A Pushdown Machine for Recursive XML Processing*

Keisuke Nakano[1] and Shin-Cheng Mu[2]

[1] Department of Mathematical Informatics, University of Tokyo, Japan
ksk@mist.i.u-tokyo.ac.jp
[2] Institute of Information Science, Academia Sinica, Taiwan
scm@iis.sinica.edu.tw

Abstract. XML transformations are most naturally defined as recursive functions on trees. A naive implementation, however, would load the entire input XML tree into memory before processing. In contrast, programs in stream processing style minimise memory usage since it may release the memory occupied by the processed prefix of the input, but they are harder to write because the programmer is left with the burden to maintain a state. In this paper, we propose a model for XML stream processing and show that all programs written in a particular style of recursive functions on XML trees, the *macro forest transducer*, can be automatically translated to our stream processors. The stream processor is declarative in style, but can be implemented efficiently by a pushdown machine. We thus get the best of both worlds — program clarity, and efficiency in execution.

1 Introduction

Since an XML document has a tree-like structure, it is natural to define XML transformations as recursive functions over trees. Several XML-oriented languages, such as XSLT [34], *fxt* [3], XDuce [11] and CDuce [2], allow the programmer to define mutual recursive functions over forests. As an example, consider the program in Figure 1. Let $\sigma\langle f_1\rangle f_2$ denote a forest where the head is a σ-labeled tree whose children constitute the forest f_1, and the tail is a sibling forest f_2. The empty forest is denoted by ϵ and is usually omitted when enclosed in other trees. The function $Main$ in Figure 1 scans through the input tree and reverses the order of all subtrees under nodes labelled `r` by calling the function Rev. For example, the input tree $\mathtt{a}\langle\mathtt{r}\langle\mathtt{b}\langle\mathtt{c}\langle\rangle\mathtt{d}\langle\rangle\rangle\mathtt{e}\langle\rangle\rangle\mathtt{f}\langle\rangle\rangle$ is transformed into $\mathtt{a}\langle\mathtt{r}\langle\mathtt{e}\langle\rangle\mathtt{b}\langle\mathtt{d}\langle\rangle\mathtt{c}\langle\rangle\rangle\rangle\mathtt{f}\langle\rangle\rangle$.

A naive way to execute functions defined in this style is to load the entire forest into memory, so that we have convenient access to the children and siblings for each node. The input stream of tokens, also called *XML events*, is parsed to build the corresponding forest, which is then transformed by the function,

* Partially supported by *Comprehensive Development of e-Society Foundation Software* of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

$$
\begin{array}{ll}
Main(\epsilon) = \epsilon & Rev(\epsilon,\ y) = y \\
Main(\mathtt{r}\langle x_1\rangle x_2) = \mathtt{r}\langle rev(x_1,\epsilon)\rangle(Main(x_2)) & Rev(\sigma\langle x_1\rangle x_2,\ y) = Rev(x_2,\ \sigma\langle Rev\ x_1\ \epsilon\rangle\ y) \\
Main(\sigma\langle x_1\rangle x_2) = \sigma\langle main\ x_1\rangle(Main(x_2)) \quad \text{if } \sigma \neq \mathtt{r} &
\end{array}
$$

Fig. 1. A functional program reversing the subtrees under nodes labelled `r`.

before the resulting forest is unparsed to an XML stream. Loading the entire tree into memory is not preferable when we have to process large input. However, many XML transformation languages such as XSLT, *fxt*, XDuce and CDuce are actually implemented this way.

To optimise space usage, the programmer may switch to programming style (e.g SAX [32]). The stream processor reads XML events one by one, and the programmer defines respectively what to do when it encounters a start tag `<`σ`>`, an end tag `</`σ`>`, or end of stream `$`. Consider performing the same task given the input `<a><r><b><c></c><d></d></b><e> </e></r><f></f></a>`. Upon reading the first event `<a>`, we can output `<a>` immediately. The next event `<r>` is also copied to the output. After that, no output event will be produced for a while, because there is no way for the processor to know what to output before the closing tag `</r>` is read. Between `<r>` and `</r>`, the computer reads the input and stores a reversed stream in some environment[3]. While stream processing saves memory usage, it is much harder to program in this style.

Can we write a recursive function on forests and have it automatically transformed to a program in the stream processing style, thereby achieve both clarity and memory efficiency? In this paper, we present a model for an XML stream processor, and shows how to automatically derive XML stream processors from a very expressive class of recursive functions on forests.

We have made two main contributions. Firstly, we propose a model for XML stream processing which is declarative in nature but has an efficient implementation. The environment can be represented uniformly by a partially evaluated stream, called a *temporary expression*. Secondly, we present a method to derive a stream processor from any function definable in terms of the *macro forest transducer* (mft), proposed by Perst and Seidl [26]. The derivation, which can be seen as a special case of program fusion [30], works by fusing the mft with an XML parser recast as a *top-down tree transducer* (tdtt). The fusion is similar Engelfriet and Vogler's method of composing a (finitary) tdtt and a *macro tree transducer* [6]. but we have a proof that the method works for our tdtt with a infinite number of states.

This paper summaries our work. Interested readers are also referred to the full version [22] available online, which contains the proof of the main theorem and more discussions.

[3] An 'environment' is a state storing information needed to carry out the computation. We use the term 'environment' to avoid confusion with mft states.

2 XML and the Macro Forest Transducer

For simplicity, we deal with a simplified model of XML with only element nodes, and assume that the input XML is well-formed.

Let Σ be an alphabet. A Σ-*forest* (also called a Σ-hedge [18]), is defined by

$$f ::= \sigma\langle f\rangle f \;\mid\; \epsilon,$$

where $\sigma \in \Sigma$ and ϵ denotes the empty forest. We denote by $\mathcal{F}_\Sigma$ the set of Σ-forests. A Σ-forest $\mathtt{a}\langle\mathtt{b}\langle\epsilon\rangle\mathtt{c}\langle\epsilon\rangle\epsilon\rangle\epsilon$ with $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ represents the XML fragment `<a><b></b><c></c></a>`. The concatenation of two forests $f_1, f_2 \in \mathcal{F}_\Sigma$ is written $f_1 f_2$. The symbol ϵ, being the unit of concatenation, is often omitted.

The Σ-*events*, written $\Sigma_{\langle\rangle}$, is defined by $\Sigma_{\langle\rangle} = \{\texttt{<}\sigma\texttt{>} \mid \sigma \in \Sigma\} \cup \{\texttt{</}\sigma\texttt{>} \mid \sigma \in \Sigma\}$. An XML stream is a sequence of Σ-*events*. We denote by $\Sigma^\circ_{\langle\rangle}$ the set of well-formed sequences of Σ-events and denote by ε the empty sequence. The symbol `$` denotes the end of an (input) XML stream, which is also regarded as an event. We write $\Sigma_{\langle\rangle\$}$ for $\Sigma_{\langle\rangle} \cup \{\$\}$.

Let Σ be an alphabet. The *streaming* of a forest is the function $\lfloor\_\_\rfloor : \mathcal{F}_\Sigma \to \Sigma^\circ_{\langle\rangle}$ defined by $\lfloor\sigma\langle f_1\rangle f_2\rfloor = \texttt{<}\sigma\texttt{>}\,\lfloor f_1\rfloor\,\texttt{</}\sigma\texttt{>}\,\lfloor f_2\rfloor$ and $\lfloor\epsilon\rfloor = \varepsilon$. For example, $\lfloor\mathtt{a}\langle\mathtt{b}\langle\rangle\mathtt{c}\langle\rangle\rangle\rfloor =$ `<a><b></b><c></c></a>`.

The *macro forest transducer* (mft) was proposed by Perst and Seidl [26] as an extension to the *macro tree transducer* (mtt) [6] by taking concatenation as a basic operator. Functional programmers can think of an mft as a recursive function mapping a forest (and possibly some accumulating parameters) to a forest, with certain restriction on their shapes — the pattern on the forest extracts only the label, the children and the sibling of the first tree; the accumulating parameters cannot be pattern-matched; each function call is passed either the children or the sibling. We do not propose using the mft as a programming language, but as an intermediate language. It was shown that mft is in fact rather expressive [17]. In particular, XPath expressions can be converted to a computation model weaker than mfts [21]. More discussions will be given in Section 6.

In the convention of mft, a function is called a *state* and its arity is called its *rank*. Let us write $\mathbb{N}$ and $\mathbb{N}^+$ for the set of non-negative integers including and excluding 0, respectively.

Definition 1. A *macro forest transducer* is a tuple $M = (Q, \Sigma, \Delta, in, R)$, where

- Q is a finite set of ranked states, the rank of a state given by a function $rank : Q \to \mathbb{N}^+$,
- 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 partitioned by $R = \bigcup_{q \in Q} R_q$. For each $q \in Q$, R_q consists of rules of the form $q(pat, y_1, \ldots, y_n) \to rhs$, where $n = rank(q) - 1$ and
  - *pat* is either ϵ or $\sigma\langle x_1\rangle x_2$ for some $\sigma \in \Sigma$,
  - *rhs* ranges over expressions defined by

$$rhs ::= q'(x_i, rhs, \ldots, rhs) \mid \epsilon \mid \delta\langle rhs\rangle \mid y_j \mid rhs\ rhs$$

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

Perst and Seidl's mft, designed for type checking, can be non-deterministic. Since our focus is on program transformation, our mft's are deterministic and total. That is, for each q and σ there is exactly one such rule $q(\sigma\langle x_1\rangle x_2, \ldots) \to rhs$. We will denote its right-hand side by $rhs^{q,\sigma}$. Similarly $rhs^{q,\epsilon}$ stands for the right-hand side rhs of the unique rule $q(\epsilon, \ldots) \to rhs$. If a rule for state q and pattern p is missing, we assume that there is an implicit rule $q(p, \ldots) \to \epsilon$. The semantics of mft's is given by translating every state into a function [26]:

Definition 2. Let $M = (Q, \Sigma, \Delta, in, R)$ be an mft. The semantics of a states $q \in Q$ is given by the function $[\![q]\!] : \mathcal{F}_\Sigma \times (\mathcal{F}_\Delta)^n \to \mathcal{F}_\Delta$ where $n = rank(q) - 1$. Each $[\![q]\!]$ is defined by:

- $[\![q]\!](\sigma\langle\omega_1\rangle\omega_2, \varphi_1, \ldots, \varphi_n) = [\![rhs^{q,\sigma}]\!]_\rho$ where $\rho(x_i) = \omega_i$ for $i = 1, 2$ and $\rho(y_j) = \varphi_j$ for $j = 1, \ldots, n$,
- $[\![q]\!](\epsilon, \varphi_1, \ldots, \varphi_n) = [\![rhs^{q,\epsilon}]\!]_\rho$ where $\rho(y_j) = \varphi_j$ for $j = 1, \ldots, n$,

where $[\![\_\_]\!]_\rho$ evaluates the right-hand side with respect to the environment ρ:

$$[\![q'(x_i, rhs_1, \ldots, rhs_{n'})]\!]_\rho = [\![q']\!](\rho(x_i), [\![rhs_1]\!]_\rho, \ldots, [\![rhs_{n'}]\!]_\rho),$$
$$[\![\epsilon]\!]_\rho = \epsilon, \qquad [\![\delta\langle rhs\rangle]\!]_\rho = \delta\langle [\![rhs]\!]_\rho\rangle,$$
$$[\![y_j]\!]_\rho = \rho(y_j), \qquad [\![rhs\ rhs']\!]_\rho = [\![rhs]\!]_\rho [\![rhs']\!]_\rho.$$

Definition 3. *The* transformation *induced by an mft $M = (Q, \Sigma, \Delta, in, R)$ is the function $\tau_M : \mathcal{F}_\Sigma \to \mathcal{F}_\Delta$ defined by $\tau_M(f) = [\![in]\!](f, \epsilon, \ldots, \epsilon)$.*

Example 1. Let $Q = \{Main, Rev\}$, Σ some alphabet containing `r` and R the rules in Figure 1 (replacing = by $\to$), then $M_{rev} = (Q, \Sigma, \Sigma, Main, R)$ is an mft.

Example 2. The mft $M_{htm} = (Q, \Sigma, \Delta, Main, R)$ defined below reads an XML document consisting of a title and several paragraphs with some keywords. The output is a (simplified) HTML document where the `para` tag is converted to `p` and `key` tag to `em`. Furthermore, before the `ps` tag we dump the list of keywords we collect so far. Text data is denoted by a node with no children.

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

$\Sigma = \Delta =$ (some alphabet containing English words and the XML/HTML tags below),

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

$Title(\texttt{title}\langle x_1\rangle x_2) \to \texttt{title}\langle Copy(x_1)\rangle$,

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

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

$InArticle(\texttt{ps}\langle x_1\rangle x_2, y_1) \to \texttt{h2}\langle\texttt{Index}\langle\rangle\rangle\ \texttt{ul}\langle y_1\rangle\ \texttt{h2}\langle\texttt{Postscript}\langle\rangle\rangle\ Copy(x_1)$,

$Key2Em(\texttt{key}\langle x_1\rangle x_2) \to \texttt{em}\langle Copy(x_1)\rangle\ Key2Em(x_2)$,

$Key2Em(\sigma\langle x_1\rangle x_2) \to \sigma\langle Key2Em(x_1)\rangle\ Key2Em(x_2)\ \ (\sigma \neq \texttt{key}), Key2Em(\epsilon) \to \epsilon$,

$AllKeys(\texttt{key}\langle x_1\rangle x_2) \to \texttt{li}\langle Copy(x_1)\rangle AllKeys(x_2)$,

$AllKeys(\sigma\langle x_1\rangle x_2) \to AllKeys(x_1) AllKeys(x_2)\ \ (\sigma \neq \texttt{key}), \qquad AllKeys(\epsilon) \to \epsilon$,

$Copy(\sigma\langle x_1\rangle x_2) \to \sigma\langle Copy(x_1)\rangle Copy(x_2)\ \ (\sigma \in \Sigma), \qquad Copy(\epsilon) \to \epsilon\ \}$

3 XML Stream Processors and its Derivation

A *temporary expression* is a partially computed stream of XML events. An XML stream processor (xsp) defines how to rewrite a temporary expression upon reading each input event.

Definition 4. An *XML stream processor* is a tuple $S = (Q, \Sigma, \Delta, in, R)$, where

- Q is a (possibly infinite) set of ranked states, the rank for each state given 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,
- $R = \{q(y_1, \ldots, y_n) \xrightarrow{\chi} rhs \mid q \in Q, \chi \in \Sigma_{\texttt{<>\$}}\}$ is a set of rules, where $n = rank(q)$ and rhs ranges over expressions defined by

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

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

3.1 Semantics of XML Stream Processors

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

Definition 5. *Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp.* A *temporary expression* for S, denoted by Tmp_S, is defined by $E ::= \varepsilon \mid \texttt{<}\delta\texttt{>}E \mid \texttt{</}\delta\texttt{>}E \mid q(E, \ldots, E)E$.

Definition 6. Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp and $s \in \Sigma^\circ_{\texttt{<>}}$. The *transition* over Tmp_S for an input Σ-event is a function $\langle\!|\_\_, \_\_|\!\rangle : Tmp_S \times \Sigma_{\texttt{<>\$}} \to Tmp_S$ defined by

- $\langle\!|\varepsilon, \chi|\!\rangle = \varepsilon$,
- $\langle\!|\texttt{<}\delta\texttt{>}e, \chi|\!\rangle = \texttt{<}\delta\texttt{>}\langle\!|e, \chi|\!\rangle$ where $\delta \in \Delta$,
- $\langle\!|\texttt{</}\delta\texttt{>}e, \chi|\!\rangle = \texttt{</}\delta\texttt{>}\langle\!|e, \chi|\!\rangle$ where $\delta \in \Delta$,
- $\langle\!|q(e_1, \ldots, e_n)e, \chi|\!\rangle = (rhs[y_j := \langle\!|e_j, \chi|\!\rangle]_{j=1,\ldots,n})\langle\!|e, \chi|\!\rangle$ where $(q(y_1, \ldots, y_n) \xrightarrow{\chi} rhs) \in R$ with $q \in Q$ and $\chi \in \Sigma_{\texttt{<>\$}}$.

The initial temporary expression is $in(\varepsilon, \ldots, \varepsilon)$. An xsp reads the input stream of events and updates the temporary expression with the transition $\langle\!|\_\_, \_\_|\!\rangle$. The end of stream is marked by \$. Let $\chi_1\chi_2 \ldots \chi_k$ be the input stream with each $\chi_j \in \Sigma_{\texttt{<>}}$. The final expression is $\langle\!|\langle\!| \ldots \langle\!|\langle\!|in(\varepsilon, \ldots, \varepsilon), \chi_1|\!\rangle, \chi_2|\!\rangle, \ldots, \chi_k|\!\rangle, \texttt{\$}|\!\rangle$. Note that the final temporary expression is always in $\Delta^\circ_{\texttt{<>}}$ since, by Definition 4, the right-hand side of a $(q, \texttt{\$})$-rule does not contain any unevaluated state $q'(\ldots)$.

Definition 7. The transformation induced by an xsp $S = (Q, \Sigma, \Delta, in, R)$ is the function $\tau_S : \Sigma^\circ_{\texttt{<>}} \to \Delta^\circ_{\texttt{<>}}$ defined by $\tau_S(s) = \theta_S(in(\varepsilon, \ldots, \varepsilon), s\texttt{\$})$ where, for $e \in Tmp_S$,

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

The induced transformation defines declaratively what the output stream is, given the input stream. The very reason we program in the stream processing style, however, is to be able to print out a prefix of the output stream while reading the input. That is, we would like to 'squeeze' some part of the result from after each event read. This will be described in Section 4.3.

3.2 Deriving Stream Processors From Macro Forest Transducers

Given an mft $M = (Q, \Sigma, \Delta, in, R)$, a stream x, and a function $Parse :: \Sigma^{\circ}_{<>} \to \mathcal{F}_{\Sigma}$ parsing a stream of events into a forest, the expression $\lfloor [\![in]\!](Parse(x), \epsilon, \ldots, \epsilon) \rfloor$ yields a $\Delta^{\circ}_{<>}$ stream. If we can fuse the three functions, $\lfloor \_\_ \rfloor$, $[\![in]\!]$, and $Parse$ into one, we may have a stream processor. Fusing $\lfloor \_\_ \rfloor$ and $[\![in]\!]$ is a relatively easy task. The interesting step is fusing them with the parser. An XML parser can be written as a *top-down tree transducer* (tdtt) with an (countably-)infinite number of states

$$Parse[1]\ (\texttt{<}\sigma\texttt{>}s) = \sigma\langle Parse[1]\ s\rangle(Parse[2]\ s),\ \ Parse[1]\ (\texttt{</}\sigma\texttt{>}s) = \epsilon,$$
$$Parse[i]\ (\texttt{<}\sigma\texttt{>}s) = Parse[i+1]\ s \quad (i > 1),\ \ Parse[i]\ (\texttt{</}\sigma\texttt{>}s) = Parse[i-1]\ s \quad (i > 1),$$
$$Parse[i]\ (\$) = \epsilon.$$

for every $\sigma \in \Sigma$. Note that we do not need a forest transducer for parsing. The forest is constructed without using forest concatenation. Therefore, although it returns a forest, $Parse$ is still technically a tree transducer where the forest is represented by a binary tree. Multiple traversals of s is in fact avoided in the implementation, to be discussed in Section 4. We will also talk about a more typical way to specify the parser, and its effects, in Section 6.

Some previous work [24, 21, 25] talked about fusing a tree transducer for parsing with a transformation, but not one as expressive as an mft. More details are given in Section 7. Engelfriet and Vogler [6] described how to fuse a *finitary* tdtt and a *macro tree transducer* (mtt). Their method, however, does not apply directly to our application because $Parse$ has a infinite number of states. Our derivation from an mft to an xsp, to be presented in this section and proved in the full paper [22], is basically Engelfriet and Vogler's transducer fusion extended to mft's and specialised to one particular infinitary tdtt, $Parse$. The readers are not required to have knowledge of their method.

For a rationale behind the derivation, consider mft $M = (Q, \Sigma, \Delta, in, R)$. For every state $q \in Q$, we introduce in the derived xsp a set of states $\{q[i] \mid i \in \mathbb{N}^+\}$. Imagine that we are building forests as we read the input stream of events. With each start tag, the forest construction descends by one level. The state $q[1]$ performs the task that the state q in the mft is supposed to do. The number 1 indicates that the current forest will be its input. The states $q[i]$ for $i > 1$, on the other hand, represent 'suspended' states which will take effect $i - 1$ levels *above* the forest currently being built. The number i denotes the number of end tags expected. When an end tag is read, the number decrease by one, until the number reaches 1 and the state gets activated. When a start tag is read, the number shall increase by one because there is one more start tag to be matched.

Definition 8. Let $M = (Q, \Sigma, \Delta, in, R)$ be an 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 $rank(q[i]) = rank(q) - 1$,
- $in' = in[1] \in Q'$,
- R' contains rules introduced by the following three cases:

  xsp-(1). for all $q \in Q$ and $\sigma \in \Sigma$, we introduce

  $$q[1](y_1, \ldots, y_n) \xrightarrow{\texttt{<}\sigma\texttt{>}} \mathcal{A}(rhs^{q,\sigma}),$$

  xsp-(2). for all $q \in Q$, $\sigma \in \Sigma$ and $i \in \mathbb{N}^+$, we introduce:

  $$q[1](y_1, \ldots, y_n) \xrightarrow{\texttt{</}\sigma\texttt{>}} \mathcal{A}(rhs^{q,\epsilon}),$$
  $$q[i](y_1, \ldots, y_n) \xrightarrow{\texttt{\$}} \mathcal{A}(rhs^{q,\epsilon}),$$

  xsp-(3). for all $q \in Q$, $\sigma \in \Sigma$ and $i > 1$, we introduce:

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

The translation $\mathcal{A}$ is defined by:

$$\mathcal{A}(q(x_i, rhs_1, \ldots, rhs_n)) = q[i](\mathcal{A}(rhs_1), \ldots, \mathcal{A}(rhs_{n'})),$$
$$\mathcal{A}(\epsilon) = \varepsilon, \qquad \mathcal{A}(\delta\langle rhs\rangle) = \texttt{<}\delta\texttt{>}\mathcal{A}(rhs)\texttt{</}\delta\texttt{>},$$
$$\mathcal{A}(y_j) = y_j, \qquad \mathcal{A}(rhs\ rhs') = \mathcal{A}(rhs)\ \mathcal{A}(rhs'),$$

where $q \in Q$, $n = rank(q)$, $\delta \in \Delta$, $i \in \{1, 2\}$ and $j \in \{1, \ldots, n\}$.

Note that among the three cases of rule introduction, **xsp-(1)** covers the situation when the state and the input symbols are $(q[1], \texttt{<}\sigma\texttt{>})$; **xsp-(2)** covers $(q[1], \texttt{</}\sigma\texttt{>})$ and $(q[i], \texttt{\$})$ for $i \in \mathbb{N}^+$; and **xsp-(3)** covers $(q[i], \texttt{<}\sigma\texttt{>})$ and $(q[i], \texttt{</}\sigma\texttt{>})$ for $i > 1$. Therefore, the derived xsp $\mathcal{SP}(M)$ is total if M is. For the examples below, we define a predicate testing whether the state and the input symbols is in the **xsp-(2)** case: $\epsilon_\Sigma(i, \chi) = (i = 1 \wedge \chi = \texttt{</}\sigma\texttt{>}) \vee \chi = \texttt{\$}$, with $\sigma \in \Sigma$. The following theorem, stating the correctness of the derivation, is proved in the full version of this paper [22].

Theorem 1. Let $M = (Q, \Sigma, \Delta, in, R)$ be an mft. Then $\tau_{\mathcal{SP}(M)}(\lfloor f \rfloor) = \lfloor \tau_M(f) \rfloor$ for every $f \in \mathcal{F}_\Sigma$.

Example 3. Apply the derivation to Example 1, we get $\mathcal{SP}(M_{rev}) = (Q', \Sigma, \Sigma, Main[1], R')$, where $Q' = \{q[i] \mid q \in \{Main, Rev\}, i \in \mathbb{N}^+\}$ and the set R' is:

$\{Main[1]() \xrightarrow{\texttt{<r>}} \texttt{<r>}\ Rev[1](\varepsilon)\ \texttt{</r>}\ Main[2](),$

$Main[1]() \xrightarrow{\texttt{<}\sigma\texttt{>}} \texttt{<}\sigma\texttt{>}\ Main[1]()\ \texttt{</}\sigma\texttt{>}\ Main[2]()$ $(\sigma \neq \texttt{r}),$ $\quad Rev[1](y_1) \xrightarrow{\texttt{<}\sigma\texttt{>}} Rev[2](\texttt{<}\sigma\texttt{>}\ Rev[1](\varepsilon)\ \texttt{</}\sigma\texttt{>}\ y_1)$ $(\sigma \in \Sigma),$

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

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

$Main[i]() \xrightarrow{\chi} \varepsilon$ $(\text{if } \epsilon_\Sigma(i, \chi)),$ $\quad Rev[i](y_1) \xrightarrow{\chi} y_1$ $(\text{if } \epsilon_\Sigma(i, \chi))\ \}.$

Figure 2(a) shows a sample run when the input is <a><r><c></c><d></d><e></e></r><f></f>$.

Example 4. The xsp derived from EXAMPLE 2 is $\mathcal{SP}(M_{htm}) = (Q', \Sigma, \Delta, Main[1], R')$, where $Q' = \{q[i] \mid q \in Q, i \in \mathbb{N}^+\}$ and R' is shown in Figure 2(b).

4 Pushdown XML Stream Processor

The semantics given in Section 3 implies a direct implementation of xsp performing term rewriting each time an event is read. However, an xsp derived from an mft follows a more regular evaluation pattern which resembles a stack. In this section, we present an efficient implementation of the xsp's derived from mft's.

4.1 Summary of Behavior

Let us look at an example first. Consider the sample run of the xsp $\mathcal{SP}(M_{rev})$ in Figure 2, when event <c> is read. We abbreviate *Rev* to r and *Main* to m. The prefix <a><r> has been 'squeezed' to the output. We need only to keep a suffix of the temporary expression in memory:

$$e_{before} = r[2](\texttt{}r[1](\varepsilon)\texttt{})\texttt{</r>}m[3]()\texttt{}m[4]().$$

After <c> is read, the expression gets updated to

$$e_{after} = r[3](\texttt{}r[2](\texttt{<c>}r[1](\varepsilon)\texttt{</c>})\texttt{})\texttt{</r>}m[4]()\texttt{}m[5]().$$

We shall present a data structure such that the update can be efficient.

We represent a temporary expression by a pair of a *main output stream* and a *pushdown*, as shown in Figure 3. The left and right parts in the figure correspond to temporary expressions e_{before} and e_{after}, respectively. Consider e_{before}. Separating the evaluated and unevaluated segments, it can be partitioned into five parts: $r[2](\dots)$, </r>, $m[3]()$, and $m[4]()$. If we abstract away the unevaluated parts and replace them with *holes* $[\,]_{\nu_i}$ using a physical address ν_i, we obtain the main output stream $[\,]_{\nu_1}\texttt{</r>}[\,]_{\nu_2}\texttt{}[\,]_{\nu_3}$.

The pushdown is a stack of sets, each set consisting of *state frames*. A state frame is a pair of a state $q(\dots)$ and a hole address ν, denoted by $q(\dots)/\nu$. The state may have a number of arguments, represented by a sequence in a way similar to the main output stream. In the pushdown representation, every state $q[i]$ appears in the i-th set from the top. Therefore the index i need not be stored in the representation. Since all states in e_{before} have distinct indexes, the pushdown contains only singleton sets, which need not be true in general.

Only the states with index 1 gets expanded. In our representation, that means we only need to update the top of the pushdown. Upon reading <c>, the rule of $r[1]$ that gets triggered is $r[1](\varepsilon) \xrightarrow{\texttt{<c>}} r[2](\texttt{<}\sigma\texttt{>}\ r[1](\varepsilon)\ \texttt{</}\sigma\texttt{>}\ \varepsilon)$. That corresponds to popping the set $\{r(\varepsilon)/\nu_4\}$ (representing $r[1](\varepsilon)$ in e_{before}), and pushing

$$
\begin{aligned}
Main[1]() &\xrightarrow{\texttt{<a>}} \texttt{<a>}Main[1]()\texttt{}Main[2]() \\
&\xrightarrow{\texttt{<r>}} \texttt{<a><r>}Rev[1](\varepsilon)\texttt{</r>}Main[2]()\texttt{}Main[3]() \\
&\xrightarrow{\texttt{}} \texttt{<a><r>}Rev[2](\texttt{}Rev[1](\varepsilon)\texttt{})\texttt{</r>}Main[3]()\texttt{}Main[4]() \\
&\xrightarrow{\texttt{<c>}} \texttt{<a><r>}Rev[3](\texttt{}Rev[2](\texttt{<c>}Rev[1](\varepsilon)\texttt{</c>})\texttt{})\texttt{</r>}Main[4]()\texttt{}Main[5]() \\
&\xrightarrow{\texttt{</c>}} \texttt{<a><r>}Rev[2](\texttt{}Rev[1](\texttt{<c></c>})\texttt{})\texttt{</r>}Main[3]()\texttt{}Main[4]() \\
&\xrightarrow{\texttt{<d>}} \texttt{<a><r>}Rev[3](\texttt{}Rev[2](\texttt{<d>}Rev[1](\varepsilon)\texttt{</d><c></c>})\texttt{})\texttt{</r>}Main[4]()\texttt{}Main[5]() \\
&\xrightarrow{\texttt{</d>}} \texttt{<a><r>}Rev[2](\texttt{}Rev[1](\texttt{<d></d><c></c>})\texttt{})\texttt{</r>}Main[3]()\texttt{}Main[4]() \\
&\xrightarrow{\texttt{}} \texttt{<a><r>}Rev[1](\texttt{<d></d><c></c>})\texttt{</r>}Main[2]()\texttt{}Main[3]() \\
&\xrightarrow{\texttt{<e>}} \texttt{<a><r>}Rev[2](\texttt{<e>}Rev[1](\varepsilon)\texttt{</e><d></d><c></c>})\texttt{</r>}Main[3]()\texttt{}Main[4]() \\
&\xrightarrow{\texttt{</e>}} \texttt{<a><r>}Rev[1](\texttt{<e></e><d></d><c></c>})\texttt{</r>}Main[2]()\texttt{}Main[3]() \\
&\xrightarrow{\texttt{</r>}} \texttt{<a><r><e></e><d></d><c></c></r>}Main[1]()\texttt{}Main[2]() \\
&\xrightarrow{\texttt{<f>}} \texttt{<a><r><e></e><d></d><c></c></r><f>}Main[1]()\texttt{</f>}Main[2]()\texttt{}Main[3]() \\
&\xrightarrow{\texttt{</f>}} \texttt{<a><r><e></e><d></d><c></c></r><f></f>}Main[1]()\texttt{}Main[2]() \\
&\xrightarrow{\texttt{}} \texttt{<a><r><e></e><d></d><c></c></r><f></f>}Main[1]() \\
&\xrightarrow{\texttt{\$}} \texttt{<a><r><e></e><d></d><c></c></r><f></f>}
\end{aligned}
$$

(a)

$$
\begin{aligned}
R' = \{\ & Main[1]() \xrightarrow{\texttt{<article>}} \texttt{<html> <head> }Title[1]()\texttt{ </head> <body> }InArticle[1](\varepsilon)\texttt{ </body> </html>}, \\
& Title[1]() \xrightarrow{\texttt{<title>}} \texttt{<title> }Copy[1]()\texttt{ </title>}, \\
& InArticle[1](y_1) \xrightarrow{\texttt{<title>}} \texttt{<h1> }Copy[1]()\texttt{ </h1> }InArticle[2](y_1), \\
& InArticle[1](y_1) \xrightarrow{\texttt{<para>}} \texttt{<p> }Key2Em[1]()\texttt{ </p> }InArticle[2](y_1\ AllKeys[1]()), \\
& InArticle[1](y_1) \xrightarrow{\texttt{<ps>}} \texttt{<h2>Index</h2>
}y_1\texttt{
<h2>Postscript</h2>}Copy[1](), \\
& Key2Em[1]() \xrightarrow{\texttt{<key>}} \texttt{ }Copy[1]()\texttt{ }Key2Em[2](), \\
& Key2Em[1]() \xrightarrow{\texttt{<}\sigma\texttt{>}} \texttt{<}\sigma\texttt{> }Key2Em[1]()\texttt{ </}\sigma\texttt{> }Key2Em[2]() \quad (\sigma \neq \texttt{key}), \\
& AllKeys[1]() \xrightarrow{\texttt{<key>}} \texttt{ }Copy[1]()\texttt{ }AllKeys[2](), \\
& AllKeys[1]() \xrightarrow{\texttt{<}\sigma\texttt{>}} AllKeys[1]()AllKeys[2]() \quad (\sigma \neq \texttt{key}), \\
& Copy[1]() \xrightarrow{\texttt{<}\sigma\texttt{>}} \texttt{<}\sigma\texttt{> }Copy[1]()\texttt{ </}\sigma\texttt{> }Copy[2]() \quad (\sigma \in \Sigma), \\
& q[i]() \xrightarrow{\texttt{<}\sigma\texttt{>}} q[i+1]() \quad (\sigma \in \Sigma,\ i > 1,\ q \neq InArticle), \\
& q[i]() \xrightarrow{\texttt{</}\sigma\texttt{>}} q[i-1]() \quad (\sigma \in \Sigma,\ i > 1,\ q \neq InArticle), \\
& q[i]() \xrightarrow{\chi} \varepsilon \quad (\text{if } \epsilon_\Sigma(i,\chi)), \\
& InArticle[i](y_1) \xrightarrow{\texttt{<}\sigma\texttt{>}} InArticle[i+1](y_1) \quad (\sigma \in \Sigma,\ i > 1), \\
& InArticle[i](y_1) \xrightarrow{\texttt{</}\sigma\texttt{>}} InArticle[i-1](y_1) \quad (\sigma \in \Sigma,\ i > 1), \\
& InArticle[i](y_1) \xrightarrow{\chi} \varepsilon \quad ((\chi, i) \in \Sigma_\epsilon)\ \}.
\end{aligned}
$$

(b)

Fig. 2. Stream processing induced by $\mathcal{SP}(M_{rev})$ and $\mathcal{SP}(M_{htm})$.

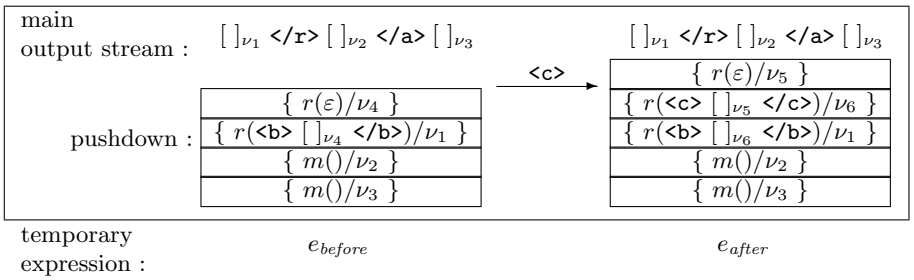

Fig. 3. Pushdown representation for temporary expressions and its updating

two sets $\{r(\varepsilon)/\nu_5\}$ (representing $r[1](\varepsilon)$ in e_{after}) and $\{r(\texttt{<c>}\ [\ ]_{\nu_5}\ \ \texttt{</c>})/\nu_6\}$ (representing $r[2](\texttt{<}\sigma\texttt{>}\ldots\texttt{</}\sigma\texttt{>})$ in e_{after}). Now that ν_4 is expanded, all occurrences of $[\ ]_{\nu_4}$ in the pushdown should be filled with $[\ ]_{\nu_6}$. Since two items are pushed, all other sets in the pushdown descend for one level. This corresponds to updating all states $q[i]$ $(i > 1)$ to $q[i+1]$ at the same time.

4.2 Pushdown Representation and its Updating

Let $M = (Q, \Sigma, \Delta, in, R)$ be an mft. An *output stream* s for M is defined by

$$m ::= \varepsilon \ \mid\ \texttt{<}\delta\texttt{>}\ m \ \mid\ \texttt{</}\delta\texttt{>}\ m \ \mid\ [\ ]_\nu\ m,$$

where $\delta \in \Delta$, and $[\ ]_\nu$ is a hole whose physical address is ν. We denote the set of output streams by $\mathcal{S}_M$. A *state frame* has the form $q(m_1, \ldots, m_n)/\nu$ where ν is a hole address, $q \in Q$, $n = rank(q)$, and $m_i \in \mathcal{S}_M$ $(i = 1, \ldots, n)$.

A *pushdown* is a mapping from a positive number, representing the depth, to a set of state frames. Furthermore, each hole address ν occurs on the right-hand side of / in a pushdown at most once. The empty pushdown is denoted by $\emptyset$. Given a set of state frames Ψ, we denote by $\{1 \mapsto \Psi, \ldots\}$ a pushdown p such that $p(1) = \Psi$. Two pushdowns p_1 and p_2 can be merged by $p_1 \oplus p_2 = \{d \mapsto p_1(d) \uplus p_2(d)\}_{d \in \mathbb{N}^+}$[4].

Definition 9. Let $M = (Q, \Sigma, \Delta, in, R)$ be an mft. A *pushdown representation* $pd(e)$ for $e \in Tmp_{\mathcal{SP}(M)}$ is a pair $\langle m, p\rangle$ of a *main output stream* m and a *pushdown* p defined by

$$pd(\varepsilon) = \langle \varepsilon, \emptyset\rangle, \qquad pd(\texttt{<}\delta\texttt{>}\ e) = \langle \texttt{<}\delta\texttt{>}\ m, p\rangle, \qquad pd(\texttt{</}\delta\texttt{>}\ e) = \langle \texttt{</}\delta\texttt{>}\ m, p\rangle,$$
$$pd(q[i](e_1, \ldots, e_n)\ e) = \langle [\ ]_\nu\ m, \{i \mapsto \{q(m_1, \ldots, m_n)/\nu\}\} \oplus p_1 \oplus \cdots \oplus p_n \oplus p\rangle,$$

where $\langle m, p\rangle = pd(e)$, $\langle m_i, p_i\rangle = pd(e_i)$ and ν is a fresh address. Denote the set of pushdown representations for temporary expressions in $Tmp_{\mathcal{SP}(M)}$ by Pdr_M.

From a pushdown representation, we can recover the temporary expression by filling every hole according to the corresponding state frame in the pushdown.

[4] For any $d \in \mathbb{N}^+$, $p_1(d)$ and $p_2(d)$ are disjoint because hole addresses are unique.

We define several operations to manipulate the pushdown representation. An *application* for a hole $[\,]_\nu$ in an output stream m with another output stream u is denoted by $m@_\nu u$, i.e., when $m = m_1[\,]_\nu m_2$, we have $m@_\nu u = m_1 u m_2$. The hole application can be extended to a set of state frames and a pushdown in the same way, denoted by $\Psi@_\nu u$ and $p@_\nu u$. Let p be a pushdown and Ψ a set of state frames. The pushdown obtained by pushing Ψ on the top of p is denoted by $p \ll \Psi = \{1 \mapsto \Psi\} \cup \{d \mapsto p(d-1)\}_{d>1}$. The dual operation popping the top of p is denoted by $\triangleright p = \{d \mapsto p(d+1)\}_{d \in \mathbb{N}^+}$.

The hole application operation can be efficiently implemented in the sense that the execution time is independent of the size of main output streams and pushdowns. Experimental implementation introduced in Section 5 uses doubly-linked cyclic lists to represent output streams, so we can implement hole application, concatenation and squeeze efficiently.

4.3 Pushdown Machines for Macro Forest Transducers

For a given mft M, we introduce a pushdown machine in stream processing style which simulates the behavior of the xsp $\mathcal{SP}(M)$. Since the semantics of an xsp is specified by a transition $\langle\!|\_\_, \_\_|\!\rangle$ on temporary expressions, we construct the pushdown machine as a transition on pushdown representations. In the following definition, the function pd° extends pd by one extra case, $pd^\circ(y_i) = y_i$ for $i \in \mathbb{N}^+$. Therefore pd° can be applied to the right-hand side of rules in an xsp.

Definition 10. Let $M = (Q, \Sigma, \Delta, in, R)$ be an mft. The *pushdown machine for* M, denoted by $\mathcal{PD}(M)$, is a function $\langle\!|\_\_, \_\_|\!\rangle : Pdr_M \times \Sigma_{\texttt{<>\$}} \to Pdr_M$. For a pushdown representation $\langle m, p\rangle \in Pdr_M$ and input event $\chi \in \Sigma_{\texttt{<>\$}}$, a new pushdown representation $\langle\!|\langle m, p\rangle, \chi|\!\rangle$ is given as follows:

- $\langle\!|\langle m, p\rangle, \texttt{<}\sigma\texttt{>}|\!\rangle = \Phi_\sigma(m, (\triangleright p) \ll \emptyset \ll \emptyset, p(1))$, where function Φ_σ is defined by

$$\Phi_\sigma(m, p, \emptyset) = \langle m, p\rangle$$
$$\Phi_\sigma(m, p, \{q(m_1, \ldots, m_n)/\nu\} \uplus \Psi) = \Phi_\sigma(m@_\nu m', p@_\nu m' \oplus p', \Psi@_\nu m')$$

  with $\langle m', p'\rangle = pd^\circ(\mathcal{A}(rhs^{q,\sigma}))[y_j := m_j]_{j=1,\ldots,n}$.
- $\langle\!|\langle m, p\rangle, \texttt{</}\sigma\texttt{>}|\!\rangle = \Phi_\epsilon(m, \triangleright p, p(1))$, where the function Φ_ϵ is defined by

$$\Phi_\epsilon(m, p, \emptyset) = \langle m, p\rangle$$
$$\Phi_\epsilon(m, p, \{q(m_1, \ldots, m_n)/\nu\} \uplus \Psi) = \Phi_\epsilon(m@_\nu m', p@_\nu m' \oplus p', \Psi@_\nu m')$$

  with $\langle m', p'\rangle = pd^\circ(\mathcal{A}(rhs^{q,\epsilon}))[y_j := m_j]_{j=1,\ldots,n}$.
- $\langle\!|\langle m, p\rangle, \$|\!\rangle = \Phi_\epsilon(m, \emptyset, \bigcup_{d \in \mathbb{N}^+} p(d))$, where Φ_ϵ is as in the case $\chi = \texttt{</}\sigma\texttt{>}$.

For an mft $M = (Q, \Sigma, \Delta, in, R)$, the initial pushdown representation of $\mathcal{PD}(M)$ is $\langle[\,]_{\nu_0}, \{1 \mapsto \{in(\varepsilon, \ldots, \varepsilon)/\nu_0\}\}\rangle$ with address ν_0. It corresponds to the initial state of an xsp $\mathcal{SP}(M)$, that is, $in[1](\varepsilon, \ldots, \varepsilon)$. For a pushdown machine $P = \mathcal{PD}(M)$, the transformation $\tau_P : \Sigma^\circ_{\texttt{<>}} \to \Delta^\circ_{\texttt{<>}}$ induced by P a defined in a way similar to $\tau_{\mathcal{SP}(M)}$, that is, $\tau_P(s) = \zeta_P(\langle[\,]_{\nu_0}, \{1 \mapsto \{in(\varepsilon, \ldots, \varepsilon)/\nu_0\}\}\rangle, s\$)$, where $\zeta_P(\langle m, p\rangle, \varepsilon) = m$ and $\zeta_P(\langle m, p\rangle, \chi s) = \zeta_P(\langle\!|\langle m, p\rangle, \chi|\!\rangle, s)$ for a pushdown representation $\langle m, p\rangle$.

For an mft M, the behaviour of $\mathcal{PD}(M)$ on pushdown representations mirrors that of $\mathcal{SP}(M)$ on temporary expressions. Consider the case when a start tag <σ> is read. In the xsp $\mathcal{SP}(M)$, every state $q[i]$ $(i > 1)$ is rewritten into $q[i+1]$. In the pushdown machine $\mathcal{PD}(M)$, the corresponding state frame $q(\dots)/\nu$ in the i-th set of the pushdown descends by one level because we perform one pop and two pushes on the pushdown. In $\mathcal{SP}(M)$, every state $q[1]$ is rewritten by $\mathcal{A}(rhs^{q,\sigma})$. In the pushdown machine $\mathcal{PD}(M)$, for each corresponding state frame $q(\dots)/\nu$ in the top set of the pushdown, the hole $[\,]_\nu$ is filled according to $\mathcal{A}(rhs^{q,\sigma})$. Since a computation of $pd^\circ(\mathcal{A}(rhs^{q,\sigma}))$ is invoked, the state $q[1]$ in $\mathcal{A}(rhs^{q,\sigma})$ is put as an element of the top set of the pushdown and $q[2]$ in $\mathcal{A}(rhs^{q,\sigma})$ is put as an element of the second set from the top.

Consider the case when an end tag </σ> is read. In the xsp $\mathcal{SP}(M)$, every state $q[i]$ $(i > 1)$ is rewritten to $q[i-1]$. The corresponding state frame $q(\dots)/\nu$ in the i-th set of the pushdown ascends by one level after popping. In the xsp $\mathcal{SP}(M)$, every state $q[1]$ is replaced according to $\mathcal{A}(rhs^{q,\epsilon})$. In the pushdown machine $\mathcal{PD}(M)$, for the corresponding state frame $q(\dots)/\nu$ in the top set of the pushdown, the hole $[\,]_\nu$ is filled according to $\mathcal{A}(rhs^{q,\epsilon})$.

After reading \$, the pushdown must be empty since $\mathcal{A}(rhs^{q,\epsilon})$ contains no pattern $q[i](\dots)$ and all state frames in the previous pushdown is consumed by $\Phi_\epsilon(s, \emptyset, \bigcup_{d\in\mathbb{N}^+} p(d))$. Therefore the final output stream has no holes.

Since every transition on pushdown representations corresponds to a transition on temporary expressions, we can see that $\tau_{\mathcal{PD}(M)}(s) = \tau_{\mathcal{SP}(M)}(s)$ for every mft M and every input stream s. From THEOREM 1, we have $\tau_{\mathcal{PD}(M)}(\lfloor f \rfloor) = \lfloor \tau_M(f) \rfloor$ for every input forest f for M, which shows the equivalence of the original mft and the derived pushdown machine.

The above definition of τ_P for a pushdown machine P can be made more efficient by *squeezing*, that is, printing out the prefix, up to the first hole, of the main output stream. We define the following function sqz for output streams:

$$sqz(\varepsilon) = (\varepsilon, \varepsilon), \qquad sqz([\,]_\nu\ m) = (\varepsilon, [\,]_\nu\ m),$$
$$sqz(\text{<}\delta\text{>}\ m) = (\text{<}\delta\text{>}\ m', m''), \qquad sqz(\text{</}\delta\text{>}\ m) = (\text{</}\delta\text{>}\ m', m''),$$

where $(m', m'') = sqz(m)$. We can then redefine ζ_P with sqz as follows:

$$\zeta_P(\langle m, p\rangle, \varepsilon) = m,$$
$$\zeta_P(\langle m, p\rangle, \chi s) = m'\ \zeta_P(\langle\!|\langle m'', p\rangle, \chi|\!\rangle, s) \text{ where } (m', m'') = sqz(\langle\!|\langle m, p\rangle, \chi|\!\rangle).$$

Some variables do not occur in the right-hand side. For example, consider a rule $q(pat, y_1) \to \epsilon$ and the corresponding xsp rule $q[1](y_1) \xrightarrow{\chi} \varepsilon$. When the top set of the pushdown contains $q(m_1)/\nu$, the occurrence of $[\,]_\nu$ will be filled with ε. To avoid ineffective updating, all hole addresses contained in m_1 should be discarded if the hole does not occur in other positions. Some variables may occur more than once. For example, consider the rule $q(pat, y_1) \to y_1\ y_1$. and the corresponding xsp rule $q[1](y_1) \xrightarrow{\chi} y_1\ y_1$. If we use doubly-linked cyclic lists to represent main output streams, a hole $[\,]_\nu$ may occur twice. When the hole $[\,]_\nu$ is required to be filled, we cannot replace both occurrence of ν with the same doubly-linked list. Therefore, we mark the state frame to remember that it appears twice.

input size	1MB	4MB	16MB	64MB	256MB
pushdown xsp	0.49sec / 1.10MB	1.19sec / 1.10MB	3.85sec / 1.10MB	15.2sec / 1.10MB	84.6sec / 1.10MB
direct impl. mft	0.52sec / 4.87MB	1.39sec / 16.7MB	4.92sec / 62.1MB	20.2sec / 250MB	588sec / 415MB
xsltproc	0.79sec / 8.73MB	3.51sec / 33.2MB	19.4sec / 129MB	162sec / 462MB	n/a
saxon	3.12sec / 24.5MB	5.40sec / 36.5MB	13.1sec / 94.4MB	43.7sec / 289MB	n/a

(execution time / max. memory usage)

(a) For transformation M_{rev}

input size	4MB	64MB
pushdown xsp	1.26sec / 3.93MB	17.1sec / 49.8MB
direct impl. mft	1.25sec / 15.9MB	17.7sec / 233MB

(execution time / max. memory usage)

(b) For transformation M_{htm}

input size	4MB	64MB
pushdown xsp	1.60sec / 11.6MB	24.8sec / 170MB
direct impl. mft	1.40sec / 16.6MB	20.4sec / 249MB

(execution time / max. memory usage)

(c) For transformation M_{frev}

Table 1. Benchmarking results

5 Benchmarking Results

We use the random sample generator XMark [33] to produce sample XML documents of sizes 1MB, 4MB, 16MB, 64MB and 256MB. A document contains a sequence of `item` nodes, each having a list of children about a dozen lines long.

The first task is to reverse the order of subtrees under `item`. The pushdown machine automatically derived from the mft M_{rev}, shown as the entry **pushdown xsp** in Table 1, is implemented in Objective Caml, with extensions to handle text nodes. The entry **direct impl. mft** is the program in Figure 1 implemented as mutual recursive functions in Objective Caml. The entry **xsltproc** is one of the fastest XSLT processors bundled with `libxslt` [31] 1.1.11, written in C, while **saxon** [13] 8.7.3 is one of the fastest XSLT processors in Java. All entries apart from **pushdown xsp** build the entire forest in memory before the transformation. The experiments were conducted on a 1.33 GHz PowerBook G4 with 768 MB of memory. Table 1(a) compares the total execution time and maximum memory size in seconds and megabytes.

As we expected, **pushdown xsp** uses the smallest heap. That it also outperforms the two XSLT processors may be due to the overhead of the latter maintaining full-fledged XML data, including e.g., namespace URI, number of children. For a fairer comparison, we added the entry **direct impl. mft**. The entry **pushdown xsp** is slightly faster than **direct impl. mft** because it incurs less garbage collection, and saves the overhead of building the trees. We expect that xsp will also deliver competitive speed even after scaling to full XML.

For other transformations, we compared only **pushdown xsp** and **direct impl. mft** for random inputs of 4MB and 64MB. Table 1(b) shows the results for transformation M_{htm} in Example 2. This result also indicates a small heap residency of **pushdown xsp** with elapsed time similar to **direct impl. mft**. Table 1(c) shows results for full reversal M_{frev}, which will be discussed later.

6 Discussion

Comparison with Lazy Evaluation. Many of our readers wondered: "Can we not just use lazy evaluation?" Consider the program $unparse\ (trans\ (parse\ input))$ in a non-strict language, where the function $parse$ builds the tree lazily upon the demand of the forest-to-forest transformation $trans$. When the program is run by a lazy evaluator, do we get the desired space behaviour? We run a number of

experiments in Haskell. The parser in Section 3.2 shares the input stream s and causes a space leak. Instead we use a definition of *parse* that returns a pair of the tree and the unprocessed tail of the stream, such that the input stream can be freed after being used. However, its space behaviour is compiler-dependent, due to a space leak of when returning pairs, addressed by Wadler [29]. The fix he proposed is actually implemented in both NHC98 [23] and GHC [7], but is fragile in presence of other valuable optimisations of GHC [12].

Example 2 shows a problem more intrinsic to the nature of lazy evaluation. The list of keywords appears very late and remain unevaluated until it is finally output. This is in fact what we expect of lazy evaluation. However, the thunk contains a reference to the beginning of the input stream, which means that the entire input stream will reside in memory. Put it in a wider context, we recall Wadler's claim [29] that we need a parallel evaluator to avoid certain classes of space leaks. Our xsp implementation, which evaluates all the states $q[1]$ indexed 1, can actually be seen as a parallel evaluator specialised for XML processing.

Streaming for Existing XML Transformation Languages. It has been shown how to convert XPath expressions into attributed tree transducers [21], which is weaker than mfts. Can we convert functions defined in languages such as XSLT [34], *fxt* [3], XDuce [11], or CDuce [2], into mft's?

TL [17] is like mft, but supports pattern matching by *monadic second-order logic* (MSO) formulae. Each TL rule has the form $q(\phi, y_1, \ldots, y_n) \to rhs$, where ϕ is an MSO formula. When q is called, the nodes satisfying ϕ is passed as it argument. Maneth et al. showed that most practical TL programs use only MSO formulae that does not select ancestor nodes, and such programs can be represented by a deterministic mft. It implies that XSLT programs using only forward XPath expressions can be expressed as mft's.

XDuce and CDuce support regular expression pattern [10]. The following *tail-capturing* XDuce program can be captured by an mft:

```
fun mkTelList (val e as (Name,Addr,Tel?)*) =
   match e with name[val n], addr[val a], tel[val t], val rest
                  -> name[n], tel[t], mkTelList (rest)
              | name[val n], addr[val a], val rest -> mkTelList (rest)
              | () -> ()
```

$$
\begin{array}{ll}
MkTelList(\texttt{name}\langle x_1\rangle x_2) & Name(\texttt{addr}\langle x_1\rangle x_2, y_1) \to NameAddr(x_2, y_1) \\
\quad \to Name(x_2, \texttt{name}\langle Val(x_1)\rangle) & NameAddr(\texttt{tel}\langle x_1\rangle x_2, y_1) \to y_1\ \texttt{tel}\langle Val(x_1)\rangle\ MkTelList(x_2) \\
MkTelList(\epsilon) \to \epsilon & NameAddr(\texttt{name}\langle x_1\rangle x_2, y_1) \to Name(x_2, \texttt{name}\langle Val(x_1)\rangle) \\
 & NameAddr(\epsilon, y_1) \to y_1
\end{array}
$$

Here we extend mft's to handle text data, and *Val* is the identity function for text. This mft is total if inputs are restricted to the type `(Name,Addr,Tel?)*` specified by the original XDuce program. Hosoya and Pierce [10] talked about how to convert non-tail-capturing patterns into tail-capturing equivalents. It will be among our future work to see how this approach works in general.

The mft can be extended to handle other datatypes. For example, we can extend the right-hand side with booleans, boolean operators, and conditional

branches: $rhs ::= \ldots \mid true \mid false \mid if(rhs, rhs, rhs)$, and correspondingly extend the xsp with some extra rules [21]: $if(true, e_1, e_2) \to e_1$ and $if(false, e_1, e_2) \to e_2$. Some extra care is needed to ensure that *if* is always in the top set of a pushdown. With booleans and conditionals we can express transformations including the `invite`/`visit` iteration with XPath expressions in XTiSP [19].

Limitation. The goal of xsp is to ensure that the input stream does not reside in memory. On the other hand, the space occupied by temporary result of the computation is a separate issue related to the nature of the computation performed. Certain transformations are *inherently memory inefficient* [25]. For example, if we replace the two rules of M_{rev} for $\mathtt{r}\langle x_1\rangle x_2$ and $\sigma\langle x_1\rangle x_2$ with a single rule: $Main(\sigma\langle x_1\rangle x_2) \to Rev(x_2, \sigma\langle Rev(x_1, \epsilon)\rangle)$, the mft (call it M_{frev}) reverses the subtrees for all nodes. The derived xsp still efficiently consumes the input stream, but the temporary expression grows linearly. Every SAX-like stream processing program has the same problem. As a trial experiment, Table 1(c) compares M_{frev} and a program **direct impl. mft** which simply loads the tree and performs the full reverse. The result shows that our implementation does not carry too much overhead even for this inherently inefficient transformation.

Memory used by the xsp's in this paper are all minimum for the desired computation, which is not true in general and remains a future work to analyse.

7 Conclusion and Related Work

We have presented a method to automatically derive an XML stream processor from a program expressed as a macro forest transducer. The XML stream processor has an efficient implementation based on a pushdown machine. The framework presented in this paper will be the core of the next release of XTiSP [19]. We believe that the mft is expressive enough that we can transform most practical programs written in existing XML processing languages [11, 2, 34] to mft, in order to streamlise them.

Most of the work devoted to automatic derivation of XML stream processors from declarative programs focus on query languages, such as XPath [1, 5, 8, 9] and a subset of XQuery [16]. They are not expressive enough to describe some useful transformation such as the structure-preserving transformation renaming all the labels `a` to `b`. The key idea of our framework was presented in the first author's previous work [21, 25], based on the composition of (stack-)attributed tree transducers (att) [20]. All programs definable in the XML transformation language XTiSP [21, 19] can be translated into att's, which are less expressive than mft's [6, 26]. Our result in this paper is therefore stronger. The formalisation here helps to produce the next version of XTiSP that is both correct and efficient.

Kodama, Suenaga, Kobayashi and Yonezawa [15] studied *ordered-linear typed* programs and how to buffer the input and process the buffered tree. In a subsequent paper [28], they tried to derive stream processors by automatically detecting which input should be buffered. The restrictions imposed by ordered linear type may not always be preferred for stream processing. In EXAMPLE 2, where

one argument is shared by two functional calls, our stream processor still consumes the input as its tokens are read. An ordered linear typeable alternative would keep a copy of the input in memory until it is pattern-matched.

Kiselyov [14] proposed defining XML transformation using a function `foldts` over rose trees. and actions `fup`, `fdown` and `fhere`. This programming style is not flexible enough and many function closures are created. STX [4] is a template-based XML transformation language that operates on stream of SAX [32] events. While the programmers can define XML transformation as well as XSLT [34], they have to explicitly manipulate the environment. TransformX by Scherzinger and Kemper [27] provides a framework for syntax-directed transformations of XML streams, using attribute grammar on the type schema for inputs. However, we must still keep in mind which information should be buffered before and after reading each subtree in the input.

Acknowledgment. The authors wish to express their gratitude to Zhenjiang Hu and Giuseppe Castagna for their comments and advice on earlier drafts.

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. A. Berlea and H. Seidl. fxt – a transformation language for XML documents. *Journal of Computing and Information Technology*, 10(1):19–35, 2002.
4. 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/`.
5. 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.
6. J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71–146, 1985.
7. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.
8. 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.
9. 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.
10. H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6):961–1004, Novermber 2003.
11. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
12. S. P. Jones. Space usage. Glasgow Haskell Users Mailing List, `http://www.haskell.org/pipermail/glasgow-haskell-users/2004-August/007023.html`, 17th August 2004.

13. M. Kay. SAXON: The XSLT and XQuery processor. http://saxon.sourceforge.net/.
14. 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.
15. 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.
16. 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.
17. S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 283–294, 2005.
18. M. Murata. Extended path expressions of XML. In *Proceedings of the 20th ACM Symp. on Principles of Database Systems*, pages 153–166, 2001.
19. K. Nakano. XTiSP: XML transformation language intended for stream processing. http://xtisp.org/.
20. K. Nakano. Composing stack-attributed transducers. Technical Report METR-2004-01, Department of Mathematical Informatics, University of Tokyo, 2004.
21. 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.
22. K. Nakano and S.-C. Mu. A pushdown machine for recursive XML processing (full version). http://www.ipl.t.u-tokyo.ac.jp/~ksk/en/?Publication.
23. The nhc98 compiler. http://www.haskell.org/nhc98/.
24. S. Nishimura. Fusion with stacks and accumulating prameters. In *the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 101–112, 2004.
25. S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*, 54:257–290, 2005.
26. T. Perst and H. Seidl. Macro forest transducers. *Information Processing Letters*, 89:141–149, 2004.
27. S. Scherzinger and A. Kemper. Syntax-directed transformations of XML streams. In *Workshop on Programming Language Technologies for XML*, pages 75–86, 2005.
28. 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.
29. P. Wadler. Fixing a space leak with a garbage collector. *Software Practice and Experience*, 17(9):595–608, September 1987.
30. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358, 1988.
31. libxslt: the XSLT C library for Gnome. http://xmlsoft.org/XSLT/.
32. SAX: the simple API for XML. http://www.saxproject.org/.
33. XMark: an XML benchmark project. http://www.xml-benchmark.org/.
34. XSL transformations (XSLT). http://www.w3c.org/TR/xslt/.