Deriving Event-Based Document Transformers from Tree-Based Specifications *

Keisuke NAKANO   Susumu NISHIMURA
Research Institute for Mathematical Sciences, Kyoto University
{ksknac,nisimura}@kurims.kyoto-u.ac.jp

Abstract

The document transformation technology is getting significant, as the increasing amount of data are stored and exchanged in structured document formats. The conventional tree-based transformers have an inherent overhead for constructing an intermediate document tree for each input document. The *event-based* document transformers, in contrast, can skip the intermediate constructions by processing a document simply as a stream of parsing events.

This paper proposes an algorithm which derives an event-based transformer from a given specification of a document transformation. We develop such an algorithm in the framework of attribute grammars. The transformation program is obtained by first applying a deforestation method and then to derive an attribute evaluation scheme relevant to the event-based transformation. We show the feasibility of our algorithm by a few typical document transformations.

Keyword: Event-based document transformation, Structured documents, Attribute evaluation scheme, Descriptional composition, Attribute grammars.

1 Introduction

Structured documents are widely used for representing a hierarchical data structure in a conventional text format. There are various structured document formats, *e. g.*, HTML for Web presentation, LaTeX for typesetting, and most notably the XML (Extensible Markup Language) standard [1], which is intended to serve as a common data representation for seamless data exchange among multiple platforms. Due to the increasing amount of data exchange in structured document formats, the technology for transforming structured documents is getting more significant.

A conventional scheme for transforming documents is a *tree-based* transformation one, where a transformation is defined as an operation over *document trees*, which model the

*This is a preliminary version of a paper which will appear in the proceedings of "Workshop on Language Descriptions, Tools and Applications — LDTA 2001" hierarchical structure of structured documents. A tree-based transformation once constructs a document tree of the input document on the memory, manipulates the document tree, and translates back the transformed document tree into the result document. Due to its high expressibility for describing general document transformations, the tree-based transformation scheme is widely used in practice (*e. g.*, an XML transformation language XSLT [2] and Document Object Model (DOM) [3]).

The tree-based transformation scheme, however, has a drawback that it must once load the entire document on the memory before starting tree manipulations. This indicates that a tree-based transformation can only deal with documents whose size fits into the actual memory size. This could be a sever problem, when the size of the input document is very large, or when the size of the memory is relatively small.

For the purpose of relaxing the strain on the memory usage, Simple API for XML (SAX) [4] has been proposed as an alternative way for programming document transformations. The SAX API, instead of constructing a document tree, creates a stream of *parsing events* (or *tokens* in the terminology of the formal language theory) to notify the backend program of what syntactic objects are encountered while it reads the input XML document. For example, when the SAX reads the following document

```
<message> <body> I like sweets. </body> </message>
```

it notifies the following parsing events: a beginning of a `message` tag, a beginning of a `body` tag, a string "`I like sweets.`", end of a `body` tag, and finally end of a `message` tag. A program using the SAX API processes a document in an event-based fashion, *i. e.*, each parsing event is captured by an event handler, which is responsible for processing the event.

Here we are faced with a tension between efficiency and expressiveness. Processing documents in the event-based fashion may dramatically reduce the memory usage, especially for simple document transformations. On the other hand, it has a less expressive power than a tree-based transformation. There is no vivid line that divides between the two. There is only a general guideline that the event-based API should be used for those relatively simple transformations which can be processed by a one-pass read of the event stream. In addition, the event-based transformation has another drawback that it makes difficult to maintain programs: Program codes are scattered into small actions responding to each parsing events, and hence even adding a small change in the original document transformation program could be very cumbersome.

This paper proposes an algorithm for automatically deriving an event-based document transformation from a given tree-based document transformation. It would be very useful if an event-based document transformation is obtained automatically. An automatic derivation enables us to specify a document transformation in a tree-based style and to obtain an event-based transformation from the specification. Hence we obtain efficiency and expressiveness at the same time. Furthermore, tree-based specifications for document transformation would be much easier to maintain than event-based document transformation programs.

$$
\begin{tikzcd}
\mathit{EventStream} \arrow[r, "\text{event-based transformation}"] \arrow[d, "\mathit{Parse}"'] & \mathit{EventStream} \\
\mathit{DocumentTree} \arrow[r, "T"'] & \mathit{DocumentTree} \arrow[u, "\mathit{Unparse}"']
\end{tikzcd}
$$

Figure 1: A diagram for document transformation

1.1 Deforestation methods

We can regard the present problem of deriving an event-based program as a *deforestation* problem [17], which eliminates intermediate data which are produced and consumed between two successive computations. A tree-based transformation can be modeled as a composition of three functions, as illustrated by the diagram in Figure 1. First, a function *Parse* translates a given event stream to an intermediate tree representation. The intermediate document tree is then processed by a function T, which is responsible for the tree transformation. Finally, a function *Unparse* translates back the resulting document tree into the corresponding event stream. (In this paper, we omit a process for generating a parsing event stream from a character stream of an input document or vice versa, which is easily achieved by applying the standard lexical analysis algorithm [6].) The tree-based transformation is represented by a composite function $\mathit{Unarse} \circ T \circ \mathit{Parse}$, and a derivation of an event-based transformation is nothing but to find an equivalent shortcut function (the upper-most arrow in the diagram), which generates no intermediate document trees.

There are several deforestation methods based on different formalisms, *e. g.*, those on functional programming [17, 15, 16] and attribute grammars [10, 11, 7, 8, 9]. Amongst them, we apply a deforestation method based on the formalism of attribute grammars. Attribute grammar (AG) was introduced by Knuth [14] as a formalism for specifying general purpose syntax-directed computation. In an AG, a set of *attributes* is attached to each (terminal or non-terminal) symbol of the underlying grammar, and a set of *semantic rules*, which specify how attributes are computed from other attributes, is given for each production rule of the grammar. A transformation from a term of a grammar L_1 to a term of another one L_2 can be defined by an AG over L_1 whose attribute domain is L_2. Ganzinger and Giegerich called such AGs attribute coupled grammars [10, 11].

Let us write $G \circ F$ to denote a composition of two AGs, where the output of F is processed by G. Ganzinger and Giegerich [10] have presented an AG deforestation method, called *descriptional composition*, which derives a single deforested AG from a given composition $G \circ F$ of two attribute coupled grammars. Following Ganzinger and Giegerich, several extensions and refinements to the descriptional composition method have been proposed [7, 8, 9].

The descriptional composition scheme has an advantage over the functional deforestation methods. In the process of deforesting a functional program, it is vital to find a site to which a transformation rule applies. However, the transformation rule applies only if certain semantic equalities over the program hold, and the discovery of a site which satisfies such semantic equalities is not easily automated. In contrast, the transformation process in the descriptional composition scheme is carried out in a purely syntax-directed way and is amenable to full automation.

1.2 Deriving one-pass interactive event-based programs

We can derive a deforested AG from an event-based document transformation program from a compositional specification of the form $UnParse \circ T \circ Parse$. However, it solely does not solve the present problem.

First, the function *Parse* in Figure 1 is a partial function, but a partial function cannot be defined in the traditional formalism of AGs, in which all attributes must be defined. The partiality is essential in defining *Parse*: If the input stream is syntactically corrupted (*e. g.*, because of unbalanced tags), the result should be undefined. This glitch can be easily fixed by allowing AGs to have *undefined* attributes, where an undefined attribute represents an error in document transformation. The undefined attributes are permissible, with a minor modification to the original descriptional composition method [11, pp. 378]. The partial function *Parse* is specified by an AG, whose result is undefined if the input is corrupted (See Figure 6). The undefined attributes also play a crucial role in the derivation of event-based programs.

Second, more fundamentally, the resulting event-based program should be a one-pass interactive program. An event-based program is called a *one-pass interactive* program if it traverses the stream of the input parsing events only once and it writes to an output stream simultaneously as it reads from the input stream, responding to each parsing event. To obtain such a one-pass interactive program, we need to find a method for deriving a relevant attribute evaluation scheme from the result of a descriptional composition. The main difficulty in the derivation is that an attribute value for a parsing event may depend on the attribute values of parsing events that will follow. We call such dependencies *forward dependencies*. (Even a simple identity document transformation results in an AG whose semantic rules contain forward dependencies. See Section 3.2.1 for details.)

In this paper, we propose an algorithm which systematically derives a one-pass interactive event-based program from an AG specification. To achieve this, we divide each attribute evaluation rule into two parts: attribute dependency and value construction. The former can be statically computed, by finite lookaheads into the input event stream, and it turns out to range over a finite set of dependency patterns. We can therefore model an interactive evaluation scheme by a finite state transition machine, where the state space is the set of dependency patterns and the state machine makes a transition every time it reads a parsing event from the input stream. We need not care about forward dependencies any more, since the state transition is subject to the dependency pattern, but not to individual attribute dependencies. The remaining part of attribute evaluation, *i. e.* the value construction, is dynamically computed for each transition, by letting each attribute hold the partial result of attribute evaluation so far. The output of transformation is obtained interactively, by letting the event-based program emit the definite part of the output for each transition. The technical contribution of this paper is to give a decidable and terminating algorithm which produces such a state transition machine from an AG specification.

Note that there are some document transformations which are not inherently interactive: they require most of (or all of, at the worst) the input event stream to be buffered temporarily until the very end of the input. For such programs our algorithm does not work effectively. However, a large class of simple document transformations does not go to extremes, *i. e.*, usually only several parsing events need to be buffered at the same time.

The rest of the paper is organized as follows: Section 2 defines the present problem in a more formal manner, by giving the definitions of the data model and the AG framework. In Section 3, we present our algorithm through a simple example. Section 4 gives several examples of program derivations. Finally, Section 5 concludes the paper, with a discussion of future investigations.

2 AGs for Document Transformations

In this section, we specify the domain of the problem stated in the previous section in a more formal way, together with some preliminary definitions.

2.1 The data model

The structured document format we deal with in the present paper is an XML-like markup language which consists of (i) an arbitrary, but fixed, finite set of tag names for markup, and (ii) named start tags and unnamed end tags. For example, the following is a structured document in our format.

```
<a>
   <b> <a></> <d></> </>
   <c> </>
</>
```

Each <*tagname*> represents a start tag named *tagname*, and `</>` an end tag, which has no tag name. A document is *well-formed*, if start tags and end tags are balanced in the usual sense. (Unlike XML, tag names are omitted from end tags, since each end tag in a well-formed document has a unique corresponding start tag.) The *depth* of a well-formed document is the maximum level of nesting, with the outermost nest being level 1. The depth of the above document is 3.

The document format presented here should be understood as an idealized one for the subsequent theoretical study. It is missing many useful features in real document formats, but the authors believe that it would be possible to incorporate them into the framework to be presented in the paper. We leave the topic to a future investigation.

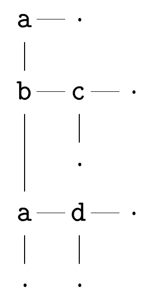

Figure 2: A binary tree representation of a document

$$
\begin{array}{rcl}
S_{\mathit{EventStream}} & ::= & E \\
E & ::= & \texttt{Begin\_tag}_1\ E \ \mid\ \cdots\ \mid\ \texttt{Begin\_tag}_n\ E \ \mid\ \texttt{End}\ E \ \mid\ \texttt{Nil} \\
S_{\mathit{DocuemntTree}} & ::= & D \\
D & ::= & \texttt{Node\_tag}_1\ D\ D \ \mid\ \cdots\ \mid\ \texttt{Node\_tag}_n\ D\ D \ \mid\ \texttt{Empty}
\end{array}
$$

Figure 3: Grammar definitions

We model document trees, the internal representation of structured documents, by binary trees. A structured document in the conventional text format can be represented by a binary tree, where each tree node, corresponding to a matching pair of a start tag and an end tag, is placed subject to the following constraints.

1. The left child of a tree node represents the node of the left-most element directly enclosed by the parent's tags. If the parent's node has no enclosing tags, the left child is an empty node.

2. The right child of a tree node represents the node of a sibling element, which immediately follows the parent's end tag. If there is no such element, the right child is an empty node.

For example, the above document can be represented by a binary tree in Figure 2.

Suppose $\texttt{tag}_1$, $\texttt{tag}_2$, ..., $\texttt{tag}_n$ be the fixed set of tag names. Figure 3 gives the grammar of event streams and document trees in BNF. The non-terminals $S_{\mathit{EventStream}}$ and $S_{\mathit{DocumentTree}}$ are start symbols. (We omit the subscripts when it is obvious from the context.) Each constructor $\texttt{Begin\_tag}_i$ corresponds to a start tag named $\texttt{tag}_i$, `End` to an end tag, and `Nil` to the end of an event stream. Similarly, each constructor $\texttt{Node\_tag}_i$ corresponds to a tree node named $\texttt{tag}_i$, and `Empty` to an empty node. For example, the above example document is represented by the following event stream expression.

```
Begin_a (Begin_b (Begin_a (End (End (Begin_c (End (End Nil)))))))
```

S: Start symbol, $E, \ldots$: Non-terminal symbols,
$a, \ldots$: Attribute names, $\mathtt{C}, \ldots$: Data constructors

$$
\begin{array}{lll}
AG & ::= & \mathtt{let}\ F = \{S \to E:\ \Sigma\}\{E \to \mathtt{C}\ \overline{E}:\ \Sigma\}^* \\
\Sigma & ::= & (occ = exp)^* \\
occ & ::= & E.a \mid S.result \\
exp & ::= & occ \mid \mathtt{C}\ \overline{exp}
\end{array}
$$

Figure 4: Attribute grammar notation

$$
\begin{array}{lll}
\mathtt{let}\ rev\ = & & \\
\ S \to E: & S.result = E.s & E.h = \mathtt{Nil} \\
\ E \to \mathtt{Begin\_a}\ E': & E.s = E'.s & E'.h = \mathtt{Begin\_a}\ E.h \\
\ E \to \mathtt{End}\ E': & E.s = E'.s & E'.h = \mathtt{End}\ E.h \\
\ E \to \mathtt{Nil}: & E.s = E.h &
\end{array}
$$

Figure 5: Attribute grammar *rev*

2.2 Attribute grammars

Our attribute grammar (AG) notation is given in Figure 4. An AG F is defined by a list of pairs $prod : \Sigma$ of a *production rule prod* of the underlying grammar and its associated set Σ of *semantic rules*. In the present paper, we restrict a production rule to have the form either $S \to E$ or $E \to \mathtt{C}\ E_1\ \cdots\ E_k$, where S is the start symbol, $E, \ldots$ are non-terminal symbols, and $\mathtt{C}$ is a data constructor. Each semantic rule in Σ has the form $occ = e$, where occ is an *attribute occurrence* $E.a$, which denotes the value of attribute a attached to the non-terminal symbol E, and e is an *expression*, which defines what value is assigned to the occurrence occ. We only allow those expressions which are either a reference to an attribute occurrence or a data construction $\mathtt{C}\ e_1\ \ldots\ e_k$. We assume that a special synthesized attribute *result* is defined only for the start production $S \to E$ to designate the overall result of attribute evaluation.

For example, an AG *rev* over the event stream, whose only tag name is `A`, is defined in Figure 5. It reverses the order of the input stream of parsing events.

Note that, in contrast to traditional AGs, we allow some attributes of a production rule to have no corresponding semantic rule. We may explicitly write such an undefined semantic rule as $E.a = undef$, which reads "the value of the attribute a on the symbol E is undefined". If the value of one of expressions $e_1, \ldots e_n$ is undefined, then the value of an expression $\mathtt{C}\ e_1\ \ldots\ e_n$ is also undefined. Furthermore, throughout the rest of the paper, we consider only those AGs which satisfy the following conditions.

- It is *noncircular* [14].
- It satisfies *quasi-SSUR* condition.

$$
\begin{array}{l|l}
\begin{array}{l}
\texttt{let}\ \mathit{parse}^{(d)}\ = \\
\quad S \to E: \\
\qquad S.\mathit{result} = E.\mathit{parse} \\
\quad E \to \texttt{Begin\_tag}_1\ E': \\
\qquad E.\mathit{parse} = \texttt{Node\_tag}_1\ E'.\mathit{parse}\ E'.\mathit{stack}_1 \\
\qquad E.\mathit{stack}_1 = E'.\mathit{stack}_2 \\
\qquad\qquad \vdots \\
\qquad E.\mathit{stack}_{d-1} = E'.\mathit{stack}_d \\
\qquad E.\mathit{stack}_d = \mathit{undef} \\
\qquad\qquad\qquad \vdots \\
\quad E \to \texttt{Begin\_tag}_n\ E': \\
\qquad E.\mathit{parse} = \texttt{Node\_tag}_n\ E'.\mathit{parse}\ E'.\mathit{stack}_1 \\
\qquad E.\mathit{stack}_1 = E'.\mathit{stack}_2 \\
\qquad\qquad \vdots \\
\qquad E.\mathit{stack}_{d-1} = E'.\mathit{stack}_d \\
\qquad E.\mathit{stack}_d = \mathit{undef}
\end{array}
&
\begin{array}{l}
\quad E \to \texttt{End}\ E': \\
\qquad E.\mathit{parse} = \texttt{Empty} \\
\qquad E.\mathit{stack}_1 = E'.\mathit{parse} \\
\qquad E.\mathit{stack}_2 = E'.\mathit{stack}_1 \\
\qquad\qquad \vdots \\
\qquad E.\mathit{stack}_d = E'.\mathit{stack}_{d-1} \\
\quad E \to \texttt{Nil}: \\
\qquad E.\mathit{parse} = \texttt{Empty} \\
\qquad E.\mathit{stack}_1 = \mathit{undef} \\
\qquad\qquad \vdots \\
\qquad E.\mathit{stack}_d = \mathit{undef}
\end{array}
\end{array}
$$

Figure 6: An AG for parsing

An AG satisfies *SSUR (syntactic single use requirement)* condition [11], if every different occurrence of a *syntactic attribute* is referred to exactly once in the defining expressions of the semantic rules for every production rule. An attribute is *syntactic* if its value ranges over the set of terms of a grammar. An AG satisfies quasi-SSUR condition, if the *exactly once use* restriction is relaxed to *at most once use*.

Note that the relaxation of the original SSUR condition is due to the existence of undefined attributes. This is legitimate up to a minor modification to the original descriptional composition method [11, pp. 378], and it shall also preserve the fundamental properties such as the closure property [11, Soundness Theorem].

2.3 AGs for parsing and unparsing

Now we define AGs for parsing and unparsing, which correspond to functions *Parse* and *Unparse*, *resp.*, in Section 1. In the following, we assume that the grammar of event streams and document trees are those defined in Section 2.1.

To define an AG for parsing, we assume that every input document, if it is well-formed, has a depth of a fixed number $d (d > 0)$ or less. Though this may seem restrictive, a large portion of document transformations used in practice can be covered by those transformations

$$
\begin{array}{l|l}
\begin{array}{l}
\texttt{let}\ \mathit{unparse}\ = \\
\ S \to T: \\
\qquad S.\mathit{result} = T.\mathit{unparse} \\
\qquad T.\mathit{acc} = \texttt{Nil} \\
\ T \to \texttt{Node\_tag}_1\ T_1\ T_2: \\
\qquad T.\mathit{unparse} = \texttt{Begin\_tag}_1\ T_1.\mathit{unparse} \\
\qquad T_1.\mathit{acc} = \texttt{End}\ T_2.\mathit{unparse} \\
\qquad T_2.\mathit{acc} = T.\mathit{acc}
\end{array}
&
\begin{array}{l}
\qquad\qquad \vdots \\
\\
T \to \texttt{Node\_tag}_n\ T_1\ T_2: \\
\qquad T.\mathit{unparse} = \texttt{Begin\_tag}_n\ T_1.\mathit{unparse} \\
\qquad T_1.\mathit{acc} = \texttt{End}\ T_2.\mathit{unparse} \\
\qquad T_2.\mathit{acc} = T.\mathit{acc} \\
T \to \texttt{Empty}: \\
\qquad T.\mathit{unparse} = T.\mathit{acc}
\end{array}
\end{array}
$$

Figure 7: An AG for unparsing

whose input documents have a bounded depth. For example, when XML documents are used for representing a database (it is one of most typical usages of XML!), the maximum depth of documents can be decided by a *database schema* [5]. Rendition languages for data presentation such as HTML and LaTeX, in contrast, do not have a finite bound to document depths. However, practically, depths of typical documents written in these languages can be bounded by a moderate number. In the present paper, we do not discuss transformations for documents of an arbitrary depth, leaving it to a future investigation.[1]

The finite-depth restriction to the input document is a key to derive a one-pass interactive event-based program. We will give an AG for parsing documents of a finite depth, which contains undefined semantic rules to flag an error when the input document has an an exceeding depth. The undefined attributes play a crucial role for eliminating forward dependencies by a finite number of static lookaheads into the parsing event stream.

Figure 6 defines an AG $\mathit{parse}^{(d)}$, which takes a document of a depth d or less and produces a document tree. (The superscript d is omitted, when it is clear from the context.) The AG $\mathit{parse}^{(d)}$ simulates a stack of a finite depth d by a set of synthesized attributes $\mathit{stack}_1, \ldots, \mathit{stack}_d$, where stack_1 is the stack top. Each attribute stack_i holds the result of parsing sibling nodes which follow the i-th outer pair of start and end tags enclosing the current event. The attribute evaluation for the AG proceeds from the end of the event stream to the beginning. The evaluation starts from the last parsing event `Nil`, with an empty parsing result, as specified by the semantic rule $E.\mathit{parse} = \texttt{Empty}$. If the event is `End`, the result of parsing the event stream that follows the `End` event is pushed onto the finite stack and the parsing continues for the inner nesting level. If the event is a start tag, a document tree node is constructed, where the left child is $E'.\mathit{parse}$, the result of parsing the event stream that follows the start tag, and the right child is $E'.\mathit{stack}_1$, the result of parsing the event stream that follows the end tag matching the start tag. The stack is popped, and the parsing

[1]A general way to obtain an AG for parsing documents of an unlimited depth would be to introduce a stack data structure and to allow semantic rules to manipulate stacks of parsing events. However, it is unclear if our method, which crucially utilizes errors caused by an exceeding depth of the input document, can effectively apply to the case.

continues for the outer nesting level. Note that evaluating AG $parse^{(d)}$ over a document of an exceeding depth causes the stack to overflow. The overflowed data are lost and, when the stack is popped, an attribute referring to a lost datum is flagged *undef*, which notifies a parsing error.

Figure 7 defines an AG *unparse*, which generates a stream of parsing events from a given document tree. In contrast to *parse*, this AG always produces a result successfully, irrespective of the depth of the input document tree. The AG is intended to perform a depth-first traversal on a document tree from the right to the left. The result of the depth-first traversal is accumulated to the inherited attribute *acc*, and the accumulated result will be propagated up to the root of the document tree as the final result.

2.4 Specifying a document transformation

The problem to be solved is now formally stated as follows: Given a specification of a document transformation in a composite form $unparse \circ T \circ parse^{(d)}$, where d is the maximum depth d of the input document and T is an AG representing a tree-based transformation, derive a one-path interactive event-based document transformation program such that the resulting event-based transformation computes the same result as the composite specification does, for every input document of a depth d or less. Note that the output document may have an arbitrary depth, in contrast to the input document.

3 The Algorithm

This section presents an algorithm for deriving a one-pass event-based document transformation program from a composite specification of tree transformation $unparse \circ T \circ parse^{(d)}$. Throughout this section, we consider the case where the AG T is an identity, *i. e.*, T is an AG which transforms a document tree to itself, as a running example. The running example may seem too simple. However, even this simple example demonstrates well what are the difficulties and how they are to be solved. For the sake of further simplification, we assume $d = 2$ and document tags have only two different names `A` and `B`. (Thus there are only two data constructors for start tags, `Begin_A` and `Begin_B`.)

Our algorithm is comprised of three phases. First, the AG T specifying a document tree transformation is composed with the AGs for parsing and unparsing, resulting a single deforested AG. Second, the resulting AG is further transformed into a finite state transition machine representing the document transformation. Finally, we obtain a one-pass interactive event-driven program from the finite state transition machine. We will explain each phase in turn in the rest of this section.

Let us define some notations. We write $\overline{x}$ for abbreviating a finite sequence $x_1, \ldots, x_n$ of syntactic objects. We write $[x := y]$ to express a substitution, which replaces all occurrences of x by y. A function f is called a finite map, if its domain, denoted by $dom(f)$, is a finite set. We conventionally write $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ to denote a finite map f such that $dom(f) = \{x_1, \ldots, x_n\}$ and $f(x_i) = v_i$ for every i. For any finite maps f and g such that

$dom(f) \cap dom(g) \neq \emptyset$, $f \uplus g$ denotes a finite map h such that $dom(h) = dom(f) \cup dom(g)$, $h(x) = f(x)$ if $x \in dom(f)$, and $h(x) = g(x)$ if $x \in dom(g)$.

3.1 Descriptional composition

In the first phase of the derivation, we obtain a single deforested AG, which does not produce any intermediate data, from a specification of a document transformation. This section only gives a short summary of the descriptional composition framework by means of the running example. For a more detailed, formal definition of the framework, readers are deferred to [10, 11, 8, 9].

To obtain a single AG from an AG specification of transformation $unparse \circ T \circ parse^{(d)}$, we need to apply the descriptional composition twice. As for the running example, we first compose T with $parse^{(d)}$, and then the resulting AG with $unparse$. Since T is an identity, apparently the first composition results in $parse^{(d)}$.[2] Therefore, we will present the non-trivial second composition of $unparse$ and $parse^{(d)}$.

Let F and G be two AGs, where F's result ranges over the set of terms of G's underlying grammar. (In the running example, F is $parse^{(d)}$ and G is $unparse$.) The descriptional composition derives a single deforested AG for a composition $G \circ F$ in three steps, as explained below.

Projection. The first step is *projection*, which derives an intermediate representation of the composed AG, where the intermediate data constructions between the two AGs still remain. In the intermediate representation, we temporarily write $e.a$ to denote an attribute occurrence on an arbitrary expression e.

The intermediate representation is obtained by projecting every attributes of G over each semantic rule $N.a = e$ in F so that a new semantic rule of the form $(N.a).b = e.b$ ($e.b = (N.a).b$, *resp.*) is created for each synthesized (inherited, *resp.*) attribute b of G. For example, projecting attributes of $unparse$ over the semantic rules for the production $E \to \texttt{Begin\_A}\ E'$ of $parse$, we obtain the following new semantic rules:

$$\Sigma_G = \begin{array}{l} (E.parse).unparse = \underline{(\texttt{Node\_A}\ E'.parse\ \ E'.stack_1).unparse} \\ \underline{(\texttt{Node\_A}\ E'.parse\ \ E'.stack_1).acc} = (E.parse).acc \end{array}$$

The projection algorithm treats semantic rules of the start production in a different way from the others. The semantic rules for the start production of the composed AG is those of G, whose every reference to an attribute of G (*i. e.*, an attribute occurrence of the form $T.a$) is replaced by the corresponding attribution on the result of F (*i. e.*, by $e_f.a$, where $S.result = e_f$ is in a semantic rule of the start production of F).[3] For example, the following new semantic rules are derived for the start production $S \to E$.

$$\begin{array}{l} S.result = (E.parse).unparse \\ (E.parse).acc = \texttt{Nil} \end{array}$$

[2]We do not show the process of this composition, which is simpler than the one presented below.
[3]This is a special case of profile symbolic evaluation described in [9].

Symbolic evaluation. The second step is *symbolic evaluation* [8, 9], which eliminates expressions of the form $(\texttt{C}\ \overline{e}).a$ $(a \in Att_G)$ representing attributions on intermediate data.

Consider the above projected semantic rules for the production $E \to \texttt{Begin\_A}\ E'$. They contain attributions to the same intermediate data construction (the underlined parts). Let $T \to \texttt{Node\_A}\ T_1\ T_2$ be the production rule for the construction, where $T = (\texttt{Node\_A}\ E'.parse\ E'.stack_1)$, $T_1 = E'.parse$, and $T_2 = E'.stack_1$. The corresponding semantic rules for this intermediate data construction are given by the following equations, according to the definition of *unparse*.

$$\Sigma_F = \begin{array}{l} \underline{(\texttt{Node\_A}\ E'.parse\ \ E'.stack_1).unparse} = \texttt{Begin\_A}\ ((E'.parse).unparse) \\ (E'.parse).acc = \texttt{End}\ ((E'.stack_1).unparse) \\ (E'.stack_1).acc = \underline{(\texttt{Node\_A}\ E'.parse\ \ E'.stack_1).acc} \end{array}$$

Using these equations, we can eliminate the intermediate datum and obtain a new set of semantic rules for the production $E \to \texttt{Begin\_A}\ E'$.

$$\begin{array}{l} (E.parse).unparse = \texttt{Begin\_A}\ ((E'.parse).unparse) \\ (E'.parse).acc = \texttt{End}\ ((E'.stack_1).unparse) \\ (E'.stack_1).acc = (E.parse).acc \end{array}$$

These semantic rules are obtained as follows: Let us write Σ^{syn} (Σ^{inh}, *resp.*) to denote a subset of Σ such that it contains only semantic rules of the form $e.a = e'$ such that a is a synthesized (inherited, *resp.*) attribute. Let Σ_G be the set of semantic rules from which we would like to eliminate intermediate data construction expressed by $(\texttt{C}\ v_1\ \ldots\ v_n)$, Σ_F be the set of semantic rules corresponding to the intermediate data construction, and $T \to \texttt{C}\ T_1\ \cdots\ T_n$ be the production rule relevant to the intermediate data construction. If $\Sigma_G^{syn} = \{occ = e.s\} \cup \Sigma'$, $e.s = e' \in \Sigma_F^{syn}$, and $\Sigma_G^{inh} = \{(\texttt{C}\ v_1\ \ldots\ v_n).h_i = e_i \mid i = 1, \ldots, m\}$, then the new set of semantic rules is the union of $\sigma(\{occ = e'\} \cup \Sigma')$ and $\sigma(\Sigma_F^{inh})$, where σ is a substitution for eliminating the intermediate data construction such that $\sigma = [T_1 := v_1] \cdots [T_n := v_n][T.h_1 := e_1] \cdots [T.h_n := e_n]$. In the above example the underlined parts are subject to this substitution.

We can eliminate all the intermediate data by repeating the above rewriting process on the semantic rules. Note that, since we allow undefined attributes, Σ_G^{inh} may not have corresponding expressions to some inherited attributes (*i. e.*, there may be no semantic rule $(\texttt{C}\ v_1\ \ldots\ v_n).h_i = e_i$ for some i). In such a case, the substitution should be understood to replace every reference to an undefined attribute occurrence (say, $T.h_i$) by an undefined value, *i. e.*, every semantic rule $e.a = e'$, where e' contains a reference to such an occurrence, is removed from the set of semantic rules (in other words, $e.a$ is set to undefined).

Renaming. The final step of descriptional composition is *renaming.* This step rewrites any successive attributions of the form $(x.a).b$ to a single attribution $x.a\_b$, by composing the successive two attribute names into a single one.

The result of descriptional composition applied to *unparse* $\circ$ *parse* (after an appropriate α-conversion on the attribute names) is shown in Figure 8. The quasi-SSUR condition is preserved through the composition process (*i. e.*, the resulting AG is quasi-SSUR).

$$
\begin{array}{l}
\texttt{let}\ \mathit{ident}\ = \\
\quad S \to E: \\
\qquad S.\mathit{result} = E.\mathit{syn}_2 \\
\qquad E.\mathit{inh}_1 = \mathit{undef} \\
\qquad E.\mathit{inh}_2 = \mathit{undef} \\
\qquad E.\mathit{inh}_3 = \texttt{Nil} \\
\\
\quad E \to \texttt{Begin\_A}\ E': \\
\qquad E.\mathit{syn}_1 = \mathit{undef} \\
\qquad E.\mathit{syn}_2 = \texttt{Begin\_A}\ E'.\mathit{syn}_2 \\
\qquad E.\mathit{syn}_3 = E'.\mathit{syn}_1 \\
\qquad E'.\mathit{inh}_1 = E.\mathit{inh}_2 \\
\qquad E'.\mathit{inh}_2 = E.\mathit{inh}_3 \\
\qquad E'.\mathit{inh}_3 = \texttt{End}\ E'.\mathit{syn}_3
\end{array}
$$

$$
\begin{array}{l}
E \to \texttt{Begin\_B}\ E': \\
\quad E.\mathit{syn}_1 = \mathit{undef} \\
\quad E.\mathit{syn}_2 = \texttt{Begin\_B}\ E'.\mathit{syn}_2 \\
\quad E.\mathit{syn}_3 = E'.\mathit{syn}_1 \\
\quad E'.\mathit{inh}_1 = E.\mathit{inh}_2 \\
\quad E'.\mathit{inh}_2 = E.\mathit{inh}_3 \\
\quad E'.\mathit{inh}_3 = \texttt{End}\ E'.\mathit{syn}_3
\end{array}
$$

$$
\begin{array}{l}
E \to \texttt{End}\ E': \\
\quad E.\mathit{syn}_1 = E'.\mathit{syn}_3 \\
\quad E.\mathit{syn}_2 = E.\mathit{inh}_3 \\
\quad E.\mathit{syn}_3 = E'.\mathit{syn}_2 \\
\quad E'.\mathit{inh}_1 = \mathit{undef} \\
\quad E'.\mathit{inh}_2 = E.\mathit{inh}_1 \\
\quad E'.\mathit{inh}_3 = E.\mathit{inh}_2 \\
\\
E \to \texttt{Nil}: \\
\quad E.\mathit{syn}_1 = \mathit{undef} \\
\quad E.\mathit{syn}_2 = E.\mathit{inh}_3 \\
\quad E.\mathit{syn}_3 = \mathit{undef}
\end{array}
$$

Figure 8: Result of descriptional composition

3.2 Deriving an event-based transformation program

The second phase of our algorithm generates a one-pass event-driven program from the result of a descriptional composition.

Let GF be the AG obtained by the descriptional composition. We can observe that the underlying grammar GF is that of event streams (produced from the start symbol $S_{EventStream}$ in Figure 3) and that every attribute of GF ranges over the set of event streams. In the following, we write $\Sigma_{\texttt{C}}$ to denote the set of semantic rules of the production rule corresponding to each constructor `C` of event streams. Every semantic rule is defined by either a reference to an attribute occurrence or a sequence of event stream constructions. We assume $\{syn_1, \ldots, syn_s\}$ be the set of synthesized attributes and $\{inh_1, \ldots, inh_h\}$ be the set of inherited attributes of GF.

3.2.1 A graph-based dependency analysis

To clarify what is the difficulty in generating an event-based program from an AG specification and to illustrate the key idea in our algorithm, we represent the set of semantic rules for each production rule by a graph as in Figure 9. In the graphical representation, we are concerned only with dependency between attribute occurrences, ignoring what semantic rules compute just for the moment.

Each graphical representation consists of two (or one) horizontal bars and some dependency edges. For every production rule of the form $E \to \texttt{C}\ E'$, where E and E' are non-terminal symbols and `C` is a data constructor, the attribute occurrences on E is placed on the upper bar and those on E' on the lower bar. (No lower bar is present for the production rule $E \to \texttt{Nil}$.) We place synthesized attributes to the left and inherited attributes to the right of each bar.

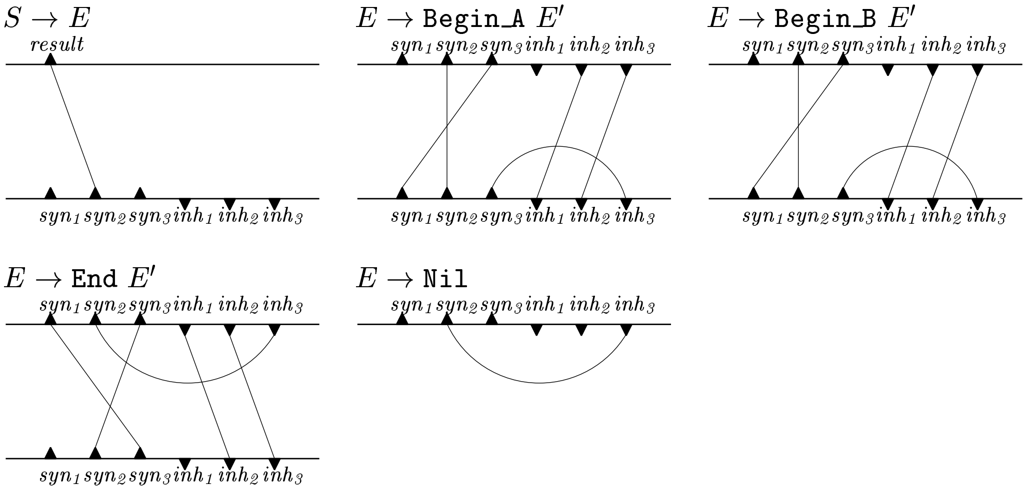

Figure 9: Attribute dependency graphs

Each dependency edge connects two attributes, which are designated by a triangle mark on the bars. The direction of data flow is indicated by either an upper or a lower triangle mark.

Note that, since each expression defining an attribute value is either a reference to an attribute occurrence or a data construction, each attribute occurrence is dependent on at most a single other occurrence. (An attribute occurrence *occ* has no incoming edges only if the semantic rule for the occurrence is either undefined or defined by an expression of the form $occ = \mathtt{C}_1(\ldots(\mathtt{C}_k\ \mathtt{Nil})\ldots)(k \geq 0)$.) Hence no attribute has more than two incoming edges. Furthermore, due to the quasi-SSUR condition, no attribute has more than two departing edges either.

We call an attribute dependency, which departs from a synthesized attribute and comes to an inherited attribute on the same bar in a graph representation, a *forward dependency*. For example, the dependency graph for the production rule `Begin_A` contains a forward dependency edge from syn_3 to inh_3 on the lower bar. Forward dependencies are difficult to be handled in a one-pass evaluator. In the present example, if `Begin_A` is followed by an event `C`, the attribute evaluation for `C` requires the value of inh_3, although it is defined by syn_3, which is the result of attribute evaluation on `C`. In this case, one might employ an evaluation scheme which resorts to the general lazy evaluation mechanism [12]. In a lazy evaluator, each evaluation of an attribute is delayed until all attribute values necessary for the evaluation are obtained. However, this causes a chain of suspended computation, which amounts to lazily constructing a list of the input stream on the memory, hence the effect of deforestation is only moderate. The evaluation scheme derived by the algorithm proposed in this paper, in contrast, dismisses the need for intermediate data constructions completely, as for the present example.

The crucial step in our algorithm is to eliminate such forward dependencies by a lookahead into the event stream. Since the structure of an event stream is linear and the input document

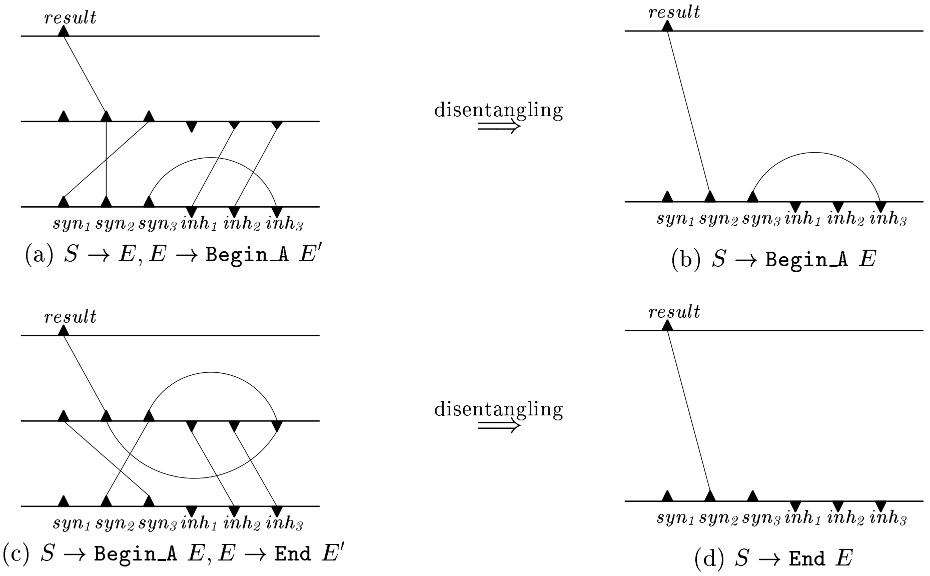

Figure 10: Pasting and disentangling dependency graphs

is limited to have only a finite depth, an attribute evaluator would be possible to determine the value of any attribute occurrence by looking into a few more parsing events. To see it, suppose the begging of the input stream is a parsing event `Begin_A`. The corresponding grammar productions are $S \to E$ followed by $E \to$ `Begin_A` E'. This successive productions promote two steps of attribute evaluation, whose attribute dependency is visualized in Figure 10(a). The graph is obtained by pasting the dependency graph of the latter production under that of the former one. This pasted graphs are unified into a single one, by omitting the middle bar (corresponding to the intermediate production) and then taking the transitive closure of dependency edges (Figure 10(b)). We call this process for taking the transitive closure a *disentangling* process. A disentangling process merges two successive production rules into a single new rule.

The above result of disentangling introduces new dependency edges, including a forward dependency. However, the forward dependency can be eliminated by a further lookahead into the event stream. Suppose the next parsing event be `End`. We paste the graph of the production rule $E \to$ `End` E' under the new graph (Figure 10(c)) and then disentangle it (Figure 10(d)). The forward dependency is canceled. Notice that the resulting graph is isomorphic to the dependency graph for the start production, hence we are at a state equivalent to the initial state.

In the following, we represent a dependency graph by a finite map $\mathcal{D}$, where $\mathcal{D}(occ) = occ'$ if the value of the semantic rule defining occ is dependent to occ', and $\mathcal{D}(occ) = none$ otherwise, where $none$ is a special symbol indicating no dependency.

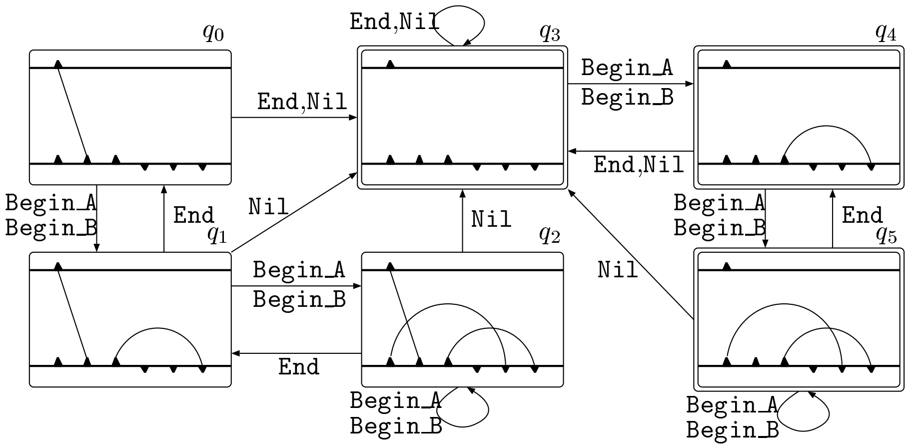

Figure 11: A state transition diagram

3.2.2 Generating a finite state transition machine

Starting from the dependency graph of the start production, our algorithm enumerates all possible dependency graphs. The algorithm exhaustively examines every possible next parsing event (namely, $\texttt{Begin\_tag}_1$, …, $\texttt{Begin\_tag}_n$, **End**, and **Nil**) for each different dependency graph. Since there are only finite attributes, there are only finite number of possible dependency graphs and therefore this process must terminate. As for the running example, we obtain 6 patterns of dependency graphs as in Figure 11. Regarding each graph as a state of parsing, we can view the diagram of Figure 11 as a finite state transition machine. Starting from the initial state q_0, the machine repeatedly changes its state according to the parsing event to be read, following an arrow labeled with the event name. The machine terminates when it reaches to a final state, where the value of attribute *result* is no more dependent on other attribute occurrences. In the figure, the final states are drawn by a double line.

Though the state transition rule can be statically determined by an an analysis on attribute dependencies, it remains unsolved how to compute attribute values during transitions. The solution in this paper is to let each attribute carry a partially evaluated value. A partially evaluated value is represented by a unary function, whose argument is the not-yet evaluated attribute value. A fully evaluated value is computed, when the not-yet evaluated attribute value ever becomes definite, by applying the function to the definite value.

Since semantic rules are restricted to event stream constructions, such a functional representation of partially evaluated values can be expressed by a composition of a finite number of function constants $\texttt{Begin\_tag}_1$, ..., $\texttt{Begin\_tag}_n$, End, Nil, Id, and Undef, where the last three constants should be understood as functions $\texttt{Nil} = \lambda x.\texttt{Nil}$, $\texttt{Id} = \lambda x.x$, and $\texttt{Undef} = \lambda x.\mathit{undef}$ (a function which is not defined for any input), *resp.* We can regard the function constants $\texttt{Begin\_tag}_1, \ldots$ as meta-symbols and $\circ$ as a composition operator over them, where the operator is associative and a few additional equalities $\texttt{Id} \circ f = f \circ \texttt{Id} = f$, $\texttt{Nil} \circ f = \texttt{Nil}$, and

$\mathtt{Undef} \circ f = f \circ \mathtt{Undef} = \mathtt{Undef}$ hold. We call such a representation a *symbolic representation* of partially evaluated attributes. We can assume that every semantic rule is given in the form either $occ = (\mathtt{C}_1 \circ \cdots \circ \mathtt{C}_k \circ \mathtt{Nil})(none)$, or $occ = (\mathtt{C}_1 \circ \cdots \circ \mathtt{C}_k)(occ')$, where $\mathtt{C}_1, \ldots, \mathtt{C}_k (k \geq 0)$ are neither `Nil` nor `Undef`.

The state transition machine changes the syntactic representation of partially evaluated values for every state transition. Let $X_{result}, X_{inh_1}, \ldots, X_{inh_h}$ be meta-variables denoting the syntactic representation at the current state. (Except *result*, synthesized attributes are not needed, since the state transition machine is initiated from the start production, whose only synthesized attribute is *result*.) Suppose the state transition machine for the running example is at the initial state of parsing, where the attribute dependency $\mathcal{D}$ of the start production $S \to E$ gives a dependency such that $\mathcal{D}(S.result) = E.syn_2$ and $\mathcal{D}(E.inh_j) = none$ for every j. If the next parsing event to be read is `Begin_A`, since the semantic rule defining the attribute syn_2 is expressed as $E.syn_2 = (\mathtt{Begin\_A})(E'.syn_2)$, the partially evaluated value of syn_2 after the transition is represented by $X_{result} \circ \mathtt{Begin\_A}$, where X_{result} denotes the initial symbolic representation for the attribute *result*.

The definition of the finite state transition machine is given below.

Definition 3.1 (Finite State Transition Machine) *Let FRep denote the set of finite maps of the form* $\{result \mapsto f_0, inh_1 \mapsto f_1, \ldots, inh_h \mapsto f_h\}$*, which assigns a symbolic representation* f_i *to each attribute.*

A finite state transition machine *is a septuple* $\mathcal{M} = (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$*, where* A *is the set of input parsing events,* Q *is the set of states (the set of patterns of dependency graphs),* $q_0 (\in Q)$ *is the initial state,* $T (\subseteq Q)$ *is the set of final states,* δ *is a finite map* $Q \times A \to Q$ *representing the set of state transition rules,* Γ *is a finite map* $Q \to (A \to FRep)$ *representing the set of rules for computing the symbolic representation of partially evaluated attribute values for each state transition, and* $\gamma_0 (\in FRep)$ *is a finite map giving the initial symbolic representation.*

The state transition machine, if the current state is q and the next parsing event to be read is `C`, makes a transition to the state $\delta(q, \mathtt{C})$. The new symbolic representation is given by a finite map $\Gamma(q)(\mathtt{C})$. As for the running example, when the machine reads a parsing event `Begin_A` at the initial state, the symbolic representation is given by $\Gamma(q_0)(\mathtt{Begin\_A}) = \{result \mapsto X_{result} \circ \mathtt{Begin\_A}, inh_1 \mapsto X_{inh_2}, inh_2 \mapsto X_{inh_3}, inh_3 \mapsto \mathtt{End}\}$, where the meta-variables $X_{result}, X_{inh_1}, \ldots, X_{inh_h}$ designate the initial symbolic representation of attribute values.

Now we give an algorithm for generating such a finite state transition machine $\mathcal{M}$.

First, we define a procedure $DISENT^{\mathcal{D},\mathtt{C}}$. The superscript $\mathcal{D}$ is the attribute dependency graph under the current parsing context and `C` is a parsing event. Let $E \to \mathtt{C}\ E'$ (or $E \to \mathtt{Nil}$) be the corresponding production rule and $\Sigma_{\mathtt{C}}$ be the set of semantic rules.

Algorithm 3.1 (Disentangling)

procedure $DISENT^{\mathcal{D},\mathtt{C}}(occ, f)$
  input *occ*: an attribute occurrence,
    f: a symbolic representation of a partially evaluated attribute value

case *occ* **of**

$S.result \Rightarrow$ **if** $\mathcal{D}(S.result) \neq none$ **then** $DISENT^{\mathcal{D},\mathtt{C}}(\mathcal{D}(S.result), X_{result})$
else return $(none, X_{result})$

$E.inh_j \Rightarrow$ **if** $\mathcal{D}(E.inh_j) \neq none$ **then** $DISENT^{\mathcal{D},\mathtt{C}}(\mathcal{D}(E.inh_j), f \circ X_{inh_j})$
else return $(none, f \circ X_{inh_j})$

$E.syn_k \Rightarrow$ **if** $E.syn_k = f'(occ') \in \Sigma_{\mathtt{C}}$ **then**
if $occ' \neq none$ **then** $DISENT^{\mathcal{D},\mathtt{C}}(occ', f \circ f')$ **else return** $(none, f \circ f')$
else return $(none, f)$

$E'.inh_j \Rightarrow$ **if** $E'.inh_j = f'(occ') \in \Sigma_{\mathtt{C}}$ **then**
if $occ' \neq none$ **then** $DISENT^{\mathcal{D},\mathtt{C}}(occ', f')$ **else return** $(none, f')$
else return $(none, \mathtt{Id})$

$E'.syn_k \Rightarrow$ **return** $(E'.syn_k, f)$

`end`

A procedure call $DISENT^{\mathcal{D},\mathtt{C}}(occ, \mathtt{Id})$ traverses the attribute dependencies and returns a pair of an occurrence occ' and a symbolic representation f. The result indicates the definition of occ depends on the value of occ' and the symbolic representation of the partially evaluated attribute value is f, if the parsing event $\mathtt{C}$ is read from the input stream at the current parsing state $\mathcal{D}$. Note that the result of a procedure call $DISENT^{\mathcal{D},\mathtt{C}}(S.result, \mathtt{Id})$ is always expressed in the form $(occ, X_{result} \circ \mathtt{C}_1 \circ \cdots \circ \mathtt{C}_k)$ for some attribute occurrence occ' and function constants $\mathtt{C}_1, \ldots, \mathtt{C}_k (k \geq 0)$.

Now we are ready to present the algorithm which translates the AG GF, which is derived in the previous descriptional composition phase, to a finite state transition machine.

Algorithm 3.2 (Construction of a finite state transition machine) *Let $\mathcal{D}_0$ and $\mathcal{F}_0$ be the attribute dependency graph and the symbolic representation for the start production $S \rightarrow E$ respectively,* i. e., $\mathcal{D}_0 = \biguplus_{occ \in \{S.result, E'.inh_1, \ldots, E'.inh_h\}} \{occ \mapsto occ' \mid occ = f(occ') \in \Sigma_S$ *or* $occ' = none$ *otherwise*$\}$ *and* $\mathcal{F}_0 = \{result \mapsto f \mid S.result = f(occ') \in \Sigma_S$ *or* $f = \mathtt{Undef}$ *when* $S.result$ *is undefined*$\} \uplus \biguplus_{h \in \{inh_1, \ldots, inh_h\}} \{h \mapsto f \mid E.h = f(occ') \in \Sigma_S$ *or* $f = \mathtt{Undef}$ *when* $E.h$ *is undefined*$\}$.

The algorithm is defined by the following procedure MakeFSTM.

procedure *MakeFSTM*(GF)
input GF: the AG derived by descriptional composition
output $\mathcal{M} = (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$: a finite state transition machine
$A := \{\mathtt{Begin\_tag}_1, \ldots, \mathtt{Begin\_tag}_n, \mathtt{End}, \mathtt{Nil}\}$; $Q := \{\mathcal{D}_0\}$; $q_0 := \mathcal{D}_0$; $T := \emptyset$
$\delta := \{\}$; $\Gamma := \{\}$; $\gamma_0 := \mathcal{F}_0$
while there exists $\mathcal{D} \in Q$ such that $\Gamma(\mathcal{D})$ is not defined **do**
$\Gamma_{\mathcal{D}} := \{\}$
for each $\mathtt{C} \in A$ **do**
$\mathcal{D}^{\mathtt{C}} := \{\}$; $\mathcal{F}^{\mathtt{C}}_{\mathcal{D}} := \{\}$
$(occ, f) := DISENT^{\mathcal{D},\mathtt{C}}(S.result, \mathtt{Id})$;
$\mathcal{D}^{\mathtt{C}} := \mathcal{D}^{\mathtt{C}} \uplus \{S.result \mapsto occ\}$; $\mathcal{F}^{\mathtt{C}}_{\mathcal{D}} := \mathcal{F}^{\mathtt{C}}_{\mathcal{D}} \uplus \{result \mapsto f\}$;

if $\mathtt{C} \neq \mathtt{Nil}$ **then**
  for each inh_j **do**
    $(occ, f) := DISENT^{\mathcal{D},\mathtt{C}}(E'.inh_j, \mathtt{Id})$;
    $\mathcal{D}^{\mathtt{C}} := \mathcal{D}^{\mathtt{C}} \uplus \{E'.inh_j \mapsto occ\}$; $\mathcal{F}^{\mathtt{C}}_{\mathcal{D}} := \mathcal{F}^{\mathtt{C}}_{\mathcal{D}} \uplus \{inh_j \mapsto f\}$
  end
  $\delta := \delta \uplus \{(\mathcal{D}, \mathtt{C}) \mapsto \mathcal{D}^{\mathtt{C}}\}$; $\Gamma_{\mathcal{D}} := \Gamma_{\mathcal{D}} \uplus \{\mathtt{C} \mapsto \mathcal{F}^{\mathtt{C}}_{\mathcal{D}}\}$
end
$Q := Q \cup \{\mathcal{D}^{\mathtt{C}}\}$
if $\mathcal{D}^{\mathtt{C}}(S.result) = none$ **then** $T := T \cup \{\mathcal{D}^{\mathtt{C}}\}$
$\Gamma := \Gamma \uplus \{\mathcal{D} \mapsto \Gamma_{\mathcal{D}}\}$
end
return $\mathcal{M} := (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$

The procedure *MakeFSTM* enumerates all possible attribute dependency graphs, starting from the initial dependency graph $\mathcal{D}_0$. For every dependency graph $\mathcal{D}$, it examines every parsing event C in turn as a possible parsing event to be read, and computes the new dependency graph $\mathcal{D}^{\mathtt{C}}$ and the new assignment $\mathcal{F}^{\mathtt{C}}_{\mathcal{D}}$ of symbolic representations. If $\mathtt{C} = \mathtt{Nil}$, since no parsing event follows it, we need not compute the assignments for inherited attributes $inh_1, \ldots, inh_h$. The dependency graph $\mathcal{D}^{\mathtt{C}}$ is added to the set T of final states, if it has no dependency for the occurrence $S.result$. This process is iterated until there is no new dependency graph is generated.

A graph based analysis of attribute dependency similar to the present scheme can be found in Kastens' visit sequence construction [13], in which attribute orderings are derived by an exhaustive enumeration of all possible attribute dependencies. The crucial difference between his and ours is that his scheme preserves the original production rules while ours pastes together production rules to create a new one. The attribute evaluators derived from these two schemes are very different as well. Kastens' visit sequence evaluator evaluates only a subset of attributes per a visit to a syntax node and hence it is a multi-pass evaluator. On the contrary, ours (given in Section 3.3) is a one-pass evaluator, which evaluates all attributes per a visit to a node, but only partially.

3.3 Translation into an event-based program

The final phase of the derivation is to translate the finite state transition machine into an event-based program.

An executable functional program is easily obtained from a given state transition machine $\mathcal{M} = (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$ by a set of mutual recursive functions $\{F_{\mathcal{D}} \mid \mathcal{D} \in Q \setminus T\}$. Each function $F_{\mathcal{D}}(s, X_{result}, X_{inh_1}, \cdots, X_{inh_h})$ corresponds to the state $\mathcal{D} \in Q$ and it takes an input stream s and partially evaluated attribute values $X_{result}, X_{inh_1}, \cdots, X_{inh_h}$ as arguments, where the partially evaluated attributes are expressed by a function closure. That is, each function constant symbol ($\mathtt{Node\_tag}_1$, etc.) is encoded by the corresponding function and the composition operator is encoded just by a function composition. Each function $F_{\mathcal{D}}$ fetches the next parsing event C and makes a transition to the next state $\delta(\mathcal{D}, \mathtt{C})$, if it is not a final state, by a mutual recursive call $F_{\delta(\mathcal{D})}(s', \Gamma(\mathcal{D})(\mathtt{C})(result), \Gamma(\mathcal{D})(\mathtt{C})(inh_1), \ldots, \Gamma(\mathcal{D})(\mathtt{C})(inh_1))$, where s' is the rest of the input stream and each $\Gamma(\mathcal{D})(\mathtt{C})(att)$ is a new representation of partially evaluated value of the attribute *att*. If the next state is a final state, the program terminates and the result is $\Gamma(\mathcal{D})(\mathtt{C})(result)$.

The resulting functional program obtained by this translation, however, is not interactive. The result is carried around in a function closure and hence the result is obtained only at the very end of parsing.

In order to obtain an interactive program, we express a partially evaluated attribute value by a concatenation list, which is either an empty list `[]`, a singleton list `[C]` (a list with only one element C), or a concatenation $L_1@L_2$ of two lists L_1 and L_2. A translation of a symbolic representation f into a concatenation list, denoted by $\lceil f \rceil$, is defined by the following equations.

$$\lceil \mathtt{Id} \rceil = \mathtt{[]},\ \lceil X_{att} \rceil = X_{att},\ \lceil \mathtt{C} \rceil = \mathtt{[C]},\ \lceil f_1 \circ f_2 \rceil = \lceil f_1 \rceil @ \lceil f_2 \rceil,$$

where C is either $\mathtt{Begin\_tag}_1, \ldots, \mathtt{Begin\_tag}_n, \mathtt{End}, \mathtt{Nil}$, or Undef. Due to the equalities that hold for the functional composition operator (Section 3.2.2), we can identify concatenation lists up to the associativity of the concatenation operator and the equalities $\mathtt{[Id]}@L = L@\mathtt{[Id]} = L$, $\mathtt{[Nil]}@L = \mathtt{[Nil]}$, and $\mathtt{[Undef]}@L = L@\mathtt{[Undef]} = \mathtt{[Undef]}$.

An event-based interactive program, which uses the list representation of attribute values, is derived by the following translation algorithm. We assume that **flush** is a primitive command which takes a concatenation list of parsing events and write out the events to the output stream. If the argument is `[Undef]`, the command aborts the execution with a notification of an error to the other end of the output stream. We also assume a primitive command **exit** terminates the execution successfully.

Algorithm 3.3 (Translation to an interactive event-based program)
Let $\mathcal{M} = (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$ be a finite state transition machine. The machine $\mathcal{M}$ is translated into a program which consists of the set of procedures $\{F_{\mathcal{D}} | \mathcal{D} \in Q \setminus T\}$ and an initial procedure $MAIN$.

For each state $\mathcal{D} \in Q \setminus T$, we derive a procedure:

procedure $F_{\mathcal{D}}(s, X_{inh_1}, \cdots, X_{inh_h})$
  case s **of**
    $\mathtt{Begin\_tag}_1 :: s' \Rightarrow P_{\mathcal{D},\mathtt{Begin\_tag}_1}$
    $\mathtt{Begin\_tag}_n :: s' \Rightarrow P_{\mathcal{D},\mathtt{Begin\_tag}_n}$
    $\mathtt{End} :: s' \Rightarrow P_{\mathcal{D},\mathtt{End}}$
    $\mathtt{Nil} \Rightarrow P_{\mathcal{D},\mathtt{Nil}}$
  end

where s is the input stream, $\mathtt{C} :: s' \Rightarrow P_{\mathcal{D},\mathtt{C}}$ (or $\mathtt{C} \Rightarrow P_{\mathcal{D},\mathtt{C}}$ if $\mathtt{C} = \mathtt{Nil}$) is a pattern matching construct, which executes the program block $P_{\mathcal{D},\mathtt{C}}$, when the next parsing event mathes C with

s' as the rest of the input stream. Each program block $P_{\mathcal{D},\mathtt{C}}$ is defined by:

$$P_{\mathcal{D},\mathtt{C}} \quad = \quad \begin{cases} \textbf{flush } L;\ \textbf{call } F_{\delta(\mathcal{D},\mathtt{C})}(s', \lceil\Gamma(\mathcal{D})(\mathtt{C})(inh_1)\rceil, \ldots, \lceil\Gamma(\mathcal{D})(\mathtt{C})(inh_h)\rceil) & \text{if } \delta(\mathcal{D},\mathtt{C}) \notin T \\ \textbf{flush } L;\ \textbf{exit} & \text{otherwise} \end{cases}$$

where L is a concatenation list such that $X_{result}@L = \lceil\Gamma(\mathcal{D})(\mathtt{C})(result)\rceil$.

Finally, the initial procedure $MAIN$ is defined as follows.

procedure $MAIN(s)$
  flush L; **call** $F_{q_0}(s, \lceil\gamma_0(inh_1)\rceil, \ldots, \lceil\gamma_0(inh_h)\rceil)$

where L is a concatenation list such that $X_{result}@L = \lceil\gamma_0(result)\rceil$.

For example, the finite state transition machine for the identity transformation (Figure 11) is translated into the following program.

```
procedure MAIN(s)
    flush [];   call F_0(s, [Undef], [Undef], [Nil])
procedure F_0(s, X_inh1, X_inh2, X_inh3)
    case s of
        Begin_A :: s' => flush [Begin_A];  call F_1(s', X_inh2, X_inh3, [End])
        Begin_B :: s' => flush [Begin_B];  call F_1(s', X_inh2, X_inh3, [End])
        End :: s'     => flush X_inh3;     exit
        Nil           => flush X_inh3;     exit
    end
procedure F_1(s, X_inh1, X_inh2, X_inh3)
    case s of
        Begin_A :: s' => flush [Begin_A];  call F_2(s', X_inh2, X_inh3, [End])
        Begin_B :: s' => flush [Begin_B];  call F_2(s', X_inh2, X_inh3, [End])
        End :: s'     => flush X_inh3;     call F_0(s', [Undef], X_inh1, X_inh2)
        Nil           => flush [Undef];    exit
    end
procedure F_2(s, X_inh1, X_inh2, X_inh3)
    case s of
        Begin_A :: s' => flush [Begin_A];  call F_2(s', [Undef], X_inh3, [End])
        Begin_B :: s' => flush [Begin_B];  call F_2(s', [Undef], X_inh3, [End])
        End :: s'     => flush X_inh3;     call F_1(s', [Undef], X_inh1, X_inh2)
        Nil           => flush [Undef];    exit
    end
```

The resulting program is a completely interacive, one-pass event-based document transformer. As it reads a parsing event, it instantly outputs a parsing event either `Begin_A`, `Begin_B`, or `End` (carried by the argument X_{inh_3}) accordingly.

4 Examples of Derivation

This section applies our derivation algorithm to a few AGs representing typical document transformations. The simple document transformations presented here can be combined together to express a more complicated transformation, which can be unified into a single deforested one by a repeated application of the descriptional composition.

In this section, we assume that document tags have only two different names `A` and `B`, for the sake of simplicity. Furthermore, we assume that the maximum depth d of the input document is 2, when applying our algorithm. Due to the lack of space, we do not exhibit all the resulting programs, only giving the number of states of the resulting finite state machine instead.

Simple filters. Filtering documents, such as renaming tag names or eliminating unnecessary tags, are the most typically used document transformations. A filter for tag renaming, which renames all the tag name `A` into `B`, is specified by the following AG.

$$\begin{array}{lcll}
\texttt{let}\ \mathit{rename} & = & S \to T : & S.\mathit{result} = T.\mathit{val} \\
 & & T \to \texttt{Node_A}\ T_1\ T_2 : & T.\mathit{val} = \texttt{Node_B}\ T_1.\mathit{val}\ T_2.\mathit{val} \\
 & & T \to \texttt{Node_B}\ T_1\ T_2 : & T.\mathit{val} = \texttt{Node_B}\ T_1.\mathit{val}\ T_2.\mathit{val} \\
 & & T \to \texttt{Empty} : & T.\mathit{val} = \texttt{Empty}
\end{array}$$

Applying our algorithm to this AG results in a program very similar to the one derived from the identity transformation (Section 3.3): They have the same state transition rules, but the program for tag renaming outputs an event `Begin_B` where the identity transformation outputs `Begin_A`.

Another filtering transformation, which eliminates all the document nodes enclosed by the tag named `A`, is specified by an AG similar to the above one, where the semantic rules of the production rule $T \to \texttt{Node\_A}\ T_1\ T_2$ is changed to $T.\mathit{val} = T_2.\mathit{val}$. Our algorithm, when it is applied to this AG, derives a finite state transition machine of 11 states (4 final states out of 11).

More advanced transformations. As an example of a more complicated transformation, let us consider a horizontal reversal transformation, which reverses the order of sibling nodes at every nesting level. The AG for horizontal reverse is defined as below.

$$\begin{array}{llll}
\multicolumn{4}{l}{\texttt{let}\ \mathit{hrev}\ =} \\
\quad S \to T : & S.\mathit{result} = T.\mathit{hrev} & T.\mathit{acc} = \texttt{Empty} & \\
\quad T \to \texttt{Node_A}\ T_1\ T_2 : & T.\mathit{hrev} = T_2.\mathit{hrev} & T_1.\mathit{acc} = \texttt{Empty} & T_2.\mathit{acc} = \texttt{Node_A}\ T_1.\mathit{hrev}\ T.\mathit{acc} \\
\quad T \to \texttt{Node_B}\ T_1\ T_2 : & T.\mathit{hrev} = T_2.\mathit{hrev} & T_1.\mathit{acc} = \texttt{Empty} & T_2.\mathit{acc} = \texttt{Node_B}\ T_1.\mathit{hrev}\ T.\mathit{acc} \\
\quad T \to \texttt{Empty} : & T.\mathit{hrev} = T.\mathit{acc} & &
\end{array}$$

The resulting event-based program is displayed in Figure 12. Note that the transformation is inherently non-interactive in the sense that it is only after the final input parsing event that something is written to the output stream. In the resulting program, all the input events are buffered until it reaches the end of the input.

$$
\begin{array}{l}
\textbf{procedure } MAIN(s) \\
\quad \textbf{flush } \texttt{[]}; \quad \textbf{call } F_0(s, \texttt{[Undef]}, \texttt{[Undef]}, \texttt{[Undef]}, \texttt{[Undef]}, \texttt{[Nil]}, \texttt{[Id]}) \\
\textbf{procedure } F_0(s, X_{inh_1}, X_{inh_2}, X_{inh_3}, X_{inh_4}, X_{inh_5}, X_{inh_6}) \\
\quad \textbf{case } s \textbf{ of} \\
\qquad \texttt{Begin\_A} :: s' \Rightarrow \textbf{flush } \texttt{[]}; \quad \textbf{call } F_1(s', X_{inh_3}, X_{inh_4}, \texttt{[Begin\_A]}, X_{inh_5}, \texttt{[End]}@X_{inh_6}, \texttt{[Id]}) \\
\qquad \texttt{Begin\_B} :: s' \Rightarrow \textbf{flush } \texttt{[]}; \quad \textbf{call } F_1(s', X_{inh_3}, X_{inh_4}, \texttt{[Begin\_B]}, X_{inh_5}, \texttt{[End]}@X_{inh_6}, \texttt{[Id]}) \\
\qquad \texttt{End} :: s' \quad\;\; \Rightarrow \textbf{flush } X_{inh_6}@X_{inh_5}; \textbf{ exit} \\
\qquad \texttt{Nil} \qquad\quad\; \Rightarrow \textbf{flush } X_{inh_6}@X_{inh_5}; \textbf{ exit} \\
\quad \textbf{end} \\
\textbf{procedure } F_1(s, X_{inh_1}, X_{inh_2}, X_{inh_3}, X_{inh_4}, X_{inh_5}, X_{inh_6}) \\
\quad \textbf{case } s \textbf{ of} \\
\qquad \texttt{Begin\_A} :: s' \Rightarrow \textbf{flush } \texttt{[]}; \quad \textbf{call } F_2(s', X_{inh_3}, X_{inh_4}, \texttt{[Begin\_A]}, X_{inh_5}, \texttt{[End]}@X_{inh_6}, \texttt{[Id]}) \\
\qquad \texttt{Begin\_B} :: s' \Rightarrow \textbf{flush } \texttt{[]}; \quad \textbf{call } F_2(s', X_{inh_3}, X_{inh_4}, \texttt{[Begin\_B]}, X_{inh_5}, \texttt{[End]}@X_{inh_6}, \texttt{[Id]}) \\
\qquad \texttt{End} :: s' \quad\;\; \Rightarrow \textbf{flush } \texttt{[]}; \quad \textbf{call } F_0(s', \texttt{[Undef]}, \texttt{[Undef]}, X_{inh_1}, X_{inh_2}, X_{inh_4}, X_{inh_3}@X_{inh_6}@X_{inh_5}) \\
\qquad \texttt{Nil} \qquad\quad\; \Rightarrow \textbf{flush } \texttt{[Undef]}; \textbf{ exit} \\
\quad \textbf{end} \\
\textbf{procedure } F_2(s, X_{inh_1}, X_{inh_2}, X_{inh_3}, X_{inh_4}, X_{inh_5}, X_{inh_6}) \\
\quad \textbf{case } s \textbf{ of} \\
\qquad \texttt{Begin\_A} :: s' \Rightarrow \textbf{flush } \texttt{[Undef]}; \textbf{ exit} \\
\qquad \texttt{Begin\_B} :: s' \Rightarrow \textbf{flush } \texttt{[Undef]}; \textbf{ exit} \\
\qquad \texttt{End} :: s' \quad\;\; \Rightarrow \textbf{flush } \texttt{[]}; \quad \textbf{call } F_0(s', \texttt{[Undef]}, \texttt{[Undef]}, X_{inh_1}, X_{inh_2}, X_{inh_4}, X_{inh_3}@X_{inh_6}@X_{inh_5}) \\
\qquad \texttt{Nil} \qquad\quad\; \Rightarrow \textbf{flush } \texttt{[Undef]}; \textbf{ exit} \\
\quad \textbf{end}
\end{array}
$$

Figure 12: An event-based horizontal reversal program

The final example is a context-dependent transformation, which eliminates every `B` node whose immediate parent is an `A` node. One might want to define an AG for this transformation as follows.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{let } elimBunderA =} \\
\quad S \to T : & S.result = T.val & \\
\quad T \to \texttt{Node\_A}\ T_1\ T_2 : & T.val = \texttt{Node\_A}\ T_1.elim\ T_2.val & T.elim = \texttt{Node\_A}\ T_1.elim\ T_2.elim \\
\quad T \to \texttt{Node\_B}\ T_1\ T_2 : & T.val = \texttt{Node\_B}\ T_1.elim\ T_2.val & T.elim = T_2.elim \\
\quad T \to \texttt{Empty} : & T.val = \texttt{Empty} & T.elim = \texttt{Empty}
\end{array}
$$

where the attribute *elim* carries the result of transformation when the immediate parent is an `A` node, and the attribute *val* carries the result when the immediate parent is not an `A` node. However, our algorithm does not apply to this AG, since it is not quasi-SSUR (due to the duplicated uses of $T_1.elim$ in the second production rule).

We can circumvent the situation, by defining an AG with a set of attributes of the form $val_{b_k b_{k-1} \dots b_1}$, where the annotation $b_k b_{k-1} \dots b_1 (k \geq 0)$ is a sequence of $\{1, 0\}$ indicating if the i-th closer parent of the current document node is `A` or not. Since we have assumed that the input document has only a finite depth d, the length of the annotation is at most $d - 1$

and hence the set of attributes is finite. We can give semantic rules without duplications of attribute references. For example, the production rule $T \to \texttt{Node\_B}\ T_1\ T_2$ is given a set of semantic rules: $T.val_{\overline{b}0} = \texttt{Node\_B}\ T_1.val_{\overline{b}00}\ T_2.val_{\overline{b}0}$ and $T.val_{\overline{b}1} = T_2.val_{\overline{b}1}$ for every $\{1,0\}$-sequence $\overline{b}$ of a length less than $d-1$, and $T.val_{\varepsilon} = \texttt{Node\_B}\ T_1.val_0\ T_2.val_{\varepsilon}$, where ε designates a null sequence. Applying our algorithm to this AG, we obtain a finite state transition machine with 28 states (7 final states out of 28).

5 Conclusion and Future Work

We have given an algorithm which derives a one-pass event-based document transformation program from a tree-based specification of a document transformation. We have formalized the problem in the framework of attribute grammars (AGs) and have solved it by applying the descriptional composition followed by a derivation of an interactive attribute evaluation scheme based on a dependency graph analysis. The contribution of the present paper is the algorithm for deriving the evaluation scheme, which generates a finite state transition machine and translates it to an interactive event-based transformer. The algorithm provides a powerful means to carry out derivations in a fully automatic fashion and covers a class of typical document transformations.

Throughout the paper, we have discussed in a rather simplified setting as described in Section 2. This simplification puts a few significant restrictions, which are to be obstacles for practical uses. The document format considered here can neither place string data nor attach additional data (*e. g.*, those which are called *attributes* in XML jargon). To process these additional elements, our AG specification for tree transformation would need to be reinforced with conditional expressions. The authors suppose that the additional elements can be introduced as terminal symbols of the underlying grammar and that Boyland and Graham's work [7] would be relevant to the extension with conditional expressions. We would like to pursuit these topics in a future investigation.

References

[1] Extensible markup language (XML) 1.0 W3C recommendation. `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[2] XSL transformations (XSLT) version 1.0 W3C recommendation. `http://www.w3.org/TR/1999/REC-xslt-19991116`, 1999.

[3] Document object model (DOM). `http://www.w3.org/DOM/`, 2000.

[4] SAX 2.0: The simple API for XML. `http://www.megginson.com/SAX/`, 2000.

[5] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 2000.

[6] A. V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling*, volume I & II. Prentice-Hall, 1972.

[7] J. Boyland and S. L. Graham. Composing tree attributions. In *Proceedings of the* 21^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 375–388. ACM Press, 1994.

[8] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Symbolic composition. Technical Report 3348, INRIA, Jan. 1998. available from `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3348.ps.gz`.

[9] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: A deforestation case-study. In *Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 360–377. Springer Verlag, 1999.

[10] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19 of *SIGPLAN Notices*, pages 157–170, June 1984.

[11] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25(4):355–423, May 1988.

[12] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer Verlag, 1987.

[13] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[14] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[15] T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings* 6^{th} *ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, New York, 1993.

[16] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of the* 7^{th} *International Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[17] P. Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer Verlag, 1988.