Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Erlang kódok refaktorálása: Függvénybehelyettesítés

Bozó István

Programozó matematikus, nappali tagozat

Konzulens: Dr. Horváth Zoltán

Budapest, 2008

Supported by ELTE IKKK, Ericsson Hungary.

Eötvös Loránd University
Faculty of Informatics
Programming Languages and
Compilers Department

Erlang Refactoring: Inline Function

István Bozó

Supervisor: Dr. Zoltán Horváth

Budapest, 2008

CONTENTS

Introduction 5

1. The tool structure 6

2. User Manual 7
- 2.1 Installation guide for Windows 7
- 2.2 Installation guide for Linux 7
- 2.3 Minimum requirements 8
  - 2.3.1 Hardware 8
  - 2.3.2 Software 8
- 2.4 Using the tool 8
  - 2.4.1 In Windows 8
  - 2.4.2 In Linux 9
- 2.5 Inline Function 9
  - 2.5.1 Conditions of applicability 10
  - 2.5.2 Inline function refactoring 10
  - 2.5.3 Flowchart diagram of the refactoring step 11
- 2.6 Development interface 15

3. Development Manual 17
- 3.1 Module description - Functions 17
  - 3.1.1 Exported functions 17
  - 3.1.2 Local functions 18
- 3.2 Testing the function and the results 58
- 3.3 Inline function test cases 59
  - 3.3.1 Application in tuple 60
  - 3.3.2 Application in clause 61
  - 3.3.3 Application in list 62
  - 3.3.4 Application in match expression 63
  - 3.3.5 Application in case pattern 64

3.3.6 Application in begin-end block 65
3.3.7 Application in infix expression 66
3.3.8 Application in the argument list of an application . . . 67
3.3.9 Application with an application in the argument list . . 68
3.3.10 Application in list generator 69
3.3.11 Variable name collisions 70
3.3.12 Application from other module with exported function in the body 71
3.3.13 Application from other module with two clauses and with elemental in argument list 72
3.3.14 Application with two clauses and with guard expression 73

4. Summary 74
4.1 Summary 74
4.2 Related work 74
4.3 Developing the tool 74

List of Figures 75

Bibliography 76

INTRODUCTION

Refactoring [1] is a controlled process of improving the design of an existing code without changing the external behavior of the program. This controlled process guarantees the behavior preservation, thus the refactoring does not produce (or remove) any bugs in the code.

In every programming language is emergence of claim refactoring tools. Such tools grantee safe and fast restructuring and transforming of the code to improve the effectiveness, the reuse and the readability of our code. Most of such tools are concentrated on object-oriented programming languages, but the increasing use of functional programming languages in the industry forces to develop refactoring tools to the functional programming languages too.

Our project group have developed a prototype of the refactoring tool for Erlang which uses an SQL database to store the source code. The refactoring of the code is done via GNU Emacs text editor. After loading an Erlang source file in Emacs the Erlang menu item will appear in the menu bar. From this menu item we can reach for example the erlang compiler and also the Refactoring tool. There are several refactoring functions implemented and included in the refactoring tool. The inline function extends the scale of these refactoring functions.

This thesis is part of this bigger project. Includes the preconditions of applicability, the realization of the inline function refactoring step and the test results.

In the first chapter is the overview of the tool. The following two chapters are the User Manual and the Development Manual. The Development manual contains the documentation of the testing process.

1. THE REFACTORING TOOL STRUCTURE

The meaning of phrase ”refactoring” is program transformation or program restructuring that preserve the external meaning of the existing programs [1]. These transformations are intended to improve the quality of program code, to make it more readable, reusable etc.

Refactoring steps as rename variable, extract function, inline function etc. are used daily in the industry. Most of the refactoring tools are designed for object-oriented programming languages as C++, Java, C# etc., thus the refactoring word has appeared in the object-oriented methodology [12].
This tool is intended to bear a hand for Erlang programmers in refactoring.

Before the appearance of refactoring tools the refactoring steps have been applied manually by the programmer. Applying the refactoring manually is very unsafe in many aspects. It claims for heavy attention for every detail in the code. For example doing the inline function manually is hard because the programmer must pay attention on renaming the variables properly, qualify the applications in the substituted code part if the functions are from other module and so on.

Refactoring in functional languages is getting spread nowadays, there are some researches on this topic. The only full-featured refactoring tool in functional programming languages is HaRe [14, 13] for Haskell programs within the Emacs [3] and VIM [15] editors. For Clean is also available [4, 5] a prototype of the refactoring tool.

The goal of our project is to develop a refactoring tool for functional programming language Erlang, which comes through in the industry. The prototype of this tool is available now. The inline function is part of this. The prototype of the tool uses the Standard Erlang Parser, and in order to store and manipulate the syntax tree and the semantical information [11] the refactoring tool uses a relational database MySQL.

2. USER MANUAL

2.1 Installation guide for Windows

The installation for Windows is very simple.
Follow the steps below :

1. Download the RefactorErl 0.2 Installer package for Windows from `http://plc.inf.elte.hu/erlang/dl/`.
2. Start the `refactorerl-0.2-setup.exe file`.
   The installation will start and install the required programs, listed below, to the computer.
   - Erlang/OTP R11
   - GNU Emacs 21
   - MySQL 5.0
3. Download the `upgadeTool.zip` file from `http://people.inf.elte.hu/bozo_i/inline/`.
4. Unzip the file and start the `upgradeTool.exe`. This will upgrade the already installed tool and add the inline function to it.

2.2 Installation guide for Linux

Follow the steps below:

1. Install Erlang/OTP R11B (http://www.erlang.org/download.html)
2. Install GNU Emacs 21 (ftp://ftp.gnu.org/gnu/emacs/)
3. Install MySQL 5.0 (http://dev.mysql.com/downloads/mysql/5.0.html)

4. Download the RefactorErl 0.2 Source Package in ZIP format from the `http://plc.inf.elte.hu/erlang/dl/`.

5. Unzip it.

6. Download the `upgradeSource.zip` file from `http://people.inf.elte.hu/bozo_i/inline/`.

7. Unzip it.

8. Overwrite the file `refactor.el` in the `RefactorErl-0.2/` directory and the `d_client.erl, refactor.erl` files in the `RefactorErl-0.2/refactor/src/` directory with these files.

9. Add the `refac_inline_fun.erl` file to the `RefactorErl-0.2/refactor/src/` directory.

10. Follow the installation guide `INSTALL-UNIX.txt` in `RefactorErl-0.2/` directory.

2.3 Minimum requirements

2.3.1 Hardware

- Approximately 400 MB free disk space for the components and 250 MB or more for the database
- 256 MB memory

2.3.2 Software

- The above mentioned four components.
- Windows XP/2000/2003/NT or Linux operating system.

2.4 Using the tool

2.4.1 In Windows

1. Start the RefactorErl program.(This will start the GNU Emacs 21 program, start MySQL server and open the erlang node.)

2. Open an erlang file which we want to restructure.(After opening the file the `Erlang` menu item will appear in the menubar.)

3. At the very first start the database must be initialized. For initializing use shortcut `Ctrl-c Ctrl-e i` (from menu `Erlang` → `Refactor` → `Initialise database`).

4. Before doing any refactoring on the opened module we need to load this module to the database. Use shortcut `Ctrl-c Ctrl-e n` (from menu `Erlang` → `Refactor` → `Into_db and reload from there`).

5. If the implementation of the application is in other module, this module should be loaded in database too.

6. Now the tool is prepared for applying refactoring steps on the loaded modules.

2.4.2 In Linux

1. Run the GNU Emacs 21.

2. Follow steps from 2 to 6 described in section Using the tool in Windows.

2.5 Inline Function

The inline function refactoring step substitutes the selected application with the corresponding function body. The function may consist of one or more function clauses and may have guard expression(s), hence the inline function must be able to handle these cases. Applying the inline function is not only a text substituting procedure, it works from the syntax tree which is stored in the database. Through the substituting the inline function does the renaming of the variables, sets the binding of the variables, maintains the applications in the substituted part of code and adds functions to the import list if it is necessary, hence other refactoring steps can be done on the substituted code.

In Fig. 2.1 a simple example is shown for inline function. Inline is applied for application `temp/1` returning the double of the given number used in function `double/1`.

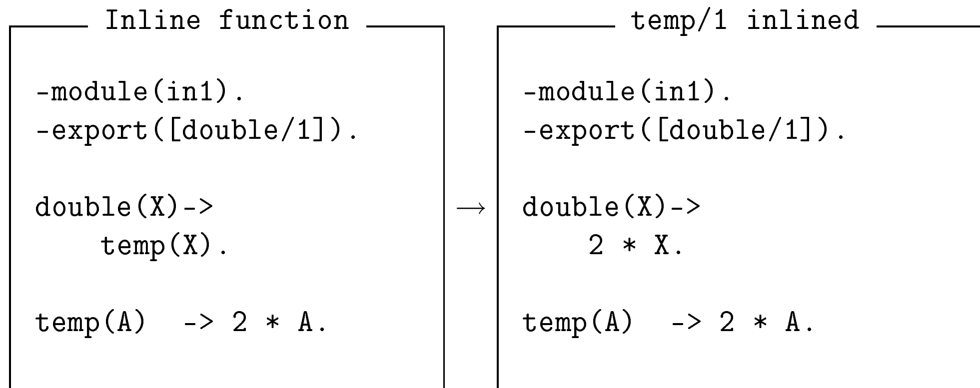


Fig. 2.1: Inline function `temp/1` in function `double/1`.

2.5.1 Conditions of applicability

1. If the implementation of the pointed application is in other module, this module should be loaded to the database too.
2. On the pointed position should be an application or a module qualifier of the application.
3. Applying the inline function can not evolve variable conflicts.
4. If the implementation module of the function differs from the module of the pointed application, it is not allowed to contain local functions (not exported functions) in the body of the function.
5. If the implementation module of the function differs from the module of the pointed application, it is not allowed to contain macro or record reference(s) in the body of the function.

2.5.2 Inline function refactoring

As most of the refactoring steps the inline function can also be divided to three parts:

1. Collecting the necessary data for the refactoring.
2. Checking the preconditions.
3. If the preconditions hold perform the refactoring.

Collecting information for refactoring

In this section the inline function collects the necessary information for refactoring: module identifier of the actual module, identifier of the selected application, the path to the pointed application, the parent node of the application, the identifier of the previous node in the syntax tree, module identifier of the function's implementation, the identifier(s) of the function clause(s). If the pointed object is not an application or the module of the function implementation is not loaded in the database, it throws terms.

Checking preconditions

Checks the applicability by using the previously collected information: checks for variable name collisions, if the functions implementation module differs from the actual module checks for macro, record expressions and local functions.

Perform refactoring

In this section the inline function replicates the function clause(s) and guard expression(s) if there is any, after the replication attaches it to the parent node of application and detaches the application node. If the function has more than one clause, or it has a guard expression, or has elemental types in the argument list a case expression will be created and attached to the parent node including the function clause(s) and guard expression(s), otherwise a begin-end block will be created and attached to the parent node including the replicated body of the function. After replicating and attaching the semantic information will be restored in the database.

Parameters

The parameters for the inline function are the followings : the path of the current file, the line and the column of the pointed application.

2.5.3 Flowchart diagram of the refactoring step

The following flowchart diagram (Figures 2.2 and 2.3) describes the structure of the inline function (refac_inline_fun) module.

The applied signs are:

- The begin and the end points are marked with green ovals.
- The error messages are marked with red ovals (the function ends with an error message).
- The decision points are marked with orange diamonds.
- The red arrow texts are the results of the decisions.
- The functions of the tool are marked with white boxes.
- The functions of the tool are marked with light blue boxes (these functions are described in more details).

Fig. 2.2: The structure of the refac_inline_fun module

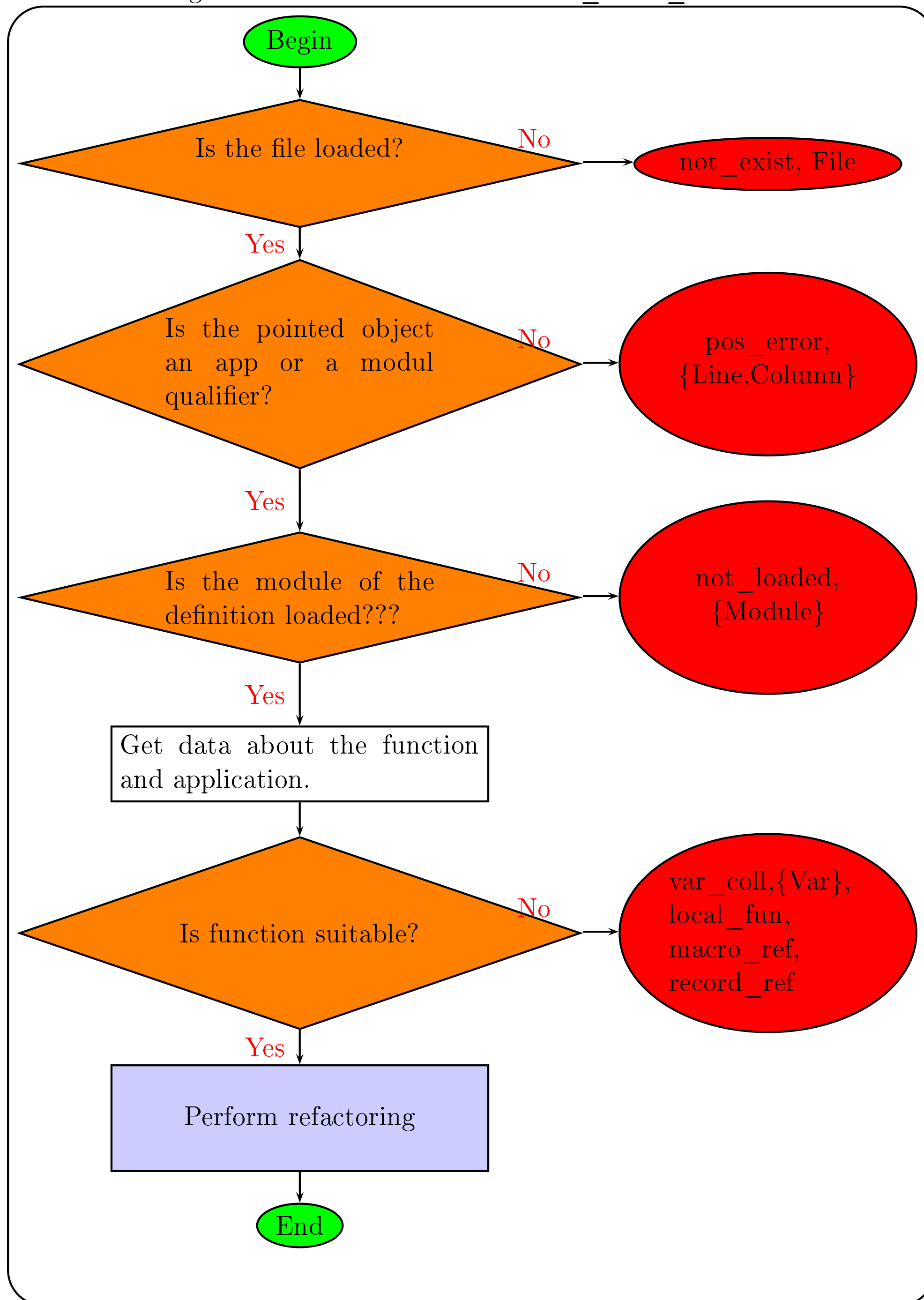
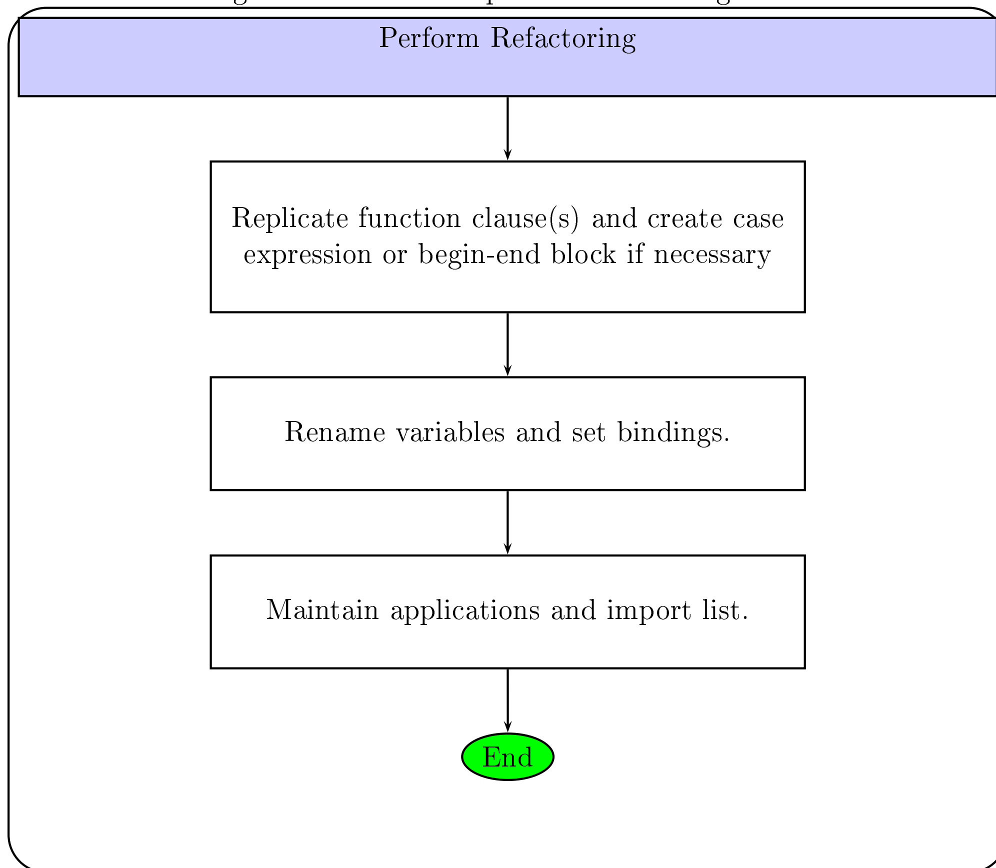


Fig. 2.3: Structure of perform refactoring function



2.6 Development interface

The function is integrated in Emacs. This choice is because most of the Erlang programmers use this text editor.

The use of the inline function refactoring step is recommended for users who want quickly and safely transform and restructure his Erlang code. The first step is to start the RefactorErl which starts the Emacs text editor, the MySQL server and the Erlang node.

Open an Erlang module which we want to reconstruct. After opening the module an Erlang menu item will appear in the menu bar. If the database is not initialized yet, than it should be initialized by shortcut `Ctrl-c Ctrl e i` (from menu `Erlang` → `Refactor` → `Initialize database`). This step is necessary only at the very first start after installing. Then we should load the module in the database by shortcut `Ctrl-c Ctrl-e n` (from menu `Erlang` → `Refactor` → `Into_db and reload from there`). If the implementation of the pointed application is in other module, this module should be loaded to the database too.
The next step is to point out an application for which we want to apply the refactoring step and apply the inline function refactoring by shortcut `Ctrl-c Ctrl-e l` (from menu `Erlang` → `Refactor` → `Inline function`).

Refactoring is for improving the design, the structure and the readability of the program without changing its external behavior. The manual refactoring is very hard even on small programs, because we need to be regard for every detail. There are some situation where we must deny the transformation.

In Fig. 2.4 the inline of the function `add/2` in `sum/2` function will be forbidden, because there is included `dummy/1` local function in the body of the `in2:sum/2` function. The inline will terminate with error message `local_function`.

If there will be a record or a macro reference included in the body of the `in2:sum/2` function in place of the local function, the inline would be forbidden too, with `macro_in_clause` or `record_in_clause` error message. If the `dummy/1` function would be exported in `in2` module and the result of the inline see on Fig. 2.5

```
Inline forbidden
 -module(in1).
 -export([sum/3]).
mul(X, Y, Z)->
     W = in2:sum(X,Y) * Z,
     W + W.

 -module(in2).
 -export([sum/2]).
 sum(A,B) -> A + dummy(B).
 dummy(X) -> X.
```

Fig. 2.4: Local function in the body of the `in2:sum/2`.

```
Inline done
-module(in1).
-export([sum/3]).
mul(X, Y, Z)->
    W =
       begin
         X + in2:dummy
       end * Z,
    W + W.

-module(in2).
-export([sum/2,dummy/1]).
sum(A,B) -> A + dummy(B).
dummy(X) -> X.
```

Fig. 2.5: Inline of the `in2:sum/2` in function `in1:mul/2`.

3. DEVELOPMENT MANUAL

3.1 Module description - Functions

The function substitutes the function call for the functions body, only if the preconditions are satisfied, or throws different atoms in case the preconditions do not hold.

3.1.1 Exported functions

inline_function/3
Performs the refactoring step. This function is responsible for collecting data about the function, check the preconditions, do the replication and maintain the variables. It throws different terms if there is an error or the preconditions do not hold. If the return value is atom ok the inline has done successfully.

- Parameter description
  1. File : The path of the source module.
  2. Line : The pointed line.
  3. Column : The pointed column.
- Implementation:
  - The first step is to determine the source module identifier in the database by invoking the `refac_common:get_module_id/1` function. If the module is not loaded in the database, it throws a tuple which consists of two elements. The first element is a `not_exist` atom and the second element is the path to the file.
  - The next step is to collect data about the pointed item by invoking `get_data_about_application/3` function. If the type of the pointed node is an application or a module qualifier it returns the identifier of the application, the path from the outermost scope

and the scope of visibility. If the type of the pointed item differs from the above mentioned it throws an error message in the tuple. The first element of the tuple is the `pos_error` atom and the second element is the pointed position in the file (line and column number).

- The next step is to find the clause(s) of the function and the module identifier of the implementation module by invoking the `get_data_about_function/3` function.
- Using the path, the next step is to determine the parent node of the function call in the syntax tree. After this it asks for the children nodes of this parent node. From the given list of identifiers and the function call identifier it can determine the previous node which is used to allocate the point where to attach the functions replicated body. This is determined by using the `parent_id_and_previous_node/3` function.
- The next step is to get the local, exported and imported functions. This is determined by invoking the `get_applications/2` function.
- The last step before performing the refactoring is to check the preconditions and the possible variable collisions. Checks the local functions, the macro and the record reference if the definition of the function is in other module. This check is done by invoking the `check_functions_suitability/6` function.
- The data for performing the refactoring is collected, so in the next step it will perform the replication of the function bodies and the guard expressions, maintain the variable names and bindings, maintain the replicated function calls and the import list. This is done by invoking the `perform_refactoring/10` function.

This function can be invoked from emacs by using the shortcut `Ctrl-c Ctrl-e l` or select from the Erlang menu.(Erlang → Refactoring → Inline function)

3.1.2 Local functions

application_maintenance/5

This function does the maintenance of the replicated applications in the

replicated function clauses. Adds module qualifiers if it is necessary and maintains the import list.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. Nodes : The list of identifiers.
  4. Exported : The list of exported functions which are included in the body of the function.
  5. Imported : The list of imported functions which are included in the body of the function.
- Implementation:
  - Gets every element from the subtrees given in the list and filters the application identifiers only.
  - If the current module and the identifier of the implementation module is the same it maintains only the `fun_cache` table in the database. Inserts a new row if it is necessary.
  - If the two module differs then selects the exported functions from the application list. Gets the name, arity and module identifier of the application from the identifier. Gets the module name from the given Exported list and qualifies the application by invoking the `qualify_app/3` function. Maintains the import list by adding only new elements.
  - Maintains the `fun_cache` table in the database by using the `maintain_fun_cache/3` function.
  - If everything has done it returns the ok atom.

  This function is used in `perform_refactoring/10`.

attach_nodes/6
Attaches the nodes to the syntaxtree. The way of attaching depends on the type of the parent node.

- Parameter description:

1. MId : The identifier of the module.
2. AppId : The identifier of the application.
3. Parent : The parent node of the application.
4. After : The identifier after which to insert.
5. Elements : The list of identifiers to be inserted.
6. WithBody : An atom to create a `begin-end` block or not. Possible values : `with_body` and `without_body` terms.

- Implementation:
  - Creates the `begin-end` block if it is required including the elements.
  - Then attaches the elements to the parent node after the identifier After given in the parameter list.
  - The following types are allowed as a parent node: CLAUSE, CASE_EXPR, MATCH_EXPR, GENERATOR, TUPLE, APPLICATION, LIST, INFIX_EXPR, BLOCK_EXPR. Otherwise it throws an exception as a tuple with elements :`bad_parent` and the identifier of the parent node.

This function is used in `replication_with_more_function_clauses/9`, `replication_without_guard_or_elemental/11`, `replication_with_guard_or_elemental/11`.

check_functions_suitability/7
Checks the functions suitability, checks the preconditions (possible variable collisions, local applications, record and macro references if the implementation module differs from the current module).

- Parameter description:

1. MId : The identifier of the application module.
2. MId2 : The identifier of the implementation module.
3. FunClauses : The list of function clause identifiers.
4. AppScope : The identifier of the scope.
5. AppId : The identifier of the selected application.

6. LocalApp : The identifiers of the local applications in the function body.
7. Path : The path to the application.

- Implementation:
  - If the module of the application differs from the definitions module and there are any local applications in the function body it throws an exception. The exception is a tuple with elements: `local_app` term and the local application identifier.
  - If the identifier of the current module differs from the identifier of the implementation module and there are macro or record references, it throws an exception. The exception is a tuple with elements: `macro_in_fun_clause` or `record_in_funclause` and the identifier of the node.
  - Checks for variable collisions. Gets the intersection of variables between the body of the function and the variables from the scope of the application. After extracts from this list the variables which collision can be resolved by renaming. If there are any left then throws an exception. The exception is a tuple with elements : `var_coll` atom and the variable names.

This function is used in `inline_function/3`.

check_macros_and_records/2
Checks for macros and records in the given list. If there are any record or macro references it throws an exception.

- Parameter description:
  1. MId : The identifier of the module.
  2. Ids : List of identifiers.
- Implementation:
  - Checks for macro or record references in the given list of identifiers and throws an exception at the first occurence.

This function is used in `check_functions_suitability/6`.

clauses_from_list/2
Selects the clause identifiers from the given list of identifiers.

- Parameter description:
  1. MId : The identifier of the module.
  2. List : List of identifiers.
- Implementation:
  - Filters the nodes from the given list with type CLAUSE.

  This function is used in `perform_refactoring/10`.

create_case_clauses/6
Replicates the function clauses for the case expression with the guards if there are any.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. AppScope : The scope of the application.
  4. AppArgIds : The argument identifiers of the application.
  5. FunClauses : The list of the function clause identifiers.
  6. Path_Clauses_reversed : The reversed list of clause identifiers in the path.
- Implementation:
  - For every clause identifier repeats the same cycle which consists from the following steps:
  - Replicates the guard expression if clause has a guard by using the `replicate_subtree_with_new_mid/4` function.
  - Replicates the nodes from the function clause by using `replication/4` function.
  - Replicates the clause arguments by using the `replicate_subtree_with_new_mid/4` function and creates a tuple from these elements.

  - Creates a clause included the previously replicated guard and nodes.
  - Creates variable pairs from the applications and function clause arguments by using `get_var_pairs_for_rename/4` function.
  - Renames the variables in the created clause by using the `rename_variables/3` function.
  - Sets the variable bindings only in the pattern by using `set_variable_binding/3` function.
  - Returns the list of clause identifiers.

This function is used in `replicate_with_more_function_clauses/9`.

create_import_attr/2
Creates an import attribute from the given list.

- Parameter description:
  1. MId : The identifier of the module.
  2. ImpElem : List of import elements ordered by the module name. Element type is Module::string(),Fun::string(),Arity::integer()
- Implementation:
  - Creates an arity qualifier for every element in the given list by using the name and arity given in the elements. From these element creates a new import attribute in the database.

This function is used in `import_list_maintenance/2`.

create_import_block/3
Processes the element from the given list with the given module name.

- Parameter description:
  1. Imported : List of tuples with elements : ModuleName::string(), FunctionName::string(), Arity::integer().
  2. ModName : The previously used module name.
  3. List : A temporary list, at the beginning an empty list.

- Implementation:
  - Sorts elements to the separate lists.
  - Elements from the same module are situated in the same list.
  - Returns a list.

  This function is used in `create_import_blocks/2`.

create_import_blocks/2
Splits the functions given in the parameter Imported. Creates groups by the name of the module.(The first element in the tuple.)

- Parameter description:
  1. Imported : List of tuples with elements : ModuleName::string(), FunctionName::string(), Arity::integer().
  2. List : A temporary list at the beginning an empty list.
- Implementation:
  - Selects elements to the separate lists. The separation is done by the function `create_import_block/3`.
  - Elements from the same module are situated in the same list.
  - Returns a list of list.

  This function is used in `import_list_maintenance/2`.

create_matches/5
Creates match expressions from the pairs which are created form the arguments of the function and arguments of the application.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. Pairs : List of attribute identifier pairs.
  4. FunClause : Function clause identifier.
  5. FunArgs : Function argument identifiers.

- Implementation:
  - Gets the unused variables from the body of the function by using the `get_unused_variables_from_fun_arg/3` function.
  - Filters the elements from the given list Pairs wherein the first element of the tuple is not an unused variable.
  - From the filtered list creates match expressions where the left side is the first element in the tuple and the right is the second element in the tuple.

  This function is used in `replication_without_guard_or_elemental-/11`.

delete_app_from_fun_cache/2
Deletes the given element from the `fun_cache` table.

- Parameter description:
  1. MId : The identifier of the module.
  2. AppId : The identifier of the application.
- Implementation:
  - Deletes the element in the `fun_cache` table with the given identifier and module.

  This function is used in `perform_refactoring/10`.

do_attribute_pairs/5
Creates and returns the pairs of attributes except the variable pairs. The attribute pairs are used to create match expressions.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. AppArgIds : Argument list of the application.
  4. FunArgIds : Argument list of the function clause.
  5. AttrPairs : List of pairs created from function and application argument list.

- Implementation:
  - Processes the arguments. Gets one from both lists in the given order. In case of both is type variable, drops these elements.
  - If at least one differs from type variable then it creates pairs, adds it to a list and proceeds with processing.
  - If the lists have been processed returns the created pairs.

  This function is used in `replication_with_one_function_clause/9`.

equable_names/2
Checks the duplicated variable names in the function argument list.

- Parameter description:
  1. MId2 : The identifier of the implementation module.
  2. FunArgs : Argument list of the function clause.
- Implementation:
  - Gets the variable identifiers from the given argument list.
  - Gets the names to the variable identifier names and usorts this list.
  - If the length of the variable identifiers list and the name list is not equal, it returns true, else it returns false value.

  This function is used in `replication_with_one_function_clause/9`.

find_binding/4
Finds the suitable binding for the variable.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : Identifier of a variable.
  3. Name : The name of the given variable.
  4. ClauseList : The first occurence would be in either of the clauses.
- Implementation:

- Gets the binding occurences from the given name and clause.
- If there is no candidate repeats the search in the outer clause.
- If it gets the binding occurences it selects the more suitable binding for the variable.
- If there is no suitable binding for the variable it throws an exception with elements : `binding_not_found` term, the module and the functions name.

This function is used in `set_variable_binding/3`.

find_the_functions_module_clause_id/2
Returns the identifier of the implementation module and the function clause identifiers from the application identifier. It throws an exception in a tuple if the functions module is not loaded. The first element of the exception is a `not_loaded` atom, the second element is the name of the module.

- Parameter description:
  1. MId : The identifier of the module.
  2. AppId : The identifier of the given application.
- Implementation:
  - Gets the identifier of the implementation module and the function identifier from the given module and application identifier.
  - If there is no such a module and function identifier it throws an exception `(not_loaded,{ModName})`.
  - In case it gets a module and a function identifier it returns a tuple with elements : the module identifier of the implementation module and a function identifier.

  This function is used in `get_data_about_function/2`.

get_app_argument_ids_from_appid/2
Returns the argument identifiers of the application.

- Parameter description:
  1. MId : The indentifier of the module.

2. AppId : Identifier of the application.

- Implementation:
  - Does a query in the database and returns the arguments of the application.

  This function is used in `perform_refactoring/10`,`check_functions_suitability/6`.

get_app_posnull_arg/2
Returns the argument of the application at position zero from the given module and with given identifier.

- Parameter description:
  1. MId : The identifier of the module.
  2. AppId : The identifier of the application.
- Implementation:
  - Gets the application arguments with the given identifier and module from the `application` table where position is 0.

  This function is used in `maintain_fun_cache/2, qualify_app/3`.

get_application_id_from_argument/2
This function returns the identifier of the application from the argument. This function is used for determining the identifier of the application in case of the selected item is a module qualifier.

- Parameter description:
  1. MId : The identifier of the current module.
  2. Argument : The identifier of the application argument.
- Implementation:
  - Does a query in the database in the `application` table with the given module identifier and argument identifier where the position is 0.

  - Returns a list of identifiers in tuple. There should be only one element in this list, because the module identifier and the argument identifier determine the function call unambiguously.

This function is used in `get_data_about_application/3`.

get_application_ids_from_list/2
Returns the filtered list of application identifiers. It is used to check the suitability of the function body for inline.

- Parameter description:
  1. MId : The identifier of the module.
  2. List : List of identifiers.
- Implementation:
  - Filters the identifiers from the given list with type application.

This function is used in `application_maintenance/5`, `get_applications/2`

get_applications/2
Returns the local, exported and imported applications from the given clause.

- Parameter description:
  1. MId : The identifier of the module.
  2. FunClauses : The list of function clause identifiers.
- Implementation:
  - Gets the export list from the module.
  - Gets the import list from the module.
  - For every function clause in the given list repeates the same step.
  - Gets the identifiers from the function body and filters from these identifiers the identifiers with type application.
  - Substracts the applications with module qualifier and the BIFs.
  - Gets the name and arity for the remaining list of applications.

- Usorts this list containing the tuples of name and arity. Substracts from this list the exported and imported functions and so leaves only the local functions.
- Summarizes the local, exported and imported functions from every function clause.
- Returns a tuple with usorted lists: list of local, exported and imported functions.

This function is used in `inline_function/3`.

get_applications_arg_type/2
Returns the argument type of the application on position 0.

- Parameter description:
  1. MId : The identifier of the module.
  2. AppId : Identifier of an application.
- Implementation:
  - Does a query in the database in the `application` table and returns the argument type at the position 0 whit the given module identifier and application identifier.

This function is used in `is_app_qualified/2, get_applications/2`.

get_applications_name_and_arity/2
This function returns the name and arity of the given application. Important: it works only for a simple applications, which is not qualified with a module qualifier.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the application.
- Implementation:
  - Gets the name from the application identifier by invoking the `get_name_to_simple_app/2` function.

  - Gets the arity from the application identifier.
  - Returns a tuple of the name and the arity.

This function is used in `application_maintenance/5`.

get_bifs_from_appid_list/2
Filters the builtin functions from the list of identifiers.

- Parameter description:
  1. MId : The identifier of the module.
  2. List : List of identifiers.
- Implementation:
  - Gets the name from the identifier.
  - Checks for the forbidden names. If the name of the application is not a forbidden name, the identifier will be dropped from the list.

This function is used in `get_applications/2`.

get_data_about_application/3
Returns the tuple of data about the application: the identifier of the selected application, the path from the outermost scope and the identifier of the scope. In case the selected item is not an application or a module qualifier it throws an exception in a tuple. The first element of the exception is the pos_error atom and the second is the position of the selected item (line and column).

- Parameter description
  1. MId : The current module identifier.
  2. Line : The pointed line.
  3. Col : The pointed column.
- Implementation:
  - Gets an identifier from the module identifier and the pointed positions.
  - From this identifier it determines the type of the pointed item.

  - If this node is a module qualifier or a function call it returns the application identifier. In case of the type differs from the above mentioned, it throws an exception (`{pos_error, {Line,Col}}`).
  - If determining the application identifier was successful it gets the path and the scope identifier to this function call from the root clause.
  - Returns the collected data in a tuple: function call identifier, path to the function call, scope identifier

The function is used in `inline_function/3`.

get_data_about_function/2
Returns the data about the function: the module identifier, the function identifier and the identifiers of the function clauses.

- Parameter description:
  1. MId : The identifier of the module.
  2. AppId : The identifier of a given application.
- Implementation:
  - Gets the definition module identifier and the function identifier.
  - In the next step gets the function clause identifiers.
  - Because it is a list of tupled identifiers, untuples these elements.
  - Returns a tuple with elements: module identifier of the implementation module, the list of function clause identifiers.

This function is used in `inline_function/3`.

get_following_id_in_list/2
Returns the identifier which follows the given identifier in the given list. Returns atom `first` if the given identifier is the last in the given list.

- Parameter description:
  1. Id : The identifier which is included in the list.
  2. List : The list of identifiers.

- Implementation:
  - It is a recursive function with two function clauses. Searches the given identifier in the list and returns the next identifier in a list. If the searched element is the last element in this list, then it returns atom `first`.

  This function is used in `parent_id_and_previous_node/3`.

get_fun_argument_ids_from_scope_id/2
Returns the argument identifiers of the function clause.

- Parameter description:
  1. MId2 : The identifier of the implementation module.
  2. FunClauseId : The clause identifier of the function.
- Implementation:
  - Does a query in the database and returns the function argument identifiers.

  This function is used in `replication_with_one_function_clause/9`, `check_functions_suitability/6`.

get_id_and_type_from_position/3
Returns an identifier and the type of this node from the given position.

- Parameter description:
  1. MId : The identifier of the current module.
  2. Line : The line of the selected item.
  3. Col : The column of the selected item.
- Implementation:
  - As a first step it gets the true positions which are stored in the database.
  - Determines the identifier of the selected item from the true position.

  - If there is no item at this position an exception in a tuple will be thrown. The first element of the tuple is the `pos_error` atom, the second is the selected position (line and column number).
  - In case there is an item at the selected position, the type of this item will be determined.
  - A tuple will be returned: the identifier of the selected item and the type of the item.

This function is used in `get_data_about_application/3`.

get_ids_from_scope/2
Returns identifiers from the given scope in the given module.

- Parameter description:
  1. MId2 : The identifier of the module.
  2. ScopeId : The identifier of the scope.
- Implementation:
  - Does a query in the database and returns a list of identifiers.

This function is used in `get_applications/2`.

get_module_and_body_from_module_qualifier/2
Returns a tuple of module and body identifier from the `module_qualifier` table.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the module qualifier.
- Implementation:
  - Does a query in the database.
  - Gets the module and body elements from the `module_qualifier` table with the given identifier in the given module.
  - Returns a tuple of these two elements.

This function is used in `maintain_fun_cache/3`.

get_module_name_from_app_id/2
Returns the module name from the function call identifier.

- Parameter description:
  1. MId : The identifier of the module.
  2. AppId : The identifier of the application.
- Implementation:
  - Gets and returns the module name from the database.

This function is used in `find_the_functions_module_clause_id/2`.

get_module_qualifier/2
Returns the module qualifier identifier from module or body identifier.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the module or body.
- Implementation:
  - Does a query in the `module_qualifier` table in the given module.
  - It searches for the identifier of the module qualifier where the module or the body matches with the given identifier Id.
  - Returns the identifier of the module qualifier.

This function is used in `get_data_about_application/2`.

get_name_to_simple_app/2
Returns the name of the simple application. Application is simple if it is not qualified. It can be used only if the application is not qualified.

- Parameter description:
  1. MId : The identifier of the module.
  2. AppId : The identifier of the application.

- Implementation:
  - Does a query in the database. Connects the `application` and `name` tables and returns the name of the application.

  This function is used in `maintain_fun_cache/3`, `get_application-_name_and_arity/2`, `get_applications/2`, `get_bifs_from_appid-_list/2`.

get_outermost_scope_id/2
Returns the outermost scope identifier.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of a node in the given module.
- Implementation:
  - Recursively looks for the outermost scope until it finds a clause node.
  - Returns the root scope identifier.

  This function is used in `get_path_to_app/2`.

get_path_to_app/2
Returns the path from the root clause and the scope of the selected function call.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of a node in the module.
- Implementation:
  - Gets the scope of the given node from the module identifier and the node identifier.
  - Gets the outermost scope from the module identifier and the determined scope identifier.

  - Gets the path from the outermost scope to the given node.
  - Returns a tuple of the path and the scope of the application.

This function is used in `get_data_about_application/3`.

get_unbound_variables/1
Returns variable identifiers where the target is 0 in the `var_visib` table in the database. These are the replicated variables, which were mounded in the parameter list.

- Parameter description:
  1. MId : The identifier of the module.
- Implementation:
  - Does a query in the `var_visib` table with the given module where the target is 0.
  - Returns a list of variable identifiers.

This function is used in `perform_refactoring/10`.

get_unused_variables_from_fun_arg/3
Returns the unused variables which are included in argument list, but are not used int the function's body.

- Parameter description:
  1. MId2 : The identifier of the implementation module.
  2. FunClause : The clause identifier of the function.
  3. FunArgs : Function argument list of the clause.
- Implementation:
  - Gets the subtrees with root from the argument identifiers.
  - Filters the variable names from the list by using the `get_var_names_from_list/2` function.
  - Gets variables from the function clause by using the `refac_common:get_variables/3` function and the names to the variables by using `get_var_names_from_list/2` function. This contains the variable names from the body and from the arguments too.

  - Substracts these elements from the doubled names of argument.
  - Thus the balance gives the unused variables.

This function is used in `create_matches/6`.

get_var_names_from_list/2
Returns the list of the variable names from the given list.

- Parameter description:
  1. MId : The identifier of the module.
  2. List : A list of identifiers.
- Implementation:
  - Filters and returns the variable names from the given list.
  - If the identifier is not a variable the name is an empty string.

This function is used in `get_unused_variables_from_fun_arg/3`.

get_var_pairs_for_rename/4
Returns the variable name pairs from the application arguments and from the function arguments. It is used for checking name collisions and for renaming.

- Parameter description:
  1. MId : The module identifier.
  2. Mid2 : The definition's module identifier.
  3. FunArgs : The function's argument list.
  4. AppArgs : The application's argument list.
- Implementation:
  - Zippes the two argument list.
  - If at the same position is a variable in both lists adds the tuple of these elements to the list.
  - If at the same position are nodes with the same type (except variables) it calls itself recursively on the subnodes of these elements.

  - If the zipped list has been processed, it returns the list of created tuples.

This function is used in `create_case_clauses/7`, `replication_without_guard_or_elemental/11`, `get_variable_collisions/6`,`replication_with_guard_or_elemental/11`.

get_variable_collisions/7
Returns the clashing variable names in a list.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the definition's module.
  3. AppScope : The scope identifier of the application.
  4. AppArgs : The application's argument identifiers.
  5. FunClause : The function's clause identifier.
  6. FunArgs : The function's argument identifiers.
  7. Path : The path to the application.
- Implementation:
  - Gets the variable names uniquely from the clause of the function.
  - Gets the variable names uniquely from the scope of the application where the scope of the variable binding is not in the list Path.
  - Gets the variable name pairs by using `get_var_pairs_for_rename/4`, which creates the pairs of names to determine which variable name is necessary to rename for what name, to resolve the name collisions.
  - Does the intersection of the variables from the scope and from the clause. Extracts from the intersection the names which will be renamed.
  - Returns the remaining variable names. The remaining elements are the collisioning variable names. If there is no name collision the return value is an empty list.

This function is used in `check_functions_suitability/6`.

import_list_maintenance/2
Does the import list maintenance after the inline if it is necessary. Inserts only the necessary imports.

- Parameter description:
  1. MId : The identifier of the module.
  2. Imported : List of imported functions which are included in the functions body.
- Implementation:
  - Gets the list of imported functions from the MId module.
  - Extracts from the given list Imported the import functions from the current module.
  - Creates blocks from the extracted list by invoking the `create_import_blocks/2` function.
  - Creates import attributes from the previously created blocks only if the name of the module differs from the current module name.
  - Attaches newly created import attributes to the current module.

  This function is used in `application_maintenance/5`.

insert_fun_call/4
Inserts a function call to the `fun_call` table.

- Parameter description:
  1. MId : The identifier of the module.
  2. CallId : The identifier of the application.
  3. TMId : The identifier of the target module.
  4. FunId : The identifier of the target function.
- Implementation:
  - Inserts to the `fun_call` table a new row with the given data.

  This function is used in `set_fun_call/4`.

is_app_qualified/2
Returns true or false value depending on the function is qualified with a module qualifier or not.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the application.
- Implementation:
  - Gets the applications argument type by using function `get_applications_arg_type/2`.
  - If the type is `MODULE_QUALIFIER` it returns `true` else returns `false` value.

  This function is used in `application_maintenance/5`.

is_case_expr_necessary/2
Checks for the elemental types in the given list and in the subtrees of the given nodes.

- Parameter description:
  1. MId2 : Identifier of the module.
  2. Ids : A list of identifiers.
- Implementation:
  - Gets the subtrees of the given nodes with root.
  - Checks for an elemental type in the subtrees and returns a Boolean value true or false.
  - Returns at the first occurence of the elemental type.

  This function is used in `replication_in_one_function_clause/9`.

is_case_necessary/2
Returns true or false atoms depending on the list contains an elemental type or does not. Elemental types are : `atom, char, float, integer, string`.

- Parameter description:

1. MId2 : The identifier of the module.
2. Nodes : List of identifiers.

- Implementation:
  - Searches for the elemental type in the given list.
  - Returns `true` value at the first occurence of the elemental type.
  - If the list does not contain elemental type it returns `false` value.

This function is used in `is_case_expr_necessary/2`.

maintain_fun_cache/3
Adds new elements to the `fun_cache` table if it is necessary.

- Parameter description:
  1. MId : The identifier of the module.
  2. AppId : The identifier of the application.
  3. Imported : List of imported functions which are included in functions body.
- Implementation:
  - Gets the applications argument from the `application` table from the corresponding row.
  - Gets the arity to this function.
  - If the argument is a `MODULE_QUALIFIER` then gets the module name identifier and the body identifier from the `module_qualifier` table.
  - From name identifiers gets the module name and the application name.
  - By using the `refactor:put_into_dbase_fun_cache/2` function inserts a new element to the `fun_cache` table.
  - If the argument is other type, gets the name of the application by using `get_name_to_simple_app/2` function and gets the module name. Searches for the module name in the Import list, if there is no such a function with previously determined arity, then gets the current module name.

  - By using the `refactor:put_into_dbase_fun_cache/2` function inserts a new element to the `fun_cache` table.

This function is used in `application_maintenance/5`.

parent_id_and_previous_node/3
Returns the parent identifier of the given node and the identifier of the left neighbor node in the syntax tree. This is necessary information where to attach the clause of the function.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the application.
  3. Path : The list of identifiers (application identifier is included), the path from the root clause to the function call.
- Implementation:
  - Reverses the path.
  - Determines the parent node from the reverse list. It is the second element in the reversed list.
  - Gets the children nodes of the previously determined parent.
  - Reverses the children list and returns the next element to the given application identifier.
  - Returns a tuple with elements : the identifier of the parent node and the identifier of the previous node. If the application does not have a left neighbor node it will return the parent identifier of the parent node and the atom `first`.

This function is `used in inline_function/3`.

perform_refactoring/10
Performs the refactoring from the previously gathered information. Performs the replication and the compulsory settings. Attaches the replicated nodes to the corresponding place in the syntaxtree. Maintains the import list and qualifies the applications if it is necessary.

- Parameter description:

1. MId : The identifier of the module.
2. MId2 : The identifier of the implementation module.
3. Path : The path to the application.
4. AppId : The application identifier.
5. AppScope : The scope identifier of the application.
6. Parent : The parent node identifier of the application.
7. AfterNode : The identifier of the AfterNode in the syntax tree.
8. FunClauses : The list of function clause identifiers.
9. Exported : The list of exported functions which are included in the function's body. List of tuples with elements: function name, function arity.
10. Imported : The list of imported functions which are included in the function body. List of tuples with elements: module name, function name, function arity.

- Implementation:
  - Does the replication. Determines how to attach nodes to the parent node.
  - If the function has one clause then it creates only a begin-end block with the replicated nodes. If the function has a guard expression or has multiple clauses or both it creates a case expression included the clause(s) of the function.
  - Sets the variable bindings.
  - Qualifies the applications if it is necessary.
  - Deletes the application node from the database.

This function is used in `inline_function/3`.

qualify_app/3
Modifies the application, adds a module qualifier to the application. It works only for the simple application without module qualifier.

- Parameter description:

1. MId : The identifier of the module.

2. MId2 : The identifier of the definition module.
3. Id : The identifier of the application.

- Implementation:
  - Gets the argument of the application at position zero by using `get_app_posnull_arg/2` function. It should be a name identifier of the application.
  - Gets the name of the definition module and creates a new atom from this name.
  - Creates a new module qualifier from the previously created atom and from the argument of the application.
  - Updates the application argument at the position zero with the newly created module qualifier by using `refactor:update_application_argument/4` function.

This function is used in `application_maintenance/5`.

rename_variables/3
Renames variables given in the list by using the given pairs.

- Parameter description:
  1. Mid : The identifier of the module.
  2. Variables : List of variables.
  3. Pairs : List of variable pairs.
- Implementation:
  - For each variable in a given list gets its name.
  - Gets the pair of this name from the pair list.
  - If the pair is the same name or the pair is atom `none` it does nothing. Else updates the name of the variable in the database.

This function is used in `replication_with_guard_or_elemental/11`, `create_case_clauses/6`, `replication_without_guard_or_elemental/11`.

replicate_attr_pairs/4
Replicates given attribute pairs.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. AppScope : The scope identifier of the application.
  4. AttrPairs : The tuple of attribute pairs.
- Implementation:
  - For every element of the given list does the same step.
  - Replicates the first element of the current tuple by using the `replicate_subtree_with_new_mid/4` function and for every variable in the subtree sets the binding. The binding for the variable will be itself.
  - Replicates the second element of the current tuple by using the `replicate_subtree_with_new_mid/4` function.
  - Returns a list of tuples with the replicated arguments.

  This function is used in `replication_without_guard_or_elemental-/11`.

replication/4
Replicates the subtree with the new module identifier and with the given scope identifier. It is used to replicate the body of the function.

- Parameter description:
  1. MId2 : The identifier of the implementation module.
  2. FunClause : The function identifier.
  3. MId : The identifier of the module.
  4. AppScope : The scope identifier of the application.
- Implementation:
  - Gets the body of the functions clause.

- Replicates the body of the function by using the `replicate_subtree_with_new_mid/4` function.
- Returns a list of the replicated identifiers.

This function is used in `replication_with_one_function_clause/9`, `create_case_clauses/4`.

replication_with_guard_or_elemental/11
Replicates the nodes from the functions clause. This function is used when the function clause has a guard expression or an elemental type is included in the argument list.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. AppId : The identifier of the application.
  4. AppScope : The scope of the application.
  5. AppArgIds : List of identifiers of the application arguments.
  6. FunClauses : List of function clauses.
  7. FunArgIds : List of identifiers of the function arguments.
  8. RepNodes : List of replicated nodes.
  9. Parent : The identifier of the parent node.
  10. After : The afters identifier.
  11. Path_Clauses_reverse : Reversed list of clauses from the path.
- Implementation:
  - Replicates the guard expression if the clause has a guard.
  - Replicates the arguments of the application arguments and does a tuple from these elements.
  - Replicates the argument of the function clause and does a tuple from these elements.
  - Creates a clause from the replicates arguments of the function and from the replicated body of the function.

  - Creates variable pairs from the argument lists by invoking the `get_var_pairs_for_rename/4` function and renames variables.
  - Creates a case expression from the previously created clause, tuple of replicated application arguments and from the replicated guard expression.
  - Sets the variable bindings in the newly created case expression.
  - Attaches the created node to the parent node after the determined node.

This function is used in `replication_with_one_function_clause/9`.

replication_with_more_function_clauses/9
Does the replication with more function clauses and attaches it to the syntaxtree.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. AppId : The identifier of the application.
  4. AppScope : The scope identifier of the application.
  5. AppArgIds : The argument identifiers of the application.
  6. FunClauses : The clause identifiers of the function.
  7. Parent : The identifier of the parent node.
  8. After : The identifier of the previous node.
  9. Path_Clauses_reversed : Reversed list of clause identifiers from the path.
- Implementation:
  - Replicates the arguments of the application invoking the `replicate_subtree_with_new_mid/4` and creates a tuple from these elements.
  - Replicates the clauses of the function using the `create_case_clauses/6`, which returns the list of clause identifiers.

– Creates a case expression from the previously created tuple of arguments and clauses.
– Attaches the created case expression to the syntaxtree below the parent node and after the After node using the `attach_nodes/6` function.
– Returns the identifier if the created case expression in a list.

This function is used in `perform_refactoring/10`.

replication_with_one_function_clause/9
Replicates the function clause, it is used for replicating function with one clause. Attaches nodes to the syntax tree.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. AppId : The identifier of the application.
  4. AppScope : The scope identifier of the application.
  5. AppArgIds : The argument identifiers of the application.
  6. FunClauses : The function clause identifier.
  7. Parent : The identifier of the parent node.
  8. After : Identifier of the left neighboring node.
  9. Path_Clauses_reversed : The reversed list of clauses from the path.
- Implementation:
  – Replicates the nodes from the body of the function.
  – Gets the argument identifiers of the function clause and checks for the elemental type. Elemental types are `ATOM, CHAR, FLOAT, INTEGER, STRING`.
  – Creates argument pairs from the application and function argument identifiers.

  - In case of there is identifier of an elemental type in the argument list or the clause has a guard expression or there are name duplicates in the function argument list a case expression is necessary, else a block expression will be created included the replicated nodes.

This function is used in `perform_refactoring/10`.

replication_without_guard_or_elemental/11
Replicates and attaches the nodes from the body of the function and adds matches used the arguments of the application and function if it is necessary.

- Parameter description:
  1. MId : The identifier of the module.
  2. MId2 : The identifier of the implementation module.
  3. AppId : The identifier of the application.
  4. AppScope : The identifier of the application scope.
  5. AppArgIds : Argument identifiers of the application.
  6. FunClauses : Function clause identifiers.
  7. RepNodes : Replicated nodes.
  8. FunArgIds : Argument identifiers of the function clause.
  9. AttrPairs : Created attribute pairs from the argument lists.
  10. Parent : The identifier of the parent node.
  11. After : Identifier of the previous node.
- Implementation:
  - Creates variable pairs from the application and function arguments for renaming.
  - Renames variables for the convenient name.
  - Creates match expressions from the attribute pairs.
  - Attaches the replicated nodes and match expressions to the parent node after the After node.
  - Returns the list of attached nodes.

This function is used in `replication_with_one_function_clause/9`.

return_pair/2
Returns the pair of the given string from the given list or if there is no pair for the given element it returns the atom `none`.

- Parameter description:
  1. Name : The name of the variable we are looking for the pair in the given list.
  2. List : List of tuples. Description of the elements : VariableName::string(), PairName::string().
- Implementation:
  - Recursively calls itself and looks for the tuple where the first element is equal with the given name.
  - In case it found the corresponding tuple in the given list it returns the second element of the tuple. Else it returns the `none` atom.

This function is used in `rename_variables/3`.

set_fun_call/4
Inserts a new row into `fun_call` table if it is necessary.

- Parameter description:
  1. MId : The identifier of the module.
  2. NewMId : The identifier of the new module.
  3. Id : The identifier of the application.
  4. NewId : The identifier of the replicated application.
- Implementation:
  - Gets the `tmid, target` values from the `fun_call` table where the module identifier is MId and identifier is Id.
  - If the return value of the previous query is not empty inserts a new element to the `fun_call` table by using `insert_fun_call/4` function with values NewMId, NewId and with the result of the previous query. Else does nothing.

This function is used in `replicator_with_new_mid/5`.

set_scope/3
Sets the scope of the nodes to the given scope identifier. If the node is a list comprehension or a function expression, then creates the a new scope identifier with the identifier and sets the variables in the subtree of this node for the new scope identifier.

- Parameter description:
  1. MId : The identifier of the module.
  2. List : List of identifiers.
  3. ScopeId : The scope identifier to be set.
- Implementation:
  - For every element given in the list calls itself recursively.
  - Gets the type of the current node if the type is list comprehension or a function call, creates the new scope in the database with the current identifier and calls itself recursively for the subtree with the new scope identifier.
  - Otherwise sets the scope of the element by using the `update_scope/3` function and repeats the recursion until the given list in the parameter is an empty list.

This function is used in `perform_refactoring/10`.

set_temporary_binding/5
Selects the clause identifiers from the given list of identifiers.

- Parameter description:
  1. NewMId : Id of the new module.
  2. MId : The current module.
  3. Id : Id of the variable.
  4. NewId : The new variable id.
- Implementation:

  - Gets the binding of the variable with identifier Id from the database.
  - If the binding identifier is equal with the id of the variable, then sets the binding of the new variable to itself.
  - Else the binding will be 0 which requires maintenance after the replicating.

  This function is used in `replicator_with_new_mid/5`.

set_variable_binding/3
Updates the variables bindings for variables given in the list.

- Parameter description:
  1. MId : The identifier of the module.
  2. VariableList : List of variables.
  3. ClauseList : List of clauses from the path.
- Implementation:
  - Gets the suitable binding for the variables and if it is unambiguous sets the new binding.

  This function is used in `perform_refactoring/10, create_case_clauses/7, replication_with_guard_or_elemental/11`.

suitable_binding/2
Selects the most suitable binding candidate from the list of candidates.

- Parameter description:
  1. MId : The identifier of the module.
  2. Candidates : List of candidates.
- Implementation:
  - For every candidate in the given list gets the binding and selects the most suitable one.
  - It calls itself recursively until the suitable binding is not found.

   - Returns the suitable binding identifier from the given list. If something is wrong it returns the `not_found` atom.

  This function is used in `find_binding/4`.

summarize/4
Summarizes the elements of the given list into one tuple.

- Parameter description:
  1. List : The given list of tuples.
  2. Local : Already processed local function list.
  3. Exported : Already processed exported function list.
  4. Imported : Already processed imported function list.
- Implementation:
  - Summarizes the elements of the given list into one tuple.
  - After the summarizing usorts the elements and returns the usorted tuple.

  This function is used in `get_applications/2`.

update_application_argument/4
Changes the old value of the application argument for the new one.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the application.
  3. SourceId : The identifier of the argument which to change.
  4. Arg : The identifier of the new argument.
- Implementation:
  - Updates the argument of the application for the new value Arg where the identifier of the application is Id and the identifier of the old argument is SourceId.

  This function is used in `attach_nodes/6`.

update_binding/3
Updates the binding of the variable in the database.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the variable.
  3. BindId : The binding target to be set.
- Implementation:
  - Updates in the `var_visib` table the target of the variable for a given target identifier in the given module.

  This function is used in `set_variable_binding/3`.

update_block_expr/4
Updates the body of the selected block expressions for the new body identifier.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the block expression.
  3. SourceId : The identifier of the body which we want to update.
  4. Body : The identifier of the new body.
- Implementation:
  - Updates the body of the block expression for the new value Body where the identifier of the block expression is Id and the identifier of the old argument is SourceId.

  This function is used in `attach_nodes/6`.

update_case_argument/4
Updates the argument of the case expression on the given position for the new argument identifier.

- Parameter description:

1. MId : The identifier of the module.
2. CaseId : The identifier of the case expression.
3. Pos : Which position to update.
4. Arg : The identifier of the new argument.

- Implementation:
  - Updates the argument of the case expression for the new value Arg where the identifier of the block expression is CaseId and the position is Pos.

  This function is used in `attach_nodes/6`.

update_generator_body/3
Updates the body of the generator. Changes the body for the new value.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the generator.
  3. Body : The identifier of the new body.
- Implementation:
  - Updates the body of the generator for the new value Body where the identifier of the generator is Id.

  This function is used in `attach_nodes/6`.

update_infix_expr/4
Updates the left or right side of the infix expression for the new value.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the infix expression.
  3. SourceId : The identifier of node which to change.
  4. Element : The identifier of the new element.
- Implementation:

  - Updates the left or right element of the infix expression for a new element where the left or right side is the SourceId for element Element.

This function is used in `attach_nodes/6`.

update_list_element/4
Updates the element identifier of the list for the new identifier.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the list.
  3. SourceId : The identifier of the old element.
  4. Element : The identifier of the new element.
- Implementation:
  - Updates the element identifier of the list for a new value Element where the identifier of the list is Id and the identifier of the old element is SourceId.

This function is used in `attach_nodes/6`.

update_scope/3
Updates the scope identifier of the given identifier in the database.

- Parameter description:
  1. MId : The identifier of the module.
  2. ScopeId : The identifier of the scope.
  3. Id : The identifier of the node.
- Implementation:
  - Updates the scope of the identifier Id for a new value ScopeId.

This function is used in `set_scope/3`.

update_tuple_element/4
Updates the element of the given tuple with the given source identifier for the given new element.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the tuple.
  3. SourceId : Identifier of the element to be changed.
  4. Element : The identifier of the new element.
- Implementation:
  - Updates the element identifier of the tuple for a new value Element where the identifier of the tuple is Id and the identifier of the old element is SourceId.

  This function is used in `attach_nodes/6`.

update_variable_name/3
Updates the variable name in the database for a new name.

- Parameter description:
  1. MId : The identifier of the module.
  2. Id : The identifier of the variable.
  3. NewName : The string of the new name.
- Implementation:
  - Updates the name of the variable with identifier Id for a new name NewName.

  This function is used in `rename_variables/3`.

3.2 Testing the function and the results

Testing the inline function via Emacs is fast and simple, because the tool is built in the Emacs text editor.

I have produced the test cases for testing the inline function. The testing

have been done on different test cases which test the preconditions and applicabilities of the inline function. The test cases, the results and the discussion are described in the Inline function test cases section.
After applying the inline function the other refactoring steps can be applied on the code, because the function restores the sintactical and semantical informations.

The inline function has been tested in cases where the application is/has

- an element of the tuple.
- in a case clause.
- member of a list.
- in a match expression.
- part of a case expression argument.
- in a begin-end block.
- on the left or right side of an infix expression.
- in the argument list of an other application.
- an application in its argument list.
- in a list generator.
- in the clause of a function and the inline of the application would produce variable name conflicts.
- defined in other module and there are exported functions in its body.
- defined in other module and it has multiple clauses.
- multiple clauses and guard expressions.

The inline function has been applied on several test cases listed above. The function works properly and returns the expected result.

3.3 Inline function test cases

3.3.1 Application in tuple

The name of the application which the inline will be applied to is `dummy/1`. The refactoring is acceptable, the result is shown below.

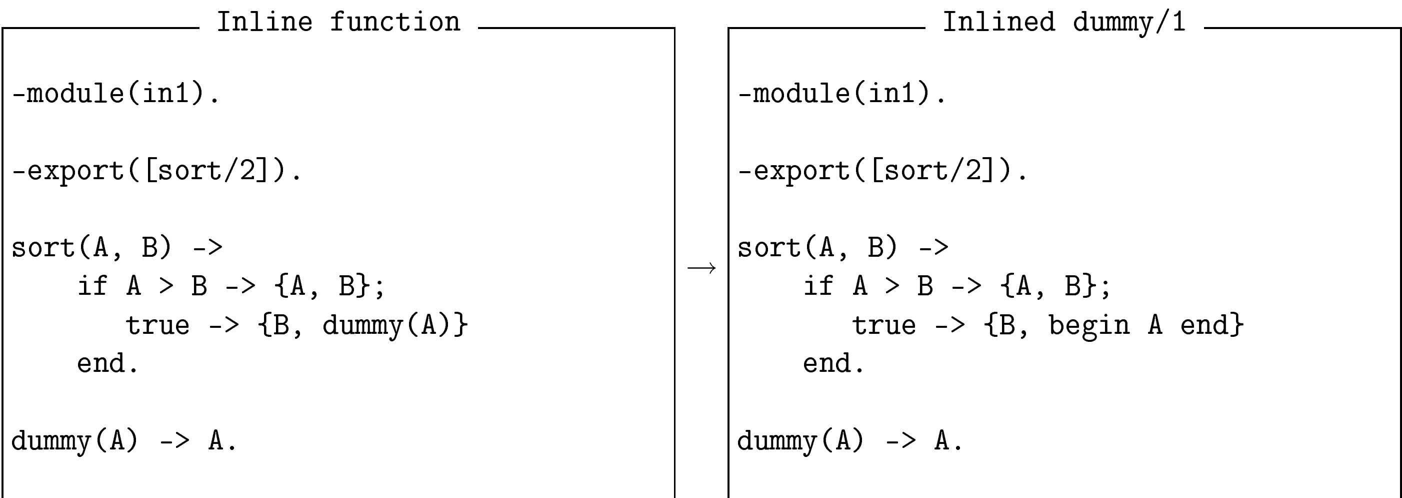
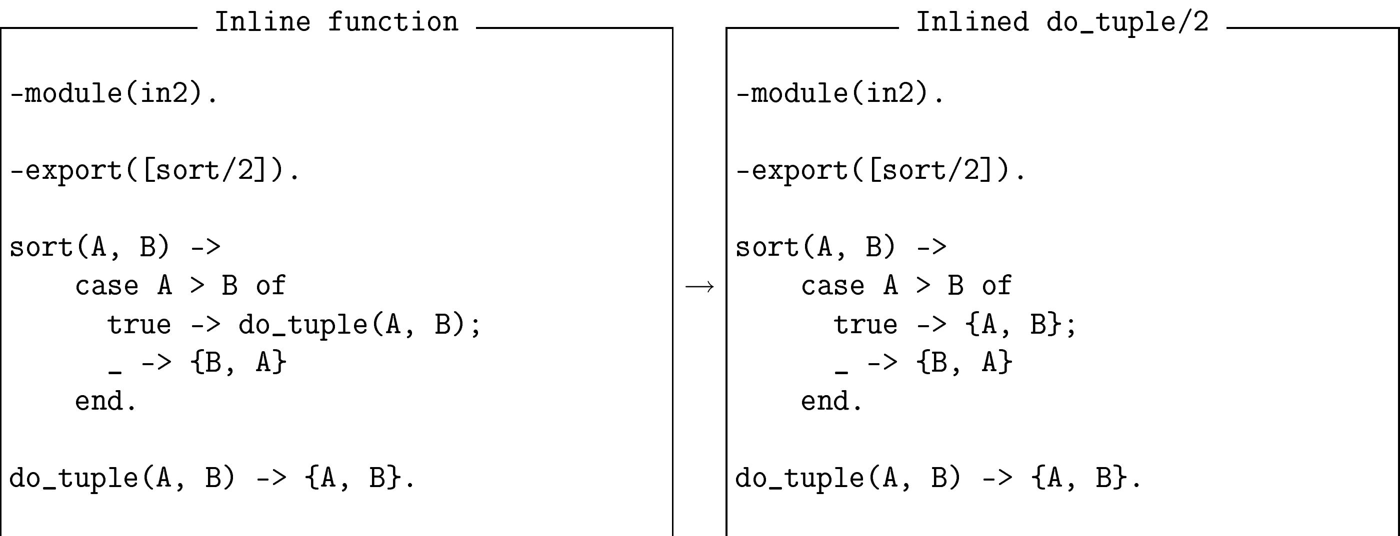


Fig. 3.1: Application in tuple

Discussion. The `dummy/1` application is in a tuple. This application is inlined and the body is replicated and put into the begin-end block. Renaming variables is not necessary.

3.3.2 Application in clause

The name of the application which the inline will be applied to is `do_tuple/2`. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in2).

-export([sort/2]).

sort(A, B) ->
    case A > B of
      true -> do_tuple(A, B);
      _ -> {B, A}
    end.

do_tuple(A, B) -> {A, B}.
```

→

Inlined do_tuple/2

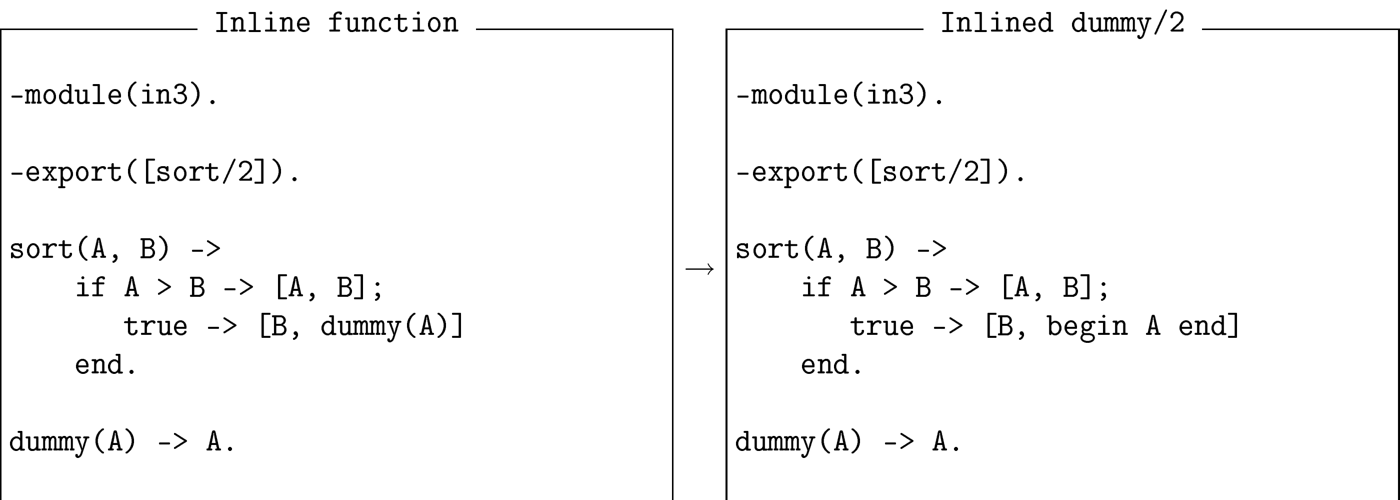
```
-module(in2).

-export([sort/2]).

sort(A, B) ->
    case A > B of
      true -> {A, B};
      _ -> {B, A}
    end.

do_tuple(A, B) -> {A, B}.
```

Fig. 3.2: Application in clause

Discussion. The `do_tuple/2` application is in the clause of the case expression. This application is inlined but did not put into the begin-end block, because the parent of the application is a clause. Renaming variables is not necessary.

3.3.3 *Application in list*

The name of the application which the inline will be applied to is `dummy/1`. The application is situated in a list. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in3).

-export([sort/2]).

sort(A, B) ->
    if A > B -> [A, B];
       true -> [B, dummy(A)]
    end.

dummy(A) -> A.
```

→

Inlined dummy/2

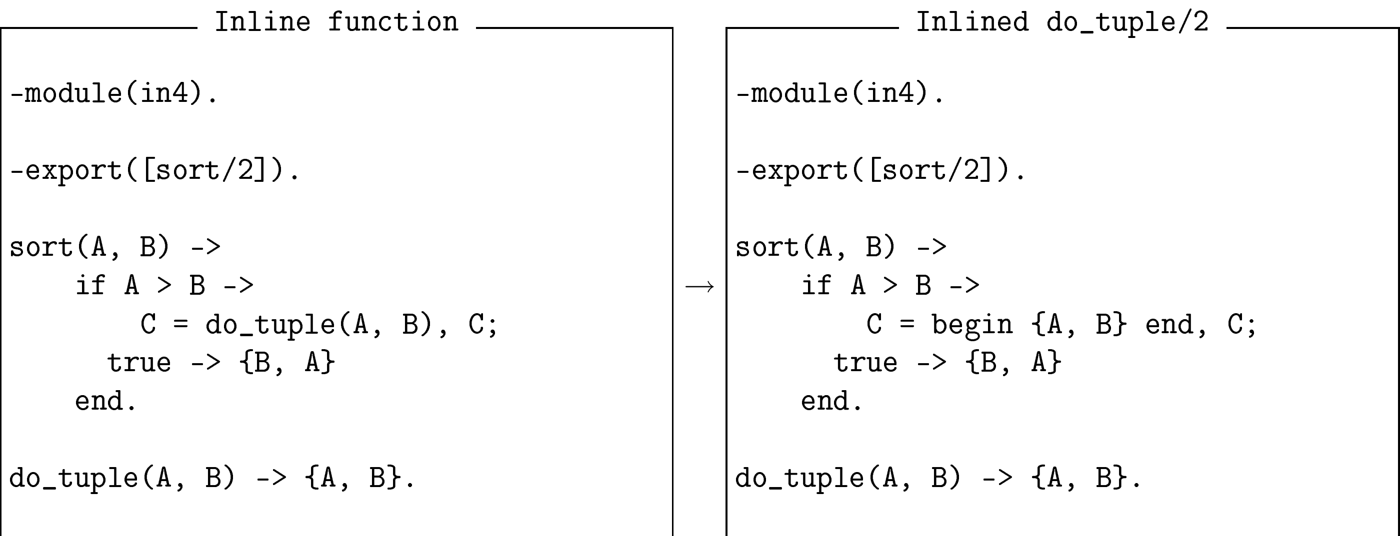
```
-module(in3).

-export([sort/2]).

sort(A, B) ->
    if A > B -> [A, B];
       true -> [B, begin A end]
    end.

dummy(A) -> A.
```

Fig. 3.3: Application in list

Discussion. The `dummy/1` application is an element of a list. This application is inlined and put into the begin-end block to preserve the meaning of the list. Renaming variables is not necessary.

3.3.4 Application in match expression

The name of the application which the inline will be applied to is `do_tuple/2`. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in4).

-export([sort/2]).

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

do_tuple(A, B) -> {A, B}.
```

→

Inlined do_tuple/2

```
-module(in4).

-export([sort/2]).

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

do_tuple(A, B) -> {A, B}.
```

Fig. 3.4: Application in match expression

Discussion. The `do_tuple/2` application is in a match expression. This application is inlined and put into the begin-end block. Renaming variables is not necessary.

3.3.5 *Application in case pattern*

The name of the application which the inline will be applied to is `is_greater/2`. The refactoring is acceptable, the result is shown below.

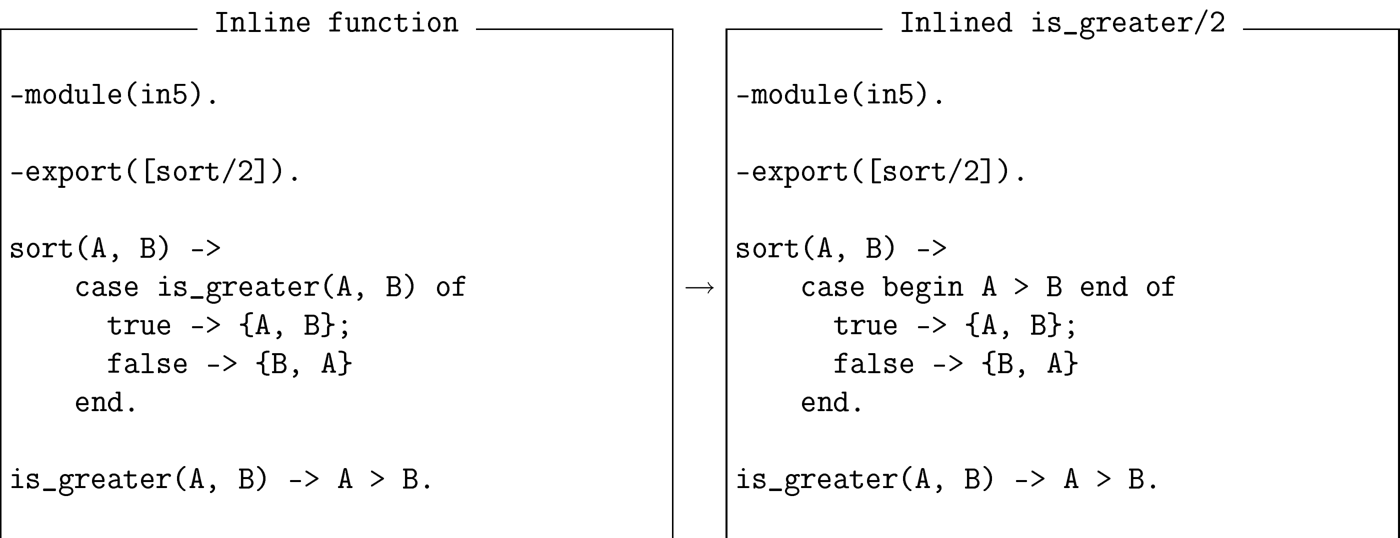
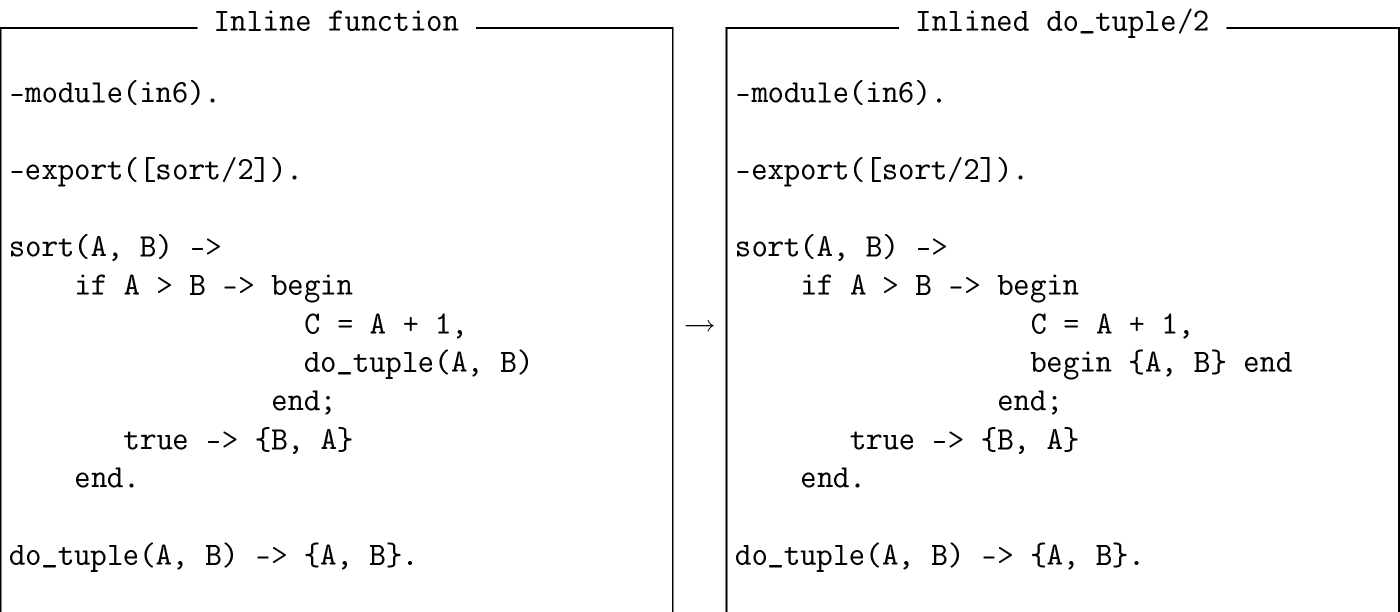


Fig. 3.5: Application in case pattern

Discussion. The `is_greater/2` application is in a case pattern. This application is inlined and put into the begin-end block. Renaming variables is not necessary.

3.3.6 Application in begin-end block

The name of the application which the inline will be applied to is `do_tuple/2`. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in6).

-export([sort/2]).

sort(A, B) ->
    if A > B -> begin
                    C = A + 1,
                    do_tuple(A, B)
                  end;
       true -> {B, A}
    end.

do_tuple(A, B) -> {A, B}.
```

→

Inlined do_tuple/2

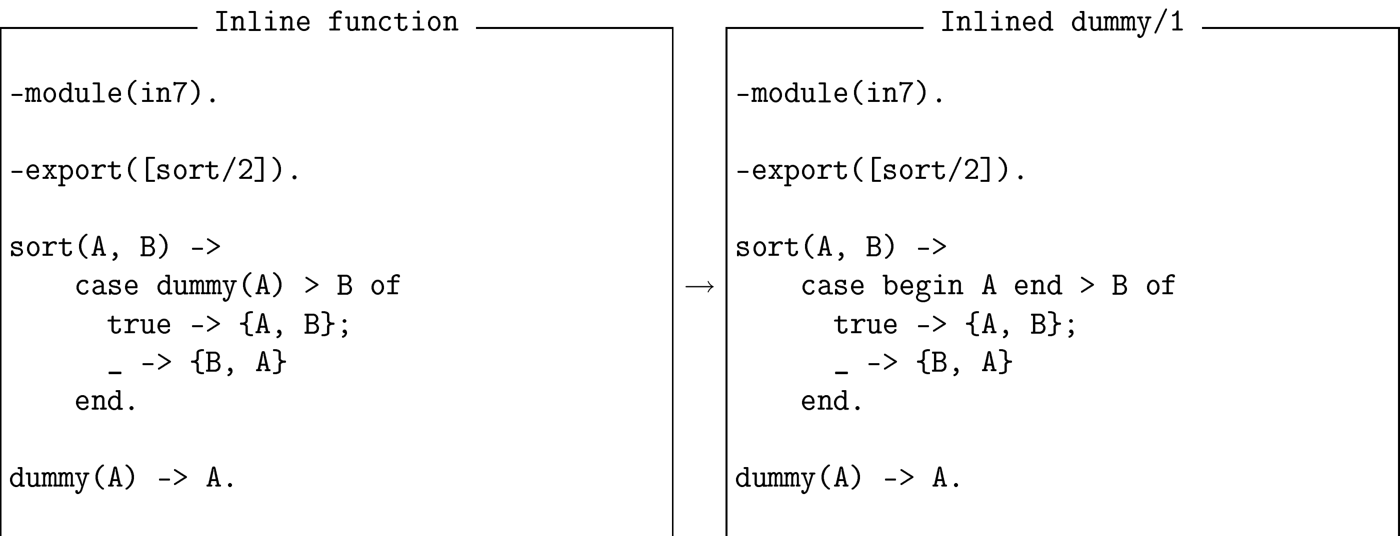
```
-module(in6).

-export([sort/2]).

sort(A, B) ->
    if A > B -> begin
                    C = A + 1,
                    begin {A, B} end
                  end;
       true -> {B, A}
    end.

do_tuple(A, B) -> {A, B}.
```

Fig. 3.6: Application in begin-end block

Discussion. The `do_tuple/2` application is in a begin-end block. This application is inlined and put into the begin-end block. Renaming variables is not necessary.

3.3.7 *Application in infix expression*

The name of the application which the inline will be applied to is `dummy/1`. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in7).

-export([sort/2]).

sort(A, B) ->
    case dummy(A) > B of
      true -> {A, B};
      _ -> {B, A}
    end.

dummy(A) -> A.
```

→

Inlined dummy/1

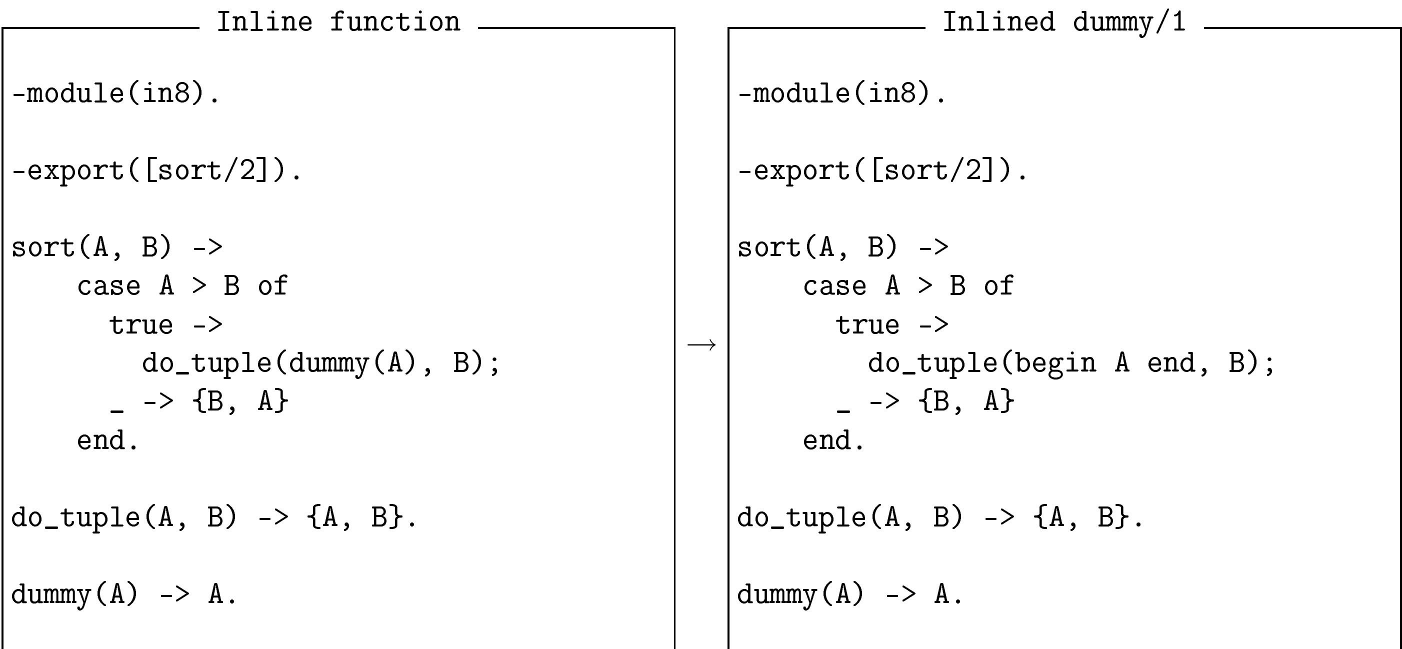
```
-module(in7).

-export([sort/2]).

sort(A, B) ->
    case begin A end > B of
      true -> {A, B};
      _ -> {B, A}
    end.

dummy(A) -> A.
```

Fig. 3.7: Application in infix expression

Discussion. The `dummy/1` application is in an infix expression. This application is inlined and put to the begin-end block. Renaming variables is not necessary.

3.3.8 *Application in the argument list of an application*

The name of the application which the inline will be applied to is `dummy/1`. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in8).

-export([sort/2]).

sort(A, B) ->
    case A > B of
      true ->
        do_tuple(dummy(A), B);
      _ -> {B, A}
    end.

do_tuple(A, B) -> {A, B}.

dummy(A) -> A.
```

→

Inlined dummy/1

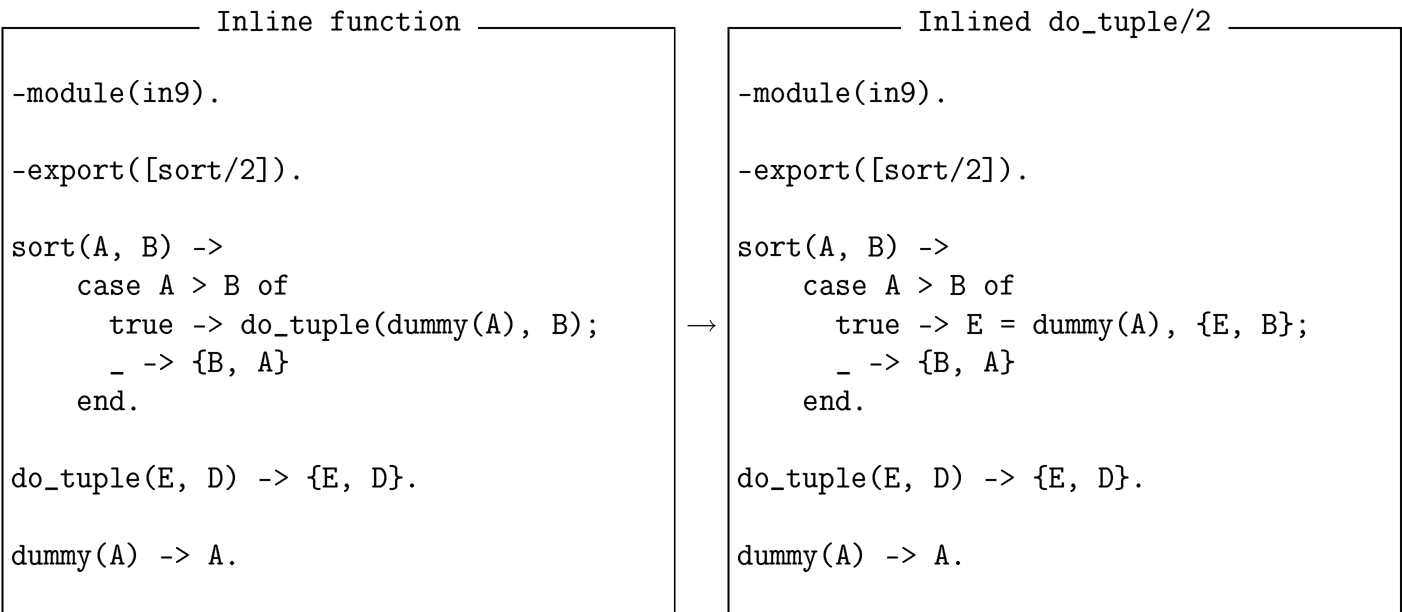
```
-module(in8).

-export([sort/2]).

sort(A, B) ->
    case A > B of
      true ->
        do_tuple(begin A end, B);
      _ -> {B, A}
    end.

do_tuple(A, B) -> {A, B}.

dummy(A) -> A.
```

Fig. 3.8: Application in the argument list of an application

Discussion. The `dummy/1` application is a parameter of the `do_tuple/2` application. This application is inlined and put to the begin-end block. Renaming variables is not necessary.

3.3.9 *Application with an application in the argument list*

The name of the application which the inline will be applied to is `do_tuple/2`. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in9).

-export([sort/2]).

sort(A, B) ->
    case A > B of
      true -> do_tuple(dummy(A), B);
      _ -> {B, A}
    end.

do_tuple(E, D) -> {E, D}.

dummy(A) -> A.
```

→

Inlined do_tuple/2

```
-module(in9).

-export([sort/2]).

sort(A, B) ->
    case A > B of
      true -> E = dummy(A), {E, B};
      _ -> {B, A}
    end.

do_tuple(E, D) -> {E, D}.

dummy(A) -> A.
```

Fig. 3.9: Application with an application in the argument list

Discussion. The `do_tuple/2` application is in the clause of the case expression and has an application `dummy/1` in its argument list. This application is inlined and added a match expression with the corresponding variable and application. Renaming variables is necessary: D $\rightarrow$ B.

3.3.10 Application in list generator

The name of the application which the inline will be applied to is `create_list/2`. The refactoring is acceptable, the result is shown below.

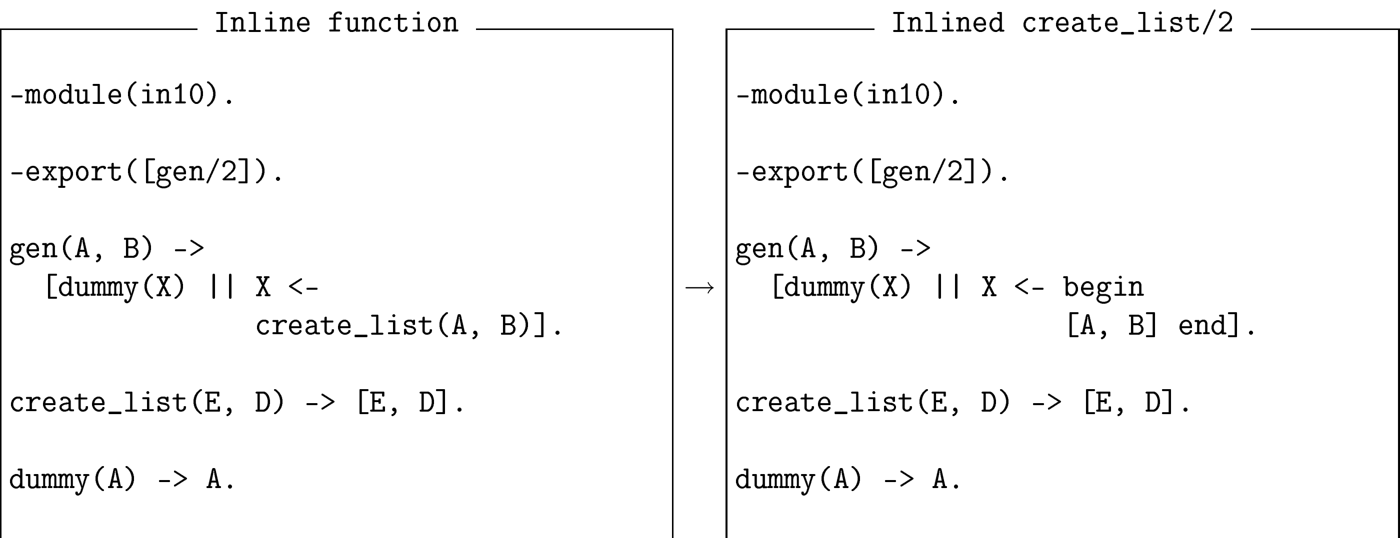


Fig. 3.10: Application in list generator

Discussion. The `create_list/2` application is situated in a list generator. This application is inlined and put to the begin-end block. Renaming variables is necessary: E $\rightarrow$ A and D $\rightarrow$ B.
`NOTE`: The inline of the application `dummy/1` in the head of the list generator would be denied, because this parent type is not supported.

3.3.11 Variable name collisions

The name of the application which the inline will be applied to is `sort_and_do_tuple/2`. The refactoring is not acceptable, because of the variable name collisions.

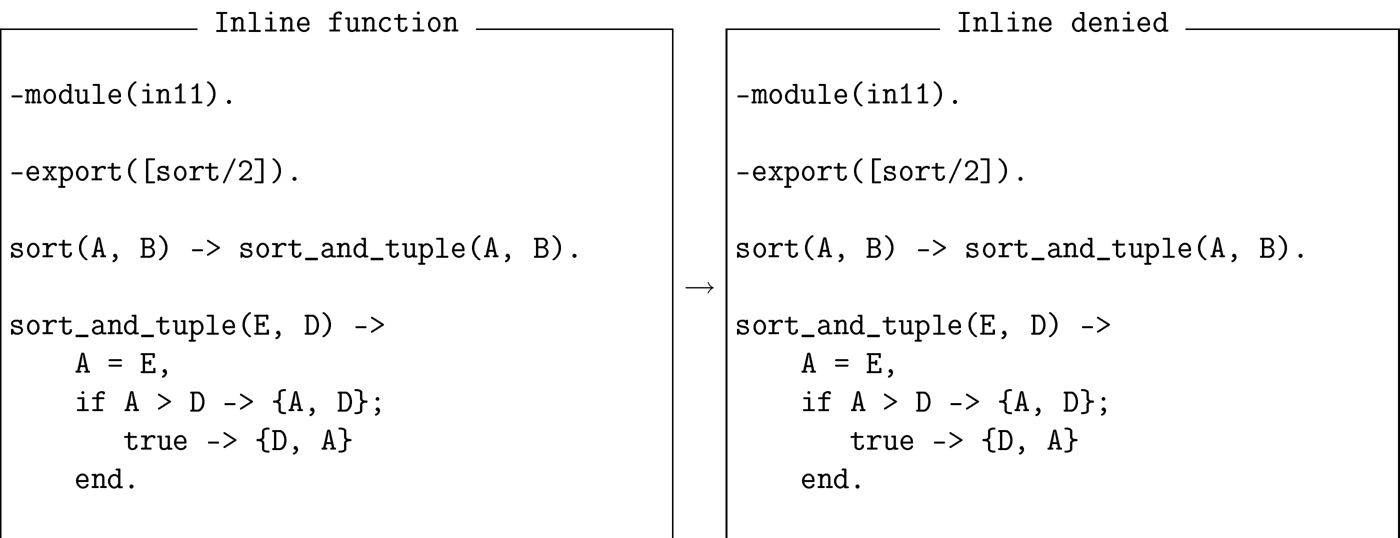
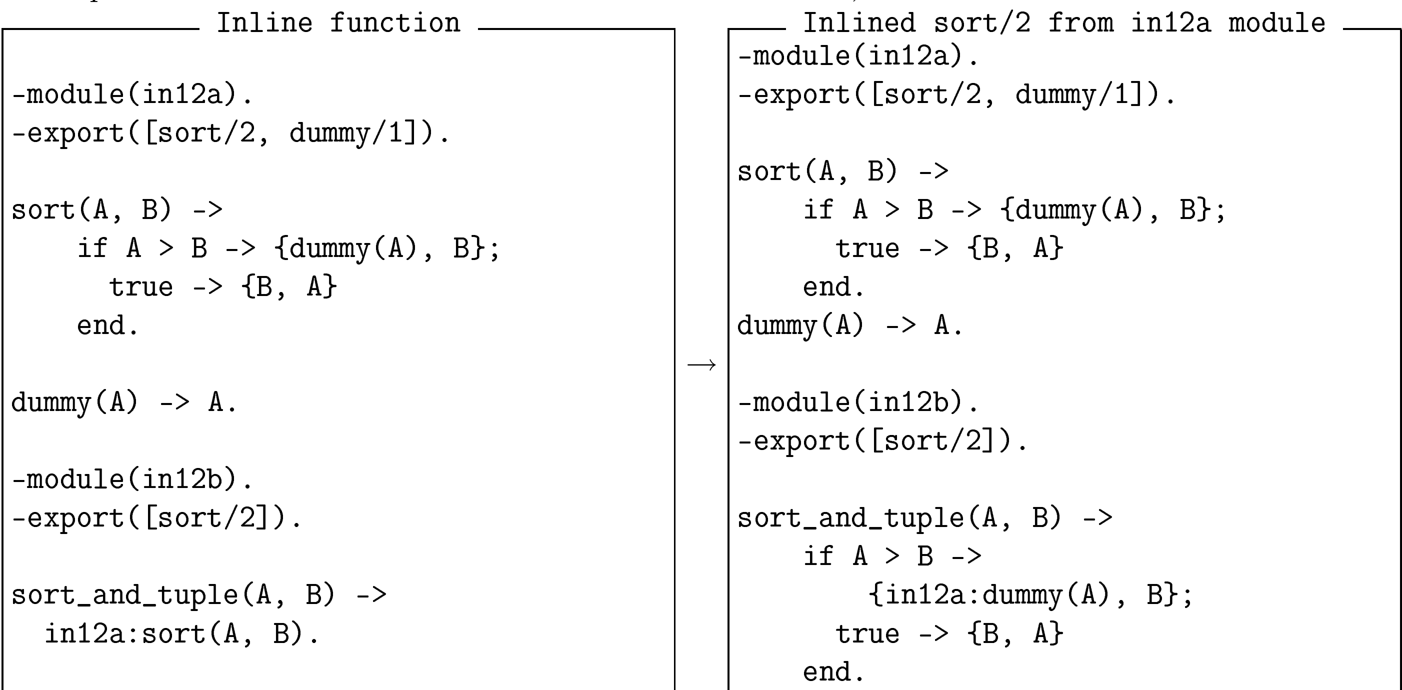


Fig. 3.11: Variable name collisions

Discussion. The `sort_and_tuple/2` function is in the body of the `sort/2` function. This application will not be inlined, because there is variable name collision between the variable names used in the body of the function and the variable names used in the scope of the application. Variable name collision: A.

3.3.12 *Application from other module with exported function in the body*

The name of the application which the inline will be applied to is `sort/2` from module `in12a`. The refactoring is acceptable if both of the modules are loaded in the database, else it is denied. The result is shown below.



Inline function

```
-module(in12a).
-export([sort/2, dummy/1]).

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

dummy(A) -> A.

-module(in12b).
-export([sort/2]).

sort_and_tuple(A, B) ->
  in12a:sort(A, B).
```

→

Inlined sort/2 from in12a module

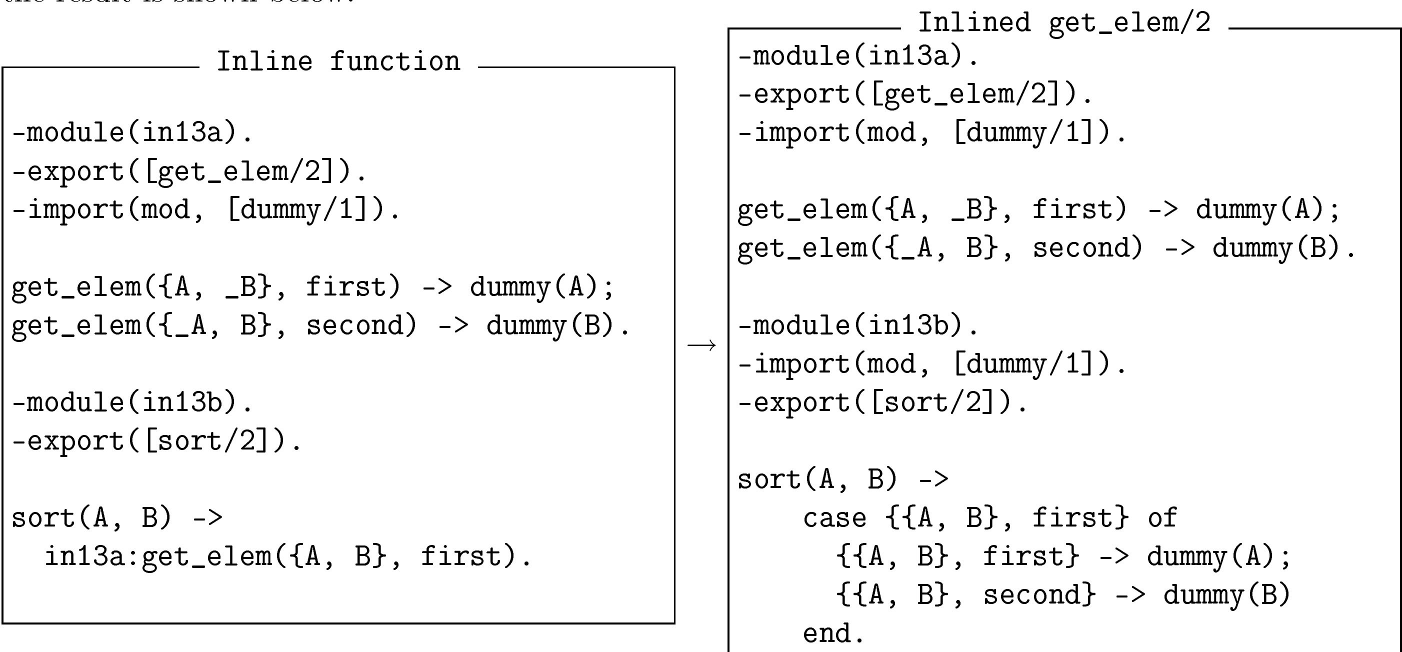
```
-module(in12a).
-export([sort/2, dummy/1]).

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

-module(in12b).
-export([sort/2]).

sort_and_tuple(A, B) ->
    if A > B ->
        {in12a:dummy(A), B};
      true -> {B, A}
    end.
```

Fig. 3.12: Application from other module with exported function in the body

Discussion. The `sort/2` application is in the body of the `sort_and_tuple/2` function . This application is inlined and put to the begin-end block. Renaming variables is not necessary.

3.3.13 *Application from other module with two clauses and with elemental in argument list*

The name of the application which the inline will be applied to is `get_elem/2`. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in13a).
-export([get_elem/2]).
-import(mod, [dummy/1]).

get_elem({A, _B}, first) -> dummy(A);
get_elem({_A, B}, second) -> dummy(B).

-module(in13b).
-export([sort/2]).

sort(A, B) ->
  in13a:get_elem({A, B}, first).
```

→

Inlined get_elem/2

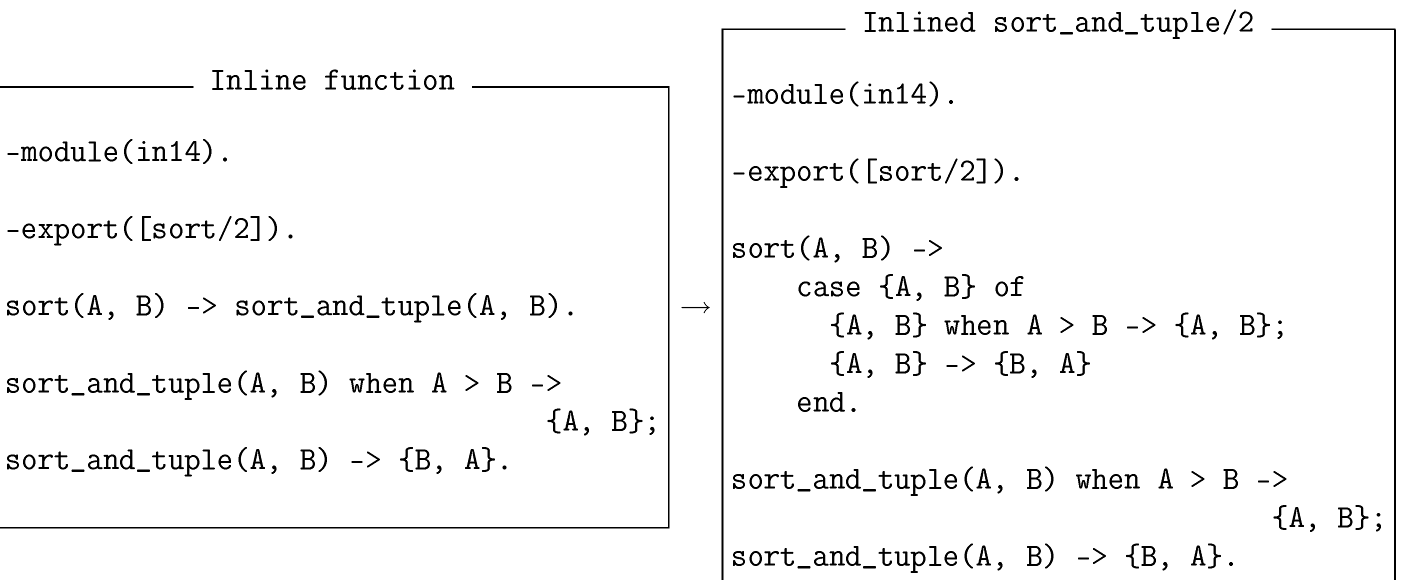
```
-module(in13a).
-export([get_elem/2]).
-import(mod, [dummy/1]).

get_elem({A, _B}, first) -> dummy(A);
get_elem({_A, B}, second) -> dummy(B).

-module(in13b).
-import(mod, [dummy/1]).
-export([sort/2]).

sort(A, B) ->
    case {{A, B}, first} of
      {{A, B}, first} -> dummy(A);
      {{A, B}, second} -> dummy(B)
    end.
```

Fig. 3.13: Application from other module with two clauses and with elemental in argument list

Discussion. The `get_elem/2` application is in the body of the `sort/2`. This application is inlined and created a case expression from the clauses of the function and the import function `dummy/1` is added to the import list. Renaming variables is not necessary.

3.3.14 Application with two clauses and with guard expression

The name of the application which the inline will be applied to is `sort_and_tuple/2`. The refactoring is acceptable, the result is shown below.



Inline function

```
-module(in14).

-export([sort/2]).

sort(A, B) -> sort_and_tuple(A, B).

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

→

Inlined sort_and_tuple/2

```
-module(in14).

-export([sort/2]).

sort(A, B) ->
    case {A, B} of
      {A, B} when A > B -> {A, B};
      {A, B} -> {B, A}
    end.

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

Fig. 3.14: Application with two clauses and with guard expression

Discussion. The `sort_and_tuple/2` application is in the body of the `sort/2` function. This application is inlined as a case expression with the clauses of the function. Renaming variables is not necessary.

4. SUMMARY

4.1 Summary

In this thesis I have described the using and the implementation of the inline function refactoring step.

First I described refactoring in general, the already existing refactoring tool and the main goal of our project.

The second chapter is about the installation for Windows, for Linux operating systems, the using of the tool, the general overview of the inline refactoring step, the conditions of applicability and the algorithm by using flowchart diagram.

The third chapter is the development manual. There is described the module itself, the exported and the local functions in detail.

4.2 Related work

The designing of the new tool is in progress. The inline function will be implemented in the new tool.

4.3 Developing the tool

Doing the tool interactive it can make the function more usable.

I mean :

- If there are variable name collisions, ask the user for new variable names.
- If there are local functions in the body, ask the user to solve it or not by exporting these functions.
- If there are macro and record references in the body of the function, ask the user to import the definitions or not.

LIST OF FIGURES

2.1 Inline function `temp/1` in function `double/1`. 10
2.2 The structure of the refac_inline_fun module 13
2.3 Structure of perform refactoring function 14
2.4 Local function in the body of the `in2:sum/2`. 16
2.5 Inline of the `in2:sum/2` in function `in1:mul/2`. 16

3.1 Application in tuple 60
3.2 Application in clause 61
3.3 Application in list 62
3.4 Application in match expression 63
3.5 Application in case pattern 64
3.6 Application in begin-end block 65
3.7 Application in infix expression 66
3.8 Application in the argument list of an application 67
3.9 Application with an application in the argument list 68
3.10 Application in list generator 69
3.11 Variable name collisions 70
3.12 Application from other module with exported function in the body 71
3.13 Application from other module with two clauses and with elemental in argument list 72
3.14 Application with two clauses and with guard expression 73

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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