Streamlining Functional XML Processing*

Keisuke Nakano

Department of Mathematical Informatics, University of Tokyo
Bunkyo-ku, Tokyo, 113-8656, Japan
ksk@mist.i.u-tokyo.ac.jp

Abstract

Since an XML document has tree structure, XML transformations are ordinarily defined as recursive functions over the tree. Their direct implementation often causes inefficient memory usage because the input XML tree needs to be completely stored in memory. In contrast, XML stream processing can minimize the memory usage and execution time since it begins to output the transformation result before reading the whole input. However, it is much harder to write the XML transformation program in stream processing style than in functional style because stream processing requires stateful programming. In this paper, we propose a method for automatic derivation of XML stream processor from XML tree transformation written in functional style. We use an extension of macro forest transducers as a model of functional XML processing. Since an XML parser is represented by (infinitary) top-down tree transducer, the automatic derivation of XML stream processor is based on the composition of the top-down tree transducer and the extension of macro forest transducers.

1 Introduction

Since an XML document has tree structure, it is natural to define XML transformations as recursive functions over the tree. Such a style will be called *functional XML processing*. Several XML transformation languages [8, 2, 22] have been presented in this style, in which programs are recursive functions over *forests* that are sequences of labeled trees. This is mainly because each node in an XML tree can have an arbitrary number of children.

Forests are defined by

$$f ::= \sigma\texttt{[}f\texttt{]}\,f \mid \texttt{\%[}str\texttt{]}\ f \mid \texttt{()}$$

where we write $\sigma\texttt{[}f_1\texttt{]}\,f_2$ for a sequence whose head is a σ-labeled tree with a child forest f_1 and tail is a sibling forest f_2, $\texttt{\%[}str\texttt{]}\ f$ for a text node of strings s which has a sibling forest f, and `()` for the empty sequence. For simplicity, we ignore attributes and allow the root to have sibling labeled trees. Text nodes are represented by a labeled tree with no child. For instance, an XML fragment

```
<p>  XML is <em>forest</em>. </p>
```

are represented by

```
p[ %[ XML is ] em[ %[forest] ] ]
```

Perst and Seidl [17] recently presented *macro forest transducers* (mft) which can be regarded as programs based on recursive XML Transformation. Roughly speaking, mft's can deal with recursive programs over forests[1]. A mft specifies a forest transformation by defining mutual recursive functions over forests with accumulating parameters. For example, we consider a simple XML transformation which creates a corresponding XHTML code and adds an index including all `key` elements before a postscript paragraph at the end of the input article. The transformation program is defined by a mft shown in Figure 1. The main function `Main` transforms an XML

*This work is partially supported by the *Comprehensive Development of e-Society Foundation Software* program of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

[1]This paper deals with mft's extended for text nodes.

```
Main(article[$x1]$x2) = html[ head[ Title($x1) body[ InArticle($x1, ()) ] ] ] ()

Title(title[$x1]$x2) = title[$x1]

InArticle(title[$x1]$x2, $y1) = h1[$x1] InArticle($x2, $y1)
InArticle(para[$x1]$x2, $y1) = p[Key2Em($x1)] InArticle($x2, $y1 AllKeys($x1))
InArticle(postscript[$x1]$x2, $y1) = h2[%[Index]] ul[$y1] h2[%[Postscript]] $x1

Key2Em(key[$x1]$x2) = em[$x1] Key2Em($x2)
Key2Em(%[$s] $x) = %[$s] Key2Em($x)
Key2Em(()) = ()

AllKeys(key[$x1]$x2) = li[$x1] AllKeys($x2)
AllKeys(%[$s] $x) = AllKeys($x)
AllKeys(()) = ()
```

Figure 1: Example of a mft-style XML transformation program

```
<article>
  <title>MFT</title>
  <para> XML is <key>forest</key>. </para>
  <para> <key>MFT</key> transforms forests. </para>
  <para> MFT transforms XML. </para>
  <postscript> MFT is quite expressive. </postscript>
</article>
```

into an XML

```
<html>
  <head><title>MFT</title></head>
  <body>
    <h1>MFT</h1>
    <p> XML is <em>forest</em>. </p>
    <p> <em>MFT</em> transforms forests. </p>
    <p> MFT transforms XML. </p>
    <h2>Index</h2>
    <ul> <li>forest</li> <li>MFT</li> </ul>
    <h2>Postscript</h2>
    <p> MFT is quite expressive. </p>
  </body>
</html>
```

using two auxiliary functions `Title` and `InArticle`. The function `Main` matches the argument with a pattern `article`$[x_1]x_2$ and call the function `InArticle` with the sub-forest x_1 and an extra argument `()`. The function `InArticle` collects all `em`'s in `para` elements into the accumulating parameter using a function `AllKeys`. The collection is appeared as an `Index` paragraph right before a `copyright` element. Every function matches the first argument with a pattern `para`$[x_1]x_2$ or `()` and calls functions with x_1 or x_2 as the first argument.

In a mft-style XML transformation, only the first argument of every function is matched with several patterns as a forest. The rest of arguments can be used for accumulating parameters such as `y1` in the definition of `InArticle` in the above example. Though several functions are partial, they can be extended to total functions just by adding a rule `F(_,...) = ()` with a wild-card pattern _.

Though a recursive XML processing is a handy style for XML transformations over forests, it frequently causes inefficiency of memory usage and execution time because the entire XML stream has to be read and passed to construct the complete input tree before the computation takes place. It is quite harmful in particular when the input is extremely long.

XML stream processing improves the efficiency such as [21]. It minimizes memory usage and execution time by not storing trees in memory. An XML stream processor begins to output the transformation result before reading the whole input. A program written in stream processing style simply consists of initial buffered value v_0 and event-associated function $\mathcal{P}$ which takes the current buffered value and an input event and returns a value to be buffered and a part of output where an input event is <σ>, </σ>, *str* or the end-of-file event `EOF`.

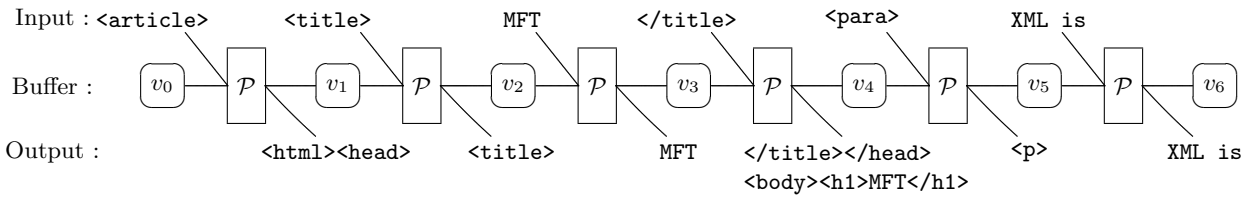

Figure 2: Example of stream processing flow

XML processing proceeds with updating a buffer as shown in Figure 2. First, the buffer has the initial value v_0. After that it processes the input stream depending on the event and the current buffered value for each event read. Now the current event is the begin tag `<article>` and the current buffered value is v_0. Then the function $\mathcal{P}$ is called with the arguments the event `<article>` and the buffered value v_0. In the result, a part of the transformation result `<html><head>` is output and the buffered value is updated to v_1. When the next event `<title>` is read the function $\mathcal{P}$ is called in a similar way. Note that enough information should be stored in the buffer since a stream processor cannot backtrack on the input stream in general. For example, v_3 should comprehend all child nodes of `title` element so that the stream processor can output an `h1` element with these nodes.

While stream processing saves memory usage and execution time, it is much harder to write a program in stream processing style than in functional XML processing style because complicated stateful programming is required.

This paper presents a method to automatically derive the XML stream processor from an XML transformation program written as recursive XML processing. To be more precise we give a method to obtain an XML stream processing program from a mft. The method is based on the composition of a top-down tree transducer (tdtt) and a mft. The tdtt represents an XML parser which transforms XML streams into XML forests. Though we need a stack in order to parse XMLs by tdtt [15, 14], it can be simulated as an infinitary tdtt, that is a tdtt with infinite number of states. The composition is done in a way similar to that of a (finitary) tdtt and a mtt presented by Engelfriet and Vogler [5]. Though it has not been proved that an infinitary tdtt and a mft can be composed by their method, we directly prove that the XML stream processor obtained by our method behaves equivalently to the original mft in this paper.

Related Work

Several researchers have discussed the automatic derivation of XML stream processors from declarative programs. Most of them, however, deals with only query languages including XPath [1, 4, 6, 7] and a subset of XQuery [11]. These querying languages are not expressive enough to specify XML transformation. For example, they could not define the structure-preserved transformation, e.g., renaming the label `a` to `b`. In recursive functional style, we can easily deal with this kind of transformation.

The key idea of our framework was presented in the author's preceding work [15, 16]. The previous work is based on the composition of (stack-)attributed tree transducers [14]. The author has released the XML transformation language XTiSP [15, 13]. All programs definable in XTiSP can be translated into attributed tree transducers. It is well known that attributed tree transducers are less expressive than macro tree transducers [5], i.e., our result in this paper is more powerful than before. Moreover, since the previous framework [15] does not give the formal model of stream processors, some part of the implementation of XTiSP is ad-hoc and that contains inefficient evaluation.

Kiselyov [9] gave an XML parser with a general folding function `foldts` over rose trees. They define an XML transformation by applying three actions `fup`, `fdown` and `fhere` to `foldts`. These actions specify how to accumulate the seed value. This programming style is not user-friendly and many function closures are stored during the processing. Furthermore, his framework does not mention whether the processor can output a part of the result when reading a single XML event, e.g., a begin tag `<a>`.

STX [3] is a template-based XML transformation language that operates on stream of SAX [21] events. While the programmers can define the XML transformation program as well as XSLT [22], they have to explicitly write when and how to store the temporary information like stream processing style.

TransformX presented by Scherzinger and Kemper [18] gives the framework for syntax-directed transformations of XML streams. We can obtain XML stream processors by defining a kind of attribute grammar on the regular tree of the type schema for inputs. Even in their framework, however, we must still keep in mind which

information should be buffered before and after reading each subtree in the input.

Kodama, Suenaga, Kobayashi and Yonezawa [10, 19] propose a translation method from tree processing programs to XML stream processors where the programmer does not have to consider which information should be buffered. Their tree processing language only deals with binary trees which can be easily parsed without end tags, that is rather far from practical XML transformation languages.

Outline

In Section 2, we introduce a simplified model of XML documents and macro forest transducers. In Section 3 we give the formal model of XML stream processors and its derivation from our transducers. We discuss an extension of our framework for applying the existing functional XML transformation languages in Section 4. Finally Section 5 concludes the paper.

2 Model of XML and its Transformation

This section formalizes a model of XML documents. For simplicity, we deal with a simplified model of XML documents. Firstly, we deal with only element nodes. Our framework is easily extended for other kinds of nodes, such as text nodes and attributes. Secondly, we assume that the input XML is *well-formed*, i.e., all begin/end tags are balanced. Therefore, in the input and output, we ignore the names of end tags The names can be recovered by keeping a stack of names whose size coincides with the depth of the XML tree.

2.1 Trees, Forests and XML Streams

Let Σ be an alphabet. Then Σ-*trees* and Σ-*forests* are defined by the following syntax:

$$t ::= \sigma\,\texttt{[}f\texttt{]} \;\mid\; \texttt{\%[}\sigma\texttt{]} \qquad\qquad f ::= \texttt{()} \;\mid\; tf$$

where $\sigma \in \Sigma$ and `()` denotes the empty forest. Σ-forest is also called Σ-hedge [12] and seen as a list of rose trees. Every tree $t \in \mathcal{T}_\Sigma$ can be seen as a forest $t\texttt{()} \in \mathcal{F}_\Sigma$ even if it is written as t. We denote $\mathcal{T}_\Sigma$ and $\mathcal{F}_\Sigma$ for sets of Σ-trees and Σ-forests, respectively. A Σ-tree `a[%[bar]b[%[foo]()]()]` with certain Σ corresponds to an XML fragment `<a>bar<b>foo</b></a>`. For two forests $f_1, f_2 \in \mathcal{F}_\Sigma$, we write $f_1 f_2$ for a Σ-forest $t_1 \dots t_n u_1 \dots u_m\texttt{()}$ where $f_1 = t_1 \dots t_n\texttt{()}$ and $f_2 = u_1 \dots u_m\texttt{()}$.

An XML stream is modeled by a sequence of named begin/end tags and texts. The model of XML stream is defined by a sequence of Σ-*events* which is an alphabet $\{\texttt{<}\sigma\texttt{>} \mid \sigma \in \Sigma\} \cup \{\texttt{</}\sigma\texttt{>} \mid \sigma \in \Sigma\} \cup \Sigma$ denoted by $\Sigma_{\texttt{</>}}$, provided that the sequence is well-formed, i.e., every begin tag has a corresponding end tag and vice versa. For instance, an XML fragment `<a>bar<b>foo</b></a>` is represented in our model by the sequence of six symbols, `<a>`, `bar`, `<b>`, `foo`, `</b>` and `</a>`. We denote by $\Sigma^*_{\texttt{</>}}$ a set of well-formed sequences of Σ-events and denote by ε the empty sequence. The set $\Sigma^*_{\texttt{</>}}$ is a subset of $\mathcal{F}_{\Sigma_{\texttt{</>}}}$ where every tree has no child. The symbol `EOF` denotes the end of an XML stream, which is also regarded as an event. We write $\Sigma_{\texttt{</>EOF}}$ for $\Sigma_{\texttt{</>}} \cup \{\texttt{EOF}\}$.

Let Σ be an alphabet. The *streaming* of a forest is the function $\lfloor\_\_\rfloor : \mathcal{F}_\Sigma \to \Sigma_{\texttt{</>}}$ defined by

$$\lfloor\sigma\,\texttt{[}f_1\texttt{]}\,f_2\rfloor = \texttt{<}\sigma\texttt{>}\,\lfloor f_1\rfloor\,\texttt{</}\sigma\texttt{>}\,\lfloor f_2\rfloor \qquad \lfloor\texttt{\%[}\sigma\texttt{]}\,f\rfloor = \sigma\;\lfloor f\rfloor \qquad \lfloor\texttt{()}\rfloor = \varepsilon.$$

For example, $\lfloor\texttt{a[\%[bar]b[\%[foo]()]()]}\rfloor = \texttt{<a>barfoo}$.

2.2 Macro Forest Transducers

Macro forest transducers (for short, mft) were proposed by Perst and Seidl [17] to define transformations from forests to (sets of) forests. They extend *macro tree transducer* (for short, mtt) [5] with the concatenation operator for forests as a primitive. Let us write $\mathbb{N}$ for the set of non-negative integers including 0.

Definition 2.1 *A* macro forest transducer *(mft) is a tuple $M = (Q, \Sigma, \Delta, in, R)$, where*

- *Q is a finite set of ranked states whose ranks are obtained by rank $: Q \to \mathbb{N} \setminus \{0\}$,*
- *Σ and Δ are alphabets with $Q \cap (\Sigma \cup \Delta) = \emptyset$, called the* input alphabet *and the* output alphabet*, respectively,*
- *$in \in Q$ is the initial ranked state,*

- *R is a set of rules such that $R = \bigcup_{q \in Q} R_q$ with sets R_q of* q-rules *of the form $q(pat, y_1, \ldots, y_n) \to rhs$ with $rank(q) = n + 1$ and* variables *y_i where*
  - *pat is either $\texttt{()}$ or $\sigma\texttt{[}x_1\texttt{]}x_2$ with $\sigma \in \Sigma$,*
  - *rhs ranges over expressions defined by*

$$rhs ::= q'(x_i, rhs, \ldots, rhs) \mid y_j \mid \texttt{()} \mid \delta\texttt{[}rhs\texttt{]} \mid \texttt{\%[}\delta\texttt{]} \mid rhs\ rhs$$

  with $q' \in Q$, $\delta \in \Delta$, $i = 1, 2$ and $j = 1, \ldots, n$. Additionally, no variable x_i occurs in rhs when $pat = \texttt{()}$.

Next we define the semantics of mft's such that every state is translated into a function with accumulating parameters following [17].

Definition 2.2 *Let $M = (Q, \Sigma, \Delta, in, R)$ be a mft and $f \in \mathcal{F}_\Sigma$. The* semantics *of states $q \in Q$ with $n = rank(q)$ is given by the function $[\![q]\!] : \mathcal{F}_\Sigma \times (\mathcal{F}_\Delta)^n \to \mathcal{F}_\Delta$. The functions are inductively defined by q-rules in M as follows:*

- *$[\![q]\!](\texttt{()}, \varphi_1, \ldots, \varphi_n) = [\![rhs]\!]_\rho$ where $(q(\texttt{()}, y_1, \ldots, y_n) \to rhs) \in R$ and $\rho(y_j) = \varphi_j$ for $j = 1, \ldots, n$,*
- *$[\![q]\!](\sigma\texttt{[}\omega_1\texttt{]}\omega_2, \varphi_1, \ldots, \varphi_n) = [\![rhs]\!]_\rho$ where $(q(\sigma\texttt{[}x_1\texttt{]}x_2, y_1, \ldots, y_n \to rhs) \in R$, $\rho(x_i) = \omega_i$ for $i = 1, 2$ and $\rho(y_j) = \psi_j$ for $j = 1, \ldots, n$,*
- *$[\![q]\!](\texttt{\%[}\sigma\texttt{]}\omega, \varphi_1, \ldots, \varphi_n) = [\![rhs]\!]_\rho$ where $(q(\texttt{\%[}\sigma\texttt{]}x, y_1, \ldots, y_n \to rhs) \in R$, $\rho(x) = \omega$ for $i = 1, 2$ and $\rho(y_j) = \psi_j$ for $j = 1, \ldots, n$,*

where $[\![\_\_]\!]_\rho$ denotes the evaluation of a right-hand side expression for states with respect to the binding ρ of the formal parameters x_i and y_j that is defined by

$$[\![q'(x_i, rhs_1, \ldots, rhs_m)]\!]_\rho = [\![q']\!](\rho(x_i), [\![rhs_1]\!]_\rho, \ldots, [\![rhs_m]\!]_\rho)$$
$$[\![y_j]\!]_\rho = \rho(y_j) \qquad [\![\texttt{()}]\!]_\rho = \texttt{()}$$
$$[\![\delta\texttt{[}rhs\texttt{]}]\!]_\rho = \delta\texttt{[}[\![rhs]\!]_\rho\texttt{]} \qquad [\![\texttt{\%[}\delta\texttt{]}]\!]_\rho = \texttt{\%[}\delta\texttt{]} \qquad [\![rhs\ rhs']\!]_\rho = [\![rhs]\!]_\rho [\![rhs']\!]_\rho.$$

Our definition is different from [17] in that we deal with text nodes explicitly and consider only deterministic mft's, i.e., every semantics of states ranges over a set of forests instead of a power set of them. Note that the semantics of a state is a partial function when there is no rule for the state and a certain pattern. For such uncovered state and pattern, we assume that the mft implicitly has rules whose right-hand side is a leaf. Then we can claim that the semantics is total. In the rest of paper, we deal with only total mft's though we may omit these additional rules.

The transformation of a forest f by an mft is defined by applying the semantics of the initial state to f and an adequate number of leaves.

Definition 2.3 *The* transformation *induced by a mft $M = (Q, \Sigma, \Delta, in, R)$ is the function $\tau_M : \mathcal{F}_\Sigma \to \mathcal{F}_\Delta$ defined by*

$$\tau_M(f) = [\![in]\!](f, \texttt{()}, \ldots, \texttt{()}).$$

We show two examples of XML transformation written in mft's. First example M_{htm} is an XML transformation shown in Section 1.

Example 2.4 *Let the mft $M_{htm} = (Q, \Sigma, \Delta, Main, R)$ be defined by*

$Q = \{Main, Title, InArticle, Key2Em, AllKeys, Copy\}$,

$\Sigma = \Delta =$ *(proper alphabet)*,

$R = \{$ $Main(\texttt{article[}x_1\texttt{]}x_2) \to \texttt{html[head[}Title(x_1)\texttt{]body[}InArticle(x_1, \texttt{()})\texttt{]()]}$,

$Title(\texttt{title[}x_1\texttt{]}x_2) \to \texttt{title[}Copy(x_1)\texttt{]}$,

$InArticle(\texttt{title[}x_1\texttt{]}x_2, y_1) \to \texttt{h1[}Copy(x_1)\texttt{]}\,InArticle(x_2, y_1)$,

$InArticle(\texttt{para[}x_1\texttt{]}x_2, y_1) \to \texttt{p[}Key2Em(x_1)\texttt{]}\,InArticle(x_2, y_1\,AllKeys(x_1))$,

$InArticle(\texttt{postscript[}x_1\texttt{]}x_2, y_1) \to \texttt{h2[\%[Index]] ul[}y_1\texttt{] h2[\%[Postscript]]}\ Copy(x_1)$,

$Key2Em(\texttt{key[}x_1\texttt{]}x_2) \to \texttt{em[}Copy(x_1)\texttt{]}\ Key2Em(x_2)$,

$Key2Em(\texttt{\%[}\sigma\texttt{]}x) \to \texttt{\%[}\sigma\texttt{]}\,Key2Em(x) \quad (\sigma \in \Sigma), \qquad Key2Em(\texttt{()}) \to \texttt{()}$,

$AllKeys(\texttt{key[}x_1\texttt{]}x_2) \to \texttt{li[}Copy(x_1)\texttt{]}\,AllKeys(x_2), \qquad AllKeys(\texttt{\%[}\sigma\texttt{]}x) \to AllKeys(x) \quad (\sigma \in \Sigma)$,

$AllKeys(\texttt{()}) \to \texttt{()}$,

$$Copy(\sigma\,\texttt{[}x_1\texttt{]}\,x_2) \to \sigma\,\texttt{[}Copy(x_1)\texttt{]}\,Copy(x_2) \qquad (\sigma \in \Sigma), \qquad Copy(\texttt{()}) \to \texttt{()}\ \}$$

Almost rules of M_{htm} are the same as the function definition in Figure 1. However, the variables matched with a pattern cannot occur in the right-hand side of rules except for the case where they are used as the first argument of the states according to the definition of *rhs*. For example, the right-hand side of the definition of `Title` is `title[$x1]` in Figure 1. The definition of mft's does not allows the expression `title[`x_1`]` in a right-hand side of rules. The mft M_{htm} solves the problem by using a state *Copy* whose semantics is an identity function, i.e., we can use `title[`$Copy(x_1)$`]` instead of `title[`x_1`]`.

Second example of a mft represents an XML transformation which reverses all descendants of `rev` node in the input. For instance, when the input XML fragment is

```
<a>
  <rev><b><c></c><d></d></b><e></e></rev>
  <f><rev><g></g><h></h></rev></f>
</a>,
```

the transformation returns

```
<a>
  <rev><e></e><b><d></d><c></c></b></rev>
  <f><rev><h></h><g></g></rev></f>
</a>.
```

The transformation can be given by an mft with only two states.

Example 2.5 *Let the mft* $M_{mir} = (Q, \Sigma, \Delta, Main, R)$ *be defined by*

$Q = \{Main, Rev\}$, $\Sigma = \Delta =$ *(proper alphabet),*

$R = \{\ Main(\texttt{rev[}x_1\texttt{]}\,x_2) \to \texttt{rev[}Rev(x_1, \texttt{()})\texttt{]}\,Main(x_2),$

$Main(\sigma\,\texttt{[}x_1\texttt{]}\,x_2) \to \sigma\,\texttt{[}Main(x_1)\texttt{]}\,Main(x_2) \quad (\sigma \neq \texttt{rev}), \quad Main(\texttt{\%[}\sigma\texttt{]}\,x) \to \texttt{\%[}\sigma\texttt{]}\,Main(x) \quad (\sigma \neq \texttt{rev}),$

$Main(\texttt{()}) \to \texttt{()},$

$Rev(\sigma\,\texttt{[}x_1\texttt{]}\,x_2, y_1) \to Rev(x_2, \sigma\,\texttt{[}Rev(x_1, \texttt{()})\texttt{]}\,y_1) \quad Rev(\texttt{\%[}\sigma\texttt{]}\,x, y_1) \to Rev(x, \texttt{\%[}\sigma\texttt{]}\,y_1) \quad (\sigma \in \Sigma),$

$Rev(\texttt{()}, y_1) \to y_1\ \}.$

3 XML Stream Processors and Its Derivation

This section presents a formal model of XML stream processors and its derivation method based on the composition of tree transducers. Since the set $\Sigma^*_{\texttt{</>}}$ is a subset of $\mathcal{F}_{\Sigma_{\texttt{</>}}}$, XML stream processor (for short, xsp) can be defined in a way similar to the definition of tree transducers such as mft's.

3.1 XML Stream Processors

An XML stream processor proceeds an XML transformation by updating the buffered value. In our framework, we consider a partially-evaluated result, called *temporary expression*, as the buffered value. The value will be the transformation result itself after all input events are read. Additionally, the stream processor can output a part of the transformation result by squeezing some decided output events at the head of the temporary expression before completing reading all input events.

Our XML stream processor consists of rules which specifies how to update the temporary expression

Definition 3.1 *An* XML stream processor *(xsp) is a tuple* $S = (Q, \Sigma, \Delta, in, R)$*, where*

- Q *is a set of ranked states, which may be countably infinite and the rank for each state is obtained by* $rank : Q \to \mathbb{N}$*,*
- Σ *and* Δ *are (finite) alphabets with* $Q \cap (\Sigma \cup \Delta) = \emptyset$*, called the* input alphabet *and the* output alphabet*, respectively,*
- $in \in Q$ *is the initial state,*

- *R is a set of rules such that $R = \{r_{(q,\chi)} \mid q \in Q, \chi \in \Sigma_{\texttt{</>EOF}}\}$ with (q,χ)-*rules *$r_{(q,\chi)}$ of the form*

$$q(y_1, \ldots, y_n) \xrightarrow{\chi} rhs$$

with variables *y_j where $n = rank(q)$ and rhs ranges over expressions defined by*

$$rhs ::= q'(rhs, \ldots, rhs) \mid y_j \mid \varepsilon \mid \texttt{<}\delta\texttt{>}rhs\texttt{</}\delta\texttt{>} \mid \delta \mid rhs\ rhs$$

where $q' \in Q$, $\delta \in \Delta$ and $j = 1, \ldots, n$. Additionally, the pattern $q'(\_\_)$ does not occur in rhs for any $q' \in Q$ when $\chi = \texttt{EOF}$.

3.2 Semantics of XML Stream Processors

The definition of semantics of states in an xsp is different from that of states in a mft because a rule of an xsp specifies how to update the temporary expression for each input event. Temporary expressions range over output XML streams with a number of unknown parts given by using states with arguments.

Definition 3.2 *Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp. A* temporary expression *E for S is defined by the following syntax:*

$$E ::= q(E, \ldots, E) \mid \varepsilon \mid \texttt{<}\delta\texttt{>}E\texttt{</}\delta\texttt{>} \mid \delta \mid E\ E$$

where $q \in Q$, $\delta \in \Delta$. We denote the set of temporary expressions by Tmp_S.

The semantics of an xsp is defined by translating every rule of the xsp into a transition for temporary expressions.

Definition 3.3 *Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp and $s \in \Sigma^*_{\texttt{</>}}$. The* transition *over Tmp_S for an input Σ-event is a function $\langle\!|\_\_, \_\_|\!\rangle : Tmp_S \times \Sigma_{\texttt{</>EOF}} \to Tmp_S$. The function is defined with another transition over Tmp_S which is a function $\langle\!|\_\_, \_\_|\!\rangle : Tmp_S \times \Sigma_{\texttt{</>EOF}} \to Tmp_S$. The definition use the evaluation function $[\![\_\_]\!]_\rho : rhs \to Tmp_S$ for right-hand side expressions with respect to the binding ρ of the formal parameters in the left-hand side. We give the definition as follows:*

- *$\langle\!|\_\_|\!\rangle$ are defined by*
  - *$\langle\!|q(E_1, \ldots, E_n), \chi|\!\rangle = [\![rhs]\!]_\rho$ where $(q(y_1, \ldots, y_n) \xrightarrow{\chi} rhs) \in R$ with $q \in Q$, $\chi \in \Sigma_{\texttt{</>EOF}}$, $\rho(y_j) = \langle\!|E_j, \chi|\!\rangle$ for $j = 1, \ldots, n$,*
  - *$\langle\!|\varepsilon, \chi|\!\rangle = \varepsilon$, $\langle\!|\texttt{<}\delta\texttt{>}E\texttt{</}\delta\texttt{>}, \chi|\!\rangle = \texttt{<}\delta\texttt{>}\langle\!|E, \chi|\!\rangle\texttt{</}\delta\texttt{>}$, and $\langle\!|\delta, \chi|\!\rangle = \delta$ where $\delta \in \Delta$,*
  - *$\langle\!|E\ E', \chi|\!\rangle = \langle\!|E, \chi|\!\rangle\ \langle\!|E', \chi|\!\rangle$,*
- *$[\![\_\_]\!]_\rho$ is defined by*

$$[\![q'(rhs_1, \ldots, rhs_m)]\!]_\rho = q'([\![rhs_1]\!]_\rho, \ldots, [\![rhs_m]\!]_\rho)$$
$$[\![y_j]\!]_\rho = \rho(y_j) \qquad [\![\varepsilon]\!]_\rho = \varepsilon$$
$$[\![\texttt{<}\delta\texttt{>}rhs\texttt{</}\delta\texttt{>}]\!]_\rho = \texttt{<}\delta\texttt{>}[\![rhs]\!]_\rho\texttt{</}\delta\texttt{>} \qquad [\![\delta]\!]_\rho = \delta \qquad [\![rhs\ rhs']\!]_\rho = [\![rhs]\!]_\rho\ [\![rhs']\!]_\rho.$$

XML processing reads the input events one by one. For each reading step, the processor computes something with stored information and store a new information for the next step. The transition $\langle\!|\_\_|\!\rangle$ defines how the processor computes the next information for each input event. In our framework, the information is represented by a temporary expression. Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp and $\chi_1\chi_2 \ldots \chi_k$ be an XML stream with $\chi_j \in \Sigma_{\texttt{</>}}$ for $j = 1, 2, \ldots, k$. The initial information is represented by $in(\varepsilon, \ldots, \varepsilon)$. When finding the end of the input XML stream, the transition for `EOF` is applied to the current information. The final information is the transformation result itself. Thus we obtain the transformation result by

$$\langle\!|\langle\!| \ldots \langle\!|\langle\!|in(\varepsilon, \ldots, \varepsilon), \chi_1|\!\rangle, \chi_2|\!\rangle, \ldots, \chi_k|\!\rangle, \texttt{EOF}|\!\rangle. \tag{1}$$

This transformation is not what we require as XML processing, however, because the XML stream processor should output part of the result if possible before reading the whole input.

We give two definitions of transformation induced by an xsp. One is called *non-squeezing*. The definition is simply given as represented in (1). Another is called *squeezing*. The squeezing transformation achieve the

best result possible, that is, the output written so far always the largest that can be determined from the input read so far. Stream processing with squeezing is a desirable behavior of real XML stream processors which can start to output a part of the result for each input event. Since squeezing will collapse the syntax of temporary expressions, we define *collapsed temporary expressions* $Tmp_S^{\times}$ with an xsp S by

$$E ::= q(E, \ldots, E) \mid \varepsilon \mid \texttt{<}\delta\texttt{>}E \mid \texttt{</}\delta\texttt{>}\ E \mid \delta\ E$$

where q is a state of S and δ is an output symbol of S. We can easily confirm that $Tmp_S \subset Tmp_S^{\times}$.

Definition 3.4 *1. The* non-squeezing transformation *induced by an xsp $S = (Q, \Sigma, \Delta, in, R)$ is the function $\tau_S : \Sigma^*_{\texttt{</>}} \to \Delta^*_{\texttt{</>}}$ defined by $\tau_S(s) = \theta_S(in(\varepsilon, \ldots, \varepsilon), s\texttt{EOF})$ where*

$$\theta_S(e, \varepsilon) = e \qquad \theta_S(e, \chi s) = \theta_S(\langle\!|e, \chi|\!\rangle, s)$$

for $e \in Tmp_S$.

2. The squeezing transformation *induced by an xsp $S = (Q, \Sigma, \Delta, in, R)$ is the function $\tau_S : \Sigma^*_{\texttt{</>}} \to \Delta^*_{\texttt{</>}}$ defined by $\tau_S(s) = \eta_S(in(\varepsilon, \ldots, \varepsilon), s\texttt{EOF}, \varepsilon)$ where for $e \in Tmp_S$*

$$\eta_S(e, \varepsilon, b) = be \qquad \eta_S(e, \chi s, b) = \eta_S(e', s, bs').$$

with $(e', s') = sqz(\langle\!|e, \chi|\!\rangle)$ and a sqeeze *function $sqz : Tmp_S^{\times} \to Tmp_S^{\times} \times \Delta^*_{\texttt{</>}}$ is defined by*

$$sqz(q(e_1, \ldots, e_n)) = (q(e_1, \ldots, e_n), \varepsilon) \quad sqz(\varepsilon) = (\varepsilon, \varepsilon) \quad sqz(\texttt{<}\delta\texttt{>}e_1) = (e_1', \texttt{<}\delta\texttt{>}s_1')$$

$$sqz(\texttt{</}\delta\texttt{>}e_1) = (e_1', \texttt{</}\delta\texttt{>}s_1') \quad sqz(\delta e_1) = (e_1', \delta s_1') \quad sqz(e_1\ e_2) = \begin{cases} (e_2', s_1's_2') & \text{if } e_1' = \varepsilon \\ (e_1'e_2, s_1') & \text{otherwise} \end{cases}$$

where $(e_1', s_1') = sqz(e_1)$ and $(e_2', s_2') = sqz(e_2)$.

The non-squeezing transformation uses the auxiliary function θ which takes two arguments, the current information as a temporary expression and the rest of the stream, and returns the next information. On the other hand, the squeezing transformation uses the auxiliary function η which takes three arguments adding one extra argument to those of θ. The extra argument is used for output buffer which does not change during the computation except for adding some events to the tail of the original buffer. In the output buffer, the second element of the result of the squeeze function *sqz* is added as a possibly-known part at the head of the result. It is easy to show that

$$s'e' = e \quad \text{if} \quad (e', s') = sqz(e) \tag{2}$$

for $e \in Tmp_S$ by induction on the structure of e.

The following lemma shows that the non-squeezing transformation and the squeezing transformation are equivalent. In the rest of the paper, we employ the non-squeezing transformation instead of the squeezing one to compare the behavior of a mft and an xsp since it is simpler than the squeezing transformation, although the implementation of XML stream processor does employ the squeezing transformation.

Lemma 3.5 *Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp and $s \in \Sigma^*_{\texttt{</>}}$. Then we have*

$$\theta_S(in(\varepsilon, \ldots, \varepsilon), s) = \eta_S(in(\varepsilon, \ldots, \varepsilon), s, \varepsilon) \tag{3}$$

where θ_S and η_S are as given in Definition 3.4.

Proof. We prove the equation

$$\theta_S(be, s) = \eta_S(e, s, b) \tag{4}$$

for $b \in \Delta^*_{\texttt{</>}}$ and $e \in Tmp_S$, which is more general than (3). Equation (3) is the special case of (4) in which $b = \varepsilon$ and $e = in(\varepsilon, \ldots, \varepsilon)$. We show at the same time

$$\eta_S(e, s, b) = \eta_S(be, s, \varepsilon) \tag{5}$$

for $s, b \in \Delta^*_{\texttt{</>}}$ and $e \in Tmp_S$.

Equations (4) and (5) are proved by induction on the length $\sharp s$ of s. If $\sharp s = 0$, then both (4) and (5) are the same, that is be.

If $\sharp s > 0$, then suppose that $s = \chi s'$ with $\chi \in \Sigma_{\texttt{</>}}$ and $s' \in \Sigma^*_{\texttt{</>}}$ and that $b = \xi_1 \dots \xi_n$ $(n \geq 0)$ with $\xi_j \in \Delta_{\texttt{</>}}$ for $j = 1, \cdots, n$. The left-hand side of (4) is

$$\begin{aligned}\theta_S(\xi_1 \dots \xi_n e, \chi s') &= \theta_S(\langle\!|\xi_1 \dots \xi_n e, \chi|\!\rangle, s') \\ &= \theta_S(\xi_1 \dots \xi_n \langle\!|e, \chi|\!\rangle, s') \\ &= \eta_S(\langle\!|e, \chi|\!\rangle, s', \xi_1 \dots \xi_n) \\ &= \eta_S(\xi_1 \dots \xi_n \langle\!|e, \chi|\!\rangle, s', \varepsilon)\end{aligned}$$

from the definitions of θ and $\langle\!|\_\_|\!\rangle$ and the induction hypotheses of (4) and (5). When $(e'', s'') = sqz(\langle\!|e, \chi|\!\rangle)$, the right-hand side of (4) is

$$\begin{aligned}\eta_S(e, \chi s', \xi_1 \dots \xi_n) &= \eta_S(e'', s', \xi_1 \dots \xi_n s'') \\ &= \eta_S(\xi_1 \dots \xi_n s'' e'', s', \varepsilon)\end{aligned}$$

from the definitions of η_S and the induction hypothesis of (5). Both sides of (4) are the same since we have $s''e'' = \langle\!|e, \chi|\!\rangle$ by (2). Hence (4) holds.

From the definition of η_S, the induction hypothesis of (5) and $(e'', s'') = sqz(\langle\!|e, \chi|\!\rangle)$, the left-hand side of (5) is

$$\begin{aligned}\eta_S(e, \chi s', \xi_1 \dots \xi_n) &= \eta_S(e'', s', \xi_1 \dots \xi_n s'') \\ &= \eta_S(\xi_1 \dots \xi_n s'' e'', s', \varepsilon).\end{aligned}$$

Since we have $sqz(\langle\!|\xi_1 \dots \xi_n e, \chi|\!\rangle) = sqz(\xi_1 \dots \xi_n \langle\!|e, \chi|\!\rangle) = (e'', \xi_1 \dots \xi_n s'')$ from the definitions of $\langle\!|\_\_|\!\rangle$ and sqz, the right-hand side of (5) is

$$\begin{aligned}\eta_S(\xi_1 \dots \xi_n e, \chi s', \varepsilon) &= \eta_S(e'', s', \xi_1 \dots \xi_n s'') \\ &= \eta_S(\xi_1 \dots \xi_n s'' e'', s'\varepsilon)\end{aligned}$$

from the definition of η_S and the induction hypothesis of (5). Hence (5) holds. ∎

3.3 Derivation of XML Stream Processors

The derivation of an xsp from a given mft is achieved in a similar way to the existing method by Engelfriet and Vogler [5] to synthesize two tree transducers, a top-down tree transducer (for short, tdtt) and a macro tree transducer (for short, mtt). That is because an XML parser which transforms XML streams to forests (binary labeled trees) can be represented by an infinitary tdtt and a mft is a simple extension of a mtt. Therefore we can give a derivation method of an xsp just as a straightforward extension of the existing method. Correctness of our method will be shown in the next subsection.

Definition 3.6 *Let $M = (Q, \Sigma, \Delta, in, R)$ be a mft. We define an xsp $\mathcal{SP}(M) = (Q', \Sigma, \Delta, in', R')$ where*

- *$Q' = \{q[i] \mid q \in Q, i \in \mathbb{N}\}$ where $rank(q[i]) = rank(q) - 1$ for every $q \in Q$ and $i \in \mathbb{N}$,*
- *$in' = in[0] \in Q$,*
- *R' contains the following rules:*
  - *For every $q \in Q$, $\sigma \in \Sigma$ and $(q(\sigma\,\texttt{[}x_1\texttt{]}\,x_2, y_1, \dots, y_n) \to rhs) \in R$, the $(q[0], \texttt{<}\sigma\texttt{>})$-rule in R' is*

$$q[0](y_1, \dots, y_n) \xrightarrow{\texttt{<}\sigma\texttt{>}} \mathcal{A}(rhs),$$

  - *For every $q \in Q$, $\sigma \in \Sigma$ and $(q(\texttt{\%[}\sigma\texttt{]}\,x, y_1, \dots, y_n) \to rhs) \in R$, the $(q[0], \sigma)$-rule in R' is*

$$q[0](y_1, \dots, y_n) \xrightarrow{\sigma} \mathcal{A}(rhs[x_1/x])$$

  where $rhs[x_1/x]$ is obtained by replacing x by x_1 in rhs,

– *For every* $q \in Q$, $\sigma \in \Sigma$, $i \in \mathbb{N}$ *and* $q(\texttt{()}, y_1, \ldots, y_n) \to rhs \in R$, *the* $(q[0], \texttt{</}\sigma\texttt{>})$*-rule and* $(q[i], \texttt{EOF})$*-rule in* R' *are*

$$q[0](y_1, \ldots, y_n) \xrightarrow{\texttt{</}\sigma\texttt{>}} \mathcal{A}(rhs), \qquad\qquad q[i](y_1, \ldots, y_n) \xrightarrow{\texttt{EOF}} \mathcal{A}(rhs),$$

respectively,

– *For every* $q \in Q$, $\sigma \in \Sigma$ *and* $i \geq 1$, *the* $(q[i], \texttt{<}\sigma\texttt{>})$*-rule and* $(q[i], \texttt{</}\sigma\texttt{>})$*-rule in* R' *are*

$$q[i](y_1, \ldots, y_n) \xrightarrow{\texttt{<}\sigma\texttt{>}} q[i+1](y_1, \ldots, y_n), \qquad q[i](y_1, \ldots, y_n) \xrightarrow{\texttt{</}\sigma\texttt{>}} q[i-1](y_1, \ldots, y_n),$$

respectively,

where $\mathcal{A}$ *is defined over right-hand side expressions of rules in mft's as follows:*

$$\begin{array}{lll}
\mathcal{A}(q'(x_1, rhs_1, \ldots, rhs_m)) = q'[0](\mathcal{A}(rhs_1), \ldots, \mathcal{A}(rhs_m)) & & \\
\mathcal{A}(q'(x_2, rhs_1, \ldots, rhs_m)) = q'[1](\mathcal{A}(rhs_1), \ldots, \mathcal{A}(rhs_m)) & & \\
\mathcal{A}(y_j) = y_j & \mathcal{A}(\texttt{()}) = \varepsilon & \\
\mathcal{A}(\delta\texttt{[}rhs\texttt{]}) = \texttt{<}\delta\texttt{>}\mathcal{A}(rhs)\texttt{</}\delta\texttt{>} & \mathcal{A}(\texttt{\%[}\delta\texttt{]}) = \delta & \mathcal{A}(rhs\ rhs') = \mathcal{A}(rhs)\ \mathcal{A}(rhs')
\end{array}$$

Now we show two examples of derivation of xsp's from a mft M_{htm} of Example 2.4 and a mft M_{mir} of Example 2.5. Additionally we illustrate how the obtained xsp $\mathcal{SP}(M_{htm})$ works for a certain input XML stream. In these examples, we omit some of rules whose right hand side is ε they are derived from omitted rules whose right-hand side is a leaf in the original mft.

Example 3.7 *The derivation method gives an xsp* $\mathcal{SP}(M_{htm}) = (Q', \Sigma, \Delta, Main[0], R')$ *from the mft* $M_{htm} = (Q, \Sigma, \Delta, Main, R)$ *in* Example 2.4 *where*

$Q' = \{q[i] \mid q \in Q, i \in \mathbb{N}\}$,

$R' = \{$ $Main[0]() \xrightarrow{\texttt{<article>}}$ <html> <head> $Title[0]()$ </head> <body> $InArticle[0](\varepsilon)$ </body> </html>,

$Title[0]() \xrightarrow{\texttt{<title>}}$ <title> $Copy[0]()$ </title>,

$InArticle[0](y_1) \xrightarrow{\texttt{<title>}}$ <h1> $Copy[0]()$ </h1> $InArticle[1](y_1)$,

$InArticle[0](y_1) \xrightarrow{\texttt{<para>}}$ $Key2Em[0]()$ $InArticle[1](y_1\ AllKeys[0]())$,

$InArticle[0](y_1) \xrightarrow{\texttt{<postscript>}}$ <h2> Index </h2> y_1 <h2> Postscript </h2> $Copy[0]()$,

$Key2Em[0]() \xrightarrow{\texttt{<key>}}$ $Copy[0]()$ $Key2Em[1]()$, $\quad Key2Em[0]() \xrightarrow{\sigma} \sigma\ Key2Em[1]() \quad (\sigma \in \Sigma)$,

$AllKeys[0]() \xrightarrow{\texttt{<key>}}$ $Copy[0]()$ $AllKeys[1]()$, $\quad AllKeys[0]() \xrightarrow{\sigma} AllKeys[0]() \quad (\sigma \in \Sigma)$,

$Copy[0]() \xrightarrow{\texttt{<}\sigma\texttt{>}} \texttt{<}\sigma\texttt{>}\ Copy[0]()\ \texttt{</}\sigma\texttt{>}\ Copy[1]() \quad (\sigma \in \Sigma)$, $\quad Copy[0]() \xrightarrow{\sigma} \sigma\ Copy[0]() \quad (\sigma \in \Sigma)$,

$q[i]() \xrightarrow{\texttt{<}\sigma\texttt{>}} q[i+1]() \quad (\sigma \in \Sigma,\ i \geq 1,\ q \neq InArticle)$,

$q[i]() \xrightarrow{\sigma} q[i]() \quad (\sigma \in \Sigma,\ i \geq 1,\ q \neq InArticle)$

$q[i]() \xrightarrow{\texttt{</}\sigma\texttt{>}} q[i-1]() \quad (\sigma \in \Sigma,\ i \geq 1,\ q \neq InArticle)$

$q[i]() \xrightarrow{\chi} \varepsilon \quad ((\chi, i) \in \Sigma_{()},\ q \neq InArticle)$,

$InArticle[i](y_1) \xrightarrow{\texttt{<}\sigma\texttt{>}} InArticle[i+1](y_1) \quad (\sigma \in \Sigma,\ i \geq 1)$,

$InArticle[i](y_1) \xrightarrow{\sigma} InArticle[i](y_1) \quad (\sigma \in \Sigma,\ i \geq 1)$,

$InArticle[i](y_1) \xrightarrow{\texttt{</}\sigma\texttt{>}} InArticle[i-1](y_1) \quad (\sigma \in \Sigma,\ i \geq 1)$,

$InArticle[i](y_1) \xrightarrow{\chi} \varepsilon \quad ((\chi, i) \in \Sigma_{()})$ $\}$,

where $\Sigma_{()} = \{(\texttt{</}\sigma\texttt{>}, 0) \mid \sigma \in \Sigma\} \cup \{(\texttt{EOF}, i) \mid i \in \mathbb{N}\}$.

Let an input XML stream for $\mathcal{SP}(M_{htm})$ be

```
<article> <title> MFT </title> <para> XML is ...
```

$Main[0]()$

$\overset{\texttt{<article>}}{\Longrightarrow}$ `<html> <head>` $Title[0]()$ `</head> <body>` $InArticle[0](\varepsilon)$ `</body> </html>`

$\overset{\texttt{<title>}}{\Longrightarrow}$ `<html> <head> <title>` $Copy[0]()$ `</title> </head>`
`<body> <h1>` $Copy[0]()$ `</h1>` $InArticle[1](\varepsilon)$ `</body> </html>`

$\overset{\texttt{MFT}}{\Longrightarrow}$ `<html> <head> <title> MFT` $Copy[0]()$ `</title> </head>`
`<body> <h1> MFT` $Copy[0]()$ `</h1>` $InArticle[1](\varepsilon)$ `</body> </html>`

$\overset{\texttt{</title>}}{\Longrightarrow}$ `<html> <head> <title> MFT </title> </head> <body> <h1> MFT </h1>` $InArticle[0](\varepsilon)$ `</body> </html>`

$\overset{\texttt{<para>}}{\Longrightarrow}$ `<html> <head> <title> MFT </title> </head>`
`<body> <h1> MFT </h1> <p>` $Key2Em[0]()$ `</p>` $InArticle[1](AllKeys[0]())$ `</body> </html>`

$\overset{\texttt{XML is}}{\Longrightarrow}$ `<html> <head> <title> MFT </title> </head>`
`<body> <h1> MFT </h1> <p> XML is` $Key2Em[0]()$ `</p>` $InArticle[1](AllKeys[0]())$ `</body> </html>`

$\Longrightarrow \cdots$

Figure 3: Stream processing induced by $\mathcal{SP}(M_{htm})$

Then an xsp proceeds as shown in Figure 3 where $\overset{\chi}{\Longrightarrow}$ stands for buffer updating when an input event χ is read. The processing is as expected in Figure 2. For each step, the stream processor outputs the head-determined part by the squeeze function. The remainder is stored in a buffer.

Example 3.8 *The derivation method gives an xsp* $\mathcal{SP}(M_{mir}) = (Q', \Sigma, \Delta, Main[0], R')$ *from the mft* $M_{mir} = (Q, \Sigma, \Delta, Main, R)$ *in* EXAMPLE 2.5 *where*

$Q' = \{q[i] \mid q \in Q, i \in \mathbb{N}\}$,

$R' = \{$ $Main[0]() \xrightarrow{\texttt{<rev>}} \texttt{<rev>}\ Rev[0](\varepsilon)\ \texttt{</rev>}\ Main[1]()$,

$Main[0]() \xrightarrow{\texttt{<}\sigma\texttt{>}} \texttt{<}\sigma\texttt{>}\ Main[0]\ \texttt{</}\sigma\texttt{>}\ Main[1] \quad (\sigma \neq \texttt{rev})$, $\quad Main[0]() \xrightarrow{\sigma} \sigma\ Main[0] \quad (\sigma \in \Sigma)$,

$Main[i]() \xrightarrow{\texttt{<}\sigma\texttt{>}} Main[i+1]() \quad (\sigma \in \Sigma,\ i \geq 1)$, $\quad Main[i]() \xrightarrow{\sigma} Main[i]() \quad (\sigma \in \Sigma,\ i \geq 1)$,

$Main[i]() \xrightarrow{\texttt{</}\sigma\texttt{>}} Main[i-1] \quad (\sigma \in \Sigma,\ i \geq 1)$ $\quad Main[i]() \xrightarrow{\chi} \varepsilon \quad ((\chi, i) \in \Sigma_{()})$,

$Rev[0](y_1) \xrightarrow{\texttt{<}\sigma\texttt{>}} Rev[1](\texttt{<}\sigma\texttt{>}\ Rev[0](\varepsilon)\ \texttt{</}\sigma\texttt{>}\ y_1) \quad (\sigma \in \Sigma)$, $\quad Rev[0](y_1) \xrightarrow{\sigma} Rev[0](\sigma\ y_1) \quad (\sigma \in \Sigma)$,

$Rev[i](y_1) \xrightarrow{\texttt{<}\sigma\texttt{>}} Rev[i+1](y_1) \quad (\sigma \in \Sigma,\ i \geq 1)$, $\quad Rev[i](y_1) \xrightarrow{\sigma} Rev[i](y_1) \quad (\sigma \in \Sigma,\ i \geq 1)$,

$Rev[i](y_1) \xrightarrow{\texttt{</}\sigma\texttt{>}} Rev[i-1](y_1) \quad (\sigma \in \Sigma,\ i \geq 1)$, $\quad Rev[i](y_1) \xrightarrow{\chi} y_1 \quad ((\chi, i) \in \Sigma_{()})$ $\}$,

where $\Sigma_{()} = \{(\texttt{</}\sigma\texttt{>}, 0) \mid \sigma \in \Sigma\} \cup \{(\texttt{EOF}, i) \mid i \in \mathbb{N}\}$.

3.4 Correctness of derivation

The correctness of our derivation of XML stream processors is that, for every mft and every input forest, the XML stream corresponding to the transformation result of the mft for the forest is equal to the transformation result of the xsp obtained by our derivation from the mft. In the rest of section we do not deal with text nodes in forests. The proof can be easily extended for text nodes. Additionally we ignore the names of end tags since we deal with only well-formed XML. We write `</□>` to denote a proper end tag. The name of the end tag can be recovered from the context.

Correctness is stated by the following theorem.

Theorem 3.9 *Let* $M = (Q, \Sigma, \Delta, in, R)$ *be a mft. Then*

$$\tau_{\mathcal{SP}(M)}(\lfloor f \rfloor) = \lfloor \tau_M(f) \rfloor$$

for every $f \in \mathcal{F}_\Sigma$.

To prove this theorem, we show several lemmas with respect to properties of an extension of θ_S. Before the definition of the extension, we introduce the *following forests representation* (for short, FFR) for a forest f that is a list of sub-forests of f whose syntax is

$$L ::= [] \mid f' :: L$$

where f' is a sub-forest of f. We use FF_f to denote a set of FFR's for a forest f. The i-th element of $L \in FF_f$ is accessed by $L.i$ where $(f' :: L).0 = f'$ and $(f' :: L).i = L.(i-1)$. The FFR can output one by one the next XML event from the XML stream corresponding to f by updating the representation. The initial FFR for a forest f is a singleton list of f, i.e., $f :: []$. The updating function ud takes the current FFR and returns the next XML event and the next FFR as follows:

$$ud(\sigma\,\texttt{[}f_1\texttt{]}\,f_2 :: L) = (\texttt{<}\sigma\texttt{>}, f_1 :: f_2 :: L) \qquad ud(\texttt{()} :: f :: L) = (\texttt{</}\Box\texttt{>}, f :: L) \qquad ud(\texttt{()} :: []) = (\texttt{EOF}, []).$$

It outputs one by one the next XML event by updating the FFR by ud, which is confirmed by the following lemma. We do not define $ud([])$ because the computation is not required in the rest of the paper.

Lemma 3.10 *Let g be the function defined over FFR's by*

$$g(l) = \begin{cases} \varepsilon & \text{if } l = [] \\ \chi\, g(l') & \text{otherwise} \end{cases}$$

where $(\chi, l') = ud(l)$. Then we have

$$g(f :: []) = \lfloor f \rfloor\, \texttt{EOF}. \tag{6}$$

Proof. First we show the following equation

$$g(f :: f' :: L) = \lfloor f \rfloor\, \texttt{</}\Box\texttt{>} g(f' :: L) \tag{7}$$

for forests f', f and a FFR l with some σ' by induction on the structure of f. If $f = \texttt{()}$, then (7) holds by the definitions of g, ud and $\lfloor \_\_ \rfloor$. If $f = \sigma\,\texttt{[}f_1\texttt{]}\,f_2$, then we have

$$\begin{aligned} g(\sigma\,\texttt{[}f_1\texttt{]}\,f_2 :: f' :: L) &= \texttt{<}\sigma\texttt{>}\, g(f_1 :: f_2 :: f' :: L) \\ &= \texttt{<}\sigma\texttt{>}\, \lfloor f_1 \rfloor\, \texttt{</}\Box\texttt{>} g(f_2 :: f' :: L) \\ &= \texttt{<}\sigma\texttt{>}\, \lfloor f_1 \rfloor\, \texttt{</}\Box\texttt{>}\, \lfloor f_2 \rfloor\, \texttt{</}\Box\texttt{>} g(f' :: L) \\ &= \lfloor \sigma\,\texttt{[}f_1\texttt{]}\,f_2 \rfloor\, \texttt{</}\Box\texttt{>} g(f' :: L) \end{aligned}$$

from the definition of g, ud and $\lfloor \_\_ \rfloor$ and the induction hypothesis. Hence (7) holds for every forest f.

Now we prove the equation (6) by induction on the structure of f. If $f = \texttt{()}$, then (6) holds by the definitions of g, ud and $\lfloor \_\_ \rfloor$. If $f = \sigma\,\texttt{[}f_1\texttt{]}\,f_2$, then we have

$$\begin{aligned} g(\sigma\,\texttt{[}f_1\texttt{]}\,f_2 :: []) &= \texttt{<}\sigma\texttt{>}\, g(f_1 :: f_2 :: []) \\ &= \texttt{<}\sigma\texttt{>}\, \lfloor f_1 \rfloor\, \texttt{</}\Box\texttt{>} g(f_2 :: []) \\ &= \texttt{<}\sigma\texttt{>}\, \lfloor f_1 \rfloor\, \texttt{</}\Box\texttt{>}\, \lfloor f_2 \rfloor\, \texttt{EOF} \\ &= \lfloor \sigma\,\texttt{[}f_1\texttt{]}\,f_2 \rfloor\, \texttt{EOF} \end{aligned}$$

from the definition g, ud and $\lfloor \_\_ \rfloor$, the equation (7) and the induction hypothesis. Therefore (6) holds for every forest f. ∎

Now we define an extension of θ_S with FFR. Let S be an xsp and f be a forest such that $\lfloor f \rfloor$ is an input stream for S. We define the function Θ_S as well as θ_S by

$$\Theta_S(E, []) = E \qquad\qquad \Theta_S(E, L) = \Theta(\langle\!|E, \chi|\!\rangle, L')$$

for $E \in Tmp_S$ and $L \in FF_f$ where $(\chi, L') = ud(L)$. The following lemma shows that the function Θ_S can simulate τ_S for an input stream $\lfloor f \rfloor$.

Lemma 3.11 *Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp. Then*

$$\tau_S(\lfloor f \rfloor) = \Theta_S(in(\varepsilon, \ldots, \varepsilon), f :: []) \tag{8}$$

for every $f \in \mathcal{F}_\Sigma$.

PROOF. From the definition of τ_S, what we have to show is

$$\theta_S(in(\varepsilon,\ldots,\varepsilon), \lfloor f \rfloor \,\mathtt{EOF}) = \Theta_S(in(\varepsilon,\ldots,\varepsilon), f :: []). \tag{9}$$

Let g be the function as given in LEMMA 3.10. and $G(L)$ be a set of FFR's occurring as an argument of g in the computation of $g(L)$, i.e., if $L = []$ then $G(L) = \{L\}$ and otherwise $G(L) = \{L\} \cup G(L')$ with $(\chi, L') = ud(L)$. The set $G(f :: [])$ is finite because $g(f :: [])$ always terminates as shown in the proof of LEMMA 3.10. We show the more general equation

$$\theta_S(E, g(L)) = \Theta_S(E, L) \tag{10}$$

for $E \in Tmp_S$ and $L \in G(f :: [])$. From LEMMA 3.10 we can claim that the equation (9) is the special case of (10) in which $E = in(\varepsilon,\ldots,\varepsilon)$ and $L = f :: []$. We prove the equation (10) by induction on the cardinality $\sharp G(L)$ of $G(L)$. If $\sharp G(L) = 1$, i.e., $L = []$, then the both sides are the same, that is e. If $\sharp G(L) > 1$, then $g(L) = \chi g(L')$ with $(\chi, L') = ud(L)$. We have $\sharp G(L') = \sharp G(L) - 1$ since $L \notin G(L')$ from the definition of G. Therefore we obtain

$$\begin{aligned}
\theta_S(E, g(L)) &= \theta_S(E, \chi\, g(L')) \\
&= \theta_S(\langle\!|E, \chi|\!\rangle, g(L')) \\
&= \Theta_S(\langle\!|E, \chi|\!\rangle, L') \\
&= \Theta_S(E, L)
\end{aligned}$$

from the induction hypothesis, the definition of Θ_S and $(\chi, L') = ud(L)$. Hence the equation (10) holds for $e \in Tmp_S$ and $L \in G(f :: [])$. ∎

Next we define the function $\mathcal{I}$ that translates temporary expressions into output forests. The function $\mathcal{I}$ has a good property that $\mathcal{I}(E, L) = \mathcal{I}(E', L')$ if $\Theta_S(E, L)$ is computed by $\Theta_S(E', L')$, which will be shown as LEMMA 3.12.

$$\begin{aligned}
&\mathcal{I}(q[i](E_1,\ldots,E_n), L) = [\![q]\!](L.i, \mathcal{I}(E_1, L),\ldots,\mathcal{I}(E_n, L)) && \mathcal{I}(\varepsilon, L) = \mathtt{()} \\
&\mathcal{I}(\mathtt{<}\delta\mathtt{>}E\mathtt{</}\square\mathtt{>}, L) = \delta\,\mathtt{[}\mathcal{I}(E, L)\mathtt{]} && \mathcal{I}(E\ E', L) = \mathcal{I}(E, L)\ \mathcal{I}(E', L)
\end{aligned}$$

Lemma 3.12 *Let $M = (Q, \Sigma, \Delta, in, R)$ be a mft, $\mathcal{SP}(M) = (Q, \Sigma, \Delta, in, R')$ be an xsp, $f \in \mathcal{F}_\Sigma$ be a forest and $L \in G(f :: [])$ be a FFR where G is a function as given in the proof of* LEMMA 3.11. *Then*

$$\mathcal{I}(E, L) = \mathcal{I}(\langle\!|E, \chi|\!\rangle, L') \tag{11}$$

where $(\chi, L') = ud(L)$.

PROOF. We prove the statements by induction on the structure of E. Here we show only the case of $E = q[0](E_1,\ldots,E_n)$ with $q \in Q$ that is the most complicated one in the induction. The other cases can be shown in a similar way, which are omitted.

If $E = q[0](E_1,\ldots,E_n)$ with $q \in Q$, then the left-hand side of (11) is equal to $[\![q]\!](L.0, \mathcal{I}(E_1),\ldots,\mathcal{I}(E_n))$ by the definition of $\mathcal{I}$. When $L = \sigma\,\mathtt{[}f_1\mathtt{]}\,f_2 :: L''$, $ud(L) = (\mathtt{<}\sigma\mathtt{>}, L')$ with $L' = f_1 :: f_2 :: L$. Then the left-hand side of (11) is

$$\begin{aligned}
[\![q]\!](L.0, \mathcal{I}(E_1, L),\ldots,\mathcal{I}(E_n, L)) &= [\![q]\!](\sigma\,\mathtt{[}f_1\mathtt{]}\,f_2, \mathcal{I}(E_1, L),\ldots,\mathcal{I}(E_n, L)) \\
&= [\![q]\!](\sigma\,\mathtt{[}f_1\mathtt{]}\,f_2, \mathcal{I}(\langle\!|E_1, \chi|\!\rangle, L'),\ldots,\mathcal{I}(\langle\!|E_n, \chi|\!\rangle, L')) \\
&= [\![rhs^{q,\sigma}]\!]_\rho
\end{aligned}$$

from the definition of $[\![\_\_]\!]$ and ud and the induction hypothesis for (11), where $\rho(x_i) = f_i$ for $i = 1, 2$ and $\rho(y_j) = \mathcal{I}(\langle\!|E_j, \chi|\!\rangle, L')$ for $j = 1,\ldots,n$. The right-hand side of (11) is

$$\mathcal{I}(\langle\!|q[0](E_1,\ldots,E_n), \mathtt{<}\sigma\mathtt{>}|\!\rangle, L') = \mathcal{I}([\![\mathcal{A}(rhs^{q,\sigma})]\!]_{\rho'}, L')$$

where $\rho'(y_j) = \langle\!|E_j, \mathtt{<}\sigma\mathtt{>}|\!\rangle$ for $j = 1,\ldots,n$. It is shown by induction on the structure of $rhs^{q,\sigma}$ that

$$[\![rhs^{q,\sigma}]\!]_\rho = \mathcal{I}([\![\mathcal{A}(rhs^{q,\sigma})]\!]_{\rho'}, L')$$

using $L'.0 = f_1$, $L'.1 = f_2$ and the definition of $\mathcal{I}$ and $\mathcal{A}$. When $L = \texttt{()} :: f :: L''$ and $L = \texttt{()} :: []$, we can show the equation (11) in a way similar to the case of $L = \sigma\,\texttt{[}\,f_1\,\texttt{]}\,f_2 :: L''$. ∎

Now we prove THEOREM 3.9. Let M be a mft and $S = \mathcal{SP}(M)$ be an xsp. The statement of LEMMA 3.12 shows that, for the computation of

$$\Theta_S(E_0, L_0) = \Theta_S(E_1, L_1) = \cdots = \Theta_S(E_n, L_n) = E_n \tag{12}$$

with $L_n = []$, all $\mathcal{I}(E_k, L_k)$ with $k = 0, \ldots, n$ are the same. Since $E_n = \langle\!|E_{n-1}, \texttt{EOF}|\!\rangle$ and the right-hand side of every rule in S with respect to `EOF` contains no occurrence of $q(\ldots)$ with a state q, E_n is just an XML stream. Therefore $\lfloor \mathcal{I}(E_n, L_n) \rfloor = E_n$ holds by the definition of $\mathcal{I}$ and $\lfloor\_\_\rfloor$. From the relation of $\mathcal{I}$ and Θ shown in LEMMA 3.12 and the equation (12), we obtain

$$\lfloor \mathcal{I}(E_0, L_0) \rfloor = E_n = \Theta_S(E_0, L_0)$$

When $E_0 = in[0](\varepsilon, \ldots, \varepsilon)$ and $L_0 = f :: []$ with an input forest f,

$$\begin{aligned} \lfloor \tau_M(f) \rfloor &= \lfloor [\![in]\!](f, \texttt{()}, \ldots, \texttt{()}) \rfloor = \lfloor \mathcal{I}(E_0, L_0) \rfloor \\ \tau_S(\lfloor f \rfloor) &= \Theta_S(E_0, L_0). \end{aligned}$$

Therefore THEOREM 3.9 has been proved.

4 Discussion

We hove shown how to derive an xsp from an arbitrary mft. Thus whether existing languages can be implemented as a program in stream processing style is whether the language can be translated into an mft. In order to make the translation easy, we discuss the extension of mft. In the formalization of mft, we cannot use even primitive functions over booleans, integers, strings, etc. In this section, we discuss how to extend our framework for such additional features and a few idea of translation for existing languages. Additionally, we add limitations of XML stream processors and show a benchmark result comparing with the existing processor.

4.1 Booleans and Conditional Branches

We consider a simple extension of mft with booleans and their operator, conditional branches. Let us extend the right-hand expression of mft with them as follows.

$$rhs ::= \ldots \mid true \mid false \mid if(rhs, rhs, rhs)$$

where *true* and *false* are boolean values and $if(e_1, e_2, e_3)$ stands for a conditional branch with a test e_1, a true-branch e_2 and a false-branch e_3.

If we regard *true*, *false* and *if* as output symbols of the mft, our algorithm derives an xsp from the mft though these symbols are left in right-hand side of rules in the obtained xsp. Hence we add the following special rules for them in a similar way to [15]:

$$if(true, e_1, e_2) \to e_1 \qquad\qquad if(false, e_1, e_2) \to e_2$$

By applying these rules in each squeezing phase, we achieve an XML stream processing for the extended mft.

4.2 Pattern-based Languages

Most of existing XML transformation languages support pattern-based recursion (iteration). For instance, XSLT [22] and XQuery [20] are based on pattern matching by XPath expressions. It is easy to encode simple forward XPath expressions in mft style. Predicates in an XPath expression can be encoded into mft-style programs using boolean values in the extended mft.

On the other hand, XDuce [8] and CDuce [2] are base on pattern matching by regular expressions. Though a regular pattern may contain the symbol `*` for Kleene-closure, both languages use the definition

```
type T = () | E T
```

for a regular expression type `E*`. Hence a program in these languages are written in recursive style which is quite similar to a mft-style program.

4.3 Limitation

There is a class of *inherently memory inefficient* transformations [16] such as M_{mir} in EXAMPLE 2.5, which reverses the order of markups at every nesting level for all descendants of `rev` nodes. Suppose the root node of an input XML is labeled with `rev`. Though our framework can deal with such a transformation, the obtained XML stream processor is not efficient because it cannot output any result until reading the end of the input stream. This problem is not specific to our framework. Every SAX-like stream processing program has the same problem: this kind of transformation is not suitable for stream processing.

5 Conclusion

We have presented a method to automatically derive an XML stream processor from a program in functional XML processing style, where we write XML transformations as recursive functions over the input XML tree. We adopt macro forest transducers (mft) as a model of functional XML processing and have shown that we can obtain an XML stream processor for every mft by our method. The framework presented in this paper will be applied to the next release of XTiSP [13]. The extension of our method will be applied to existing languages [8, 2, 22] in which programs are given a set of recursive functions over XML trees (forests).

Acknowledgment

The author is grateful to Giuseppe Castagna and Shin-Cheng Mu for their kind help and advice on the manuscript.

References

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. *International Journal on Very Large Data Bases*, pages 53–64, 2000.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th International Conference of Functional Programming*, pages 51–63, 2003.

[3] P. Cimprich, O. Becker, C. Nentwich, M. K. H. Jiroušek, P. Brown, M. Batsis, T. Kaiser, P. Hlavnička, N. Matsakis, C. Dolph, and N. Wiechmann. Streaming transformations for XML (STX) version 1.0. `http://stx.sourceforge.net/documents/`.

[4] Y. Diao and M. J. Franklin. High-performance XML filtering: An overview of YFilter. In *IEEE Data Engineering Bulletin*, volume 26(1), pages 41–48, 2003.

[5] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71–146, 1985.

[6] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, 2004.

[7] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2003.

[8] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[9] O. Kiselyov. A better XML parser through functional programming. In *4th International Symposium on Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 209–224, 2002.

[10] K. Kodama, K. Suenaga, N. Kobayashi, and A. Yonezawa. Translation of tree-processing programs into stream-processing programs based on ordered linear type. In *The 2nd ASIAN Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 41–56, 2004.

[11] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 227–238, 2002.

[12] M. Murata. Extended path expressions of XML. In *Proceedings of the 20th ACM Symp. on Principles of Database Systems*, pages 153–166, 2001.

[13] K. Nakano. XTiSP: XML transformation language intended for stream processing. `http://xtisp.psdlab.org/`.

[14] K. Nakano. Composing stack-attributed transducers. Technical Report METR-2004-01, Department of Mathematical Informatics, University of Tokyo, 2004.

[15] K. Nakano. An implementation scheme for XML transformation lauguages through derivation of stream processors. In *The 2nd ASIAN Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 74–90, 2004.

[16] S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*, 54:257–290, 2005.

[17] T. Perst and H. Seidl. Macro forest transducers. *Information Processing Letters*, 89:141–149, 2004.

[18] S. Scherzinger and A. Kemper. Syntax-directed transformations of XML streams. In *The workshop on Programming Language Technologies for XML*, pages 75–86, 2005.

[19] K. Suenaga, N. Kobayashi, and A. Yonezawa. Extension of type-based approach to generation of stream processing programs by automatic insertion of buffering primitives. In *International workshop on Logic-based Program Synthesis and Transformation*, 2005. To appear.

[20] XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`.

[21] SAX: the simple api for XML. `http://www.saxproject.org/`.

[22] XSL transformations (XSLT). `http://www.w3c.org/TR/xslt/`.