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, 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,

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

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 340–356, 2006.
© Springer-Verlag Berlin Heidelberg 2006

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

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

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><c></c><d></d><e> </e></r><f></f>. 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[1]. 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.

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.

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