Building a Refactoring Tool for Erlang [⋆]

Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei,
Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, Roland Király
Eötvös Loránd University, Budapest, Hungary

Abstract

This paper presents RefactorErl, a refactoring tool for the Erlang programming language. Based on experience obtained during a major redesign of this tool, we describe some general principles that proved to be important for developing refactoring tools – not only for Erlang, but for other languages as well.

Key words: refactoring, Erlang, RefactorErl tool

1. Introduction

This paper presents a refactoring tool for the Erlang programming language and describes some general principles we found important for building that tool. We believe that these principles apply not only to Erlang, but to other languages as well.

Erlang [1] is an eager, impure, dynamically typed functional programming language developed by Ericsson. It was designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics, like telecommunication systems. The Erlang language consists of simple functional constructs extended with message passing to manage concurrency. Processes are fairly cheap in Erlang; the idiom to express a (thread-safe) stateful object is to use a process with a tail-recursive event loop and to achieve abstraction with a module. Erlang has a module system with export/import lists. It provides exception handling and reflective programming facilities, preprocessing mechanism to support macros and file inclusion, and a comprehensive standard library.

[⋆] Supported by Ericsson Hungary and ELTE IKKK.
Email addresses: `hz@inf.elte.hu` (Zoltán Horváth), `lovei@inf.elte.hu` (László Lövei), `kto@elte.hu` (Tamás Kozsik), `kitlei@elte.hu` (Róbert Kitlei), `viganiko@inf.elte.hu` (Anikó Nagyné Víg), `lestat@elte.hu` (Tamás Nagy), `melinda_toth@mad.hu` (Melinda Tóth), `kiralyroland@inf.elte.hu` (Roland Király).

Refactoring [6,7] means changing the program code without changing what the code does. Tool support for performing refactoring is available for many object-oriented programming languages, and for some functional ones as well [10,21]. The refactoring tool not only automates systematic transformations of programs, but also ensures that the semantics of the refactored programs are preserved. For this reason the refactoring tool will analyse the structure of the refactored program (based on the syntactic rules of the underlying programming language), and it will also collect and use semantical information about the program.

The rest of the paper is structured as follows. In Sect. 2 RefactorErl, our experimental research tool, a refactorer for Erlang is briefly described. Sect. 3 introduces our theses on how to build a refactoring tool for Erlang, and how these theses apply to building refactoring tools for other languages. Finally, Sect. 4 concludes the paper.

2. The RefactorErl tool

From technical point of view, there are two major versions of our Erlang refactorer. The first major version has been made available for public as a prototype tool (RefactorErl 0.2 at [5]), and the second one is being developed currently (not yet released). The first major version, thereafter referred to as "old tool", provides seven refactoring transformations:

– renaming a variable,
– renaming a function,
– extracting a new function definition,
– changing the order of function arguments,
– turning consecutive function arguments into tuples,
– substituting occurrences of a variable with its definition and
– discovering multiple occurrences of an expression and replacing them with a newly introduced variable.

Finally, there is a draft implementation of a transformation that turns tuples into records. The tool itself is written in Erlang, and it is available on several platforms. Its user interface is integrated into the Emacs editor, which is very popular among programmers who use Erlang in the software industry.

Experiments with the old tool led us to redesign and reimplement essentially the whole tool. The second major version of RefactorErl, thereafter referred to as "new tool", improves efficiency and usability significantly. However, many of the transformations available in the old tool are not yet implemented in the new tool, and for this reason the new tool is not released yet. Besides the eight transformations mentioned above, we are currently working on the following refactorings:

– renaming a module, a record or a record field,
– moving a function, a macro or a record definition into another module,
– inlining a function definition,
– turning a tuple argument of a function into consecutive arguments and
– generalizing a function definition by introducing further arguments.

The transformation that can turn tuples into records is the theoretically most challenging one – and there is a strong industrial demand for it [14]. In the future we are also going to research refactorings that interact with inter-process communication. Currently, however, most of our efforts goes into the development of a stable and robust refactoring infrastructure, which can be easily extended with further transformations. Our aim is that once the new tool is released, it should be possible to use it in an industrial environment.

RefactorErl represents an Erlang program as a graph containing all relevant lexical, syntactical and semantical information. At first glance, this graph can be regarded as an enriched abstract syntax tree (see Sect. 2.2). To make refactoring of large programs more efficient, the construction of graphs representing programs is incremental: the graphs are persisted in a database, so a (sub)graph representing a module needs to be recomputed only when the module is altered manually (i.e. not with the refactoring tool). This approach is especially useful when a large number of refactoring transformations are to be applied on a program without intervening editing of the code by programmers. This happens, for instance, before introducing a new feature into an (otherwise fully functional) program, which may require a substantial reorganization of the code.

2.1. *Limitations in refactoring Erlang programs*

Some features of the Erlang language are advantageous for refactoring: side effects are restricted to message passing and built-in functions, variables are assigned a value only once in their lifetime, and code is organised into modules with explicit interface definitions and static export and import lists. There are, however, some features that are disadvantageous for refactoring, e.g. the possibility to run dynamically constructed code, and the lack of programmer defined types. For a more detailed analysis see [9].

Refactorings, by definition, should preserve semantics. Unfortunately, it is practically impossible to guarantee this in the case of a language supporting reflection. Worse still, industrial Erlang code makes use of such facilities very frequently. On the one hand, if we design a conservative refactoring tool that always refuses to perform transformations which might alter the meaning of the refactored program, we might end up with a tool that, albeit perfectly safe, is completely useless in practice. On the other hand, a tool offering insufficient support for the preservation of semantics will never be used in practice: nobody will ever dare to refactor large programs with it. A good refactoring tool will be sufficiently safe, but not too restrictive. To achieve this, the decision mechanism in the tool should be customizable and/or interactive.

Since in general it is not possible to completely determine the meaning of an Erlang program by static analysis, a refactoring tool might decide to compensate for otherwise unsafe transformations. Inserting dynamic checks into the refactored programs often helps to bring a refactoring into effect depending on run-time information. Consider the (admittedly extreme) example in Fig. 1. In Erlang it is possible to construct a function call by computing the name of the function to be called and the actual arguments, and pass them to the built-in function `apply`. If `factor/1` is a prime-factorization function returning the list of prime factors of a given number, then the code fragment on the left will apply `a:egg/2` (that is the binary `egg` function from module `a`) on the actual arguments `2` and `3`. The `list_to_atom` function is a built-in function that takes a string – represented as a list of ISO Latin-1 codes in Erlang – and creates an atom from it. Function and module names are atoms in Erlang. A refactoring tool has no chance to find out by static analysis that `a:egg/2` is executed here. The only way to preserve program behaviour when refactoring `a:egg/2`, for example when swapping its arguments is to insert dynamic checks. The previous call to `apply/3` could be replaced with the

```
apply( list_to_atom(factor(97))
     , list_to_atom(factor(1071509))
     , factor(6)
     )
```

$\longrightarrow$

```
begin
  M=list_to_atom(factor(97)),
  F=list_to_atom(factor(1071509)),
  A=factor(6),
  case {M,F,A} of
     {a,egg,[A1,A2]} -> a:egg(A2,A1);
     _               -> apply(M,F,A)
  end
end
```

Fig. 1. Compensation for swapping the arguments of the function `a:egg/2`.

expression shown on the right-hand side of Fig. 1, assuming that `M`, `F`, `A`, `A1` and `A2` are fresh variables.

Although this was an extreme example, it is important to note that production Erlang code regularly utilizes the built-in function `apply/3` and the alike. Compensation techniques, such as the one above, are often applicable, but they also degrade the readability and the efficiency of the resulting code, and hence must be used with care.

2.2. *The model used by the new tool*

RefactorErl represents an Erlang program as a "program graph": a directed, rooted graph with typed nodes and edges. The skeleton of this graph is the abstract syntax tree of the program. Apart from syntactical information, the graph contains lexical and semantical information as well. These latter kinds of information are provided as additional nodes and edges in the graph. For example, each function in the program is represented as a semantic node in the graph; the definition of the function and all the calls to the function are linked to this semantic node with semantic edges. The maintenance of semantic information is useful for boosting side condition checking. Usually, the hardest part of refactoring is not the application of the requested transformation, but the evaluation of the conditions that are required to hold for the refactoring to be safe. These conditions often depend on a large amount of semantical information – which can be efficiently picked out from the program graph. Apparently, the stored semantical information, similarly to the AST, must be updated when a transformation is applied.

Lexical information, such as the tokens produced by the scanner, is also essential. Even information about the whitespace separating the tokens must be kept available so that the refactoring tool can preserve the layout of the refactored program.

The kinds of semantic information to be gathered and maintained by the refactoring tool depend on the transformations the tool supports. The new RefactorErl tool is designed to be open-ended: it should be possible to implement a new refactoring with the relevant semantical analysis and add them to the refactoring framework. To achieve this goal, the different kinds of semantical analysis are organized into independent modules, and result in independent sets of semantic nodes and edges in the program graph. Examples of semantic analysis modules are analysing scopes, analysing function definitions and calls, or analysing variable bindings.

The new tool also includes a query language, similar to XPath [27], for retrieving information from the program graph. Links of the graph can be traversed forwards and backwards, and filtering by semantical information is also supported.

To optimize the shape of the program graph for fast information retrieval, the syntax of the language is reflected in the tool at two levels of abstraction. In the more abstract view there are four syntactical categories: files, forms, clauses and expressions. Files (including header files) contain forms. Forms can be, among others, function definitions, which are made up of one or more clauses (clauses are basic building blocks of several compound expressions as well, such as `case`-expressions). The right-hand side of a clause is a sequence of expressions (and the left-hand side of a clause contains further expressions such as patterns and guards). The rich syntactical structure of Erlang (reflected in the close to fifty rules of the grammar) can be abstracted into these four kinds of graph nodes. Many details of the syntax are encoded in the types of the graph edges, forming the less abstract syntactic view of the language.

The low number of types of syntax nodes improves efficiency of the queries written in the query language. Another important source of efficiency is that chains of applications of the same production rule are not represented by an unbalanced tree, but rather by a single graph node, which collects all of the syntax edges of the productions, retaining the order of the edges.

In order to improve the reusability of the refactoring infrastructure, one could design the model of the refactored programs as general and language-independent as possible. Our experience in refactoring Clean [21] and Erlang did not foster this approach. Even in these two functional languages the syntactic and semantic differences are so significant that it is not worth to introduce a common model, not even with language-specific extensions (like in [22]). Our approach is to keep the focus of the tool on a single language, and achieve a level of precision and efficiency that is sufficient to make the tool applicable in practice.

3. Building a refactoring tool for Erlang

This section explains why we decided to redesign and reimplement RefactorErl after experimenting with the old tool, and how this decision is justified by experiences with the new tool. Eight theses are presented here, which, according to our position, describe important rules to keep in mind [20,19] when building a refactoring tool – either for Erlang or for other languages.

3.1. *Language specific model*

Besides a general, language-independent refactoring software infrastructure [3,4,22,24] a language specific model supporting refactoring concepts is required. Based on that model a model-driven architecture should be developed. In order to guarantee consistency and to avoid ad-hoc and conflicting solutions, all the components of the refactoring tool (i.e. the parser, the semantic analyser, the code generator, the construction utilities for insertion and replacement of code parts, and the source code formatter) should either be generated from, or controlled by, the same model. This is a declarative approach which maintains the refactoring-specific lexical, syntactical and static semantical rules of the investigated language as data. Modifying these data should result in the (as far as possible) automatic adaptation of the code of all the components of the refactoring tool.

For example, our new tool represents the model as an XML-document containing information about the lexical and syntactical rules of Erlang together with instructions for creating the internal graph representation of programs. This format was chosen because it is easy to maintain, should the language definition change. Furthermore, it can be handled easily with XMErl [29], a standard Erlang tool for traversing XML documents, and it can be easily transformed with e.g. XSLT tools.

Two refactorings are already implemented in the new tool. One of them is extracting functions, which was supported by the old tool as well. The new implementation is about half the size of the old one, and it is much more comprehensible. Most of this achievement comes from the usage of the graph query language, which can follow semantic links the same way as AST links, and from the fact that much of the work is performed by code auto-generated from the model.

3.2. *Full support for preserving layout*

The refactoring-specific model of the investigated programming language enables the maintenance of lexical information in such a way that the layout of refactored programs can be preserved during the transformations – except maybe for code fragments directly affected by the transformations. In our experience, the preservation of layout is essential for the tool to gain industrial acceptance.

The old tool used standard Erlang scanners, pre-processors and parsers (basically those used also in the compiler). These tools removed all lexical information that was irrelevant for syntactical and static semantical checking and compilation of programs, including indentation, redundant parentheses in expressions, and comments. Furthermore, the refactoring tool could not access the original (before pre-processing) source code, hence refactoring programs containing macros [8] (that is every interesting, commercial code) was practically impossible.

The problem with macros is that they can cross-cut the structure of the parse tree: code containing macro applications do not fit into an AST. One way to tackle this problem is to restrict the use of macros in such a way that they can be treated as syntactic entities. This excludes some complex cases, but it makes macros part of the syntax. This is the approach taken in Xrefactory [25]. A more advanced approach does not impose restrictions on the use of macros. Proteus [28] is based on an extended, layout preserving AST which uses recorded macro expansion. This tool, applicable to refactor C++ programs is similar to our approach in many respect.

Our new tool introduces its own lexical and syntactical analyzers (generated automatically from the model) that preserves layout information, i.e.

(i) whitespace is preserved in the lexical layer;
(ii) redundant parentheses are not removed when building the AST;
(iii) the preprocessor stores information about include files, macro definitions and macro applications.

The lexical and syntactical analyzers of the new tool maintain information on both the original and the pre-processed source code (see Fig.2).

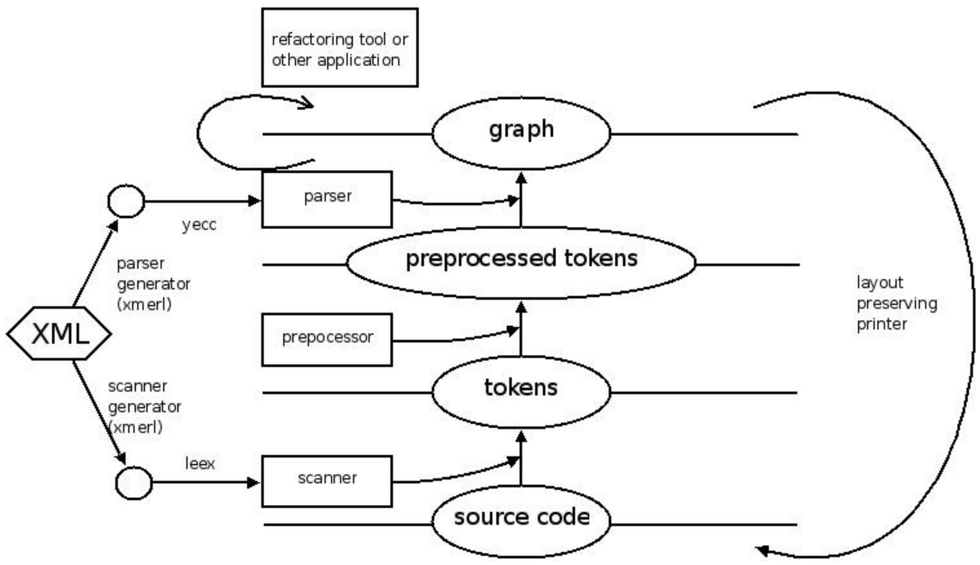

Fig. 2. Scanning, preprocessing and parsing in RefactorErl

3.3. *Create your own parser*

As a consequence of Sect. 3.1 and 3.2, it is worth to create a parser and a storage back-end for the refactorer instead of relying on the compiler interface. The generation of a parser and a back-end from an appropriate model is usually straightforward, and due to the model-driven approach, the parser remains consistent with the rest of the refactoring tool.

Although Erlang is a language defined by its compiler, the above argument about being independent from the compiler is still valid. If the language (and the compiler) changes, it is not more expensive to update the model than to adjust the rest of the refactoring tool to the changed compiler interface.

As revealed in Sect. 2.2, a further advantage of a refactoring-specific parser is that the model mentioned in Sect. 3.1 enables the representation of the abstract syntax tree in a way that is optimized for refactoring – this is not possible with the standard Erlang parser.

3.4. *Step-by-step syntactical and semantical coverage*

It is not feasible to figure out all aspects of the syntax of the investigated language during the design (and before the implementation) of the refactoring tool. First, the language might change, and so might its syntax. Furthermore, based on experience on using the tool and user feedback, novel, refactoring-specific syntactical categories may be necessary to introduce. For example, in Erlang there are no static type declarations, but programmers often provide type information in comments (in the future these typing comments may become part of the language proper [17]). Since programmer provided type information can facilitate data-centric transformations, it might become necessary to extend the model with the processing of such comments long after the general infrastructure of the tool has been built.

The same reasoning is valid for semantical coverage – it is not possible to know in advance which kinds of semantic information is useful for future transformations. Moreover, there is a theoretical obstacle as well: it is not feasible to cover every semantical property by static analysis [19], especially in a language that emphasises dynamic semantic rules, like Erlang.

In practice it is not very useful to try to maximize syntactic and semantic coverage. It is more useful to focus on your users, on the properties of the code that your users want to refactor. The capabilities of the refactoring tool should be improved to support commercial code as much as possible, taking coding conventions and company policies into account.

There might be features in a language that are rarely (if ever) used in production code, like packages in Erlang. There might be features that make static refactoring transformations practically impossible.

To support step-by-step semantical coverage, the new RefactorErl tool introduces a framework for semantical analyses, which can be extended with novel semantic analysis modules.

3.5. *Accept rebuilding semantic information*

Experience with the development of the new tool showed us that semantic information might be better maintained differently than syntactic and lexical information: it is slower to figure out how to modify semantic nodes and edges during a transformation (during the modification of the AST and related lexical information) than to *recompute* the semantic nodes and edges of the modified subgraph based on the resulting AST. Redirecting semantic edges in a graph representation of the refactored program requires complex graph rewrite rules, which usually requires hand-crafted code (they are hardly possible to automatically generate from the model). We found that recomputing the semantic information in the subgraph affected by a transformation improves the safety of the transformation as well.

3.6. *Improve queries with shortcuts*

During the checking of side conditions and the computation of transformation results the refactoring tool often collects information that is scattered in an AST or in a semantical graph. The naïve approach, also used by the old RefactorErl tool, to collecting such information is to traverse the graph. This proved to be both tedious to implement and slow in practice.

Therefore in the new tool we decided to reduce the depth of queries and traversals by introducing frequently needed shortcuts (redundant edges) in the graph. These provide connections between nodes that are distant in the syntax tree but close in meaning. By adding or removing analyser modules, the number of the semantic edges can be controlled. This is a trade-off between the number of semantic edges in the graph and the complexity of refactorings. This enriched semantic graph provides enough information for the refactorings to verify the side conditions and reach the nodes to be transformed in an efficient way. Moreover, semantical analyses are possible to perform in the background (while the programmer browses the source code) or even lazily – these techniques conserve the responsiveness of the refactoring tool even in the presence of many semantical edges.

3.7. *Ensuring safety with cross-testing*

RefactorErl is developed in collaboration with University of Kent, where another refactoring tool for Erlang, the Wrangler [12] is being developed. Wrangler has a design and an internal graph representation different from that of RefactorErl, but building an interface for data interchange between the two systems is possible [23]. The transformations implemented in the two tools are not the same either. We plan to integrate the two systems in the near future.

There is, however, another possibility for collaboration. Since the objectives of both tools are the same, cross-testing seems to be the adequate way to increase trust in the tools, like in [2].

3.8. *Reuse of analyses is possible*

The syntactic and semantical analyses that a refactoring tool applies for checking side conditions turn out to be useful for solving other problems as well. They can be used to understand the structure of large and complex software systems, and to partition that structure into subsystems. Currently we are experimenting with different clustering and remodularization algorithms for such purposes using information about (few MLOC) software collected with the analysis modules of RefactorErl.

4. Conclusions

This paper presented some principles on how to build a refactoring tool and supported these principles by describing experiments with a refactoring tool for the Erlang language. First of all, all the components of the refactoring tool are based upon a refactoring specific model of the investigated language. This model can support the preservation of layout. The parser and the storage back-end of the tool should be generated from the model. It is reasonable to refine this model step-by-step to reach the desired level of syntactic and semantical coverage. Instead of writing ad-hoc hand-crafted code fragments for redirecting semantic edges with respect to different transformations, it is better to recompute some parts of the semantic graph representing the refactored program. Efficiency of queries for the evaluation of side conditions can be improved by adding redundant semantic edges to the graph. Cross testing between different refactoring tools for the same programming language provides invaluable help for increasing trust in the tools. Finally, analyses developed for the refactorer often turn out to be reusable in other tools. Applying these principles for redesigning RefactorErl led to increased performance (acceptable response time for huge bodies of commercial code) and also to more maintainable software.

References

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

[2] Brett D., et al., *Automated Testing of Refactoring Engines*, In Proc. of the the 6th joint meeting of the European software engineering conference, pages 185-194, Dubrovnik, Croatia, 2007.

[3] Charles, P., Fuhrer, R.M., and Sutton, Jr., S., M., *IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse*, In Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pages 485-488, Atlanta, Georgia, USA, 2007.

[4] Ducasse, S., Gîrba, T., and Nierstrasz, O., *Moose: an Agile Reengineering Environment* In Proceedings of ESEC/FSE 2005, September 2005, pages 99-102.

[5] Eötvös Loránd University, *Refactoring Erlang Programs* (project homepage). `http://plc.inf.elte.hu/erlang/`

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

[7] Fowler, M., *Refactoring Home Page*. `http://www.refactoring.com/`.

[8] Garrido, A., *Program Refactoring in the Presence of Preprocessor Directives.* Tesis Doctoral. Univ. of Illinois at Urbana-Champaign Technical Report No. UIUCDCS-R-2005-2617.

[9] Kozsik, T., Csörnyei, Z., Horváth, Z., Király, R., Kitlei, R., Lövei, L., Nagy, T., Tóth, M., Víg, A., *Use Cases for Refactoring Erlang Programs.* To appear in Central European Functional Programming School, Revised Selected Lectures, Springer LNCS series.

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

[11] Li, H., Thompson, S.J., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., and T. Nagy, T., *Refactoring Erlang Programs.* In Proceedings of the 12th International Erlang/OTP User Conference, November 2006.

[12] Li, H., Thompson, S.J., *Testing Erlang Refactorings with QuickCheck.* In Proc. of the 19th International Symposium on Implementation and Application of Functional Languages, IFL2007, Freiburg, Germany, Septemper 2007.

[13] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A., and Nagy, T., *Refactoring in Erlang, a Dynamic Functional Language.*, In Proceedings of the 1st Workshop on Refactoring Tools, pages 45-46, Berlin, Germany, July 2007.

[14] Lövei, L., Horváth, Z., Kozsik, T., Király, R., *Introducing records by refactoring.* In Proceedings of the 2007 ACM SIGPLAN Erlang Workshop, pages 18-28. ACM Press, 2007.

[15] Lövei, L., Horváth, Z., Kozsik, T., Király, R., and Kitlei, R., *Static rules of variable scoping in Erlang*, In Proceedings of the 7th International Conference on Applied Informatics, volume 2, pages 137-145. 2008.

[16] Mitchell, B.,S., *A heuristic search approach to solving the software clustering problem*, PhD thesis, Drexel University, Philadelphia, PA, USA, 2002.

[17] Nyström, S., *A soft-typing system for Erlang*, Proceedings of the 2003 ACM SIGPLAN workshop on Erlang, pp. 56-71, Uppsala, Sweden, 2003.

[18] Robbes, R., Lanza, M., *The "Extract Refactoring" Refactoring*, In Proceedings of WRT 2007 (1st International Workshop on Refactoring Tools), pp. 29 - 30, Berlin, Germany, 2007.

[19] Roberts, D.: *Practical Analysis for Refactoring*, PhD thesis, University of Illinois at Urbana Champaign, 1999.

[20] Roberts, D., Brant, J., and Johnson, R., *A Refactoring Tool for Smalltalk*, Theory and Practice of Object Systems. V3 N4, October 1997.

[21] Szabó-Nacsa, R., Diviánszky, P., and Horváth, Z., *Prototype environment for refactoring Clean programs.*, In The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, July 1–4, 2004. Full paper is available at `http://aszt.inf.elte.hu/~fun_ver/` (10 pages).

[22] Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O., *A Meta-model for Language-Independent Refactoring*, Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00), IEEE Computer Society Press, 2000, pp. 157-167, Kanazawa, Japan, November 2000.

[23] Vinju, J.J.: *Uptr: a simple parse tree representation format*, In Software Transformation Systems Workshop, October 2006.

[24] Vinju, J.J, *Analysis and Transformation of Source Code by Parsing and Rewriting*, PhD thesis, November 2005.

[25] Vittek, M., *Refactoring Browser with Preprocessor*. In Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, page 101, 2003.

[26] Wloka, J., Hirschfeld, R., and Hänsel, J., *Tool-supported Refactoring of Aspect-oriented Programs*, In Proceedings of the Conference on Aspect-oriented Software Development (AOSD), pages 132-143, Brussels, Belgium, March 31 - April 4, 2008.

[27] World Wide Web Consortium: *XML Path Language (XPath) Version 1.0.* W3C Recommendation, Nov. 16, 1999, `http://www.w3.org/TR/xpath.html`

[28] Waddington, D. G., Yao, B. *High Fidelity C++ Code Transformation.* In Boyland, J. and Hedin, G., editors, Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005), 2005.

[29] Wiger, U., *XMErl – Interfacing XML and Erlang*, In the Sixth International Erlang/OTP User Conference (EUC 2000), Stockholm, Sweden, October 3, 2000.
`http://www.erlang.se/euc/00/xmerl.ppt`