Chapter 1

Detaching and Reconstructing the Documentary Structure of Source Code

Péter Diviánszky, Attila Góbi, Dániel Leskó, Mónika Mészáros, [1]
Gábor Páli [2]
Category: Research [3]

Abstract: Comments and white space 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, which 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 F# too. The proposed approach can be adapted to other programming languages.

1.1 INTRODUCTION

Source code is primarily created for humans to read, and not for machines to compile, it is way of communication between programmers [2, 3]. For humans, a source code is a document written in a formal language, though formal languages

[1]Eötvös Loránd University, Budapest, Hungary,
{divip,gobi}@aszt.inf.elte.hu, {lesda,bonnie}@inf.elte.hu
[2]Babeş-Bolyai University, Cluj Napoca, Romania; Eötvös Loránd University, Budapest, Hungary pgj@inf.elte.hu
[3]Supported by ELTE IKKK, Morgan Stanley, POSDRU/6/1.5/S/3/2008

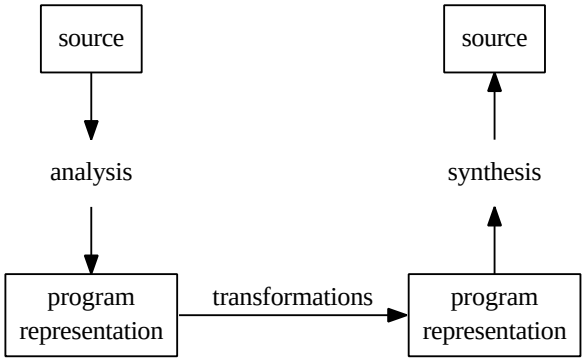

FIGURE 1.1. Typical source-to-source transformation

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, layout (where the lines break), exact number and type of whitespace characters between tokens. This information will be referred as *documentary structure* [10]. Documentary structure should be 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 the documentary structure in the syntax tree [12, 11, 9, 8, 5] or besides the syntax tree [6]. 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 tries to preserve 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 1.2, the fundamental concept is introduced, then in Section 1.3, it is extended to the F# [1] programming language. A short excerpt of the model specification can be found in Section 1.4. In Section 1.5, an F# prototype implementation is presented for the proposed model.

1.2 LAYERS OF ABSTRACTION

Most of the program transformation tools process the source by analyzing it, then synthesize the results.

A common source-to-source transformation scheme is presented on Fig. 1.1. Note that unlike in compilers, documentary structure must be taken with extra care, hence it is usually required to build dedicated lexical, syntactical and semantical analyzers and store the documentary structure in the representation of the program.

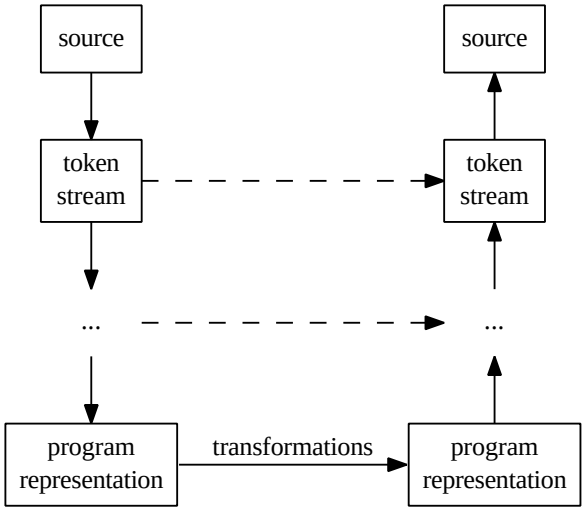

FIGURE 1.2. Layered source-to-source transformation

This model has been modified by us in the following way as Figure 1.2 shows. Documentary structure is detached from and re-attached to the program code in several steps. The detached information is represented by dashed lines. Left side dots denote the usual phases of source code analysis like lexical analysis, syntactical analysis and other semantical analysis steps as necessary. Rows of the diagram are the representation *layers*. Note that downward layers are referred as *higher level* layers.

The main advantage is that the detached information will not trouble the storage and processing of program code in the higher abstraction levels.

1.2.1 Layer Structure

Figure 1.3 contains a general layer structure. The black arrows denote information flow on the figure. Splitting information is denoted by more than one black arrows starting from a common box.

Detached information is always grouped into defaults and differences. Grey arrows define the order of the operations: default values for the actual layer should be determined first in order to derive the differences. Generation of default values and differences to them is explained in 1.4.2 and 1.4.4.

Splitting information into default and difference values is beneficial from several aspects:

- Offers a more suitable data structure for program transformations and analysis (see 1.3.1).
- Default values can be used when synthesizing new elements, for example, to

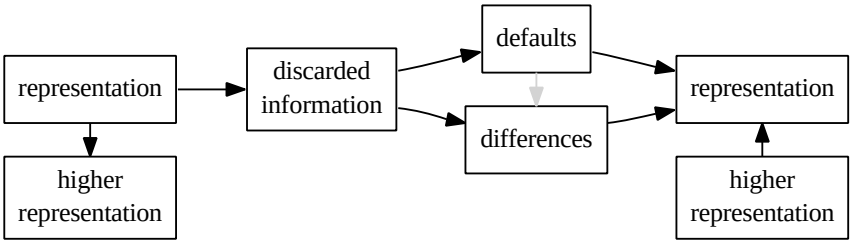

FIGURE 1.3. General layer structure

determine their style. It is a common problem of refactorer programs how to determine the visual representation of the newly generated code snippets.

1.3 APPLYING THE MODEL TO F#

The rest of this paper focuses on applying the proposed model to process programs written in F#. Although this application can be easily adapted to other programming languages without support for macro definitions.

Figure 1.4 shows the detailed model. The meaning of layered representation, double headed arrows and token arrangement are explained in the following subsections.

The detailed model has four layers: formatting, comments, layout and parse tree. Starting from the parse tree layer, it is possible to add other higher abstraction levels (see Section 1.6 for related future work). The paper discusses only the lower layers mentioned above.

1.3.1 Layered Representation

Layered representation of the processed source code consists of the data presented on Figure 1.4. These data will be referred as *components* to the layered representation. Components provide a representation of the original sources, where different program transformations can be performed in a comfortable manner. To demonstrate the uses of these components, some functions are described below:

Formatting differences* and *layout differences Possible refactoring steps on these components are token formatting unification and layout unification within a source file. The implementation is just the deletion of the differences. Of course, partial unification is also possible, for example, when only the formatting of numeric literals is unified.

A possible program analysis on these components is to measure the number of formatting or layout differences. The higher number indicates that the code was mixed from several sources or written by different programmers.

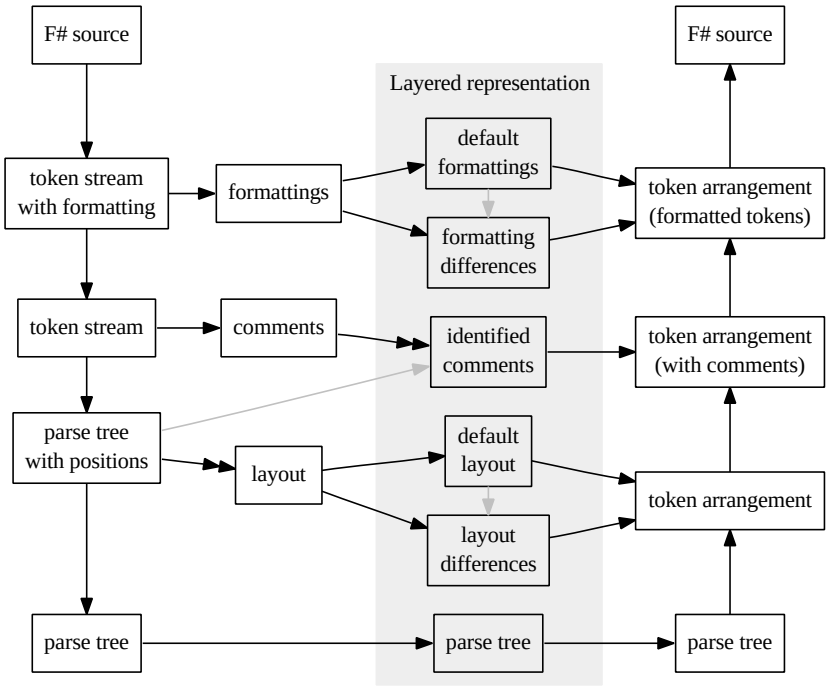

FIGURE 1.4. The detailed model

The programming environment can be extended to generate style warnings from the differences.

Default formatting* and *default layout Possible refactoring steps: Adjust token formatting or layout. For example, one can easily convert all alphabetical digits to lowercase in hexadecimal numeric literals.

Possible program analysis: Find the most common programming style.

These components can also help development. Version control systems are not immune to the extra "noise" cased by different programming styles. This can be cured so that on commit the formatting of tokens and the layout are set to a common style while on checkout the formattings and layout are set to the style of the individual developer.

Parse tree All the classical refactor and program analysis steps are performed on this representation. By splitting the parse tree component further, definitions of these steps could be even more simplified (see Section 1.6 on related future work).

1.3.2 Using Heuristics

On Figure 1.4, double-headed arrows denote injection of additional information. For example, it requires information not present in the original source code to identify which comment can be assigned to which program construct. It is preferred to eliminate the need for human interaction in such decisions, so heuristics are added. The applied heuristics are further described in Section 1.4.4.

1.3.3 Token Arrangement

Token arrangement is a data structure which can be rendered to a token stream given a particular page width. It has the following constructs: horizontal and vertical composition, indentation, and a table construct similar to HTML tables. The possible rendering algorithms are described in [4] and [13].

Suppose that the next code snippet is represented by a table construct in a token arrangement data structure.

```
x < 0.11 -> -x
_        ->  x
```

Suppose that the formattings of fraction literals are set to scientific style. This change in formatting result in the next code after rendering:

```
x < 1.1e-1 -> -x
_          ->  x
```

1.4 SPECIFICATION OF THE DETAILED MODEL

In this section, layers from the lower to the higher levels are discussed based on Figure 1.4.

1.4.1 F# Source

At the moment F# source is a single F# source program file, a unit of compilation. An F# source may use the light syntax option.

1.4.2 Formatting Layer

In the formatting layer, token formatting information is detached.

Formattings

Tokens can be arranged in several classes according to their formatting. This classification results in the following classes: space, space at end of line, line break, character, string, integer numbers, fractional numbers, light syntax token, and other tokens. Not every formatting for every token class is discussed here, only examples are mentioned.

Formatting for *spaces* stores the information where a tabulator character is. Number of spaces is not stored among the formatting but in the token itself as this information will be detached in the layout layer.

Formatting for *spaces at the end of line* stores the number of spaces and where the tabulator characters are. These tokens will be discarded from token stream for the next layer.

Formatting for *line breaks* can have three values: Windows (CR+LF), Unix (LF), Macintosh (CR). Because of this, there is only one constructor for new line tokens. In general, it can be noted that a token and its formatting are complementers. Formally, for arbitrary formatting for arbitrary token, there is a source code where the given token appears with the given formatting.

One formatting information for *characters* is its source code representation: simple (`'a'`), trigraph format (`'\097'`), or Unicode short format (`'\u0061'`). For example, the token of the lexeme `'\097'` is `Char 'a'`, and its formatting is `Trigraph`. Note that the token itself does not contain any information on formatting.

Formatting for *integers* consists of several attributes, like the number of leading zeros, and the cases of hexadecimal digits. For example, the token of lexeme `0x0000fFfu` has the following attributes: the number itself (65535), its representation (a 32-bit unsigned integer, it is flagged by the `u` at the end), and that it was given in hexadecimal from. Formatting information for this lexeme are: it contains four leading zeros, and that hexadecimal digits are mostly in lowercase, but the sixth digit is written in uppercase.

Numeral system information is not part of the integer formatting, since many good quality sources mix hexadecimal and decimal integer literals on purpose. (It was a design decision that there should be no formatting differences in case of good quality source code.)

Other tokens have no formatting information.

Token Stream with Formatting

Token stream contains position, token and formatting triples. Comments and white space are also tokens. Neighboring spaces and tabulators are merged into a token, but line breaks are separate tokens. Every other token is the same as in case of a lexical analysis: literals, identifiers, keywords.

Default Formatting and Formatting Differences

Formatting attributes are divided into two logical groups whether they have a global default value or not. For example, the following attributes has global default values:

- Number of leading zeros (defaulted to 0).
- Formatting of space characters (defaulted to contain no tabulators).

The following attributes has no global default values. Default values are calculated as the most frequent value in the domain range:

- Formatting of new lines (possible values: CR+LF, LF, CR).
- Formatting of `ExtraExponentSign`. In scientific notation, exponent signs might be omitted for literals in floating point representation, and the formatting will store whether they are given or not.

Formatting differences store the differences to the default values, independently of whether they are calculated or predefined. To illustrate this, let us take the previously discussed lexeme `0x0000fFffu`. Assume that the source code contains lowercase hexadecimal integer literals in general. The attributes stored in formatting differences are as follows: four leading zeros, the sixth digit is written in uppercase. Note that the attribute on hexadecimal digits are in lowercase is not present here as it is a default.

Token Arrangement with Formatted Tokens

The generation of this data structure is as follows: The individual tokens in the token arrangement (received from higher level) are searched for the accompanied formatting in the list of formatting differences based on the old position for each non-modified token. If there is no formatting information found, then the token will have a default formatting. Note that this means that elements generated or changed during a refactoring step will be formatted by the defaults.

Example: The Formatting Layer in Action

The most difficult is the reconstruction of space formatting, because space tokens are discarded from the token stream in the higher layers. Application of white space formatting is controlled by the original and modified token positions.

Consider the next code segment (_ denotes a space; `\t` denotes a tabulator character).

```
four_\t=_3_+_1
y_=_4____
z_=_5
```

Data structure for formatting differences will contain the following components:

- Spaces in line 1 column 4–8 has a tabulator in columns 5–8.
- There are spaces at the end of line 2, columns 8–12.

These components are discarded in the next layer:

```
four    = 3 + 1
y = 4
z = 5
```

Note that the next layer still keep the position information (like `four` is in line 1, columns 0–4).

Suppose that after some transformation steps (moving the first line down and increasing the indentation) the synthesized token stream (in the next layer):

```
y = 16
z = 5
four    = 3 + 1
```

The synthesis of the token stream in the formatting layer is as follows.

Go through the token stream one by one. The old position of tokens (unique identifiers) are supposed to be known in advance because it is added to the tokens as an attribute.

Old position for token `4` is line 2, columns 4–5. Search the formatting differences data for line 2, column 5 and it will be found that there are spaces at the end of line 2, at columns 5–9. Now check that token `4` is still at the end of line in the transformed code, and insert 4 spaces after it. (If the check has failed, formatting information should be discarded.)

Old position for token `four` is line 1, columns 0–4. Search the formatting differences for line 1, column 4, and it will be found that spaces at line 1, columns 4–8 have a tabulator characters in columns 5–8. Now check that whether the next token in the transformed code the token `=` has old start position of line 1, column 8. It is true, so insert a space between `four` and token `=` with a similar formatting.

The final result is:

```
____y_=_4____
____z_=_5
____four_\t=_3_+_1
```

1.4.3 Comment Layer

Comments are detached in the comment layer.

Token Stream

Token stream consists of pairs of tokens and positions, therefore it does not provide information on token formatting. This token stream also does not contain any whitespace token.

Comments

Comments are stored in a token as a string in verbatim format.

Identified Comments

Position range information is added to abstract syntax tree, and it is capable of identifying program constructs. Data structure for identified comments is composed of such identifiers, relative positions, and comments. An identifier denote

the syntactical construct (a part of source code) where the given comment possibly belongs to. This assignment is guided by heuristics, see Section 1.4.4. A relative position gives the relative position of the given comment to the assigned construct.

Token Arrangement with Comments

The creation process of this data structure is as follows: The input token arrangement comes from a higher layer, and it does not contain comments. For each token in the input arrangement, search for the comments assigned to the old position of the token. If there is a comment found for a token, then find a place to insert the comment according to its relative position, and insert it into the arrangement.

1.4.4 Layout Layer

Source code layout is detached in the layout layer.

The #light declaration makes white space significant in F# and instructs the F# compiler and interpreter to use the indentation of F# code to determine where constructs start and finish. This feature is similar in spirit to the use of indentation by Python and Haskell.

The light syntax force the programmer to use specific code layout, however layout is not completely determined by the light syntax. For example, indentation for a complete code block might be changed in a sensible manner, but lines within a block should not be indented differently.

Parse Tree with Positions

A special parse tree is built for this layer. It is very similar to the parse trees employed by compilers, however there are some notable differences, so the constructed syntax tree:

- Omits abstractions as much as possible, i.e. it tries to follow the concrete structure of the source code.
- Contains one or more positions for every construct, i.e. it is full of positions.

One must be able to restore the original token stream together with the original positions by a proper traversal of the tree.

Resolution of the light syntax option is performed directly before the parse tree generation. Extra tokens added during this phase are labeled to ensure their correct processing.

Layout

Data structure for layout stores the concrete indentation structure of the source code. Here, layout is considered to be all possible indentations for the program

what does not change the semantics. (In case of light syntax this is a real constraint.)

One must be able to restore this type of information from the parse tree by deleting all the position information.

Description of the layout data structure is omitted because of space limitation of the paper. However, some of its important elements are highlighted by the sample source code shown on the next code:

```
let rec length l =
  match   l with
      []      ->  0

      | x::xs ->
         1 + length xs
```

Data structure for layout is designed to catch all the original intentions of the source code author as much as possible. For example, based on a deeper analysis of the previous code, it is expected that all the patterns, the right arrows, and the expressions following the arrows in the `match` construct are indented to be in the same column.

Layout information is assigned to syntactical constructs. Layout for each construct stores information only about the indentation between its starting and ending positions. Locations for separate syntactical constructs are controlled by information stored in the smallest enclosing construct (when needed).

```
[ [1;  3; 5]
; [4;  8; 12]
; [14; 6; 8]
]
```

Layout information describing that the columns of the matrix start in the same column, is stored in the outer list's layout for this code snippet.

Determining the layout requires many additional information to be added to find out what the author's original intention was. It is presented in a form of different heuristics, integrated into the implementation in advance.

Construction of the layout is as follows: Traverse the parse tree in a top-down, left-right style. For each node heuristics are invoked according to the actual syntactical construct, and it generates the corresponding layout information. An advantage of this approach that different heuristics might be invoked for different programming constructs, implementors are not forced to use a monolithic one.

Finally, description of some sample layout data structures for the F# programming language are included. The next type definitions shows the common types used in the implementation. The values of `Floating` hold the distance between tokens in the same line.

```
type BlankLines = BlankLines of int
type IndentedBy = IndentedBy of int
```

```
type Floating   = Floating of int
type Index      = Index of int
```

Type `RelativeIdentation` has two data constructors. Data constructor `InNewLine` represents the distance between two neighboring tokens when they are in two different lines physically. Their distance is measured in characters. It stores the number of blank lines between the tokens, and a difference for their effective indentation in the line. This data structure is ideal for describing subconstructs in a syntactical construct.

```
type RelativeIndentation =
    | InNewLine of BlankLines * IndentedBy
    | InSameLine of Floating
```

Type `RelativeIndentations` represents a data structure for storing locations for more complicated programming constructs with more components, e.g. relative positions for elements of a list. Information is stored in form of defaults (`Uniform`) and differences to them (`RelativeIndentation`).

```
type RelativeIndentations =
    | RelativeIndentations of
        Uniform  * (Index * RelativeIndentation) list
```

Type `Uniform` stores the default relative locations of more elements. Data constructor `InTheSameColumn` used only when syntactical units are in the same column, and it stores the minimal distance measured from the previous construct. Data constructor `SameIndentedBy` stores the number of blank lines between the elements, and the indentation relative to a position of the enclosing syntactical construct. Data constructor `SameFloating` stores the distance information between the elements in the same line.

```
type Uniform =
    | InTheSameColumn of Floating
    | SameIndentedBy of BlankLines * IndentedBy
    | SameFloating of Floating
```

Default Layout and Layout Differences

A default layout for each type of the syntactical constructs is derived from the instances found in the source code.

Token Arrangement

This data structure is constructed by traversing the parse tree while applying the default layout and layout differences on the language constructs. As before, the old position helps to identify the corresponding parts.

1.4.5 Parse Tree Layer

Currently no further information is detached from the parse tree so this layer has a simple structure. The two arrows on the bottom of Figure 1.4 denote identity functions.

1.5 IMPLEMENTATION

The specified system was implemented in F#. We used F# as a functional language with rare exceptions. Arrows on Figure 1.4 are implemented as functions. The "discard comments" and "discard whitespace" functions was implemented as applications of the well-known `filter` function, while others are applications of `map` etc.

The lexical analyzer was generated by the `fslex` tool, which is similar to `ocamllex` and `lex`. The source for `fslex` is partially based on the source code of F# compiler's lexical routines, and it has been modified to suit the needs. Notable differences:

- Representation for numeric literals is more structural thus it is more suitable for program analysis.
- The lexer also emits token formatting beside tokens.

The syntactical analyzer was generated by the `fsyacc` tool which is similar to `ocamlyacc` and `yacc`. The grammar for `fsyacc` is based on the original F# grammar, and at the moment there are no major modifications compared to that. The major difference is that the output of the parser is a concrete syntax tree (parse tree), not an abstract syntax tree.

There are some minor differences from the specification in the current implementation. In the specification, it has been mentioned that tokens can be arranged in several classes according to their formatting (see Section 1.4.2). In the implementation, there is only one formatting class at the moment, but it is planned to classify token formatting here as well. The "token arrangement" data structure is not yet implemented, token stream is used instead.

1.5.1 Heuristics

Implementation of the used heuristics can be refined as needed. To get a grasp of the capabilities of the current implementation, consider the following code segment:

```
(* Main comment *)
// comment_1
open System; open System
// comment_2
// comment_3
let a =
```

```
   let b = 5
   b
let c = 10
```

In the example the multi line comment has been assigned to all syntactical constructs on the top level. The one line comment `comment_1` has been assigned only to the two `open` constructs. Directly connected one line comments (i.e. no blank lines between them) are merged, because it is assumed that it was the original intention. Hence `comment_2` and `comment_3` have been assigned to the `let a` and `let c` constructs. Note that if the `let c` expression would have been also indented, then it had no comments assigned.

1.6 FUTURE PLAN

It is possible to define abstraction levels higher than the parse tree. To be more specific, we plan to add scope analysis, which would detach the entity name information (variable, function, type and module names) from the parse tree.

It is planned to extend the model to deal with more files files, Visual Studio projects or solutions.

Undoing may be managed by some of the programming environments (for example Wisual Studio). However, much faster undoing can be achieved by a built-in method which stores the source code in its layered representation. Storing the source in the layered representation also saves the cost of re-analysing the source after program transformation steps. To implement these ideas is future work.

As an other future work, it is planned to parse these comment tokens to determine whether they include any code snippets by using heuristics. This would make possible for refactoring steps to transform source code segments even in comments, e.g. in case of renaming variables.

1.7 RELATED WORKS

We found that the paper [9] is closest to idea we introduced here, it tries to solve the very same problem. In this paper “documentary structure” denotes exactly the same information (indentation, comments, line breaks, and extra spaces). Comments are also assigned to the abstract syntax tree nodes. It recognizes the importance of “vertical alignment”. However, there are some important differences: it distinguishes important and not important structures among the possible ones. It also does not feel so important to move comments together with the assigned constructs, because it has to be revisited after each refactor step anyway. 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, and we already have a working implementation.

In [6] a Haskell refactoring tool, called HaRe is described. HaRe also preserves the layout and the comments. Although, a major difference to our model 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.

[8] discusses a refactoring tool for the programming language Smalltalk, what 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 [7] defines refactor steps for OO languages. It is commonly referenced and it is comprehensive work on the topic, however it was made popular by [2]. It does not discuss the layout.

[5] 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 behavior of program transformations. For this reason, we assign comments to syntactical constructs.

Another paper [12] discusses preservation of style for C/C++ programs, and handling of macro expansions. One of its goals 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. On the other hand, the LL-AST is unable to describe that two tokens are in the same column. A similar approach can be found in [11].

1.8 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 explained for the F# programming language, but the proposed approach can be adapted to any other programming language if it does not support macro expansions. Operability of the approach is also supported by an implementation written in F#.

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

REFERENCES

[1] The F# 1.9.6 Draft Language Specification, 09 2008. Microsoft Research and the Microsoft Developer Division, http://research.microsoft.com/en-us/um/cambridge

/projects/fsharp/manual/spec.pdf.

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

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

[4] J. Hughes. The design of a pretty-printing library. *Lecture Notes in Computer Science*, 925:53–96, 1995.

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

[6] 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.

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

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

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

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

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

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

[13] P. Wadler. A prettier printer. *Unpublished manuscript*, 1998.