An Algorithm for Layout Preservation in Refactoring Transformations

Maartje de Jonge and Eelco Visser

Dept. of Software Technology, Delft University of Technology, The Netherlands
m.dejonge@tudelft.nl, visser@acm.org

Abstract. Transformations and semantic analysis for source-to-source transformations such as refactorings are most effectively implemented using an abstract representation of the source code. An intrinsic limitation of transformation techniques based on abstract syntax trees is the loss of layout, i.e. comments and whitespace. This is especially relevant in the context of refactorings, which produce source code for human consumption. In this paper, we present an algorithm for fully automatic source code reconstruction for source-to-source transformations. The algorithm preserves the layout and comments of the unaffected parts and reconstructs the indentation of the affected parts, using a set of clearly defined heuristic rules to handle comments.

1 Introduction

The successful development of new languages is currently hindered by the high cost of tool building. Developers are accustomed to the integrated development environments (IDEs) that exist for general purpose languages, and expect the same services for new languages. For the development of Domain Specific Languages (DSLs) this requirement is a particular problem, since these languages are often developed with fewer resources than general purpose languages. Language workbenches aim at reducing that effort by facilitating efficient development of IDE support for software languages [10]. The Spoofax language workbench [12] generates a complete implementation of an editor plugin with common syntactic services based on the syntax definition of a language in SDF [23]. Services that require semantic analysis and/or transformation are implemented in the Stratego transformation language [3]. We are extending Spoofax with a framework for the implementation of refactorings.

Refactorings are transformations applied to the source code of a program. Source code has a formal *linguistic structure* [6] defined by the programming language in which it is written, which includes identifiers, keywords, and lexical tokens. Whitespace and comments form the *documentary structure* [6] of the program that is not formally part of the linguistic structure, but determines the visual appearance of the code, which is essential for readability. A fundamental problem for refactoring tools is the informal connection between linguistic and documentary structure.

Refactorings transform the formal structure of a program and are specified on the abstract syntax tree (AST) representation of the source code, also used in the compiler for the language. Compilers translate source code from a high-level programming language

A. Sloane and U. Aßmann (Eds.): SLE 2011, LNCS 6940, pp. 40–59, 2012.

to a lower level language (e.g., assembly language or machine code), which is intended for consumption by machines. In the context of compilation, the layout of the output is irrelevant. Thus, compiler architectures typically abstract over layout. Comments and whitespace are discarded during parsing and are not stored in the AST.

In the case of refactoring, the result of the transformation is intended for human consumption. Contrary to computers, humans consider comments and layout important for readability. Comments explain the purpose of code fragments in natural language, while indentation visualizes the hierarchical structure of the program. Extra whitespace helps to clarify the connections between code blocks. A refactoring tool that loses all comments and changes the original appearance of the source code completely, is not useful in practice.

The loss of comments and layout is an intrinsic problem of AST-based transformation techniques when they are applied to refactorings. To address the concern of layout preservation, these techniques use layout-sensitive pretty-printing to construct the textual representation [13,14,16,18,20]. Layout is stored in the AST, either in the form of special layout nodes or in the form of tree annotations. After the transformation, the new source code is reconstructed entirely by unparsing (or pretty-printing) of the transformed AST. This approach is promising because it uses language independent techniques. However, preservation of layout is still problematic. Known limitations are imperfections in the whitespace surrounding the affected parts (indentation and inter-token layout), and the handling of comments, which may end up in the wrong locations. The cause of these limitations lies in the orthogonality of the linguistic and documentary structure; projecting documentary structure onto linguistic structure loses crucial information (Van De Vanter [6]).

In this paper, we address the limitations of existing approaches to layout preservation with an approach based on automated text patching. A text patch is an incremental modification of the original text, which can consist of a deletion, insertion or replacement of a text fragment at a given location. The patches are computed automatically by comparing the terms in the transformed tree, with their original term in the tree before the transformation. The changes in the abstract terms are translated to text patches, based on origin tracking information, which relates transformed terms to original terms, and original terms to text positions [22]. A layout adjustment strategy corrects the whitespace at the beginning and end of the changed parts, and migrates comments so that they remain associated with the linguistic structures to which they refer. The layout adjustment strategy uses explicit, separately specified layout handling rules that are language independent. Automated text patching offers more flexibility regarding layout handling compared to the pretty-print approach. At the same time, the layout handling is language generic and fully automatic, allowing the refactoring developer to abstract from layout-specific issues.

The paper provides the following contributions:

- A formal analysis of the layout preservation problem, including correctness and preservation proofs for the reconstruction algorithm;
- A set of clearly defined heuristic rules to determine the connection of layout with the linguistic structure;

– An algorithm that reconstructs the source code when the underlying AST is changed. The algorithm maximally preserves the whitespace and comments of the program text.

We start with a formalization of the problem of layout preservation. Origin tracking is introduced in Section 3. Section 4 explains the basic reconstruction algorithm, refined with layout adjustment and comment heuristics in Section 5. Finally, in Section 6 we report on experimental results.

2 Layout Preservation in Refactoring

Refactorings are behavior-preserving source-to-source transformations with the objective to 'improve the design of existing code' [9]. Although it is possible to refactor manually, tool support reduces evolution costs by automating error-prone and tedious tasks. Refactoring tools automatically apply modifications to the source code, attempting to preserve the original layout, which is not trivial to accomplish.

2.1 Example

We discuss the problems related to layout preservation using an example in WebDSL, a domain specific language for web applications [24]. Extract-entity is a refactoring implemented for WebDSL, Figure 1 shows the textual transformation. The required source code modifications are non trivial. A new entity (`Account`) is created from the selected properties, and inserted after the `User` entity. The selected properties are replaced by a new property that refers to the extracted entity. Comments remain attached to the code structures to which they refer. Thus, the comments in the selected region are moved jointly with the selected properties. Furthermore, the comment `/*Blog Info*/` still precedes the `Blog` entity. The layout of the affected parts is adjusted to conform to the style used in the rest of the file. In particular, indentation and a separating empty line are added to the inserted entity fragment.

```
entity User {
  name : String
  //account info
  pwd : String //6ch
  user : String
  expire : Date
}

/*Blog info*/
entity Blog { ... }
```

```
entity User {
  name : String
  account : Account
  expire : Date
}

entity Account {
  //account info
  pwd : String //6ch
  user : String
}

/*Blog info*/
entity Blog { ... }
```

Fig. 1. Textual transformation

Figure 2 displays the abstract syntax of the program fragment. Abstract syntax trees represent the formal structure of the program, abstracting from comments and layout. Automatic refactorings are typically defined on abstract syntax trees; the structural representation of the program is necessary to reliably perform the analyses and transformations needed for correct application. Moreover, abstracting from the arbitrary layout of the source code simplifies the specification of the refactoring.

```
[Entity(
  "User"
, [ Prop("name", "String")
  , Prop("pwd", "String")
  , Prop("user", "String")
  , Prop("expire", "Date")])
,Entity("Blog", [...])]
```

Fig. 2. Abstract syntax

2.2 Problem Analysis

The refactoring transformation applied to the AST results in a modified abstract syntax tree. The AST modifications must be propagated to the concrete source text in order to restore the consistency between the concrete and abstract representation. Figure 3 illustrates the idea. S and T denote the concrete and the abstract representation of the program, the PARSE function maps the concrete representation into the abstract representation, while TRANSF applies the transformation to the abstract syntax tree. To construct the textual representation of the transformed AST, an UNPARSE function must be implemented that maps abstract terms to strings.

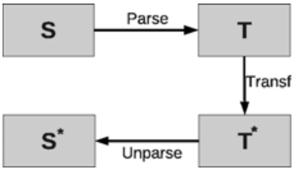

Fig. 3. Unparsing

The PARSE function is surjective, so for each well-formed abstract syntax term t, there exists at least one string that forms a textual representation of t. An UNPARSE function can be defined that constructs such a string [21]. The PARSE function is not injective; strings with the same linguistic structure but different layout are mapped to the same abstract structure, that is $\exists s : \textsc{unparse}(\textsc{parse}(s)) \neq s$. It follows that layout preservation can not be achieved by a function that only takes the abstract syntax as input, without having access to the original source text.

In the context of refactoring, it is required that the layout of the original text is preserved. A text reconstruction function that maps the abstract syntax tree to a concrete representation must take the original text into account to preserve the layout (Figure 4). We define two criteria for text reconstruction:

Correctness. $\textsc{Parse}(\textsc{ConstrText}(\textsc{transf}(\textsc{parse}(s)))) = \textsc{transf}(\textsc{parse}(s))$
Preservation. $\textsc{ConstrText}(\textsc{parse}(s)) = s$

The correctness criterion states that text reconstruction followed by parsing is the identity function on the AST after transformation. The preservation criterion states that parsing followed by text reconstruction returns the original source text. Preservation as defined above only covers the identity transformation. Section 4 gives a more precise criterion that defines preservation in the context of (non-trivial) transformations.

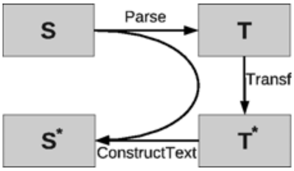

Fig. 4. Text reconstruction

The layout preservation problem falls in the wider category of view update problems. Foster et al. [8] define a semantic framework for the view update problem in the context of tree structured data. They introduce lenses, which are bi-directional tree transformations. In one direction (GET), lenses map a concrete tree into a simplified abstract tree, in the other direction (PUTBACK), they map a modified abstract view, together with the original concrete tree to a correspondingly modified concrete tree. A lens is well-behaved if and only if the GET and PUTBACK functions obey the following laws: $\text{GET}(\text{PUTBACK}(t, s)) = t$ and $\text{PUTBACK}(\text{GET}(s), s) = s$. These laws resemble our correctness and preservation criterion. Indeed, the bi-directional transformation PARSE, CONSTRUCTTEXT forms a well-behaved lens.

3 Origin Tracking

Text reconstruction implements an unparsing strategy by applying patches to the original source code. The technique requires a mechanism to relate nodes in the transformed tree to fragments in the source code. This section describes an infrastructure for preserving origin information. Figure 5 illustrates the internal representation of the source code. The program structure is represented by an abstract syntax tree (AST). Each node in the AST keeps a reference to its leftmost and rightmost token in the token stream, which in turn keep a reference to their start and end offset in the character stream. Epsilon productions are represented by a token for which the start- and end- offset are equal. This architecture makes it possible to locate AST-nodes in the source text and retrieve the corresponding text fragment. The layout structure surrounding the text fragment is accessible via the token stream, which contains layout and comment tokens.

When the AST is transformed during refactoring, location information is automatically preserved through *origin tracking* (Figure 6, dashed line arrows). Origin tracking is a general technique which relates subterms in the resulting tree back to their originating term in the original tree. The rewrite engine takes care of propagating origin information, such that nodes in the new tree point to the node from which they originate. Origin tracking is introduced by Van Deursen et al. in [22], and implemented in Spoofax [12]. We implemented a library for retrieving origin information. The library

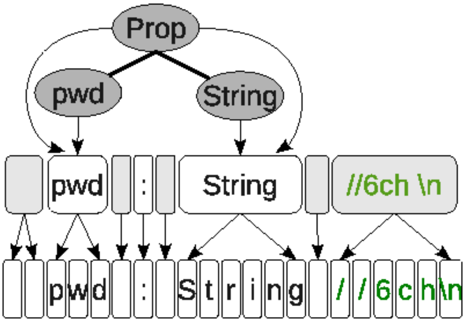

Fig. 5. Internal representation

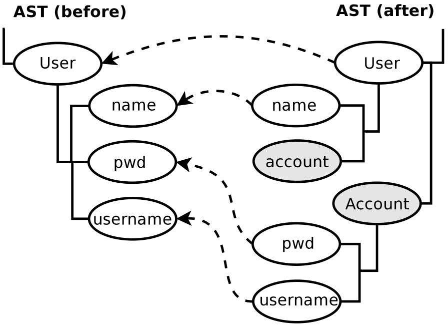

Fig. 6. Origin tracking

exposes the original node, its associated source code fragment, and details about surrounding layout such as indentation, separating whitespace and surrounding comments.

4 Layout Preservation for Transformed AST

In this section we describe the basic reconstruction algorithm and prove correctness and preservation.

4.1 Formalization

We introduce a formal notation for terms in concrete and abstract syntax, which stresses the correspondence between both representations. Given a grammar G, let S_G be the set of strings that represent a concrete syntax (sub)tree, and let T_G be the set of well-formed abstract syntax (sub)trees. We use the following notation for tree structures: $(t, [t_0...t_k]) \in T_G$ denotes a term $t \in T_G$ with subterms $[t_0...t_k] \in T_G$ (t_i denotes the subterm at the ith position). Equivalently, $(s, [s_0...s_k]) \in S_G$ means a string $s \in S_G$ with substrings $[s_0...s_k] \in S_G$, so that each s_i represents an abstract term $t_i \in T_G$, and s represents a term $(t, [t_0...t_k]) \in T_G$. Terms are characterized by their signature, consisting of the constructor name and the number of subterms and their sorts. When the constructor of a term is important, it is added in superscript ($(x^N, [x_0, ...x_k])$). Finally, for list terms the notation $[x_0, ...x_k]$ is used as short notation for $(x^{[]}, [x_0, ...x_k])$, leaving out the list constructor node.

We define the following operations on S_G and T_G, using the subscripts $_S$ and $_T$ to specify on which term representation the operation applies. Given a term $(x, [x_0...x_k])$ with subterm x_i, then $\mathrm{R}(x_i, x_{new})(x)$ replaces the subterm at position i with a new term x_{new} in term x. In case x is a list, additional operations are defined for deletion and insertion. $\mathrm{D}(x_i)(x)$ defines the deletion of the subterm at position i in x, while $\mathrm{IB}(x_i, x_{new})(x)$ and $\mathrm{IA}(x_i, x_{new})(x)$ define the insertion of x_{new} before (IB) or after (IA) the i^{th} element.

Assumption 1. *Let* PRS : $S_G \to T_G$ *the parse function that maps concrete terms onto their corresponding abstract terms.* PRS *is a homomorphism on tree structures.*

The text reconstruction algorithm translates the transformation in the abstract representation to the corresponding transformation in the concrete representation. This translation essentially exploits the homomorphic relationship between abstract and concrete terms. The applicability of the homomorphism assumption and techniques to overcome exceptional cases are discussed later in this section.

Lemma. *Let* PRS : $S_G \to T_G$ *the parse function, and assume* PRS : $S_G \to T_G$ *is a homomorphism on tree structures. Then the following equations hold:*

L 1. $\mathrm{PRS} \circ \mathrm{R}_S(s'_i, s_i)(s) = \mathrm{R}_T(\mathrm{PRS}(s'_i), \mathrm{PRS}(s_i)) \circ \mathrm{PRS}(s)$

L 2. $\mathrm{PRS} \circ \mathrm{D}_S(s'_i)(s) = \mathrm{D}_T(\mathrm{PRS}(s'_i)) \circ \mathrm{PRS}(s)$

L 3. $\mathrm{PRS} \circ \mathrm{IB}_S(s'_i, s_i)(s) = \mathrm{IB}_T(\mathrm{PRS}(s'_i), \mathrm{PRS}(s_i)) \circ \mathrm{PRS}(s)$

L 4. $\textsc{prs} \circ \mathrm{IA}_S(s_i', s_i)(s) = \mathrm{IA}_T(\textsc{prs}(s_i'), \textsc{prs}(s_i)) \circ \textsc{prs}(s)$

Proof. This follows from the assumption that PRS is a homomorphism on tree structures. □

Definition. *Given the functions* $\textsc{prs} : S_G \rightarrow T_G$*,* $\textsc{pp} : T_G \rightarrow S_G$*,* $\textsc{OrTrm} : T_G \rightarrow T_G$*,* $\textsc{OrTxt} : T_G \rightarrow S_G$*, with* PP *a pretty-print function and* ORTRM *and* ORTXT *functions that return the origin term respectively the origin source fragment of a term. The following properties hold:*

D 1. $\textsc{OrTrm}(\textsc{prs}(s)) = \textsc{prs}(s)$

D 2. $\textsc{OrTxt}(\textsc{prs}(s)) = s$

D 3. $\textsc{prs}(\textsc{OrTxt}(\textsc{OrTrm}(t))) = \textsc{OrTrm}(t)$

D 4. $\textsc{prs}(\textsc{pp}(t)) = t$

D 5. $\textsc{pp}(s) = s \; for \; all \; string \; terms \; s$

4.2 Algorithm

We define an algorithm that reconstructs the source code after the refactoring transformation (Figure 8). CONSTRUCTTEXT$(node)$ takes an abstract syntax term as input and constructs a string representation for this term. Three cases are distinguished; reconstruction for nodes (l. 1-5), reconstruction for lists (l. 6-11), and pretty printing in case the origin term is missing, i.e. when a term is newly created in the transformation (l. 12-14). We discuss those cases.

If an origin term with the same signature exists (l. 2-3), the text fragment is reconstructed from the original text fragment, corrected for possible changes in the subterms. The function $\mathrm{R}_S(t_i', ti) : String \rightarrow String$ subsequently replaces the substrings that represent original subterms with substrings for the new subterms constructed by a recursive call to CONSTRUCTTEXT (l. 5). The (relative) offset is used to locate the text fragment associated to the original subterm ($\textsc{OrTxt}(t_i')$), this detail is left out of the pseudo code.

Text reconstruction for list terms (line 6-11) implements the same idea, except that the changes in the subterms may include insertions and deletions. The textual modifications are calculated by a differencing function (DIFF) and subsequently applied to the original list fragment (line 11). The DIFF function matches elements of the new list with their origin term in the original list; the matched elements are returned as replacements (line 25), the unmatched elements of the old list form the deletions (lines 21, 29), while the insertions consist of the unmatched elements in the new list (lines 23, 30). It is crucial that the elements of the new list are correctly matched with related elements from the old list, since they automatically adopt the surrounding layout at the position of the old term, which may contain explanatory comments.

New terms are reconstructed by pretty-printing. To preserve the layout of subterms associated with an origin fragment, the pretty print function is applied after replacing the subterms with their textual representation, constructed recursively (line 14).

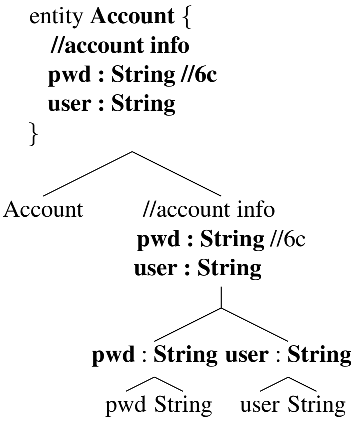

Fig. 7. Reconstruction example

The reconstruction algorithm implements a postorder traversal of the transformed abstract syntax tree, constructing the text fragment of the visited term from the text fragments of its subterms that were already constructed in the traversal. Figure 7 illustrates the reconstruction of the account entity. The substrings printed in bold are constructed by traversing the subterms, while the surrounding characters are either retrieved from the origin fragment, or constructed by pretty printing.

4.3 Correctness

We prove correctness of $\textsc{ConstructText} : T_G \rightarrow S_G$ (Figure 8, abbreviated as $\textsc{ct}$), assuming that $\textsc{parse} : S_G \rightarrow T_G$ is a homomorphism on tree structures.

Theorem (Correctness). $\forall t \in T_G$ $\textsc{parse}(\textsc{ct}(t)) = t$

The proof is by induction on tree structures. We distinguish two cases for the leaf nodes, dependent on whether an origin term exists with the same signature.

Base case (a). Let $t = (t^N, [])$ a leaf node with origin term $(t^N, [])$.
$\textsc{prs}(\textsc{ct}(t)) =^{line\ 5} \textsc{prs}(\textsc{OrTxt}(\textsc{OrTrm}(t))) =^{D\ 3} \textsc{OrTrm}(t) = (t^N, [])$ □

Base case (b). Let $(t, [])$ a leaf node for which no origin term exists.
$\textsc{prs}(\textsc{ct}(t)) =^{line 13-14} \textsc{prs}(\textsc{pp}(t)) =^{D\ 4} t$ □

IH. $\textsc{parse}(\textsc{ct}(t_i)) = t_i$ *holds for all subterms* t_0 *to* t_k *of a term* $(t, [t_0, ..., t_k])$.

We now proof the induction step $\textsc{parse}(\textsc{ct}(t)) = t$.

Induction step (a). Assuming the induction hypothesis, we first prove a property of text modification operations as applied in lines 5, 11.

```
ConstructText($term$) ▷ Abbreviated as CT
1  if
2      $(t^N, [t_0, ...t_k]) \leftarrow term$
3      $(t'^N, [t'_0, ...t'_k]) \leftarrow$ OrTrm($term$)
4  then  ▷ Term with origin info
5      return $R_S$(OrTxt($t'_0$), CT($t_0$)) $\circ \cdots \circ$ $R_S$(OrTxt($t'_k$), CT($t_k$))
           $\circ$ OrTxt(OrTrm($term$))
6  else if
7      $[t_0, ...t_k] \leftarrow term$
8      $[t'_0, ...t'_j] \leftarrow$ OrTrm($term$)
9  then  ▷ List term
10     [Mod$_0$, ...Mod$_z$] $\leftarrow$ Diff(OrTrm($term$), $term$)
11     return Mod$_0$ $\circ \cdots \circ$ Mod$_z$(OrTxt(OrTrm($term$)))
12 else  ▷ New constructed term
13     $(t^N, [t_0, ...t_k]) \leftarrow term$
14     return PP $\circ$ R$_T$($t_0$, CT($t_0$)) $\circ \cdots \circ$ R$_T$($t_k$, CT($t_k$))($t^N$)

Diff($originLst$, $newLst$)
15 $diffs, unmatched \leftarrow []$
16 for each $el$ in $newLst$ do
17     if OrTrm($el$) $\in originLst$ then
18         $el' \leftarrow$ OrTrm($el$)
19         if Prefix($el'$, $originLst$) $\neq []$ then
20             $deletedElems \leftarrow$ Prefix($el'$, $originLst$)
21             $diffs \leftarrow D_S$(OrTxt($deletedElems$)) :: $diffs$
22         if $unmatched \neq []$ then
23             $diffs \leftarrow IB_S$(OrTxt($el'$), CT($unmatched$)) :: $diffs$
24             $unmatched \leftarrow []$
25         $diffs \leftarrow R_S$(OrTxt($el'$), CT($el$)) :: $diffs$
26         $originLst \leftarrow$ Suffix($el'$, $originLst$)
27     else
28         $unmatched \leftarrow unmatched$ ::: $[el]$
29 $diffs \leftarrow D_S$(OrTxt($originLst$)) :: $diffs$
30 $diffs \leftarrow IA_S$(OrTxt($originLst$), CT($unmatched$)) :: $diffs$
31 return Reverse($diffs$)
```

Fig. 8. Pseudo code reconstruction algorithm

p 1. *Given a concrete syntax term* $(s, [...\textsc{OrTxt}(t'_i)...])$. *The following holds for modification operations* $\textsc{Mod} \in R,\ IB,\ IA,\ D$.
$\textsc{prs} \circ \textsc{Mod}_S(\textsc{OrTxt}(t'_i), \textsc{CT}(t_i))(s) =^{L\ 1, L\ 2, L\ 3, L\ 4}$
$\textsc{Mod}_T(\textsc{prs} \circ \textsc{OrTxt}(t'_i), \textsc{prs} \circ \textsc{CT}(t_i)) \circ \textsc{prs}(s) =^{D\ 3, IH}$
$\textsc{Mod}_T(t'_i, t_i) \circ \textsc{prs}(s)$

We prove the induction step for constructor terms (t^N) below, the proof for list terms follows the same logic. Let $t = (t^N, [t_0...t_k])$ a term with origin term $t' = (t'^N, [t'_0...t'_k])$.
$\textsc{prs} \circ \textsc{ct}(t) =^{line\ 5-6}$

$\textsc{prs} \circ \mathrm{R}_S(\textsc{OrTxt}(t'_0), \textsc{ct}(t_0))... \circ \mathrm{R}_S(\textsc{OrTxt}(t'_k), \textsc{ct}(t_k)) \circ \textsc{OrTxt}(t') =^{p\ 1}$
$\mathrm{R}_T(t'_0, t_0) \circ ... \circ \mathrm{R}_T(t'_k, t_k) \circ \textsc{prs} \circ \textsc{OrTxt}(t') = D\ 3$
$\mathrm{R}_T(t'_0, t_0) \circ ... \circ \mathrm{R}_T(t'_k, t_k)(t'^N, [t'_0...t'_k]) = (t^N, [t_0...t_k]) = t$ □

Induction step (b). First, we prove a property for pretty printing.

p 2. $\textsc{prs} \circ \textsc{pp} \circ \mathrm{R}_T(t'_i, t_i)(t) =^{D\ 4}$
$\mathrm{R}_T(t'_i, t_i)(t) =^{D\ 4}$
$\mathrm{R}_T(\textsc{prs} \circ \textsc{pp}(t'_i), \textsc{prs} \circ \textsc{pp}(t_i)) \circ \textsc{prs} \circ \textsc{pp}(t) =^{L\ 1}$
$\textsc{prs} \circ \mathrm{R}_S(\textsc{pp}(t'_i), \textsc{pp}(t_i)) \circ \textsc{pp}(t)$

Let $(t, [t_0...t_k])$ a node for which no origin term exists.
$\textsc{prs} \circ \textsc{ct}(t) =^{line\ 14}$
$\textsc{prs} \circ \textsc{pp} \circ \mathrm{R}_T(t_0, \mathrm{CT}(t_0)) \circ ... \circ \mathrm{R}_T(t_k, \mathrm{CT}(t_k))(t) =^{p\ 2}$
$\textsc{prs} \circ \mathrm{R}_S(\textsc{pp}(t_0), \textsc{pp} \circ \mathrm{CT}(t_0)) \circ ... \circ \mathrm{R}_S(\textsc{pp}(t_k), \textsc{pp} \circ \mathrm{CT}(t_k)) \circ \textsc{pp}(t) =^{L\ 1, D\ 5}$
$\mathrm{R}_T(\textsc{prs}{\circ}\textsc{pp}(t_0), \textsc{prs}{\circ}\mathrm{CT}(t_0)){\circ}...{\circ}\mathrm{R}_T(\textsc{prs}{\circ}\textsc{pp}(t_k), \textsc{prs}{\circ}\mathrm{CT}(t_k)){\circ}\textsc{prs}{\circ}\textsc{pp}(t) =^{D\ 4, IH}$
$\mathrm{R}_T(t_0, t_0) \circ ... \circ \mathrm{R}_T(t_k, t_k)(t) =^{D\ 4}\ t$ □

Applicability. The correctness proof depends on the assumption that parsing is a homomorphism on tree structures, we discuss two common exceptions. Tree structures in the concrete syntax representation can be ambiguous, in which case the parse result is determined by disambiguation rules. Syntactic ambiguities invalidate the homomorphic nature of the parse function. For instance, "2∗4+5", is parsed as $(t^{Plus}, [(t^{Mult}, [2, 4]), 5])$, while the alternate parse $((t^{Mult}, [2, (t^{Plus}, [4, 5])]))$ is rejected. Thus, bottom up text reconstruction fails to produce the correct code fragment for $(t^{Mult}, [2, (t^{Plus}, [4, 5])])$ in case $(t^{Plus}, [4, 5])$ is reconstructed as "$4 + 5$" instead of "$(4 + 5)$". To guarantee correctness, a preprocessor step is required that adds parentheses at the necessary places in the tree, where text reconstruction does not yield an expression between parentheses. The rules for parentheses insertion can be derived from the syntax definition [21]. This approach is taken in GPP [4], the generic pretty printer that is used in Spoofax. Another exception with respect to the homomorphism property concerns separation between list elements. When a list element is inserted (or deleted), it must be inserted (deleted) inclusive a possible separator, which is determined by the parent node. The separation is retrieved from the original source text in case the origin list has two or more elements, otherwise its looked up in the pretty-print table, based on te signature of the parent term.

4.4 Layout Preservation

Abstract syntax terms in general have multiple textual representations. These representations differ in the use of layout between the linguistic elements. In addition, small differences may occur in the linguistic elements; typically the use of braces is optional in some cases. We introduce the notion of formatting that covers these differences. Then we prove that the text reconstruction algorithm preserves formatting for terms that are not changed in the transformation, although they may have changes in their subterms.

Definition. *Given $(s, [s_0, ...s_k]) \in S_G$. The formatting of s is defined as the list consisting of the substring preceding s_0, the substrings that appear between the subterms $s_0, ...s_k$, plus the substring succeeding s_k*

Theorem (Maximal Layout Preservation). *Let $t \in T_G$ with origin term* $\textsc{OrTrm}(t) \in T_G$. *If t and $\textsc{OrTrm}(t)$ have the same signature, then $\textsc{ct}(t)$ and $\textsc{OrTxt}(\textsc{OrTrm}(t))$ have the same formatting.*

Proof. Let $(t^N, [t_0...t_k])$ a term with origin term $(t'^N, [t'_0...t'_k])$, then $\textsc{ct}(t) = \mathrm{R}_S(\textsc{OrTxt}(t'_0), \textsc{ct}(t_0)) \circ ... \circ \mathrm{R}_S(\textsc{OrTxt}(t'_k), \textsc{ct}(t_k)) \circ \textsc{OrTxt}(\textsc{OrTrm}(t))$. Since R_S only affects the substrings that represent the child nodes, the formatting of the parent string is left intact. For list terms: Let $t = [t_0...t_k]$ a list with origin term $\textsc{OrTrm}(t) = [t_0...t_l]$, then $\textsc{ct}(t) = \textsc{Mod}_{t'_0} \circ ... \circ \textsc{Mod}_{t'_l} \circ \textsc{OrTxt}(\textsc{OrTrm}(t))$ $\textsc{Mod}_{t'_i} \in \{\mathrm{R}_S, \mathrm{D}_S, \mathrm{IB}_S, \mathrm{IA}_S\}$. By definition, the modification functions affect the substrings representing the child nodes, or insert a new substring. In both cases the formatting of the parent string is preserved. □

5 Whitespace Adjustment and Comment Migration

The algorithm of Figure 8 preserves the layout of the unaffected regions, but fails to manage spacing and comments at the frontier between the changed parts and the unchanged parts. Figure 9 shows the result of applying the algorithm to the refactoring described in section 2 (Figure 1). Comments end up at the wrong location (`//account info`, `/*Blog info*/`), the whitespace separation around the `account` property and `Account` entity is not in accordance with the separation in the original text, and the indentation of the `Account` entity is disorderly.

The algorithm in Figure 8 translates AST-changes to modifications on code structures, but ignores the layout that surrounds these structures. To overcome this shortcoming, we refine the implementation of the algorithm so that whitespace and

```
entity User {
  name : String
  //account info

  account : Account expire : Date
}

/*Blog info*/
entity Account {
password : String //6 chars
  username : String
}entity Blog { ... }
```

Fig. 9. Layout deviation

```
IBADJUSTED($t_{old}, t_{new}$)
    $text \leftarrow \textsc{ct}(t_{new})$
    $text \leftarrow \textsc{RemoveIndent}(text)$
    $text \leftarrow \textsc{AddIndent}($
        $text,$
        $\textsc{OriginIndent}(t_{old}))$
    $text \leftarrow \textsc{ConcatStrings}([$
        $text,$
        $\textsc{OrSeparation}(t_{old})\ ])$
    $offset \leftarrow \textsc{OffsetWithLO}(t_{old})$
    return $\mathrm{IB}_S(offset, text)$
```

Fig. 10. Layout adjustment function

```
/**
 * Processes income data and displays statistics #1
 */
public static void displayStatistics(Scanner input) {
   //Initialize variables #2a
   int    count = 0;   // Number of values #3a
   double total = 0;   // Sum of all incomes #3b

   //Process input values until EOF #2b
   System.out.println("Enter income values");
   while (input.hasNextDouble()) {
      double income = input.nextDouble();
      //System.out.println("processing: " + income); #4
      if(income>=0){
         count++;            // Keep track of count
         total += income;    // and total income #5
      }
   }

   //Display statistics #2c
   double average = calcAverage(count, /*sum*/ total); #6
   System.out.println("Number of values = " + count);
   System.out.println("Average = "  + average);
}
```

Fig. 11. Comment styles

comments are migrated together with their associated code structures. This is implemented by using the layout-sensitive versions of the origin tracking functions to access origin fragments and locate textual changes. Language generic layout adjustment functions are implemented that correct the whitespace of reconstructed fragments, so that the spacing of the surrounding code is adopted. In particular, an inserted fragment is indented and separated according to the layout of the adjacent nodes. Figure 10 shows the layout adjustment steps for IB$_S$. First, the text is reconstructed with its associated comments. Then, the existing separation and (start)indentation is removed, leaving the nesting indentation intact. Subsequently, the start indentation at the insert location is retrieved from the adjacent term (t_{old}) and appended to all lines. Finally, separation is added (retrieved by inspecting the layout surrounding t_{old}) to separate the node from its successor.

5.1 Comment Heuristics

Comment migration requires a proper interpretation of how comments attach to the linguistic structure, which is problematic because of the informal nature of comments. The use of comments differs, depending on style conventions for a particular language and the personal preference of the programmer. Van De Vanter [6] gives a detailed analysis.

Figure 11 illustrates the use of comments with different style conventions used in combination. Fragment #1 is a block comment that explains the purpose of the accompanying method. The comment resides in front of its structural referent. This is also the case for the comments in #2a,b,c. However, these comments do not attach to a single structure element, but instead relate to a group of statements. The blank lines that surround these grouped statements are essential in understanding the scope of the comments. Contrary to the previous examples, the line comment in #3 points backwards to the preceding statement. #6 provides an example of a comment in the context of list elements separated by a comma. In this case, the location of the comma determines whether the comment points forward or backward. The commented-out `println` statement in #4 does not have a structural referent. It can best be seen as lying between the surrounding code elements. Finally, #5 illustrates a single comment that is spread over two lines. A human reader will recognize it as a single comment, although it is structurally split in two separate parts. In this case, the vertical alignment hints at the fact that both parts belong together.

Figure 11 makes clear why attaching comments to AST nodes is problematic. The connection of comments with AST-nodes only becomes clear when taking into account the full documentary structure, including newlines, indentation and separator tokens. Comments can point forward, as well as backward and, purely based on analysis of the tree structure, it is impossible to decide which one is the case. Even more problematic are #2 and #4; both comment lines lack an explicit referent in terms of a single AST node. The former refers to a sublist, while the latter falls between the surrounding nodes.

Text reconstruction allows for a more flexible approach towards the interpretation of comments. Instead of a fixed mapping between comments and AST nodes, heuristic rules are defined that interpret the documentary structure around the moved AST-part. Comment heuristics are defined as layout patterns using newlines, indentation, and separators as building blocks (Figure 12). If a pattern applies to a given node (or group of nodes), the node is considered as the structural referent of the comment(s) that take part in the pattern. The binding heuristics have the following effect on the textual transformation; if a node / group of nodes is (re)moved, all adjacent comments that bind to the node(s) are (re)moved as well. Adjacent comments that do not bind, stay at their original position in the source code. Comments that lie inside the region of the migrated node(s) automatically migrate jointly.

The patterns in Figure 12 handle the majority of comment styles correctly. The comment styles in Figure 11 are recognized by the patterns, with the exception of vertical alignment (#5), which is not detected. Preceding(1) binds #1 to the `displayStatistics` method, and #2a,b,c to the statement groups they refer to. #3 is interpreted by Succeeding(1). None of the patterns applies to #4, which indeed neither binds to the preceding nor to the succeeding node. The comment in #6 is associated with the succeeding node by application of Preceding(2). Finally, #5 is associated to its preceding statement, but not recognized as a single comment spread over two lines.

Heuristic rules will never handle all cases correctly; ultimately, it requires understanding of the natural language to decide the meaning of the comment and how it relates to the program structure. While our experience so far suggests that the heuristics are adequate, further experience with other languages, other refactorings, and other code bases is needed to determine whether these rules are sufficient.

Preceding(1):

```
<newline OR lower-indent><newline>
<comments><newline>
<nodes><newline>
<newline OR lower-indent>
```

```
{
    /*..*/
    int i
    int j
}
```

Preceding(2):

```
<separator><comments><node>
```

```
int i, /*..*/ int j
```

Succeeding(1):

```
<node><comments><newline>
```

```
int i /*..*/
int j
```

Succeeding(2):

```
<node><comments><separator>
```

```
int i /*..*/ , int j
```

Succeeding(3):

```
<node><separator><comments><newline>
```

```
int i, /*..*/
int j
```

Fig. 12. Comment patterns

6 Evaluation

We implemented the layout preservation algorithm in Spoofax [12], the sources of the library are available on-line [2]. We successfully applied the algorithm to renaming, extraction and inlining refactorings defined in WebDSL [24], MoBL [11] and Stratego [3]. In addition, we applied the algorithm to the Java refactorings mentioned in this section. For future work we will implement more refactorings and we will experiment with different languages and layout conventions.

Van De Vanter [6] points out the importance of the documentary structure for the comprehensibility and maintainability of source code. The paper gives a detailed analysis of the documentary structure consisting of indentation, line breaks, extra spaces and comments. The paper sketches the prerequisites for a better layout handling by transformation tools. We use the examples and requirements pointed out by Van De Vanter to provide a qualitative evaluation of our approach.

It is impossible for automatic tools to handle all layout correctly. After all, textual comments are written for human beings. Ultimately, comments can only be related to the code by understanding natural language. Therefore, instead of trying to prove that our tool handles layout correctly, we show that our approach meets practical standards for refactoring tools. We compare the layout handling of our technique with the refactoring support in Eclipse Java Development Tools (JDT), which is widely used in practice. We use a test set consisting of Java fragments with different layout styles. This set includes test cases for indentation and separating whitespace, as well as test cases for different comment styles, covering all comment styles discussed by Van De Vanter [6] and illustrated in Figure 12.

The results are summarized in Table 1; + means that the layout is accurately handled, -/+ indicates some minor issues, while - is used in case more serious defects were found. A minor issue is reported when the layout is acceptable but doen not precisely

Table 1. Layout Preservation Results

	Cat.	Description	E	CT
1	P1	Inline on method preceded by block comment	+	+
2		Inline on method preceded by a commented-out method	-	+
3		Move method preceded by multiple comments	+	+
4		Convert-to-field on the first statement of a group preceded by a comment	-	+
5		Convert-to-field on statement below commented-out line	-	+
6	P2	Change method signature	+	+
7	S1	Extract method, last stm ends with line comments	+	+
8		Extract method, preceding stm ends with line comments	+	+
9		Convert-to-field, decl with succeeding line comments	-	+
10	S2	Change method signature	+	+
11	S3	Change method signature	-/+	-/+
12	Inside	Extract method with comments in body	+	+
13		Inline method with comments in body	+	+
14	Selection	Extract method, preceding comments in selection	+	+
15		Extract method, preceding comments outside selection	+	+
16	Indent	Extract method, code style follows standards	+	+
17		Extract method, code style deviates from standards	-	-/+
18	Sep. ws	Extract method, code style follows standards	+	+
19		Extract method, code style deviates from standards	-/+	+
20	Format	Extract method, standard code style	+	+
21		Extract method, code style deviates from standard	-/+	-/+
22	V. align	Renaming so that v. alignment of "=" is spoiled	-	-
23		Renaming so that v. alignment of comments is spoiled	-	-

E : Eclipse Helios (3.6.2)
CT: Text Construction

follows the style used in the rest of the code, a serious defect is reported in case the layout is untidy or when comments are lost. The results show that our approach handles layout adequately in most cases. Different comment styles are supported (1-15), and the adjustment of whitespace gives acceptable results (16-19). 17, 19, and 23 show that variations in code style only led to some minor issues. For example in 17, the indent of the new inserted method correctly follows the indentation of the adjacent methods, but the indentation in the body follows the style defined in the pretty-print definition. Vertical alignment (22, 23) is not restored. A possible improvement is to restore vertical alignment in a separate phase, using a post processor.

Eclipse does not implement the same refined heuristic patterns as our technique, which explains the deviating results in 2, 4, and 5. In those three cases, the comments were incorrectly associated with the moved code structures and, consequently, did not remain at their original location. In all three cases the comment did not show up in the modified source code. In 9, the comment was not migrated to the new inserted field, although it was (correctly) associated to the selected variable declaration. The reason is that the relation between the inserted field and the deleted local variable is not set. In our implementation, the origin tracking mechanism keeps track of this relation. Eclipse uses editor settings to adjust the whitespace surrounding new inserted fragments, which works well under the condition that the file being edit adopts these settings.

We implemented a general solution for layout preservation with the objective to support the implementation of refactorings for new (domain specific) languages. Using our approach, the layout preservation is not a concern for the refactoring programmer but it is automatically provided by the reconstruction algorithm. The evaluation indicates that our generic approach produces results of comparable and in some cases even better quality then refactorings implemented in current IDEs.

7 Related Work

We implemented an algorithm for layout preservation in refactoring transformations. Instead of trying to construct the entire source code from the AST, the algorithm uses the original source text to construct the text for the transformed AST. Origin tracking is used to relate terms in the AST with their original code fragments, while internal changes are propagated and applied as text patches. As a result, the original layout is preserved for the unaffected parts of the program. The main challenge is the treatment of spacing and comments on the frontier between the changed and the unchanged code. Layout adjustment functions correct the whitespace of reconstructed fragments, so that the spacing of the surrounding code is adopted. Comments are migrated according to their intent. We define heuristic patterns for comment binding, that interpret the documentary structure near the node. The comment patterns are flexible in the sense that they do not assume a one-to-one relation between comments and AST nodes. The heuristic rules are language generic and cover the layout styles commonly seen in practice.

7.1 AST Approaches

Various attempts have been made to address the concern of appearance preservation by adding layout information to the AST. For a complete reconstruction, all characters that do not take part in the linguistic structure should be stored. This includes whitespace, comments and (redundant) parentheses. The modified source code is reconstructed from the transformed AST by layout-aware pretty printing [5].

Van den Brand and Vinju [20] use full parse trees in combination with rewrite rules in concrete syntax. The rewrite engine is adapted to deal with the extra layout branches, by using the assumption that any two layout nodes always match. The approach described in [18] also relies on extra layout branches. Instead of adapting the rewrite engine, the authors propose an automated migration of the transformation rules to take care of the layout branches. Layout annotations are used in [14] (Kort, Lämmel), while the RefactorErl tool [13] stores the layout information in a semantic graph.

All approaches based on extended ASTs succeed, to a certain extent, in preserving the original layout. In most approaches, layout is preserved for the unaffected parts, but the reconstruction of the affected parts has limitations. The implicit assumption is that the documentary structure can be mapped satisfactorily onto abstract syntax trees. However, the mapping of layout elements to AST nodes has intrinsic limitations. Attaching comments to preceding (or succeeding) AST nodes is a simplification that fails in cases when a comment is not associated with a single AST node, as is shown in examples provided by Van De Vanter [6]. Another shortcoming is related to indentation and whitespace separation at the beginning and end of changed parts. Migrating whitespace is not sufficient since the indentation at the new position may differ from the indentation at the old position, due to a different nesting level. Furthermore, newly constructed structures should be inserted with indentation and separating whitespace.

7.2 HaRe

HaRe [15,17] is a refactoring tool for Haskell that preserves layout. The program is internally represented by the Abstract Syntax Tree and the token stream, which are linked by source location information. Layout preservation is performed explicitly in the transformation steps, which process the token stream and the AST in parallel. After the transformation, the source code is extracted from the modified token stream.

Haskell programs can be written in layout-sensitive style for which the meaning of a syntax phrase may depend on its layout. For this reason, it is essential for the refactoring tool not to violate the layout rules when transforming the program. HaRe implements a layout adjustment algorithm to keep the layout correct. The algorithm ensures that the meaning of the code fragments is not changed, which does not necessarily mean that the code is as much as possible like the original one in appearance. HaRe uses heuristic rules to move/remove comments together with the associated program structures. These heuristics include rules for comments that precede a program structure and end-of-line comments that follow after a structure.

Similar to our approach, HaRe uses the token stream to apply layout analysis and to extract source code fragments. The main difference is that HaRe modifies the token stream during the transformation, while we reconstruct the source code afterwards, using origin-tracking to access the original source. The requirement to change the AST and token stream in parallel makes it harder to implement new transformations and requires an extension of the rewrite machinery specific for source-to-source transformations. We clearly separate layout handling from rewriting, which enables us to use the existing compiler infrastructure for refactoring transformations.

7.3 Eclipse

The Java Developer Toolkit (JDT) used in Eclipse offers an infrastructure for implementing refactorings [1]. Refactoring transformations are specified with replace, insert and remove operations on AST nodes, which are used afterwards to calculate the corresponding textual changes. Common to our approach, the replace, insert and remove operations on AST nodes are translated to textual modifications of the source code. However, instead of being restricted to the replace, delete and insert operations on AST nodes, we compute the primitive AST modifications by applying a tree differencing algorithm to the transformed abstract syntax tree. As a result, the transformation and text reconstruction are clearly separated. Thanks to this separation of concerns, we can specify refactorings in a specialized transformation language (Stratego).

7.4 Text Patching

The LS/2000 system [7,19] is a design-recovery and transformation system, implemented in TXL. LS/2000 is successfully applied for ”year 2000” remediation of legacy COBOL, PL/I, and RPG applications. The system implements an approach based on automated text patching. The differences between the original code and the transformed code are calculated with a standard differencing algorithm, operating on the token stream. The deviating text regions are merged back into the original text.

The token based differencing successfully captured changes that were relatively small. For millennium bug renovations, typical changes were the local insertion of a few lines of code. When the changes are large, or involve code movement, standard differencing algorithms do not work well [19]. We implemented a tree differencing algorithm that reconstructs moved code fragments by using origin tracking, furthermore, fragments with nested changes are reconstructed by recursion on subtrees.

7.5 Lenses

Foster et al. [8] implement a generic framework for synchronizing tree-structured data. Their approach to the view update problem is based on compoundable bi-directional transformations, called lenses. In the GET direction, the abstract view is created from the concrete view, projecting away some information; in the PUTBACK direction, the modified abstract view is mapped to a concrete representation, restoring the projected elements from the original concrete representation. The lens laws, which resemble our preservation and correctness criteria, impose some constraints on the behavior of the lens. Given a certain GET function, in general, many different PUTBACK functions can be defined. The real problem is to define a PUTBACK function that does what is required for a given situation. We define CONSTRUCTTEXT as a PUTBACK function for parsing, and prove that it fulfills the correctness and (maximal) layout preservation criteria.

Our approach is based on origin tracking as a mechanism to relate abstract terms with their corresponding concrete representation. Origin tracking makes it possible to locate moved subtrees in the original text. Furthermore, lists are compared using the origin relation to match corresponding elements. In contrast, lenses use the concrete representation as an input parameter to the PUTBACK function. As a consequence, details are lost about how subterms relate to text fragments. This seems especially problematic in case terms have nested changes, or when they are moved to another location in the tree. We defined heuristic rules for comment binding and layout adjustment functions to correct the spacing surrounding the changed parts. Layout adjustment and comment migration might be hard to express in the lenses framework. Foster et al. [8] mention the expressiveness of their approach as an open question. Layout preservation seems a challenging problem in this respect.

8 Conclusion

Refactorings are source-to-source transformations that help programmers to improve the structure of their code. With the popularity and ubiquity of IDEs for mainstream general purpose languages, software developers come to expect rich editor support including refactorings also for domain-specific software languages. Since the effort that can be spent on implementations of DSLs is often significantly smaller than the effort that is spent on (IDEs for) languages such as Java, this requires tool support for the high-level definition of refactorings for new (domain-specific) software languages.

An important requirement for the acceptability of refactorings for daily use is their faithful preservation of the layout of programs. Precisely this aspect, as trivial as it often seems compared to the actual refactoring transformation, has confounded meta-tool developers. The result is typically that the definitions of refactorings are contaminated with code for layout preservation. The lack of a generic solution for layout preservation has held back widespread development of refactoring tools for general purpose and domain-specific languages.

In this paper, we have presented an approach to layout preservation that separates layout preservation from the structural definition of refactorings, allowing the refactoring developer to concentrate on the structural transformation, leaving layout reconstruction to a generic library. The library computes text patches based on the differences between the old and the new abstract syntax tree, relying on origin tracking to identify the origins of subtrees. The approach applies layout conventions for indentation and vertical layout (blank lines) from the old code to newly created pieces of code; heuristic rules are defined for comment migration.

The separation of layout preservation from transformation enables the implementation of refactorings by the common meta-programmer. With this framework in place we expect to develop a further library of generic refactorings that will further simplify the development of refactorings for a wide range of software languages.

References

1. Eclipse documentation: Astrewrite Eclipse, JDT 3.6 (2010), `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/rewrite/ASTRewrite.html`
2. The Spoofax language workbench (2010), `http://strategoxt.org/Spoofax`
3. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming 72(1-2), 52–70 (2008)
4. de Jonge, M.: A pretty-printer for every occasion. In: The International Symposium on Constructing Software Engineering Tools (CoSET 2000), pp. 68–77. University of Wollongong, Australia (2000)
5. de Jonge, M.: Pretty-printing for software reengineering. In: ICSM 2002: Proceedings of the International Conference on Software Maintenance (ICSM 2002), p. 550. IEEE Computer Society, Washington, DC (2002)
6. Van de Vanter, M.L.: Preserving the documentary structure of source code in language-based transformation tools. In: 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), November 10, pp. 133–143. IEEE Computer Society, Florence (2001)
7. Dean, T.R., Cordy, J.R., Schneider, K.A., Malton, A.J.: Using design recovery techniques to transform legacy systems. In: ICSM 2001: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001), p. 622. IEEE Computer Society, Washington, DC (2001)

8. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
9. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
10. Fowler, M.: Language workbenches: The killer-app for domain specific languages? (2005)
11. Hemel, Z., Visser, E.: Programming the Mobile Web with Mobl. Technical Report 2011-01, Delft University of Technology (January 2011)
12. Kats, L.C.L., Visser, E.: The Spoofax language workbench. Rules for declarative specification of languages and ides. In: Rinard, M. (ed.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, Reno, NV, USA, October 17-21 (2010)
13. Kitlei, R., Lóvei, L., Nagy, T., Horváth, Z., Kozsik, T.: Layout preserving parser for refactoring in Erlang. Acta Electrotechnica et Informatica 9(3), 54–63 (2009)
14. Kort, J., Lämmel, R.: Parse-tree annotations meet re-engineering concerns. In: Proceedings of Third IEEE International Workshop on Source Code Analysis and Manipulation (September 2003)
15. Li, H., Thompson, S.: A comparative study of refactoring Haskell and Erlang programs. In: Penta, M.D., Moonen, L. (eds.) Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), pp. 197–206. IEEE (September 2006)
16. Li, H., Thompson, S., Orosz, G., Toth, M.: Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In: Horvath, Z., Teoh, T. (eds.) Proceedings of the Seventh ACM SIGPLAN Erlang Workshop, p. 12. ACM Press (September 2008)
17. Li, H., Thompson, S., Reinke, C.: The Haskell Refactorer: HaRe, and its API. In: Boyland, J., Hedin, G. (eds.) Proceedings of the 5th Workshop on Language Descriptions, Tools and Applications (LDTA 2005) (April 2005)
18. Lohmann, W., Riedewald, G.: Towards automatical migration of transformation rules after grammar extension. In: CSMR 2003: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, p. 30. IEEE Computer Society, Washington, DC (2003)
19. Malton, A., Schneider, K.A., Cordy, J.R., Dean, T.R., Cousineau, D., Reynolds, J.: Processing software source text in automated design recovery and transformation. In: Proc. International Workshop on Program Comprehension (IWPC 2001), pp. 127–134. IEEE Press (2001)
20. van den Brand, M., Vinju, J.: Rewriting with layout. In: Kirchner, C., Dershowitz, N. (eds.) Proceedings of RULE (2000)
21. van den Brand, M., Visser, E.: Generation of formatters for context-free languages. ACM Transactions on Software Engineering Methodology 5(1), 1–41 (1996)
22. van Deursen, A., Klint, P., Tip, F.: Origin tracking. J. Symb. Comput. 15(5-6), 523–545 (1993)
23. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
24. Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)