Automated syntax manipulation in RefactorErl*

Róbert Kitlei László Lövei Melinda Tóth
Zoltán Horváth Tamás Kozsik Roland Király
István Bozó Csaba Hoch Dániel Horpácsi

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
e-mail: {hz, kto, kitlei, lovei,kiralyroland,bozo_i,hoch,toth_m,daniel_h}@inf.elte.hu

Abstract

Refactorings often have to change the source code by adding, changing or replacing parts of the syntax tree. It is important to make these changes convenient and secure for the developer of refactorings.

In this paper, we introduce a method that helps us create, replace and insert syntactically correct subtrees in an Erlang refactoring tool.

1 Introduction

Refactoring is the systematic changing of source code while retaining the semantics of the code. Some refactorings, e.g. renaming a variable or a function, do not change the shape of the syntax tree, only update the information in the nodes, while most of the other refactorings construct, delete, move, insert or replace subtrees in the syntax tree. Deletion does not pose a problem, and moving a subtree is equivalent to its removal and reinsertion, therefore the most intriguing questions of the above are the construction, insertion and replacement of subtrees.

In addition to the above, refactorings have to gather additional information about semantic aspects of the source code as well. Since these bits of information can only be collected by visiting diverse parts of a syntax tree, another representation design may prove more efficient. A novel graph representation is proposed by the Erlang refactoring group at Eötvös Loránd University (ELTE, Budapest, Hungary). The ELTE group proposed this representation after previous experience with refactoring [5, 8]. Details about the representation and the refactoring tool RefactorErl are found in [4].

The structure of this paper is as follows. Section 2 and 3 describe methods that facilitate the creation, insertion and replacement of subtrees.

*Supported by ELTE IKKK and Ericsson Hungary.

Section 4 describes our experience with the above methods through a case study.

2 Subtree construction

One possible solution for constructing AST subtrees would be to use the parser itself by providing the source code that corresponds to that of the desired subtree. This approach would require the user to manually fill in all the punctuation, and would require separate grammars for each non-terminal to be generated. Another possibility is to do all the construction by hand, which is tedious and error-prone. Here we present another alternative.

The user has to supply two pieces of information for the node creation algorithm. One is the contents of the newly created node, which also contains its node type. By supplying the type of the node, the relevant rule structure can be selected from the grammar. The other piece of information required is a description of the desired contents of the node.

Since keywords and separator tokens can be automatically generated, and the skeleton determines the structure of the nodes, these tokens are not included in the description. All other tokens (e.g. variable names or function names) and all symbols have to be listed in order.

The algorithm processes the rule description and the content description. If the rule prescribes an automatically created token, it is created; otherwise, one or more elements supplied by the user are consumed when creating the next symbol or construct from the rule description.

Subtrees can be created by repeated use of the above algorithm. When constructing a new node, previously created nodes can be used as well as nodes that were already present in the graph.

3 Subtree replacement or insertion

Node replacement is done in a similar way to that of node construction. As parameters, the new nodes to be inserted and the place of insertion or replacement has to be specified. One of the following can be specified.

- Replacement of all or part of nodes of the same type.
- Replacement of a range of nodes.
- Insertion before or after a node.

The insertion-replacement algorithm scans the grammar description, the given description and the actual structure. It determines the affected part in the syntax tree, makes the change and controls whether the resulting structure conforms to the grammar description.

All of the algorithms described above use an automatically generated scanner to check whether the tokens given in the descriptions are valid.

4 Case study: extract function

Refactorings usually consist of three different parts: collection of necessary information, checking preconditions and performing the transformation. The first and second part are used to reject the refactoring if its preconditions are not met, and the requested actions would change the behaviour of the code. The first and second part are necessary to guarantee the semantic consistency and preserve the externally observable behaviour. When performing the transformation, new code parts are created from the previously collected data. The new code parts strongly depend on the representation of the source, and have to be syntactically valid. Most of the time, the parts to be created and inserted cannot be composed only of parts that were present in the original code.

The "extract function" transformation extracts an expression or a sequence of expressions to a new function. Its parameters are the name of the containing file, the selection that will be extracted and the name of the new function. The transformation creates a new function definition, and replaces the selection with a function application. The variables which are used inside but bound outside the selection become the formal parameters of the new function. (See in Figure: 1).

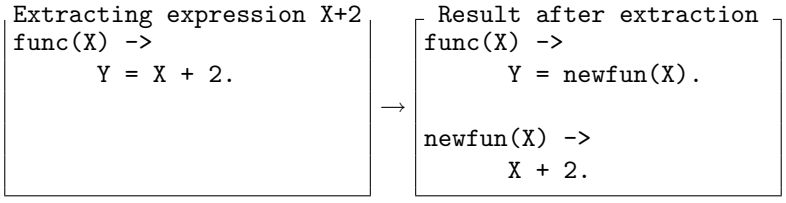


Figure 1: Extracting expression `X+2` into function `newfun`.

The created function definition (See in Figure: 3) contains opening and closing parentheses, an arrow and a stop token as punctuation, and the function name, the parameters of the function and the expressions of the body of the function. Also, if multiple parameters or body expressions are present, additional separating comma tokens are present. The function application contains a function name, opening and closing parentheses and the parameters with separating commas. (See in Figure: 4)

Most of the above tokens can be automatically generated. Creating the function definition requires only the function name, parameter names and the body expressions; creating the function application requires the function name and the actual parameters. The syntactical and lexical representation of the body of the extracted function are available from the selection, but the other parts have to be constructed.

Previously, this refactoring constructed all parts using manually crafted code. Such code proved to be long, hard to maintain and error prone. In contrast, using the utilities in section 2, the corresponding code consists of only a few lines. Furthermore, the code is much more readable, as it highlights those parts that cannot be automatically generated.

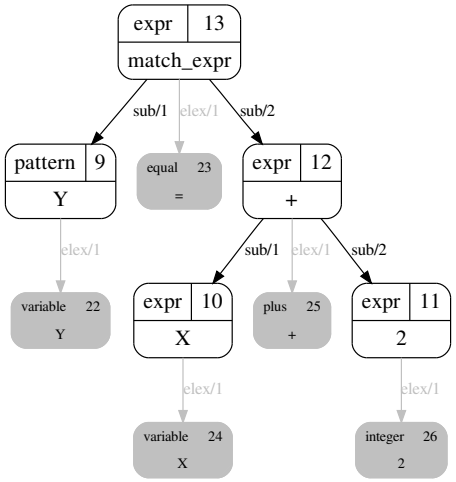


Figure 2: The representation of `X+2`, which is about to be extracted.

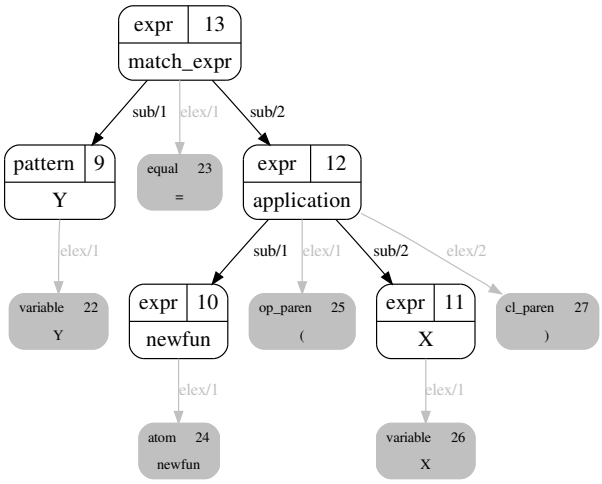


Figure 3: The representation of the function application that replaces the selection.

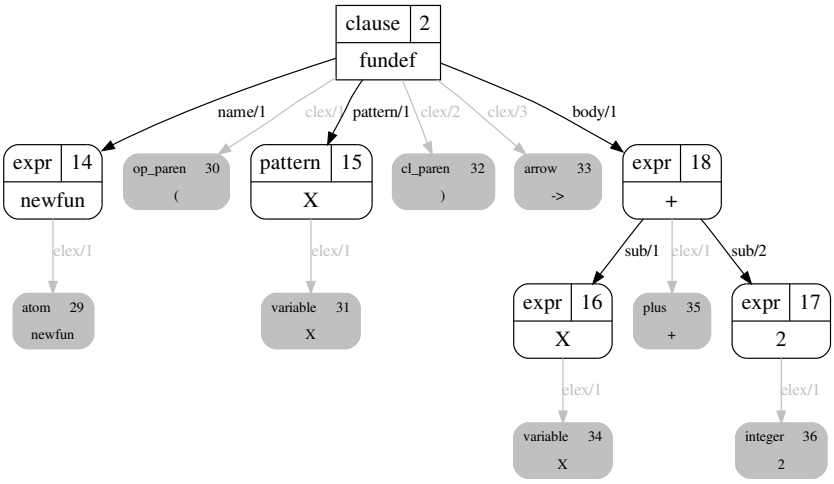


Figure 4: The representation of the newly created function definition.

5 Conclusion

In this paper, we have presented methods for creating, replacing and inserting subtrees in the RefactorErl refactorer. Our experience shows that it greatly enhances the readability and reliability of the invocation of these operations, in contrast to their hand coded equivalents. Using these methods, we were able to implement rudimentary functionality of twelve new refactorings in a time frame of two months, compared to only two before. The two already implemented refactorings were *Extract function* and *Move function definition*, which were improved using the methods described in this paper. The new refactorings are the following: *Rename variable*, *Rename function*, *Rename module*, *Rename record*, *Rename record field*, *Reorder function arguments*, *Tuple function arguments*, *Eliminate variable*, *Merge subexpression duplicates*, *Move record*, *Inline function*, *Generalize function*.

References

[1] G. Fischer, J. Lusiardi, and J. Wolff v. Gudenberg, *Abstract syntax trees and their role in model driven software development.* In ICSEA online proceedings. IEEE, 2007.

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

[3] Greg J. Badros, *Javaml: a markup language for java source code.* In Proceedings of the 9th international World Wide Web conference

on Computer networks: the international journal of computer and telecommunications networking, pages 159177. North-Holland Publishing Co. Amsterdam, The Netherlands, The Netherlands, 2000.

[4] Róbert Kitlei, László Lövei, Tamás Nagy, Zoltán Horváth, Tamás Kozsik, *Preprocessor and whitespace-aware toolset for Erlang source code manipulation.* Abstract submitted to the 20th International Symposium on the Implementation and Application of Functional Languages, Hatfield UK.

[5] R. Szabó-Nacsa, P. Divinszky, 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.

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

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

[8] 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, Berlin, Germany, July 2007, pp. 45-46.

[9] Jonathan I. Maletic, Michael L. Collard, and Adrian Marcus, *Source code files as structured documents.* In Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), pp. 289–292. IEEE Computer Society Washington, DC, USA, 2002.