Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 401–408.

Detaching and Reconstructing the Documentary Structure of Source Code*

Attila Góbi, Andrea Kovács, Dániel Leskó, Mónika Mészáros

Dept. Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
{gobi, koaqabi, lesda, bonnie}@inf.elte.hu

Abstract

Formattings, comments and whitespaces in source code, ie. the documentary structure of the source code should be preserved during program transformations if the produced source is for human consumption. However, program transformations are easier to be described if the documentary structure may be discarded.

This paper presents a solution that the documentary structure is detached before the transformations and it is re-attached after the transformations. The detached information is stored in three layers: token formattings, comments, and layout, what we call layered representation. This representation is also suitable for analysing and refactoring the documentary structure of the source code. This model is worked out for the F# programming language, and it has been implemented in, too. The proposed approach can be adapted to other programming languages.

Keywords: Functional programming, Refactoring, Pretty printing

MSC: 68N15 Programming languages

1. Introduction

Source code is primarily created for humans to read, and not for machines to compile, it is a way of communication between programmers [1, 2]. For humans, a source code is a document written in a formal language, though formal languages and syntax trees built from these well-formed documents are usually unable to describe the document's structure completely.

The following information is not present in a typical syntax tree: comments, indentation, formattings (different notations of the same token). This information will be referred as *documentary structure* [8]. Documentary structure should be

*Supported by Morgan Stanley, KMOP-1.1.2-08/1-2008-0002

preserved during program transformations if the produced source is for human consumption.

The typical approach for handling documentary structure regarding program transformations is to store it in the syntax tree [10, 9, 7, 6, 3] or besides the syntax tree [4]. This has the drawback that the storage and processing of program code needs more effort.

In this paper, a new approach for handling the documentary structure is presented. This approach preserves the whole documentary structure while it makes possible for a program transformation step to work on only a suitable representation of the source code without unnecessary information.

In Section 2, the concept is introduced, then in Section 3 the method of detachment is described in details, and Section 4 is about the synthesis process.

2. Model

Our basic idea was to detach that information from the abstract representation of a program, which had not any meaning at that specific abstraction level. It turned out that these information appear on at least two abstraction levels. To be more precise, most of these information come up when we step up to higher abstraction.

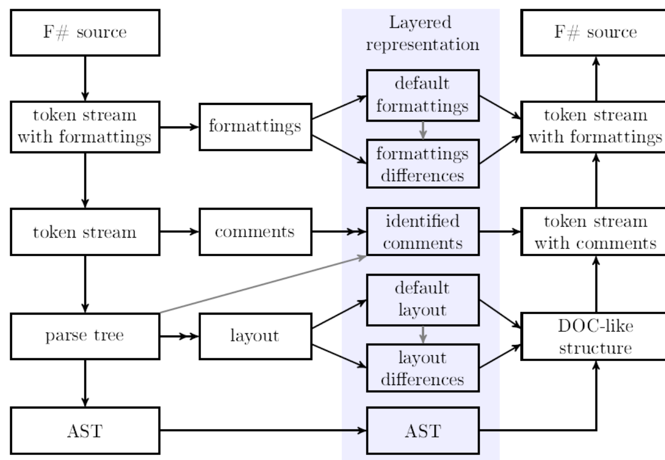

Figure 1: Data flow diagram of our model

The figure 1. is the data flow model of our approach. The arrows are converter functions, which work in the shown direction. The concept is lossless, so when more than one arrow comes out from a box, it means that the information is split into disjunct parts. The double headed arrow means that function uses heuristics to find out the programmers intentions. The grey coloured arrows are not real data streams, they just show the use of the data.

The model is layered vertically and horizontally, too. The first vertical layer (on the left hand side of the figure) is the analysis, which starts with an almost typical lexical analysis, except that we tokenize the content of the comments. The next step is grouping the content of the comments into one multi-token. After that comes the syntactical analysis, which produces an extended abstract syntax tree, we call it *parse tree*. It is extended, because it store a lot of position information. And finally the *AST* is a typical abstract syntax tree.

The detached information makes the second layer. The third layer contains all the information of the analysed program in an analysed, compressed and layered way. This kind of program representation is our main goal. Section 3. tells us more about the process of detaching and compressing information.

The last vertical layer is the synthesis, which is where we reattach the layers, to get back a textual representation of the analysed program. The details of this method are in section 4.

There are also 5 horizontal layers, with 3 different abstraction levels (top is the lowest, bottom is the highest). The first layer (top of the figure) is the layer of the source code. The next two layers are in token stream level, and the final two is in abstract syntax tree. At the fifth layer there is no detached information, just the real refactoring steps (if there are), which modifies the *AST*. It has a usage of this model, where the refactoring step is an identical function (for example transform the source code according to a selected coding convention).

3. Detached information

During the analysis of the source code we detach information at more than one level, because of the layered architecture (figure 1.). In this section we will discuss about the three main sources, where we have such extra information, which has not an influence on higher analysis levels, but we need them, to be able to do the synthesis.

3.1. Formattings

Detaching It is common that during a normal (e.g. used by compilers) lexical analysis more than one kind of literals are mapped to the same token. In most of the cases, these differences appear in the stored values of the tokens, but sometimes they do not, because they are not needed at higher levels. Information that affects the uniformity of the source code are formattings. Formattings information may not change the behaviour of the compiled program. For example this kind of information is the scale of a number or a representation style of a character. The data types are language dependent. These types are defined by designer.

Example (␣x␣␣+␣0xFF␣)*␣␣0X00ff
In this example, we save formatting information only for lexeme `0xFF` and lexeme `0X00ff`. The token stores information, that it is a hexadecimal number with value

255, so both lexeme results in the same token. The formattings store, that the `'x'` and the digits of the hexadecimal numbers are lower- or uppercase.

Compressing When we detach the information, every single piece of information is stored. We examine the source code, determine the most frequent value for all types of information, and store it as *default formattings.* After that we have the *default* values, we can use them to compress the formattings, by storing only the *differences* in reference to the *default* and not the whole data. The result of this method is the *formatting differences.* With this we not only compress the data, but the *defaults* make a metrics of the source code, which creates the possibility for a syntactical refactoring step (details in section 4.).

3.2. Comments

Detaching The detachment of the comments is in the third horizontal layer, which is a thin layer, because there is no abstraction level change between that and the previous one. The comments are handled as a part of the documentary structure, because we believe that their purposes are to have more readable and understandable programs. And if we accept that, then the comments are as important part of the documentary structure, as the whitespaces, or the token *formattings.*

Processing Instead of compressing the comments (which has no meaning), we are using heuristics to recognize the programmers' intention. The heuristics consider the position and the size of the particular comment, and examine some surrounding language constructs to figure out where that comment belongs to, it is the grey arrow on our model from parse tree to identified comments. The results of this analysis are called *identified comments.*

With this information it is manageable that, for example, when a refactoring step moves a function, it can carry the *identified comments*, too.

3.3. Layout

Detaching After the syntactical analysis we get a *parse tree*, which has a very complex structure, because it shows us not just the physical arrangement, but carries a lot of information about the meaning of the program. The problem is the variety of the types; too much information is stored by the types and by the names of the data constructors. We need all this information, though, to be able to recognize a *default layout* for every language construct. In order to reduce the complexity the *parse tree* is converted to a simpler data structure, which contains all the information needed to detach the layout. So we designed the *DOC structure*, which has only 6 simple data constructors. The DOC data structure represents the physical arrangement of the code. The information that was carried by the names of the types and by data constructors are stored in DOC too. We will refer to this information as *labels.*

```
DOC =    | Token(token, position)
         | Pair(DOC, DOC, labels)
         | SepPair(DOC, token, DOC, labels)
         | Block(token, DOC, token, labels)
         | List(DOC list, labels)
         | SepList(DOC list, token list, labels)
```

The base of the DOC is a token with its absolute position in the source code. There are 5 recursive constructs to describe if two DOC is next to each other (`Pair`), or almost neighbouring, but there is a simple token (we call them separator) between them (e.g. semicolon or an equal sign), in this case we use `SepPair`. The third case is if the DOC is surrounded by two separators (`Block`). The last two constructors are about DOC lists, where the elements are next to each other (`List`) or when there is a separator between the elements (`SepList`).

From DOC structure we can effectively detach a complex system of relative positions, and we call it *layout*. The basic idea is to convert the absolute positions into relatives, and say that something like these: "these two DOC is in the same line, and there is 6 whitespaces between them, or the second is in a newline, and indented with 3 more columns, as the first DOC element". Obviously we store one relative position for `Pair` and two for `SepPair` and `Block`. We handle lists as a set of `Pairs` or `SepPairs`, after the relative positions are calculated, we find the most frequent of them (it's a kind of a local default), and compress the others to be relative to the previously mentioned local default.

The *parse tree* ⇒ DOC conversion is a designer defined relation, which is language dependent, for one piece of *parse tree* we could create a thousand different DOC representation. We designed the conversion to group those tokens and language constructs into DOC elements and subgraphs, which physical arrangements behave alike, or close to it. In other words, if we have a lucidly formatted source code, where the *default layout* and *layout differences* (see next paragraph) are already recognized, then the differences should be small or even empty.

Example (␣x␣␣+␣0xFF␣)*␣␣0X00ff
The ‘x␣␣+␣0xFF‘ expression is a *SepPair* with `+` token, it stores the space before and after the separator. The ‘(␣x␣␣+␣0xFF␣)‘ expression is a *Block*, it stores the space after the opening parenthesis, and before the closing parenthesis. The ‘(␣x␣␣+␣0xFF␣)*␣␣0X00ff‘ expression is a *SepPair* with `*` token. The *parse tree* ⇒ DOC conversion is defined by the designer.

Compressing The principle is pretty much the same as in 3.1. subsection, except that the distinctions of the different kind of information are not only based on the type system, because the DOC structure is too simple for that. We use the *labels* to differentiate the language constructs.

The naming, of course is different, too. We create the *default layout*, and the *layout differences* in this phase.

4. Usage of layered representation

The layered representation is useless, if we can not synthetize the source code from it. In the following, we shortly describe a method which does the job.

We start from AST which has the same structure as *parse tree* apart from the absolute positions. Firstly, we convert the AST into a DOC-like structure (the structure is the same, but it does not store positions at all) by using the exact same principles what we used during the *parse tree* $\Rightarrow$ DOC conversion. With this DOC, the *default layout*, and *layout difference* we can restore a token stream which contains the same type and amount of whitespace and newline tokens as the *layout* says. The *identified comments* can be inserted into the token stream where they belong based on the results of our comment alignment heuristics. After this, we just reattach the *formattings* information to the tokens.

At this point, all of the information is in the token stream, or attached to it. The last phase is to create the textual representation of the program.

Note that we use unique identifiers in every representation of the programs, and in the detached information, too. These identifiers are managed during the analysis and synthesis, and they create the ability to say something like this: "... and now we reattach this layer to the main representation". Based on unique identifiers we know, where each detached information belongs to, if there was no refactor steps around, otherwise we use the default values.

If there are no refactoring steps, then this model is lossless, and we can produce exactly the same source code as the original. Note that the model gives us the opportunity of a syntactical refactoring step, if we just forget about the *differences*, and use only the *defaults* to synthetize the source code. With this, we standardized our program.

5. Future Plans

After we designed and implemented this framework for refactoring, our main plan is to implement some real refactoring steps, and to show that it is much easier to do in this way.

During the design and implementation we found secondary goals which come from the storage structure of the detached and compressed information. As we described in 3.1. and 3.3. subsections, we always calculate *default* information, and store the *differences* relatively to it. With this, we can create a subversion controller plug-in, which controls the *documentary structure* of the source code. The repository would have the conventional *default formatting* and *layout*, but every user could use their own *default* settings, when they check out.

At this time, the comment analysis is based on the positions and sizes, of the comment, and the surrounding other constructs. We want to enhance the heuristics by interpreting the comments.

It would be impressive if we could recognize that some construct/code fragment are in the same column with a reason (e.g. it's a convention, or the programmer thinks, that it increases the readability of the code), and not accidentally.

6. Related Works

We found that the paper [7] is the closest to the idea we introduced here, it tries to solve the same problem. A significant difference that it does not provide any implementation, it just theorizes and speculates on the topic. In the contrast to this, we provide a concrete solution based on a layered representation approach, give an excerpt for its specification by discussing many questions.

In [4] a Haskell refactoring tool, called HaRe is described. HaRe also preserves the layout and the comments. However, a major difference to our model is that program transformation steps implemented in HaRe process the token stream and the abstract syntax tree in parallel, and it makes harder to add new transformations.

[6] discusses a refactoring tool for the programming language Smalltalk, which was the first of the most known ones. It does not take layout into account, but it is acceptable for Smalltalk as comments are parts of a method; therefore, they are parts of the syntax tree.

Opdyke's PhD thesis [5] defines refactor steps for OO languages. It is commonly referenced and it is a comprehensive work on the topic; however, it was made popular by [1]. It does not discuss the layout.

[3] analyzes the problem of handling macros, comments, and layout in Erlang refactoring. Like in our paper, it is also important for the authors to preserve the visual representation of the source code to be transformed, but they choose different principles for the implementation. Layout is encoded on the token stream's level as follows (subsection 4.1): every token has a pre and post white space field, and it can contain comments too. An "ad hoc" algorithm divides the space between tokens into two segments. In our opinion, assigning comments to tokens is not suitable for the optimal behaviour of program transformations. For this reason, we assign comments to syntactical constructs.

Another paper [10] discusses preservation of style for C/C++ programs, and handling of macro expansions. One of its goals is to minimize the "noise" created by the program transformations in different version control systems. It also emphasizes the importance of layout preservation. A notable difference from our approach, that the documentary structure is stored in the abstract syntax tree directly (it is called LL-AST for Literal-Layout AST), but, due to this, the complexity of the syntactical analysis is increased. A similar approach can be found in [9].

7. Conclusion

We proposed a flexible solution for handling the low-level structure of the source code for refactoring and code analyzer tools. The solution was designed for the F# programming language, but during the model designing it was always a point, that the model should be general enough, to be able to fit for other languages. Operability of the approach is also supported by an implementation written in F# (release coming soon at `http://fsharp.inf.elte.hu/`).

To our knowledge, layered representation of the source code has not been proposed so far in the literature.

References

[1] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[2] A. Goldberg. Programmer as reader. *IEEE Software*, 4(5):62–70, 1987.

[3] R. Kitlei, L. Lövei, T. Nagy, and Z. Horváth. Layout preserving, automatically generated parser for Erlang refactoring.

[4] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38. ACM New York, NY, USA, 2003.

[5] W. Opdyke, D. of Computer Science, and U. of Illinois at Urbana-Champaign. *Refactoring object-oriented frameworks.* University of Illinois at Urbana-Champaign, 1992.

[6] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object systems*, 3(4):253–263, 1997.

[7] M. Van De Vanter. Preserving the documentary structure of source code in language-based transformation tools. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 131–141, 2001.

[8] M. Van De Vanter. The documentary structure of source code. *Information and software technology*, 44(13):767–782, 2002.

[9] M. van den Brand and J. Vinju. Rewriting with layout. In *Proceedings of RULE2000*, 2000.

[10] D. Waddington and B. Yao. High-fidelity C/C++ code transformation. *Science of Computer Programming*, 68(2):64–78, 2007.

A. Góbi, A. Kovács, D. Leskó, M. Mészáros
Eötvös Loránd Tudományegyetem, Programozási Nyelvek és Fordítóprogramok Tanszék
Pázmány Péter sétány 1/C., H-1117 Budapest, Hungary