SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

November 2015, Vol. 58 112106:1–112106:14
doi: 10.1007/s11432-015-5368-9

Constructing format-preserving printing from syntax-directed definitions

WANG LiChao[1], LI GuoQiang[1*] & HU ZhenJiang[2]

[1]*BASICS, School of Software, Shanghai Jiao Tong University, Shanghai* 200240*, China;*
[2]*National Institute of Informatics, Tokyo* 101-8430*, Japan*

Received October 21, 2014; accepted May 6, 2015; published online September 29, 2015

Abstract Transformations between source codes, such as refactorings and program analysis, are frequently used in software engineering. Typically, transformations are effectively implemented using an abstract syntax tree (AST) on the origin source code. However, a critical limitation of ASTs is the loss of layout information such as whitespace and comments, which can result in poor readability. To overcome this shortcoming, this paper proposes a bidirectional transformation (BX) method that maintains consistency in the layout between the origin and transformed. First, a section of origin source code will be translated to a concrete syntax tree (CST) that includes layout information. Second, to make the BX practical, a new method is constructed that matches an AST with its respective CST. Finally, to get a reasonable CST, a method to amend the CST is also provided. We prove that the BX is well-behaved, which implies that it satisfies both the `PutGet` and `GetPut` laws. Furthermore, we illustrate the correctness of the methodology by treating XML language as a case study.

Keywords format-preserving, bidirectional transformation, pretty printing, unparsing, tree matching

Citation Wang L C, Li G Q, Hu Z J. Constructing format-preserving printing from syntax-directed definitions. Sci China Inf Sci, 2015, 58: 112106(14), doi: 10.1007/s11432-015-5368-9

1 Introduction

A source code has a formal linguistic structure, and an informal documentary structure [1] specified by various languages. Typically, a linguistic structure consists of identifiers, key words, and lexical tokens, and a documentary structure includes whitespace and comments. Although a documentary structure is not a formal part of the linguistic structure, it determines the appearance of a piece of code, and is essential for readability. The elements that make up linguistic structure are referred to as layout information. Layout information is significant. Comments can be used to explain the purpose of a piece of code in natural language, while indentations can be used to visually convey hierarchical structure. Extra whitespace helps us to distinguish one block of code from another.

In many situations, computers transform the source code of a program based on its abstract syntax tree (AST). It is more efficient and convenient than performing transformations on the code's textual presentation [2], since the tree-like structure is more readable and analyzable than the H-language defined by human beings. As shown in Figure 1, a traditional transformation based on AST works as follows.

* Corresponding author (email: li.g@sjtu.edu.cn)

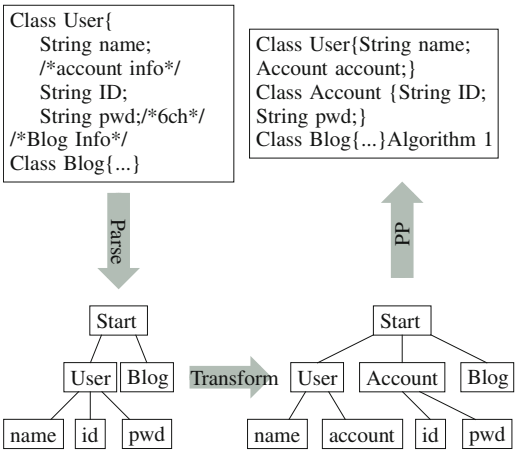

Figure 1 Transformation based on abstract syntax tree.

First the origin source code is parsed into its corresponding AST, then the AST is converted to the target AST, and finally, the target AST is unparsed to generate the target code for human use. The target code will have a different appearance from the origin source code after unparsing due to the lack of layout information. It is very difficult for human users to understand the target code. In many situations, it is necessary to retain certain elements of formatting when performing transformations on ASTs.

This paper proposes an algorithm to retain the appearance when transformations are implemented on ASTs. Unlike the traditional method, the target AST will be compared to the origin source code instead of unparsing it directly. Corresponding parts will be obtained during comparison. As a source code contains layout information, the corresponding parts of the target code will have some of the layout information from the source code. When constructing target code, layout information from the corresponding part is used, so that the target code will have a similar appearance to the origin code. Finding the corresponding parts between the origin source code and target AST is challenging, as it is difficult to directly discover correspondences between textual representations and abstract representations of a program. To overcome this, the origin source code is converted to a concrete syntax tree (CST) with layout information. Then, the CST is matched with the target AST so that the corresponding parts are revealed. A new CST with layout information is constructed using the corresponding parts. By printing all the leaf nodes of the new CST, the target code is generated. The new CST may be unreasonable, which means some sentences generated by the CST do not follow a correct context-free grammar. Of course, it is not necessary to find all the correct sentences (in fact, it is impossible). It is a separate challenge to determine whether a CST is reasonable or not. A method to amend the possibly unreasonable new CST is proposed, and target code with similar appearances to the origin code can be obtained after amending. We illustrate the correctness by adopting XML transformations as a case study.

In summary, the paper provides the following contributions:

• a new method to match an AST with a CST.

• a bidirectional transformation to preserve the appearance of a code fragment when the transformation is based on an AST.

• a new method to amend unreasonable CSTs.

Related work: Early work in format preserving that supports a specific language was done in the InterLisp system for the Lisp language [3], and common principles and experiences of early interactive Lisp environments are described in [4]. A generic unparsing generator was used for Pascal in the DICE system [5,6], and it was later implemented on Ads. None of these approaches preserve comments when unparsing. More recently, Jonge and Visser devised an algorithm for layout preservation in refactoring transformations [7]. Unlike our algorithm, their algorithm mainly works on code refactoring while our algorithm can work on two ASTs that have no corresponding parts. Moreover, their method reconstructed source code after transformation while ours constructs the source code during transformation. The group

of Li and Thompson has implemented a Haskell refactoring tool named HaRe [8]. HaRe is intended exclusively for Haskell. As Haskell can be written in a layout-sensitive manner, the refactoring tool of Haskell must obey layout rules to be reasonable. The algorithm retains the behavior of the code fragment but does not necessarily retain its appearance. Eclipse has a method to generate machine-generated standard indentations[1)]. This method is accepted by some users, while others are not satisfied as it is not user defined. Ranjitha Kumar along with his partner presented a flexible tree-matching method [9]. They offered a rather accurate method to match two trees and prove that it is NP-complete in the strong sense. The algorithm they offer is feasible, but the input tree cannot have two nodes with the same label. Because the method we use is for matching grammar trees, it is possible that a node and its descendants would have the same label. In our method, a descendant node N can match a node, that is the ancestor of N's ancestor's corresponding node by using our tree match method. Bidirectional transformations (BX) provide a mechanism for synchronizing and maintaining the consistency of information between input and output [10]. Not every combination of forward and backward functions constitutes a reasonable bidirectional transformation. To be well-behaved, the BX we construct should satisfy the PutGet and GetPut laws.

Paper organization: The rest of the paper is organized as follows. In Section 2, we explain our method in abstract terms. Syntax-directed definitions together with other background knowledge is introduced in Section 3. The concrete explanation of our method is given in Section 4. In Section 5, we give a proof that our method satisfies certain laws, and then we present our tree-matching method and extend the method on XML in Section 6. The conclusions and future work are discussed in Section 7.

2 Preliminaries

One typically encounters syntax errors when amending a piece of code written in an unfamiliar language. To correct the errors, we have to amend the code repeatedly. Performing transformations on the abstract syntax tree of a code fragment is easier than on the source code directly if we are not familiar with its syntax, as some parts of different languages are quite similar, such as arithmetic operation and control flow sentences. Furthermore, dealing with tree structure is more efficient than textual representations when implementing algorithms.

2.1 Significance of format preservation

We discuss the format preservation issue using the running example in Figure 1. It is written in Java with two classes. One is user and the other is Blog. We want to pull out two fields of User to create a new class named account, which consists of the information to login (Id and PWD). Transformations are performed as follows [11]. First, the origin source code is parsed to an AST, and the AST is converted to the resultant AST, then we pretty-print [11] the resulting AST so that the target source code is achieved. The result from the pretty-print has a markedly different appearance from the origin code, even with the loss of comments after pretty-printing [12]. The lake of readability is an issue, as it makes the target code difficult to understand. Hence, it is necessary to preserve the layout format, particularly in code transformations based on AST [13]. In this paper, we present an algorithm that can achieve this.

2.2 Problem analysis

To keep the layout information, we need to know how to discover the corresponding parts between the abstract representation and the textual representation. Two parts in different representative model are the same, if they refer to the same piece of source code. Discovering the corresponding parts between abstract representations and textual representations directly is challenging, because one is a tree-structure while the other is made up of a string of lexical tokens.

There are two main methods to store the layout information in a parse tree. Layout information can either be stored in the form of special layout nodes or in the form of tree annotations. We mixed

1) "Eclipse website": http://www.eclipse.org.

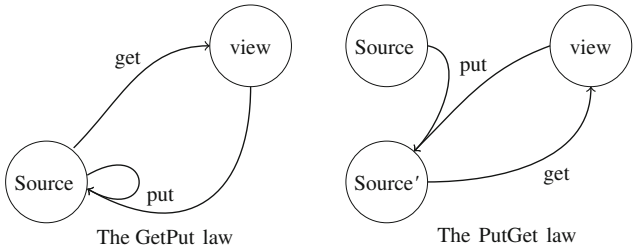

Figure 2 `GetPut` and `PutGet` laws.

the two methods, such that comments are considered as parse tree nodes while the information such as whitespace, newlines, and indentations are annotations in tree nodes. As comments are stored as tree nodes in the CST, it is necessary to consider them as special lexical tokens when doing lexical analysis by adding them into the grammar.

2.3 Bidirectional transformation

A bidirectional transformation (BX) consists of a pair of transformations. The forward transformation is used to produce a target view from a source, termed `get`. The backward transformation is used to put back modifications on the view to the source [10], and is termed `put`.

Not every combination of `put` and `get` is practical. A BX that satisfies the `GetPut` law and `PutGet` law is well-behaved [14]. The two laws are defined as follows,

$$\texttt{GetPut law}: \texttt{put}\ s\ (\texttt{get}\ s) = s;$$
$$\texttt{PutGet law}: \texttt{get}\ (\texttt{put}\ s\ v) = v.$$

As shown in Figure 2, The `GetPut` law stipulates that the put function should yield an unmodified source when passing the view obtained by get that is unchanged along with the original source. This property captures the intuition that if no view update takes place the source should not be updated either. The `PutGet` law implies that when passing a source obtained by put to get then get should yield the same view that was passed to put. This property captures the intuition that view updates are reflected in the source type by put and can be observed by get.

3 Syntax-directed definitions

A context-free grammar consists of four components: a set of terminal symbols, a set of nonterminals, a set of productions and a designation of one of the nonterminals as the start symbol [15]. A parse tree pictorially shows how the start symbol of a grammar derives a string in the language [15]. Formally, given a context-free grammar, a parse tree according to the grammar is a tree with the following properties:

- The root is labeled by the start symbol.
- Each leaf is labeled by a terminal or by ε.
- Each interior is labeled by a nonterminal.
- If A is a nonterminal labeling some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \to X_1X_2...X_n$.

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions [15]. An instance of an SDD is shown in Table 1 in a Bison way [16].

The appearance of an AST is determined by its construction. SDD is used in constructing an AST [17]. AST differs in different compilers and languages, but most AST nodes contain three components, the constructor that creates the node, some attributes, and children. There are some key attributes of an abstract syntax tree node. Namely the key attributes are values of numbers or IDs of variables. The node's key attributes and its constructor constitute the node's signature.

Table 1 A syntax-directed definition

Productions	Semantic rules
Start symbol	
expseq	execute($1)
expseq:	
exp	$$=$1
| expseq semicoloncom exp	$$=new seq($1,$3)
exp:	
exp2	$$=$2
| exp pluscom exp2	$$=new plus($1,$3)
exp2:	
exp3	$$=$1
| exp2 multcom exp3	$$=new plus($1,$3)
exp3:	
Lparen exp Rparen	$$=$2
| Num	$$=new value($1)
semicoloncom	
| semicolon comment	
pluscom	
plus	
| plus comment	
multcom	
mult	
| mult comment	

Definition 1. Two AST nodes A, B without any child are the same, denoted by $A = B$ if their signatures are the same. The abstract syntax tree nodes A with children $[a_1, a_2, \ldots, a_n]$ and B with children $[b_1, b_2, \ldots, b_n]$ are the same if A's signature is the same as B's signature and their children recursively satisfy $a_1 = b_1, a_2 = b_2, \ldots, a_n = b_n$. If their signatures are the same but the children are not all the same, we name it approximated, denoted by $A \approx B$.

4 Processes of main functions

We develop an algorithm named *putback* to preserve the appearance when performing transformations based on AST with three parameters: a parse tree C, an abstract syntax tree A, and a syntax directed definition S. The output is a CST, C'.

In this section, a simple example illustrates how *putback* works. The SDD we used is in Table 1, together with the input parse tree and the input abstract syntax tree shown in Figure 3.

The process of *putback* is shown below:

(1) Convert the input parse tree C to an abstract syntax tree A'.

(2) Discover the corresponding parts between A and A', then find the corresponding parts between A and C.

(3) Construct a new parse tree T, which keeps part of layout information stored in C. T may be unreasonable.

(4) Amend the parse tree T obtained in step 3, to a reasonable one C'.

As indicated in Figure 4, *putback* consists of two subprocess, *Construct* and *Modify*. *Construct* is used to construct a new CST that contains some layout information stored in origin CST, and *Modify* is used to amend the possibly unreasonable CST produced by *Construct*. Moreover, with the help of SDD, a CST can be converted into its corresponding AST, a process referred to as *C2A* (shown as Algorithm 1).

If C' is taken as an argument of function *C2A*, then the transformed AST A can be obtained, just as

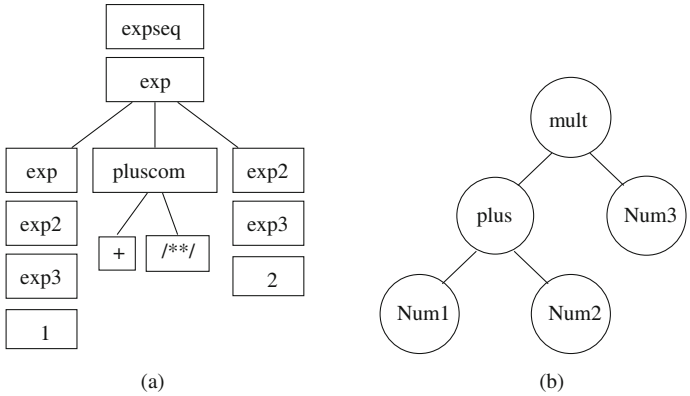

Figure 3 The instance to the input parse tree and abstract syntax tree. (a) CST for $1+/*\text{plus}*/2$; (b) AST for $(1+2)*3$.

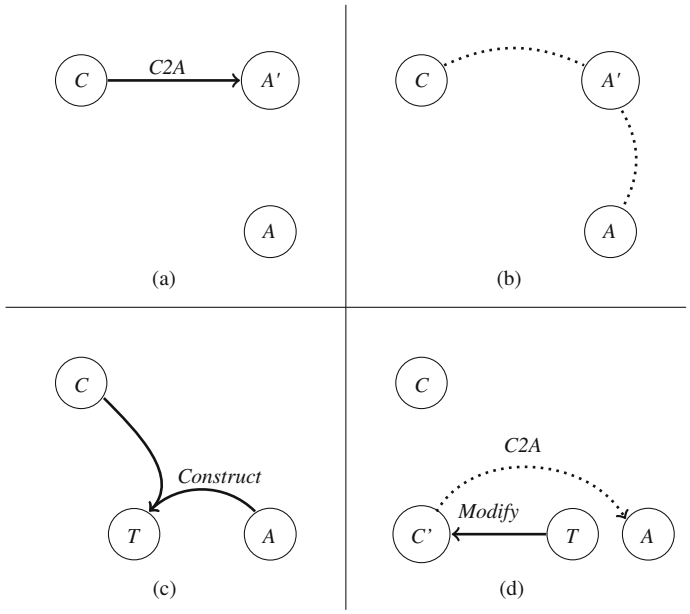

Figure 4 Process to *putback*. (a) Step 1; (b) step 2; (c) step 3; (d) step 4.

Figure 4 shows. *putback* and *C2A* constitute a well-behaved BX; the well-behavedness will be proven in Section 6.

4.1 Converting a CST to an AST

The purpose of *putback* is to create a CST C' that preserves the layout information stored in C as much as possible; its corresponding AST should be the input AST A as shown in Figure 4. To get layout information from C, we should discover the corresponding parts between A and C.

C2A is a function that converts the input CST to its corresponding AST with the help of input SDD. Algorithm 1 is the pseudo-code in JAVA style to *C2A*, which is one step of semantic analysis. To convert a CST to an AST, we should search the root of the CST in SDD to match the left sides of the productions. Then, it is converted to an AST node; this step is recursively performed on all root's children. SDD determines the appearance of the AST.

In Algorithm 1, C refers to the root of the input CST. Here it is the definition of *C2A*. First, we should check whether C can produce a new AST node by the definition of SDD. If C is null or it cannot produce an AST node, the function will return null. If C is a node that will produce a new AST node by SDD, then we can get its corresponding AST node (by *Convert2AST* function in Line 7), and then for all of C's children that can produce a new AST node at position *LegalPos*, we use *C2A* recursively. Finally, if

Algorithm 1 AST *C2A*(CST C, SDD S)

```
1: if C == null then
2:     return null;
3: else if C cannot produce new AST node then
4:     return null;
5: else if C can produce new AST node then
6:     List ⟨int⟩ LegalPos = FindPosition(C, S);
7:     AST temp = Convert2AST(C, S);
8:     for int i = 0; i < LegalPos.length; i++ do
9:         temp.sons[i] =
10:        C2A(C.sons[legalPos.get[i]], S);
11:        return temp;
12:    end for
13: else
14:    CST Cson = C.nextSon(S);
15:    return C2A(Cson, S);
16: end if
```

C itself cannot produce an AST node but one of its children can, then the node that C points to is the same as that child (represented by *Cson* in Line 15).

Furthermore, we use a technique to add flags in the CST and its corresponding AST when adopting *C2A*,

• If a CST node C doesn not produce a new AST node, and C is an inner node, then C points to the same AST node with all of C's children.

• If a CST node C with all its sub-nodes does not produce a new AST node, the AST node they point to is the same as C's father.

• If a CST node C produces a new AST node A, then C points to A.

The opinion is introduced in origin tracking [18]. Using this method, we can find the corresponding part between an AST and its corresponding CST directly.

A2C is the reverse process of *C2A*. Given an AST and an SDD, we can perform *A2C* to get the input AST's corresponding CST. The pseudo code of *A2C* is given in [19]. SDD plays an important role in *A2C*, because it is the rule that is used to create an AST's corresponding CST. SDD also works when matching two syntax trees, since it is the key to link them.

4.2 Corresponding part between AST and CST

To get the layout information when constructing a new parse tree, the location of the layout information is necessary. To get it, we must have the corresponding parts between an AST A and a CST C, which have already been discovered at the first step.

Since the AST and CST are constructed by different rules, we should convert the CST to an AST to discover the common parts. We can match two ASTs to find their correspondences, and we will give our method on matching two trees below in Section 6.

To achieve this, we define a procedure named *GCP*. The function $GCP(A, C)$ returns A's corresponding part in C, so that the input A must be a subtree of $C2A(C, S)$, or it will return null.

Instead of given a piece of pseudo code, we just define what *GCP* should do, since there are numerous ways to match two trees. We can match two trees by the method mentioned in Section 6, as well as other reasonable methods. Two propositions are necessary to build GCP. This function should satisfy the following two properties:

Proposition 1. Given a CST C, which is defined by SDD S, we have $GCP(C2A(C, S), C) = C$.

Proposition 2. Given an AST A, and a CST C, they are defined by SDD S, if A is a subtree of $C2A(C, S)$, then $C2A(GCP(A, C), S) = A$.

Function *GetAST*(AST A_1, AST A_2) that is defined in Algorithm 2 is used to find the corresponding parts between A_1 and A_2, and function *GetCST*(AST A, CST C, SDD S) in Algorithm 3 is used to discover A's corresponding parts in C. In function *GetCST*, we convert C to an intermediate AST A_C

by using *C2A*, then we use *GetAST* to find the corresponding parts between A and A_C. With the result above, we use *GCP* to find A's corresponding parts in C and return them.

Algorithm 2 AST *GetAST*(AST A_1,AST A_2)

```
1: N is a subtree of A_2;
2: if A_1 == N&&N ∈ A_2 then
3:     return N;
4: else if A_1 ≈ N&&N ∈ A_2 then
5:     return N.root;
6: else
7:     return null;
8: end if
```

Algorithm 3 CST *GetCST*(AST A, CST C, SDD S)

```
1: return GCP(GetCST(A, C2A(C, S)), C);
```

Given the above functions, the following properties must be satisfied:

Proposition 3 ([19]). Given an AST A and the SDD S that it should obey, we have

$$C2A(A2C(A,S),S) = A.$$

Lemma 1. Given an AST A, we have

$$GetCST(A,A) = A.$$

Proof. With the definition of *GetAST* in Algorithm 4, since $A == A$ and $A \in A$, the result is A.

Lemma 2. Given a CST C and the SDD S that it should obey, we have

$$GetCST(C2A(C,S),C,S) = C.$$

Proof. By Algorithm 3, we have that

$$GetCST(C2A(C,S),C,S) = GCP(GetCST(C2A(C,S),C2A(C,S)),C,S).$$

It is equal to $GCP(C2A(C,S),C)$ by Lemma 1, and $GCP(C2A(C,S),C)$ is equal to C by Proposition 1.

Lemma 3. Given a CST C and the SDD S it should obey, we have

$$C2A(GetCST(GetCST(A,C2A(C,S)),C,S)) = GetCST(A,C2A(C,S)).$$

Proof. We use GA to replace for $GetCST(A,C2A(C,S))$.

$C2A(GetCST(GetCST(A,C2A(C,S)),C,S))$ is equal to $C2A(GetCST(GA,C,S))$, and then equal to $C2A(GCP(GetCST(GA,C2A(C,S)),C))$ with the help of Algorithm 3. GA is a part of $C2A(C,S)$, which means that $GetCST(GA,C2A(C,S))$ is equal to GA, and thus $C2A(GCP(GetCST(GA,C2A(C,S)),C))$ is equal to $C2A(GCP(GA,C))$.$C2A(GCP(GA,C))$ is equal to GA. At last, $C2A(GCP(GA,C))$ is the same with $GetCST(A,C2A(C,S))$.

Proposition 3 implies that given an AST A and an SDD S, if we convert A to a CST by *A2C* and perform the reverse process by *C2A* on the CST, then we can get the original AST A. Lemma 1 means that if we match two ASTs that are the same, then the algorithm will return one of them. Lemma 2 implies that if we match a CST C and its corresponding AST, we will get C. Lemma 3 is more complex. We use A_C to substitute for the corresponding AST of C. $GetCST(A,C2A(C,S))$ returns the A's corresponding parts in A_C, and we use A' to represent it. Lemma 3 means that if we convert the result CST of $GetCST(A',C,S)$ to an AST, the result AST is equal to A'.

With the help of these properties, we will later prove that *C2A* and *putback* function constitute a well-behaved bidirectional transformation.

The process to discover the corresponding parts between A and C is shown in Figure 5. The nodes with the same color indicate that they get the same flag when using the *C2A* function.

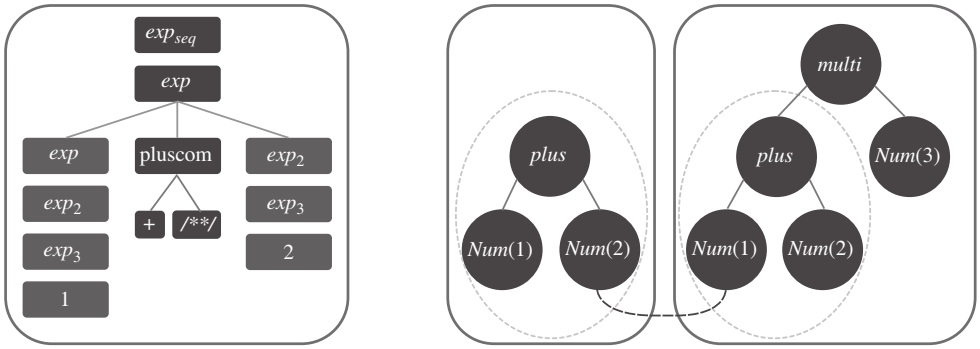

Figure 5 The process to find corresponding parts between C and A.

4.3 Constructing a new CST

After obtaining the corresponding parts between CST C and AST A, which are the arguments of *putback*, we can continue constructing a new CST with the help of their relationship.

Given an AST A, we traverse its nodes from top to bottom. If we meet an AST node in A that has a corresponding parts in C, we add the corresponding parts directly when constructing the new CST. When we meet a node in A that does not have a corresponding part in C, we add $A2C(A, S)$ to the constructed CST.

The process of discovering corresponding parts and constructing a new CST are shown below in Algorithm 5. This function may construct an unreasonable CST, and the *Modify* function will give us hint as to how to amend it to a reasonable CST as the final output of *putback*.

Algorithm 4 CST *Putback*(AST A, CST C, SDD S)

```
return Modify(Construct(A, C, S), S);
```

Algorithm 5 CST *ConstructCST*(AST A, CST C, SDD S)

```
if GetCST(A, C2A(C, S)) == A then
    return GetCST(A, C, S);
else if GetCST(A, C2A(C, S)) ≈≈ A then
    CST temp = GetCST(A.root, C, S);
    List⟨int⟩ LegalPos = FindPosition(temp, S);
    for int i = 0; i < LegalPos.length; i++ do
        temp.sons[LegalPos.get(i)] =
        ConstructCST(A.sons[i], C, S);
    end for
    return temp;
else
    CST temp = A2C(A.root);
    for int i = 0; i < A.sonsCount; i++ do
        ConstructCST(A.sons[i], C, S);
    end for
    return temp
end if
```

4.4 Modifying unreasonable CSTs

The CST produced by *ConstructCST* may not be reasonable, as illustrated in Figure 6. A technique for amending it is necessary. The function that performs amending is named after *Modify*, and it has two parameters: one is a CST C while the other is an SDD S. First, we check the root of C to see whether any errors have occured. An error means that a CST node and its children do not obey the SDD. If

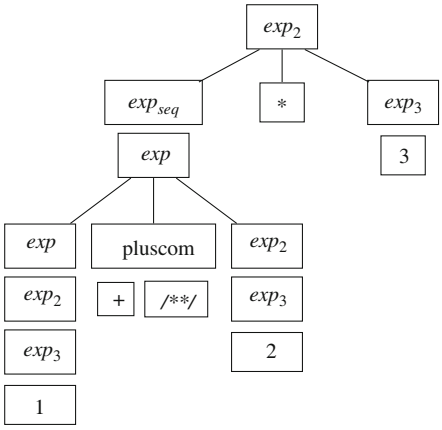

Figure 6 The CST after constructing.

there is no error, the method performs checking on C's children recursively. If there is something wrong with one of C's children, we use the *Link* function to link C to the reasonable CST node. The function *Patch*(CST C, SDD S) is used to "patch" all C's children.

Algorithm 6 CST *Modify*(CST C, SDD S)

```
1: if C has no children then
2:     return Patch(C, S);
3: end if
4: for int i = 0; i < C.size(); i++ do
5:     if legal(C, C.sons[i], S) then
6:         C.sons[i] = Modify(C.sons[i], S);
7:     else
8:         C.sons[i] =
9:         Link(C, i, Findtarget(C.sons[i]), S);
10:    end if
11: end for
12: return C;
```

Algorithm 7 CST *Patch*(CST C, SDD S)

```
1: Adding legal children to C that donnot produce new ASTs;
2: if C has no child then
3:     return C;
4: else
5:     for all son[i] that son[i] ∈ C.sons do
6:         son[i] = Patch(son, S);
7:     end for
8: end if
9: return C;
```

The *Link* function is used to connect a CST Node with one of its children. It is defined as CST *Link*(CST C, int *pos*, List *target*, SDD S). *pos* is the location of the unreasonable child of C and *target* is a list of CST nodes, which are the objectives for C to link. In Figure 6, exp_2's left node violates the rule, so when we use *Modify*(exp_2, S), *Link*($exp_2, 1, [exp_{seq}, exp], sdd$) will be called to link exp_2 and the target $[xp_{seq}, exp]$. The *Link* function works as below. At first, if input C is null, the method will return null, else it will find every possible child of C in position *pos* (*pos* represents a concrete number, in Lines 4 and 5). If a possible child *PoCST* belongs to the target (in Line 10), then we set *PoCST* equal to the target's corresponding node and patch C(*PoCST*'s father)'s other children (Line 11,12). If a *PoCST* of C does not match the target, we should use *Link*(*PoCST*, *pos*, *target*, S) recursively, while *pos* here means the possible position of *PoCST*'s children to link with the target. In the example in Figure 6, we can get

the correct leftmost node of exp_2 by the SDD in Table 1, then $exp_2 \to exp_3 \to exp$, and we find exp will match the target ($[exp_{seq}, exp]$).

Algorithm 8 CST *Link*(CST C, int *pos*, List *target*, SDD *sdd*)

```
1: if C == null then
2:     return null;
3: end if
4: List Possible_CST =
5: Check_SDD_no_New_AST(C, pos, sdd);
6: if Possible_CST == null then
7:     return null;
8: end if
9: for CST PoCST:Possible_CST do
10:    if PoCST ∈ target then
11:        for CST patcher : C.sons do
12:            patcher = Patch(patcher, sdd);
13:        end for
14:        PoCST = target.find(PoCST);
15:        return null;
16:    end if
17:    ListLegal_pos =
18:    Find_sons_pos_no_new_AST(PoCST, sdd);
19:    for int pos : Legal_pos do
20:        PoCST.sons[pos] =
21:        Link(PoCST, pos, target, sdd);
22:    end for
23: end for
24: if C.sons[pos] == null then
25:    return null;
26: end if
27:
28: return C;
```

By using *Modify*, we can get a reasonable CST that fits the SDD rules. *Modify* function has such properties.

Lemma 4. For a reasonable CST C and its SDD S, we have

$$Modify(C, S) = C.$$

Proof. For a reasonable CST C, $Modify(C, S)$ will always go through Line 6 in Algorithm 6, until meeting the terminals of C. Since $Patch(c, S) = c$, while C is a terminal, $Modify(C, sdd)$ will not change C when C is reasonable.

5 A well-behaved bidirectional transformation

As shown in Subsection 2.3, a BX that consists of `put` and `get` functions is well-behaved if it satisfies the following laws:

$$\texttt{GetPut law} : \texttt{put}\ s\ (\texttt{get}\ s) = s;$$
$$\texttt{PutGet law} : \texttt{get}\ (\texttt{put}\ s\ v) = v.$$

The `GetPut` property requires that if there is no modification on the view, then there should not be any modification on the source if we put it back. The `PutGet` property requires all modifications on the view to be completely reflected to the source so that the changed view can be computed again by applying the forward transformation to the changed source [20].

If CST is treated as source while AST as view, the *C2A* function can be considered as the forward transformation while the *putback* function as the backward transformation. *C2A* and *putback* constitute a bidirectional transformation. The BX they make up is well-behaved.

To prove the bidirectional transformation satisfies the `GetPut` law, we have

Theorem 1.

$$Putback(C2A(C,S),C,S) = C.$$

Proof. $Putback(C2A(C,S),C,S)$ is equal to $Modify(ConstructCST(C2A(C,S),C,S))$ by the definition of $PutBack$ in Algorithm 4. Through Line 1, Line 2 in Algorithm 5, and the truth that $C2A(C,S)$ is same to $C2A(C,S)$, $Modify(ConstructCST(C2A(C,S),C,S))$ is equal to $Modify(GetCST(C2A(C,S),C,S)$, which is the same as $Modify(C,S)$ by lemma 2. Because C is a reasonable CST, $Modify(C,S)$ is just the same as C; thus the `GetPut` law holds.

To prove the bidirectional transformation satisfies the `PutGet` law, we have

Theorem 2.

$$C2A(Putback(A,C,S)) = A.$$

Proof. $C2A(Putback(A,C,S))$ is equal to $C2A(Modify(ConstructCST(A,C,S)))$ by the definition of $PutBack$ in Algorithm 4. $C2A(Modify(ConstructCST(A,C,S))) = C2A(ConstructCST(A,C,S))$ holds since *Modify* does not change the input CST's corresponding AST.

If $A \subseteq C2A(C,S)$, then $GetCST(A, C2A(C,S))$ is equal to $C2A(GetCST(A,C,S))$ by Lemma 3, and thus $C2A(GetCST(A,C,S)) = GetCST(A, C2A(C,S)) = A$.

If $A \approx\approx C2A(C)$ or $A \neq C2A(C)$ holds, which implies A does not belong to $C2A(C,S)$, and by the definition of $ConstructCST$ in Algorithm 5, we get $C2A(ConstructCST(A,C,S).root,S) = A.root$. For every AST A' and CST C', $C2A(ConstructCST(A',C',S).root) = A'.root$. Because $ConstructCST$ is recursively defined, $C2A(ConstructCST(A,C,S)) = A$, which implies the `PutGet` law holds.

6 Additional remarks

Handling code fragments written in a language without SDD is different with the algorithm we present in this paper. For example, XML. It is not necessary to create an XML tree from context-free grammar, and we do not need to convert an XML tree (i.e., an XML document tree) to an AST. Although the XML tree is not generated by grammar, without loss of generality we call it the XML CST. Instead of converting the input CST to an AST and then matching it with the input AST (i.e., the transformed AST), we just need to match the input XML CST with the transformed XML CST. The key process in the implementation is the tree-matching method, and we will explain the process below.

XML can be converted to a CST by tools such as Dom and the layout information will be removed. After pretreatment, a CST with no layout information can be obtained. It is named source CST (SCST), and then we match the transformed CST (TCST) with it by using the tree-matching method recursively. We use the tree-matching method when we want to get the corresponding parts between two trees. There are also some methods to match two trees, and we use a method that we developed to deal with the possible nested tree nodes in XML trees as well as in AST matching, presented above. The function $GetAST$ in Section 4 is produced by this tree matching method. The name of the elements can be the same in an XML tree, even a child's name and its father node's name can be the same. It is really a challenge when performing matching on such two trees, because a node in TCST may have more than one corresponding node in the SCST.

Definition 2. Two XML elements A and B without any child element are the same if their label and attributes are the same. For two elements, A with children elements $[a_1, a_2, \ldots, a_n]$ and B with children elements $[b_1, b_2, \ldots, b_n]$, if A's label and attributes are equal with B's, and $a_1 = b_1, a_2 = b_2, \ldots, a_n = b_n$ recursively, we name them as the same.

If their label and attributes are the same while their child elements are not the same, we define them as the approximated.

Aside from the definition of the same and approximated, we also define the similarity between two tree nodes.

Definition 3. If two tree nodes A and B with children $[a_1, a_2, \ldots, a_m]$ and $[b_1, b_2, \ldots, b_n]$ separately are approximated, the similarity between them is the amount of $[a_i, b_j]$, where a_i and b_j are approximated and one node can only exist in one pair.

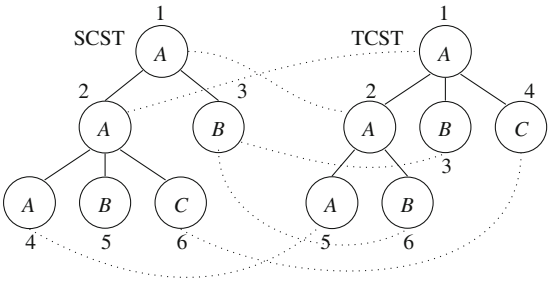

Figure 7 Results of tree-matching.

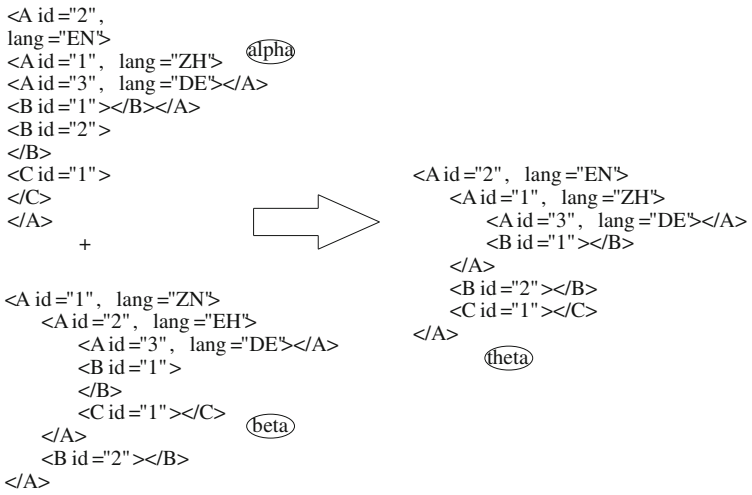

Figure 8 The XML example.

The process of tree-matching is shown below. XML is used to illustrate the method:

(1) For TCST's root node, we traverse SCST to find if there exists a same node in SCST. If it is true, then match it, else find every approximated node in SCST in order to determine the most approximated node. In Figure 7, nodes 1, 2, and 4 in the SCST are approximated with the root node in TCST, because they have the same signature (labels are all A and they have no attribute).

(2) If a node N in TCST does not have the same node in SCST but has many approximated nodes, count the similarity between it and all its approximated nodes in TCST, and then match the node that has the largest similarity. If many nodes have the same similarity with N, choose one of them to match.

(3) If a node N in TCST does not have the same node in SCST and additionally does not have the approximated nodes, then it matches no nodes.

The results of tree-matching are shown in Figure 7.

Using the results of the tree-matching method, we can produce a new XML CST with layout information. The process to produce a new CST can be seen as a sub-method of *putback* because it does not have to convert between a CST and an AST.

Figure 8 is an example illustrating our method. XML `alpha` is a piece of XML that has beautiful appearance, while XML `beta` is rather chaotic. We use our method *putback* to recreate `beta`. We then use the tree matching method presented in Section 6 to obtain XML `theta`, which has a similar appearance as `alpha`.

7 Conclusion

In this paper, a method of program code transformation with layout information preservation was proposed and its properties were proved. Furthermore, as a case study, an analyzer of XML was also introduce to illustrate our methodology.

Indeed, the method has some shortcomings. To develop a robust method, we should not only focus on how to produce the resulting CST, but also how to store and attach the layout information to a CST.

A number of further studies are planned based on the groundwork presented in this paper, including the extension our algorithm to deal with the source code directly, instead of a CST with layout information, to extend the tree match method to handle more syntax, such as "*while...do*" and "*if...else*" sentences, and to implement a tool using this method to achieve our goal.

Acknowledgements

This work was supported by National Natural Science Foundation of China (Grant Nos. 61472240, 91318301, 61261130589).

References

1 Van De Wanter M L. Preserving the documentary structure of source code in language-based transformation tools. In: Proceedings of 1st IEEE International Workshop on Source Code Analysis and Manipulation, Florence, 2001. 131–141
2 Erdweg S, Rendel T, Kästner C, et al. Layout-sensitive generalized parsing. Lect Notes Comput Sci, 2013, 7745: 244–263
3 Teltelman W, Goodwin J W, Bobrow D G. Interlisp reference manual. Xerox Palo Alto Research Centers, 1978
4 Sandewall E. Programming in an interactive environment: the "Lisp" experience. ACM Comput Surv, 1978, 10: 35–71
5 Fritzson P. Towards a distributed programming environment based on incremental compilation. Dissertation for the Doctoral Degree. Linkopings Universitet, 1984
6 Fritzson P. Symbolic debugging through incremental compilation in an integrated environment. J Syst Softw, 1983, 3: 285–294
7 Jonge M D, Visser E. An algorithm for layout preservation in refactoring transformations. Lect Notes Comput Sci, 2012, 6940: 40–59
8 Brown C. Tool Support for Refactoring Haskell programs. Dissertation for the Doctoral Degree. Canterbury: University of Kent, 2008
9 Kumar R, Talton J O, Ahmad S, et al. Flexible tree matching. In: Proceedings of 22nd International Joint Conference on Artificial Intelligence. Reston: AAAI Press, 2011. 2674–2679
10 Fischer S, Hu Z J, Pacheco H. "Putback" is the essence of bidirectional programming. Technical Report GRACE-TR 2012-08, GRACE Center, National Institute of Informatics, 2012
11 Vaucher J G. Pretty-printing of trees. Softw Pract Exp, 1980, 10: 553–561
12 Merijn D J. Pretty-printing for software reengineering. In: Proceedings of the International Conference on Software Maintenance, Montréal, 2002. 550–559
13 Neamtiu I, Foster J S, Hicks M. Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Softw Eng Notes, 2005, 30: 1–5
14 Foster J N, Greenwald M B, Moore J T, et al. Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans Program Lang Syst, 2007, 29: 17
15 Aho A V, Lam M S, Sethi R, et al. Compilers: Principles, Techniques, and Tools. 2nd ed. New York: Addison-Wesley, 2006
16 Donnelly C, Stallman R M. Bison Manual: Using the YACC-compatible Parser Generator. 7th ed. Free Software Foundation, 2002
17 Visser E. Syntax definition for language prototyping. Dissertation for the Doctoral Degree. Amsterdam: University of Amsterdam, 1997
18 Vaucher J G, Klint P, Tip F. Origin tracking. J Symb Comput, 1993, 15: 523–545
19 Wang L C. Constructing format-preserving printing from syntax-directed definition. Dissertation for the Master Degree. Shanghai: Shanghai Jiao Tong University, 2015
20 Danvy O. Functional unparsing. J Funct Program, 1998, 8: 621–625