periodica polytechnica

Electrical Engineering
51/3-4 (2007) 75–84
doi: 10.3311/pp.ee.2007-3-4.02
web: http://www.pp.bme.hu/ee
© Periodica Polytechnica 2007

RESEARCH ARTICLE

Refactoring Erlang programs

László Lövei / *Zoltán* Horváth / *Tamás* Kozsik / *Anikó* Víg / *Tamás* Nagy

Received 2007-10-03

Abstract

This paper presents the model, the design principles and the prototype of a refactoring toolset for Erlang programs. With this toolset one can incrementally carry out programmer-guided meaning-preserving program transformations. Erlang is a mostly dynamically typed language, and many of its semantical rules are also dynamic. Therefore the main challenge in this research is to ensure the safety of (the statically performed) refactoring steps. The paper analyses the language constructs of Erlang with respect to refactoring.

A novelty of the presented approach is that programs are represented, stored and manipulated in a relational database. This feature makes it possible to express refactoring steps in a fairly compact and comprehensible way.

The proposed software development environment with the integrated refactoring tool provides multiple editing modes. These editing modes support changes ranging from fully controlled (allowing only meaning-preserving transformations) to uncontrolled (editing program text freely). Transformations are performed more safely and efficiently in an editing mode with higher control.

Keywords

Erlang · refactoring tool · semantical analysis · functional language

Acknowledgement

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

László Lövei
Department of Programming Languages and Compilers, ELTE, Hungary
e-mail: lovei@inf.elte.hu

Zoltán Horváth
Tamás Kozsik
Anikó Víg
Tamás Nagy

1 Introduction

The phrase “refactoring” stands for program transformations that preserve the meaning of programs [11]. Such transformations are often applied in order to improve the quality of program code: make it more readable, satisfy coding conventions, prepare it for further development etc. Simple refactorings are used by developers almost every day; they rename variables, introduce new arguments to functions, or create new functions from duplicated code. The object-oriented paradigm is especially well suited for refactoring-supported programming. In this area refactoring has already appeared in programming methodologies [4] and it is used heavily in the industry.

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

However, it *is* possible to perform most refactoring steps in an automated way with an appropriate software tool—a tool that is aware of the syntactic and semantical rules of the programming language in use. Such tools exist for many programming languages [10], and modern programming environments often incorporate such capabilities [5, 8]. Refactoring in functional languages is not really wide-spread yet, but there are many ongoing researches on the topic. For the functional programmer, the only full-featured refactoring tool is HaRe [14], which provides refactoring capabilities for Haskell programs within the editors Emacs and VIM. A prototype of a refactoring tool for Clean is also available [17]. The work presented in this paper has an approach similar to that of that prototype tool.

The goal of this paper is to describe the design highlights of a refactoring tool for the functional programming language Erlang (and for the Erlang/OTP environment). The focus is on

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

A novelty of the presented approach is that the strong dynamic nature of Erlang programs is handled by static analysis, and that programs are represented, stored and manipulated in a relational database. This feature makes it possible to express refactoring steps in a fairly compact and comprehensible way. This paper describes a prototype of the designed refactoring tool. During the development of this prototype some well-known refactoring steps were analysed and implemented, namely renaming variables, renaming functions and reordering function arguments. In the future a broad selection of refactoring steps will be added into the refactoring tool.

2 Refactoring in Erlang

Erlang/OTP [2] is a functional programming language and environment developed by Ericsson, designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics (like telecommunication systems). The core Erlang language consists of simple functional constructs extended with message passing to handle concurrency, and OTP is a set of design principles and libraries that support building fault-tolerant systems [1]. The language has a very strong dynamic nature that partly comes from concurrency and partly from dynamic language features.

From the refactoring point of view, the most important characteristic of a programming language is the extent of semantical information available by static analysis. As Erlang is a functional language, most language constructs can be analysed easily. Side effects are restricted to message passing and built-in functions[1], variables are assigned a value only once in their lifes, and the code is organized into modules with explicit interface definitions and static export and import lists. An unusual feature (at least in a functional language) is that variables are not typed statically, they can have a value of any data type, but even that does not make the life of a refactor tool much harder.

[1]Built-in functions, or BIFs, are functions that are implemented in the runtime system.

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

Another kind of problem is the possibility of running dynamically created code. The most prominent example of this is the `erl_eval` standard module, which contains functions that evaluate Erlang code constructed at runtime. This functionality is clearly out of the scope of a static refactoring tool, but there are other constructs similar to this that are widely used and should be covered, like the `spawn` function that starts the evaluation of a function in a new process (and the function name and arguments might be constructed at runtime), or the `apply` function that calls a function (with the same runtime-related problems). The normal function call syntax has some runtime features too: variables are allowed instead of static module or function names.

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

In the following, we try to outline the extent of the language features that we plan to support with the tool. As we have investigated three different refactorings in detail so far, various language features are described through examples originating from them.

2.1 Transformations on variables

One of the most simple refactorings is the *rename variable* transformation. Its goal is obvious from the name, and it is one of the refactorings that can be supported by a tool without restrictions. The only semantical information that is necessary for it is the scope and visibility of the variables. The exact rules for them are given in the form of input and output contexts for every language construct [3], which is hard to follow and not really helpful in defining the conditions of a refactoring, so we have created a more suitable definition which is given below.

2.1.1 Rules for variable scoping

In Erlang, variables have a name (always beginning with a capital letter or an underscore) and a value bound to them (which never changes during the life of the variable). For our purposes, we define the *scope* of a variable as a region of the program text where the variable is bound to its name, and a variable is *visible* in a region where its name can be used to refer to the variable.

The scope of a variable is always limited to a function clause, there are no global variables. 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. Pattern matches are used in heads of function clauses (every formal argument is a pattern match), heads of clauses in function expressions, pattern matching expressions, `case` and `receive` constructs and list comprehension expressions.

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

- A function expression creates a new scope for its variables. When an existing variable name is used in one of the formal arguments, it creates a new variable that shadows the existing one.
- Variables created in a `catch` expression are unsafe to be used outside that expression, so they are visible only inside the innermost enclosing `catch` expression.
- Variables that are created inside a branch of a branching expression (those are: `if`, `case` and `receive`), but are not bound to a value in every branch, are unsafe to be used outside that expression, so they are not visible outside the expression.
- Variables created in the timeout expression of a `receive` construct's `after` branch, but are not bound in the body of that branch, are not considered to be bound in that branch at all.
- Inside list comprehensions, every generator introduces a new scope, so existing variable names bound in them refer to new variables that shadow the existing ones.

2.1.2 Renaming a variable

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

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

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

The third requirement can be violated by breaking the rule of disjoint visibilities, that is, re-using an existing variable name inside its scope. This situation can be detected easily, and the tool can either deny to perform the transformation, or offer a compensation step (viz. rename the existing variable with the problematic name first) as illustrated in Fig. 1.

There are situations when renaming a variable does not change the way the program works, but it does influence the readability of the code—possibly in a negative way. Renaming a variable can introduce shadowing, as illustrated in the example in Fig. 2. The example makes use of the built-in function `length` and the higher-order function `filter` from the standard library module `lists`.

If `L`, the second argument of function `multiplicity`, is renamed to `X`, or `X`, the argument of the local function expression is renamed to `L`, then the argument of the local function shadows the second argument of the enclosing function within the body of the local function. Such renamings can derogate the readability of the resulting code. Therefore a refactoring tool might want to issue a warning to the programmer.

2.2 Refactoring functions

The next simple transformation is *rename function*, which seems similar to *rename variable*, but different problems arise when dealing with it. Function visibility is much simpler than variable visibility: a named function is either exported, in this case it is visible from every module, or not exported, and then it is visible only in the defining module. There are no function name hiding, embedded functions or any other kinds of complication. Here the real problem comes from the already mentioned dynamic language constructs. While a variable can only be used statically, a function name can be constructed dynamically, and functions can be called using built-in functions.

2.2.1 Functions in Erlang

Functions are always defined in modules, and they are identified by three components: the module name, the function name and the arity of the function. The module and function names are so-called *atoms*, which are string-like data terms usually used as labels throughout the code. Two functions with the same name but different arities are permitted.

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

Functions are exported using the `export` module attribute. An export list contains function names together with arities (the syntax is `function/arity`), and defines the interface of the module. Outside the defining module an exported function can be accessed by supplying the module name with the function

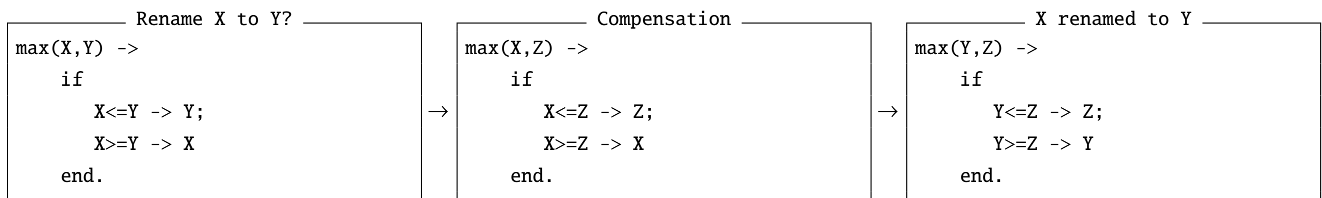

Fig. 1. The use of a compensation step when renaming variable `X` to `Y`

Count the occurrences of value V in list L

```
multiplicity(V,L) ->
      length(lists:filter( fun (X) -> X==V end, L )).
```

Fig. 2. Example of variable shadowing: `X` can be renamed to `L` without semantical change, but it is misleading.

name (the syntax is `module:function(args)`). The module name is an expression again, that must result in an atom. Module names passed as parameters are used throughout the OTP to access callback modules. Although OTP code contains a lot of module names supplied as variables, this is not usual in normal application code.

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

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

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

2.2.2 Scope of function-related refactorings

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

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

Constructs with limited support. Function calls using the built-in functions mentioned above, when used with explicit module and function names, can be found by static analysis, and the only missing information is the function arity. In the case of function renaming this can be handled depending on the situation:

- When the function to be renamed has no variants with the same name but different arities, the transformation can be carried out without problem.
- When there are functions with the same name but different arities, renaming one of these functions will inevitably change the semantics: calls to functions with different names cannot be handled by one `apply` call. Our proposed solution is to deny this kind of renaming, and provide a variation of the refactoring that renames all of the functions with the same name. This often meets a good programming style where functions with the same name do the same thing, so they should be renamed together.
- There is one special case, when the argument list is given as a static list skeleton. This case is essentially the same as a normal function call, it just needs a bit more work to be recognized.

For the second case, there is no generic solution that works with every kind of refactoring. Usually this construct can be recognized and an error message or a warning can be reported.

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

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

2.2.3 Transformations on function arguments

The last refactoring that we have investigated so far is called *reorder function arguments*. It represents a class of transformations where we change the way a function is called. It is easy to update a static function call according to this refactoring, but what is the case with the more dynamic constructs? When a built-in function is used for the function call and parameters are passed in a dynamically created list, it seems hard to change the order of the list members. The same applies to other refactorings where the number, the structure or the order of the arguments are modified.

Fortunately, we can solve this problem using functional features of Erlang. These transformations can be described by functions which operate on lists: in the case of reordering, for example, the function should change the order of the list elements according to the refactoring. Such a function can be constructed by a so-called "explicit function expression", which can be inserted into the place of the function call, and can be applied to the list that contains the arguments of the refactored function (see Fig. 3). The problem of different functions with different arities is easily solved with this approach: the generated function expression uses pattern matching to decompose the argument list, and this is done only on lists with appropriate length.

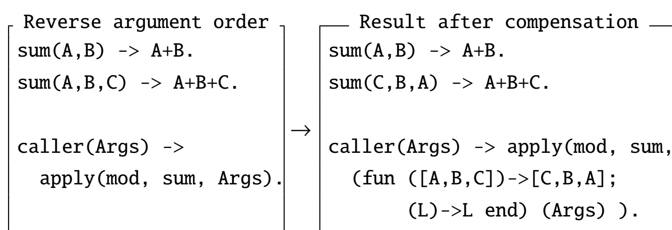

Fig. 3. Reorder arguments of a function that is called semi-dynamically.

2.3 Properties of subexpressions

The next set of transformations that we investigate concerns subexpressions. Such refactorings are *generalization* when a subexpression of a function is replaced with an argument, *merge duplicated subexpressions* when two or more instances of a subexpression are replaced with a variable that stores the value of the subexpression, and *extract function* when a subexpression is substituted with a call to a new function that uses the subexpression as body.

This kind of refactoring, when subexpressions are moved around in the code, has two main sources of problems. The first is the possibility of having a side effect in a subexpression, and the other is that variable visibilities usually change during such a transformation. Fortunately, these problems can be detected statically and can be compensated in many cases.

Side effects. Most of the language constructs do not have any side effects, only message passing expressions and built-in functions. BIFs that can be used in guard conditions are guaranteed to be side effect free. Functions that use only side effect free constructs and calls to side effect free functions can be marked as side effect free too. Using this technique, every expression with a possible side effect can be tracked down.

When a subexpression has a possible side effect, it usually cannot be moved without problems. It cannot be pre-calculated and merged with other instances. Generally speaking, a variable cannot replace an expression with side effects. However, in some cases, it is feasible to incorporate the side effect in a function expression, and replace the original subexpression with the invocation of this new function expression – this technique can be used in generalization, like in Fig. 4.

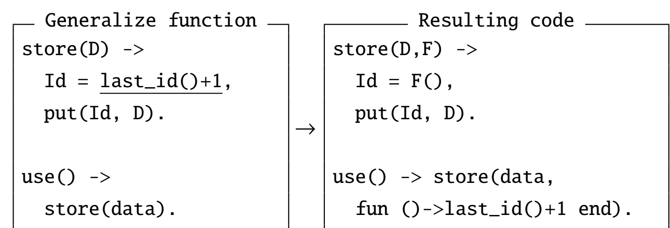

Fig. 4. Generalization on a subexpression that has a side effect.

Variable bindings. There are two possible problems with variables when we try to move a subexpression: variables bound outside and used inside the expression, and variables bound inside and used outside the expression change their semantics when the subexpression is placed in another context. These situations can be detected using the definition of scope and visibility given in 2.1.1.

A subexpression *depends* on a variable used inside it when the scope of the variable begins outside the subexpression. To preserve the semantics of such a subexpression while moving it to another context, the bindings must be maintained for every variable the subexpression depends on. The natural way to do this is to create a function using the variables as parameters and the subexpression as body, as shown in Fig. 5. This approach works with generalization too, but a function expression is generated instead of a named function.

A much more problematic situation is when the subexpression to be moved contains a binding for a variable that is used outside the subexpression. Such variable bindings are hard to reconstruct; it seems natural to include the intended value of the

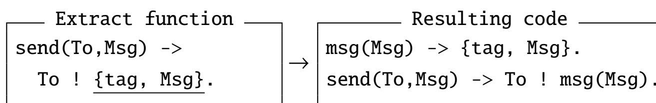

Fig. 5. Extract a subexpression that depends on a variable.

variable in the result of the subexpression, and use the return value to maintain the binding, but this approach does not work when the subexpression is used in a function composition. Furthermore, to export a variable binding in such a way is quite obscure; it seems more practical to deny the refactoring in this case.

3 The refactoring tool

Traditionally programs are stored and maintained in textual format—but even in this case they have got a certain structure behind the scene. During project development programmers work with a set of files stored in different directories of a filesystem (or a network of filesystems) maintaining them via different services of some "file manager programs". 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", which is able to handle the elements of programs in a more disciplined way. An adequate tool for storing and maintaining information is a database manager. The approach presented here—similarly to [7]—is to represent programs in relational databases in order to facilitate refactoring.

3.1 Refactoring using a 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 syntactical structure of the analysed program code. The semantical rules of the programming language can be supported by the extension of ASTs with additional information. 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. Our approach is to represent the resulting structure as a set of relations in a relational database, and use SQL to manipulate it.

In the relational database representation, there are two kinds 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 semantical 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 Fig 6. This is one clause of a function that computes the greatest common divisor of two numbers, the whole module and it's AST is presented in Appendix A. Each non-leaf (non-terminal) 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 Fig. 6 is given in Figs. 10 and 11 in the Appendix).

The database representation of the AST is illustrated in Fig. 6. 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.[2] In order to make information retrieval faster, an auxiliary table, `node_type` was introduced. This table binds the identifier of each parent node to the table corresponding to its type.

Semantical information about Erlang programs are stored in tables such as `var_visib` and `fun_visib`. The table `var_visib` stores visibility information on variables, namely which occurrences of a variable name identify the same variable. This table has columns that contain the identifier of a variable occurrence and the identifier of that variable's first occurrence. The `var_visib` table contains the following pairs regarding the code in Fig. 6: (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 *rename variable* transformation is supported with a further table, called `forbidden_names`, which describes names that are not allowed to use 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.

Renaming a variable is performed in the following way. The programmer selects the *rename variable* transformation and specifies that the variable it line 4 at column 16 (which happens to be an occurrence of the first formal argument, `N`, of the second alternative of function `gcd`) should be renamed to `K`. The refactoring tool, using a table containing position information of every node in the AST, makes sure that a variable occurs at the given position: a variable occurrence identified with 17. The `var_visib` table tells the tool that the first occurrence of the concerned variable is identified with 15. Now assume that table `forbidden_names` does not prohibit the use of `K` as a variable name. Another table, containing information on the scope of variables, helps the tool verify that the requested transforma-

[2] 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.

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

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

Fig. 6. The source code and database representation of the example function clause.

Rename variable N to K

```
UPDATE names SET name='K' WHERE id IN
(SELECT occurrence FROM var_visib WHERE first_occurrence=15)
```

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

Fig. 7. An SQL statement for renaming a variable and the resulting code

tion is indeed safe: there is no variable named K in the scope (identified with 30) of variable 15 yet. Furthermore, there is no nested scope within clause 30 that introduces a variable K, hence no warning about shadowing variables should be issued by the refactoring tool. The final step, namely the realization of the transformation, and its result is illustrated in Fig. 7.

3.2 Design principles for the user interface

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

The refactoring tool will be interactive. It will be started within the programmers' 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.

When being active, the refactoring tool will support two different "editing modes". In the first mode, the programmer can choose from a set of predefined transformations. The parameters to an initiated transformation will be provided interactively, as well as further responses required by the tool (confirmation to safe, but unfortunate transformations, selection of compensation steps etc.). Editing the source code will be prohibited in this mode: this eliminates the need for frequent re-parsing of the code and rebuilding of the database. Hence a sequence of refactoring steps can be performed efficiently in this mode.

In the second mode, editing the program code will be allowed, but only in a restricted way. The goal of this editing mode is to enable the incremental maintenance of the database describing the program code. A set of activities (such as "insert a function") will be available for the programmer, but all these editing activities result in local changes of the AST, hence only a small fraction of the program code need to be re-parsed.

3.3 The prototype tool

We have implemented a prototype of the Erlang refactoring tool, supporting a limited set of transformations. The refactoring tool is written in Erlang and SQL. The Erlang ODBC interface is used to access a MySQL database server, which contains the databases describing the refactored Erlang sources. The front-end of the refactoring tool is built into Emacs [9], while the back-end of the tool is connected to Emacs through Distel [6].

Distel extends Emacs Lisp with Erlang-style processes and message passing, and the Erlang distribution protocol. It supports writing Emacs Lisp processes that communicate with Erlang processes in Erlang nodes, therefore it facilitates the development of Emacs user-interfaces to Erlang programs.

For parsing Erlang source code "epp_dodger", a modified version of the standard Erlang parser ("epp") is used, which skips pre-processing of Erlang source files. Another necessary modification was to extend the scanner, "erl_scan", to keep track of precise column information. Processing of comments had to be adjusted similarly by enhancing "erl_recomment". Finally, a technical problem related to the placement of pre- and post-comments (an Erlang feature) into the AST had to be solved.

4 Related work

Refactoring was first recognized as a distinct programming technique of its own in Fowler's refactoring bible [11] which addressed a wide range of refactorings for object-oriented software providing examples in Java. Most research activites in this field focus on object-oriented environments, an exhaustive survey on the existing techniques and formalisms is [15].

Tool support for refactoring was first provided by the refactoring browser for Smalltalk [16]. Many tools are available for Java, sometimes embedded into a development environment (e.g. Eclipse [8], JFactor, IntelliJ Idea, Together-J etc.), and some for C# (ReSharper, C# Refactory) and C++ (SlickEdit, Ref++). These tools support various kinds of renamings, extracting or inlining code, and manipulating the class hierarchy. There is a good summary of the available tools and a catalogue of well-known refactorings in [10].

Refactoring in functional languages has received much less attention. Haskell was the first functional language to gain tool support for refactoring, and so far the Haskell Refactorer prototype [14] is the only functional refactorer software that is actually usable in practice. Refactoring functional programs using database representation first appeared in [7] for the Clean language, and a standalone prototype is available [17] from this research.

Refactoring Erlang programs is a joint research with the University of Kent, building on experiences with Haskell and Clean. While we are sharing ideas and experiences, they are investigating a completely different implementation approach using traversals on annotated abstract syntax trees [13].

5 Conclusion

This paper analyses the Erlang programming language with respect to refactoring. The investigation is based on the "rename variable", the "rename function" and the "reorder function arguments" transformations. The main problem is to express static program transformations and statically computable conditions for such transformations in a dynamic environment like Erlang/OTP. The article introduces concepts like scoping and visibility with static semantical rules instead of the dynamic semantical rules found in the Erlang Reference Manual. It is shown that there are many situations when static analysis is not applicable, therefore a refactoring tool for Erlang must explicitly document which are the fully supported, partially supported, and unsupported language constructs, and which constructs are inherently hopeless to support.

The paper describes an approach to build a refactoring tool based on a relational database. The abstract syntax tree of an Erlang program and all the semantical information extracted from the program are represented as relations (tables) in the database. This representation makes it possible to formulate refactorings (both conditions and transformations) easily, at a high abstraction level, and implement them in SQL. Extending such a tool with further semantical analysis and further refactorings is also straightforward.

The design principles of the user interface of our refactoring tool are explained. This user interface provides different editing modes with different levels of control on the program text. An editing mode with higher level of control enables a more efficient execution of refactorings, because it ensures that after a modification only a smaller fraction of the code has to be reparsed and re-analysed.

5.1 Future work

The three refactorings presented in detail here are simple, but the concepts introduced by them provide deep analysis of Erlang source code. We believe that more complex transformations like "generalization" and "extract function" require the same analysis, and although their implementation needs much more work, the concepts introduced in the paper are sufficient for them. In Fowler's opinion a refactoring tool is "over the Rubicon" [10] when it has complete support for "extract function". Our short-term goal is to reach this stage. On the longer term, we plan to investigate transformations that require data flow analysis, like "convert tuple to record" or transforming message passing into standard library-based program design [12].

References

1 **Armstrong J**, *Making reliable distributed systems in the presence of software errors*, The Royal Institute of Technology, Stockholm, Sweden, 2003.
2 **Armstrong J., Virding R, Williams M, Wikstrom C**, *Concurrent Programming in Erlang*, Prentice Hall, 1996.
3 **Barklund J, Virding R**, *Erlang Reference Manual*, 1999. Available from http://www.erlang.org/download/erl_spec47.ps.gz.
4 **Beck K**, *Extreme Programming Explained*, Addison-Wesley, 1999.
5 *C# Refactory homepage*, available at `www.xtreme-simplicity.net/`.
6 *Distel: Distributed Emacs Lisp*, available at `http://fresh.homeunix.net/~luke/distel/`.
7 **Diviánszky P, Szabó-Nacsa R, Horváth Z**, *Refactoring via Database Representation*, The Sixth International Conference on Applied Informatics (ICAI 2004) (Csőke L, Olajos P, Szigetváry P, Tómács T, eds.), Eger, Hungary, 2004, pp. 129–135.
8 *Eclipse Project homepage*, available at `http://www.eclipse.org/`.
9 *GNU Emacs homepage*, available at `www.gnu.org/software/emacs/`.
10 *Martin Fowler's refactoring site*, available at `www.refactoring.com/`.
11 **Fowler M, Beck K, Brant J, Opdyke W, Roberts D**, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
12 **Király R**, *Transforming client-server based Erlang programs to Erlang OTP design*. Accepted to NetWorkshop 2007, 7 pages.
13 **Li H, Thompson S, Lövei L, Horváth Z, Kozsik T, Víg A, Nagy T**, *Refactoring Erlang Programs*, The Proceedings of 12th International Erlang/OTP User Conference, 2006.
14 **Li H, Thompson S, Reinke C**, *The Haskell Refactorer, HaRe, and its API*, Electronic Notes in Theoretical Computer Science **141** (2005), no. 4, 29-34.
15 **Mens T, Tourwe T**, *A Survey of Software Refactoring*, IEEE Transactions on Software Engineering **30** (2004), no. 2, 126-139.
16 **Roberts D, Brant J, Johnson R**, *A Refactoring Tool for Smalltalk*, Theory and Practice of Object Systems (TAPOS) **3** (1997), no. 4, 253–263.
17 **Szabó-Nacsa R, Diviánszky P, Horváth Z**, *Prototype Environment for Refactoring Clean Programs*, The Fourth Conf. of PhD Students in Computer Science (CSCS 2004), Volume of extended abstracts, 2004, pp. 113.

A Abstract syntax trees

In the following, the source code and the ASTs of the module used as an example in 3.1 is presented. The tree is split into multiple parts for easier reading. The figures show the result of the Erlang parser, extended with the database identifiers for the non-leaf nodes written 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).
```

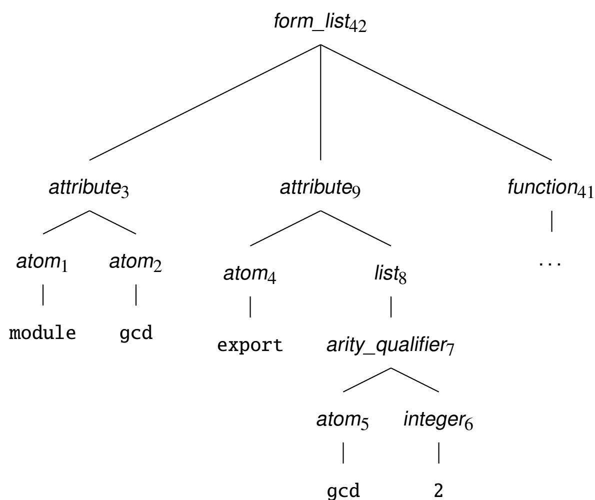

Fig. 8. The AST of gcd (Part 1)

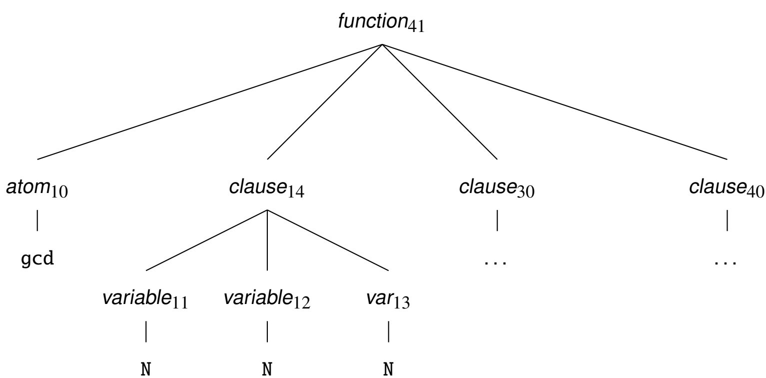

Fig. 9. The AST of gcd (Part 2)

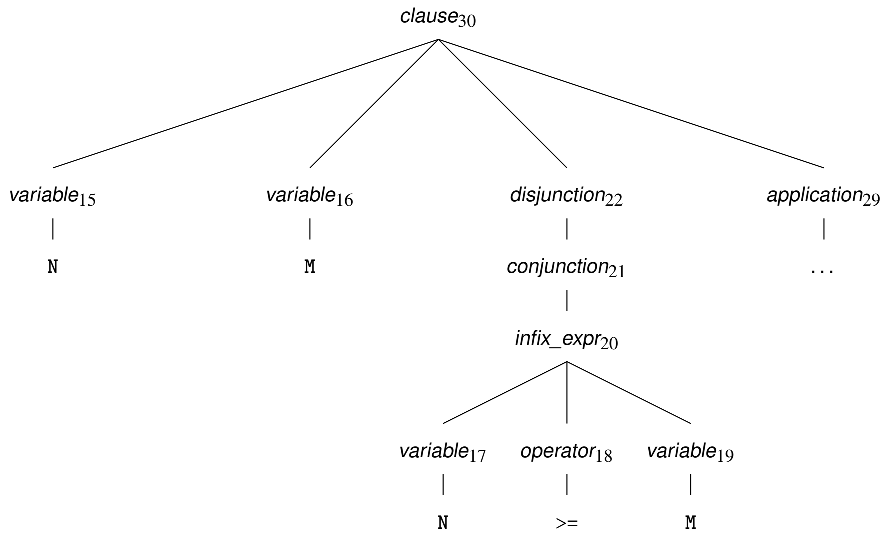

Fig. 10. The AST of `gcd` (Part 3)

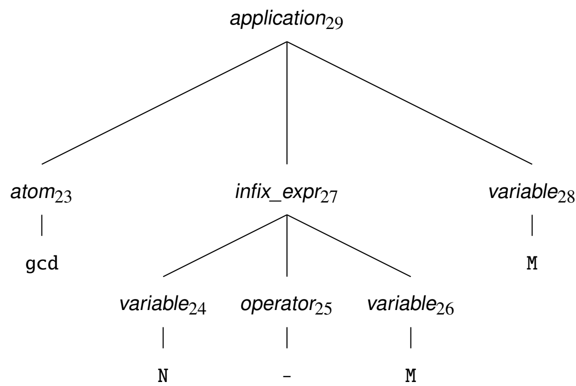

Fig. 11. The AST of `gcd` (Part 4)

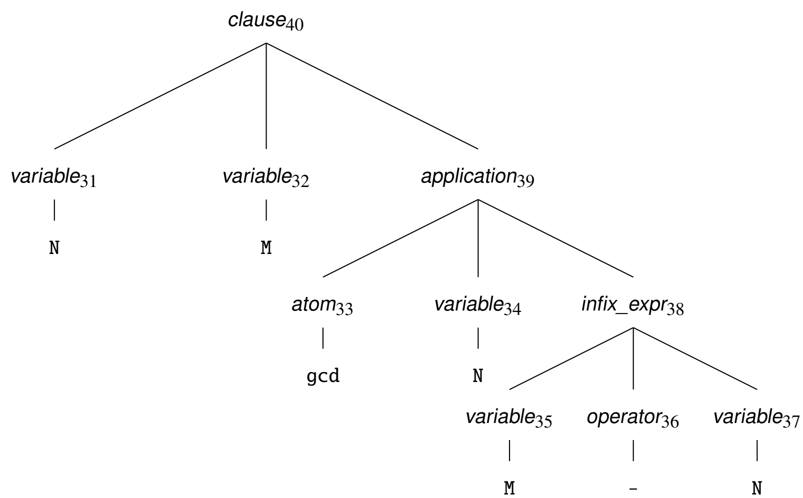

Fig. 12. The AST of `gcd` (Part 5)