Erlang kód tárolása adatbázisban

Szakdolgozat

Nagy Tamás
e-mail: {n_tamas}@inf.elte.hu

Társszerző: Víg Anikó
e-mail: {viganiko}@inf.elte.hu
Konzulens: Horváth Zoltán
Programozási Nyelvek és Fordítóprogramok Tanszék, ELTE

2006. december 30.

Erlang kód tárolása adatbázisban

Szakdolgozat

Víg Anikó
e-mail: {viganiko}@inf.elte.hu

Társszerző: Nagy Tamás
e-mail: {n_tamas}@inf.elte.hu
Konzulens: Horváth Zoltán
Programozási Nyelvek és Fordítóprogramok Tanszék, ELTE

2006. december 30.

Storing Erlang source code in database

Thesis

Tamás Nagy and Anikó Víg
e-mail: {n_tamas viganiko}@inf.elte.hu

Supervisor: Zoltán Horváth
Department of Programming Languages and Compilers, ELTE

2006. december 19.

CONTENTS

Introduction 6

1. *Source and database (Tamás Nagy)* 7
   - 1.1 Refactoring using a database 7
   - 1.2 AST and database 7
   - 1.3 The database representation 8
   - 1.4 Semantical information storing in the database approach 9

2. *User Manual (Anikó Víg)* 13
   - 2.1 Installation guide for Windows 13
     - 2.1.1 MySQL: 13
     - 2.1.2 MySQL/ODBC_connector: 13
     - 2.1.3 Erlang compiler: 14
     - 2.1.4 Emacs: 14
     - 2.1.5 Cygwin: 14
     - 2.1.6 Distel: 15
     - 2.1.7 Source files of the rector tool: 15
   - 2.2 Installation guide for Linux 15
   - 2.3 Minimum requirements 15
     - 2.3.1 Hardware 15
     - 2.3.2 Software 15
   - 2.4 Running the tool 16
     - 2.4.1 Starting the applications 16
     - 2.4.2 Initialize the database 16
     - 2.4.3 Loading the source into the database and recover it . . . 18

3. *Development Manual* 20
   - 3.1 System architecture (Anikó Víg) 20
     - 3.1.1 Design principles for the user interface 20
     - 3.1.2 The structure of the tool 20
   - 3.2 Module d_client (Tamás Nagy) 22
     - 3.2.1 Module description 22
     - 3.2.2 Functions 23
   - 3.3 Module db_init (Tamás Nagy) 25
     - 3.3.1 Module description 25
     - 3.3.2 Functions 25

3.4 Module into_db (Anikó Víg) 28
3.4.1 Module description 28
3.4.2 Functions 28
3.5 Module out_from_db (Tamás Nagy) 71
3.5.1 Module description 71
3.5.2 Functions 71
3.6 Module refactor (Anikó Víg) 79
3.6.1 Module description 79
3.6.2 Functions 80
3.7 Testing procedures and result 86

4. Appendix (Anikó Víg) 94
4.1 Denied names 95
4.1.1 Auto-imported BIFs 95
4.1.2 RESERVED WORDS 99
4.2 Structure of the database 99
4.2.1 Database tables 99

List of Figures 103

List of Tables 104

Bibliography 105

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 extend 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 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 id to store the syntactical and semantical information. The main difference of the two implementations is the editor: the Clean refactorer has a syntactical based editor.However, we are using Emacs because it seems most Erlang programmers use this as their editor of choice.

This thesis is part of this bigger project to create an Erlang refactor tool. We have to store the Erlang source code, which is displayed in Emacs, using an Emacs menu point into a SQL database. We have to store not only the syntactic, but also some semantic information. We have to recover the source from the database and display in Emacs.

Aniko Vig's role is the implementation of the storing process (`into_db.erl`, `refactor.erl` modules). Tamas Nagy's role is the recovering part, the modification of the already existing modules and the connection between the Erlang node and Emacs through Distel (modules `d_client.erl`, `db_init.erl`, `out_from_db.erl`; modifications in `epp_dodger.erl`, `erl_recomment.erl`, `erl_scan.erl` and `distel.el` files). In the contents the author of the current part is marked in paranthesis.

In the first chapter we give an overview of the used database structure. In the next two part we present the User and the Development Manual, including a description of the testing method. In the appendix we show the tables of the database and the denied names for variables or functions.

1. SOURCE AND DATABASE

1.1 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.

1.2 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 with ODBC connection 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 handle joining tables together.

$\texttt{gcd}_{30}(\texttt{N}_{15},\texttt{M}_{16})$ 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.

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 [14].

1.3 The database representation

In the relational database representation, there are two kind of tables: tables that store the AST, and tables that store semantical information. 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 1.4. 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 corresponding 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.

1.4 Semantical information storing in the database approach

The source code and the ASTs of the module used as an example in figure 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.

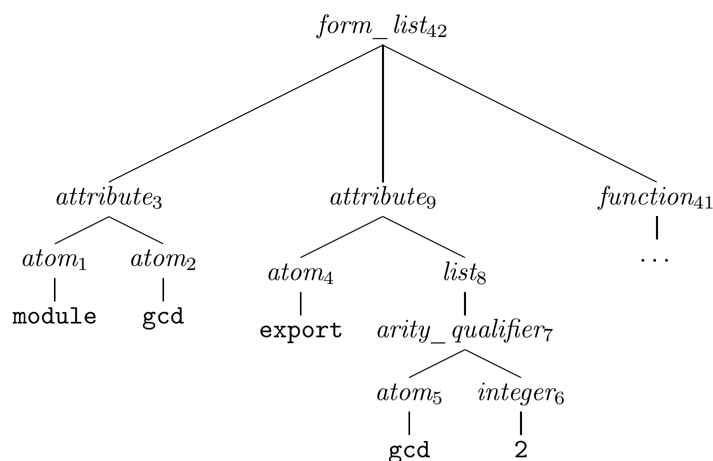

Fig. 1.3: The AST of `gcd` (Part 1)

Semantical information about Erlang programs are stored in tables such as “var_visib”, “fun_visib”, "scope", "scope_visib" and “fun_def”. The table “var_visib” stores visibility information on variables, namely which occurrences of a variable name identify the same variable. This table has two columns:

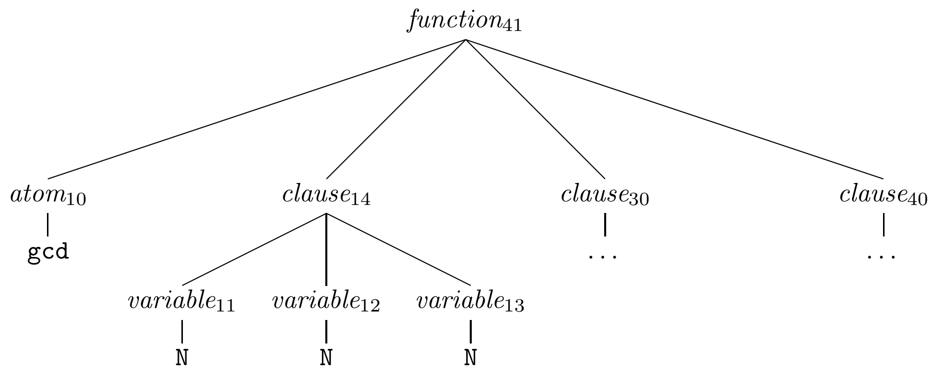

Fig. 1.4: The AST of `gcd` (Part 2)

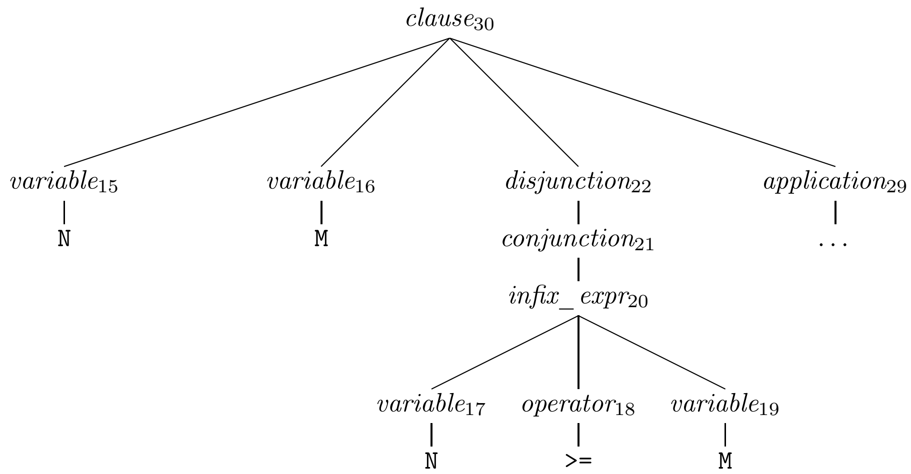

Fig. 1.5: The AST of `gcd` (Part 3)

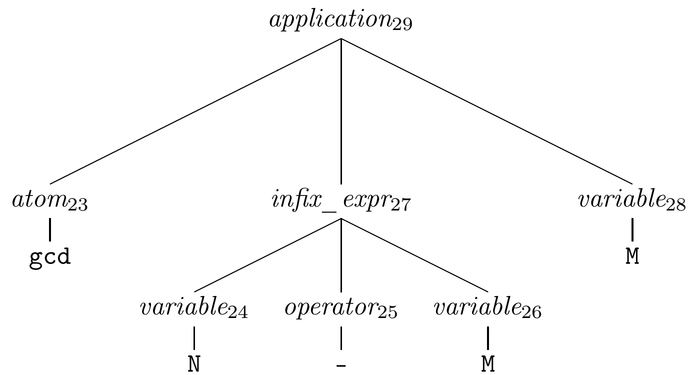

Fig. 1.6: The AST of `gcd` (Part 4)

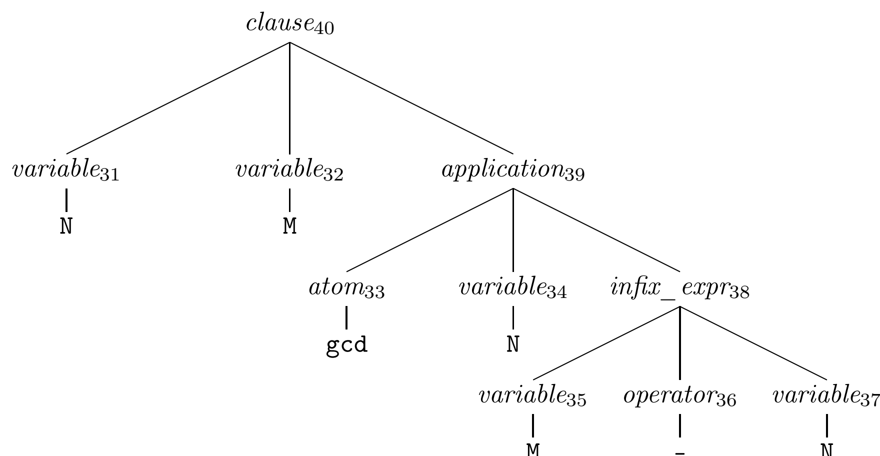

Fig. 1.7: The AST of `gcd` (Part 5)

"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.

Scope of a variable: The scope of a variable is always limited to a function clause. Variables are created by the pattern matching mechanism; the scope of a variable begins at the corresponding pattern match, and extends to the end of the innermost enclosing function clause or list comprehension expression. Every expression have to be 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.

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.

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 see in 4.1.

2. USER MANUAL

2.1 Installation guide for Windows

There are seven main component of the tool which need to install.

2.1.1 MySQL:

1. Download the file mysql-5.0.18-win32.zip free from http://www.mysql.com/
2. Start the installation.

   Choose custom mode, then next.

   After the installation the following configuration steps needed (it start automatically when choosing at the last step of the installation tool):

   - server-type:Developer Machine
   - database usage: multifunctional or transactional
   - concurrent connections: manual settings: 10
   - default character set
   - standard character set
3. (Optional) If you have enough memory you can make the following step to fasten the database:

   Write innodb_buffer_pool_size = 128M to my.ini file.
4. Start MySQL and create the "parse" database.

2.1.2 MySQL/ODBC_connector:

1. Download the file mysql-connector-odbc-3.51.12-win32.msi free from http://www.mysql.com/.
2. Start the installation Choose custom mode.
3. In the Start menu of Windows open the control panel, choose the administrative tools and inside it open the Data Sources (ODBC)

   Choose the Add... button and click double on MySQL ODBC... from the list.

   In the opening window fill out the following text-boxes:

- Data source name : Erlang(for example)
- User
- Password
- Database: parse

2.1.3 Erlang compiler:

1. Download the file otp_win32_R10B-9.exe free from http://www.erlang.org/. The installation needs 105 MB free disk space.
2. Start the installation, choose or create a directory-name which does not contain space: c:\erl5_4_15 for example
3. After the installation right click on the icon of Erlang and open the properties:

   Set Target: ...\werl.exe and add in the same line the followings:

   - sname optional_something +R9
   - pz c:/distel/share/distel/ebin (and after a space) dictionary for refactor modules (it is similar to add path you don't need to made it by hand)

2.1.4 Emacs:

1. Copy the source to an arbitrary directory (for example c:\emacs). The source can be downloaded from http://www.gnu.org/software/emacs/.
2. Run (c:\emacs)\bin\addpm.exe
3. Start emacs choose the Save options from Options menu this will generate your .emacs file
4. See the example: _emacs file, copy the marked parts into your own .emacs file
5. Copy the .erlang.cookie file from the user's default directory (usually c:\Do-cuments and Settings\Username) to the same place, where your .emacs file is.

2.1.5 Cygwin:

1. Download the file cygwin.zip free from http://cygwin.org/.
2. Download the Cygwin installer it is quite small, and after it you can choose the server and the packages which you want, we need at least this three package above the default:

   - Devel/gcc-core
   - Devel/gcc-g++
   - Devel/make

2.1.6 Distel:

1. Download distel.zip free from http://fresh.homeunix.net/˜luke/distel/.
2. Before the installation add the following directories to the User's PATH environment variable via the System control panel:
   - c:\erl5_4_15\bin
   - c:\emacs\bin
   - c:\cygwin\bin
   - c:\distel\bin (after installation of Distel)
3. Choose a directory to install distel, e.g. C:\distel, and do: ./configure –prefix=c:/distel
4. Do a "make" and then a "make install".

2.1.7 Source files of the rector tool:

1. Download the file refactor.zip free from the repository.
2. Copy the files to your dictionary for refactor modules.
3. Write over the distel.el file in your ...distel\share\emacs\site-lisp\distel directory.

2.2 Installation guide for Linux

The components and the methods are similar as the installation in Windows system, just the cygwin component does not needed.

During the testing period the tool worked really slow in Linux system, because the ODBC connection was not effective.

2.3 Minimum requirements

2.3.1 Hardware

... GB free disk space (including all the components)
128 MB memory
Others:............

2.3.2 Software

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

2.4 Running the tool

The user has to start the application and initialize the database before he start to put the source code into the database.

2.4.1 Starting the applications

At first the user has to start an Erlang node by running Erlang. The user has to start Emacs and open the .erl file, which he want to put into the database. If an Erlang source code is open, an Erlang menu appears in Emacs as shown in the Figure 2.1.

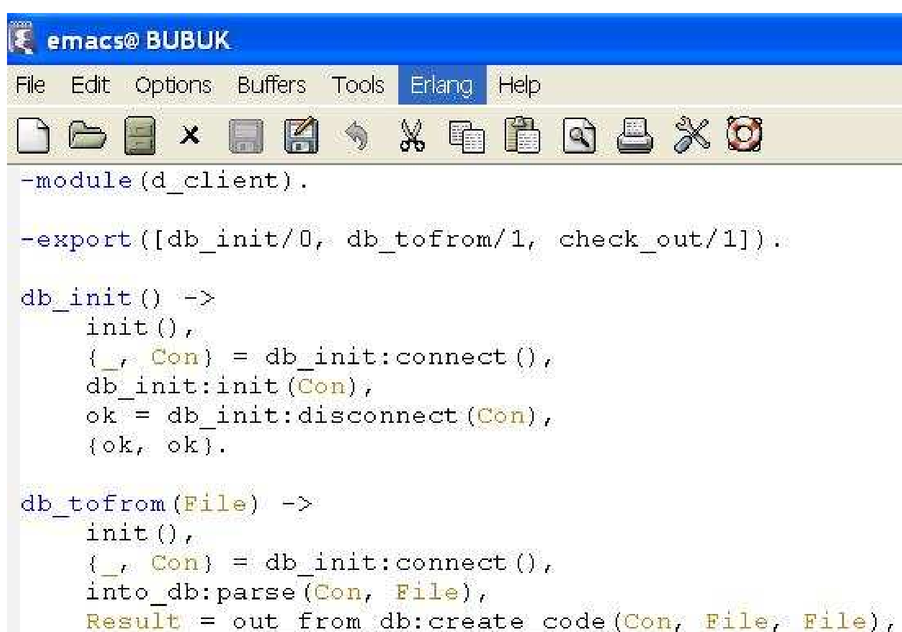

Fig. 2.1: The Erlang menu in Emacs

2.4.2 Initialize the database

The user has to choose the *Initialize database* command from the Erlang/Refactor menupoint as shown in the Figure 2.2.

The program will ask for the name of the Erlang node down in the minibar. The name can be copied from the running Erlang as shown in the Figure 2.3.

When the initialization is ready the following message appears in the minibar: *Initialized*.

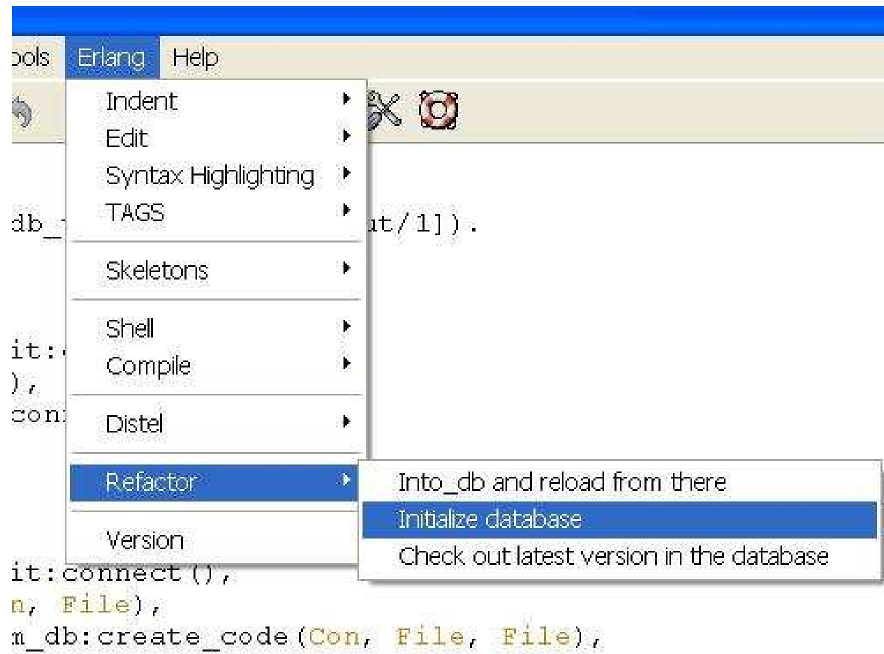

Fig. 2.2: Initialize the database

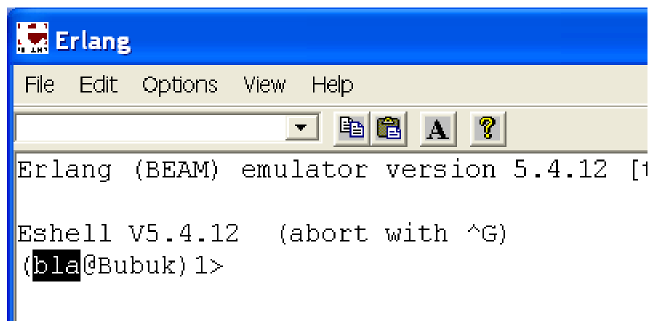

Fig. 2.3: Getting the name of the Erlang node during the initialization

2.4.3 Loading the source into the database and recover it

After the user opened the .erl file, which he want to put into the database, he has to choose the *Into_ db and reload from there* command from the Erlang/Refactor menupoint as shown in the Figure 2.4. When the initialization is successfully done, the following message appears in the minibar: *Reloaded*. In the same time the program displays the pretty-printed version of the source code in the same window. The current file/module remain in the database until the next initialization. If the user put the newer version of the same file into the database, the previous version will be deleted, and the new file will be parsed. If the user wants to put more than one file into the database, he can open them and puts them into the database in the same way. The database will contain and handle all files until the next initialization.

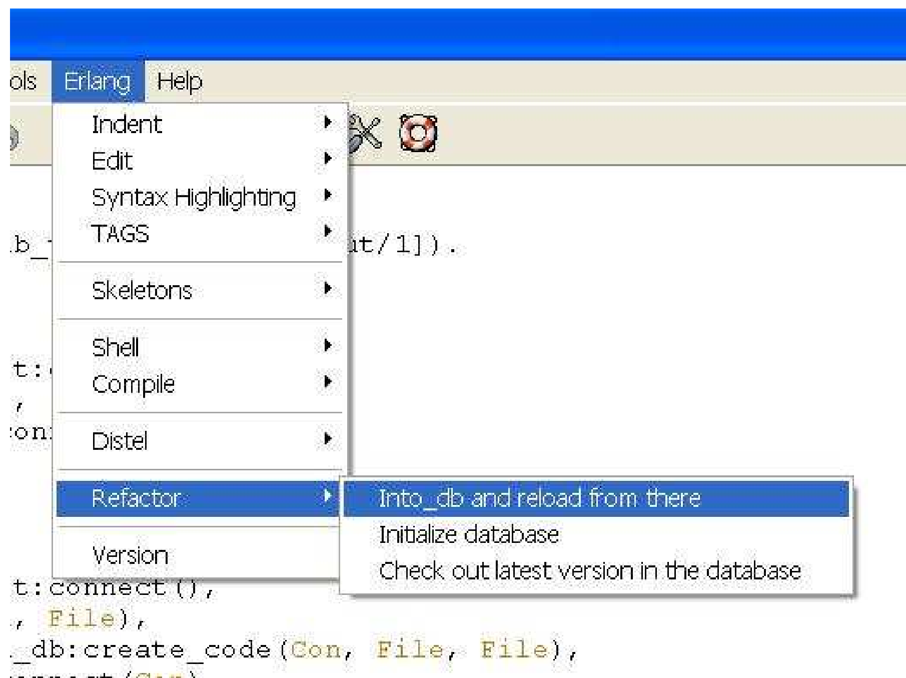

Fig. 2.4: Into_db and reload from there

If the user once has putted the source into the database at any time he can ask for the latest version of the source by choosing the *Check out latest version in the database* command from the Erlang/Refactor menupoint as shown in the Figure 2.5. When the initialization is successfully done, the following message appears in the minibar: *Reloaded*.

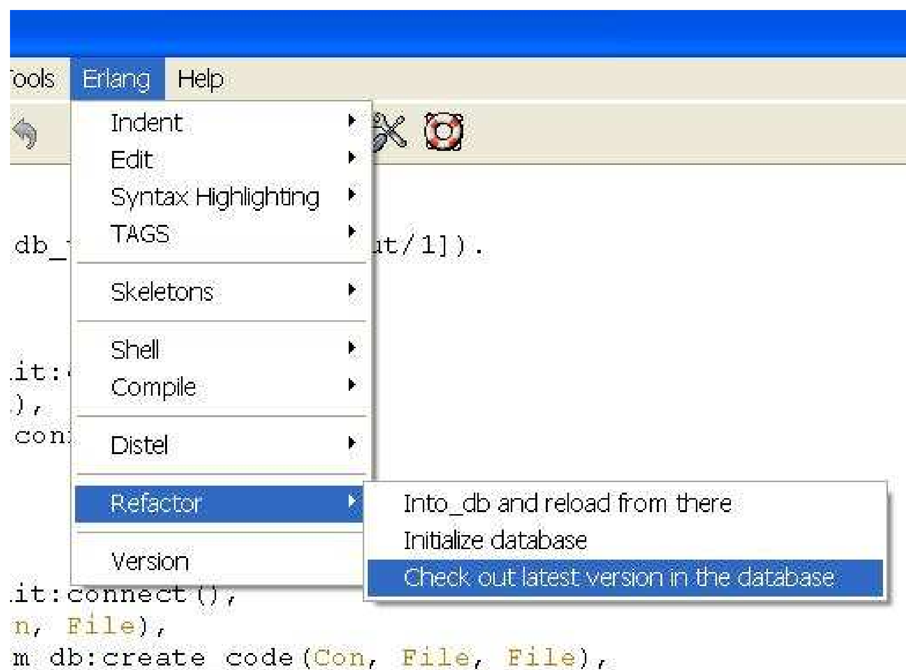

Fig. 2.5: Check out latest version in the database

3. DEVELOPMENT MANUAL

3.1 System architecture

3.1.1 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.

3.1.2 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** [11]:

   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** [13]:

   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** [12] (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.

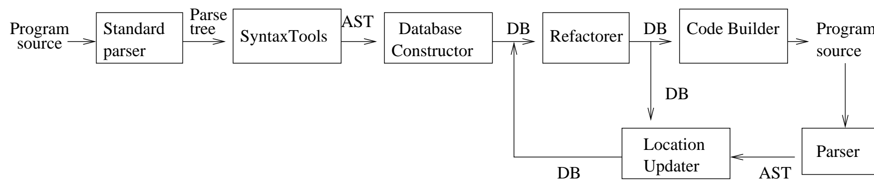

Fig. 3.1: The Implementation Architecture

Figure 3.1 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 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 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.

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`.

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).

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

3.2 Module d_client

3.2.1 Module description

This module composes an interface of the refactor tool to the Distel. Every function is exported in the module except the init/0 and stop/0 functions.

3.2.2 Functions

Exported functions

db_tofrom/1

Description The function stores the parameter file into the database and rebuild it again from the database.

The parameter:

1. File: the erlang source file, the module which the user want to store to the database.

This function is called from Distel in ELisp language. The user can call through the Into_db and reload from the database menu point in Emacs.

Example:
```
erl-send-rpc node
  'd_client
  'db_tofrom
  (list name)
```

Implementation The function stores the source code to the database and recovers it.

- At first the function calls the init/0 function.
- After it builds up the ODBC connection and gets the identifier of the connection with db_init:connect/0 function which returns a tuple with two element. The identifier of the connection is the second element of the returned tuple.
- After it the function parses the current file into the database by calling into_db:parse/2 with the reference of the connection and the path of the current file as parameters.
- After it the function creates the code from the database by calling out_from_db:create_code/3 function. The result of the called function is stored into a variable.
- After it closes the connection by db_init:disconnect/1 function with the identifier of the connection as parameter.
- At the end the function returns with the result of the code creating.

db_init/0

Description The function builds up the connection to the database and creates an empty database.

The function has no parameter.

This function is called from Distel in ELisp language. The user can call through the Initialise database menu point in Emacs.

Example:
```
erl-send-rpc node
    'd_client
    'db_init
    (list )
```

Implementation

- At first the function calls the init/0 function.
- After it builds up the ODBC connection and gets the identifier of the connection with db_init:connect/0 function which returns a tuple with two element. The identifier of the connection is the second element of the returned tuple.
- After it the function creates an empty database by calling db_init/1 function with the identifier of the connection.
- After it closes the connection by db_init:disconnect/1 function with the identifier of the connection as parameter.
- At the end the function returns with a tuple: ok, ok.

Local functions

init/0

Description The function starts the ODBC application as permanent if it is not running already. The application starts automatically too, but as temporary application.

Implementation The function uses a case structure to the result of the lists: keymember/3 function with the following parameters: odbc, 1 and application:which_application/0 (This keymember function checks, if the first parameter is a member of the list in the third parameter on the second parameters position. The which_application function returns with the running applications in a list.)

1. When the return value is true, the odbc application is already running.
2. When the return value is false, the function calls application:start/2 function with odbc and permanent atoms as parameters.

stop/0

Description The function stops the ODBC application if it is already running.

Implementation The function uses a case structure to the result of the lists: keymember/3 function with the following parameters: odbc, 1 and application:which_application/0 (This keymember function checks, if the first parameter is a member of the list in the third parameter on the second parameters position. The which_application function returns with the running applications in a list.)

1. When the return value is true, the odbc application is already running. The function calls application:stop/1 function with odbc as parameter.
2. When the return value is false, the application is not running.

3.3 Module db_init

3.3.1 Module description

The module constructs the database, it can build up the ODBC connection, drop and create the tables, and stop the connection.

3.3.2 Functions

Exported functions

connect/0

Description The function builds up the ODBC connection to the database. The function returns a tuple. If the connecting was successful the first element of the tuple is an ok atom, and the second element is the identifier of the connection.

This function is used in the d_client module.

Example: `{_,Con}=db_init:connect()`

Implementation The function contains one ODBC function-calling which builds up the connection. In the ODBC:connect/2 calling are the following parameters:

- The first parameter is a string with four components:
  - DSN is the name of the ODBC connection.
  - DBQ is the name of the database.
  - UID is the MYSQL user name.
  - PWD is the MYSQL user's password.
- The parameter autocommit off is necessary for aspect of speed. If autocommit is on there is a commit after every MYSQL command, and it causes almost 80 times slower execution of the Init function.

disconnect/1

Description The function closes the connection between Erlang and the MYSQL server. The parameter is an ODBC connection identifier.

Parameter of the function:

- Ref is the identifier of the connection.

This function is used in the d_client module.

Example: `db_init:disconnect(Con)`

Implementation The function contains one ODBC function-calling which clo-ses up the connection.

init/1

Description The function creates the database structure for the parsed code. The parameter is an ODBC connection identifier.

This function is used in the d_client module.

Example: `db_init:init(Con)`

Implementation At first it drops every previous tables with the same name as we will use. After it executes a commit. Then it creates the tables. The structure of the database is in the appendix (4.2). In the table the primary keys are marked with bold character-type. The calling of add_forbidden_names(Ref) calling fills up the forbidden_names table. The forbidden names are in the appendix (4.1)

The structure of the database:

- The lines came from the syntax tree (the names of the tables are the same as the node types of the syntax tree).
- The column of the tables are one identifier (integer), one module identifier and the children of the node:
  - if they are syntax tree then the attribute name is the type of the node,
  - if one of them is list of syntax trees then we made an argument position attribute(it is the order of the children and see the comment which describes which position belongs to which child) and the argument attribute (the identifier of the node)
  - if more of them are list of syntax trees then we have the argument position, and the argument id, but we need an argument qualifier, which shows the origin of the child syntax tree.(which list did it come from)
- Not all erlang node_types are in the database as separate tables: the nodes, which has only one attribute can fully rebuild from the name table, because it contains the identifier.
- We have own tables:
  - **forbidden_names**:This table contains the forbidden names at renaming variables or functions. The forbidden names has three main groups:

    * Auto-imported BIFs. By convention, most built-in functions (BIFs) are seen as being in the module erlang. A number of the BIFs are viewed more or less as part of the Erlang programming language and are auto-imported. Thus, it is not necessary to specify the module name and both the calls atom_to_list (Erlang) and erlang:atom_to_list (Erlang) are identical.
    * The reserved words in Erlang
    * The reserved words added by the user.
- **fun_visib**: in this table we store which clauses belongs to the function. The 0 argument stores the arity of the function.
- **id_count**: it contains the identifiers of the modules with their formlist identifiers and the greatest identifier in the module.
- **module**: it contains the identifier of the modules and their full path in as a varchar.
- **name**: it contains the name of the node and the identifiers (module and node).
- **node-type**: it is a help table which will help us find the child nodes faster than skimming through all the tables. This way when we have to build a node and we have the ids (module and node) of it, we just have to look up the type of it from this new table, and then we will know which table describes the node, and we don't have to browse through the whole database which could be very big, and could cause a very slow program.
- **pos**: this table contains the position informations(line, column) from a node by identifiers (module and node).
- **precomment and postcomment**: it is possible to add post- and precomments to a node. We already knew this, what we didn't know that the erl_recomment uses it, and it cannot be replaced by a Comment node, because it is handled different way (the positioning is not the same). These pre- and postcomments are normal Comment nodes, but they are linked to another node, therefor our traversal didn't visited it. Because we have to preserve these values, we have to add this extra traversing to the database builder, and to the opposite way to have a correct syntax tree, that represents the code we started up with. Besides that we get the original code back.
- **scope**: This table contains the scope of the current variable. In this time it means in which expression is the binding occurrence of the current variable. This identifier can be a clause, a function expression a list comprehension.
- **scope_visib**: This table contains the relations (hierarchical structure) of the scopes. The identifier belongs to the inner scope and the target belongs to the outer scope.

- **var_visib and fun_call**:This tables contains identifiers of variables. The first identifier is the module identifier, the second is the current variable/function's id, and the third id is the target (parent) variable/function's id.

Local functions

add_forbidden_names/1

Description The function fills the forbidden_names table in the database. The parameter is an ODBC connection identifier.

This function is used in the init/1 function.

Example: `add_forbidden_names(Ref)`

Implementation The function fills the forbidden_names table in the data-base by inserting the type and the forbidden_name during an SQL query. The type is 1 if the name is a BIF, and 2 if the name is a reserved word.

Parameter of the function:

- Ref is the identifier of the connection.

3.4 Module into_db

3.4.1 Module description

The module's main function is to put the abstract syntax tree (and the original code) into the database.

Exported functions:parse/2, increment/1, get_id/1, set_positions/2

Imported functions from erl_syntax: subtrees/1, update_tree/2, set_ann/2, add_ann/2, get_ann/1

3.4.2 Functions

Exported functions

get_id/1

Description The function gives a unique identifier.

The parameters:

1. Pid : the pid of the spawn in parse_database

This function is used in the add_id/4 function.

Example: `Id = get_id(Pid)`

Implementation The function is cooperate with increment/1 function. The get_id sends to the Pid with self(), when increment receive this number it will send N+1, a unique value back, and call itself again with this new parameter.In the same time get_id is waiting to receive the unique number, and when get it returns with it.

increment/1

Description The function produces the unique identifier. It cooperates with function get_id/1.

The parameters:

1. N : the previous identifier.

This function is used in the increment/1 and parse_database/3 functions.

Example: `increment(N+1)`

Implementation The get_id sends to the Pid with self(), when increment receive this number it will send N+1, a unique value back, and call itself again with this new parameter.In the same time get_id is waiting to receive the unique number, and when get it returns with it.

parse/2

Description The function parses the source file, and if there was no error it recomments the file and uses some format tools and write out the result to the file, and after that it puts into the database.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. File : The source file

This function is used in the d_client module to store the source into the database.

Example: `into_db:parse(Con,File)`

Implementation At first the function parses the file with epp_dodger1: parse_file/1 - epp_dodger1 is a modified version of epp_dodger.

1. If the parsing was successful with the Forms as result, then the function calls the check_error function to the Forms.

   After the error guessing the function calls erl_comment_scan:file (File) to get the Comments, and erl_recomment1:recomment_forms (Forms, Comments) to put back the comments into the Forms. This two step is needed, because the comments were removed during the parsing, but we will need them to build up the source code again.

After that we have to made some format tools on the Forms. The function calls erl_syntax:flatten_form_list and erl_prettypr:format and the result will be written back into the File (after opened it to write with file:open/2 function).

In the next block the function makes the first steps (except the format and write out) again but on the new file, and this result will be stored into the database with calling parse_database/3 function with the Result Forms and Ref.

2. If there was error during the parsing, then the procedure is exit with an error message.

set_positions/2

Description The function updates the positions in the database after the refactor step using a simultaneous traverse on the database and the syntax tree.

The parameters:

1. Ref : The reference of the ODBC connection.
2. File : The path of the current file.

This function is used in the d_client:rename_variable/4 and d_client:reorder_funpar/4 functions.

Example: `into_db:set_positions( Con, File)`

Implementation

- At first the function parse the file by calling epp_dodger1:parse_file/1 function. The second element of the result tuple are the forms.
- After it the function collects the comments from the file by calling erl_comment_scan:file/1 function.
- After it the function concatenates the forms with the comments by calling erl_recomment1:recomment_forms/2 function.
- After it the function uses erl_syntax:flatten_form_list/1 function to the forms.
- At the end the function starts the simultaneous traverse of the forms and the database.

Local functions

add_id/4

Description The function adds a unique identifier to the node, and puts into the correct database table the datas of the node.

The parameters:

1. Ref : the reference id of the ODBC connection.
2. Pid : the pid of the spawn in parse_database.
3. MId : the identifier of the module.
4. Node : the actual node which will be put into the database.

This function is used in the parse_database/3 as the first parameter of postorder/5 function.

Example: `fun add_id/4`

Implementation

- At first the function gets a unique identifier for the Node by calling get_id(Pid).
- After it the function adds the identifier to the syntax tree as an annotation by calling set_ann(Node, [Id]), and the result node will be our new Node (Node2).
- The functions gets the position of the Node by calling erl_syntax: get_pos.
- After it the function puts into the database the Node and the datas of the Node by lists:foldl where is an inside function with a case structure to put every node by types into the database: by node to node
  - at first the function checks having pre- or postcomment the node or not, and if it has, the comments will putted into the database (into the pre- or postcomment table).
  - The position information of the node will putted into the pos table, the is_tuple case is needed to get the correct position, because if the position is not tuple, it means the 0,0 position.
  - And after it is a big case structure where every branch is a node-type, and this guaranties that every node will be putting into the correct table (to his own node-type table). With every node the node_type information is putting into the databa-se, to the node_type table. The datas are the children nodes id-s. These id-s can get with a get_ann function to the children nodes. To get the children nodes there are node_type specific erl_syntax functions - for example erl_syntax:application_ope-rator. If the result of these function is not a simple node, but lists of node - for example erl_syntax:application_arguments - every element of the list have to put into the database with a lists:foldl function to the list with an internal function which put the single nodes into the database.
    It is an important thing when you want to put into the databa-se an integer value through the ODBC connection, you have to convert it to list, but the string are correct with only between "".
    The description of node_types are in a separate file.

check_error/1

Description The function checks the parsed file not to contain error_marker node.

The parameters:

1. Fs : The result Forms of the parsing method.

This function is used in the parse/2 function.

Example: `check_error(Forms)`

Implementation The main part is lists:foreach(Fun, Fs) function, which walks the Forms, and checks every node that it is a error_marker or not in the Fun, and gives an error message and exits when guessing an error.

parse_database/3

Description This function stores the datas of the File (module) from the Forms into the database. At first the function checks if the current file already in the database or not. If it is in then drop the datas of the file. After it fills up the database with the information from the file (the syntax tree nodes and the visibility informations too).

The parameters:

1. Forms : the tree which have to be stored to the database.
2. Ref : The reference id of the ODBC connection.
3. File : the current file which is under updating.

This function is used in the parse/2 function.

Example: `parse_database(Forms5, Ref, File)`

Implementation

- The function first check the database if the current file (third parameter) already in the database inside a case structure by calling exists_in_dbase/2 with the last two parameter.
    1. When a previous version of the file was in the database - the returned tuple was true, and the identifier of the module: the previous datas are deleted from the database by calling drop_ from_dbase/2 function with the identifier of the connection and the module.
    2. When the file is new for the database - the returned tuple was false and the identifier of the module - no special things needed.
- The function starts the produce of unique identifiers by calling increment with 0 as parameter.
- After it starts the walking of the Form by postorder ingress with the function of add_id/4.

- After it stops the communication when every process had finished by sending stop to function increment.
- After it starts the storing of the visibility informations by preorder ingress with the function of visib/3.
- After it calls the put_function_calls_in_db/1 function with the reference id of the ODBC connection. The called put_function_calls_in_db will search the code for every possible function-calling in the source and stores the into the database.
- At the end commits the database, to store the datas.

exists_in_dbase/2

Description The function checks the current file if it exists already in the database. Returns with a tuple: the first element is a boolean (existing module is true), the second element is the identifier of the current module. The parameters:

1. Ref : The reference id of the ODBC connection.
2. File : The path of the current file.

This function is used in the parse_database/3 function.

Example:`exists_in_dbase(Ref, File)`

Implementation

- At first the function gets the identifier of the module from the module table where the path was equal to the second parameter by a select SQL query. The conversion of the second parameter is by calling io_lib:write_string/1 function.
- The following steps are depends on the result of the previous query.

  Inside a case structure the function checks if the identifier is empty or not.

  1. When the identifier is empty, the return value of the condition check is true then the file is not already in the database. The function creates a new module identifier and returns with the "false, module identifier" tuple.
     - At first the function gets the maximum from the existing module identifiers from the module table.
     - If the identifier was null (checking in a case structure) the new identifier will be 1. Else the identifier will be the greatest identifier plus 1. The conversion from the integer to the query is made by integer_to_list/1 function.
     - The function inserts into the module table the new identifier as module identifier and the second parameter as the path of the module.

2. When the module list is not empty, a previous version of the module is already in the database. The function returns with `"true, identifier of the module (header of the list)"`.

postorder/5

Description The function walks the Tree by postorder ingress, and in every step uses a function to the subtree/node and returns with the first parameter's return value.

The parameters:

1. F : the function which will execute on every node(the function has three parameters).
2. Ref : the reference id of the ODBC connection.
3. Pid : the pid of the spawn in parse_database.
4. MId : the identifier of the module.
5. Tree : the structure, which have to be walked and update.

This function is used in the parse_database/3 function to give unique identifier and put it to the database to every syntax-tree node.

We used postorder ingress because when we give an identifier and store the node to the database we already have to know the identifier of its subtrees.

Example: `Forms2 = postorder(fun add_id/3, Ref, Pid, MId, Forms)`

Implementation The function calls its first parameter, which is a function with four parameter. In its fourth parameter is a case structure

- when the fifth parameter of the postorder function has no subtrees then it gives back the Tree parameter
- else calls the update_tree function and at when executing the second parameter there is the recursive calling of postorder function with a list generator.

drop_from_dbase/2

Description The function delete every data from the database which depends to the previous version to the file.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. MId : The identifier of the current module.

The function is used in the parse_database/3 function.

Example: `drop_from_dbase(Ref, MId)`

Implementation The function deletes every information of the current module from the database. The deleting is made by a delete from every of the database where the module identifier is equal with the second parameter. One-one query belong to every table of the database (4.2).

preorder/4

Description The function walks the Tree by preorder ingress, and in every step uses a function to the subtree/node and returns with the first parameter's return value.

The parameters:

1. F : the function which will use on every node(the function has three parameters).
2. Ref : the reference id of the ODBC connection.
3. MId : the identifier of the module.
4. Tree : the structure, which have to be walked and update.

This function is used in the parse_database/3 function with the visib/3 function to store the visibility informations to the database.

We used preorder ingress because the binding informations can be easily found in this direction (when we reach a variable, we already know its statement(existing or not,visible or not)).

Example: `preorder(fun visib/3, Ref, MId, Forms2)`

Implementation The function starts with a case structure, because when the node type is function it needs special process

- when the third parameter's type is function then we call the first parameter with the second and the third parameter.
- else we have to call the recursive preorder function to every subtrees of the node. The collecting of the subtrees is made by lists:map function.

visib/3

Description The function stores the function-node's visibility information (variables and functions) into the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. MId : The identifier of the module.

3. Node: The subtree of the current function-node.

This function is used in the preorder/4 function.

Example: `fun visib/3`

Implementation
- At first the function gets the identifier of the functi-on-node with get_ann/1 function.
- After it gets the arity of the function with erl_syntax:function_arity/1 function
- After it gets the list of the function-clauses with erl_syntax:function_clauses/1 function
- After it gets the identifiers of the clauses to a list with get_ann/1 function inside a lists:map/2 function on the clauses.
- After it for every clause calls the visib_clause/3 function inside a lists:map/2 function on the clauses. This called functions will store the variable visibility informations to the database.
- After it stores the module, the function and its arity to the fun_visib table with an odbc:sql_query. The query inserts the identifier of the module and the function and its arity with 0 position to the fun_visib table.
- At the end it stores the clauses of the function to the fun_visib table with an odbc:sql_query inside lists:foldl/3 function to the identifiers' list to the clauses. The query inserts the identifiers of the module and the function, the position (1 or greater) of the current clause and identifier of the current close to the fun_visib table.

visib_clause/3

Description This function searches the current function-clause for the variable visibility informations and stores them to the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. MId : The identifier of the module.
3. Clause : The subtree of the current clause.

This function is used in visib/3 function.

Example: `visib_clause(Ref, MId, Clause)`

Implementation The main problem is with the implementation of the clause node was that it has two main component: pattern and body, and variables and their binding occurrence can be in both part.

- At first the function gets the pattern of the clause (it means the parameter list) with erl_syntax:clause_patterns/1 function.

- After it starts a separate process which connects the same variables to each other and works as a server with the visibility/3 function. The spawn returns with a pid after starting which can be identify the server.
- After it starts a separate process which calculates the scopes of the variables and works as a server with the scope/2 function. The spawn returns with a pid after starting which can be identify the server.
- The function needs only the variables from the parameter list. The function uses a lists:filter/2 to the pattern to choose the nodes which type is variable with erl_syntax:type/1 function is a case structure.
- After it the function gives the element from the pattern-variables list to the server with lists:map/2 and put_in/2 functions. At this point is the end of the pattern piece handling.
- After it uses a preorder ingress to the current clause (it means the body part) to handle the variables visibility informations with visib_ fun_expr/5 function which will give the variables to the ser-ver.
- After it gets the collected variables from the server with get_last_ variables/1 function which returns with the variable visibility informations.
- After it gets the collected scope informations from the scope server with get_scope_list/' function.
- After it puts the variable visibility informations to the database with put_visib_in_database/3 function.
- After it puts the scope informations to the database with put_scope_visib_in_database/3 function.
- At the end stops the servers.

put_visib_in_database/3

Description This function puts the variable visibility informations (come from the third parameter) to the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. MId : The identifier of the module.
3. Visibility: The list of the variable visibility information. The list contains an other list of pairs of variable identifiers (actual and target variable).

This function is used in visib_clause/3 function.

Example: `put_visib_in_database(Ref, MId, Visib)`

Implementation The function calls an embedded sql_query which put the three identifiers - of the module and a variable and its target variable pair - to the var_visib table in the database. The pair come from two lists:map/2 functions to the third parameter end embedded its elements.

put_into_scope/4

Description This function puts the scope informations (come from the third and fourth parameter) to the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. MId : The identifier of the module.
3. Pid : The identifier of the scope server.
4. Node : The identifier of the current node.

This function is used in preorder_2/6 function.

Example: `put_into_scope( Ref, MId, Pid2, hd( get_ann( Tree)))`

Implementation

- At first the function get the current scope by calling get_current_scope/1 function.
- The function stores the scope of the node to the scope table with an insert SQL query. The module identifier will be the converted second parameter. The identifier will be the converted fourth parameter. The scope will be the converted scope identifier. The conversions made by integer_to_list/1 function.

put_scope_visib_in_database/3

Description This function puts the scope visibility informations (come from the third parameter) to the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. MId : The identifier of the module.
3. Scope : A list which contains pair of identifiers. The pair contains the scoping informations: the first element is the inner scope and the second element is the outer.

This function is used in visib_clause/3 function.

Example: `put_scope_visib_in_database(Ref, MId, Scope)`

Implementation The function calls an embedded sql_query which put the three identifiers - of the module and a scope and its outer scope pair - to the scope_visib table in the database. The pair come from two lists:map/2 functions to the third parameter end embedded its elements.

put_function_calls_in_db/1

Description The function search the database and get every possible calling of the functions and stores into the fun_call table in the database.

The parameter:

1. Ref: The reference id of the ODBC connection.

This function is used in parse_database/3 function.

Example: `put_function_calls_in_db(Ref)`

Implementation The main problem is to find every possible calling of the function and store the different calling-types to the database. The possible function call expression involving module names in Erlang:

The sample functions are the following:

1. `test_fun(P)` →
   `{echo, P}.`
2. `test_fun(P1, P2)` →
   `{echo, P1, P2}.`
3. `caller(F,P)` →
   `F(P).`
4. `caller_2(F,P1,P2)` →
   `F(P1,P2).`

The possible function calling for the sample functions:

1. `func_mod:test_fun(P)` or `func_mod:test_fun(P1,P2)`
2. `{func_mod, test_fun}(P)` or `{func_mod, test_fun} (P1,P2)` this mode is deprecated in the documentation
3. `apply({func_mod, test_fun}, [P])` or `apply({func_mod, test_fun}, [P1,P2])` this mode is deprecated in the documentation
4. `apply(func_mod, test_fun, [P])` or `apply(func_mod, test_fun, [P1, P2])`
5. `caller(fun func_mod:test_fun/1, P)` or `caller_2(fun func_mod:test_fun/2, P1,P2)` this mode is available only from ERTS 5.5

The possible function calling expressions inside BIFs:

1. `spawn( func_mod, test_fun, [P])` or `spawn( func_mod, test_fun, [P1,P2])`
2. `spawn(node(),func_mod, test_fun, [P])` or `spawn( node(), func_mod, test_fun, [P1,P2])`

3. `spawn_link( func_mod, test_fun, [P])` or `spawn_link(func_mod, test_fun, [P1,P2])`
4. `spawn_link( node(), func_mod, test_fun, [P])` or `spawn_link(node(), func_mod, test_fun, [P1,P2])`
5. `spawn_opt( func_mod, test_fun, [P])` or `spawn_opt( func_mod, test_fun, [P1,P2])`
6. `spawn_opt( node(), func_mod, test_fun, [P])` or `spawn_opt( node(), func_mod, test_fun, [P1,P2])`
7. `erlang:hibernate( func_mod, test_fun, [P])` or `erlang:hibernate(func_mod, test_fun, [P1,P2])`

The implementation of the problem:

The first problem is to get every possible functions with their informations, and every possible function callings. After the collecting the function stores the datas to the database.

- At first the function collects every functions and their datas (module name, function name, function arity, function identifier, module identifier, exported the function or not) by calling get_fun_da-tas/1 function with the reference of the ODBC connection.
- After it collects the datas of the function callings which are in applications by calling get_application_data/1 function with the reference of the ODBC connection.
- After it collects the datas of the function callings which are implicit functions by calling get_implicit_fun_datas/1 function with the reference of the ODBC connection.
- After it collects the names of every modules which are already in the database by calling refactor:get_module_names/1 function with the reference of the ODBC connection.
- After it concatenates the two function calling lists into a static function calling list.
- After it filters the static function calling list to the already existing modules of the database. This step is only for effectiveness: for example with this filter runs 9062 check, without this 54530 (for d_client and into_db modules).
- At the end it stores the collected datas to the database with the embedded put_fun_calls_into_dbase/3 function to the current element and the filtered list inside a lists:foreach/2 function to the function datas.

get_fun_arity/3

Description The function gets the arity of the current function from the database.

The parameters:

1. Ref : The reference of the ODBC connection.
2. MId : The identifier of the module.
3. FunId : The identifier of the current function.

The function is used in get_fun_datas/1 function.

Example: `FunArity = get_fun_arity(Ref, MId, FunId)`

Implementation The function gets the arity of the function from the data-base, and returns with it.

- At first the function gets the arity number from the function and clause tables by selecting and counting the identifiers of the clauses which belong to the current function. The select SQL queries return with a tuple with three elements, where the third element is a list of tuple which is the result of the select. We only need this result's element.
- The returned element is a list, so the function converts it to integer with list_to_integer/1 function.

fun_is_exported/4

Description The function returns with a boolean value which shows if the current function is exported or not.

The parameters:

1. Ref : The reference of the ODBC connection.
2. MId : The identifier of the module.
3. FunName : The name of the current function.
4. Arity: The arity of the current function.

The function is used in get_fun_datas/1 function.

Example: `IsExported = fun_is_exported( Ref, MId, Name, FunArity)`

Implementation

- At first the function collects the exported functions in the module to a list by calling refactor:get_export_list/2 function.
- After it the function executes the test if the current function (identified by the name and arity) is in the export list or not by calling refactor:simple_member_b/3 function.
- At the end the function returns with the boolean.

get_fun_datas/1

Description The function collects the informations of every functions in the database. The informations are the followings:

- the name of the function
- the identifier of the function
- the identifier of the module
- the arity of the function
- exported the function or not

The function is used in the put_function_calls_in_db/1 function.

Example: `FunDatas = get_fun_datas(Ref)`

Implementation

- At first the function collects the function names, the function identifiers and the module names into a list from the function and name tables. The module identifiers have to be the same in both tables. The position have to be 0 (function name) and the identifier of the clause has to be equal to the identifier in the name table.
- After it the function completes the tuple with the module name. The completion is made inside a lists:map/2 function to the previous list (with three elements tuples). The embedded function gets the module name by calling refactor:get_module_name/2 and returns with the four elements tuple.
- After it the function completes the tuple with the function arity. The completion is made inside a lists:map/2 function to the previous list (with four elements tuples). The embedded function gets the module name by calling refactor:get_fun_arity/3 and returns with the five elements tuple.
- After it the function completes the tuple with the boolean if the function is exported or not. The completion is made inside a lists:map/2 function to the previous list (with five elements tuples). The embedded function gets the module name by calling fun_is_exported/4 and returns with the six elements tuple.
- At the end the function returns with the list of the six elements tuples.

get_applications_data/1

Description The function collects the static function calling expressions and their datas where the function is not in an implicit function calling.

The parameter:

- Ref : The identifier of the ODBC connection.

The function is used in the put_function_calls_in_db/1 function.

Example: `ApplicationDatas = get_applications_data(Ref)`

Implementation The function collects the datas three separate lists, and concatenate the lists at the end. The three list represent the three kind of function calling expressions: the simple, the identified by a module name and deprecated.

1. Simple applications:
   - At first the function collects the module identifier, the function name, the identifier from the application, node_type, name tables with a select SQL query. The module identifiers have to be the same in every three tables. The identifiers have to be the same in the node_type and name table and with the argument in the application table. The position has to be 0 (function name) and the type has to be 3 (application).
   - After it the function completes the list with the function arity. The completion is made inside a lists:map/2 function to the previous list (with three elements tuples). The embedded function gets the arity by a count function in a select SQL query from the application table. The module identifier and the identifier have to be the same as the current element's identifiers but the position can not be 0. The conversion from the result one element list is made by list_to_integer/1 function.
   - After it the function completes the list with the module name. The completion is made inside a lists:map/2 function to the previous list (with four elements tuples). The embedded function gets the module name by calling get_module_name/4 function.
2. Applications qualified by module names:
   - At first the function collects the module and function name and identifier from the application, node_type, name (duplicated) , module_qualifier tables with a select SQL query. The module identifiers have to be the same in every five tables. The identifiers have to be the same in the node_type, module_qualifier and name table and with the argument in the application table. The position has to be 0 (function name) and the type has to be 31 (module_qualifier). The first identifier in the name table has to be the same as the module from the module_qualifier table, and the second has to be the same as the body.
   - After it the function completes the list with the function arity. The completion is made inside a lists:map/2 function to the previous list (with four elements tuples). The embedded function gets the arity by a count function in a select SQL query from the application table. The module identifier and the identifier have

to be the same as the current element's identifiers but the position can not be 0. The conversion from the result one element list is made by list_to_integer/1 function.

3. Deprecated function calls:
   - At first the function collects the module identifier, the function name identifier, the identifier from the application, node_type (duplicated), tuple tables with a select SQL query. The module identifiers have to be the same in every four tables. The identifiers in the node_type table has to be the same as the argument in the application table and as the identifier in the tuple table. The position has to be 0 (function name) and the type has to be 48 (tuple). The element of the tuple table has to be the same as the identifier in the second node_type table. The type in the second node_type table has to be 3 (application).
   - After it the function completes the list with the function arity, function name and module name. The completion is made inside a lists:map/2 function to the previous list (with three elements tuples). The first embedded function gets the arity by a count function in a select SQL query from the application table. The module identifier and the identifier have to be the same as the current element's identifiers but the position can not be 0. The conversion from the result one element list is made by list_to_integer/1 function. The second embedded function gets the module and function name with a select SQL query from the name and tuple tables. The module identifiers have to be the same in both tables and as the current element's module identifier. The identifier in the tuple table has to be the same as the current element's tuple identifier. The identifier of the name table has to be the same as the element in the tuple table. The result will two name in a list: the header is the module name and the tail is the function name. The three new element will be part of the result five elements tuple.

get_implicit_fun_datas/1

Description The function collects every informations of implicit fun call expressions which are stored in the database.

The parameter:

1. Ref : the reference of the ODBC connection

The function is used in the put_function_calls_in_db/1 function.

Example: `ImplicitFunDatas = get_implicit_fun_datas(Ref)`

Implementation
- At first the function collects the module identifier, the function identifier, the function name identifier and the type from

the implicit_fun and node_types tables with a select SQL query. The module identifiers have to be the same in both tables. The name identifier in the implicit_fun table has to be the same as the identifier in the node_type table.

- After it the function completes the list with the function arity, function name and module name. The completion is made inside a lists:map/2 function to the previous list (with four elements tuples). The function executes a pattern matching on the last element of the tuple (type):

  1. When the type is 2 (arity_qualifier):
     The first embedded function gets the name and arity by a select SQL query from the arity_qualifier, name and integer tables. The module identifiers have to be the same as the current element's module identifier . The identifier in the arity_qualifier table has to be the same as the name identifier of the current element. The second embedded function gets the module name by calling get_module_name/4 function. The three new element will be part of the result five elements tuple.
  2. When the type is 31 (module_qualifier):
     The embedded function gets the module name, the function name and arity by a select SQL query from the module_qualifier, arity_qualifier, name (duplicated)and integer tables. The module identifiers have to be the same in all tables as the current element's module identifier . The module in the module_qualifier table has to be the same as the identifier in the first name table. The body in the module_qualifier table has to be the same as the identifier in the arity_qualifier table. The body in the arity_qualifier table has to be the same as the identifier in the second name table. The arity in the arity_qualifier table has to be the same as the identifier in the integer table. The identifier in the module_qualifier table has to be the same as the name identifier of the current element. The new elements will be part of the result five elements tuple.

- At the end the function returns with the list of the five elements tuples.

get_module_name/4

Description The function gets the original module name of the function which are not qualified by module name, but imported or auto imported.

The parameters:

1. Ref : The reference of the ODBC connection.
2. MId : The identifier of the current module.

3. Name : The name of the function.
4. Arity : The arity of the function.

The function used in get_implicit_fun_datas/1 and get_applications_data/1 functions.

Example: `Module = get_module_name( Ref, MId, Name, Arity)`

Implementation • The function uses a case structure to the return value of the is_imported/4 function to decide if the current function is imported in the module.

1. When the return value is false, the current function is not imported.The next step is an other case structure to the return value of the refactor:get_module_name_if_exists_in_module/4 function to decide if the current function is exists in the module and has module qualifier:
   (a) When the return value is false the function returns with erlang module name.
   (b) Else the function returns with the module name from the get_module_name _if_exists_in_module/4 function.
2. Else the function returns with the module name from the is_imported/4 function.

is_imported/4

Description The function decides if the current function is imported in the current module or not. If it is imported the function returns with the name of the original module. Else it returns with false.

The parameters:

1. Ref : The reference of the ODBC connection.
2. MId : The identifier of the current module.
3. Name : The name of the function.
4. Arity : The arity of the function.

The function used in get_module_name/4 functions.

Example: `case is_imported(Ref,MId,Name,Arity) of`

Implementation • At first the function gets the identifiers of the functions in the import list by calling refactor:get_import_list_ids/2 function.

- The function uses a case structure on imported identifiers list:
  1. When the list is empty the function returns with false value.
  2. Else the function get the imported functions (module name, function name, function arity in a tuple) by calling refactor:get_imported_functions/3 function.

- The function uses a case structure to the result of the module_member/3 function:
  1. When the returned value was false the function returns with false value.
  2. When the returned value was a module name the function returns with this name.

module_member/3

Description The function checks if the current function is a member of which module between the imported functions. The function returns with false value when the current function is not a member of the imported functions, and returns with the original module name if it is a member.

The parameters:

1. Name : The name of the current function.
2. Arity : The arity of the current function.
3. Importedfunctions : A list of tuples (Module name, function name, function arity) of the imported functions in the current module.

The function is used in the is_imported/4 function.

Example: `case module_member( Name, Arity, ImportedFunctions) of`

Implementation The function uses a pattern matching on the last parameter:

1. When the list is empty the function returns with false.
2. When the current function's name and arity are the same as the last two element in the header of the list the function returns with the module name - the first element from the header.
3. In every other case the function call itself recursively. The first two parameter remain the same, the third parameter will be the tail of the original third parameter.

put_fun_calls_into_dbase/3

Description The function stores the function calls into the database.

The parameters:

1. Ref : The reference of the ODBC connection.
2. Data : The datas of the current function in a tuple (module name, function name, function arity, function identifier, module identifier, is the function exported or not).

3. FilteredStaticFunCalls : The list of tuples which stores the informations of the function calling expressions just in the modules which are already in the database.

The function used in the put_function_calls_in_db/1 function.

Example: `put_fun_calls_into_dbase( Ref, Element, FilteredStaticFunCalls)`

Implementation The function uses a pattern matching on the last two parameters:

1. When the third parameter is an empty list the function returns with ok.
2. When the same function is in the second and the header of the third parameter (the module name, the function name and the function arity is the same) the function stores the function call into the fun_call table. The new module and function identifier will be the second function and module identifier from the header of the third parameter. The target module and function identifier will be current function and module identifier from the second parameter. The conversion to list is made by integer_to_list/1 function to the datas. The function calls itself recursively with the same parameters, just with the tail of the third parameter.
3. In every other cases the function calls itself recursively with the same parameters, just with the tail of the third parameter.

preorder_2/6

Description The function walks the Tree by preorder ingress, and in every step uses a function to the subtree/node and returns with the first parameter's return value.

The parameters:

1. Newf : The function which will use on every node(the function has four parameters).
2. Ref : The reference id of the ODBC connection.
3. Pid : The pid of the spawn connection in visib_fun_expr_cla-use/5
4. Pid2 : The pid of the scope server.
5. MId : The identifier of the module.
6. Tree : The structure, which have to be walked and update.

This function is used in the visib_fun_expr_clause/5 and visib_clause/3 functions to store the visibility informations to the database.

We used preorder ingress because the binding informations can be easily found in this direction (when we reach a variable or a function, we already know its statement(existing or not, visible or not)).

Example: `preorder_2(fun visib_fun_expr/5, Ref, Pid, Pid2, MId, Clause)`

Implementation At first the function stores the scope informations to the database by calling put_into_scope/4 function. The function starts with a case structure, because when the node type is fun_expr or variable it needs special process

- when the sixth parameter's type is fun_expr then we call the first parameter with the remaining parameters.
- else if when the sixth parameter's type is variable then we call the put_in/2 function with the last two parameters to add it to the variable list in the function.
- else if when sixth parameter's type is list_comp then we call the visib_list_comp/5 function to store the informations.
- else if when sixth parameter's type is generator then we call the visib_generator/5 function to store the informations.
- else we have to call the recursive preorder function to every subtrees of the node (if it is not empty). The collecting of the subtrees is made by lists:map function.

visib_fun_expr/5

Description The function gets every clauses of the fun_expr node parameter and handles every clause to store the visibility informations to the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. Pid : The pid of the spawn connection in visib_clause/3
3. Pid2 : The pid of the spawn connection in visib_clause/3 for the scope informations.
4. MId : The identifier of the module.
5. Node: The subtree of the current function-node.

This function is used in the preorder_2/6 function.

Example: `fun visib_fun_expr/5`

Implementation

- At first the function gets the clauses of the current fun_expr node to a list by calling erl_syntax:fun_expr_clauses/1 function.
- At the end it calls visib_fun_expr_clause/5 to every element of the list. The elements of the list are getting with a lists:map/2 function.

visib_fun_expr_clause/5

Description This function searches the current fun_expr-clause for the variable visibility informations and stores them to the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. Pid : The pid of the spawn connection in visib_clause/3 for the variable visibility informations.
3. Pid2 : The pid of the spawn connection in visib_clause/3 for the scope informations.
4. MId : The identifier of the module.
5. Clause : The subtree of the current clause.

This function is used in visib_fun_expr/5 function.

Example: `visib_fun_expr_clause(Ref, Pid, Pid2, MId, Clause)`

Implementation The main problem is with the implementation of the clause node was that it has two main component: pattern and body, and variables and their binding occurrence can be in both part.

- At first the function gets the pattern of the clause (it means the parameter list) with erl_syntax:clause_patterns/1 function.
- After it calls split/2 function with the pid to split the connection.
- After it introduces a new scope with new_scope/2 with the scope server reference and the last parameter.
- The function needs only the variables from the parameter list. The function uses a lists:filter/2 to the pattern to choose the nodes which type is variable with erl_syntax:type/1 function is a case structure.
- After it the function gives the element from the pattern-variables list to the server with lists:map/2 and put_in_sh_check/2 functions. At this point is the end of the pattern piece handling.
- After it uses a preorder ingress to the current clause (it means the body part) to handle the variables visibility informations with visib_fun_expr/5 function which will give the variables to the server.
- After it gets the collected variables from the server with get_last_variables/1 function which returns with the variable visibility informations.
- After it closes the scope by calling end_of_scope/1 function.
- After it puts the variable visibility informations to the database with put_visib_in_database/3 function.

visib_list_comp/5

Description This function searches the current list_comp expressions for the variable visibility informations and stores them to the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. Pid : The pid of the spawn connection in visib_clause/3
3. Pid2 : The pid of the spawn connection in visib_clause/3 for the scope informations.
4. MId : The identifier of the module.
5. Node : The subtree of the current list_comp.

This function is used in preorder_2/6 function.

Example: `visib_list_comp(Ref, Pid, Pid2, MId, Tree)`

Implementation

- At first the function gets the bodies of the node with erl_syntax:list_comp_body/1 function.
- After it the function gets the template of the node by calling erl_syntax:list_comp_template/1 function.
- After it calls split/2 function with the pid to split the connection and with listcomp mode.
- After it introduces a new scope with new_scope/2 with the scope server reference and the last parameter.
- After it uses a preorder ingress to the list of bodies (it means the body part) to handle the variables visibility informations with visib_fun_expr/5 function which will give the variables to the server. The element of the list is getting by lists:map/2 function.
- After it changes the state to template by calling change_state/2 function.
- After it uses a preorder ingress to the template part to handle the variables visibility informations with visib_fun_expr/5 function which will give the variables to the server.
- After it changes the state to reverse by calling change_state/2 function.
- After it gets the collected variables from the server with get_last_variables/1 function which returns with the variable visibility informations.
- After it closes the scope by calling end_of_scope/1 function.
- After it puts the variable visibility informations to the database with put_visib_in_database/3 function.

visib_generator/5

Description This function searches the current generator expressions for the variable visibility informations and stores them to the database.

The parameters:

1. Ref: The reference id of the ODBC connection.
2. Pid : The pid of the spawn connection in visib_clause/3
3. Pid2 : The pid of the spawn connection in visib_clause/3 for the scope informations.
4. MId : The identifier of the module.
5. Node : The subtree of the current generator.

This function is used in preorder_2/6 function.

Example: `visib_generator(Ref, Pid, Pid2, MId, Tree)`

Implementation

- At first the function gets the body of the node with erl_syntax:generator_body/1 function.
- After it the function gets the pattern of the node by calling erl_syntax:generator_pattern/1 function.
- After it uses a preorder ingress to the body (it means the body part) to handle the variables visibility informations with visib_fun_expr/5 function which will give the variables to the server.
- After it changes the state to generator by calling change_state/2 function.
- After it uses a preorder ingress to the pattern part to handle the variables visibility informations with vi-sib_fun_expr/5 function which will give the variables to the server.
- After it changes the state to reverse by calling change_state/2 function.

visibility/3

Description This function is a server function which collects the variables from separate functions to a list. The result can be get by get_last_vari-able function.

The parameters:

1. N: An integer which handle the variable list of separate clauses.
2. List: A list of list of the current variables
3. Mode: The functionality mode of the server.

This function is starts in visib_clause/3 function.

Example: `visibility(1,[],normal)`

Implementation The function works as a server, it receives tuples which generates to working of the function to handle the variables. The first element of the tuple is always a command: stop, add, add_sh_check, get_last or split. The other element of the tuple is helping parameters to execute the command, but the pid of the spawn is fixed. The function uses a pattern matching to the first parameter:

1. When the first parameter is 1 the following the following cases are possibilities when receiving a tuple:
   - When the first element of the tuple is stop command then the server send back an ok message to the pid.
   - When the first element of the tuple is add command then the server sends back an ok message to the pid and calls recursively itself by 1, add_simple(Name,Id,List) parameters and normal mode. The Name and Id are from the received message, the List is the second parameter.
   - When the first element of the tuple is add_sh_check command then the server sends back an ok message to the pid and calls recursively itself by 1, add_simple(Name,Id,List) parameters and normal mode. The Name and Id are from the received message, the List is the second parameter.
   - When the first element of the tuple is get_last command then the server sends back an ok,List message to the pid and calls recursively itself by 1, [] parameters. The List is the second parameter. This command sends back the collected variables and empties the List.
   - When the first element of the tuple is split command then the server sends back an ok message to the pid and calls recursively itself by 2, [[],List] parameters and [Mode|normal] mode. The List is the second parameter. The Mode is the second element of the tuple.
2. When the first parameter is not 1 the following and the header of the mode list is normal the following cases are possibilities when receiving a tuple:
   - When the first element of the tuple is stop command then the server send back an ok message to the pid.
   - When the first element of the tuple is add command then the server sends back an ok message to the pid and calls recursively itself by N, add(Name,Id,List,N), Modes parameters. The Name and Id are from the received message, the List is the second parameter, N is the first parameter and Modes is the third parameter.
   - When the first element of the tuple is add_sh_check command then the server sends back an ok message to the pid and calls

recursively itself by N, add_sh_check(Name,Id,List,N), Modes parameters. The Name and Id are from the received message, the List is the second, N is the first and Modes is the third parameter.

- When the first element of the tuple is get_last command then the server sends back an {ok,X} message to the pid (X is the header of the second parameter) and uses a case structure if the first parameter is 2 or not as a logical expression:
  - When it is 2 then calls recursively itself by 1, hd(Xs), Ms parameters. The Xs is the tail of the second , Ms is the tail of the third parameter. This command sends back the collected variables and empties the current variable list (header of the List).
  - When it is not 2 then calls recursively itself by N-1, Xs, Ms parameters. The Xs is the tail of the second, Ms is the tail of the third parameter. This command sends back the collected variables and empties the current variable list (header of the List) and N is the first parameter.
- When the first element of the tuple is split command then the server sends back an ok message to the pid and calls recursively itself by N+1, [[]]++List, [Mode]++Modes parameters. The List is the second parameter, Modes is the third parameter and N is the first parameter.
- When the first element of the tuple is change command then the server sends back an ok message to the pid and calls recursively itself. The parameters depends on the State (second element of the tuple):
  - When the State is reverse the parameters are N, List, Ms: the original parameters just the third parameter is the tail of the original one.
  - When the State is not reverse the parameters are N, List, [State]++Modes. N, List, Modes are the original parameters.

3. When the first parameter is not 1 the following and the header of the mode list is listcomp the following cases are possibilities when receiving a tuple:
   - When the first element of the tuple is stop command then the server send back an ok message to the pid.
   - When the first element of the tuple is add command then the server sends back an ok message to the pid and calls recursively itself by N, add(Name,Id,List,N), Modes parameters. The Name and Id are from the received message, the List is the second parameter, N is the first parameter and Modes is the third parameter.

- When the first element of the tuple is add_sh_check command then the server sends back an ok message to the pid and calls recursively itself by N, add_sh_check(Name,Id,List,N), Modes parameters. The Name and Id are from the received message, the List is the second, N is the first and Modes is the third parameter.
- When the first element of the tuple is get_last command then the server sends back an {ok,X} message to the pid (X is the header of the second parameter) and uses a case structure if the first parameter is 2 or not as a logical expression:
  - When it is 2 then calls recursively itself by 1, hd(Xs), Ms parameters. The Xs is the tail of the second , Ms is the tail of the third parameter. This command sends back the collected variables and empties the current variable list (header of the List).
  - When it is not 2 then calls recursively itself by N-1, Xs, Ms parameters. The Xs is the tail of the second, Ms is the tail of the third parameter. This command sends back the collected variables and empties the current variable list (header of the List) and N is the first parameter.
- When the first element of the tuple is split command then the server sends back an ok message to the pid and calls recursively itself by N+1, [[]]++List, [Mode]++Modes parameters. The List is the second parameter, Modes is the third parameter and N is the first parameter.
- When the first element of the tuple is change command then the server sends back an ok message to the pid and calls recursively itself. The parameters depends on the State (second element of the tuple):
  - When the State is reverse the parameters are N, List, Ms: the original parameters just the third parameter is the tail of the original one.
  - When the State is not reverse the parameters are N, List, [State]++Modes. N, List, Modes are the original parameters.

4. When the first parameter is not 1 the following and the header of the mode list is generator the following cases are possibilities when receiving a tuple:
   - When the first element of the tuple is stop command then the server send back an ok message to the pid.
   - When the first element of the tuple is add command then the server sends back an ok message to the pid and calls recursively itself by N, add_generator(Name,Id,List,N), Modes parameters. The Name and Id are from the received message, the List is the second parameter, N is the first parameter and Modes is the third parameter.

- When the first element of the tuple is add_sh_check command then the server sends back an ok message to the pid and calls recursively itself by N, add_sh_check(Name,Id,List,N), Modes parameters. The Name and Id are from the received message, the List is the second, N is the first and Modes is the third parameter.
- When the first element of the tuple is get_last command then the server sends back an {ok,X} message to the pid (X is the header of the second parameter) and uses a case structure if the first parameter is 2 or not as a logical expression:
  - When it is 2 then calls recursively itself by 1, hd(Xs), Ms parameters. The Xs is the tail of the second , Ms is the tail of the third parameter. This command sends back the collected variables and empties the current variable list (header of the List).
  - When it is not 2 then calls recursively itself by N-1, Xs, Ms parameters. The Xs is the tail of the second, Ms is the tail of the third parameter. This command sends back the collected variables and empties the current variable list (header of the List) and N is the first parameter.
- When the first element of the tuple is split command then the server sends back an ok message to the pid and calls recursively itself by N+1, [[]]++List, [Mode]++Modes parameters. The List is the second parameter, Modes is the third parameter and N is the first parameter.
- When the first element of the tuple is change command then the server sends back an ok message to the pid and calls recursively itself. The parameters depends on the State (second element of the tuple):
  - When the State is reverse the parameters are N, List, Ms: the original parameters just the third parameter is the tail of the original one.
  - When the State is not reverse the parameters are N, List, [State]++Modes. N, List, Modes are the original parameters.

5. When the first parameter is not 1 the following and the header of the mode list is template the following cases are possibilities when receiving a tuple:
   - When the first element of the tuple is stop command then the server send back an ok message to the pid.
   - When the first element of the tuple is add command then the server sends back an ok message to the pid and calls recursively itself by N, add_template(Name,Id,List,N), Modes parameters. The Name and Id are from the received message, the List is the second parameter, N is the first parameter and Modes is the third parameter.

- When the first element of the tuple is add_sh_check command then the server sends back an ok message to the pid and calls recursively itself by N, add_sh_check(Name,Id,List,N), Modes parameters. The Name and Id are from the received message, the List is the second, N is the first and Modes is the third parameter.
- When the first element of the tuple is get_last command then the server sends back an {ok,X} message to the pid (X is the header of the second parameter) and uses a case structure if the first parameter is 2 or not as a logical expression:
  - When it is 2 then calls recursively itself by 1, hd(Xs), Ms parameters. The Xs is the tail of the second , Ms is the tail of the third parameter. This command sends back the collected variables and empties the current variable list (header of the List).
  - When it is not 2 then calls recursively itself by N-1, Xs, Ms parameters. The Xs is the tail of the second, Ms is the tail of the third parameter. This command sends back the collected variables and empties the current variable list (header of the List) and N is the first parameter.
- When the first element of the tuple is split command then the server sends back an ok message to the pid and calls recursively itself by N+1, [[]]++List, [Mode]++Modes parameters. The List is the second parameter, Modes is the third parameter and N is the first parameter.
- When the first element of the tuple is change command then the server sends back an ok message to the pid and calls recursively itself. The parameters depends on the State (second element of the tuple):
  - When the State is reverse the parameters are N, List, Ms: the original parameters just the third parameter is the tail of the original one.
  - When the State is not reverse the parameters are N, List, [State]++Modes. N, List, Modes are the original parameters.

add_simple/3

Description The function adds the identifier of the variable to the current list and returns the updated list.

The parameters:

1. Name: The name of the variable.
2. Id: The identifier of the variable.
3. List: List of tuples (variable names, variable identifiers, list of identifiers of the variable).

This function is used in visibility/3 and add/4 functions.

Example: `add_simple(Name,Id,List)`

Implementation The function adds the identifier parameter to the list of the third parameter The function uses a pattern matching to the third parameter:

1. When the third parameter is an empty list the function creates a list with one element. The element is a tuple with three elements: Name, Id, [Id]. The Name is the first parameter and the Id is the second.
2. When the third element is not an empty list then a further separation is needed because every element of the List in the third element is belongs to a variable. Name (every variable name has own list) and the identifier have to be added to the correct variable name's list. The function search the correct variable's name list from the header to the end of the list by elements. This separation is coded by case structure which analyses a logical expression on the equivalence of the first parameter(Name) and the first element of the header of the third parameter.
   - If it is true then this is the correct list. The function updates the header with added the identifier (second parameter) to the third element of the header, and returns with the updated list.
   - Else the function updates the tail of the list by calling recursively itself to the tail of the third parameter, and returns with the updated list.

add/4

Description The function adds the identifier of the variable to the current list and returns the updated list.

The parameters:

1. Name: The name of the variable.
2. Id: The identifier of the variable.
3. List: List of list of tuples (variable names, variable identifiers, list of identifiers of the variable).
4. N: Number of list elements (integer)

This function is used in visibility/3 function.

Example: `add(Name,Id,List,N)`

Implementation The function adds the identifier parameter to the list of the third parameter with add_simple or add_to_nth

- At first the function checks if there any element in the list which has the same name as the first parameter and creates a list of booleans from the result. The function uses lists:map/2 function to the third parameter and inside it a lists:any/2 function to its elements to check the condition.
- After it gets the positions of the current element from the lists by calling get_true_pos/1 with the list of booleans.
- At the end it makes a condition check inside a case structure to check if there any list with the same name or not(the position is one or greater than the fourth parameter)
  - When the condition is true then the function returns with the updated third parameter. The update means that the header of the list will be the returned value of add_simple/3 function with the first, the second and the header of the third parameters as parameters.
  - Else it returns with the result of the add_to_nth/4 function with the first three parameters and the received position as parameters.

add_generator/4

Description The function adds the identifier of the variable to the current list and returns the updated list.

The parameters:

1. Name: The name of the variable.
2. Id: The identifier of the variable.
3. List: List of list of tuples (variable names, variable identifiers, list of identifiers of the variable).
4. N: Number of list elements (integer)

This function is used in visibility/3 function.

Example: `add_generator(Name,Id,List,N)`

Implementation The function updates the header of the list by calling add_simple_generator/3 with the first, second and the header of the third parameters. After it returns with the updated list.

add_simple_generator/3

Description The function updates the list of the variables with the current variable.

The parameters:

1. Name : The name of the current variable.

2. Id : The identifier of the current variable.
3. List : The list of the previously added variables.

The function is used in the add_generator/4 function.

Example: `add_simple_generator(Name,Id,X)`

Implementation The function returns with the updated list after concatenated the third parameter with the [{Name,Id,[Id]}] expression from the first two parameter.

add_template/4

Description The function adds the identifier of the variable to the current list and returns the updated list.

The parameters:

1. Name: The name of the variable.
2. Id: The identifier of the variable.
3. List: List of list of tuples (variable names, variable identifiers, list of identifiers of the variable).
4. N: Number of list elements (integer)

This function is used in visibility/3 function.

Example: `add(Name,Id,List,N)`

Implementation The function adds the identifier parameter to the list of the third parameter with add_simple_template or add_to_nth

- At first the function checks if there any element in the list which has the same name as the first parameter and creates a list of booleans from the result. The function uses lists:map/2 function to the third parameter and inside it a lists:any/2 function to its elements to check the condition.
- After it gets the positions of the current element from the lists by calling get_true_pos/1 with the list of booleans.
- At the end it makes a condition check inside a case structure to check if there any list with the same name or not(the position is one or greater than the fourth parameter)
  - When the condition is true then the function returns with the updated third parameter. The update means that the header of the list will be the returned value of add_simple_template/3 function with the first, the second and the header of the third parameters as parameters.
  - Else it returns with the result of the add_to_nth/4 function with the first three parameters and the received position as parameters.

add_simple_template/3

Description The function adds the identifier of the variable to the current list and returns the updated list.

The parameters:

1. Name: The name of the variable.
2. Id: The identifier of the variable.
3. List: List of tuples (variable names, variable identifiers, list of identifiers of the variable).

This function is used in add_template/4 functions.

Example: `add_simple_template(Name,Id,X)`

Implementation The function adds the identifier parameter to the list of the third parameter The function uses a pattern matching to the third parameter:

1. When the third parameter is an empty list the function creates a list with one element. The element is a tuple with three elements: Name, Id, [Id]. The Name is the first parameter and the Id is the second.
2. When the third element is not an empty list then a further separation is needed because every element of the List in the third element is belongs to a variable. Name (every variable name has own list) and the identifier have to be added to the correct variable name's list. The function search the correct variable's name list from the header to the end of the list by elements. This separation is coded by case structure which analyses a logical expression on the equivalence of the first parameter(Name) and the first element of the header of the third parameter.
   - If it is true then this is the correct list. The function updates the header with added the identifier (second parameter) to the third element of the header, and returns with the updated list.
   - Else the function updates the tail of the list by calling recursively itself to the tail of the third parameter, and returns with the updated list.

add_sh_check/4

Description The function adds the identifier of the variable to the current list and returns the updated list.

The parameters:

1. Name: The name of the variable.
2. Id: The identifier of the variable.

3. List: List of list of tuples (variable names, variable identifiers, list of identifiers of the variable).
4. N: Number of list elements (integer)

This function is used in visibility/3 function.

Example: `add_sh_check(Name,Id,List,N)`

Implementation The function adds the identifier (second) parameter to the header of the list of the third parameter with add_simple function with the first, second and the header of the third parameter as parameters.

put_in/2

Description The function starts the putting into the database a variable visibility informations.

The parameters:

1. Pid : The pid of the spawn connection in visib_fun_expr_clause/5
2. Node : The node of the variable

This function is used in visib_clause/3 and preorder_2/6 function.

Example: `put_in(Pid, Element)`

Implementation
- At first it sends to the server the following four elements tuple:
    1. The add command
    2. Name of the variable getting by erl_syntax:variable_name/1 function with the second parameter as parameter
    3. The identifier of the variable getting by the header of the result of get_ann/1 function with the second parameter as parameter
    4. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with this.

put_in_sh_check/2

Description The function starts the putting into the database a variable visibility informations.

The parameters:

1. Pid : The pid of the spawn connection in visib_fun_expr_clause/5
2. Node : The node of the variable

This function is used in visib_fun_expr_clause/5 function.

Example: `put_in_sh_check(Pid, Element)`

Implementation

- At first it sends to the server the following four elements tuple:
  1. The add_sh_check command
  2. Name of the variable getting by erl_syntax:variable_name/1 function with the second parameter as parameter
  3. The identifier of the variable getting by the header of the result of get_ann/1 function with the second parameter as parameter
  4. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with this.

get_last_variables/1

Description The function gives the collected variable informations to the caller.

The parameters:

1. Pid : The pid of the spawn connection in visib_fun_expr_clause/5

This function is used in visib_clause/3, visib_list_comp/5 and visib_fun_expr_clause/4 functions.

Example: `get_last_variables(Pid)`

Implementation

- At first it sends to the server the following two elements tuple:
  1. The get_last command
  2. The own identifier by calling self/0 function.
- After it waits to receive an ok, integer tuple and returns with the integer. If the message was not like above the program throw a failure message by using io:format/2.

split/2

Description The function starts to split the connection.

The parameters:

1. Pid : The pid of the spawn connection in visib_fun_expr_clause/5
2. Mode : the mode of the server (normal, template, ...)

This function is used in visib_list_comp/5 and visib_fun_expr_clause/5 functions.

Example: `split(Pid, normal)`

Implementation

- At first it sends to the server the following three elements tuple:
  1. The split command
  2. The running mode, the second parameter.
  3. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with it.

stop/1

Description The function starts to stop the connection.

The parameters:

1. Pid : The pid of the spawn connection in visib_fun_expr_clause/5

This function is used in visib_clause/3 function.

Example: `stop(Pid)`

Implementation
- At first it sends to the server the following two elements tuple:
  1. The stop command
  2. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with it.

change_state/2

Description The function starts to split the connection.

The parameters:

1. Pid : The pid of the spawn connection in visib_fun_expr_clause/5
2. State : the state of the server (normal, reverse, ...)

This function is used in visib_generator/4 and visib_list_comp/4 functions.

Example: `change_state(Pid, reverse)`

Implementation
- At first it sends to the server the following three elements tuple:
  1. The change command
  2. The state, the second parameter.
  3. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with it.

get_true_pos/1

Description The function gives the position of the first true element in a list of booleans.

The parameter:

1. List: the list of booleans

This function is used in add/4 and add_template/4 function.

Example: `get_true_pos(Boollist)`

Implementation The function calls the get_true_pos/2 function with the parameter and 1 as parameters.

get_true_pos/2

Description The function gives the position of the first true element in a list of booleans by analysing every element in the list from the header to the tail until the first true element or the end.

The parameter:

1. List : The list of booleans
2. Position : The actual position.

This function is used in get_true_pos/1 and get_true_pos/2 functions.

Example: `get_true_pos(List,1)`

Implementation The function makes a pattern matching to the first parameter:

1. When the first parameter is empty then returns with the second parameter.
2. When the header of the first parameter is true, then returns with the second parameter.
3. Others the function calls itself recursively with the tail of the first element and the second element increased by one as parameters.

add_to_nth/4

Description The function adds the identifier of the variable to the current list and returns the updated list when there is same variable name in the lists as the first parameter

The parameters:

1. Name: The name of the variable.
2. Id: The identifier of the variable.
3. List: List of list of tuples (variable names, variable identifiers, list of identifiers of the variable).
4. N: Number of list elements (integer)

This function is used in add/4 function.

Example: `add_to_nth(Name,Id,Lists,Member)`

Implementation The function makes a pattern matching to the first parameter:

1. The function adds the identifier (second) parameter to the header of the list of the third parameter with add_simple function with the first, second and the header of the third parameter as parameters.
2. The function adds the identifier (second) parameter to the tail of the list of the third parameter with recursively calling itself function with the first, second, the tail of the third and the fourth decreased by one parameters as parameters.

add_to_nth_template/4

Description The function adds the identifier of the variable to the current list and returns the updated list when there is same variable name in the lists as the first parameter

The parameters:

1. Name: The name of the variable.
2. Id: The identifier of the variable.
3. List: List of list of tuples (variable names, variable identifiers, list of identifiers of the variable).
4. N: Number of list elements (integer)

This function is used in add_template/4 function.

Example: `add_to_nth_template(Name,Id,Lists,Member)`

Implementation The function makes a pattern matching to the first parameter:

1. The function adds the identifier (second) parameter to the header of the list of the third parameter with add_simple function with the first, second and the header of the third parameter as parameters.
2. The function adds the identifier (second) parameter to the tail of the list of the third parameter with recursively calling itself function with the first, second, the tail of the third and the fourth decreased by one parameters as parameters.

get_current_scope/1

Description The function starts to get the current scope.

The parameters:

1. Pid : The pid of the spawn connection in visib_clause/3

This function is used in put_into_scope/4 function.

Example: `ScopeId = get_current_scope(Pid)`

Implementation

- At first it sends to the server the following two elements tuple:
  1. The current command
  2. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with it.

get_scope_list/1

Description The function starts to get the scope list.

The parameters:

1. Pid : The pid of the spawn connection in visib_clause/3

This function is used in visib_clause/3 function.

Example: `get_scope_list(Pid)`

Implementation
- At first it sends to the server the following two elements tuple:
  1. The list command
  2. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with it.

new_scope/2

Description The function starts to introduces a new scope to the second parameter node.

The parameters:

1. Pid : The pid of the spawn connection in visib_clause/3
2. Node : The starting node of the scope.

This function is used in visib_list_comp/5 and visib_fun_expr_clause/5 function.

Example: `get_scope_list(Pid)`

Implementation
- At first it sends to the server the following three elements tuple:
  1. The new_scope command,
  2. The identifier of the second parameter (head of the get_ann (Node) function)
  3. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with it.

end_of_scope/1

Description The function starts to close the scope.

The parameters:

1. Pid : The pid of the spawn connection in visib_clause/3

This function is used in visib_list_comp/5 and visib_fun_expr_clause/5 function.

Example: `end_of_scope(Pid)`

Implementation
- At first it sends to the server the following two elements tuple:
  1. The end_scope command,
  2. The own identifier by calling self/0 function.
- After it waits to receive an integer and returns with it.

scope/2

Description The function is the server function to the process, which stores the scope informations to the database.

The parameters:

1. Scope : The list of identifiers. The head of the list id the current scope.
2. List : The list of the inner-outer scoping identifier pairs.

The function is used in the visib_clause/3 function where the server is started.

Example: `scope([ hd( get_ann( Clause ))],[ hd( get_ann( Clause )), hd( get_ann( Clause ))])`

Implementation The function receives the command and parameter(s) tuples and executes the current task.

1. When the tuple is a stop, pid pair: the function closes the connection sending ok to the pid.
2. When the tuple new_scope, node, pid: the function sends back an ok message and call itself with adding the node to the first parameters list (the current node will the current scope) and add the node and head of the first parameter pair to the second parameter (the current node is the inner node, and the head is the outer node).
3. When the tuple is end_scope, pid: the function sends back an ok message and call itself with the tail of the first parameter and the original second parameter (the current scope is ready, we just cut the head of the list of the scopes).
4. When the tuple is list, pid: the function sends back the collected inner-outer scope list. After it calls itself with the original parameters.
5. When the tuple is current, pid: the function sends back the head of the first parameter, the current scope. After it calls itself with the original parameters.

simultaneous_visiting/3

Description The function starts the simultaneous preorder traverse to synchronize the position informations of the nodes between the database and the source.

The parameters:

1. Ref : The reference of the ODBC connection.
2. Forms : The root of the parsered source code
3. File : The path of the current source.

This function is used in the set_positions/2 function.

Example: `simultaneous_visiting(Ref, Forms3, File)`

Implementation
- At first the function gets the identifier of the module which belongs to the current source file from the module table with a select SQL query. The path has to be the same as the converted third parameter. The conversion made by io_lib:write_string/1 function.
- After it gets the identifier of the form list of the module from the id_count table with a select SQL query. The identifier of the module has to be the same as the previous module identifier.
- After it the function starts the simultaneous traverse of the database and the syntax tree by calling simultaneous_preorder/5 function with the set_position/4 function as the parameter function.
- At the end the function executes a commit command to the database.

simultaneous_preorder/5

Description The function executes the simultaneous preorder traverse to synchronize the position informations of the nodes between the databa-se and the source.

The parameters:

1. F : The function which used to the remained parameters (with 4 parameters).
2. Ref : The reference of the ODBC connection.
3. MId : The identifier of the module.
4. Id : The identifier of the current node.
5. Tree : The tree of the current node in the syntax tree.

This function is used in the simultaneous_visiting/3 function.

Example: `simultaneous_preorder(fun set_position/4, Ref, MId, FormListId, Forms)`

Implementation

- At first the function execute the function in the first parameter to the remaining parameters.
- The function uses a case structure to the result list of the subtrees/1 unction to the last parameter:
  1. When the list is empty, the function returns with ok.
  2. When the list contains elements then the function zips the following lists to a joined list with lists:zip function:
     - The flattened subtree list of the syntax tree by lists:flatten/1 function
     - The flattened list of the result of the erl_syntax_db:subtrees/3 (the same subtrees in the database) by lists:flatten function.
       After it the function uses recursively this simultaneous traverse to the elements of the joined list inside a lists:map function.

set_position/4

Description The function updates the position of a syntax tree node in the database.

The parameters:

1. Ref : The reference of the ODBC connection.
2. MId : The identifier of the module.
3. Id : The identifier of the current node.
4. Node : The current node in the syntax tree.

This function is used in the simultaneous_visiting/3 function.

Example: `(fun set_position/4, Ref, MId, FormListId, Forms)`

Implementation

- At first the function gets the position of the node from the source by calling erl_syntax:get_pos/1 function with the last parameter.
- The function uses a case structure to the result of the is_tuple/1 function:
  1. When the return value is true, the position was a tuple with two element: the first is the line information and the second is the column information.
     The function updates the pos table to the new line and column values. The identifiers has to be the same as the converted second and third parameters.
  2. When the result was not a tuple, the function returns with ok.

3.5 Module out_from_db

3.5.1 Module description

The module builds up the abstract syntax tree and the source code from the database.The principle is that the root of every syntax tree is a form_list node, and every node has its own children's identifiers, and every node_type has constructor functions. The only problems are the pre- and postcomments, which are not in the syntax tree, they need separate methods to give them back to the code.

Exported function: create_code/3, create_code/2

3.5.2 Functions

Exported functions

create_code/3

Description The function recovers the source code of the current file and writes it out to a file.

The parameters:

1. Ref : The reference id of the ODBC connection. (integer)
2. File : The path of the file which we want to recover from the database.
3. OutFile : The output file which will contain the rebuilt source code.

This function is used in the d_client module when it executes the refactor steps or updates the database.

Example: `Result = out_from_db:create_code(Con, File, File)`

Implementation

- At first the function get the module identifier of the original file (second parameter) from the module table with a select SQL query. The path is the same as the converted second parameter. The conversion made by io_lib:write_string/1 function.
- After it the function check in a case structure if the module identifier is an empty list or not.
  1. When it is empty list, it means that the file is not stored in the database. The function returns with a not_exists error message and the file as parameter.
  2. When it is not empty the function get the identifier of the form list in the module from id_count table with a select SQL query. The module identifier has to be the same as the converted head of the list. The conversion is made with the integer_to_list/1 function.
     After it the function calls the create_file/2 function with the following parameters:
     (a) The return value of the build_tree/3 function,
     (b) The path of the output file (last parameter).

create_code/2

Description The function recovers the source code of the current module and write it out to the original file.

The parameters:

1. Ref : The reference id of the ODBC connection. (integer)
2. MId : The identifier of the module.

This function is used in the refac_ren_fun module after renaming function names not only the current module, for example in import lists.

Example: `out_from_db:create_code(Ref, MId)`

Implementation
- At first the function get the identifier of the form list in the module from id_count table with a select SQL query. The module identifier has to be the same as the converted second parameter. The conversion is made with the integer_to_list/1 function.
- After it the function gets the path of the module's file from the module table with a select SQL query. The module identifier has to be the same as the converted second parameter. The conversion is made with the integer_to_list/1 function.
- After it the function calls the create_file/2 function with the following parameters:
  1. The return value of the build_tree/3 function,
  2. The path of the output file (last parameter).

Local functions

build_tree/3

Description The function recovers the source code and writes it out to a file.

The parameters:

1. Ref : The reference id of the ODBC connection. (integer)
2. Id : The identifier of the root of the syntax-tree, a form list identifier. (integer)
3. MId : The identifier of the current module.

This function is used in the create_code/3 and create_code/2 functions.

Example: `build_tree( Ref, FormLId, MId)`

Implementation
- At first the functions gets every rows of the form_list table with a select SQL query. The identifier is the same as the second parameter, the module identifier id the same as the third parameter. When not only one module is in the database it can be possible to get more rows as result as a list. The root of the syntax-tree is always a form_list node.

- After it the function builds up the syntax-trees to every element in the list by using build_element/3 function with the connection identifier (first parameter) and the fourth element of the current element of the list as parameters. The elements of the list is getting with lists:map/2 function.
- At the end the function uses an erl_syntax:form_list/1 to the trees. This solution is not guarantees the unchangingness of the original source code. To reach this the tool should have an own prettypinter.

create_file/2

Description The function recovers the source code and writes it out to a file.

The parameters:

1. Tree : The syntax tree.
2. Outfile : The path of the output file.

This function is used in the create_code/3 and create_code/2 functions.

Example: `create_file( build_tree( Ref, FormLId, MId ), OutFile)`

Implementation

- At first the function opens the output file (second parameter) to write by calling file:open/2 function with the second parameter and write atom in a list as parameters. The result of the function is a tuple which second element is the identifier of the opened io connection.
- After it writes out the trees to the opened file by calling io:put_chars/2 function with the io identifier and the formatted trees. This solution is not guarantees the unchangingness of the original source code. To reach this the tool should have an own prettypinter.
- After it the function closes the file by calling file:close/1 function with the io identifier as parameter.
- At the end the function returns with an {ok,OutFile} tuple.

add_precomment/6

Description The function adds precomment to the node. It will use erl_syntax:add_precomments,but the precomment can be list of comments so we have to create this list first.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. Node : The selected node.
3. Id : The identifier of the node.

4. MId : The identifier of the current module.
5. Num : The number of the remaining comments, witch have to added to the node.
6. Comments : The text of the comments.

This function is used in the give_pp_comments/4 function.

Example: `add_precomment( Ref, Node, Id, MId, length(Precomm), [])`

Implementation The function makes a pattern matching to the fifth parameter (how many remaining comments are):

1. If Num=0 we call the erl_syntax:add_precomments/2 function with our list of the comment nodes and with the Node as second parameter.
2. If Num>0 then the function makes the following commands:
   - Selects the padding informations and the comments from the precomment table of the database with select SQL queries where the identifier is equal with the Id parameter, and the pos is equal with the Num parameter. The data in the argument column is padding information, when the qualifier value is 0, and else it is comment.
   - There are two variation of erl_syntax:comment: one when we have padding, and one when we haven't got padding. The function separates these by a case structure:
     - If we haven't got padding (it is an empty list), the we select the first branch in the case, we create the comment node without padding.
     - If we have padding, the we have to retrieve the integer value of the Padding from the list of tuples by calling list_to_integer/1 function to the first element of the list of the Padding. This integer will be the first parameter of the second variation of erl_syntax:comment function, and the second parameter will be the comment. The comment is getting with a lists:map/2 function from the comment list.
   - After we created the comment node we call this function again with the same Ref, Node and Id, MId but reduced by one the Num parameter, and concatenated the comments parameter with the new comment node.

add_postcomment/6

Description The function adds postcomment to the node. It will use erl_syntax:add_postcomments,but the postcomment can be list of comments so we have to create this list first.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. Node : The selected node.
3. Id : The identifier of the node.
4. MId : The identifier of the current module.
5. Num : The number of the remaining comments, witch have to added to the node.
6. Comments : The text of the comments.

This function is used in the give_pp_comments/4 function.

Example: `add_postcomment(Ref, Node2, Id, MId, length( Postcomm), [])`

Implementation The function makes a pattern matching to the fifth parameter (how many remaining comments are):

1. If Num=0 we call the erl_syntax:add_postcomments/2 function with our list of the comment nodes and with the Node as second parameter.
2. If Num>0 then the function makes the following commands:
   - Selects the padding informations and the comments from the postcomment table of the database with select SQL queries where the identifier is equal with the Id parameter, and the pos is equal with the Num parameter. The data in the argument column is padding information, when the qualifier value is 0, and else it is comment.
   - There are two variation of erl_syntax:comment: one when we have padding, and one when we haven't got padding. The function separates these by a case structure:
     – If we haven't got padding (it is an empty list), the we select the first branch in the case, we create the comment node without padding.
     – If we have padding, the we have to retrieve the integer value of the Padding from the list of tuples by calling list_to_integer/1 function to the first element of the list of the Padding. This integer will be the first parameter of the second variation of erl_syntax:comment function, and the second parameter will be the comment. The comment is getting with a lists:map/2 function from the comment list.
   - After we created the comment node we call this function again with the same Ref, Node and Id, MId but reduced by one the Num parameter, and concatenated the comments parameter with the new comment node.

build_element/3

Description This function picks the type of the node of the Id using the node_type table of the database, and after it calls the give_pp_comments function with the needed node constructor function calling in the second parameter. The parameters:

1. Ref : The reference id of the ODBC connection.
2. Id : The identifier of the node.
3. MId : The identifier of the current module.

This function is used in the build_tree/3 function to builds up the syntax tree.

Example: `build_element(Ref, element(4, Element), MId)`

Implementation

- At first the function queries the type id from the node_type table of the database with a select SQL query. The id in the table is equal to the Id parameter and the module identifier is the same as the third parameter. The results third elements will be list of tuples. The tuple's second element will be the type of the node.
- After it in a case structure the function calls give_pp_comments function with the following three parameters:
  1. the reference id of the ODBC connection,
  2. a node, which is the output of the constructor function of the selected node type,
  3. the identifier of the node,
  4. the identifier of the module.

  The 50 branches are the same as node tables in the database, and node types in the abstract syntax tree of Erlang language.

give_pp_comments/4

Description The function extends the node with its pre- and postcomments. The parameters:

1. Ref : The reference id of the ODBC connection.
2. Node : The node without its pre- and postcomments.
3. Id : The identifier of the node.
4. MId : The identifier of the current module.

This function is used in the build_element/3 function.

Example: `give_pp_comments( Ref, make_module_qualifier( Ref, Id), Id, MId)`

Implementation
- This function queries the pre- and postcomments of the node from the pre- and postcomment tables of the database with select SQL queries where identifiers are equals with the third and fourth parameters.
- After it creates a new node (Node2) with the precomments (calling add_precomment/5)
- After it creates the ready node with postcomments too (calling add_postcomments/5). The Num(fourth) parameters of the adding functions will be the length of the result lists of the data-base queries, and the Comments parameters is at the beginning an empty list.

Function group make_*/3

Description These functions call the needed constructor function of the syntax tree, and gives the position informations to the node, and if there any children of the node they queries the children from the database and call the build_element functions to them. The list of the correct constructor functions are in the nodes appendix.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. Id : The identifier of the node.
3. MId : The identifier of the current module.

This functions are used in the build_element/3 function.

Example: `make_function(Ref, Id, MId)`

The make_* functions:

1. make_application
2. make_arity_qualifier
3. make_atom
4. make_attribute
5. make_binary
6. make_binary_field
7. make_block_expr
8. make_case_expr
9. make_catch_expr
10. make_char
11. make_class_qualifier
12. make_clause
13. make_comment

14. make_cond_expr
15. make_conjunction
16. make_disjunction
17. make_eof_marker
18. make_float
19. make_form_list
20. make_fun_expr
21. make_function
22. make_generator
23. make_if_expr
24. make_implicit_fun
25. make_infix_expr
26. make_integer
27. make_list
28. make_list_comp
29. make_macro
30. make_match_expr
31. make_module_qualifier
32. make_nil
33. make_operator
34. make_parentheses
35. make_prefix_expr
36. make_qualified_name
37. make_query_expr
38. make_receive_expr
39. make_record_access
40. make_record_expr
41. make_record_field
42. make_record_index_expr
43. make_rule
44. make_size_qualifier
45. make_string
46. make_text
47. make_try_expr
48. make_tuple

49. make_underscore
50. make_variable

Implementation Every node needs the position information, so the functions query it from the pos table with a select SQL query where identifiers is equal with the second and third parameters.

- If the node has not got children - for example atom -, the functions queries the needed parameter of the constructor function from the database and calls the constructor, and gives the position information to the result node of the previous constructor function.
- If the node has simple children (one child is one node, not list of nodes) - for example arity_qualifier -, the functions queries id-s of the needed children, and put them into a list of tuples, and it calls the constructor function with the returned nodes of build_elements functions, which called with the children node id-s. At the end it gives the position information.
- If the node has complex children (one or more children is a list of nodes) - for example applications -the function queries id-s of the needed children, and put them into a list of tuples, and it calls the constructor function with the returned nodes of build_elements functions for the simple children. The function calls for the complex children the build_element function inside a lists:map/2 function to get all of the nodes. At the end it gives the position information.
- If the node has optional children - for example clause-, the constructor function is putted into a case structure, which two bran-ches represent that the node has the optional parameter actually or not. If it has not got the parameter is an empty list.

3.6 Module refactor

3.6.1 Module description

The module contains functions which are used usually in more refactor steps.

Exported functions:

- get_module_names/1
- get_module_name/2
- get_module_name_if_exists_in_module/4
- get_import_list_ids/2
- get_imported_functions/3
- get_export_list/2

- get_export_list_id/2
- get_arity_qualifier_data/3
- add_element_to_tuples/2
- simple_member_b/3

3.6.2 Functions

Exported functions

get_module_names/1

Description The function collects every module name from the database and returns with the list.

The parameter:

1. Ref : The reference id of the ODBC connection.

The function is used in into_db:put_function_calls_in_db/1 function.

Example: `ModulesInDbase = refactor:get_module_names(Ref)`

Implementation
- At first the function collects the module identifiers from the module table with a select SQL query.
- After it the function collects the module names inside a lists:map/2 function to the module identifiers list. The embedded function gets the name of the current element (module) by calling get_module_name/2 function.
- At the end the function returns with the module name list.

get_module_name/2

Description The function get the name of the current module an returns with it.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. MId : The identifier of the module.

The function is used in the

- get_module_names/1
- into_db:get_fun_datas/1

functions.

Example: `ModuleName = refactor:get_module_name( Ref, MId)`

Implementation The function gets the name of the module from the form_ list, node_type, attribute and name tables with a select SQL query. The module identifiers have to be the same in all table and with the converted second parameter. The identifier in the node_type table have to be the same as the form in the form_list table. The type in the node_type table has to be 4 (attribute). The identifier in the attribute_ table has to be the same as the form. The identifier in the name table has to be the same as the argument. The position in the attribute_ table has to be 1. The identifier in the attribute_ table has to be between the following identifiers: the argument in the attribute_table has to be the same as the identifier in the name table and the name has to be "module".

The function returns with the module name.

get_module_name_if_exists_in_module/4

Description The function gets the name of the module, if the current function (identified by the last two parameter) is exists in the module. If the function is exists in the module the return value is the name of the module, else false.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. MId : The identifier of the module.
3. Name : The name of the function.
4. Arity : The arity of the function.

The function is used in the into_db:get_module_name/4 function.

Example: `case refactor:get_module_name_if_exists_in_module( Ref, MId, Name, Arity) of`

Implementation

- At first the function gets the module identifier where the current function exists from the function, fun_visib, name tables with a select SQL query. The module identifiers have to be the same in all tables and with the converted second parameter. The argument in the fun_visib table has to be the same as the converted arity (fourth parameter) and the position has to be 0 in the fun_visib table. The identifiers have to be the same in the fun_visib and function tables. The clause in the function table has to be the same as the identifier in the name table and the position has to be 0 in the function table. The name in the name table has to be the same as the third parameter.
- After it the function checks in a case structure if the module identifier list is empty:
  1. When the list is empty, it means that the current function is not exists in the module: the return value is false.

2. When the list is not empty the function gets the module name by calling get_module_name/2 and returns with the name.

get_import_list_ids/2

Description The function get the identifiers of the import lists into a list and returns with it.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. MId : The identifier of the module.

The function is used in the into_db:is_imported/4 function.

Example: `ImportListIds = refactor:get_import_list_ids( Ref, MId)`

Implementation The function get the identifiers from the form_list, node_ type, attribute_ tables with a select SQL query. The module identifiers have to be the same in all tables and with the converted second parameter. The identifier in the node_type table has to be the same as the form in the form_list table and the type in the node_type table has to be 4 (attribute). The identifier in the attribute_ table has to be the same as the form. The identifier in the attribute_table has to be between the following identifiers: name of the argument in the attribute_ table is "import".

get_imported_functions/3

Description The function gets the informations (module name, function name, arity) of the imported functions to a list.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. MId : The identifier of the module.
3. ImportListId: The list of the identifiers of the imported functions.

The function is used in into_db:is_imported/4 function.

Example: `ImportedFunctions = refactor:get_imported_functions ( Ref, MId, ImportListIds)`

Implementation

- At first the function gets the datas of the imported functions to a list inside a lists:map/2 function to the last parameter (identifiers of imported functions). The embedded functions are the following:

  - The function gets the name of the module from the name and attribute_ tables with a select SQL query. The module identifiers are the same in the two table and with the converted second parameter. The identifier in the name table is the same as the argument in the attribute_ table and the position is 1. The identifier in the attribute_table is the same as the converted current element (function identifier).
  - The function gets the identifiers (body, arity) of the arity qualifiers from the arity_qualifier, attribute_, list tables. The module identifiers are the same in the tables and with the converted second parameter. The argument in the attribute_table is the same as the identifier in the list table. The element in the list table is the same as the identifier in the arity_qualifier table. The identifier in the attribute_ table is the same as the converted current element (function identifier).
  - The function gets the list which contains the name and the arity of the functions by calling get_arity_qualifier_data/3 function.
  - The function add the module name into the result tuple by add_element_to_tuple/2 function.
- At the end the function uses a lists:flatten/1 function to the list and returns with the result.

get_export_list/2

Description The function collects the module name, function name and arity informations to a list from the exported functions.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. MId : The identifier of the module.

The function is used in into_db:fun_is_exported/4 function.

Example: `ExportList = refactor:get_export_list( Ref, MId)`

Implementation
- The function collects the identifiers of the export list elements in the module by calling get_export_list_id/2 function.
- The function collects the module name, function name and arity informations of the exported functions by calling get_arity_qualifier_data/3. The function returns with the result of the last function calling.

get_export_list_id/2

Description The function collects the identifiers of the function body and arity to a list from the exported functions.

The parameters:

1. Ref : The reference id of the ODBC connection.
2. MId : The identifier of the module.

The function is used in get_export_list/2 function.

Example: `ExportListIds = refactor:get_export_list_id( Ref, MId))`

Implementation

- The function collects the identifiers of the export list elements from the form_list, node_type, attribute_, list tables with a select SQL query. The module identifiers are the same in the tables and with the converted second parameter. The identifier in the node_type table is the same as the form in the form_list table and the type in the node_type is 4 (attribute). The identifier in the attribute_ table is the same as the form and the position is 1. The identifier in the attribute_table has to be between the following identifiers: name of the argument in the attribute_ table is "export".
- The function collects the body and arity identifiers inside a lists: map/2 functions to the export list element identifiers and returns with the new list. The embedded function get the identifiers from the arity_qualifier table with a select SQL query. The module identifier is the same as the converted second parameter. The identifier is the same as the converted current element (element identifier).

simple_member_b/3

Description The function checks if the current function (defined in the first two parameter) is element of the list in the third parameter. The function returns with a boolean.

The parameters:

1. FunName : The name of the current function
2. FunArity : The arity of the current function
3. List : The list which contains functions

The function is used in into_db:fun_is_exported/4 functions.

Example: `IsExported = refactor:simple_member_b( FunName, Arity, ExportList)`

Implementation The function uses a pattern matching to the last parameter:

1. When the list is empty the function return with false value.
2. When the header of the list is the same as the first two element, the function returns with true value.
3. Else the function calls itself recursively with the same parameters just the third parameter is the tail of the original third parameter.

Local functions

get_arity_qualifier_data/3

Description The function gets the name and the arity list from the body and arity identifier list.

1. Ref : The reference id of the ODBC connection.
2. MId : The identifier of the module.
3. ArityQualifierIds : The list of the identifier pairs - body and arity.

The function is used in get_export_list/2 function.

Example: `ExportList = refactor:get_arity_qualifier_data( Ref, MId, ExportListIds)`

Implementation The function get the name and the arity value from their identifiers inside a lists:map function to the last parameter.

- The function gets the name of the function first from the name table with a select SQL query. The module identifier is the same as the converted second parameter. The identifier is the same as the first element of the current element (body, arity identifier tuple).
- The function gets the arity of the function from the integer_ table with a select SQL query. The module identifier is the same as the converted second parameter. The identifier is the same as the second element of the current element (body, arity identifier tuple).

add_element_to_tuple/2

Description The function add the first parameter to the tuple in every element in the second parameter and returns with the result list.

The parameters:

1. Element : The element which has to be added to the tuples.
2. List : List of tuples.

The function is used get_imported_functions/3 function.

Example: `ImportDatas = refactor:add_element_to_tuples( ModuleName, ArityQualifierDatas)`

Implementation The function uses a pattern matching to the second parameter:

1. When the second parameter is an empty list, the function return with empty list.

Length of the code (byte)	Whole time (sec)	Put into time (sec)	Recover time (sec)
4925	2.76600	1.32800	1.43800
8949	2.17200	1.25000	0.922000
6617	3.84400	2.45300	1.39100
1709	0.313000	0.172000	0.141000
7504	1.34300	0.765000	0.578000
466	0.844000	0.656000	0.188000
22402	9.07800	5.15600	3.92200
2160	0.234000	0.141000	9.30000e-2
43587	22.4840	11.4220	11.0620
4299	2.15700	1.06300	1.09400
2511	0.546000	0.390000	0.156000
2983	1.57800	0.781000	0.797000
3083	0.750000	0.500000	0.250000

Tab. 3.1: Measure times when the database is initialised after every module (part 1)

2. Else the function concatenates the first parameter with the converted head of the second parameter. The conversion is made by tuple_to_list/1 function. At the end the function convert the concatenated list to tuple and this will be the head of the new list. The function call itself recursively with the first parameter and the tail of the second parameter and the result will be the new tail of the list.

3.7 Testing procedures and result

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

We tested the tool in a big project (approximately 90 modules and 1 MB Erlang source code) to test the tool for 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 got the following results for individual modules (see Tables 3.1, 3.2, 3.3, 3.4):

Length of the code (byte)	Whole time (sec)	Put into time (sec)	Recover time (sec)
1518	0.265000	0.140000	0.125000
8490	2.23400	1.23400	1.00000
4250	1.93700	0.937000	1.00000
10387	3.82800	1.93700	1.89100
6368	2.57800	1.15600	1.42200
1914	0.157000	9.40000e-2	6.30000e-2
2448	0.562000	0.265000	0.297000
4305	0.532000	0.313000	0.219000
1723	0.265000	0.156000	0.109000
5696	3.01600	1.50000	1.51600
1619	0.266000	0.141000	0.125000
3510	0.938000	0.469000	0.469000
155	0.140000	7.80000e-2	6.20000e-2
2781	0.531000	0.297000	0.234000
18043	7.21900	3.98400	3.23500
1470	0.265000	0.140000	0.125000
95	0.125000	7.80000e-2	4.70000e-2
8826	5.81200	2.92100	2.89100
761	0.375000	0.187000	0.188000
83439	64.1560	34.3430	29.8130
8285	7.37500	4.14000	3.23500
7911	4.75000	2.37500	2.37500
2190	0.938000	0.484000	0.454000
73565	63.9840	40.2810	23.7030
2379	2.03100	1.14000	0.891000
808	0.281000	0.156000	0.125000
351	0.156000	0.125000	3.10000e-2
1465	0.453000	0.235000	0.218000
3465	2.03100	1.09400	0.937000
4516	0.890000	0.421000	0.469000
4299	2.45300	1.32800	1.12500
2651	1.84400	1.00000	0.844000

Tab. 3.2: Measure times when the database is initialised after every module (part 2)

Length of the code (byte)	Whole time (sec)	Put into time (sec)	Recover time (sec)
3444	2.26500	1.12500	1.14000
12989	5.11000	2.64100	2.46900
1891	0.250000	0.125000	0.125000
4453	0.594000	0.313000	0.281000
3360	0.562000	0.296000	0.266000
8006	2.28100	1.23500	1.04600
1623	0.297000	0.157000	0.140000
5449	1.35900	0.656000	0.703000
2452	0.453000	0.219000	0.234000
26488	14.3590	7.76600	6.59300
26307	12.0630	6.28200	5.78100
4002	1.04700	0.516000	0.531000
3290	2.15700	1.03200	1.12500
51480	38.2180	20.1720	18.0460
1538	0.625000	0.328000	0.297000
499	0.203000	0.125000	7.80000e-2
11493	7.26500	3.60900	3.65600
15379	9.67200	4.78200	4.89000
38053	27.5940	13.5470	14.0470
5757	3.32800	1.64100	1.68700
10373	6.68800	3.53200	3.15600
7336	6.00000	2.98500	3.01500
8133	2.95300	1.48400	1.46900
12894	8.09400	4.48500	3.60900
1465	0.765000	0.421000	0.344000
10284	4.95300	2.32800	2.62500
8492	4.01500	1.82800	2.18700
12678	7.42200	3.73500	3.68700

Tab. 3.3: Measure times when the database is initialised after every module (part 3)

Length of the code (byte)	Whole time (sec)	Put into time (sec)	Recover time (sec)
9711	3.62500	1.81300	1.81200
17730	7.82800	4.29700	3.53100
6171	2.65600	1.32800	1.32800
4082	1.78100	0.875000	0.906000
286	0.234000	0.141000	9.30000e-2
7295	2.61000	1.36000	1.25000
2307	2.31200	1.25000	1.06200
408	0.281000	0.172000	0.109000
9807	5.67100	2.64000	3.03100
617	0.250000	0.141000	0.109000

Tab. 3.4: Measure times when the database is initialised after every module (part 4)

We got the following results for the whole system, when the database is initialised just at the beginning. The main difference can be seen in the parse time, when a big multi module system is already in the database, the calculating time of the semantic information (connections between the modules) is growing (see Tables 3.5, 3.6, 3.7, 3.8):

Length of the code (byte)	Whole time (sec)	Put into time (sec)	Recover time (sec)
4925	3.93800	2.17200	1.76600
8949	2.53100	1.56200	0.969000
6617	3.21900	1.95300	1.26600
1709	1.12500	0.983999	0.141000
7504	2.21900	1.61000	0.609000
466	1.45300	1.25000	0.203000
22402	10.4690	6.56300	3.90600
2160	2.81200	2.70300	0.109000
43587	27.5470	15.2820	12.2650
4299	8.15600	6.89100	1.26500
2511	5.96900	5.81300	0.156000
2983	7.57800	6.59400	0.984000
3083	6.51600	6.23500	0.281000
1518	6.40600	6.26600	0.140000
8490	9.92200	8.84400	1.07800
4250	8.26600	7.15600	1.11000
10387	11.2500	9.15600	2.09400
6368	8.82900	7.57900	1.25000
1914	6.73400	6.65600	7.80000e-2
2448	7.12500	6.79700	0.328000
4305	7.11000	6.86000	0.250000
1723	6.82800	6.70300	0.125000

Tab. 3.5: Measure times when the database is initialised just at the beginning (part 1)

Length of the code (byte)	Whole time (sec)	Put into time (sec)	Recover time (sec)
5696	10.4220	8.70300	1.71900
1619	7.75000	7.60900	0.141000
3510	8.29700	7.76600	0.531000
155	7.46900	7.39100	7.80000e-2
2781	7.87500	7.60900	0.266000
18043	14.9680	11.4060	3.56200
1470	8.11000	7.96900	0.141000
95	8.12500	8.07800	4.70000e-2
8826	15.5470	12.2650	3.28200
761	9.12500	8.90600	0.219000
83439	81.6100	49.3600	32.2500
8285	27.0780	23.6560	3.42200
7911	25.4530	22.8900	2.56300
2190	21.2820	20.7820	0.500000
73565	99.1720	74.0000	25.1720
2379	49.6710	48.2650	1.40600
808	65.3600	65.1250	0.235000
351	54.8440	54.7650	7.90000e-2
1465	54.2650	54.0000	0.265000
3465	52.6250	51.1720	1.45300
4516	51.1720	50.5000	0.672000
4299	52.7810	51.3280	1.45300
2651	45.1880	44.3130	0.875000
3444	45.6090	44.2650	1.34400
12989	70.0620	66.4690	3.59300

Tab. 3.6: Measure times when the database is initialised just at the beginning (part 2)

Length of the code (byte)	Whole time (sec)	Put into time (sec)	Recover time (sec)
1891	51.1880	51.0780	0.110000
4453	48.3590	47.9370	0.422000
3360	53.6720	53.2810	0.391000
8006	50.7970	49.5940	1.20300
1623	45.6090	45.4530	0.156000
5449	51.6100	50.9220	0.688000
2452	47.3750	47.0930	0.282000
26488	65.8910	57.9380	7.95300
26307	66.7190	59.6410	7.07800
4002	54.1090	53.4680	0.641000
3290	58.5620	57.3280	1.23400
51480	99.5160	79.3750	20.1410
1538	57.1090	56.5940	0.515000
499	60.6410	60.5320	0.109000
11493	72.4840	67.2340	5.25000
15379	76.6090	71.5310	5.07800
38053	84.6100	70.1250	14.4850
5757	63.5470	61.7340	1.81300
10373	67.0000	63.4370	3.56300
7336	66.3590	63.0930	3.26600
8133	62.2970	60.6720	1.62500
12894	70.6720	66.1410	4.53100
1465	70.8900	70.4840	0.406000
10284	75.7190	72.4380	3.28100
8492	71.8750	69.3750	2.50000

Tab. 3.7: Measure times when the database is initialised just at the beginning (part 3)

Length of the code (byte)	Whole time (sec)	Put into time (sec)	Recover time (sec)
12678	78.6720	74.2190	4.45300
9711	77.6560	75.3910	2.26500
17730	76.6720	73.0160	3.65600
6171	63.2190	61.8910	1.32800
4082	73.7970	72.6720	1.12500
286	77.5000	77.2030	0.297000
7295	66.9370	65.6720	1.26500
2307	67.7970	66.8280	0.969000
389	70.4690	70.2040	0.265000
9328	71.3750	68.3750	3.00000
605	67.8290	67.7040	0.125000

Tab. 3.8: Measure times when the database is initialised just at the beginning (part 4)

4. APPENDIX

4.1 Denied names

4.1.1 Auto-imported BIFs

abs/1	abs(Number) -> int() | float()
apply/2	apply(Fun, Args) -> term() | empty()
apply/3	apply(Module, Function, Args)
atom_to_list/1	atom_to_list(Atom) -> string()
binary_to_list/1	binary_to_list(Binary) -> [char()]
binary_to_list/3	binary_to_list(Binary, Start, Stop) -> [char()]
binary_to_term/1	binary_to_term(Binary) -> term()
check_process_code/2	check_process_code(Pid, Module) -> bool()
concat_binary/1	concat_binary(ListOfBinaries)
date/0	date() -> Year, Month, Day
delete_module/1	delete_module(Module) -> true | undefined
disconnect_node/1	disconnect_node(Node) -> bool() | ignored
element/2	element(N, Tuple) -> term()
erase/1	erase() -> [Key, Val]
exit/1	exit(Reason)
exit/2	exit(Pid, Reason) -> true
float/1	float(Number) -> float()
float_to_list/1	float_to_list(Float) -> string()
garbage_collect/0	garbage_collect() -> true
garbage_collect/1	garbage_collect(Pid) -> bool()
get/0	get() -> [Key, Val]
get/1	get(Key) -> Val | undefined
get_keys/1	get_keys(Val) -> [Key]
group_leader/0	group_leader() -> GroupLeader
group_leader/2	group_leader(GroupLeader, Pid) -> true
halt/0	halt()

halt/1	halt(Status)
hd/1	hd(List) -> term()
integer_to_list/1	integer_to_list(Integer) -> string()
iolist_to_binary/1	iolist_to_binary(IoListOrBinary) -> binary()
iolist_size/1	iolist_size(Item) -> int()
is_alive/0	is_alive() -> bool()
is_atom/1	is_atom(Term) -> bool()
is_binary/1	is_binary(Term) -> bool()
is_boolean/1	is_boolean(Term) -> bool()
is_float/1	is_float(Term) -> bool()
is_function/1	is_function(Term) -> bool()
is_function/2	is_function(Term, Arity) -> bool()
is_integer/1	is_integer(Term) -> bool()
is_list/1	is_list(Term) -> bool()
is_number/1	is_number(Term) -> bool()
is_pid/1	is_pid(Term) -> bool()
is_port/1	is_port(Term) -> bool()
is_process_alive/1	is_process_alive(Pid) -> bool()
is_record/2	is_record(Term, RecordTag) -> bool()
is_record/3	is_record(Term, RecordTag, Size) -> bool()
is_reference/1	is_reference(Term) -> bool()
is_tuple/1	is_tuple(Term) -> bool()
length/1	length(List) -> int()
link/1	link(Pid) -> true
list_to_atom/1	list_to_atom(String) -> atom()
list_to_binary/1	list_to_binary(IoList) -> binary()
list_to_existing_atom/1	list_to_existing_atom(String) -> atom()
list_to_float/1	list_to_float(String) -> float()

list_to_integer/1	list_to_integer(String) -> int()
list_to_pid/1	list_to_pid(String) -> pid()
list_to_tuple/1	list_to_tuple(List) -> tuple()
load_module/2	load_module(Module, Binary) -> module, Module | error, Reason
make_ref/0	make_ref() -> ref()
module_loaded/1	module_loaded(Module) -> bool()
monitor_node/2	monitor_node(Node, Flag) -> true
node/0	node() -> Node
node/1	node(Arg) -> Node
nodes/0	nodes() -> Nodes
nodes/1	nodes(Arg | [Arg]) -> Nodes
now/0	now() -> MegaSecs, Secs, MicroSecs
open_port/2	open_port(PortName, PortSettings) -> port()
pid_to_list/1	pid_to_list(Pid) -> string()
port_close/1	port_close(Port) -> true
port_command/2	port_command(Port, Data) -> true
port_connect/2	port_connect(Port, Pid) -> true
port_control/3	port_control(Port, Operation, Data) -> Res
pre_loaded/0	pre_loaded() -> [Module]
process_flag/2	process_flag(Flag, Value) -> OldValue
process_flag/3	process_flag(Pid, Flag, Value) -> OldValue
process_info/1	process_info(Pid) -> [Item, Info] | undefined
process_info/2	process_info(Pid, Item) -> Item, Info | undefined | []
processes/0	processes() -> [pid()]
purge_module/1	purge_module(Module) -> void()
put/2	put(Key, Val) -> OldVal | undefined
register/2	register(RegName, Pid | Port) -> true
registered/0	registered() -> [RegName]

round/1	round(Number) -> int()
self/0	self() -> pid()
setelement/3	setelement(Index, Tuple1, Value) -> Tuple2
size/1	size(Item) -> int()
spawn/1	spawn(Fun) -> pid()
spawn/2	spawn(Node, Fun) -> pid()
spawn/3	spawn(Module, Function, Args) -> pid()
spawn/4	spawn (Node, Module, Function, ArgumentList) -> pid()
spawn_link/1	spawn_link(Fun) -> pid()
spawn_link/2	spawn_link(Node, Fun) -> pid()
spawn_link/3	spawn_link(Module, Function, Args) -> pid()
spawn_link/4	spawn_link(Node, Module, Function, Args) -> pid()
spawn_opt/2	spawn_opt(Fun, [Option]) -> pid()
spawn_opt/3	spawn_opt(Node, Fun, [Option]) -> pid()
spawn_opt/4	spawn_opt (Module, Function, Args, [Option]) -> pid()
spawn_opt/5	spawn_opt(Node, Module, Function, Args, [Option]) -> pid()
split_binary/2	split_binary(Bin, Pos) -> Bin1, Bin2
statistics/1	statistics(Type) -> Res
term_to_binary/1	term_to_binary(Term) -> ext_binary()
term_to_binary/2	term_to_binary(Term, [Option]) -> ext_binary()
throw/1	throw(Any)
time/0	time() -> Hour, Minute, Second
tl/1	tl(List1) -> List2
trunc/1	trunc(Number) -> int()
tuple_to_list/1	tuple_to_list(Tuple) -> [term()]
unlink/1	unlink(Id) -> true
unregister/1	unregister(RegName) -> true
whereis/1	whereis (RegName) -> pid() | port() | undefined

4.1.2 RESERVED WORDS

after	and
andalso	band
begin	bnot
bor	bsl
bsr	bxor
case	catch
cond	div
end	fun
if	let
not	of
or	orelse
query	receive
rem	try
when	xor

Source: Kernel Reference Manual
Version 2.11

4.2 Structure of the database

4.2.1 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.

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		
binary	**mid**	**id**	**pos**	field	
binary_field	**mid**	**id**	**pos**	argument	
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	
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	
fun_call	**mid**	**id**	tmid	target	

Name of the table	**Col 1**	**Col 2**	**Col 3**	**Col 4**	**Col 5**
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		
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)		
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		

Name of the table	**Col 1**	**Col 2**	**Col 3**	**Col 4**	**Col 5**
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	value	
record_in-dex_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 255)		
text	**mid**	**id**	value (vchar 255)		
try_expr	**mid**	**id**	**pos**	**qualifier**	argument
tuple	**mid**	**id**	**pos**	element	
var_visib	**mid**	**id**	target		

LIST OF FIGURES

1.1 Source code of the example function clause. 8
1.2 A module containing and exporting a single function. 9
1.3 The AST of `gcd` (Part 1) 9
1.4 The AST of `gcd` (Part 2) 10
1.5 The AST of `gcd` (Part 3) 10
1.6 The AST of `gcd` (Part 4) 10
1.7 The AST of `gcd` (Part 5) 11

2.1 The Erlang menu in Emacs 16
2.2 Initialize the database 17
2.3 Getting the name of the Erlang node during the initialization . . 17
2.4 Into_db and reload from there 18
2.5 Check out latest version in the database 19

3.1 The Implementation Architecture 21

LIST OF TABLES

1.1 The representation of the code in Figure 1.1 in the database. . . 12

3.1 Measure times when the database is initialised after every module (part 1) 86
3.2 Measure times when the database is initialised after every module (part 2) 87
3.3 Measure times when the database is initialised after every module (part 3) 88
3.4 Measure times when the database is initialised after every module (part 4) 89
3.5 Measure times when the database is initialised just at the beginning (part 1) 90
3.6 Measure times when the database is initialised just at the beginning (part 2) 91
3.7 Measure times when the database is initialised just at the beginning (part 3) 92
3.8 Measure times when the database is initialised just at the beginning (part 4) 93

BIBLIOGRAPHY

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

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

[3] 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*, pages 27–38, 2003.

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

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

[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, page 129.

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

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

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

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

[12] MySQL homepage. http://www.mysql.com/.

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

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

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