Generic syntactic analyser: ParsErl *

Róbert Kitlei, László Lövei, Tamás Nagy, Anikó Nagyné Vig,
Zoltán Horváth, Zoltán Csörnyei
Department of Programming Languages and Compilers,
Eötvös Loránd University, Budapest, Hungary
{kitlei,lovei,n_tamas,viganiko,hz,csz}@inf.elte.hu

Abstract

The increasing demand in automatic code transformation tools – which can preserve the layout, and can handle the whole macro syntax – led us to develop our scanner and parser tool. ParsErl is a generic syntactic analyser for Erlang. The scanner and the parser are generated from an XML definition of the grammar. The result of the scanning process is a graph, which can be optimised or balanced for applications. The tool can preserve the original layout of the source code, including the original macro definitions. Our preprocessor creates connection between the original source code's tokens and syntax tree's nodes. We can provide the substituted and parsed code for the applications and we can generate the original source code back, when it is needed.

1 Introduction

The increasing amount of codebase which has to be maintained resulted in an increasing demand in automatic code transformation tools. For example, refactoring tools which can change (usually applied in order to improve) the structure of the code without changing its behaviour [2, 3, 4, 6, 5].

These tools work on a higher abstraction layer than textual format. The usual approach is to apply syntax analysis that produces an abstract syntax tree (AST) of the source code. The standard Erlang parser with the `syntax_tools` application provides an interface to produce and work with such an AST [1, 7, 8].

The problem with this approach is that this parser was designed for code generation. It provides an interface which can generate text from the AST, but this result will be pretty printed, because the parser discards the layout, whitespace, and punctuation while building the syntax tree. These information are irrelevant for code generation but highly valuable for the code transformation

*Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK and Ericsson Hungary.
A full technical paper about the design of the internal structure was submitted to CC2008.

tools. Preserving this information we can preserve the original layout in contrast to pretty printing.

The other problem with the standard tools arise when the language supports macros. Macros are usually substituted with their definition by a preprocessor before parsing. The Erlang tools can support macros without substituting them if they “behave” well. If a macro cuts syntactic entities in half, the tools cannot parse it. This means that some code can be compiled, but cannot be parsed by the standard tools before prepocessing.

In this paper we will show that these problems can be solved with a new parser, if the design aim is the layout preserving, and the support of the full macro syntax. Furthermore we will give an API which makes it possible to apply the framework in different projects.

2 Motivation

The prototype version of our refactoring tool, RefactorErl, suffered from the above problems. Because in refactoring tools it is a crucial point to be able to support the whole syntax and to keep the layout as it was as much as possible. This is very important, because the pretty printing makes it hard to follow the changes made in the code, let alone carry out further changes even with the refactoring tool not to mention by hand.

3 Structure of the tool

In Figure 1 we show the layers and parts of the tool. We generate the scanner and the parser based on an extensible XML description of the Erlang grammar. After the scanning and parsing process is done, an application can work on the graph representation. Our layout preserving printer can restore the original source code in textual format from the graph representation.

3.1 XML

Both the lexical elements and the syntactic rules, and the resulting structure reside in the same XML file. This makes the definition easily adoptable, customisable to language changes, and to different application needs. Obviously the former happens really rare, because that would need changes in a lot of applications, and could cause issues with backward compatibility.

3.1.1 Lexical elements

Lexical elements are described by the element *lexical*. Patterns (elements *pattern*) are quite similar, only they don’t constitute an element themselves. Important patterns are the whitespace-and-comments before and after the tokens (named PRE and POST), which are included in the token text themselves. Patterns can be incorporated using a *match* element. Elements may further consist

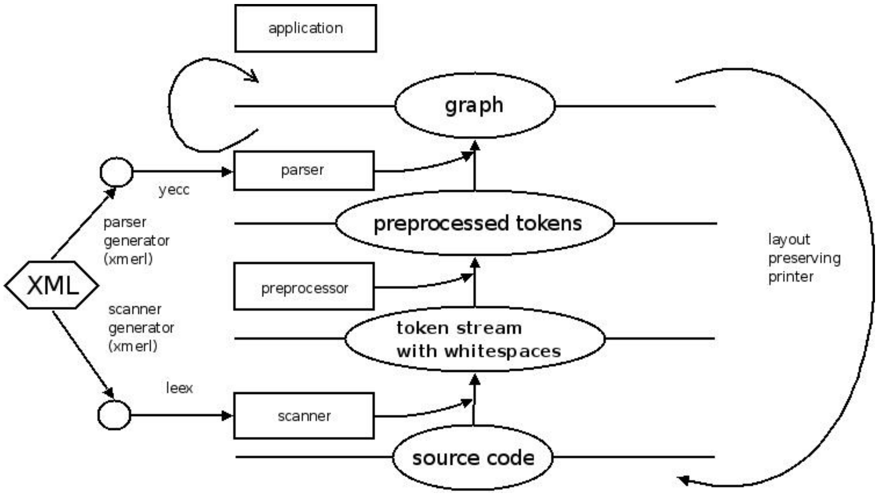

Figure 1: The structure of ParsErl

of plain text, branches, optional and repeated parts, tagged by *text*, *branches*, *opt* and *rep* respectively. There are some additional facilities for easier character inclusion: *chars-of* and *chars-but*, which permit all characters included in (or excluded of) a given set.

The following example shows the description of integers.

```
<lexical name="integer">
  <match name="PRE"/>
  <opt>
    <branches>
       <br>
         <text>1</text>
         <chars-of><range><from>0</from><to>6</to></range>
         </chars-of>
       </br>
       <br>
         <chars-of><range><from>2</from><to>9</to></range>
         </chars-of>
       </br>
     </branches>
     <text>#</text>
   </opt>
   <match name="Digit"/>
   <rep>
     <match name="Digit"/>
   </rep>
   <match name="POST"/>
 </lexical>
```

The following regular expression is generated from this description:

```
{PRE}(1[0-6]|[2-9]#)?{Digit}{Digit}*{POST}|
```

3.1.2 Syntax elements

The syntax elements describe the context free grammar rules. All rules with the same head symbol are organised under a *ruleset* element. They may contain rules that are not represented in the graph themselves: these are called *copy-rule*. All other rules have to specify in which class do they belong. This way we can simplify the syntax graph by storing only so much information as necessary. For example, all different kinds of expressions are in class *expr*, and are not distinguished from each other further on.

Rule elements (the right hand sides of rules) consist of *tokens* and *symbols*. For the sake of brevity, elements *optional* and *repeat* are also available.

With the *symbols* we have to define how we want it to be connected to the head symbol. For example the function clause's pattern elements are connected to it with a link tagged with *pattern*, the guards with *guard* and the body's elements with *body*. These tags will be used for information retrieval.

Rules may also contain attributes that are stored as additional information in the graph during parsing. For example, guard sequences may contain *conjunctions* and *disjunctions*; both are represented as an *expr* node with the appropriate kind as attribute.

The following example and Figure 2 show the rules for function clauses.

```
<ruleset head="FunClause">
  <rule class="clause">
    <attrib name="type">funcl</attrib>
    <symbol name="Atom" link="name"/>
    <token type="op_paren"/>
    <optional>
      <symbol name="Expr" link="pattern"/>
      <repeat>
        <token type="comma"/>
        <symbol name="Expr" link="pattern"/>
      </repeat>
    </optional>
    <token type="cl_paren"/>
    <optional>
      <token type="when"/>
      <symbol name="Guard_seq" link="guard"/>
    </optional>
    <token type="arrow"/>
    <symbol name="Expr" link="body"/>
    <repeat>
      <token type="comma"/>
      <symbol name="Expr" link="body"/>
    </repeat>
  </rule>
</ruleset>
```

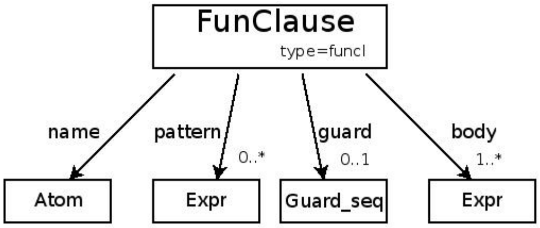

Figure 2: The function clause rule without the tokens

3.2 Scanner

The scanner is automatically generated from the XML definition with an XSLT. The XSLT is written with the Erlang's xmerl application [13, 14, 15]. The XSLT transformation's result is the input of the leex application [12].

The definition can be easily adjusted to keep the comments and the whitespace information or discard them. In our case we chose to attach the whitespace information and comments to the lexical categories of the language, therefore there is no whitespace token. The interface of this module is the standard interface provided by the leex application.

3.3 Preprocessor

A middle layer has been introduced between the scanner and parser to be able to support any kind of macros which are allowed in the languages definition. This layer defines connection between the original source code's tokens and syntax tree's nodes. This relation is not trivial because the syntax tree can only be built when the macros had been substituted. Therefore our preprocessor has to be aware of the structure being built by the parser. As a result of this the preprocessor does not provide a standard interface for invocations. It is embedded into the parser.

3.4 Parser

The parser is also generated from the XML definition. The XSLT transformation's result is the input of the yecc application [9, 10, 11]. The built structure can be adjusted just by adjusting the definition in the XML. The API is extended compared to the yecc's default interface in order to get the correct result structure. Parsing one form at a time is supported, and additionally the extended API provides means to parse a whole file as well.

3.5 Graph

A graph is the result structure of the parsing. The shape of the graph only depends on the definition of the syntax in the XML file. Therefore based on our preferences/needs the shape can be adjusted to a certain extent. Because the yecc uses LALR-1 analysation method the structure has to be resemble a tree.

For example in our refactor tool we decided not distinguish between the different expressions. The expressions' type is always expression. The standard parser's expression types are just attributes.

4 Information retrieval

High level information retrieval is supported by a query language that makes it easier to traverse graph structures with fixed depth. This query language consists of *path expressions*. To evaluate it a start node and the list of links we want to follow from the start node is required. The direction and filters of the links can be given. Direction can be forward or backward. The possible filters are:

```
Filter = {Filter, 'and', Filter} | {Filter, 'or', Filter} |
         {'not', Filter} | {Attrib, Op, term()}
Attrib = atom()
Op = '==' | '/=' | '=<' | '>=' | '<' | '>'
```

The links also have indices which start at 1. Therefore it is possible to choose one link or an interval of links.

```
Index = integer() | {integer(), integer()} | {integer(), last}
```

`Start, End` means the indices larger or equal than `Start` and smaller than `End`.

For example (using our structure) if we want to retrieve the module name of the source file we can write a *path expression* like this. Suppose we have the root of the file in the `Root` variable:

```
path(Root, [{form, {kind, '==', module}}, {attr}])
```

The result would be a list containing one element. The module name is the result node's attribute which can be obtained by another function call.

5 Linking with other applications

The API demonstrated in Section 4 provides an interface to other applications which can be easily used. The built graph structure can be fine tuned to specific applications. The information retrieval mechanism - the functions, parameters - do not change when the defined structure changes. These altogether yield a highly adoptable/optimisable structure.

6 Conclusion and Future work

In this paper we have shown that it is possible to support the whole syntax of the Erlang language with a parser which can retain the original layout of the code. Furthermore, having the definition of the whole language and the result structure defined in one XML file makes the language definition easily adjustable to changes in the language. The resulting structure can be easily adapted to any specific problem. For example balancing the resulting graph's height and width for optimising to the application's algorithm. The framework even makes it possible to add extra information to the graph which cannot be derived directly from the syntactic rules.

The `ifdef, ifndef` macros introduce further difficulty for example consider the following code:

```
-ifdef(debug).
-define(LOG(X), io:format("{~p,~p}: ~p~n",
                          [?MODULE,?LINE,X])).
-else.
-define(LOG(X), true).
-endif.
```

The macro's body is different depending on the value of the condition. Even if we work on the unsubstitued version of the source, we have to consider what the substituted code would be.

A further development would be to develop the printing mechanism to be able to parameterise it with design rules to enforce the same layout of different developers.

References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Erlang 4.7.3 reference manual.
http://www.erlang.org/download/erl_spec47.ps.gz

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

[9] Torbjörn Törnkvist
How to improve the performance of YECC-generated Erlang (JAM) parsers
Published in the Software Engineering Research Center of the RMIT University(SERC), Melbourne, Australia. December 12, 1997.
http://www.erlang-projects.org/Members/mremond/serc/
how_to_improve_the_p/block_10914819836344/file

[10] Magnus Fröberg
Automatic Code Generation from SDL to a Declarative Programming Language
In Proceedings of the Sixth SDL Forum, Darmstadt, Germany, October 1993.
www.erlang.se/publications/sdl2erlang.ps

[11] yecc documentation.
www.erlang.org/doc/man/yecc.html

[12] Leex beta version download page
http://tinyurl.com/yvl6tp

[13] Ulf Wiger
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

[14] Mickael Rémond
XML and Erlang: Building a Powerful Data Management Tool.
In the Sixth International Erlang/OTP User Conference (EUC 2000),

Stockholm, Sweden, October 3, 2000.
http://www.erlang.se/euc/00/remond/mgp00001.html

[15] xmerl documentation.
http://www.erlang.org/doc/apps/xmerl/