STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

THE RECONSTRUCTION OF A CONTRACTED ABSTRACT SYNTAX TREE

RÓBERT KITLEI

Abstract. Syntax trees are commonly used by compilers to represent the structure of the source code of a program, but they are not convenient enough for other tasks. One such task is refactoring, a technique to improve program code by changing its structure.

In this paper, we shortly describe a representation of the abstract syntax tree (AST), which is better suited for the needs of refactoring. This is achieved by contracting nodes and edges in the tree. The representation serves as the basis of the back-end of a prototype Erlang refactoring tool, however, it is adaptable to languages different from Erlang.

In turn, we introduce an algorithm to reconstruct the AST from the representation. This is required in turn to reproduce the source code, the ultimate step of refactoring.

1. Introduction

The ASTs constructed using context-free grammars is the representation most applications choose to describe the syntactic structure of source code of programming languages. Most applications use standard lexers and parsers that are designed with the goals of compilers in mind. Compilers – and therefore their standard tools – drop inessential infomation such as punctuation and whitespace after using them to determine token boundaries. Such information is important if one has to preserve the source code as a whole. Also, ASTs are not designed to support searching, as this feature is not required in compilers, the most common users of ASTs.

The above representation does not sufficiently support some applications. An alternative representation is proposed by the Erlang refactoring group at

Received by the editors: September 16, 2008.

2000 *Mathematics Subject Classification.* 68N20,68Q42.

1998 *CR Categories and Descriptors.* D.2.10. [**Software**]: Software engineering – *Representation*.

Key words and phrases. Abstract syntax tree, Syntactic reconstruction.

Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK and Ericsson Hungary.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

Eötvös Loránd University (Budapest, Hungary), although this representation has proved useful for purposes other than refactoring as well. The group proposed this representation after previous experience with refactoring [5, 7]. Details about the representation and the refactoring tool can be found in [4].

Although the new representation is more convenient for many purposes, e.g. refactoring, there was a trade-off between usability and functionality, described in detail in section 3.1. Namely, for standard compiler tools, pretty-printing the source from the constructed AST is straightforward using a depth-first algorithm. However, since the new representation does not have all child edges of a node in order, a more elaborate algorithm was needed, which is described in section 3.

The structure of the paper is as follows. In section 2, the representation of the graph is described to such depth as is necessary for understanding the rest of the paper. Section 3.1 poses the central problem of the paper. The rest of section 3 proposes an algorithm that solves this problem. Sections 3.2 and 3.4 in this section describe the contribution of the paper. Finally, section 4 lists related work.

2. Representation structure

2.1. **Node and edge contractions.** ASTs built on top of source codes are typically created by compilers in compilation time. Such syntax trees are discarded after they have been used, and their construction does not involve complex traversals: they follow the construction of the tree. 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 can be acquired from different locations.

In order to facilitate these traversals, a new representation of the AST was introduced, which is described in detail in [4]. Here we give an overview of the relevant parts of the representation.

ASTs 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 information contained in them could be expressed as a single node, yet the traversing code has to be different for each node that occurs on the way.

Another case can be described by their functionality: the edges of the nodes can be grouped so that one traversal should follow exactly those that are in one group. To give a concrete example, clauses in Erlang have parameters, guard expressions and a body, and there are associated tokens: parentheses and an arrow. Yet the actual appearance of the clauses can be vastly different, see Figures 1 and 2. When collecting information, often either all parameters

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

FIGURE 1. If clauses.

```
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.
```

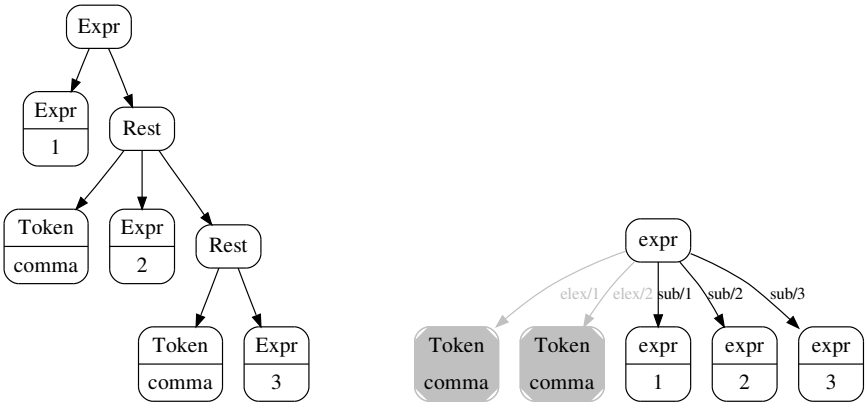
FIGURE 2. Function clauses with guards.

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 along 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 as few groups have to be introduced as needed in a given application.

Another way to make the representation more compact is to contract repetitions. Repetitions are common constructs in programming languages: they are repeated uses of a rule with intercalated tokens as separators. Instead of having a slanted tree as constructed by an AST, it is more convenient for traversal purposes to represent them by a parent node with all of the repeated nodes and the intermediate tokens as its children. As a matter of fact, in the example in the above paragraph the parameters and guard expressions are already a result of such a contraction. These contractions are similar to the list formation annotations in Overbey and Johnson [2].

Performing 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 contracted node groups: forms, clauses and expressions.

Since the contraction groups are different for each language (and may even differ in each application, depending on the needed level of detail), it is important that the approach should be adaptable to a wide range of grammars. This is one of the reasons why an XML representation was chosen. The grammar rules, the contraction groups and the edge labels are described in this file. The scanner and parser are automatically generated from this file. The contracted



(a) AST. (b) Contracted AST.

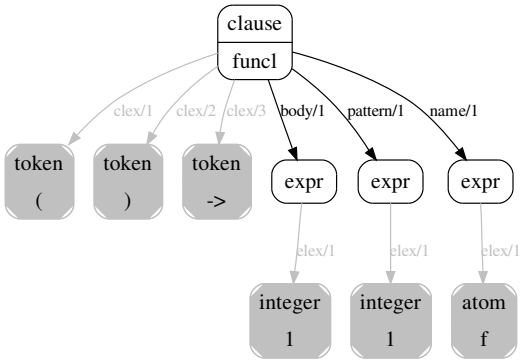
FIGURE 3. Repetition in the expression `1,2,3`.

structure is automatically constructed during parse time (not converted from an AST).

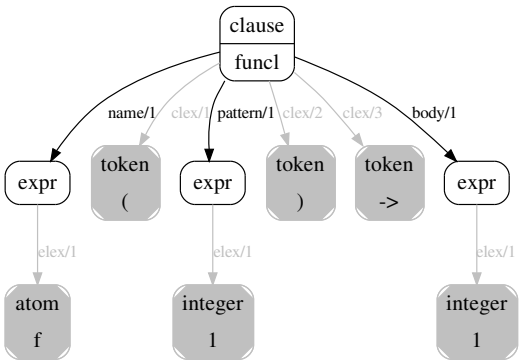
2.2. **Representation of the contracted AST.** The inner nodes of the contracted AST are the contracted nodes, which also contain the originating nonterminal as information. The leaf nodes of the contracted AST are the tokens, which contain the token text and the whitespace before and after the token. The nodes are connected by labelled edges; the labels determine the contraction classes they can connect.

Contractions do not fully preserve edge ordering: order is preserved only between the edges with the same label, not between different labels. This is why the original AST cannot be restored easily: in Figure 4b, it is not possible to determine whether the tokens of the clause come before, after or in between the expressions. To make it possible, more information about the structure of the contracted nodes is needed.

The lack of order between label groups is the result of using a database for storage, which is required for fast queries. However, it is expected to be a good trade-off, since the exact AST order of the nodes is seldom needed (most importantly, when reprinting the contents of the graph into a file), while it provides queries in linear time of their length. The order of the links with the same label, which is important during queries, is retained.



(a) Part of an automatically printed contracted AST. The order of the edges between groups in unknown.



(b) The nodes rearranged in the right order. The order within the groups is retained. The tokens read: `f(1) -> 1`.

FIGURE 4. A contracted AST node with a `body`, a `pattern`, a `name` and three `clex` edges.

3. Reconstruction of the AST

3.1. **Problem when reproducing the original token order.** In the previous versions of RefactorErl, the token nodes in a file were linked by edges with the special label `next`, with the first token linked from the file by `first_token`.

This solved the problem of getting the original tokens: they could be acquired by getting the first token, then iterating on the `next` edges until there were none left. Another related question, determining the token at a given position in the file, was also solved easily by iteration on the `next` edges, and calculating the remaining positions. However, these edges have proved to be too difficult to handle when manipulating the syntax tree: the `next` edges would have to be synchronised each time parts of the syntax tree were inserted, removed or moved. Also, when manipulating repeat constructs such as lists, some tokens (in the case of lists, the separating commas) would have to be dealt with.

The approach taken in this paper is different. Instead of repairing the `next` edge links, they are omitted altogether. This immediately solves the problem that occurs when manipulating the syntax tree, because the adjacent tokens are not linked anymore. At the same time, the two other questions are reopened: how to get the token by position and how to print the file. In the rest of the chapter, a method is presented to reproduce the AST. This also yields the original tokens as the front of the tree. Using the original tokens, both questions are trivially answered.

3.2. **Grammar rule constructs.** The chosen grammar description is close to a BNF description. The grammar rules are grouped by what contraction group their head belongs to. Rules, of course, may have more alternatives. The right hand sides of rules consist of a sequence of the following:

- **tokens**, that contain the token node label,
- **symbols**, that contain the child symbol's nonterminal and the edge label,
- **optional constructs**, sequences that either appear or not in a concrete instance and
- **repeat constructs** that contain a symbol and a token; its instances are several (at least one) symbols with tokens intercalated.

Since optionals and repeats may contain one another, we shall refer to the number of contained nestings as the depth of the construct.

As an example that contains both constructs described above, let us examine the structure of lists. The structure of lists is described as follows. Lists start with an opening bracket token and end with a closing bracket token. Between them is an optional construct. The optional part consists of a repeat construct. The repeat construct uses comma tokens to separate symbols that are linked using "sub" edges from the parent node. The portion of the actual Erlang code that shows the above structure is shown in figure 5 in order to have a more concise overview.

Lists can be empty lists, or lists containing expression symbols separated by comment tokens. In the first case, the optional part is not present. In

```
[{token,"op_bracket"},
 {optional,[{repeat,"comma","sub"}]},
 {token,"cl_bracket"}];
```

FIGURE 5. The structure of lists as an Erlang structure used in the actual implementation. Slightly abridged.

the second case, the optional is present. If there is one element in the repeat construct, there is exactly one symbol element present which denotes the expression.

$$\underbrace{[}_{\text{token}}\ \underbrace{\ }_{\text{empty optional}}\ \underbrace{]}_{\text{token}}$$

$$\underbrace{[}_{\text{token}}\ \underbrace{1}_{\text{repeat in optional}}\ \underbrace{]}_{\text{token}}$$

$$\underbrace{[}_{\text{token}}\ \underbrace{1,2,3,4,5,6,7,8+9,f()}_{\text{repeat in optional}}\ \underbrace{]}_{\text{token}}$$

The grammar description contains the following restrictions. First, no optionals may start with another optional. Second, two repeats in the same rule may not contain the same symbols, nor tokens. Third, no constructs (optionals and repeats) may have a depth of more than two.

The main reason for these restrictions is that they help prevent ambiguities, as seen in the `absence` and `multichoice` constructs in the description or reconstruction.

The third restriction is not necessary for theoretical, but for practical purposes: it is there to keep the processing algorithm described later at a manageable size and complexity while not deducing the expressive power of the constructs too much. Indeed, for a construct at any depth, a new nonterminal can be introduced to take its place, thereby reducing the depth of the parent construct. This way, the depth of the constructs could be limited to one; practice has shown that two is a reasonable limit.

The grammar is expressive enough, as even without the constructs it has Chomsky class L_2.

The first restriction can be enforced by the DTD of the XML. The third restriction could also be enforced if the inner optionals and repeats would have different names, at the expense of comfort.

3.3. **Derived constructs used in reconstruction.** The rule descriptions above are sufficient in most cases to reconstruct the original node order of a node in the contracted AST by looking at only the nonterminal of the node, the node's child links and the rule description. Yet there are two types of rules where these data are not enough. In these cases, another kinds of constructs have to be prepared before reconstruction. These structural constructs are automatically derived from the syntax description like the scanner and the parser. Both of these constructs require information about the children nodes, and conglomerate several grammar rules.

The first skeleton construct is called **absence multi-rule construct** because it selects the appropriate grammar rule based on the absence or presence of a token or a symbol. The following example shows a fun-expression that can either have explicit clauses (in the first case) or can be an implicit fun expression, just showing the function name and arity (the second case). Here, the only way to decide which rule to use is to check for the `end` token: if it is present, it is the first rule, if it is not, the second.

$$\underbrace{fun}_{\text{token}}\ \underbrace{(1) \to ok;\ (2) \to error}_{\text{repeat}}\ \underbrace{end}_{\text{token}}$$

$$\underbrace{fun}_{\text{token}}\ \underbrace{\text{another\_module/2}}_{\text{repeat}}$$

Named functions have the name of the function as a subexpression in the beginning of each clause. The clauses of unnamed functions start immediately with the parameter list in parentheses. The only way to decide between them is to search for the symbol at the beginning. (Note that symbols also contain the link label. Its omission, similar to calling the parameter list a "repeat in optional," is a simplification.)

$$\underbrace{search}_{\text{symbol}}\ \underbrace{(}_{\text{token}}\ \underbrace{Structure, Pattern}_{\text{repeat in optional}}\ \underbrace{)}_{\text{token}}\ \underbrace{\to}_{\text{token}}\ \underbrace{\ldots}_{\text{repeat}}$$

$$\underbrace{(}_{\text{token}}\ \underbrace{Structure, Pattern}_{\text{repeat in optional}}\ \underbrace{)}_{\text{token}}\ \underbrace{\to}_{\text{token}}\ \underbrace{\ldots}_{\text{repeat}}$$

The second skeleton construct the **multichoice multi-rule construct**. In it, there is a list of possible present symbols or tokens. The actual rule can be decided depending on which of the symbols (or tokens) occur. The symbols (or tokens) listed are mutually exclusive: one and only one occurs, provided that the source is valid.

Both if and case clauses are branch clauses and they may look identical. Similarity occurs when the case clause has no guard and the guard of the

if expression is a single variable. They can be separated only if they make different links to their first symbol as "guard" and "pattern" respectively.

Infix expressions provide an example for a token-based multichoice construct. Logical operators `andalso` and `orelse` (and several other operators) can function on the same pair of arguments. Here, checking all the possible token types, exactly one will be present, and this of course determines the operation as well.

3.4. **AST reconstruction.** From the XML syntax description, a node structure skeleton is automatically generated. It assigns to each contracted node type either a one-rule structure, or an absence or multichoice multi-rule construct.

The syntax tree can be reconstructed using a recursive algorithm. Starting from the node in the tree that corresponds to the file, we do the following.

(1) We determine the structure of the actual rule which is used. If a one-rule structure is assigned to the parent node, it is the structure; if a multi-rule construct describes it, we have to check the children of the node as well.
(2) The sequence in the structure is processed.
    (a) For any token or symbol, take the next fitting one.
    (b) For repeats, take all symbols (altogether n) with the appropriate edge label, and take $n - 1$ fitting tokens.
    (c) For optionals beginning with a token or symbol, use the optional sequence if a fitting child is present.

Tokens' edge labels are determined by the type of the parent node. We call a token node fitting the token in the description if it is linked to the parent node by such a link. A symbol is called fitting in a similar way, except that for symbols, the description explicitly contains their expected links.

Using the above algorithm, the original AST can be recovered. Strictly speaking, this is not the AST, as chain rules are still not expanded; this does not add significant information, and can easily be done, should the need arise.

The front of the AST contains the token nodes in their original order. Since all whitespace information is contained in the tokens, and punctuation tokens are not 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.

With an additional layer between the lexer and the parser, it is possible to handle preprocessor constructs such as include files and macros (even ones that cross-cut the syntax). Additional information relating to such preprocessor constructs can be stored in the graph as well. During reconstruction, finding

a node that originates from such a construct does not pose a challenge, as these constructs mostly involve directly storing all of their relevant tokens.

```
{absence, "end", token,
    [{token, "fun"}, {repeat, ";", "exprcl"}, {token, "end"}],
    [{token, "fun"}, {symbol, "sub"}]
}
```

FIGURE 6. The skeleton description of a fun expression.

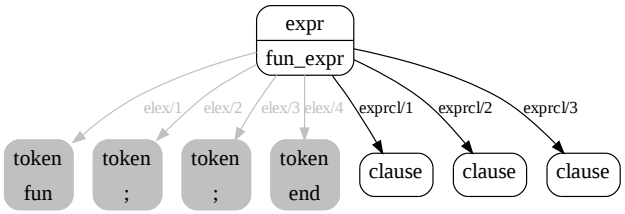


FIGURE 7. Graph representation of a function expression with three clauses. One instance of such a function is the following.

```
fun
(X) when X > 0 -> ok, X;
(X) when X < 0 -> ok, -X;
(0) -> error
end
```

3.5. **Example.** Figure 7 shows a fun expression with three subclauses in the graph representation. The nodes representing the clauses are not in order, as ordering exists only within the `elex` and the `exprcl` edge classes. Let us use the algorithm described in Section 3.4 in order to recover the order of the nodes.

The description of the fun expression in 6 contains a skeleton construct. In order to eliminate it, we have to check whether the actual structure contains an `end` token. It does, therefore the first of the two descriptions is chosen. This description starts with a `token`, therefore the first element in the order is the first token that is connected to the `expr` parent node. The label of the connecting edge is determined by class of the parent node, `expr`: `elex`. Thus, the first child node in order is the one connected by `elex/1`. Next

in the description is a repeat construct with the `exprcl` symbol link and the semicolon. For this, we take all three nodes that are linked by `exprcl`, and one less token (linked, as before, by `elex/1`). The restored order is the symbols with the tokens intercalated between them. The last element of the description is another token, for which we take the last remaining token. Since all the description and the actual nodes are consumed, the representation is syntactically valid. The restored order can be seen in Figure 8; restoration is continued for all child nodes.

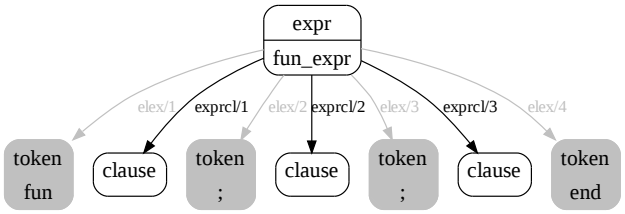


FIGURE 8. Graph representation of a function expression with three clauses.

4. Related work

The design of the representation was shaped through years of experimentation and experience with refactoring functional programs. The first refactoring tools produced at Eötvös Loránd University [5, 7] used standard ASTs for representing the syntax. It became evident that such a representation is not convenient enough for refactoring purposes, and a new design was needed. The resulting design [4] used the contracted graph described in section 2 as representation of the syntax tree, but it relied on superfluous `next` edges to maintain the order of tokens. Section 3.1 argues why having these was undesirable, and the whole of section 3 describes the new structures and algorithms that were necessary to avoid them.

The Java language tools srcML [8], JavaML [3] and JaML [1] use XML to model Java source code. Since XML naturally outlines a tree structure, these representations conserve node order, which enables them to easily reprint the source.

Since the representation outlined in this paper differs so much from the usual approach taken – using a contracted representation instead of the more conventional ASTs – the problem of reproducing the original nodes in order does not appear in other works, as this task is trivial when using an AST.

References

[1] G. Fischer, J. Lusiardi, J. Wolff v. Gudenberg, *Abstract syntax trees and their role in model driven software development,* in Proceedings of the International Conference on Software Engineering Advances, IEEE Computer Society (2007), page 38.

[2] J. Overbey, R. Johnson, *Generating Rewritable Abstract Syntax Trees,* in Proceedings of the 1st International Conference on Software Language Engineering (SLE 2008), Toulouse, France, 2008.

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

[4] R. Kitlei, L. Lövei, T. Nagy, Z. Horváth, T. Kozsik, *Preprocessor and whitespace-aware toolset for Erlang source code manipulation,* in Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, Hatfield, UK, 2008.

[5] R. Szabó-Nacsa, P. Diviánszky, Z. Horváth, *Prototype environment for refactoring Clean programs,* in Proceedings of the 4th Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, 2004.

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

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

[8] J. I. Maletic, M. L. Collard, A. Marcus, *Source code files as structured documents,* in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), IEEE Computer Society, Washington, DC, USA, 2002, pp. 289–292.

Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/c, H–1117 Budapest, Hungary

E-mail address: kitlei@elte.hu