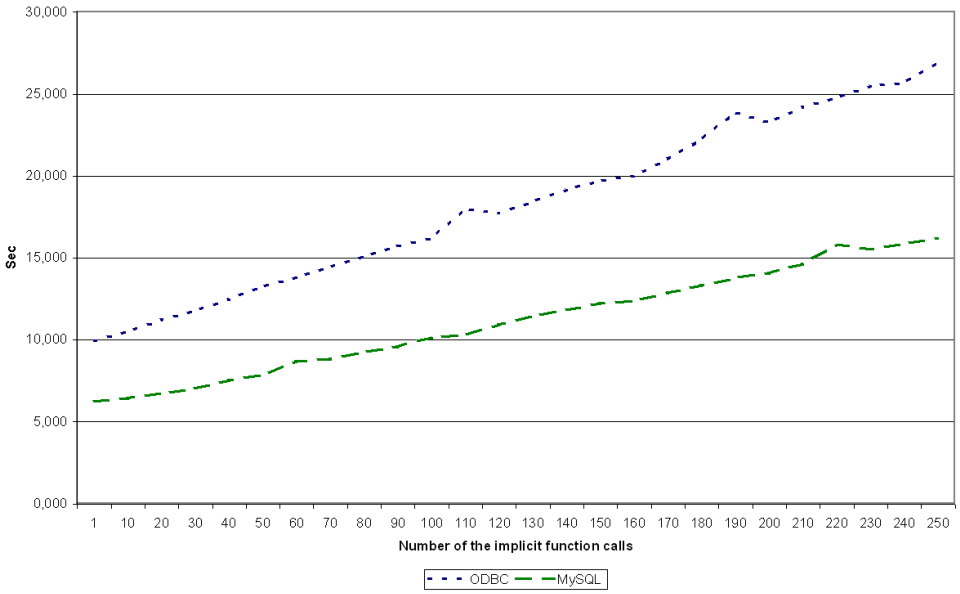


Fig. 3.13: Runtimes of the reorder function arguments refactoring by increasing the implicit function call number with 250 function calls in the files

tion calls and the runtime. Generating new codeparts (deleting and creating subtrees), needs much more resources as the traditional refactorings which contains only precondition checks and simple performing with small database modifications.

The MySQL connection give better achievement with smaller gradient.

3.5.2 Experimental analysis of the eliminate variable refactoring

In the eliminate variable refactoring the changeable code parts is just the current function, where the variable is defined (inside the scope of the variable). We created the test files to show the best and the worst cases by increasing the occurrences of the variable and in the other side the difficulty of the expression.

Effect of the occurrences' number in the best case

In this test series we used the same `test/1` function. The target variable is `X`. This examples mean the best case for this refactoring, because the

variable and all of it occurrences can be deleted; and the expression is just one constant (an integer).

In the series we increased the number of the occurrences of the variable. We made 40 test files. In the first ten files we added one occurrence at once, but after the 20th files we added 10 new occurrences into the function.

The first (shortest) file

```
-module(test1).

-export([test/1]).

test(A) -> X = 3, A.
```

The 10th file

```
-module(test10).

-export([test/1]).

test(A) -> X = 3, X, X, X, X, X, X, X, X, X, A.
```

The result will be the same for every file. The * in the module represents all of the test files.

The result file

```
-module(test*).

-export([test/1]).

test(A) -> A.
```

Result Table

Occ	ODBC	MySQL	SQL	Occ	ODBC	MySQL	SQL
1	0,250	0,141	182	30	0,453	0,281	849
2	0,187	0,125	205	40	0,609	0,359	1079
3	0,204	0,156	228	50	0,750	0,469	1309
4	0,203	0,172	251	60	0,860	0,547	1539
5	0,187	0,140	274	70	1,109	0,687	1769
6	0,266	0,204	297	80	1,141	0,829	1999
7	0,203	0,218	320	90	1,281	0,890	2229
8	0,203	0,157	343	100	1,391	0,985	2459
9	0,203	0,156	366	110	2,656	1,125	2689
10	0,235	0,156	389	120	1,844	1,234	2919
11	0,218	0,172	412	130	2,031	2,187	3149
12	0,250	0,156	435	140	2,187	1,563	3379
13	0,235	0,203	458	150	3,016	2,234	3609
14	0,250	0,391	481	160	4,422	2,360	3839
15	0,281	0,187	504	170	2,828	2,672	4069
16	0,282	0,188	527	180	4,672	3,531	4299
17	0,312	0,203	550	190	4,391	3,625	4529
18	0,328	0,234	573	200	4,875	3,078	4759
19	0,313	0,203	596	210	5,109	4,609	4989
20	0,328	0,235	619	220	6,062	4,297	5219

The result of the measures can be seen in the Figure 3.14.

The result is almost linear with a lot of fluctuation. When the table size, which is needed for the modifications, reaches a limit, it will be too big for the cache, and these jumps can cause a part of the fluctuations.

There is no significant difference between the MySQL and ODBC con-

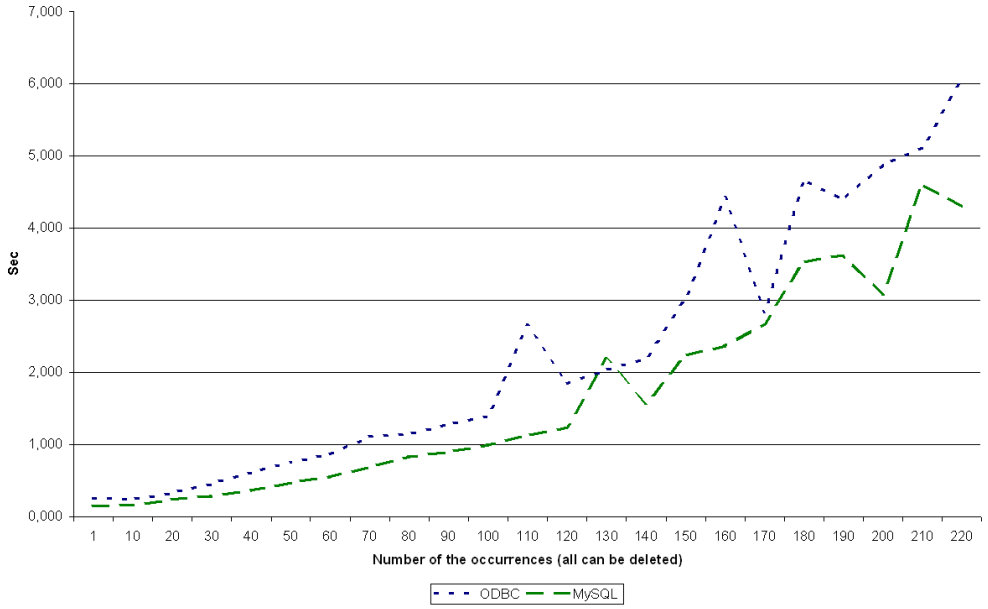


Fig. 3.14: Runtimes of the eliminate variable refactoring by increasing the occurrences of the variable

nections' achievements.

Effect of the expression length

In this test series we used the same `test/1` function. The target variable is `X`. This examples mean the worst case for this refactoring, because the variable and all of it occurrences need to exchange to the expression, which means more expression replications; and the expression is increasing.

In the series we increased the length of the expression of the eliminate variable. We made 20 test files. In every test case we have 220 occurrences of the target variable as the parameter of the function calls. In every occurrence the expression will be the new parameter. We increasing the length of expression in every case by plus 1 addition.

In the following code examples we reduced the number of the function calls to 10 from 220.

The first file

```
-module(test1).

-export([test/1]).

test(A) -> f(X = 1), f(X), f(X), f(X), f(X),
           f(X), f(X), f(X), f(X), f(X), A.

f(X) ->
    X+1.
```

The last file

```
-module(test20).

-export([test/1]).

test(A) ->
  f(X = 1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20),
  f(X), f(X), f(X), f(X), f(X), f(X), f(X), f(X), f(X), A.

f(X) ->
  X+1.
```

The result of the first file

```
-module(test1).

-export([test/1]).

test(A) -> f(1), f(1), f(1), f(1), f(1), f(1),
           f(1), f(1), f(1), f(1), A.

f(X) ->
    X+1.
```

Result Table

Expr elem.	ODBC	MySQL	SQL
1	5,110	3,156	9446
2	10,281	5,750	21384
3	14,406	8,625	33323
4	19,406	11,360	45262
5	24,594	14,093	57201
6	28,563	17,032	69140
7	33,234	19,234	81079
8	37,672	22,922	93018
9	42,891	24,391	104957
10	47,656	28,171	116896
11	52,204	30,125	128835
12	57,078	33,172	140774
13	61,015	35,953	152713
14	66,032	37,579	164652
15	71,140	41,484	176591
16	75,453	44,937	188530
17	80,563	47,532	200469
18	85,172	49,422	212408
19	89,718	53,218	224347
20	95,813	54,812	236286

The result of the measures can be seen in the Figure 3.15.

The result shows a really nice linear structure with quite big gradient. The big gradient is due to the growing number of the replicated subtrees. In the last test file the length of the expression is 20 elements, which is at least 5 level high subtree. And we replicated every expression 220 times.

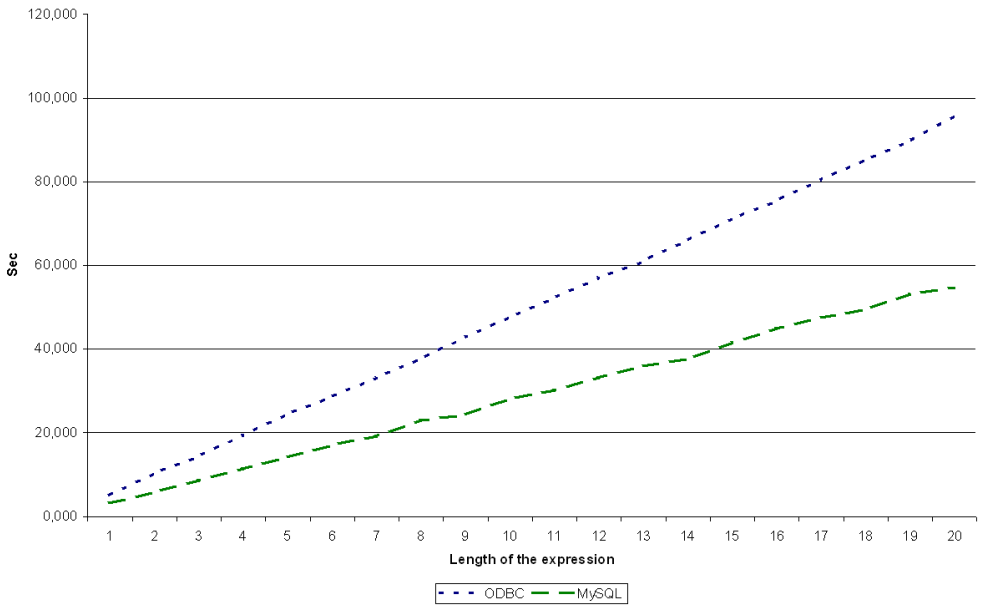


Fig. 3.15: Runtimes of the eliminate variable refactoring by increasing the length of the expression

The achievement of the MySQL connection is far better than the other.

The runtimes are not acceptable, but this test files represents the worst case of this refactorings, and the example has no reality. Nobody will use the same variable more than 200 times in the same function.

Effect of the increasing number of the remaining occurrences

In the series we increased the number of the remaining occurrences of the variable by the same occurrence number (220). We made 40 test files. In the first 20 files we added one remaining occurrence (the variable is the parameter of a function) at once, but after the 20th files we added 10 new remaining occurrences into the function.

In the following code examples we reduced the number of the function calls to 10 from 220.

The first file

```
-module(test1).

-export([test/1]).

test(A) -> f(X = 1), X, X, X, X, X, X, X, X, X, A.

f(X) ->
    X+1.
```

The 5th file

```
-module(test5).

-export([test/1]).

test(A) -> f(X = 1), f(X), f(X), f(X), f(X), X, X, X, X, X, A.

f(X) ->
    X+1.
```

The result of the 5th file

```
-module(test1).

-export([test/1]).

test(A) -> f(1), f(1), f(1), f(1), f(1), A.

f(X) ->
    X+1.
```

Result Table

Calls	ODBC	MySQL	SQL	Calls	ODBC	MySQL	SQL
1	3,796	2,781	3971	30	6,609	3,844	4696
2	4,532	3,625	3996	40	4,516	4,047	4946
3	5,922	4,078	4021	50	4,813	3,609	5196
4	5,296	3,594	4046	60	5,828	4,688	5446
5	5,594	4,062	4071	70	4,656	3,547	5696
6	5,672	6,235	4096	80	5,625	3,406	5946
7	6,266	4,437	4121	90	4,844	4,250	6196
8	5,531	6,828	4146	100	4,562	3,406	6446
9	5,406	5,485	4171	110	5,313	3,766	6696
10	5,828	4,062	4196	120	5,468	3,797	6946
11	5,422	5,703	4221	130	4,516	3,547	7196
12	4,766	4,954	4246	140	5,391	4,843	7446
13	6,594	4,265	4271	150	4,625	2,828	7696
14	4,984	4,125	4296	160	4,688	3,532	7946
15	6,187	4,906	4321	170	5,109	3,468	8196
16	4,500	4,469	4346	180	4,922	3,188	8446
17	6,235	4,313	4371	190	5,375	3,828	8696
18	4,390	4,234	4396	200	5,688	3,063	8946
19	6,969	4,047	4421	210	5,296	4,375	9196
20	4,422	4,609	4446	220	5,454	3,343	9446

The result of the measures can be seen in the Figure 3.16.

The relation is not exact, it is better to say, that there is no relation between the number of the remaining occurrences of a variable and the runtime, if the expression is small. The reason is that the cost of deleting an occurrence, or replace it with a constant is almost the same. The result could be

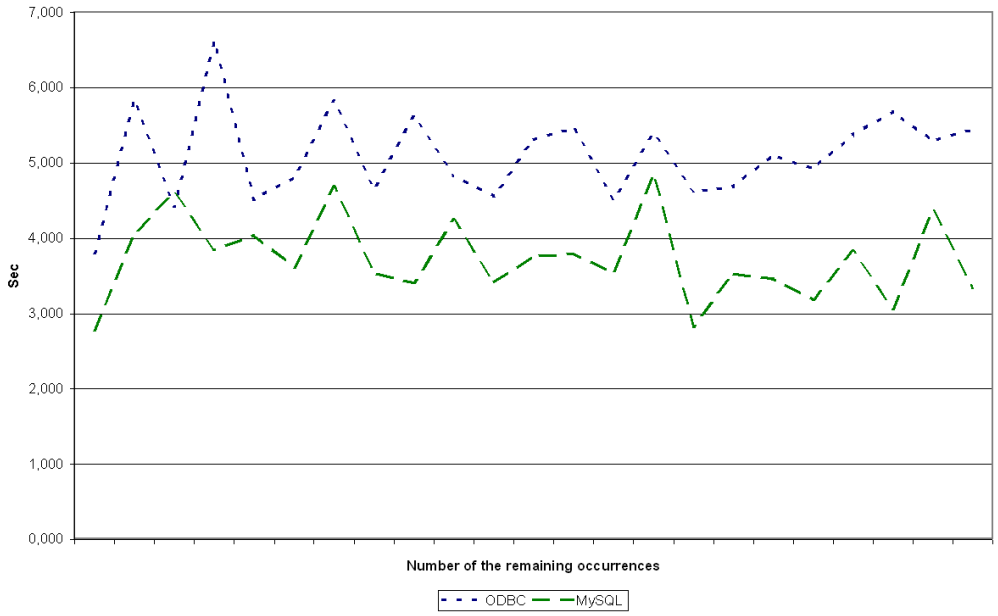


Fig. 3.16: Runtimes of the eliminate variable refactoring by increasing the number of the remaining occurrences from the 220

different if the expression would be longer (the effect of the expression length is linear with quite a big gradient as it is in shown in the 3.15). In that case the result would be linear with a lot of fluctuation.

3.6 Main results

The dependency of the refactoring's execution time was linear proportional to both the file size and the code complexity.

In the reorder function argument refactoring the main effects were the following:

- The dependency is linear between the length of the file and the runtime.
- Increasing the function calls does not increase the runtime as much as the increasing of the functions. The runtime depends more on the precondition checks as on the performing the refactoring.
- The relation is linear between the increasing number of the implicit function calls and the runtime. Generating new codeparts (deleting

and creating subtrees), needs much more resources as the traditional refactorings which contains only precondition checks and simple performing with small database modifications. The MySQL connection give better achievement with smaller gradient.

In the eliminate variable refactoring the main effects were the following:

- The increasing the length of the expression causes a really nice linear structure with quite big gradient. The big gradient is due to the growing number of the replicated subtrees. The achievement of the MySQL connection is far better than the odbc.
- There is no relation between the number of the remaining occurrences of a variable and the runtime, if the expression is small. The reason is that the cost of deleting an occurrence, or replace it with a constant is almost the same. The result could be different if the expression would be longer
- The effect of the increasing number of the variable occurrences is almost linear with a lot of fluctuation.

The refactorer suspends the most of the time: it can be various between 50-90%. The suspends come from the suspend of the sql queries and even from the io procedures. The main overhead of the tool come from the database connections and suspends. In the future this part should be improved by searching an efficient database (for example Mnesia).

The main part of the runtime is executing sql queries. The refactorer spent 70-80 % of the whole time with executing sql queries. The result in different commands:

- In the storing and recovering the source it is 50-60%. In this commands we are using really big non database dependent codeparts (we have to parse and traverse the original code to build up the abstract syntax tree, which we can store in the database.)
- In the eliminate variable refactoring it is 70-80%.
- In the reorder function parameters refactoring it is approximately 60-65%.

If we want to make our tool reasonably fast, we had to find the optimal solution to handle these calls. The tool was originally written by ODBC connection, but we added a possible alternative by choosing native mysql connection. The mysql connection is faster, but much more specific, than the ODBC connection.

The slowest query type is the `commit`. If we are calling `commit` after every query, then the connection opening and closing will appear two times for every query. We minimised the number of the `commits` with switching off the `autocommit` option, and call commit from the tool, where it was needed.

We should use as complex queries as we can instead of many small queries to reduce the connection and disconnection times. We get a new problem with using complex queries: are the new selects optimised? If it is possible *ref* and *eq_ref* select types should be used even in the joined and embedded queries.

The database structure is optimised to the problem, it even has some tables to improve the efficiency. In the *speed and efficiency* category of the tables sometime contain redundant information, what can be found in the other tables, but here they are collected for a different reason and into a different structure. The gain on the efficiency with using these tables makes it worth to store the redundant information, as it is detailed in each table description.

We had to write a new algorithm for the function call storing because the old one was very slow, when more than one module were already in the database. After there was a big module put into the database, for every new module it took almost the same time to put the code into the database as the big ones. This resulted some unacceptable running times. The old one was almost constant four time slower for every module as the average value of the new version.

SUMMARY

In this thesis we presented the description of the individual refactorings and the refactorer.

At first we gave a short overview of the refactoring in general, and the already existing refactor tools for objected oriented languages (Java, C#) [10, 11], and some for functional languages (Haskell, Clean) [6, 18]. We collected the specialities of refactoring Erlang programs and some possible refactorings. We explained the structure of the database based on the node types of the AST.

After summarising the rules of the Erlang language, especially for the functions and variables, we analysed the five refactorings (preconditions, execution orders, current limitations).

In the second chapter we wrote about the realization of the problem through the most difficult point with example code sorted by problem types: storing and recovering the code; using SQL queries and list operation (mainly from the lists Erlang module); handling of branches; deleting and creating code parts (according to the AST) directly in the database. The two main parts of the implementation period are:

- storing the source code (static and semantic information) into the database and recover it;
- providing a safe refactoring with comprihensive error messages.

We also presented the algorithm for the two chosen refactorings using flowchart diagrams.

In the last part we made the efficiency analysis. We built it up from the basic elements (sql queries) of the tool to the whole refactorings.

Related work

Modern software development systems like Visual Studio,and Eclipse [10], tend to have support for refactoring. Microsoft Visual Studio introduced refactoring support for C# in the latest release, and in Eclipse it is possible to use a wide range of refactorings from renaming a variable to restructuring the whole code base, even to allow the user to write their own refactorings. It is straightforward that there are common things in every programming language, like variables and functions, which would allow the same refactorings for every language(such as renaming a variable, or a function). These refactorings have the same meaning, but because every language has a different definition of its elements, the preconditions and effect vary. For an example lets consider the different definitions of the variable in the OO and the functional languages. However, the two in detail introduced refactorings are so special: they are focused on the Erlang language, there's no such tool available at the moment that would support the same refactorings in an other way. Using the AAST approach [17], the University of Kent team implemented various refactorings, but these steps are not common in the tools. Furthermore the database approach was only used in the Clean (ref) project, which we extended and developed further.

There is only one released refactoring tool for functional languages, namely for Haskell: HaRe [18, 3]. This tool uses the AST to do the refactorings, and the source code token stream to preserve the layout. Both HaRe and the AAST approach of the Erlang refactoring has been developed at the University of Kent.

Own results

We successfully planned, designed, implemented and tested the database structure and the five refactorings.

The main difficulty was in the planning of the table structure to maintain the storing of both the static and semantic information. And, in the same time, to keep the effectiveness of the searching and queries (we presented this part of the work in our bachelor thesis [19]).

Before the implementation we chose the components of the environment, for details see 1.1.3. We tried to find the best combinations of the usable algorithms (traversals, collector processes, already written data structure

manipulating functions in Erlang) to obtain the fastest and most effective code.

At the designing of the refactoring module we had to cover all the statically analisable possibilities of the language, all precondition given by the previous analysis of the other project members.

The source code for these refactor tool is more than 100 inserted ELisp lines to distel.el and 15000 Erlang lines (counting in the documentation comments): 10-12000 lines are the general part, usable in every other refactoring (storing the source into the database, recovering from it, reusable queries, the database version of the needed syntax_tools functions); the refactoring are each 300-500 lines. (the checking of the preconditions and the execution or throwing error messages).

After the implementation we tested the tool with more than 200 already written test cases, and we wrote and tested some own test cases to cover the missing parts, for example the cases of the implicit function (3.5.1).

We made an efficiency analysis starting with the SQL queries, as the basic building blocks of the tool. We restructured the SQL queries to maintain as complex, but effective queries as we can to reduce the cost of the connection buildings. We found and rewrote the most ineffective selects, what we found during the explain plan analysis (3.1.1).

We measured the runtime of the different code parts. From the traces it was clear, that the main part of the runtime is the executing of sql queries. We wrote the tool originally with the ODBC connection. We found it slow, so we searched for an other possibly connection. We came accross the native mysql module, and added it into the tool as a possible connection. The comparison of the two connection can be seen in 3.5 table. We found some really inefficient algorithm parts for the storing of the function calls and we rewrite it to a most efficient one.

We made experimental analysis for the two chosen refactorings. We measured the tool with best case and worst case inputs. We separated the effects of the file size an the code complexity. It was obvious from the measures that the runtime depends lineary on these, and the precondition checks have significantly higher percentage in runtime than the the performing of the refactoring.

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN-0201-48567-2.

[2] Martin Fowler's refactoring site. http://www.refactoring.com/. 2007.06.01.

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

[4] GNU Emacs homepage. http://www.gnu.org/software/emacs/. 2006.06.01.

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

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

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

[8] J. Armstrong, R. Virding, M. Williams, and C. Wikstrom. *Concurrent Programming in Erlang*. Prentice Hall, 1996. ISBN-0135-08301-X.

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

[10] Eclipse Project homepage. http://www.eclipse.org/. 2007.06.01.

[11] C# Refactory homepage. http://www.xtreme-simplicity.net/. 2007.06.01.

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

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

[14] Distel: Distributed Emacs Lisp. http://fresh.homeunix.net/~luke/distel/. 2007.06.01.

[15] MySQL homepage. http://www.mysql.com/. 2007.06.01.

[16] Erlang homepage. http://www.erlang.org/. 2007.06.01.

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

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

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