# Refactoring in Erlang, a Dynamic Functional Language*

László Lövei, Zoltán Horváth, Tamás Kozsik, Roland Király,
Anikó Víg, and Tamás Nagy
Eötvös Loránd University, Budapest, Hungary

## Abstract

Refactoring in object-oriented languages has been well studied, but functional languages have received much less attention. This paper presents our ideas about refactoring in Erlang, a functional programming language developed by Ericsson for building telecommunications systems. The highlights of our work is dealing with the strong dynamic nature of Erlang and doing program manipulations using a relational database.

## 1 The Erlang programming language

Erlang/OTP [1] 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 supports building fault-tolerant systems.

Erlang is a functional language which means that a program is run by successively applying functions. Branches of execution are selected based on pattern matching of data and conditional expressions, and loops are constructed using recursive functions. Variables are bound a value only once in their life, they cannot be modified. Most constructs are side effect free, exceptions are message passing and built-in functions (BIFs).

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

The speciality of Erlang is its strong dynamic nature. Variables are dynamically typed, there is no compile time type checking. The identifiers of functions are of a special data type called *atom* and they can be generated at run-time and passed around in variables. Execution threads are also created at run time, and they are identified by a dynamic system.

The challenge in building an Erlang refactoring tool is to cover as wide area of language constructs as possible by static (compile-time) analysis, and to identify the exact conditions when we can guarantee behaviour-preserving transformations.

## 2 Refactoring in Erlang

While refactoring in object-oriented languages has been well studied [3], functional languages have received much less attention, and most work is oriented towards pure functional languages with a strict type system. Our work has been focused on those refactorings that are applicable in Erlang as well and help us to develop a framework that makes implementation of other refactorings easy.

### 2.1 Transforming expressions

Expressions are the basic building blocks of functional programs, and many of the refactorings move, restructure, or modify expressions. We found that to preserve the behaviour of an expression, the most important thing is to maintain the binding structure of its variables. We defined the binding structure using the concepts of variable scope and visibility. Another expression-related concept is whether an expression is side effect-free.

We have studied the *rename variable* and *extract function* refactorings that use only these concepts.

### 2.2 Tracking function references

The most frequently used expression is the function application, and refactorings that transform a function call, must transform the function definition accordingly. Unfortunately, finding the relation between function calls and function definitions is not always possible by static analysis. Remember that the identifier of a function is really a data tag. Most function calls include this tag as a constant, but it is possible to create the tag at run-time, and there are built-in functions that call a function with an argument list constructed at run-time.

We classified these constructs as directly supported (e.g. constant name and static argument list), partially supported (e.g. static name and dynamic argument list) and not supported (e.g. name read from standard input) calls, and plan to cover a broader range using data flow analysis (e.g. function name stored in a variable, or lambda expressions).

Refactorings that use only this kind of information are *rename function* and *reorder function arguments*, and *generalisation* needs the binding structure and function reference tracking as well.

### 2.3 Restructuring data types

Erlang has no static type information attached to variables, but types exist in the language, and they are strictly checked at run-time. Available compound types are lists, tuples, and records, these can be used to build more complex data structures. Sometimes the transformation of such a data structure is desired, but it is hard to describe what changes are to be made, and usually data flow analysis is required to find the expressions that manipulate the data.

Our most recent work is the analysis of such a transformation, when a record is introduced to store the elements of a tuple. This refactoring transforms the expressions that work with the same (or slightly modified) tuple, and these expressions can be found by a kind of data flow analysis. Tracking the way of a piece of data is easy when there are no side effects, the complicating factors are function references and constructs where more than one type of data is handled.

A simpler refactoring on data structures we dealt with is *tuple function arguments*.

## 3 Implementation

Our approach to refactoring is that we express the side conditions and code transformations by graph manipulation. We build a semantic graph starting from the abstract syntax tree of the source code and extending it with edges that represent semantic relations between nodes. Semantic concepts like variable scoping or function references are encapsulated into the graph this way.

A working prototype software is built using these concepts, written in Erlang. Building on previous experiences with Clean refactoring [2], we decided to represent the semantic graph in a relational database, and use SQL to describe the manipulations. Every node type has a table that contains the attributes of the nodes and the links to other nodes (represented by their unique ID). A nice feature of this representation is that fixed length graph traversals can be expressed by joining tables.

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 [4].

## References


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

[2] P. Diviánszky, R. Szabó-Nacsa, and Z. Horváth. Refactoring via database representation. In *The Sixth International Conference on Applied Informatics (ICAI 2004)*, volume 1, pages 129–135, Eger, Hungary, 2004.

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

[4] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy. Refactoring Erlang programs. In *The Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2006.