A Pushdown Machine for Recursive XML Processing[*]
(full version)

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. Their direct implementation, however, causes inefficient memory usage because the input XML tree is completely built in memory before being processed. In contrast, programs in stream processing style minimise memory usage since it may effectively release the memory occupied by a prefix of the input not needed by the computation. Stream processing programs, however, are harder to write because the programmer is left with the burden to maintain the 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 `a⟨r⟨b⟨c⟨⟩d⟨⟩⟩e⟨⟩⟩f⟨⟩⟩` is transformed into `a⟨r⟨e⟨⟩b⟨d⟨⟩c⟨⟩⟩⟩f⟨⟩⟩`.

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. The forest is then transformed by the function, 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, we may ask the programmer to switch to another programming style facilitated by APIs such as 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 `$`. Figure 2 shows what a stream processor would do to accomplish the same task, given the input `<a><r><b><c></c><d></d></b><e></e></r><f></f></a>`. The function $\mathcal{P}$ takes an input XML event and an environment[3], and returns an updated environment, possibly

---

[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.

$$Main(\epsilon) = \epsilon$$
$$Main(\mathbf{r}\langle x_1\rangle x_2) = \mathbf{r}\langle rev(x_1, \epsilon)\rangle(Main(x_2))$$
$$Main(\sigma\langle x_1\rangle x_2) = \sigma\langle main\ x_1\rangle(Main(x_2)) \quad \text{if } \sigma \neq \mathbf{r}$$

$$Rev(\epsilon,\ y) = y$$
$$Rev(\sigma\langle x_1\rangle x_2,\ y) = Rev(x_2,\ \sigma\langle Rev\ x_1\ \epsilon\rangle\ y)$$

Fig. 1. A functional program reversing the subtrees under nodes labelled $\mathbf{r}$.

input:

```
<a>    <r>    <b>    <c>   </c>    <d>   </d>   </b>    <e>   </e>   </r>    <f>   </f>   </a>    $
```

$\gamma_0 \to \mathcal{P} \to \gamma_1 \to \mathcal{P} \to \gamma_2 \to \mathcal{P} \to \gamma_3 \to \mathcal{P} \to \gamma_4 \to \mathcal{P} \to \gamma_5 \to \mathcal{P} \to \gamma_6 \to \mathcal{P} \to \gamma_7 \to \mathcal{P} \to \gamma_8 \to \mathcal{P} \to \gamma_9 \to \mathcal{P} \to \gamma_{10} \to \mathcal{P} \to \gamma_{11} \to \mathcal{P} \to \gamma_{12} \to \mathcal{P} \to \gamma_{13} \to \mathcal{P} \to \gamma_{14} \to \mathcal{P} \to$

```
<a>    <r>                                                      <e></e>   <f>   </f>   </a>
                                                                <b><d></d>
output:                                                         <c></c></b></r>
```
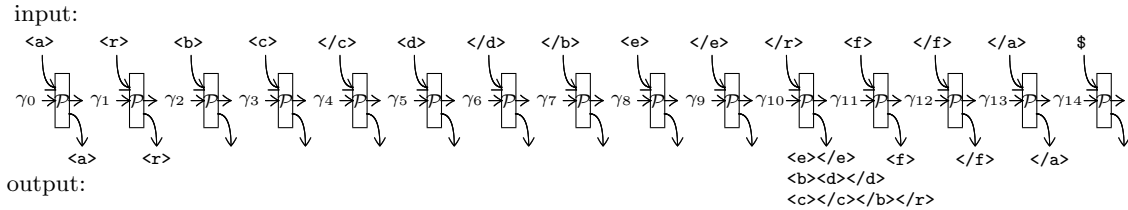
Fig. 2. A flow of stream processing

outputting some XML events. The initial environment in the running example is γ_0. Reading the first event `<a>` yields a new environment γ_1 and an output event `<a>`. 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 partially computed result in the environment. While stream processing saves memory usage, it is much harder to program in this style. Each problem seems to employ a different data structure to represent the environment, and burden is on the programmer to explicitly maintain it.

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.

Some readers may wonder, can we not use a non-strict programming language, compose a function on forests with a parser constructing the forest on demand, and simply let lazy evaluation do the job? In the latter part of this paper, we will talk about the difficulties of relying on lazy evaluation alone, and the necessities of a more specific model for XML streaming.

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*. Reading a token triggers a rewriting on the expression. It can be argued that our stream processor is of a higher level than finite automaton based approaches. Yet, for the class of stream processors we are concerned with, there is an efficient method, basing on a pushdown machine, to implement the rewriting.

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 give a proof that the method works for our tdtt, which has a infinite number of states.

The XML stream processor in this paper is implemented in Objective Caml [25] and compared with a few existing tools for forest-style XML transformation.

Outline In the next section, we introduce a simplified model of XML documents and mft's. Section 3 gives a formal model of XML stream processors and its derivation from an mft. Section 4 proposes an efficient implementation of the stream processor based on a pushdown machine. In Section 5 we show the benchmark result for a simple example. Section 6 discusses capabilities of

our framework. In Section 7 we review some related work and conclude the paper. Appendix A gives a proof of the correctness of our derivation.

2 XML and the Macro Forest Transducer

This section formally defines our model of XML and the mft. For simplicity, we deal with a simplified model of XML with only element nodes, and assume that the input XML is well-formed. It is easy to extend our approach to capture other aspects of XML, and it is always possible to perform a well-formedness check before hand.

2.1 Forests and XML Streams

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

$$f ::= \sigma\langle f\rangle f \quad | \quad \epsilon,$$

where $\sigma \in \Sigma$ and ϵ denotes the empty forest. We denote by $\mathcal{F}_\Sigma$ the set of Σ-forests. A Σ-forest $\mathsf{a}\langle\mathsf{b}\langle\epsilon\rangle\mathsf{c}\langle\epsilon\rangle\epsilon\rangle\epsilon$ with $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ 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 $\Sigma_{\Leftrightarrow}$, is defined by $\Sigma_{\Leftrightarrow} = \{\texttt{<}\sigma\texttt{>} \mid \sigma \in \Sigma\} \cup \{\texttt{</}\sigma\texttt{>} \mid \sigma \in \Sigma\}$. An XML stream is a sequence of Σ-*events*. We denote by $\Sigma_{\Leftrightarrow}^{\circ}$ 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 $\Sigma_{\Leftrightarrow\$}$ for $\Sigma_{\Leftrightarrow} \cup \{\$\}$.

Let Σ be an alphabet. The *streaming* of a forest is the function $\lfloor\_\rfloor : \mathcal{F}_\Sigma \to \Sigma_{\Leftrightarrow}^{\circ}$ 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\mathsf{a}\langle\mathsf{b}\langle\rangle\mathsf{c}\langle\rangle\rangle\rfloor = $ `<a><b></b><c></c></a>`.

2.2 Macro Forest Transducers

The *macro forest transducer* (mft) was proposed by Perst and Seidl [26] as an extension to the *macro tree transducer* (mtt) [6] by taking forest 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. A natural question is how expressive it is under these restrictions. It was shown that mft is in fact rather expressive [17], and it has been shown how to convert XPath expressions into a computation model weaker than mfts [21]. More discussions will be given in Section 6, where we also show how to convert certain classes of functional programs to mft.

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* (mft) 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 defined 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:

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

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 $Q = \{Main, Rev\}$, $\Sigma = \Delta$ be some alphabet containing $\mathtt{r}$ and R be the rules in Figure 1 (with the = sign replaced by $\to$), then $M_{rev} = (Q, \Sigma, \Delta, Main, R)$ 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 alphabets and the XML/HTML tags below),
$R = \{\ Main(\mathtt{article}\langle x_1 \rangle x_2) \to \mathtt{html}\langle \mathtt{head}\langle Title(x_1) \rangle \mathtt{body}\langle InArticle(x_1, \epsilon) \rangle \rangle$,
  $Title(\mathtt{title}\langle x_1 \rangle x_2) \to \mathtt{title}\langle Copy(x_1) \rangle$,
  $InArticle(\mathtt{title}\langle x_1 \rangle x_2, y_1) \to \mathtt{h1}\langle Copy(x_1) \rangle InArticle(x_2, y_1)$,
  $InArticle(\mathtt{para}\langle x_1 \rangle x_2, y_1) \to \mathtt{p}\langle Key2Em(x_1) \rangle InArticle(x_2, y_1\ AllKeys(x_1))$,
  $InArticle(\mathtt{ps}\langle x_1 \rangle x_2, y_1) \to \mathtt{h2}\langle \mathtt{Index}\langle \rangle \rangle\ \mathtt{ul}\langle y_1 \rangle\ \mathtt{h2}\langle \mathtt{Postscript}\langle \rangle \rangle\ Copy(x_1)$,
  $Key2Em(\mathtt{key}\langle x_1 \rangle x_2) \to \mathtt{em}\langle Copy(x_1) \rangle\ Key2Em(x_2)$,
  $Key2Em(\sigma\langle x_1 \rangle x_2) \to \sigma\langle Key2Em(x_1) \rangle\ Key2Em(x_2)$ $(\sigma \neq \mathtt{key})$, $Key2Em(\epsilon) \to \epsilon$,
  $AllKeys(\mathtt{key}\langle x_1 \rangle x_2) \to \mathtt{li}\langle Copy(x_1) \rangle AllKeys(x_2)$,
  $AllKeys(\sigma\langle x_1 \rangle x_2) \to AllKeys(x_1)AllKeys(x_2)$ $(\sigma \neq \mathtt{key})$, $AllKeys(\epsilon) \to \epsilon$,
  $Copy(\sigma\langle x_1 \rangle x_2) \to \sigma\langle Copy(x_1) \rangle Copy(x_2)$ $(\sigma \in \Sigma)$, $Copy(\epsilon) \to \epsilon\ \}$

3 XML Stream Processors and its Derivation

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

Definition 4. An *XML stream processor* (xsp) 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 $s \in \Sigma_{\texttt{<>}}^{\circ}$. The *transition* over Tmp_S for an input Σ-event is a function $\langle\!\langle \_\_, \_\_ \rangle\!\rangle : Tmp_S \times \Sigma_{\texttt{<>\$}} \to Tmp_S$ defined by

- $\langle\!\langle \varepsilon, \chi \rangle\!\rangle = \varepsilon$,
- $\langle\!\langle \texttt{<}\delta\texttt{>}e, \chi \rangle\!\rangle = \texttt{<}\delta\texttt{>}\langle\!\langle e, \chi \rangle\!\rangle$ where $\delta \in \Delta$,
- $\langle\!\langle \texttt{</}\delta\texttt{>}e, \chi \rangle\!\rangle = \texttt{</}\delta\texttt{>}\langle\!\langle e, \chi \rangle\!\rangle$ where $\delta \in \Delta$,
- $\langle\!\langle q(e_1, \ldots, e_n)e, \chi \rangle\!\rangle = (rhs[y_j := \langle\!\langle e_j, \chi \rangle\!\rangle]_{j=1,\ldots,n})\langle\!\langle e, \chi \rangle\!\rangle$ 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 one by one, and updates the temporary expression with the transition $\langle\!\langle \_\_, \_\_ \rangle\!\rangle$. The end of the stream is marked by $\texttt{\$}$. Let $\chi_1 \chi_2 \ldots \chi_k$ be the input stream with each $\chi_j \in \Sigma_{\texttt{<>}}$. The final expression is

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

Note that the final temporary expression is always in $\Delta_{\texttt{<>}}^{\circ}$ 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 $\tau_S : \Sigma_{\texttt{<>}}^{\circ} \to \Delta_{\texttt{<>}}^{\circ}$ defined by $\tau_S(s) = \theta_S(in(\varepsilon, \ldots, \varepsilon), s\texttt{\$})$ where, for $e \in Tmp_S$,

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

We will see some examples in the next section.

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 $M = (Q, \Sigma, \Delta, in, R)$, an input 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

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

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. Functional programmers may alert that the first clause seems to imply that s is traversed twice. However, it is a common style of tdtt specifications, and multiple traversals 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 6.1.

Some previous work [23, 21, 24] 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, 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. We will give a new proof of the fusion in Appendix A.

The rationale behind the derivation is as follows. Consider an 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 $rank(q[i]) = rank(q) - 1$ for every $q \in Q$ and $i \in \mathbb{N}^+$,
- $in' = in[1] \in Q'$,
- 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:

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

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:

$$\epsilon_\Sigma(i, \chi) = (i = 1 \wedge \chi = \texttt{</}\sigma\texttt{>}) \vee \chi = \texttt{\$}, \text{ with } \sigma \in \Sigma.$$

The correctness of the derivation is stated by the following theorem whose proof is given in Appendix A.

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' of rules is:

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

$\qquad Main[1]() \xrightarrow{\texttt{<}\sigma\texttt{>}} \texttt{<}\sigma\texttt{>}\ Main[1]()\ \texttt{</}\sigma\texttt{>}\ Main[2]()$
$\qquad\quad (\sigma \neq \mathtt{r}),$

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

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

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

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

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

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

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

Figure 3 shows a sample run when the input is `<a><r><b><c></c><d></d><e></e></r><f></f></a>$`.

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

$R' = \{$ $Main[1]() \xrightarrow{\texttt{<article>}}$ `<html> <head>` $Title[1]()$ `</head> <body>` $InArticle[1](\varepsilon)$ `</body> </html>`,

   $Title[1]() \xrightarrow{\texttt{<title>}}$ `<title>` $Copy[1]()$ `</title>`,

   $InArticle[1](y_1) \xrightarrow{\texttt{<title>}}$ `<h1>` $Copy[1]()$ `</h1>` $InArticle[2](y_1)$,

   $InArticle[1](y_1) \xrightarrow{\texttt{<para>}}$ `<p>` $Key2Em[1]()$ `</p>` $InArticle[2](y_1\ AllKeys[1]())$,

   $InArticle[1](y_1) \xrightarrow{\texttt{<ps>}}$ `<h2> Index </h2> <ul>`y_1 `</ul> <h2> Postscript </h2>` $Copy[1]()$,

   $Key2Em[1]() \xrightarrow{\texttt{<key>}}$ `<em>` $Copy[1]()$ `</em>` $Key2Em[2]()$,

   $Key2Em[1]() \xrightarrow{\texttt{<}\sigma\texttt{>}}$ `<`σ`>` $Key2Em[1]()$ `</`σ`>` $Key2Em[2]()$ $(\sigma \neq \texttt{key})$,

   $AllKeys[1]() \xrightarrow{\texttt{<key>}}$ `<li>` $Copy[1]()$ `</li>` $AllKeys[2]()$,

   $AllKeys[1]() \xrightarrow{\texttt{<}\sigma\texttt{>}} AllKeys[1]()AllKeys[2]()$ $(\sigma \neq \texttt{key})$,

   $Copy[1]() \xrightarrow{\texttt{<}\sigma\texttt{>}}$ `<`σ`>` $Copy[1]()$ `</`σ`>` $Copy[2]()$ $(\sigma \in \Sigma)$,

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

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

   $q[i]() \xrightarrow{\chi} \varepsilon$ $(\text{if } \epsilon_\Sigma(i, \chi))$,

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

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

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

$Main[1]() \xrightarrow{\texttt{<a>}}$ `<a>`$Main[1]()$`</a>`$Main[2]()$

   $\xrightarrow{\texttt{<r>}}$ `<a><r>`$Rev[1](\varepsilon)$`</r>`$Main[2]()$`</a>`$Main[3]()$

   $\xrightarrow{\texttt{}}$ `<a><r>`$Rev[2]($`<b>`$Rev[1](\varepsilon)$`</b>`$)$`</r>`$Main[3]()$`</a>`$Main[4]()$

   $\xrightarrow{\texttt{<c>}}$ `<a><r>`$Rev[3]($`<b>`$Rev[2]($`<c>`$Rev[1](\varepsilon)$`</c>`$)$`</b>`$)$`</r>`$Main[4]()$`</a>`$Main[5]()$

   $\xrightarrow{\texttt{</c>}}$ `<a><r>`$Rev[2]($`<b>`$Rev[1]($`<c></c>`$)$`</b>`$)$`</r>`$Main[3]()$`</a>`$Main[4]()$

   $\xrightarrow{\texttt{<d>}}$ `<a><r>`$Rev[3]($`<b>`$Rev[2]($`<d>`$Rev[1](\varepsilon)$`</d><c></c>`$)$`</b>`$)$`</r>`$Main[4]()$`</a>`$Main[5]()$

   $\xrightarrow{\texttt{</d>}}$ `<a><r>`$Rev[2]($`<b>`$Rev[1]($`<d></d><c></c>`$)$`</b>`$)$`</r>`$Main[3]()$`</a>`$Main[4]()$

   $\xrightarrow{\texttt{}}$ `<a><r>`$Rev[1]($`<b><d></d><c></c></b>`$)$`</r>`$Main[2]()$`</a>`$Main[3]()$

   $\xrightarrow{\texttt{<e>}}$ `<a><r>`$Rev[2]($`<e>`$Rev[1](\varepsilon)$`</e><b><d></d><c></c></b>`$)$`</r>`$Main[3]()$`</a>`$Main[4]()$

   $\xrightarrow{\texttt{</e>}}$ `<a><r>`$Rev[1]($`<e></e><b><d></d><c></c></b>`$)$`</r>`$Main[2]()$`</a>`$Main[3]()$

   $\xrightarrow{\texttt{</r>}}$ `<a><r><e></e><b><d></d><c></c></b></r>`$Main[1]()$`</a>`$Main[2]()$

   $\xrightarrow{\texttt{<f>}}$ `<a><r><e></e><b><d></d><c></c></b></r><f>`$Main[1]()$`</f>`$Main[2]()$`</a>`$Main[3]()$

   $\xrightarrow{\texttt{</f>}}$ `<a><r><e></e><b><d></d><c></c></b></r><f></f>`$Main[1]()$`</a>`$Main[2]()$

   $\xrightarrow{\texttt{}}$ `<a><r><e></e><b><d></d><c></c></b></r><f></f></a>`$Main[1]()$

   $\xrightarrow{\$}$ `<a><r><e></e><b><d></d><c></c></b></r><f></f></a>`

Fig. 3. Stream processing induced by $\mathcal{SP}(M_{rev})$

| main output stream : | $[\,]_{\nu_1}$ `</r>` $[\,]_{\nu_2}$ `</a>` $[\,]_{\nu_3}$ | | $[\,]_{\nu_1}$ `</r>` $[\,]_{\nu_2}$ `</a>` $[\,]_{\nu_3}$ |

Figure 4 content:

main output stream : $[\,]_{\nu_1}$ `</r>` $[\,]_{\nu_2}$ `</a>` $[\,]_{\nu_3}$ $[\,]_{\nu_1}$ `</r>` $[\,]_{\nu_2}$ `</a>` $[\,]_{\nu_3}$

$\xrightarrow{\texttt{<c>}}$

pushdown :

| $\{\, r(\varepsilon)/\nu_4 \,\}$ |
| $\{\, r(\texttt{}\,[\,]_{\nu_4}\,\texttt{})/\nu_1 \,\}$ |
| $\{\, m()/\nu_2 \,\}$ |
| $\{\, m()/\nu_3 \,\}$ |

| $\{\, r(\varepsilon)/\nu_5 \,\}$ |
| $\{\, r(\texttt{<c>}\,[\,]_{\nu_5}\,\texttt{</c>})/\nu_6 \,\}$ |
| $\{\, r(\texttt{}\,[\,]_{\nu_6}\,\texttt{})/\nu_1 \,\}$ |
| $\{\, m()/\nu_2 \,\}$ |
| $\{\, m()/\nu_3 \,\}$ |

temporary expression : e_{before} e_{after}
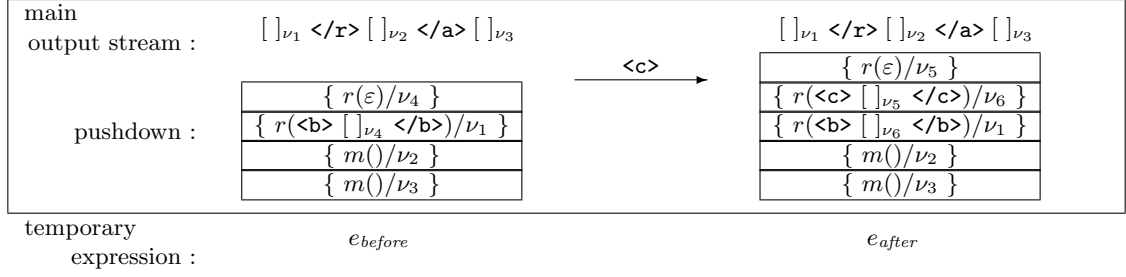
Fig. 4. Pushdown representation for temporary expressions and its updating

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: most rules are introduced by **xsp-(3)**, which merely increment or decrement the indexes of the states, and term expansion happens only for states indexed 1, triggered by rules in **xsp-(1)**. Such behaviour resembles a stack. In this section, we present an efficient implementation of the xsp's derived from mft's, using a pushdown machine based representation of the temporary expressions.

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 3, 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 in of the temporary expression that remains 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 performed efficiently.

We represent a temporary expression by a pair of a *main output stream* and a *pushdown*, as shown in Figure 4. 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: $r[2](\ldots)$, `</r>`, $m[3]()$, `</a>` 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 $q(\ldots)$ and a hole address ν, denoted by $q(\ldots)/\nu$. 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 the 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.

In this example, the main output stream remains the same after the updating. In general, the main output stream will be altered when it contains an address referred by the top set of the pushdown, or when its prefix is ready to be squeezed.

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

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

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 ø. 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].

Now we define formally how a temporary expression is represented.

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

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

where $\langle m, p \rangle = pd(e)$, $\langle m_i, p_i \rangle = pd(e_i)$ and ν is a fresh address. We 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. The function pd^{-1} is defined by

$$pd^{-1}(\langle \varepsilon, p \rangle) = \varepsilon$$
$$pd^{-1}(\langle \text{<}\delta\text{>} \; m, p \rangle) = \text{<}\delta\text{>} \; pd^{-1}(\langle m, p \rangle)$$
$$pd^{-1}(\langle \text{</}\delta\text{>} \; m, p \rangle) = \text{</}\delta\text{>} \; pd^{-1}(\langle m, p \rangle)$$
$$pd^{-1}(\langle [\,]_\nu \; m, p \rangle) = (p\%\nu) \; pd^{-1}(\langle m, p \rangle)$$

where $p\%\nu$ accesses the corresponding state frame in the pushdown and expands the arguments of the state. When a state frame $q(m_1, \ldots, m_n)/\nu$ is an element of the set $p(i)$, we have $p\%\nu = q[i](pd^{-1}(\langle m_1, p \rangle), \ldots, pd^{-1}(\langle m_n, p \rangle))$. For any temporary expression e, we have

$$e = pd^{-1}(pd(e)) \tag{2}$$

whose proof is omitted.

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

---

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

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. because we know the physical address of the hole. 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 $\langle\!\!\!\langle \_\_, \_\_ \rangle\!\!\!\rangle$ 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 $\langle\!\!\!\langle \_\_, \_\_ \rangle\!\!\!\rangle : Pdr_M \times \Sigma_{<>\$} \to Pdr_M$. For a pushdown representation $\langle m, p \rangle \in Pdr_M$ and input event $\chi \in \Sigma_{<>\$}$, a new pushdown representation $\langle\!\!\!\langle \langle m, p \rangle, \chi \rangle\!\!\!\rangle$ is given as follows:

– $\langle\!\!\!\langle \langle m, p \rangle, \texttt{<}\sigma\texttt{>} \rangle\!\!\!\rangle = \Phi_\sigma(m, (\triangleright p) \ll \emptyset \ll \emptyset, p(1))$, where the 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 $\langle m', p' \rangle = pd^\circ(\mathcal{A}(rhs^{q,\sigma}))[y_j := m_j]_{j=1,\ldots,n}$.

– $\langle\!\!\!\langle \langle m, p \rangle, \texttt{</}\sigma\texttt{>} \rangle\!\!\!\rangle = \Phi_\epsilon(m, \triangleright p, p(1))$, 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 $\langle m', p' \rangle = pd^\circ(\mathcal{A}(rhs^{q,\epsilon}))[y_j := m_j]_{j=1,\ldots,n}$.

– $\langle\!\!\!\langle \langle m, p \rangle, \$ \rangle\!\!\!\rangle = \Phi_\epsilon(m, \emptyset, \bigcup_{d \in \mathbb{N}^+} p(d))$, where Φ_ϵ is as given in the previous case of $\chi = \texttt{</}\sigma\texttt{>}$.

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 $\tau_P : \Sigma_{<>}^\circ \to \Delta_{<>}^\circ$ induced by P a defined in a way similar to $\tau_{\mathcal{SP}(M)}$:

$$\tau_P(s) = \zeta_P(\langle [\ ]_{\nu_0}, \{1 \mapsto \{in(\varepsilon, \ldots, \varepsilon)/\nu_0\}\} \rangle, s\$)$$

where

$$\zeta_P(\langle m, p \rangle, \varepsilon) = m \qquad\qquad \zeta_P(\langle m, p \rangle, \chi s) = \zeta_P(\langle\!\!\!\langle \langle m, p \rangle, \chi \rangle\!\!\!\rangle, s)$$

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 $\texttt{<}\sigma\texttt{>}$ 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 $q(\ldots)/\nu$ 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 $q(\ldots)/\nu$ 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 $\texttt{</}\sigma\texttt{>}$ is read. In the xsp $\mathcal{SP}(M)$, every state $q[i]$ $(i > 1)$ is rewritten to $q[i-1]$. In the pushdown machine $\mathcal{PD}(M)$, the corresponding state frame $q(\ldots)/\nu$ in the i-th set of the pushdown ascends by one level after popping the pushdown. 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 $q(\ldots)/\nu$ in the top set of the pushdown, the hole $[\,]_\nu$ is filled according to $\mathcal{A}(rhs^{q,\epsilon})$.

After reading the end of stream, the pushdown must be empty since $\mathcal{A}(rhs^{q,\epsilon})$ contains no pattern $q[i](\ldots)$ and all state frames in the previous pushdown is consumed by $\Phi_\epsilon(s, \varnothing, \bigcup_{d \in \mathbb{N}^+} p(d))$. Thereby the final main output stream does not contain any hole.

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, since they are not going to change by subsequent transitions. We define the following function sqz for output streams:

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

where $(m', m'') = sqz(m)$. The function sqz splits the leading output events and the rest of the output stream, i.e., if $(m', m'') = sqz(m)$, then $m'm'' = m$.

We can then redefine ζ_P with sqz as follows:

$$\zeta_P(\langle m, p \rangle, \varepsilon) = m \qquad\qquad \zeta_P(\langle m, p \rangle, \chi s) = m'\ \zeta_P(\langle\!\langle m'', p \rangle, \chi \rangle\!\rangle, s)$$

where $(m', m'') = sqz(\langle\!\langle m, p \rangle, \chi \rangle\!\rangle)$.

Remark. In the actual implementation, we have to take care of dealing with mft rules whose right-hand side does not contain exactly one occurrence of parameter variable y_j for each j.

Consider the case that there are some variables which does not occur in the right-hand side. For example, let an mft have a rule $q(pat, y_1) \to \epsilon$. Then the corresponding xsp has a 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. In order to avoid ineffectual updating after that, all hole addresses contained in m_1 should be discarded as long as the hole does not occur in the other position.

Consider the case that there are some variables which occurs more than once in the right-hand side. If we use doubly-linked cyclic lists to represent main output streams for efficient updating, we should be carefully deal with them. For example, let an mft have a rule $q(pat, y_1) \to y_1\ y_1$. Then the corresponding xsp has a rule $q[1](y_1) \xrightarrow{\chi} y_1\ y_1$. In this case, 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. To solve this problem, we mark the state frame so as to remember 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)

Table 1. Performance comparison for varying input size

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)

(a) For a practical 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)

(b) For a full reverse transformation M_{frev}

Table 2. Benchmark result of other transformations

5 Benchmarking Results

For benchmarking, we use the random sample generator XMark [33] to produce sample XML documents of various sizes: 1MB, 4MB, 16MB, 64MB and 256MB. Every sample document contains a sequence of `item` nodes, each having a list of children about a dozen lines long. The benchmarking task would be to reverse the order of the subtrees under `item`.

The pushdown machine automatically derived from the mft specification of 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[5]. 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 compares the total execution time and maximum memory size in seconds and megabytes.

As we expected, **pushdown xsp** uses a much smaller heap comparing too all other entries. 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, etc. 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 xsp saves the overhead of building the trees. Therefore we expect that the xsp approach will also deliver competitive speed even after being scaled to handle full XML.

We also conducted performance comparisons for other transformations. The result is shown in Table 2. For simplicity, we compared only **pushdown xsp** and **direct impl. mft** for random inputs of 4MB and 64MB. Table (a) shows comparison results for a practical transformation M_{htm} shown in EXAMPLE 2. This result also indicates a small heap consumption of **pushdown xsp** with elapsed time similar to **direct impl. mft**. Table (b) shows results for a full reverse transformation M_{frev} which will be discussed later in Section 6.3.

6 Discussion

This section discusses capabilities of our framework. Firstly we show the difference between lazy evaluation mechanism and ours. Next possibilities of our framework for existing XML transformation languages are discussed. Finally we describe some limitation of stream processing.

---

[5] Since a straightforward implementation of an mft may contain lots of append operation of lists which is inefficient, we use a 'join list' of trees rather than a 'cons list' of trees for an output forest.

6.1 Comparison with Lazy Evaluation

Many of our readers wondered: "Is this all necessary? Can we not just use lazy evaluation?" Consider the program $unparse\ (trans\ (parse\ input))$ in a non-strict functional 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?

To answer the question, we run a number of experiments in the functional language Haskell. We start with choosing a parser to be fused with. The tdtt-based parser in Section 3.2 is not a good choice because sharing the input stream s among functional causes a space leak. Another possibility is to thread the input stream through recursive calls to the parser, and have the parser return the tree as well as an unprocessed tail of the stream, such as:

$$parse(\texttt{<a>}s) = \textbf{let}\ (ts, s') = parse\ s$$
$$(us, s'') = parse\ s'$$
$$\textbf{in}\ (a\langle ts\rangle us, s'')$$

The intention is that the input stream can be freed right after being used.

Unfortunately, the space behaviour of the parser is compiler-dependent. It runs fine when compiled with NHC98 [22]. With GHC [7], however, the entire input stream resides in memory throughout the execution of the program. The reason is that s'' may be viewed as $snd\ (parse\ (snd\ (parse\ s)))$, so while the recursive calls are being executed, the system still keeps a pointer to s until s'' is finally demanded by the environment. This problem was addressed by Wadler [29], who proposed a fix in the garbage collector to plug the space leak. The fix is actually implemented in both NHC and GHC, but is fragile in presence of otherwise-valuable optimisations that GHC performs [12].

One may argue that the problem is specific to this particular parser. EXAMPLE 2, however, shows a problem more intrinsic to the nature of lazy evaluation. The thunk that evaluates to 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 – if the programmer applies the function $take$ to the generated stream, the thunk need not be evaluated at all. However, the thunk contains a reference to the beginning of the input stream, which means that the entire input stream will reside in memory.

For every xsp example we have, we were able to eventually come up with a Haskell equivalent that uses a minimal amount of memory, by inserting strictness annotations after careful tuning and heap profiling, which was no easy task.

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. This clarifies the relationship between lazy evaluation and xsp. Our xsp implementation, which evaluates all the states $q[1]$ indexed 1, can actually be seen as a parallel evaluator specialised for XML processing.

6.2 Streaming for Existing XML Transformation Languages

Is it possibilities to apply our framework to existing transformation languages? It has been shown how to convert XPath expressions into attributed tree transducers [21], which is weaker than mfts. We would like to know whether we can convert functions defined in languages such as XSLT [34], fxt [3], XDuce [11], or CDuce [2], probably with some restrictions imposed, into mft's.

The XML transformation language TL [17] by Maneth, Berlea, Perst and Seidl may give us some hints about what the restrictions could be. In TL programmers also define a collection of mutual recursive functions. TL is like mft, but supports pattern matching by *monadic second-order logic* (MSO) formulae. Each rule of TL has the form of $q(\phi, y_1, \ldots, y_n) \rightarrow rhs$, 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 in a single deterministic mft. It implies that a simple XSLT program which contains only forward XPath expressions can be expressed as an mft and implemented by our pushdown machine.

XDuce and CDuce support regular expression pattern [10]. They do not bind variables to the parent of the current node. It is easier to see how *tail-capturing* programs can be translated to mft's. For example, the following XDuce program converting an address book to a telephone list:

```
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)
   | () -> ()
```

can be defined by an mft with rules:

$$MkTelList(\texttt{name}\langle x_1\rangle x_2) \rightarrow Name(x_2, \texttt{name}\langle Val(x_1)\rangle)$$
$$MkTelList(\epsilon) \rightarrow \epsilon$$
$$Name(\texttt{addr}\langle x_1\rangle x_2, y_1) \rightarrow NameAddr(x_2, y_1)$$
$$NameAddr(\texttt{tel}\langle x_1\rangle x_2, y_1) \rightarrow y_1 \, \texttt{tel}\langle Val(x_1)\rangle \, MkTelList(x_2)$$
$$NameAddr(\texttt{name}\langle x_1\rangle x_2, y_1) \rightarrow Name(x_2, \texttt{name}\langle Val(x_1)\rangle)$$
$$NameAddr(\epsilon, y_1) \rightarrow y_1$$

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 primitive datatype other than forests. For example, we can extend the right-hand side with booleans, boolean operators, and conditional branches as follows:

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

The corresponding extension we need in the xsp is some extra rules [21]:

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

By applying these rules in each squeezing phase, we achieve an XML stream processing for the extended mft. Additionally we need to keep the position of *if* as a state frame which always locates in the top set of a pushdown. Once we have booleans and conditionals in mft, we can express many transformations, including the `invite`/`visit` iteration with XPath expressions in XTiSP [19], the first author's previous work.

6.3 Limitation

The purpose of stream processing is to save space. However, it is important to distinguish between two kinds of space usage. The design goal of xsp is to be sure that each token of the input stream is consumed and processed immediately, so that the input stream does not reside in memory. On the other hand, the space occupied by temporary result of the computation, represented by the temporary expression, is a separate issue related to the nature of the computation performed.

There is a class of *inherently memory inefficient* transformations [24] that inevitably requires lots of space for the temporary result. For example, if we replace the two rules of M_{rev} for $\mathbf{r}\langle x_1\rangle x_2$ and $\sigma\langle x_1\rangle x_2$ with a single rule: $Main(\sigma\langle x_1\rangle x_2) \rightarrow Rev(x_2, \sigma\langle Rev(x_1, \epsilon)\rangle)$. The mft (call it M_{frev}) reverses the subtrees at every nesting level for all nodes. We can still derive an xsp from it, and the xsp still efficiently consumes the input stream. However, the temporary expression grows in size linear to the length of read input. The resulting xsp is actually a bit slower than the naive load-all implementation, because it cannot output any result until reading the end of the input stream. Every SAX-like stream processing program has the same problem: this kind of transformation is just not suitable for stream processing.

As a trial experiment, Table 2 (b) 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 an inherently inefficient transformation M_{frev}.

In all examples in this paper, the memory used by the derived xsp's are made minimum for the desired computation. This may not be true in general, and it will be a future work to analyse the space usage of an xsp.

7 Conclusion and Related Work

We have presented a method to automatically derive an XML stream processor from a program expressed as the macro forest transducer — mutual recursive functions on XML forests with certain constraints. 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. That will be one of our future work too.

While plenty of work has been devoted to the automatic derivation of XML stream processors from declarative programs, most them deals with query languages, such as XPath [1, 5, 8, 9] and a subset of XQuery [16]. They are not expressive enough to describe some transformation we would like to have, such as the structure-preserving transformation renaming all the labels a to b, which can be expressed naturally in recursive functional style.

The key idea of our framework was presented in the first author's previous work [21, 24], based on the composition of (stack-)attributed tree transducers (att) [20]. All programs definable in XTiSP, an XML transformation language designed by the first author [21, 19], can be translated into att's, which can then be composed with an XML parser in a way similar our derivation of xsp's. However, it is well known that att's are less expressive than mft's [6, 26]. Our result in this paper is therefore much more powerful than before. Moreover, without a formal model of stream processors, some part of the implementation of XTiSP is rather ad-hoc and could have been made more efficient. The formalisation in the present helps to produce an implementation that is both correct and efficient.

Kodama, Suenaga, Kobayashi and Yonezawa [15] studied *ordered-linear type*, which guarantees that the input is processed in top-down, left-to-right order. Given a program satisfying such restrictions, they discussed how to buffer the input and how to process the buffered tree. In a subsequent paper [28], they tried to derive stream processors by automatically detecting which input should be buffered. However, the restrictions imposed by ordered linear type may not always be preferred for stream processing. Consider the right-hand side $\mathrm{p}\langle Key2Em(x_1)\rangle InArticle(x_2, y_1 AllKeys(x_1))$ in EXAMPLE 2, where x_1 is shared by two functional calls. Our stream processor consume both x_1's as its tokens are read. An ordered linear typeable alternative would have to keep a copy of x_1 in memory until it is pattern-matched by *AllKeys*.

Kiselyov [14] gave an XML parser with a general folding function `foldts` over rose trees. An XML transformation is defined by three actions `fup`, `fdown` and `fhere` that 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 start tag `<a>`.

STX [4] is a template-based XML transformation language that operates on stream of SAX [32] events. While the programmers can define the XML transformation program as well as XSLT [34], they have to explicitly write when and how to store the temporary information like stream processing style.

TransformX presented by Scherzinger and Kemper [27] provides a 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.

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/documents/`.
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. The nhc98 compiler. `http://www.haskell.org/nhc98/`.
23. 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.
24. S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*, 54:257–290, 2005.
25. Objective Caml. `http://caml.inria.fr/ocaml/`.
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 *The 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/`.

A   Proof of Theorem 1

We introduce another definition for each of $\lfloor\_\rfloor$ and θ_S to prove THEOREM 1.

An *event generator gen* is another function of streaming $\lfloor\_\rfloor$. Let f be a Σ-forest with an alphabet Σ. The function *gen* can output one by one the next XML event from the XML stream corresponding to f by updating a *following forest list* (for short, FFL). An FFL ranges over a set given by a syntax:

$$L ::= (F, \#) \ \mid \ (F, \sigma) :: L$$

where F ranges over a set of sub-forest of f, $\sigma \in \Sigma$ and $\#$ represents an additional root symbol for f. The i-th forest in an FFL l is accessed by $l[i]$ where $((f', \sigma) :: l)[1] = f'$, $(f', \#)[1] = f'$ $((f', \sigma) :: l)[i] = l[i-1]$ and $(f', \#)[i] = \epsilon$ for $i > 1$.

A event generator *gen* takes an FFL and returns a new FFL following an XML event, which is defined by

$$gen((\sigma\langle f_1 \rangle f_2, \sigma') :: l) = \texttt{<}\sigma\texttt{>}gen((f_1, \sigma) :: (f_2, \sigma') :: l)$$
$$gen((\sigma\langle f_1 \rangle f_2, \#)) = \texttt{<}\sigma\texttt{>}gen((f_1, \sigma) :: (f_2, \#))$$
$$gen((\epsilon, \sigma) :: l) = \texttt{</}\sigma\texttt{>}gen(l)$$
$$gen((\epsilon, \#)) = \texttt{\$}.$$

The function *gen* always terminates because every step reduces a total number of nodes of forests in the FFL. We write $G(l)$ for a set of FFL's occurring as an argument of *gen* in the computation of $gen(l)$, i.e., if $l = (\epsilon, \#)$ then $G(l) = \{(\epsilon, \#)\}$ and otherwise $G(l) = \{l\} \cup G(l')$ with $gen(l) = \chi gen(l')$ with $\chi \in \Sigma_{\texttt{<>}}$. The set $G((f, \#))$ for a $f \in \mathcal{F}_\Sigma$ is finite because of the termination of *gen*. The following lemma shows that *gen* simulates a streaming $\lfloor f \rfloor$ for a forest f when the initial FFL is $(f, \#)$.

Lemma 1. *Let f be a Σ-forest with an alphabet Σ. Then we have*

$$gen((f, \#)) = \lfloor f \rfloor\texttt{\$}. \tag{3}$$

Proof. We use an equation

$$gen((f', \sigma) :: l) = \lfloor f' \rfloor\texttt{</}\sigma\texttt{>}gen(l) \tag{4}$$

for a sub-forest f' of f, $\sigma \in \Sigma$ and an FFL l to show (3). The equation (4) can be proved by induction on the structure of f'. We prove the equation (3) by induction on the structure f. If $f = \epsilon$, then (3) holds by the definitions of *gen* and $\lfloor\_\rfloor$. If $f = \sigma\langle f_1 \rangle f_2$, then we have

$$gen((\sigma\langle f_1 \rangle f_2, \#)) = \texttt{<}\sigma\texttt{>}gen((f_1, \sigma) :: (f_2, \#))$$
$$= \texttt{<}\sigma\texttt{>}\lfloor f_1 \rfloor\texttt{</}\sigma\texttt{>}gen((f_2, \#))$$
$$= \texttt{<}\sigma\texttt{>}\lfloor f_1 \rfloor\texttt{</}\sigma\texttt{>}\lfloor f_2 \rfloor\texttt{\$}$$
$$= \lfloor\sigma\langle f_1 \rangle f_2\rfloor\texttt{\$}$$

from the definitions of *gen* and $\lfloor\_\rfloor$, the equation (4) and the induction hypothesis. Therefore (3) holds for every forest f. $\square$

Now we introduce another definition for θ_S in DEFINITION 7. Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp and f be a Σ-forest. We define a function Θ_S as well as θ_S by

$$\Theta_S(e, (\epsilon, \#)) = \langle e, \$ \rangle \qquad\qquad \Theta_S(e, l) = \Theta_S(\langle e, \chi \rangle, l')$$

for $e \in Tmp_S$ and an FFL l where $gen(l) = \chi gen(l')$ with $\chi \in \Sigma_{\langle\rangle}$. The function Θ_S takes an FFL as the second argument while θ_S takes an event χ. The following lemma shows that the function Θ_S can simulate a transformation τ_S for an input stream $\lfloor f \rfloor$.

Lemma 2. *Let $S = (Q, \Sigma, \Delta, in, R)$ be an xsp. Then*

$$\tau_S(\lfloor f \rfloor) = \Theta_S(in(\varepsilon, \ldots, \varepsilon), (f, \#)) \tag{5}$$

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

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

$$\theta_S(in(\varepsilon, \ldots, \varepsilon), \lfloor f \rfloor \$) = \Theta_S(in(\varepsilon, \ldots, \varepsilon), (f, \#)). \tag{6}$$

We show an equation

$$\theta_S(e, gen(l)) = \Theta_S(e, l) \tag{7}$$

for $e \in Tmp_S$ and $l \in G((f, \#))$, which is more general than (6) owing to LEMMA 1. The equation (7) is proved by induction on the cardinality $\sharp G(l)$ of $G(l)$. If $\sharp G(l) = 1$, i.e., $l = (\epsilon, \#)$, then both sides are the same, that is $\langle e, \$ \rangle$. If $\sharp G(l) > 1$, then $gen(l) = \chi gen(l')$ with $\chi \in \Sigma_{\langle\rangle}$. We have $\sharp G(l') = \sharp G(l) - 1$ since $l \notin G(l')$ from the definition of G. Therefore we obtain

$$\begin{aligned}
\theta_S(e, gen(l)) &= \theta_S(e, \chi gen(l')) \\
&= \theta_S(\langle e, \chi \rangle, gen(l')) \\
&= \Theta_S(\langle e, \chi \rangle, l') \\
&= \Theta_S(e, l)
\end{aligned}$$

from the induction hypothesis, the definitions of θ_S and Θ_S. Hence the equation (7) holds for $e \in Tmp_S$ and $l \in G((f, \#))$. $\qquad\square$

Next we define the function $\mathcal{I} : Tmp_{\mathcal{SP}(M)} \to \mathcal{F}_\Delta$ for an mft $M = (Q, \Sigma, \Delta, in, R)$. The function $\mathcal{I}$ translates a temporary expression into a corresponding output forest and has an *invariant 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 4. Since all temporary expressions for $\mathcal{SP}(M)$ are *well-defined* from the definition of $\mathcal{SP}(M)$ in the sense that they range over

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

we define the function $\mathcal{I}$ by

$$\begin{aligned}
&\mathcal{I}(q[i](e_1, \ldots, e_n), l) = [\![q]\!](l[i], \mathcal{I}(e_1, l), \ldots, \mathcal{I}(e_n, l)), \\
&\mathcal{I}(\varepsilon, l) = \epsilon, \qquad\qquad\qquad\qquad \mathcal{I}(\texttt{<}\delta\texttt{>}e\texttt{</}\delta\texttt{>}, l) = \delta\langle \mathcal{I}(e, l) \rangle, \\
&\mathcal{I}(e\,e', l) = \mathcal{I}(e, l)\ \mathcal{I}(e', l).
\end{aligned}$$

To prove the invariant property of $\mathcal{I}$, we show the following lemma which correlates the right-hand sides of rules of M and $\mathcal{SP}(M)$.

Lemma 3. *Let $M = (Q, \Sigma, \Delta, in, R)$ be an mft, rhs be a right-hand side of a rule in R_q with $q \in Q$ and $rank(q) = n + 1$, l be an FFL for $f \in \mathcal{F}_\Sigma$ and e_j with $j = 1, \ldots, n$ be a (well-formed) temporary expression for $\mathcal{SP}(M)$. For a binding ρ given by $\rho(x_i) = l[i]$ $(i = 1, 2)$ and $\rho(y_j) = \mathcal{I}(e_j, l)$ $(j = 1, \ldots, n)$, we have*

$$[\![rhs]\!]_\rho = \mathcal{I}(\mathcal{A}(rhs)[y_j := e_j]_{j=1,\ldots,n}, l) \tag{8}$$

where $\mathcal{A}$ is as given in DEFINITION 8. In particular, when rhs does not contain any pattern $q(x_i, \ldots)$, we have

$$\lfloor [\![rhs]\!]_\rho \rfloor = \mathcal{A}(rhs)[y_j := \lfloor \mathcal{I}(e_j, l) \rfloor]_{j=1,\ldots,n}. \tag{9}$$

Proof. The statement is proved by induction on the structure of *rhs*. In the rest of this proof, we write just $\rho'(\xi)$ for $\xi[y_j := e_j]_{j=1,\ldots,n}$. Thus we should prove the equation $[\![rhs]\!]_\rho = \mathcal{I}(\rho'(\mathcal{A}(rhs)), l)$. If $rhs = \epsilon$, then both sides of (8) are the same, that is ϵ. If $rhs = \delta\langle rhs'\rangle$, then the right-hand side of (8) is $[\![\delta\langle rhs'\rangle]\!]_\rho = \delta\langle[\![rhs']\!]_\rho\rangle$. From the definitions of $\mathcal{A}$ and $\mathcal{I}$, the left-hand side of (8) is

$$\mathcal{I}(\rho'(\mathcal{A}(\delta\langle rhs'\rangle)), l)$$
$$= \mathcal{I}(\texttt{<}\delta\texttt{>}\rho'(\mathcal{A}(rhs'))\texttt{</}\delta\texttt{>}, l)$$
$$= \delta\langle\mathcal{I}(\rho'(\mathcal{A}(rhs'))), l)\rangle$$

which is equal to $\delta\langle[\![rhs']\!]_\rho\rangle$ from the induction hypothesis. If $rhs = y_j$, then both sides of (8) are the same, that is $\mathcal{I}(e_j, l)$. If $rhs = rhs_1 rhs_2$, then both sides of (8) are the same from the definitions of $\mathcal{A}$ and $\mathcal{I}$ and the induction hypothesis. If $rhs = q(x_i, rhs_1, \ldots, rhs_{n'})$, then the right-hand side of (8) is

$$[\![q(x_i, rhs_1, \ldots, rhs_{n'})]\!]_\rho$$
$$= [\![q]\!](l[i], [\![rhs_1]\!]_\rho, \ldots, [\![rhs_{n'}]\!]_\rho)$$
$$= [\![q]\!](l[i], \mathcal{I}(\rho'(\mathcal{A}(rhs_1)), l), \ldots, \mathcal{I}(\rho'(\mathcal{A}(rhs_{n'})), l))$$

from the definition of $[\![\_\_]\!]_\rho$ and the induction hypothesis. The left-hand side of (8) is

$$\mathcal{I}(\rho'(\mathcal{A}(q(x_i, rhs_1, \ldots, rhs_{n'}))), l)$$
$$= \mathcal{I}(\rho'(q[i](\mathcal{A}(rhs_1), \ldots, \mathcal{A}(rhs_{n'}))), l)$$
$$= \mathcal{I}(q[i](\rho'(\mathcal{A}(rhs_1)), \ldots, \rho'(\mathcal{A}(rhs_{n'}))), l)$$
$$= [\![q]\!](l[i], \mathcal{I}(\rho'(\mathcal{A}(rhs_1)), l), \ldots, \mathcal{I}(\rho'(\mathcal{A}(rhs_{n'})), l))$$

from the definitions of $\mathcal{A}$ and $\mathcal{I}$. Hence we have the equation (8).

In the particular case that *rhs* does not contain any pattern $q(x_i, \ldots)$, the equation (9) is shown by the induction of the structure of *rhs*. When $rhs = \epsilon$, the both sides are the same, that is ε. When $rhs = \delta\langle rhs'\rangle$, the left-hand side of (9) is $\texttt{<}\delta\texttt{>}\lfloor[\![rhs']\!]_\rho\rfloor\texttt{</}\delta\texttt{>}$ and the right-hand side is $\texttt{<}\delta\texttt{>}\rho'(\mathcal{A}(rhs'))\texttt{</}\delta\texttt{>}$. These are equal from the induction hypothesis. When $rhs = y_j$, the both sides are the same, that is $\lfloor\mathcal{I}(e_j, l)\rfloor$. When $rhs = rhs_1 rhs_2$, the left-hand side of (9) is $\lfloor[\![rhs_1]\!]_\rho\rfloor\lfloor[\![rhs_2]\!]_\rho\rfloor$ and the right-hand side is $\rho'(\mathcal{A}(rhs_1))\rho'(\mathcal{A}(rhs_2))$. These are equal from the induction hypothesis. Therefore (9) holds. $\square$

Then we show the following lemma with respect to the invariant property of $\mathcal{I}$.

Lemma 4. *Let* $M = (Q, \Sigma, \Delta, in, R)$ *be an mft,* e *be a (well-defined) temporary expression for an xsp* $\mathcal{SP}(M)$ *and* l *be an FFL for a forest* $f \in \mathcal{F}_\Sigma$. *For* $gen(l) = \chi gen(l')$, *we have*

$$\mathcal{I}(e, l) = \mathcal{I}(\langle\!| e, \chi|\!\rangle, l'). \tag{10}$$

Proof. We prove the statement by induction on the cardinality $\sharp G(l)$ of $G(l)$. We start with the case of $\sharp G(l) = 2$ since the equation $gen(l) = \chi gen(l')$ is assumed.

In the case of $\sharp G(l) = 2$, we have $l = (\epsilon, \sigma) :: (\epsilon, \#)$ with $gen(l) = \texttt{</}\sigma\texttt{>}gen((\epsilon, \#))$. We have to show an equation

$$\mathcal{I}(e, (\epsilon, \sigma) :: (\epsilon, \#)) = \mathcal{I}(\langle\!| e, \texttt{</}\sigma\texttt{>}|\!\rangle, (\epsilon, \#)). \tag{11}$$

It is proved by induction on the structure of the (well-defined) temporary expression e. We only show the case of $e = q[i](e_1, \ldots, e_n)$ with $q \in Q$ and $i \in \mathbb{N}^+$ that is the most complicated one in the induction. The other cases can be shown from the definitions of $\mathcal{I}$ and $\langle\!|\_\_, \_\_|\!\rangle$.

If $e = q[1](e_1, \ldots, e_n)$ with $q \in Q$, then we have

$$\mathcal{I}(q[1](e_1, \ldots, e_n), l)$$
$$= [\![q]\!](l[1], \mathcal{I}(e_1, l), \ldots, \mathcal{I}(e_n, l))$$
$$= [\![q]\!](\epsilon, \mathcal{I}(\langle\!| e_1, \texttt{</}\sigma\texttt{>}|\!\rangle, l'), \ldots, \mathcal{I}(\langle\!| e_n, \texttt{</}\sigma\texttt{>}|\!\rangle, l'))$$
$$= [\![rhs^{q,\epsilon}]\!]_\rho$$
$$= \mathcal{I}(\mathcal{A}(rhs^{q,\epsilon})[y_j := \langle\!| e_j, \texttt{</}\sigma\texttt{>}|\!\rangle]_{j=1,\ldots,n}, l')$$
$$= \mathcal{I}(\langle\!| q[1](e_1, \ldots, e_n), \texttt{</}\sigma\texttt{>}|\!\rangle, l')$$

from the definitions of $\mathcal{I}$, $[\![q]\!]$ and $(\!|\_\_,\_\_|\!)$, $l = (\epsilon, \sigma) :: (\epsilon, \#)$, the induction hypothesis and LEMMA 3 where ρ is a binding defined by $\rho(y_j) = \mathcal{I}((\!|e_j, \texttt{</σ>}|\!), l')$ for $j = 1, \ldots, n$. Hence the equation (11) holds.

If $e = q[i](e_1, \ldots, e_n)$ with $q \in Q$ and $i > 1$, then we have

$$\mathcal{I}(q[i](e_1, \ldots, e_n), l)$$
$$= [\![q]\!](\epsilon, \mathcal{I}((\!|e_1, \texttt{</σ>}|\!), l'), \ldots, \mathcal{I}((\!|e_n, \texttt{</σ>}|\!), l'))$$

in a way similar to the case of $e = q[1](e_1, \ldots, e_n)$. The right-hand side of (11) is

$$\mathcal{I}((\!|q[i](e_1, \ldots, e_n), \texttt{</σ>}|\!), l')$$
$$= \mathcal{I}(q[i-1]((\!|e_1, \texttt{</σ>}|\!), \ldots, (\!|e_n, \texttt{</σ>}|\!)), l')$$
$$= [\![q]\!](l'[i-1], \mathcal{I}((\!|e_1, \texttt{</σ>}|\!), l'), \ldots, \mathcal{I}((\!|e_n, \texttt{</σ>}|\!), l'))$$

from the definitions $(\!|\_\_,\_\_|\!)$ and $\mathcal{I}$, a $(q, \texttt{</$\sigma$>})$-rule of $\mathcal{SP}(M)$ The equation (11) holds from $l'[i-1] = \epsilon$.

Next we show the case of $\sharp G(l) > 2$ in (10). Assume that $gen(l) = \chi gen(l')$. The equation (10) is proved by induction on the structure of the (well-defined) temporary expression e. Again we only show the case of $e = q[i](e_1, \ldots, e_n)$ with $q \in Q$ and $i \in \mathbb{N}^+$ since the other cases are easily shown from the definitions of $\mathcal{I}$ and $(\!|\_\_,\_\_|\!)$.

If $e = q[1](e_1, \ldots, e_n)$ with $q \in Q$, then the left-hand side of (10) is equal to $[\![q]\!](l[1], \mathcal{I}(e_1, l), \ldots, \mathcal{I}(e_n, l))$ by the definition of $\mathcal{I}$. There are three possibilities for the form of l, $(\sigma\langle f_1 \rangle f_2, \sigma') :: l''$, $(\sigma\langle f_1 \rangle f_2, \#)$ and $(\epsilon, \sigma) :: l''$. In the first two cases, we have $l[1] = \sigma\langle f_1 \rangle f_2$ and $gen(l) = \texttt{<$\sigma$>} gen(l')$ with $l'[i] = f_i$ ($i = 1, 2$). The left-hand side of (10) is

$$[\![q]\!](l[1], \mathcal{I}(e_1, l), \ldots, \mathcal{I}(e_n, l))$$
$$= [\![q]\!](\sigma\langle f_1 \rangle f_2, \mathcal{I}((\!|e_1, \texttt{<σ>}|\!), l'), \ldots, \mathcal{I}((\!|e_n, \texttt{<σ>}|\!), l'))$$
$$= [\![rhs^{q,\sigma}]\!]_\rho$$
$$= \mathcal{I}(\mathcal{A}(rhs^{q,\sigma})[y_j := (\!|e_j, \texttt{<σ>}|\!)]_{j=1,\ldots,n}, l')$$
$$= \mathcal{I}((\!|q[1](e_1, \ldots, e_n), \texttt{<σ>}|\!), l')$$

from the definitions of $[\![\_\_]\!]$ and $(\!|\_\_,\_\_|\!)$, the induction hypothesis and LEMMA 3 where ρ is a binding defined by $\rho(x_i) = f_i$ for $i = 1, 2$ and $\rho(y_j) = \mathcal{I}((\!|e_j, \texttt{<σ>}|\!), l')$ for $j = 1, \ldots, n$. Hence we have the equation (10) in this case of l. In the case of $l = (\epsilon, \sigma) :: l''$, the equation (10) can be shown in a way similar to the case of $G(l) = 2$ and $e = q[1](e_1, \ldots, e_n)$ because $l[1] = \epsilon$.

If $e = q[i](e_1, \ldots, e_n)$ with $q \in Q$ and $i > 1$, then the left-hand side of (10) is equal to $[\![q]\!](l[i], \mathcal{I}(e_1, l), \ldots, \mathcal{I}(e_n, l))$ by the definition of $\mathcal{I}$. There are three possibilities for the form of l, $(\sigma\langle f_1 \rangle f_2, \sigma') :: l''$, $(\sigma\langle f_1 \rangle f_2, \#)$ and $(\epsilon, \sigma) :: l''$. In the first two cases, we have $gen(l) = \texttt{<$\sigma$>} gen(l')$ with $l[i] = l'[i+1]$ and $G(l) = G(l') + 1$. Then we obtain

$$[\![q]\!](l[i], \mathcal{I}(e_1, l), \ldots, \mathcal{I}(e_n, l))$$
$$= [\![q]\!](l'[i+1], \mathcal{I}((\!|e_1, \texttt{<σ>}|\!), l'), \ldots, \mathcal{I}((\!|e_n, \texttt{<σ>}|\!), l'))$$
$$= \mathcal{I}(q[i+1]((\!|e_1, \texttt{<σ>}|\!), \ldots, (\!|e_n, \texttt{<σ>}|\!)), l')$$
$$= \mathcal{I}((\!|q[i](e_1, \ldots, e_n), \texttt{<σ>}|\!), l')$$

from the induction hypotheses, the definition of $(\!|\_\_,\_\_|\!)$ and a rule in an xsp $\mathcal{SP}(M)$. Hence the equation (10) holds in this case of l. In the case of $(\epsilon, \sigma) :: l''$, we have $gen(l) = \texttt{</$\sigma$>} gen(l')$ with $l[i] = l'[i-1]$ and $G(l) = G(l') - 1$. Then the equation (10) is shown in a way similar to the previous case. $\square$

The following lemma shows the relation of Θ_S and $\mathcal{I}$.

Lemma 5. *Let $M = (Q, \Sigma, \Delta, in, R)$ be an mft and $l \in G((f, \#))$ be an FFL for $f \in \mathcal{F}_\Sigma$. For any temporary expression $e \in \mathcal{SPM}$, we have*

$$\lfloor \mathcal{I}(e, l) \rfloor = \Theta_{\mathcal{SP}(M)}(e, l). \tag{12}$$

Proof. For the computation chain of $\Theta_S(e_0, (f, \#)) = \cdots = \Theta_S(e, (\epsilon, \#))$, we have $\mathcal{I}(e_0, (f, \#)) = \cdots = \mathcal{I}(e, (\epsilon, \#))$ due to LEMMA 4. Hence it is enough to show

$$\lfloor \mathcal{I}(e, (\epsilon, \#)) \rfloor = \Theta_S(e, (\epsilon, \#)) (= \langle\!| e, \$ |\!\rangle) \tag{13}$$

It is proved by induction on e. We only show the case of $e = q[i](e_1, \ldots, e_n)$ with $q \in Q$ and $i \in \mathbb{N}^+$ because the other cases can be shown from the definitions of $\Theta_{\mathcal{SP}(M)}$, $\lfloor \_\! \rfloor$ and $\mathcal{I}$. Since $((\epsilon, \#))[i] = \epsilon$ for any $i \in \mathbb{N}^+$, the left-hand side of (13) is

$$\begin{aligned}
&\lfloor \mathcal{I}(q[i](e_1, \ldots, e_n), (\epsilon, \#)) \rfloor \\
&= \lfloor [\![q]\!](\epsilon, \mathcal{I}(e_1, (\epsilon, \#)), \ldots, \mathcal{I}(e_n, (\epsilon, \#))) \rfloor \\
&= \lfloor [\![rhs^{q,\epsilon}]\!]_\rho \rfloor
\end{aligned}$$

with $\rho(y_j) = \mathcal{I}(e_j, (\epsilon, \#))$ $(j = 1, \ldots, n)$ from the definitions of $\mathcal{I}$ and $[\![\_\!]\!]$. The right-hand side of (13) is

$$\begin{aligned}
&\langle\!| q[i](e_1, \ldots, e_n), \$ |\!\rangle \\
&= \mathcal{A}(rhs^{q,\epsilon})[y_j := \langle\!| e_j, \$ |\!\rangle]_{j=1,\ldots,n} \\
&= \mathcal{A}(rhs^{q,\epsilon})[y_j := \lfloor \mathcal{I}(e_j, (\epsilon, \#)) \rfloor]_{j=1,\ldots,n}
\end{aligned}$$

from the definitions of $\langle\!| \_\! |\!\rangle$ and the induction hypothesis. Then these are equal from (9) in LEMMA 3 since no pattern $q'(x_i, \ldots)$ occurs in $rhs^{q,\epsilon}$. $\qquad\square$

Now we prove THEOREM 1. Let $M = (Q, \Sigma, \Delta, in, R)$ be an mft and $S = \mathcal{SP}(M)$ be an xsp. For a forest $f \in \mathcal{F}_\Sigma$, we have

$$\begin{aligned}
\tau_S(\lfloor f \rfloor) &= \Theta_S(in[1](\varepsilon, \ldots, \varepsilon), (f, \#)) \\
&= \lfloor \mathcal{I}(in[1](\varepsilon, \ldots, \varepsilon), (f, \#)) \rfloor \\
&= \lfloor [\![in]\!](f, \epsilon, \ldots, \epsilon) \rfloor \\
&= \lfloor \tau_M(f) \rfloor
\end{aligned}$$

from LEMMA 2, LEMMA 5 and the definitions of $\mathcal{I}$ and τ_M. Therefore THEOREM 1 has been proved.