LAYOUT PRESERVING PARSER FOR REFACTORING IN ERLANG

Róbert KITLEI, László LÖVEI, Tamás NAGY, Zoltán HORVÁTH, Tamás KOZSIK
Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary
{kitlei,lovei,n_tamas,hz,kto}@inf.elte.hu

ABSTRACT

This paper describes preprocessor and whitespace-aware tools for parsing and transforming Erlang source code. The presented tools are part of RefactorErl, a refactoring tool for Erlang programs. RefactorErl represents programs as a "semantic graph" that extends the AST with semantic nodes and edges for efficient information retrieval. The paper focuses on describing the construction of the AST and syntax based transformations of the semantic graph.

Keywords: *refactoring, layout preservation*

1. INTRODUCTION

Refactoring is a computer-aided technology for improving code design by programmer-guided, semantics-preserving source code transformations [1, 2]. Tool support for refactoring is available for many imperative programming languages, and also for some functional ones. We are developing a refactoring tool, RefactorErl [3], for the Erlang language [4–6].

Developing a refactorer for Erlang programs is a challenge as some properties of this language make refactoring cumbersome, and sometimes even either unsound or unproductive [7]. One such property is that certain language concepts (such as scopes and visibility) are very hard to capture properly in a static semantics description [8]. Another property, an issue addressed in this paper, is the presence of a preprocessing facility (e.g. file inclusion and macros), which makes source code manipulation rather difficult. Further problematic properties include the support for reflective function calls, inter-process communication and dynamic typing.

Erlang has a standard scanner and parser, used by the de facto standard compiler [9]. However, refactoring Erlang programs requires more advanced tools for at least three reasons. Firstly, the standard tools can handle preprocessor constructs by modifying the token stream that the scanner produces, but in this process they discard the original tokens and consequently the connections between the original and preprocessed tokens as well. Secondly, they lose whitespace information. Finally, they ignore the concrete syntax and produce an abstract syntax tree directly. For example, the expression `(A+B)-C` will be represented in the same way as the expression `A+B-C`. Removing the parentheses does not change the meaning of the program, but it may upset the programmer who inserted the seemingly superfluous punctuation for his own purposes. All of these factors make it impossible to retain lexical information necessary to do refactoring in such a way that pleases the user of the refactoring tool – who expects that refactoring transformations preserve the formatting of the source code as much as possible. Experiments with earlier (v0.1 to v0.2, 2007) releases of RefactorErl – which were based on the standard Erlang scanner/parser – proved that preserving source code layout is a major issue for the practical acceptance of the tool.

This paper describes our preprocessor and whitespace-aware tools for parsing and transforming Erlang source code. These tools are applied in the first three of the static analysis phases performed by RefactorErl v0.6 (lexical analysis, preprocessing, syntactic analysis and semantic analysis), and also in the implementation of refactoring transformations. The tools presented in the paper address the following goals.

- Represent program text in such a way that information about the source code both before and after preprocessing is available.
- Preserve code layout (whitespace, redundant parentheses, comments, number formatting) during scanning and parsing.
- Support the programmatic construction of syntax subtrees.
- Facilitate the insertion of subtrees into, and deletion from, the syntax tree.
- Preserve code layout during the manipulation of the syntax tree.

The rest of the paper is structured as follows. Section 2 gives an overview of the representation of source code used in the tool. In particular, section 2.3 describes how preprocessor constructs, which can cross-cut the syntax, can be handled. Section 3 describes how the original AST can be recovered, and also how syntax based changes can be applied to the tree. Section 4 describes related work and section 5 draws the conclusions and points out future directions of research.

2. OVERVIEW OF REFACTORERL

RefactorErl is based on an AST that is constructed using tools partly generated from an XML grammar description. This section describes the grammar description, the syntax tree constructed using the generated tools and the preprocessing that has to handle the source code before parsing can begin.

2.1. Semantic graph

RefactorErl represents Erlang source code as a *semantic graph*. The skeleton of the semantic graph is a customized abstract syntax tree built using our scanner, preprocessor and parser.

A semantic graph consists of

- *nodes* representing various linguistic entities of an Erlang system,
- directed, labeled *edges* between nodes representing relations between entities and
- a *total ordering* on the outgoing edges of nodes; this is further elaborated in section 3.2.

A subset of these nodes and edges form the customized AST, other nodes and edges carry lexical and semantic information. The total order describes the order of subtrees of a node in the syntax tree.

Nodes are classified based on the kind of entity they represent: *variable*, *expression*, *function* and *file* are examples of classes. Nodes also have a set of attributes based on their class – for example, variables have a *name* attribute, and files have a *path* attribute. The graph has a unique *root node*, the single occurrence of the *root* class, from which all other nodes are accessible.

Edges have an associated *tag* (label) based on the kind of relation they represent. The set of tags used in the graph is fixed, and they are used consistently with respect to node classes. For example, each edge tagged *funref* connects an *expression* node to a *function* node – the meaning of the edge is that the expression explicitly refers to the function.

By the time the scanner, preprocessor and parser did their jobs, the customized AST and, additionally, some semantic nodes and edges are already available in the semantic graph. The tokens of the AST also preserve the whitespace and comments around them. Currently all whitetext is joined to the token after it; this will change to a more sophisticated handling scheme in the future. After the semantic graph has been prepared, it is further enriched with semantical information (e.g. the binding structure of variables, scope and visibility information, function call graph) by different, independent analyzer modules. These modules extend the semantic graph with nodes and edges that represent language concepts not covered by the AST, but which are considered relevant for computing side conditions that may invalidate refactorings or require additional compensations, and the effect of refactoring transformations. The modular structure of the semantic analysis helps the representation of linguistic information to evolve as novel refactoring transformations are introduced.

Fig. 1 shows the AST built atop `Y=G(F(F(1)).` This code fragment is revisited in section 3.3.

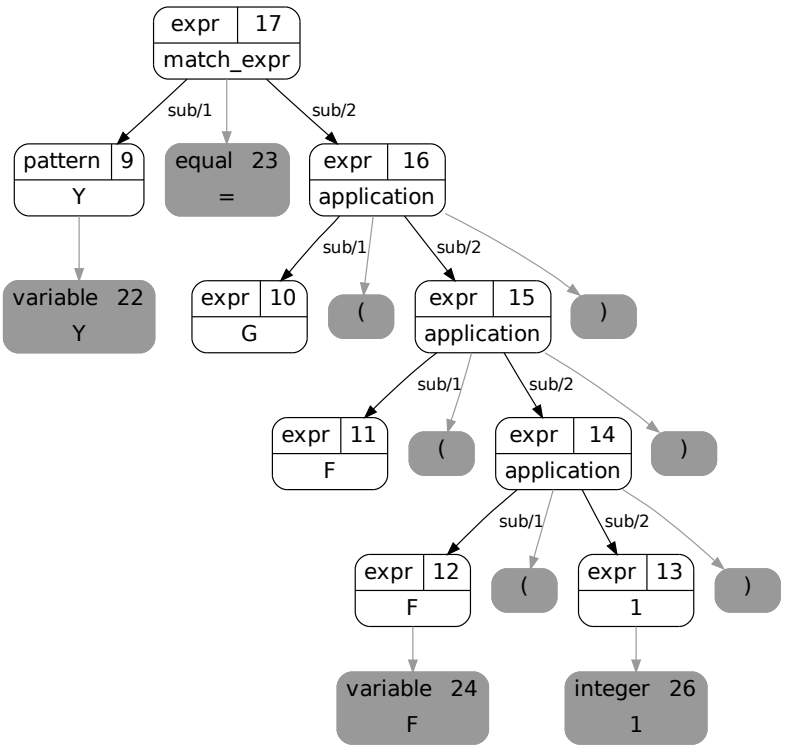


Fig. 1 The representation of a match expression in Erlang.

```
<ruleset head="Pattern_clause">
  <rule>
    <attrib name="kind" value="pattern"/>
    <symbol name="Expr" link="pattern"/>
    <optional>
      <token type="when"/>
      <symbol name="Guard" link="guard"/>
    </optional>
    <token type="arrow"/>
    <repeat symbol="Expr" link="body" separator="comma"/>
  </rule>
</ruleset>
```

Fig. 2 XML grammar example

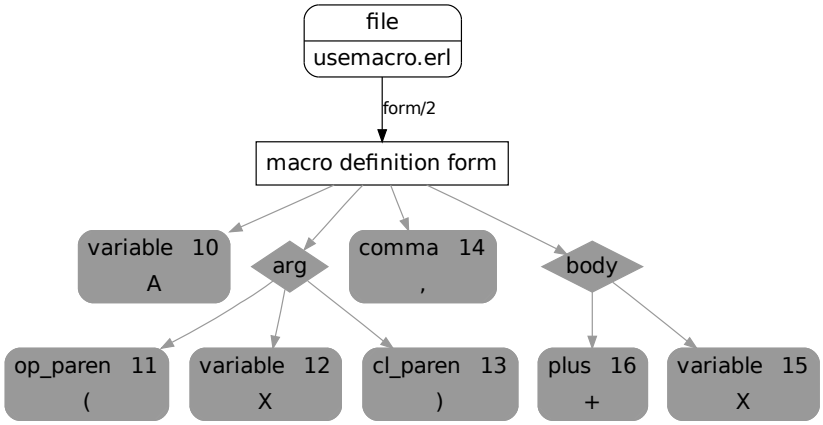


Fig. 3 Part of the graph representation of macro definition `-define(A(X),X+).`

2.2. Grammar description

The lexical and syntactical rules of Erlang are described in an XML document, which also contains annotations that are used for creating the internal graph representation of programs.

A large part of our tools is automatically generated from this high-level grammar description. The XML format was chosen because it is easy to maintain, should the language definition change. It occasionally does: specifications [10] were introduced recently. Furthermore, XML can be handled easily with XMErl [11], a standard Erlang tool for traversing XML documents, and it can be easily transformed with existing tools, e.g. XSLT. The grammar rules are described in the XML document as regular expressions and a slightly modified BNF syntax.

The selection of the grammar description in Fig. 2 shows a pattern clause; for example, case expressions may have several pattern clauses. The `attrib` element describes the kind of the graph node, which distinguishes it from other node classes. The `attrib` element also provides space for further expansion, as a rule may contain more than one such element. The rest of the contents describe the structure of the node in a manner similar to a BNF. The elements `token`, `symbol` and `optional` should be evident; `repeat` designates a sequence of symbols separated by a given token type.

Besides the lexical and syntactical rules of Erlang, the XML code contains *annotations* providing information about how the semantic graph representation of programs should be built. For example, `symbol` contains a `link` annotation (an XML attribute of the `symbol` XML tag), which is the tag to the appropriate child, or in the case of repeats, all symbol children. In this case, the edges from the parent pattern clause node to the symbol children will be tagged `body`. The `separator` shows the type of the tokens inserted between the symbols.

2.3. Support for preprocessor constructs

In Erlang, preprocessor constructs are macros, include files and directives for conditional compilation. Of these, the first two are supported in RefactorErl, while the last is currently a work in progress and not described here.

The preprocessor analyzer, which runs after the scanner has already tokenized the source code, has three tasks in order to handle preprocessor constructs. Firstly, it has to store every original (pre-preprocessing) token that the construct contains in the semantic graph in order to enable the reconstruction of the original source later. Secondly, it performs macro expansion and file inclusion, and produces the processed (post-preprocessing) tokens. The processed tokens are also placed in the graph, and constitute the input of the parser. Thirdly, it makes additional semantic nodes and edges in the graph, which facilitate identifying and handling the constructs.

Handling include files. Erlang source files consist of forms, most of which describe functions, and some with special purposes. File inclusion and macro definition are described by such special forms. In fact, there are two distinct types of file inclusion forms, which differ in the paths where the included file is searched for, but the inclusion mechanism is the same for both. The preprocessor creates a form node for the inclusion form, and stores the tokens of the inclusion form below it. The preprocessor also creates a file node, which is linked from both the file node of the including file and the including form node. The contents of the include file are tokenized and passed on to the parser.

Handling macros. Macro definition forms, just like the include forms, are stored in the semantic graph along with all of their tokens. The macro may have arguments; all of its arguments and the rest of the macro body are marked by specific edges, see Fig. 3. Note that in this figure, some edges and some tags have been removed for clarity, and only selected children of the macro definition node are shown.

When a macro application is encountered, a macro substitution node is created in the semantic graph and linked to the macro definition node. This is shown in Fig. 4 as the rhombus `subst` node. The original tokens of the macro application are stored; the arguments are marked in order to facilitate access upon macro expansion. Virtual token nodes are created, and these are passed on to the parser. Such virtual token nodes are displayed as rhombus `token` nodes connected to the AST in Fig. 4.

3. AST CONSTRUCTION AND MANIPULATION

The purpose of using a semantic graph is to make information retrieval and modifications possible and convenient. Naturally, it should also be possible to recover the original source code stored in the graph. This section describes how these tasks are done in our tool and what choices we made.

3.1. Contractions in the skeleton of the semantic graph

Typically, ASTs are created by compilers during compilation. Such syntax trees are discarded after they have been used, and they are not subject to complex traversals. There are, however, applications in which the role of ASTs are augmented. In refactoring, for example, tree traversals are extensively used, because a lot of information is required that has to be acquired from different locations.

Syntax trees inherently involve parts that are unnecessary for information collection, or are structured so that they make it more tedious. One obvious case is that of chain rules: the code for their traversal has to be different for each node that occurs on the way, yet they contain no semantic information that could not be expressed using a single node.

Another case can be described by their functionality: nodes that contain similar information should be put in the same node group, and edges that describe similar connections should have the same tag. To give a concrete example, clauses in Erlang have parameters, guard expressions and a body, and there are associated tokens: parentheses and an arrow (for the grammar description of a node, see Fig. 2). Yet the actual appearance of the clauses can be vastly different, see Fig. 5 and 6 for other specific types of clauses.

The “if” clauses and the function clauses of Erlang both should have the same tags for the edges of their respective parameters (`sub`), guard expressions (`guard`), bodies (`body`), and tokens (`clex`, for ’lexical node of a clause’).

When collecting information, often either all parameters or all guard expressions are required at a time during a traversal pass, but seldom both at the same time of the traversal. Therefore, it is natural to partition the edges into groups depending on their uses. Since the partitions depend on the traversals used, the programmer has to decide by hand how groups should be made. This way, only a few groups have to be introduced as needed in a given application.

Another way to make the representation more convenient to traverse is to contract sequences. Sequences are common constructs in programming languages: they are repeated uses of a rule with inserted tokens as separators. Instead of having a slanted tree as constructed by a parser, it is more convenient for traversal purposes to represent sequences by a parent node with all of the repeated nodes and the intermediate tokens as its children.

Carrying out the above contractions has two main advantages. One is that much fewer cases have to be considered. In the case of Erlang, the grammar contained 69 nonterminals, which was reduced to three types of contracted nodes: `form`, `clause` and `expr`. Similar contractions are found in other tools as well, for example [12] and [13].

Even though currently there is only one language description available in our format, the one used for Erlang refactoring, the format is general enough to describe the grammars of other languages.

It is important that the approach should be adaptable to a wide range of grammars, because the choice of contraction node types, which correspond to the way abstactions are made over the concrete syntax tree, is different for each language. Indeed, the same application may employ several levels of abstraction with a different way of choosing contraction node types for each one.

3.2. Reacquiring the AST

When creating an abstract view of a syntax tree, the question of reproducing the original tree is always present. Sometimes only a fraction of the original tree is needed: for the very common task of reprinting the original source code, only the front of the tree has to be reproduced.

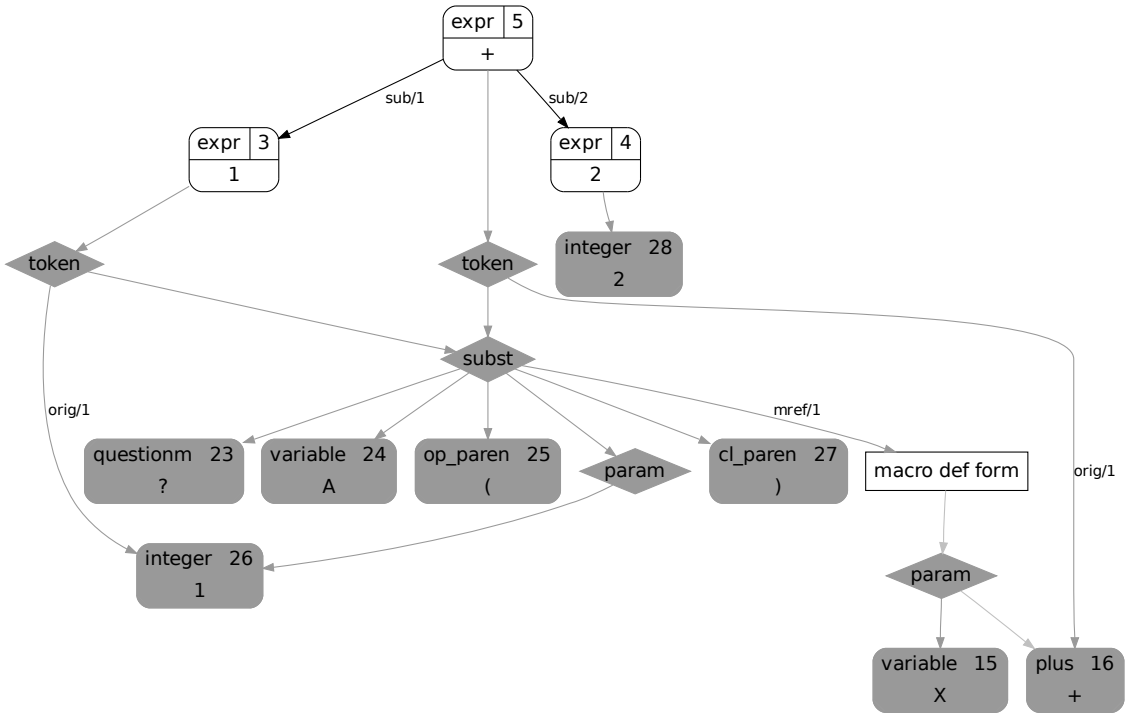


Fig. 4 Macro application in the expression `?A(1) + 2`.

```
if
    X == 1 -> Y = 2;
    true   -> Y = 3
end
```

Fig. 5 If clauses in Erlang.

```
to_list(Text) when is_atom(Text)    -> atom_to_list(Text);
to_list(Text) when is_integer(Text) -> integer_to_list(Text);
to_list(Text) when is_float(Text)   -> float_to_list(Text);
to_list(Text) when is_list(Text)    -> Text.
```

Fig. 6 Function clauses with guards.

Section 2.1 indicated that the outgoing edges of the nodes of the syntax tree are totally ordered. However, for efficiency reasons, the outgoing edges of the semantic graph are only partially ordered: order is defined only between edges that have the same tag. Whilst the leaves themselves are present in the abstract tree, their recollection is hindered by this partial ordering: given a total ordering, a simple walk could collect the leaves. This section discusses how the partial ordering can be turned into the desired total ordering.

One solution to this problem would be to store the actual order of the edges in the parent node. This would incur some space penalty, as this choice would store twice the amount of information about the order of the edges. Also, this information would have to be updated when modifying the tree, and the modifications would require considerations similar to the one presented below.

Another choice is to explicitly connect the leaves of the tree to each other in order. Enumerating them in such a way makes it trivial to collect them (provided that the first one is linked from the file node), but the task of updating them upon changes to the tree becomes unwieldy. Practically, changes in the syntax tree would have to be done at the level of the actual manipulation and at the leaf level in

parallel. Our experiments have shown that this is not manageable, partly because of the high amount of code needed.

Our current approach walks the tree by locally restoring the total order in the reached nodes. In addition to the partially ordered edges in the graph, the node structure described in the grammar is also needed. The simple BNF-like structure guarantees that these two can be merged, and together they produce the original order of the children of the node. Walking them in order gives us the leaves of the original syntax tree, or if it is needed, the tree itself. Strictly speaking, the tree is recovered only if the contractions (collapsed chain rules and the repeat shorthand) are expanded.

The front of the tree contains the token nodes in their original order. Since all whitespace information is contained in the tokens, and no punctuation tokens are omitted, the whole original file can be reprinted. Determining the token at a given position of the file can be done by doing a linear search on the original tokens in order.

The preprocessor layer between the scanner and the parser handles include files and macros (even ones that cross-cut the syntax), see section 2.3. During reconstruction, finding a node that originates from such a construct does not pose a challenge, as the preprocessor directly stores all of the relevant tokens of these constructs and links them to the appropriate skeleton nodes. A curious case is a macro that contains only whitespace and comments, as it does not naturally connect to the syntax tree; this can be treated as a special comment.

3.3. Subtree construction

Refactorings often have to create new syntax subtrees in their transformation phase. One possible solution for constructing such 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 that all the punctuation be manually filled in the refactoring tool, and would require separate grammars for each nonterminal to be generated. Another possibility is to do all the AST construction by hand, which is tedious and error-prone.

It would be a viable option to use code text and already present nodes in the construction, that is, start the parser with a sentential form as its input. This is a possible approach, but it would require extensive modification to the parser generator; practically, the generator would have to be rewritten. Since we are currently using an existing parser generator (yecc), and node creation is satisfactorily easy using our current method, we are not planning to implement this approach in the near future.

The rest of this section describes our proposal, which has been implemented in RefactorErl. Subtree construction is described along with an example, the ”extract function” transformation of RefactorErl [14]. The transformation extracts a sequence of expressions, or in this case, the single expression `F(F(1))` to a 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 Fig. 8).

Fig. 1 shows the AST of the match expression, part of whose right hand side is selected and is about to be extracted. Fig. 9 shows the match expression after the selection has been replaced by a function application. The function application contains a function name, opening and closing parentheses and the parameters with separating commas.

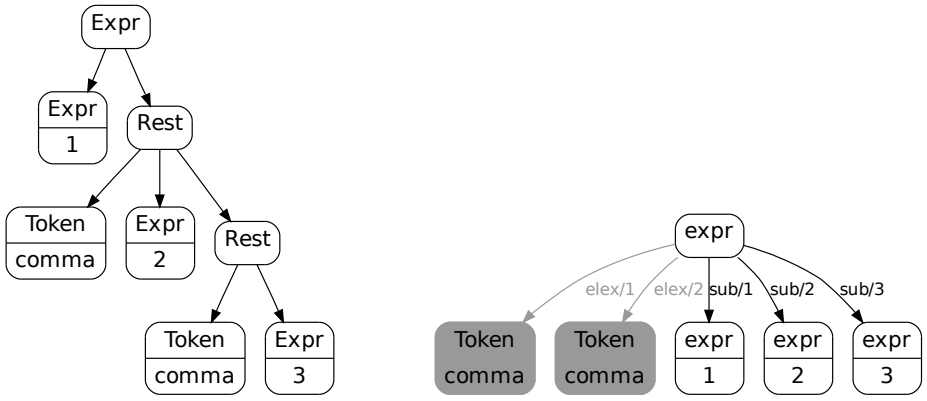
Fig. 10 shows the function definition to be created. The function definition contains opening and closing parentheses, an arrow and a stop token as punctuation, and the function name, the parameters of the function and the expressions of the body of the function. Also, if multiple parameters or body expressions were present, the function definition would contain additional separating comma tokens.

Most of the newly appearing tokens function as syntactic delimiters and can be automatically generated. Inserting the function definition into the AST requires only the function name, parameter names and the body expressions; inserting the function application into the AST 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.

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

The refactoring tool has to supply two pieces of information for the node creation algorithm. One is the type of the newly created node. The type of the node identifies the relevant rule in the grammar, which determines its structure. The other required piece of information is a description of the desired contents of the node. Since keywords and separator tokens can be automatically generated, and the grammar description 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 tool are consumed when creating the next symbol or construct from the rule description.

For example, when creating the function definition (node 2 in Fig. 10), the type of the node about to be created is a function definition. According to the grammar, function definitions consist of clauses that are similar to pattern clauses (see Fig. 2), the difference being that all function definition clauses begin with an expression, the name of the function, and that they have formal parameters. In the case of the extract function transformation, the name of the function is supplied by the user and the transformation determines the names of the formal parameters (`F`). The function name and the parameters are turned into expressions 21 and 22 by calling the node creation algorithm; as these construction steps involve only one node each, only the respective names are required in the content descriptions. Now the transformation can call the node creation algorithm for the function definition with expressions 21, 22 and 15, the latter removed from its previous position in the tree.



(a) AST. (b) Contracted AST.

Fig. 7 Sequence in the head of the list expression `[1,2,3|ListTail]`.

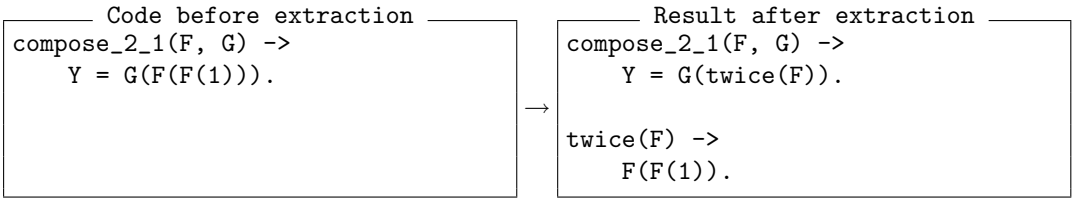


Fig. 8 Extracting expression `F(F(1))` into function `twice`.

Node creation is quite convenient in practice as well, requiring only a few function calls to the node creation algorithm. The addition of this feature made it possible to implement 15 refactorings in less than three months.

3.4. 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 given.

- 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 node description generated from the grammar, the user-supplied description of desired structure and the actual node present in the graph representation. 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. RELATED WORK

The design of the representation was shaped through years of experimentation and experience with refactoring functional programs [7]. Our previous refactoring tools [15, 16] used standard ASTs for representing the syntax.

Preprocessor structures such as file includes or macros are traditionally hard to handle, since they change the structure of the code in a cross-cutting way.

The easiest strategy to handle them is to use standard compiler tools. This is not sufficient, because the original source code is discarded. Also, this approach cannot preserve the original code layout.

Another way is to restrict 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 [17].

YSpec [18] uses macro productions, which are specialised when a macro application is found. In order to ensure termination, some restrictions have to be posed.

Macros in Lisp [19] cannot cross-cut the syntax: they are syntactically well-behaved. This property makes it possible to make macros a more integral part of the syntax tree in such languages. However, syntactically correct macros can still cause problems, as they can clash with, or override, predefined symbols. Hygienic macros present in Scheme [20], a dialect of Lisp, deal with this problem.

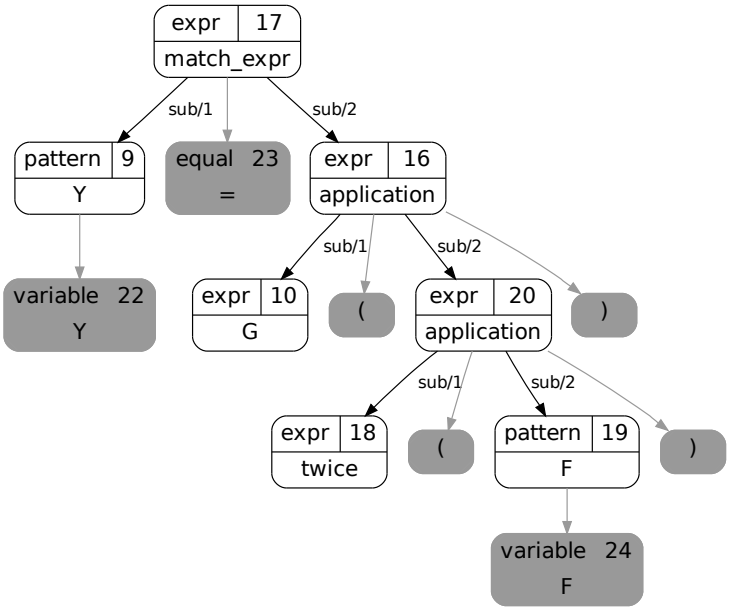


Fig. 9 The new function application replaces the selection.

Proteus [21] uses a similar representation to ours to refactor C++ programs. It is based on an extended, layout preserving AST which uses recorded macro expansion. For Erlang, no such tool exists yet. Advantages of our approach also include being able to fine tune the generated graph for a specific problem because of the extensible grammar definition and the graph representation that contains all necessary information in a unified way.

There are already some refactoring tools for other functional languages. A prototype refactoring tool for Clean [15] uses a database for storing the syntax and semantic information. Since the database schema is fixed, it is harder to adapt the tool in case of changes in the language. Since parsing Clean doesn't involve macros, no preprocessing phase is needed.

Overbey and Johnson [12] propose an annotated AST generator framework. There are direct correspondences between their work and ours, e.g. the field distinguishing annotations and the edge labels, and how they add whitetext in front of, or after, the tokens. The main difference between RefactorErl and their tool Ludwig is how they represent data. Ludwig generates Java classes whose interfaces describe the edges of the AST while the private parts contain whitetext information. RefactorErl stores a complete semantic graph in a database, which enables it to collect and then efficiently query static semantic information.

Spinellis [22] describes the tool CScout that can inspect and refactor C programs. CScout tags the identifiers found in the source code with their location and then unifies them; in RefactorErl, this task is relegated to the semantic analyser modules not discussed in this paper. In CScout, meticulous attention was given to the preprocessor facilities of the language. In particular, conditional compilation is better supported than in the current version of RefactorErl. This is due to the fact that conditional compilation is much more frequently used in C than in Erlang.

Wrangler [23], another refactoring tool for Erlang also uses an annotated AST for its inner representation of programs. It transforms the tree by walking it, fusing information collection and refactoring. We are currently investigating whether this approach or using our semantic graph queries yield better performance for large bodies of code.

SableCC [13] uses semantic annotations to transform a concrete syntax tree into an abstract one. Apart from where the AST transformation is done (in the case of RefactorErl, the transformations are syntactic, in SableCC, the annotated semantic routines are executed), they are quite similar: the most common AST abstractions in SableCC seem to be rule contractions.

The Java language tools srcML [24], JavaML [25] and JaML [26] use XML to model Java source code. XML documents can be equipped with schema information against which they can be checked. If the schema is formulated in XML itself, subtree construction algorithms similar to the one presented in this paper can be devised.

NetBeans and Eclipse are two popular IDEs that support refactoring. While NetBeans supports development in Java only, Eclipse has plugins for many other languages. While the transformations they offer are sound, they are not complete: they do not support all language features, e.g. they have problems with refactoring and syntax highlighting identifiers that contain Unicode characters. Also, NetBeans and the C++ utilities of Eclipse reformat the indentation of moved code parts, which is not appealing to the user; the Java refactorer of Eclipse handles this case nicely.

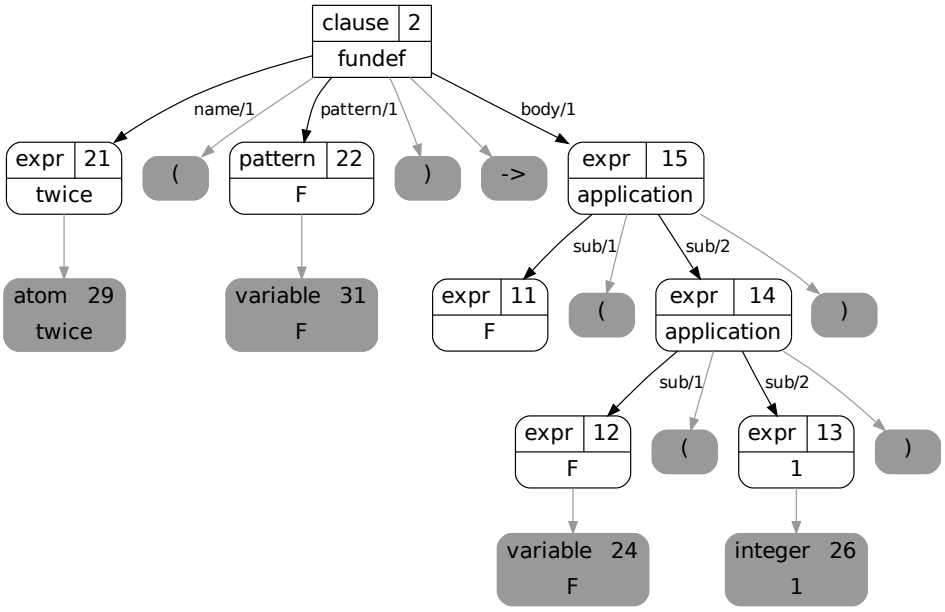


Fig. 10 The representation of the newly created function definition.

HaRe [27] represents the informations it needs in an AST with an additional token stream. This choice allows them to preserve the layout, but since the token stream and the AST are not in direct connection to one another, the tool spends many resources keeping them synchronised.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have described an approach to represent program code in such a way that makes refactoring convenient. The representation is based on the AST, however, unlike other existing systems that add annotations to the tree, this representation extends it to a semantic graph. The current implementation of the tool, RefactorErl v0.6, is available from [3]. The tool has been tested on millions of lines of industrial Erlang source code, and it also turned out to be very adaptable and useful for purposes other than refactoring, e.g. reorganizing the module structure, testing and visualization.

The representation is suitable for reproducing the original program code with punctuation, whitespace, macros and file inclusion intact. It can even load all published entries of the obfuscated Erlang competitions in 2005, 2006 and 2007 [28]. On the other hand, when performing syntactic transformations, whitespace on the beginning and end of changed parts and the indentation of the changed code is far from ideal yet. This is one area that we plan to improve in the future.

The tool also contains preliminary support for another kind of preprocessor construct, conditional compilation. This feature is expected to appear in future versions of RefactorErl.

ACKNOWLEDGEMENT

The authors are grateful for the support from Ericsson Hungary and ELTE IKKK. Preliminary results to this paper were presented at CSE'08 and IFL'08.

REFERENCES

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

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

[3] The RefactorErl homepage. `http://plc.inf.elte.hu/erlang/`.

[4] Erlang Official homepage. `http://erlang.org/`.

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

[6] Erlang 5.5.5 reference manual. `http://www.erlang.org/doc/reference_manual/part_frame.html`

[7] 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.

[8] L. Lövei, Z. Horváth, T. Kozsik, R. Király and R. Kitlei. *Static rules of variable scoping in Erlang.* In The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, July 1–4, 2004.

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

[10] K. Sagonas, D. Luna. *Gradual Typing of Erlang Programs: A Wrangler Experience.* Seventh ACM SIGPLAN Erlang Workshop (Erlang'08), pp. 73-82, 2008.

[11] U. Wiger. *XMErl - Interfacing XML and Erlang.* In the Sixth International Erlang/OTP User Conference (EUC 2000), Stockholm, Sweden, October 3, 2000.

[12] J. Overbey and R. Johnson. *Generating Rewritable Abstract Syntax Trees.* In proceedings of the 1st international conference on software language engineering (SLE 2008), Toulouse, France, September 29–30, 2008.

[13] É. Gagnon. *SableCC, an object-oriented compiler framework.* Thesis submitted to the Faculty of Graduate Studies and Research, McGill University, Montreal, 1998.

[14] M. Tóth. *Erlang Refactoring: Extract Function.* Master thesis, ELTE, Budapest, Hungary, 2008.

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

[16] Z. Horváth, L. Lövei, T. Kozsik, A. Víg, and T. Nagy. *Refactoring Erlang programs.* Periodica Polytechnica, Electrical Engineering, 51(3-4):75-84, 2007.

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

[18] P. Thiemann, M. Neubauer. *Macros for Context-Free Grammars.* `http://www.informatik.uni-freiburg.de/~thiemann/haskell/YSPEC/`

[19] D. Weise and R. Crew. *Programmable syntax macros.* ACM SIGPLAN Notices, v.28 n.6, p.156-165, June 1993.

[20] W. Clinger. *Hygienic macros through explicit renaming.* ACM SIGPLAN Lisp Pointers, Volume IV, Issue 4, p. 25–28, 1991.

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

[22] D. Spinellis. *Global analysis and transformations in preprocessed languages.* IEEE Transactions on Software Engineering 29 (11) (2003) 1019–1030.

[23] H. Li and S. Thompson. *Tool Support for Refactoring Functional Programs.* In Partial Evaluation and Program Manipulation (PEPM'08), San Francisco, California, USA, January 2008.

[24] J. I. Maletic, M. L. Collard, and A. 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.

[25] G. 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 159–177. North-Holland Publishing Co. Amsterdam, The Netherlands, The Netherlands, 2000.

[26] 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.

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

[28] Erlang Consulting. *2008 Obfuscated Erlang Programming Competition homepage.* `http://www.erlang-consulting.com/obfuscatederlang.html`

Received May 5, 2009, accepted August 24, 2009

BIOGRAPHIES

Róbert Kitlei is Assistant Lecturer of Programming Languages and Compilers at Eötvös Loránd University in Budapest, Hungary. His scientific research is focusing on syntactic and semantic program representation, and its application to refactoring Erlang programs.

László Lövei is Assistant Lecturer of Programming Languages and Compilers at Eötvös Loránd University in Budapest, Hungary. His scientific research is focusing on syntactic and semantic program representation, and its application to refactoring Erlang programs. He is the lead developer of the RefactorErl tool.

Tamás Nagy is PhD student at the Department of Programming Languages and Compilers at Eötvös Loránd University in Budapest, Hungary. His research topic is the application of data flow analysis in refactoring.

Zoltán Horváth is Professor at, and Head of, the Department of Programming Languages and Compilers and Vice Dean of the Faculty of Informatics at Eötvös Loránd University in Budapest, Hungary. He defended his habilitation thesis in 2004; the title of his thesis was "Verification and Semantics of Mobile Code Written in a Functional Programming Language". Current topics researched under his supervision include language design, construction of programming language processing tools, formal methods and scheduling in grids.

Tamás Kozsik is Associate Professor at the Department of Programming Languages and Compilers at Eötvös Loránd University in Budapest, Hungary, where he teaches programming languages (Ada, Eiffel, Haskell, Java) and synthesis of correct parallel programs. He received his PhD in 2006 on program verification and type systems. He is researching formal methods, programming language processing tools, and scheduling in grids.