A Functional Language for Hyperstreaming XSLT *

Pavel Labath
Commenius University, Bratislava, Slovakia
labath@dcs.fmph.uniba.sk

Joachim Niehren
INRIA Lille

Abstract

The problem of how to transform large data trees received on streams with a much smaller memory is still an open challenge despite of a decade of research on XML. Therefore, the current approach of the XSLT working of the W3C is to provide streaming support only for a smaller fragment of XSLT 3.0. This has the drawback that many existing XSLT programs need to be rewritten in order to become executable on XML streams, while many others cannot be rewritten at all, since defining nonstreamble transformations. In this paper, we propose a new hyperstreaming approach that does not require any a priori restrictions. The model of hyperstreaming generalizes on the model of streaming by adding shredding operations for the output stream, so that its parts may be plugged together later on. Many transformations such as flips of document pairs are hyperstreamable but not streamable. We then present the functional language X-Fun for defining transformations between XML data trees, while providing shredding instructions. X-Fun can be understood as an extension of Frisch's XSTREAM language with output shredding, while pattern matching is replaced by tree navigation with XPATH expressions. We provide a compiler from XSLT into a fragment of X-Fun, which can be considered as the core of XSLT. We then present a hyperstreaming algorithm for evaluating X-Fun programs which combines a recent XPath evaluator with a traditional functional programming engine. We have implemented a hyperstreaming evaluator for X-Fun and thus for XSLT and compare it experimentally with SAXON's XSLT implementation. It turns out that many XSLT programs become hyperstreamable with good efficiency and without any manual rewriting.

Note from February 2014: The first contribution of this report is the definition of X-Fun. This definition is outdated meanwhile, since X-Fun evolved at lot. See our follow-up paper at http://hal.inria.fr/hal-00954692. The second contribution on hyperstreaming is not described there though.

1. Introduction

The problem of how to transform large data trees received on streams with a much smaller memory is still an open challenge despite of a decade of research on XML [1, 10, 13, 15, 17, 19, 22, 24, 25]. The most used programming language for defining transformations of data trees are XSLT, JAVASCRIPT for JSON [5], NOSQL languages [4], CDUCE [3], beside many others.

Memory efficiency is essential for processing data trees of several giga bytes that do not fit into main memory, while time efficiency is important too. However, some transformations cannot be streamed with a bounded memory. An example is the insertion of table of contents into a book:

$$book(t) \Rightarrow book(toc(t), t)$$

A streaming implementation of this transformation has to read the content t of the book from the input stream while computing the table of contents $toc(t)$, but it has also to buffer t for output once the output of $toc(t)$ was completed. Clearly, this cannot be done with a buffer of bounded size since the size of t may be unbounded.

Therefore, the current approach of the XSLT working group of the W3C is to restrict the streaming support for XSLT 3.0 to a smaller fragment [14], as followed by the SAXON implementation [17]. For illustration, we show how to define the above transformation by an XSLT like program in a functional language with XPATH navigation:

```
addtoc(x) =
 case x of [self::book]
 then tree(book,toc(x)
                for x' in x/child::* do subtree(x'))
toc(x) =  ...
```

The corresponding XSLT 3.0 program does not belong to the streamable fragment, since `fork`-operators are required around sequences for enabling parallel evaluation, such as for the sequence:

```
toc(x) for x' in x/child::* do subtree(x')
```

Indeed, only a very small class of XSLT 3.0 programs without `fork`-operators is streamable [8, 18]. So one might wonder, why the `fork`-operator has not yet been implemented in Saxon. In the above example, the problem is that the introduction of a `fork`-operator does not solve the streamability problem, since the content of the book `t` must still be buffered until its table of contents `toc(t)` got output completely.

In this paper, we propose a new hyperstreaming model in order to solve the above problem. The idea is to shred the output stream into fragments that can be produced independently in parallel, and plugged together later on. In the above example, for instance, we would like to output `toc(t)` to its own stream shred, in order to unblock the output of the remainder of the book, so that no buffering becomes necessary any more. For this purpose, will will replace `toc(x)` by `shred(toc(x))` in the above program. Even flips of document pairs, such as flipping $book(toc, t)$ to $book(t, toc)$ can be computed in a hyperstreaming mode:

```
flip(x) = case x of [self::book] then tree(book,
    shred(for x' in x/child[2] do subtree(x'))
    for x' in x/child[1] do subtree(x'))
```

* This research was supported in part by the grant VEGA 1/0979/12

It should also be notice that output shredding is similar to using futures for parallelizing the output production of functional programs [11, 16, 23]. The difference here is that the output is written to an external stream, rather then being stored in main memory.

We present X-Fun, a functional core language for transforming data trees into shredded data trees, as needed to support hyperstreaming. The input stream will contain a nested word [19] obtained by linearization a data tree, in any of the common formats (XML, JSON, etc.) while the output will be a shredded nested word written on a collection of output streams. X-Fun can be derived from the functional language XSTREAM [12] by adding shred instructions and XPATH navigation (instead of pattern matching). Alternatively, X-Fun can be obtained from walking macro tree transducers [21], by shred instructions and a composition operator making it Turing complete. We then show how to compile a large subset of XSLT to X-Fun, while introducing `shred` operations systematically for reaching hyperstreamability. Navigation with XPATH is essential here, while raising quite some challenges to streaming evaluation compared to simpler pattern matching. Indeed, a fragment of X-Fun without nested loops can be understood as the core language of XSLT. With nested loops, many queries from XQUERY can be expressed in X-Fun too.

The second contribution is a hyperstreaming algorithm for evaluating X-Fun transformations. Our algorithm extracts all XPATH expressions from a given X-Fun program, and runs an XPATH streaming machines for all of them, in order to compute their answer sets incrementally and in parallel to the input stream. Furthermore, our algorithm runs a functional engine in parallel, which computes a shredded data tree from the input stream, and outputs it as much as possible to the shredded output stream. The synchronization control between the functional engine and the many XPATH streaming machines is done by variables, as supported by the FXP language [6, 13]. The XPATH streaming machines of FXP 1.1 tool produce answer nodes in an almost earliest manner, so that they buffer only a minimal number of answer candidates at any time point. In contrast to the hardness of earliest query answering for XPATH [2, 13], the approximation done by FXP can be computed highly efficiently and is tight in practice. We refer to [6] for a large scale experimental comparison of FXP to the many alternative XPATH streaming tools in the literature. It shows that FXP outperforms most previous XPATH streaming tools in time and memory efficiency, while having the largest coverage. The main missing feature of FXP 1.1 are XPATH joins which would be essential for reducing XQUERY streaming [10] to X-Fun.

We have implemented the evaluation algorithm for X-Fun based on the FXP 1.1 tool in JAVA, as well as the compiler from XSLT to X-Fun. In combination, we obtain a hyperstreaming evaluator for XSLT. Its already proves good efficiency, even though we did not yet spend much time with optimization. Due to hyperstreaming, the memory efficiency is excellent in many examples where none of the existing streaming approaches (XSTREAM or SAXON) could be applied. For small documents, the time efficiency is usually a little lower than for SAXON in-memory processing, for instance by a factor of 7 for converting an address book from XML to HTML. Since we assumed address books with unordered address fields, and HTML outputs with rigidly ordered address fields, such a conversion can be written within the streamable fragment of XSLT 3.0 only when usuing `fork`-operators, which are not yet supported by Saxon streaming though, nor can it be written naturally in XSTREAM. We have also compared the efficiency or our XSLT tool with Saxon's streaming XSLT 3.0 implementation. On a transformation that deletes a field in an address book, our implementation runs are only by a factor of 2-3 slower, so it is already competitive without optimizations.

Outline. After some premiminaries in Section 2, we will introduce X-Fun in Section 3 and compare its expressiveness to walking macro tree transducers and XSTREAM. Section 4 sketches our compiler from XSLT to X-Fun. Section 5 presents an algorithm for hyperstreaming evaluation of X-Fun. Section 6 discusses our implementation and presents first experimental results.

2. Preliminaries

Before we define the X-Fun language, we first introduce some preliminary definitions and fix the notation. In the rest of the paper, Σ and Δ will refer to a finite set of tags and internal letters, respectively.

Data Trees and Nested Words. Data trees are ordered unranked trees, which may contain textual contents in leaf nodes, and data hedges are sequences of data trees. More formally, the set of *data hedges* $\mathcal{H}$ over Σ and Δ is the least set that contains all strings in Δ^*, all sequences $(h_1, \ldots, h_n)$ in $\mathcal{H}^*$, and all pairs $a(h)$ where $a \in \Sigma$ and $h \in \mathcal{H}$. A *data tree* t is a data hedge of the form $a(h)$. Given that every hedge is a sequence of data trees, we can define the concatenation of two hedges $h \cdot h'$ as the concatenation of these sequences. The set of nodes of a data tree can be defined as usual, as well as the relations *ch* and *nextsibling*. From this we can define the relations *descendant* = ch^+, *ancestor* = $descendant^{-1}$, and *followingsibling* = $nextsibling^+$. The text of a data tree is the concatenation of all texts in its text leafs:

$$\begin{aligned} &text(w) =_{def} w \qquad \text{for } w \in \Delta^* \\ &text(a(h)) =_{def} text(h) \\ &text((h_1, \ldots, h_n)) =_{def} text(h_1) \cdot \ldots \cdot text(h_n) \end{aligned}$$

The subtree of a hedge h at node v is denoted by $h_{|v}$. The label $lab(t, v)$ of a node v in hedge h is the root label of $t_{|v}$. I.e., if this subtree has the form $a(h)$ then the label is a and in case of text leafs, it is the constant $text$.

Sometimes, we will have to transform data hedges into data trees. This can be done by fixing a symbol $fakeroot \in \Sigma$ and adding an artificial root node with this label:

$$to\_tree(h) =_{def} fakeroot(h)$$

In order to write a data tree to a stream, we have to linearize it into a nested word. A *nested word* over Σ and Δ is word consisting of opening tags <*a*>, closing tags </*a*> where $a \in \Sigma$ and internal letters from Δ, such that the sequence is balanced. I.e., there exists a closing tag for every opening tag and the tags are well-nested. Every data hedge h can be linearized in a left-first depth-first manner into a unique nested word $lin(h)$.

$$\begin{aligned} &lin(w) =_{def} \texttt{<}text\texttt{>}w\texttt{</}text\texttt{>} \\ &lin(a(h)) =_{def} \texttt{<}a\texttt{>}lin(h)\texttt{</}a\texttt{>} \\ &lin((h_1, \ldots, h_n)) =_{def} lin(h_1) \cdot \ldots \cdot lin(h_n) \end{aligned}$$

For example, the data tree $book(author(\text{“}Ullman\text{”}), \ldots)$ is linearized to the XML document:

`<book><author><text>Ullman</text></author>...</book>`.

The positions of nested word are called *events*. There are three kind of events: internal, opening and closing, corresponding to internal letters, opening and closing tags, respectively. Also note that every node in a data hedge h labeled in Σ corresponds to exactly two events in $lin(h)$: an opening event and the corresponding close event. Every text node corresponds to an opening event <$text$> a sequence of events for all letters of the text, and a closing event </$text$>.

XML Data Model and XPath. The XML data model defines data trees with a typed signature Σ. There are five different types: docu-

ment, element, attribute, comment, and processing instruction. Text nodes of XML are mapped to text leafs in our data trees. In order to deal with attribute nodes in examples, we will use labels such as `href_attr` where the type is indicated in the labels name. The XML data model imposes some additional restrictions on data trees with the above types, such as for instance, that attribute children must proceed all other children.

XPATH is a navigational language for selecting nodes in XML data trees. For instance, one can use the XPATH expression:

```
child::a/descendant::b[ancestor::c and
                       contains(text(),"Jemal")]
```

for navigating from some start node in an XML data tree to some child labeled by a, and then to some descendant labeled by b that is the target node, under the condition, that this target node has an ancestor labeled by c and that its text contains "Jemal". The basic relations of XPATH expressions are the axis *ch*, *followingsibling*, *descendant*, *ancestor*, etc. The axes of XPATH are basically the relations of data trees named alike, except that the XPATH axes are restricted by types. Attributes, for instance, cannot be accessed by child axis, but there is a special operator for accessing them, for instance, `@href` for accessing `href`-attributes.

The semantics of an XPATH expression P on a data tree t is a set $[\![P]\!]_t^{path}$ of pairs of nodes of t. An XPATH filter is a sub-expression such as: `[ancestor::c and contains(text(),"Jemal")]` The semantics of XPATH filters F on a data tree t is a set $[\![F]\!]_t^{filter}$ of nodes of t that satisfy the filter. It should be noticed that XPATH like expressions or filters can also be defined for general data trees not satisfying the XML data model. The main difference will then be in the typing information.

3. X-Fun

We introduce functional core language X-Fun for transforming data trees into shredded data hedges and compare its expressivness to walking macros tree transducers and XSTREAM.

3.1 Shredded Data Hedges

We first lift the notion of data hedges to shredded data hedge. In order to do so, we assume a infinite set of hedge variables ranged over by y.

Definition 1. A *shredded data hedge* over Σ and Δ is a pair $(h_0, (y_i, h_i)_{i=1}^n)$ where h_i is a data hedge over Σ and $\Delta \uplus \{y_{i+1}, \ldots, y_n\}$ and y_i are pairwise distinct hedge variables.

Let $h = (h_0, (y_i, h_i)_{i=1}^n)$ and $h' = (h'_0, (y_i, h_i)_{i=n+1}^m)$ two shredded data hedges. The following unshredding operation plugs the parts of a shedded hedge together, resulting in an unshredded hedge, by substituting hedge variables by their values:

$$unshred(h) =_{def} h_0[h_1/y_1] \ldots [h_n/y_n]$$

We also need to lift the constructors for shredded trees and concatenation operation on shedded hedges. These can be defined as follows where $a \in \Sigma$:

$$a(h) =_{def} (a(h_0), (y_i, h_i)_{i=1}^n)$$
$$h \cdot h' = (h_0 \cdot h'_0,\ (y_i, h_i)_{i=1\ldots m})$$

Furthermore, we need a constructor of hedges with more shreds $\mathbf{shred}\ y_{n+1} = h_{n+1}\ \mathbf{in}\ h$, where $h = (h_0, (y_i, h_1)_{i=1}^n)$ is a shredded hedge with signature $\Sigma \cup \{y_{n+1}\}$ for some fresh variable y_{n+1}:

$$\mathbf{shred}\ y_{n+1} = h_{n+1}\ \mathbf{in}\ h\ =_{def} (h_0, (y_i, h_i)_{i=1}^{n+1})$$

We next need to lift the linearization to shredded data hedges. In order to do so we need the notion of a shredded nested word.

Program	A	::=	y	(hedge variable)
		|	$subtree(N)$	
		|	$tree(L, A)$	
		|	T	
		|	$A_1 \cdots A_n$	
		|	$\mathbf{let}\ D\ \mathbf{in}\ A$	
		|	$q(N, A_1, \ldots, A_n)$	
		|	$\mathbf{for}\ x\ \mathbf{in}\ N/P\ \mathbf{do}\ A$	$(\mathcal{V}_{node}(A) \subseteq \{x\})$
		|	$\mathbf{case}\ N\ \mathbf{of}\ F_1 \Rightarrow A_1\ \ldots\ F_n \Rightarrow A_n$	
		|	$\mathbf{compose}\ x : A\ \mathbf{in}\ A'$	
		|	$shred(A, T)$	
Definitions	D	::=	$y = A$	
		|	$q(x, y_1, \ldots, y_n) = A$	
		|	$x = N$	
		|	$D_1 \& D_2$	
Node	N	::=	x	(node variable)
		|	v	(node identifier)
Text	T	::=	$text(N)$	
		|	"w"	$(w \in \Delta^*)$
		|	$T_1 \ldots T_n$	
Label	L	::=	$lab(N)$	
		|	a	$(a \in \Sigma)$
Path	P	::=	path expressions over Σ and Δ	
Filter	F	::=	filter expression over Σ and Δ	

Figure 1. Syntax of X-Fun.

Definition 2. A *shredded nested word* over Σ and Δ is a pair $(w_0, (y_i, w_i)_{i=1}^n)$ where w_i is a nested word over Σ and $\Delta \uplus \{y_{i+1}, \ldots, y_n\}$, and all y_i are pairwise distinct.

The linearization of a shredded data tree into a shredded nested word over the same alphabets can now be defined as follows:

$$lin(h_0, (y_i, h_i)_{i=1}^n) =_{def} (lin(h_0), (y_i, lin(h_i))_{i=1}^n)$$

A shredded nested word can be written into a collection of files, one for each of its $n+1$ fragments. In order to be able to assign names to such files, we will assume that for every file name $w \in \Delta^*$ there exists a hedge variable y_w in the set of our hedge variables.

3.2 Syntax

We assume an infinite set of node variables ranged over by x, and a ranked alphabet of function names ranged over by q. Whenever we will use an application $q(N, A_1, \ldots, A_n)$ we will assume that n is the rank of q. We also assume that we are given a set of node identifiers, which contains the identfier *root* for the root node of a data tree.

The abstract syntax of X-Fun over these sets is then in Figure 1. An closed X-Fun program, as written by a user, may not contain free variables (for hedges, nodes, or functions) and no other node identifier than *root*. All other node identifier will be computed by path navigation over the innput tree at evaluation time. The semantics of a closed X-Fun program will be a function from data trees to shredded data hedges. For open X-Fun programs, the semantics will be defined relatively to a closing environment that

maps hedge variables to shredded hedges, node variables to node identifiers, and function variables to function definitions.

The main idea of the evaluation process is that an X-Fun program will navigate up, down, left, and right on the input tree, while constructing the output shredded hedge in a top-down manner. For this we assume as a parameter of the X-Fun language a set of path expressions ranged over by P and a set of filters ranged over by F, with evaluation functions, such that $[\![P]\!]^{path}_t$ is a subset of pairs of nodes of t, and $[\![F]\!]^{filter}_t$ is a subset of nodes of t.

We now consider the different types of programs one by one. A program y returns the shredded hedge assigned to y by the environment. A program $subtree(N)$ returns the subtree of of the input tree rooted by the node designed by N. A program T returns a text leaf containing the text designed by T. A program $A_1 \dots A_n$ returns the concatenation of the shredded hedges computed by A_1, …, A_n. A program $\mathbf{let}\ D\ \mathbf{in}\ A$ extends the environment by the definitions in D, and the returns the output of A in the extended environment. A program $\mathbf{for}\ x\ \mathbf{in}\ N/P\ \mathbf{do}\ A$ computes the set of all nodes v that can be reached over P from the node of the input tree denoted by N, computes for all these nodes the output of A, and returns these output in the order of the nodes in the input tree. We restrict loops such that x may be the only free node variable occuring in body A, i.e. $\mathcal{V}_{node}(A) \subseteq \{x\}$, in order to simplify the evaluation algorithms later on. A program $p(N, A_1, \dots, A_n)$ is an application of function p to the node value of N and the outputs hedges computed by A_1, …, A_n. A program $\mathbf{case}\ N\ \mathbf{of}\ F_1 \Rightarrow A_1 \dots F_n \Rightarrow A_n$ choses the first filter F_i that is satisfied by the node value of N and returs the output of A_i. A program $\mathbf{compose}\ x : A\ \mathbf{in}\ A'$ composes the transformations of A and A'. More precisely, The transformation A' is applied to the result of the transformation A, to whose root x will refer. A program $\mathbf{shred}\ (A, T)$ output the result of A onto the shred y_w where w is the name of a file computed from T. The only variables binders are let-expressions and function definitions.

A node descriptor N is either a node variable x or a node identifier v. A text descriptor $text(N)$ returns the text of the subtree of the input tree at the node described by N. A text descriptor "w" desribes w itself, and $T_1 \cdots T_n$ desribes the concatenation of the texts described by the T_i. A label descriptor $lab(N)$ returns the label of the node described by N, and label constant $a \in \Sigma$ describes itself.

3.3 Semantics

We will use definitions D as environments for the evaluation of X-Fun programs. We will always assume that all variables of the program are assigned to a value of D, and also that all variables used in D are assigned a value further on the right in D. When starting evaluating a closed program this assumption will always remain satisfied, as long as we assume that values of hedge variables are not defined in cycles. This can be checked statically by a simple acyclicity test. In slight abuse of notation, we will denote by $D(x)$, $D(y)$, and $D(q)$ the value that D assigns to the respective variable.

Given a data tree t, an enviroment D, we define the evaluator $[\![.]\!]_{t,D}$ for X-Fun program in in Figure 2. The definition is recursive in order to deal with least fix points of recursively defined functions. We also need auxilary evaluators $[\![.]\!]^{lab}_{t,D}$ for label descriptors, $[\![.]\!]^{text}_{t,D}$ for text descriptors, $[\![.]\!]^{node}_{t,D}$ for node descriptors. Most rules of the evaluator straightforwardly follow the intuition, so we concentrate on the nasty details. When computing $[\![\mathbf{for}\ x\ \mathbf{in}\ N/P\ \mathbf{do}\ A]\!]_{t,D}$ the document order of nodes on the tree t becomes relevant: we write $v_1 < v_2$ if the opening event of node v_1 comes before the opening event of v_2 in $lin(t)$. Let $v' = [\![N]\!]^{node}_{t,D}$. Let $\{v_1, \dots, v_n\}$ be the set of nodes reachable over path P from v, i.e., $\{v_1, \dots, v_n\} = \{v \mid (v', v) \in [\![P]\!]^{path}_t\}$, such that $v_1 < \dots < v_n\}$. In this case, the body of the loop A is evaluated for all $x = v_1$, …, $x = v_n$ in this order, and the results are concatenated. For computing $[\![\mathbf{case}\ N\ \mathbf{of}\ F_1 \Rightarrow A_1 \dots A_n \Rightarrow F_n]\!]_{t,D}$ there are two issues. If more then one filter F_i selects the value of N, then the leftmost filter is applied. If none of the filter matches, then the empty hedge is returned. Alternatively, we could have added an explict else statement, or we could have considered the lack of a match as a program error. When computing $[\![\mathbf{compose}\ x : A\ \mathbf{in}\ A']\!]_{t,D}$ on has to be careful that $[\![A]\!]_{t,D}$ returns a shredded data hedge, which must be converted into a non-shredded data tree t', before the transformation $[\![A']\!]_{t',D\&x=root}$ can be applied to it. The result of $[\![shred(A,T)]\!]_{t,D}$ is the shredded hedge $\mathbf{shred}\ y_w = [\![A]\!]_{t,D}\ \mathbf{in}\ y_w$, so that output of A will be written onto file w of a shredded stream.

Program

$$[\![y]\!]_{t,D} = D(y)$$
$$[\![subtree(N)]\!]_{t,D} = t_{|[\![N]\!]^{node}_{t,D}}$$
$$[\![tree(L,A)]\!]_{t,D} = [\![L]\!]^{lab}_{t,D}([\![A]\!]_{t,D})$$
$$[\![T]\!]_{t,D} = [\![T]\!]^{text}_{t,D}$$
$$[\![A_1 \cdots A_n]\!]_{t,D} = [\![A_1]\!]_{t,D} \cdot \ldots \cdot [\![A_n]\!]_{t,D}$$
$$[\![\mathbf{let}\ D'\ \mathbf{in}\ A]\!]_{t,D} = [\![A]\!]_{t,D\&D'}$$

$$\frac{q(x, y_1, \dots, y_n) = A \text{ in } D \qquad D' = (x = [\![N]\!]_{t,D} \& y_1 = [\![A_1]\!]_{t,D} \& \dots \& y_n = [\![A_n]\!]_{t,D})}{[\![q(N, A_1, \dots, A_n)]\!]_{t,D} = [\![A]\!]_{t,D\&D'}}$$

$$\frac{\{v_1, \dots, v_n\} = \{v \mid ([\![N]\!]^{node}_{t,D}, v) \in [\![P]\!]^{path}_t\} \qquad v_1 < \dots < v_n}{[\![\mathbf{for}\ x\ \mathbf{in}\ N/P\ \mathbf{do}\ A]\!]_{t,D} = [\![A]\!]_{t,D\&x=v_1} \cdot \ldots \cdot [\![A]\!]_{t,D\&x=v_n}}$$

$$\frac{\text{if } i \text{ is the smallest number such that } [\![N]\!]^{node}_{t,D} \in [\![F_i]\!]^{filter}_t \text{ then } h = [\![A_i]\!]_{t,D} \text{ else } h \text{ is empty hedge}}{[\![\mathbf{case}\ N\ \mathbf{of}\ F_1 \Rightarrow A_1\ \dots\ A_n \Rightarrow F_n]\!]_{t,D} = h}$$

$$\frac{t' = to\_tree(unshred([\![A]\!]_{t,D}))}{[\![\mathbf{compose}\ x : A\ \mathbf{in}\ A']\!]_{t,D} = [\![A']\!]_{t',D\ \&\ x=root}}$$

$$\frac{w = [\![T]\!]^{text}_{t,D}}{[\![shred(A,T)]\!]_{t,D} = \mathbf{shred}\ y_w = [\![A]\!]_{t,D}\ \mathbf{in}\ y_w}$$

Node

$$[\![v]\!]^{node}_{t,D} = v$$
$$[\![x]\!]^{node}_{t,D} = t[D(x)]$$

Text

$$[\![\text{“}w\text{”}]\!]^{text}_{t,D} = \text{“}w\text{”}$$
$$[\![text(N)]\!]^{text}_{t,D} = text(t_{|[\![N]\!]^{node}_{D,t}})$$
$$[\![T_1 \dots T_n]\!]^{text}_{t,D} = [\![T_1]\!]^{text}_{t,D} \dots [\![T_n]\!]^{text}_{t,D}$$

Label

$$[\![a]\!]^{lab}_{t,D} = a$$
$$[\![lab(N)]\!]^{lab}_{t,D} = t[[\![N]\!]^{node}_{t,D}]$$

Figure 2. Semantics of X-Fun.

```
addresses(
  address(
    name("Jemal Antidze")
    phone("99532 305072",secret)
    email("jeantidze@yahoo.com")
    phone("99532 231231")
    email("jeantidze@ip.osgf.ge")
    city("Tblissi")
    street("Gia Tetelashvili Street"))
  address(
    name("Joachim Niehren")
    city("Lille")
    street("Rue Esquermoise"))
)
```

⇒

```
<ol>
  <li>
    <p> Jemal Antidze </p>
    <p> Gia Tetelashvili Street</p>
    <p> Tblissi <p/>
    <p> Phone: 99532 231231 <p/>
    <a href="mailto: Jemal Antidze
       &lt; jeantidze@yahoo.com &gt;">
       jeantidze@yahoo.com </a>
    <a href="mailto: Jemal Antidze
       &lt; jeantidze@ip.osgf.ge &gt;">
       jeantidze@ip.osgf.ge </a>
  </li>
  <li>
    <p> Joachim Niehren </p>
    <p> Rue Esquermoise</p>
    <p> Lille </p>
  </li>
</ol>
```

Figure 3. Publication of an address book in HTML except for secret entries.

```
let
  convert_address(x) =
    let
      name   = for x' in x/child::name
               do subtree(x')
      street= for x' in x/child::street
               do subtree(x')
      city   = for x' in x/child::city
               do subtree(x')
      convert_email(x') =
         tree(a,tree(href_attr, "mailto: "
                     name "<" text(x') ">")
              text(x'))
    in
      tree(li,
        tree(p,name)
        tree(p,street)
        tree(p,city)
        for x' in x/child::phone
                  [not(child::secret)] do
          tree(p,"Phone: " subtree(x'))
        end
        for x' in x/child::email do
           convert_email(x')
        end)
    end
in
  tree(ol,for x in root/descendant::address do
            convert_address(x)
          end)
end
```

Figure 4. X-Fun program converting address books to HTML.

3.4 Examples

In Figure 3 we illustrate a transformation that converts an address book into HTML, while leaving out secret information. The address fields are assumed to be unordered in the input data tree, while the fields of the output HTML addresses should be published in the order `name`, `street`, `city`, `phone` and `email`. This transformation is perfectly streamable in practice, so shredding should not be necessary here. In theory, however, one might still need an unbounded memory for buffering an unbounded number of phone numbers or email addresses.

An X-Fun program defining this transformation is given in Figure 4. Starting at the root, if first selects all descendants `x` locating an address records, and applies the function `convert_address` to all of them. For each address record, the program first extracts the values of the fields `name`, `street`, and `city` located at some children of `x`. These values are then bound to hedge variables named alike. In the scope of these variables, the function `convert_email` is defined, which outputs a hyperlink by which one can send Email to this address, while displaying the `name` too. Recall that we treat HTML attributes as normal children but with the typed label `href_attr` instead of the untyped label `href`.

The question of how to write the same transformation in the streaming fragment of XSLT 3.0 is not that simple. First of all, the many parallel applications would require to introduce `xsl-fork` statement in many places (which however is not yet implemented in Saxon). Second, one cannot use XPATH expressions such as

```
child::phone[not(child::secret)]
```

in XSLT 3.0 in contrast to X-Fun, since such XPATH queries cannot select answer nodes directly at their opening event with 0-delay. One has to wait until the closing event in the worst case. In XSLT 3.0, a work around enabling larger delays would be to store a copy of the whole subtree in a variable (such as does our **compose** operator) and then process the variable in a non-streaming mode. In the XSLT specification, this technique is called burst-mode streaming. All these complications show that X-Fun has already important advantages for streaming compared to XSLT 3.0 even without any output shredding, since no a priori rewriting is needed for running the streaming engine.

Parameter Passing. In our X-Fun example, the `name`-field is used as a parameter of the function `convert_email`. We have thus passed a global parameter by defining this function in the scope of the hedge variable `name`. Another possibility would have been define the function `convert_email` at top-level with an explicit second argument received at calling time:

```
convert_email2(x',name) =
    tree(a,tree(href_attr,"mailto: " name
                "<" text(x') ">")
         text(x'))
```

More generally, it is always possible to move definitions to the top-level, by adding arguments for all its parameters. Conversely, one can also always use function definitions where all parameters

are defined by free variables, so that only a single node valued argument remains for all function definitions.

A third alternative to parameters is to use navigation with backward axis in path expressions. Here, the `name` field is found at a sibling of the `email`-field, so we can access it as follows:

```
convert_email3(x') =
  let
    name  = for x' in x/parent::*/child::name
            do subtree(x')
  in
    tree(a,tree(href_attr,"mailto:" name
                "<" text(x) ">") text(x))
  end
```

Such a programming style may appear appropriate, if one only wants publish all email addresses of the addrees book, so that one does not want to program the whole navigation manually.

```
let
  convert_email3(x') = ...
in
  for x in root/descendant do convert_email2(x)
end
```

It should also be noticed that the programming styles with global parameters (alternative 2) or backward axis (altenative 3) are supported by XSLT, while local parameters (alternative 1) are not.

Composing Transformations. The compose expression allows to change the status of parameters, so that they can transformed again. For instance, we can implement a transformation, that recursivley applies the rewrite rule:

$$p(a(y_1), b(y_2)) \to p(y_1, y_2)$$

This can be done as follows:

```
let
  q(x) = case x of [child::a and child::b] ⇒
     let
       y1 = for x' in x/child::a/child::* do
            subtree(x')
       y2 = for x' in x/child::b/child::* do
            subtree(x')
       y = p(y1,y2)
     in
       compose x:y in q(x)
     end
  in
    q(root)
end
```

By using such a function, one can test for instance whether the input tree has the form $p(a^n, b^n)$ for some n. More generally, one can express arbitary term rewrite systems, so one can encode Turing machines.

3.5 Expressiveness

We first argue that the sublanguage of X-Fun without **compose** and **shred** , and texts, is equivalent in expressive power to a variant of macro tree transducers [9] for unranked trees. We will show that XSTREAM is subsumed [12] by X-Fun without **shred** -expressions.

Walking macro tree transducers A top-down tree transducer performs tree-to-tree transformations by performing a single top-down pass over the input tree while producing an output tree in the process. It can perform simple operations such as renaming, reordering children and filtering. However, more complex operations are not possible as it can only remember a constant amount of information in the state. A macro tree transducer (MTT) is a generalization of a top-down tree transducer with parameters, by which partial output trees can be stored and output later on. In [20], macro tree transducers were extended to *walking* MTT*s on unranked trees*, which can navigate using path expressions P and filters F for some class of formulas.

Definition 3. A *walking* MTT is a pair (R, A_0), where A_0 is an initial action and R is a finite set of rules of the form

$$q(x_1 \in F, y_1, \ldots y_k) \Rightarrow A \ ,$$

where q is a function name, x_1 a node variables, and $y_1, \ldots, y_k$ hedge variables, and A an action with the following abstract sytnax:

$$\begin{array}{rcl} A & ::= & y_j \mid \mathit{subtree}(x_1) \mid \mathit{tree}(a, A) \mid A_1 \ldots A_n \\ & \mid & q'(x_2 \in x_1/P, A_1, \ldots A_m) \ , \end{array}$$

We illustrate the relationship between walking MTTs and X-Fun by demonstrating how to translate them into X-Fun programs. Basically, the translation is straight-forward, since we used already symbols for coinciding concepts.

$$\begin{array}{l} y_i \mapsto y_i \\ \mathit{subtree}(x_1) \mapsto \mathit{subtree}(x_1) \\ \mathit{tree}(a, A) \mapsto \mathit{tree}(a, A) \\ A_1 \ldots A_n \mapsto A_1 \ldots A_n \\ q(x_2 \in x_1/P, A_1, \ldots, A_n) \mapsto \left\{ \begin{array}{l} \textbf{for } x_2 \textbf{ in } x_1/P \textbf{ do} \\ \quad q(x_2, A_1, \ldots A_n) \end{array} \right. \\ \left. \begin{array}{l} q(F_1(x_1), y_1, \ldots, y_n) \Rightarrow A_1 \\ \ldots \\ q(F_m(x_1), y_1, \ldots, y_n) \Rightarrow A_m \end{array} \right\} \mapsto \left\{ \begin{array}{l} q(x_1, y_1, \ldots, y_n) = \\ \quad \textbf{case } x_1 \textbf{ of} \\ \quad F_1 \Rightarrow A_1 \\ \quad \ldots \\ \quad F_m \Rightarrow A_m \end{array} \right. \end{array}$$

A top-level **let** expression is needed in order to combine translate an walking tree transducer (R, A) into **let** D **in** A' where D is the set of rules obtained by translating R and A' the translation of A. Conversely, the fragment of X-Fun without **compose** , **shred** , and text can be expressed by walking tree transducers, mainly, since environment for binding hedge variables can be eliminated. With **compose** instructions, however, X-Fun becomes Turing-complete.

XStream X-Fun also subsumes to XSTREAM [12], a functional language for transforming XML documents in a streaming fashion. However, XSTREAM lacks support for both shredding and XPATH navigation. The latter makes it difficult to translate non-descending transformations. The usual pattern matching that XSTREAM provides leads to a quite different programming style, mainly based on term rewriting.

4. Compiler of XSLT to X-Fun

We recall the main concepts of XSLT and show how to translate them to X-Fun.

***Concepts on* XSLT** A fragment of an XSLT program computing the table of contents for a book could look like the following

```
<xsl:stylesheet>
<xsl:template match="book">
    <book>
        <xsl:apply-templates select="." mode="toc"/>
        <xsl:copy-of select=child::*/>
    </book>
</xsl:template>
...
</xsl:stylesheet>
```

An XSLT program consists of a set of templates, which can be applied to some node of an input document. The output produced

by the template application is a combination of static data, data copied from the input and output produced by recursive calls. There are two types of calls in the XSLT language. The `call-template` instruction calls a template by directly specifying its name, while `apply-templates` dynamically selects a template (which is then called a *template rule*) to be evaluated from a set of templates belonging to the specified *mode* according to the match filters of the templates. Ambiguities in the second case are solved by assigning explicit priorities to individual template rules. Both types of calls allow passing additional document fragments as parameters. In the second case, one can also navigate in the input document.

Compiler to X-Fun In the rest of this section, we show how to translate the most important constructs of XSLT to X-Fun. The supported fragment of XSLT includes named templates, template rules, instructions `call-template`, `apply-templates`, `if`, `choice`, `for-each`, `copy`, `copy-of` and dynamic content creation instructions (`attribute`, `value-of`, ...). Templates can be called with any number of parameters. However, the set of allowed operations on the parameters is restricted to two: output and apply a template to the value of the parameter.[1]

A complete XSLT program is translated into an X-Fun program, with a top-level let binder introducing all function definitions, followed by an application of the default mode function to the root node of the input. Each template is translated to a function definition with the corresponding number of parameters. Furthermore, for each mode, we specify a special function which selects the right rule in a case expression and calls the corresponding functions. Priorities of the rules are implemented using a suitable ordering of the individual case clauses. As an example we We consider the following XSLT program:

```
<xsl:template name="p" match="a">
    <B><xsl:apply-templates select="child::*"/></B>
</xsl:template>
<xsl:template name="q" match="b">
    <C><xsl:call-template name="p"/></C>
</xsl:template>
```

It will be translated to the following definitions in X-Fun:

```
p(x)=tree(B,for x' in x/child::* do default(x'))
q(x)=tree(C,p(x))
default(x) = case x in
   [self::a] ⇒ p(x)
   [self::b] ⇒ q(x)
```

The `call-template` instruction is translated to a simple call $p(x, \ldots)$, where p is the function corresponding to the called template and ... stands for the translation of the call parameters, described later. The `apply-templates` instruction is more complicated as it involves calling a template for each node selected by an XPATH expression. It can be translated to

$$\textbf{let } params = \ldots \textbf{ in for } x' \textbf{ in } x/P \textbf{ do } m(x', params) \ ,$$

where P is the XPATH expression, and m is the rule-selecting function for the specified mode. There are two ways of specifying template parameters. The first is using an XPATH expression and this is translated as $\textbf{for } x' \textbf{ in } P(x) \textbf{ do } subtree(x')$. The second possibility is to use XSLT expressions to construct an output tree, and it is translated in the same way as template body. The rules for translating the individual constructs in the template body are very straight-forward. The `if` and `choice` instructions are translated as **case** expressions, `for-each` results in a **for** expression. Instruction `copy-of` translates to a combination of **for** and *subtree* expressions, since it can navigate in the tree. The shallow copy `<xsl:copy>...</xsl:copy>` construct translates to $tree(lab(x), \ldots)$. Dynamic content creation instructions like `attribute`, which enables us to output an XML attribute whose name and value are dynamically computed, can also be translated.

[1] Specifically, this excludes running complex XPATH expressions, which access the parameter alongside the main document.

5. Hyperstreaming Evaluation

We first illustrate the main ideas, then discuss XSMs that we use for streaming XPATH evaluation, and finally present our functional engine and its interaction with XSMs.

5.1 Basic Ideas

We sketch the main ideas of the hyperstreaming evaluator for X-Fun and illustrate them by example. Suppose we are given a close X-Fun program A. The first issue is to replace all XPATH expressions in A by an XPATH streaming machine (XSM), which will interact with the functional engine evaluating A. For an XPATH expression that occurs in a loop $\textbf{for } x' \textbf{ in } x/P \textbf{ do } A'$ the corresponding XSM M will compute all nodes x' reachable from x via P, so that the functional machine can evaluate the loop $\textbf{for } x' \textbf{ in } M(x) \textbf{ do } A'$. It will tell the XSM $M(x)$ whenever x can be bound to a node opened on the stream, while $M(x)$ will report all selected nodes for x' to the functional machine.

One of the problems is that some nodes may become a candidate for an x' answering $M(x)$, but that one has to wait for more information on the stream to come in order to decide whether this node is really an answer. In this case, the functional machine has to start evaluating A' for this node v for x'. The intermediate result must be buffered by the functional machine. It will become valid only once the selection of v gets confirmed, and must be discarded once the rejection of v is reported. Therefore, an XSM machine must also inform the functional machine about the creation of alive candidates, their selection and rejection. As an example, we reconsider the X-Fun program converting the address book. Beside others, it contains a loop converting all address records, and another loop converting all non-secret phone numbers.

$M_1(root)$: **for** x **in** root/child::adresse **do** A_1
$M_2(x)$: **for** x' **in** x/phone[not(child::secret)] **do** A_2

When ever the opening tag of an address node is visited, machine M_1 will report the selection of identifier v of this node. The functional machine will then have to evaluate A_1 with the environment $x = v$, i.e., compute $[\![A_1]\!]_t x = v$. Furthermore, it will have to pass v as a possible value of x to M_2. Whenever the phone field v' of address v is opened, M_2 has to report that v' is an alive candidate for x', but cannot know yet, whether it will be selected. In this example, M_2 can decide rejection once the `secret`-child v is opened.

Generally, all parallel calls in the alive part of the program need to be evaluated in parallel. The functional program has to evaluate subexpressions for all alive, and to buffer the intermediate results, until selection or rejection is decided by some XSM machine. The XSM machines communicate with the functional machine, which in turn communicate to possible other XSM machines. This way, only the alive part of the program is evolved.

Hyper-stream comes in a most natural manner, since the functional machine is evaluating all alive parts of the program in parallel. For all shreds computed anyway, it simply output the maximal part on the corresponding output stream.

5.2 Streaming Evaluation of XPATH

In this section we introduce a formal model for a machine evaluating an XPATH expression on a data tree. Since XPATH evaluation is not the subject of this paper, we only give a high-level input-output specification of the machine in sufficient scope so that we

can later reason about it. It subsequent sections we show how to connect multiple machines together and use them in evaluation of X-Fun programs.

Let P be an XPATH expression. An *XPath streaming machine* (XSM) for expression P is a machine with two input tapes and a single output tape. The input and output tapes are read (or written, respectively) in a left-to-right manner. The first input tape contains a nested word serialization of a data tree t. The second input tape contains a word u over $\{0,1\}$ such that $|v| = |nodes(t)|$. I.e., each letter in u corresponds to a node of t. If v is a node of t, we shall denote the corresponding letter of u by u_v. The letters u_v in u are ordered according to the preorder ordering of t.

An XPath streaming machine computes the relations $[\![F]\!]_t^{path}$ and $[\![F]\!]_t^{filter}$ incrementally. The operation of an XSM can be defined in terms of variables. An XSM has one or two variables, which range over nodes of the input tree. An XSM used for evaluating path expressions (present in **for** -expressions) has two variables x and z. The x variable defines the starting point of the expression evaluating and z defines the selected tree nodes. An XSM derived from XPath filters (from **case** -expressions) contain only the variable x, since they only return a boolean result (whether a node satisfies the filter).

During the processing of the input stream, the XSM maintains a list of *candidates* — tuples of nodes, which have already been read but it is yet unclear whether they belong to the corresponding semantic relation. These candidates are said to be *alive*. Once the machine reads a sufficiently large prefix of the input stream to determine the status of the candidate, it either *selects* or *rejects* it. The machine can also have partial candidates like $(v, \bullet)$, $(\bullet, v)$ or $(\bullet)$ which represent tuples, where some component has not yet been read from the stream. When an appropriate node arrives, $\bullet$-entries are instantiated to form new candidates. The output tape of an XSM then contains tuples (v, v', S) or (v, S) respectively, where $S \in \{\text{ALIVE}, \text{REJECT}, \text{SELECT}\}$. ALIVE is output as soon the opening event of a node is read and the SELECT and REJECT events are output when the candidate status is determined. In case of path expressions, we also output the tuples $(\bullet, v, \text{ALIVE})$ once a candidate with $z = v$ is instantiated. It is important that XSM also reports aliveness information, since this information is needed in the X-Fun machine, to compute the transformation output. Additionally, the XSM for the path expression reports a special tuple $(v, \bullet, \text{REJECT})$ when it can determine that there will be no more v-tuples produced (all candidates with $x = v$ are completed, and no new candidates will be generated). The handling of candidates is illustrated in the following examples (for the moment, we ignore the second input tape of the machine):

Example 1. Let us assume the input tree $a(a(b))$ and let v_1, v_2 and v_3 refer to the first, second and third node of the tree, respectively. An XPath streaming machine for the filter `[self::a[child::b]]`, selecting all nodes `a` that have a child `b`, will operate as follows on the stream containing `<a><a><b></b></a></a>`. After reading the first event, it outputs (v_1, ALIVE) since it cannot determine from the processed prefix whether node v_1 has a `b`-labeled child. After reading the opening event of v_2, it outputs (v_2, ALIVE) for the same reason. At this moment, the machine maintains two output candidates, corresponding to the two tree nodes. After reading the next event, it becomes clear that v_2 is selected by the XPATH expression and the machine outputs (v_2, SELECT), completing the evaluation of the second candidate. This node is not even considered for a candidate, since its label is incompatible with the filter. Since the node was immediately rejected, the machine does not output any v_3-tuple. The closing event of v_3 generates no new candidates or output. The final event, closing of v_1 completes the evaluation of the first candidate, since it is now clear that it has no `b`-children. The machine therefore outputs (v_1, REJECT).

Example 2. A machine for path expressions will operate similarly, except that it will output pairs of nodes. For example, an XSM for the expression `self::a/descendant::b` on the same input document will output the tuples $(v_1, v_3, \text{SELECT})$ and $(v_2, v_3, \text{SELECT})$ after the third event and $(v_2, \bullet, \text{REJECT})$ and $(v_1, \bullet, \text{REJECT})$ after the respective close event.

The second input tape restricts the tuples which are written to the output tape. Specifically, if $u_v = 0$ the machine does not output any tuples, which have v as the first member. If $u_v = 1$, then the tuples are output as described in the previous paragraph. This can be used decrease the resource (time and memory) of a XSM—resource usage is dependant on the number of nodes we start the XPATH evaluation from and normally we will be only interested in a fraction of nodes.

The XSM then operates as follows. First, it reads an event from the input stream and tries to advance already all alive candidates and outputs any tuples it has produced. Let this event correspond to node v. Next, if the event is an open event, then a letter from the second input tape (which will be the letter u_v, because of the ordering of the tape) is read. If $u_v = 1$ then it starts an evaluation of candidates for $x = v$ and outputs any additional tuples.[2] The processing then resumes with the next input event.

5.3 Systems of XPATH Streaming Machines

When evaluating an X-Fun program on an input document, we will usually need to run evaluate multiple XPATH expressions. To achieve this, we will use multiple XSM instances, one for each path and filter expression.[3] These machines will operate synchronously, reading the same input document on the first tape. The benefit of this approach is that we can easily discard an event once it has been processed by all XSM.

The second tape of XSM is not shared by all the XPath streaming machines. Instead, it is computed separately for each XSM by the X-Fun machine that controls them. The X-Fun machine determines whether it needs to start at XPath expression from the current node using the output produced in the first phase (before reading u_v from the second tape) of the XSM computation on the current event. Then it sets the value of u_v for individual machines and resumes their computation.

Example 3. Let the currently evaluated X-Fun fragment be $A_1 = \mathbf{for}\ x\ \mathbf{in}\ \mathit{root}/P_1\ \mathbf{do}\ A_2$. Suppose that the expression P_1 selects the current node v. Then, the XSM machine for P_1 will return (root, v, S) with $S = \text{SELECT}$ or $S = \text{ALIVE}$. In either case, we know that v is a candidate for output and we start evaluating A_2 with $x = v$. We unfold A_2, including all the function calls, looking references to x.

Suppose now that A_2 contains another for-loop, say $\mathbf{for}\ x'\ \mathbf{in}\ x/P_2\ \mathbf{do}\ A_3$. This means that we need to evaluate P_2 starting from v. Therefore, we set $u_v = 1$ for the machine for P_2 and let it proceed with the second phase of evaluation. During this phase the machine may return (v, v, SELECT), which means we must continue to unfold A_3, etc. This process stops once a machine does not return a tuple with $z = v$ or reach a machine which already has $u_v = 1$ set. Note that in the first case, we detect a loop in the program, and therefore the X-Fun machine will loop also. However, the process of u_v-saturation, which is discussed here, will always complete.

[2] For example, the tuple (v, v, S).

[3] If a program contains multiple instances of some expression (which can be quite common for expressions like `child::*`), they can all be handled by the same XSM.

Example 3 illustrates why we have to split the XSM processing into two phases, and also the algorithm which controls the individual machines and computes the u_v values.

Output reprocessing In the previous section we showed how to run multiple XSM on a single input stream. However, in X-Fun it is possible to process an output hedge, which is dynamically constructed (transformation composition). Evaluating a **compose** instruction consists of creating a new set of XSM machines, which will process the newly constructed document. The control of the machines remains the same, the only difference is that the new XSM set reads the input stream directly from the sub-process which creates it instead of some external source.

5.4 X-Fun Evaluation

The data structure holding the computed-but-not-written part of output consists of records, which are connected in a DAG structure. Each record has two components: an output buffer and a *kind*. The output buffers can hold output events and other records. The contents of the buffers is well nested in the sense that if a buffer contains an open-element event, it will also contain the corresponding close event. However, there is no restriction on the number of elements of the same level (the buffer can contain an arbitrary hedge) and it is not necessary to create a new buffer for every nesting level.

A sub-record can be shared between multiple records that contain it. This means that e.g., records containing the computed values of function parameters are stored only once, although the parameter can be output many times. We track the number of references to each record so that we can release it as soon as it is no longer needed. In fact, when outputting with a reference count of 1, so that we are sure that this is the last usage, we release the memory for the members in parallel to the reading of the buffer. This is especially important since a lot of programs have for-each-record loops at the top level, with a potentially unbounded number of records on the input. If we released the memory for the buffer only after reading the whole buffer, we would have memory usage linear to the size of the input even in case of a strictly streamable transformation. With the immediate removal from the buffer, the output buffer will contain at most one entry for a perfectly streamable transformation.

The kind of the record determines which algorithm fills the buffer. We have separate algorithms for loops, case expressions, etc. The algorithm may create additional working buffers in the record for storing computations corresponding to alive candidates, which still may become rejected. However, to minimize the memory used for buffering, it should move to computed output to the output buffer as soon as it is sure the output will be correct. The operation of the individual algorithms will be described in Section 5.5.

At the start of the evaluation, the evaluator shall have one top level record, which corresponds to the initial term of the program. The evaluation as soon as some output events appear in the output buffers of this record hierarchy, the evaluator, will write them to the output stream and, if the containing record is not referenced elsewhere, remove them from the buffer. Additional top-level records can be added by the *shred*-operation and also by the for-loops, which need to speculatively evaluate some terms.

Similarly, at the beginning there will be only one set of XSM machines responsible for evaluating path queries and filters on the main input tree. Once we start reprocessing computed trees (instruction **compose**), we create additional XSMs which will process the temporary tree operating in the same manner as on the main tree. To avoid buffering of the computed parts of the tree, we will prefer to process this tree instead of the main one, processing each event as it is computed. Since computing the temporary tree will involve reading the input from the main tree, we will not starve the computation on the main tree.

We shall write the records as tuples $R = (B, K)$, where B is the output buffer and K is the record kind.Furthermore, output buffers will be written as sequences $(r_1, r_2, \ldots, r_n, \rightarrow)$, where r_i can be either a computed output event, or another record. Newly computed elements are added to the end of the sequence. When we know that there will be no more new events added to the buffer (the evaluation algorithm has completed), we omit the arrow at the end of the buffer: $(r_1, \ldots, r_n)$.

5.4.1 Example

We will illustrate the process of X-Fun evaluation on the following program:

```
let q(x) = case x in
       [self::b[c]] ⇒ "c"
       [true]       ⇒ "b"
in tree(a, for x' in root/child::b do q(x')) end
```

Let us assume we are evaluating this program on the data tree $a(b)$, so the input stream will contain the nested word `<a><b></b></a>`. We shall refer to the a-nodes of the tree by v_1 and the b-node by v_2, while for the 4 events of the nested word, we shall use $e_1, \ldots, e_4$ in their order. After reading the event e_1, we create the following record structure:

$$R_1 = ((\texttt{<a>}, R_1', \texttt{}), tree) \qquad R_1' = ((\rightarrow), for)$$

R_1 holds the evaluated form of the topmost tree-expression. The closing and opening events of the `a`-node have already been written to the buffer (and this completes the work of $\mathcal{A}_{tree}$). The non-static parts of the expression, in this case the for-loop, have been replaced by references to sub-records. The buffer of R_1' will hold the result of the evaluation of the for-loop. At first, it is empty, because we have not encountered any `b`-children yet (the XSM machine has not produced any output). On the other hand, buffer of R_1 has a prefix ready for output, so we write `<a>` to the output stream. Since R_1 is not referenced by other buffers, we remove the event from the buffer.

After reading the event e_2, $\mathcal{A}_{for}$ receives the $(v_1, v_2, \text{SELECT})$ tuple from the XSM. Since now we know that v_2 got selected, we start evaluating the body of the for loop with $x' = v_2$. We enter the function call and start evaluating the case-expression. To determine which branch of the expression to choose, we signal the XSM machines to examine the current node (we set $u_{v_2} = 1$ on the corresponding input tape). The machine for the second filter reports selection, but the first machine doesn't have the definitive answer yet. Therefore, we must start evaluating both branches and the record structure becomes:

$$R_1 = ((R_1', \texttt{}), tree) \qquad R_1' = ((R_2, \rightarrow), for)$$
$$R_2 = ((\rightarrow), case) \qquad R_2^1 = ((\text{“}\texttt{c}\text{”}), text)$$
$$R_2^2 = ((\text{“}\texttt{b}\text{”}), text)$$

To the buffer of R_1' we have added R_2, which will hold the result of the function call. However, its buffer is still empty. The evaluation of the two **case** -branches is performed in records R_2^1 and R_2^2. We cannot generate any output in this step.

The third event closes node v_2, which means that now we are sure that the b-node has no c-labelled children. Therefore, the XSM for the first branch reports rejection and we can definitely say that the computation will proceed with the second branch. Therefore, $\mathcal{A}_{case}$ puts R_2^2 into the buffer of R_2, resulting in the configuration:

$$R_1 = ((R_1', \texttt{}), tree) \qquad R_1' = ((R_2, \rightarrow), for)$$
$$R_2 = ((R_2^2), case) \qquad R_2^1 = ((\text{“}\texttt{c}\text{”}), text)$$
$$R_2^2 = ((\text{“}\texttt{b}\text{”}), text)$$

Now, we see that we can output “`c`”. After the output and cleanup of empty (R_2^2, R_2) and unreferenced records (R_2^1), the structure

```
let S be a global stack of subtree buffering records

function eval_subtree(node v)
  if S is empty then
    push R' = ((→), subtree) to S
    // R' shall refer to v
  else
    R = top(S)
    if R refers to v then
      return R
    suspend buffering of R
    R' = ((→), subtree(v))
    push R' to S
    add R' to R
end

procedure subtree_event(event e)
  R = top(S)
  // let R refer to node v
  add e to buffer of R
  if e is the close event of v then
    mark R as completed
    pop(S)
    if S is not empty then
    resume buffering of top(S)
end
```

Figure 5. Evaluation functions for expression $subtree(N)$. Function `eval_subtree` is called when we reach $subtree(N)$ while unfolding the program. The function parameter is the value of the node variable. The `subtree_event` of the topmost buffer on the stack is called for every event processed.

simplifies to:

$$R_1 = ((R'_1, \texttt{}), tree) \qquad R'_1 = ((R_2, \rightarrow), for)$$

When we close the root node v_1, the XSM for the path expression reports that the evaluation for $x = v_1$ is complete, which completes the evaluation of R'_1. Therefore, we output `</a>` and remove all the buffers, which completes the transformation.

5.5 Algorithm

We illustrated the operation of the basic X-Fun constructs on the example in the previous section. Now, we give the details of the evaluation algorithms for different kinds of output records.

***Deep copy* — $subtree(N)$.** If the expression $subtree(root)$ is present in the program, we need to start buffering the whole input. Otherwise, we start buffering when we reach the expression $subtree(x)$. We buffer all the input events, starting from the current event (which will always be the event pointed to by the node variable) up to the corresponding close event. If multiple buffering requests come in for the same node, we they shall share the same output record. If a buffering request comes in while we are already buffering an input subtree (starting at some parent v' of the current node v), we shall create a new record. However, to avoid the buffering of the same event multiple times, we shall suspend the buffering for the v'-node. Instead, we shall simply place the newly created buffer for v in the v' buffer. We resume filling the old buffer once the new buffer completes. The pseudocode for this procedure is presented in Figure 5.

***Text values* — $text(N)$.** Similar to $subtree(N)$, except we buffer only text value of the subtree.

***Shallow copy* — $tree(L, A)$.** We create a three-element buffer (e_x, R', e'_x), where e_x and e'_x are the opening and closing events of the current node and R' stores the result of the evaluation of A.

```
let q(x) =
    let y = A1 in
      for x' in x/P do A end
    end
in A3 /* calls q */ end
```

Figure 6. Program fragment demonstrating for loop evaluation

***For loop* — for x' in N/P do A.** For loop is the most complicated construct to evaluate. Because the path expression in the loop can return nodes which precede the starting node, we must prepare for the evaluation of the loop even before we reach the loop while evaluating the program. We do this by listening to the $(\bullet, v', S)$ notifications from the corresponding XSM machine. For every selected and alive node, we start the evaluation (create a new top level record) of the loop body, with $x' = v'$. If we later learn that an alive node was rejected, we terminate the computation in the body and release the associated memory. The loop body can refer to parameters, which are not yet available. However, this is not a problem, since the only possible operation for the parameters is outputing them, so we can just output a placeholder record, which will wait for the real parameter value. Essentially, this creates a record template $R(R_{y_1}, \ldots, R_{y_n})$, which can then be instantiated to form an actual record.

When we later reach the loop while evaluating other an X-Fun term containing it, we signal the XSM machine to bind the first variable to the current node. After this, we start receiving (v, v', S), where v is the value of N. Now, if we receive a select or alive event, and v' comes before v on the input stream, we locate the record template for v' and instantiate it with real values of the parameters found in the environment. If v' comes after v, we simply create a new record and start evaluating the body of the loop into it. In either case, we put new record in the working buffer for v.

The working buffer is a list of sub-buffers, which have not yet been written into the output buffer for the loop. We store the sub-buffers here instead of the output buffer since some of the buffers may correspond to alive nodes, which may become rejected in the future. The buffers in the list are ordered according to the nodes they correspond to. The buffers linked to selected nodes, which are at the head of the list, are immediately removed and put in the output buffer, because we can be sure that no future events will change their status. Therefore, the first buffer in the waiting corresponds to an alive node. Once we receive the $(v, \bullet, \text{REJECT})$ tuple and the list becomes empty, we can conclude the evaluation of the for loop.

An illustration of the for loop evaluation can be seen on Figure 7 and we the pseudocode for the loop-handling algorithm is presented in the Appendix.

***Let expression* — let D in A.** Function definitions require no evaluation. When evaluating a definition of a hedge variable, we create a new buffer for the output hedge and start evaluating the term into the buffer. The buffer remains in the execution environment while it is referenced in the body of **let** term. Evaluating node variable definition means assigning the node identifier to the variable in the environment.

***Function call* — $q(N, A_1, \ldots, A_n)$.** We create buffers for the function parameters, like in the **let** term and we start evaluating the body of the function in the new environment.

***Case expression* — case N of $F_1 \Rightarrow A_1 \ldots F_n \Rightarrow A_n$.** Let the current node be v. We set $u_v = 1$ for all XSM machines linked to the filters in the case expression. If it is not immediately clear which branch will be selected (the non-rejecting machine with the lowest index reports (v, ALIVE)), we create sub-records

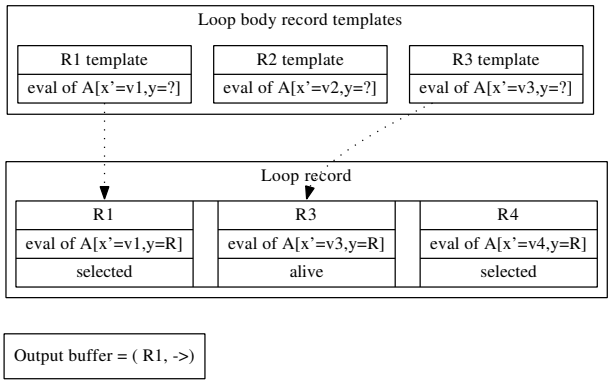

Figure 7. Illustration of for loop the computation for program fragment in Figure 6. Before $A3$ called q, the XSM has returned three alive candidates for $x' = v_1, v_2, v_3$, so we have started evaluating A with an unknown value of y (loop body templates box). The situation on the picture is after q has been called with $x = v_4$. The XSM has returned the following notifications: $(v_4, v_1, \text{SELECT})(v_4, v_3, \text{ALIVE})(v_4, v_4, \text{SELECT})$. Since v_1 and v_3 can be selected starting from the current node, we instantiate the corresponding records with the value of parameter y, which is now known from the environment. Furthermore, v_4 is also selected, so we begin a new computation of A with $x' = v_4$. As the first record is selected, we move it directly to the output buffer. The record for v_4 will not enter the output buffer even though the node is selected, because we don't know the status of v_3 and we need to preserve the node order in the output buffer.

and start evaluating the body of the loop for all branches that have a chance of being selected (this are the expressions whose XSM machine reports ALIVE or SELECT and there is no selecting machine with a lower index) into the sub-records. We then create an new empty record and return it. In subsequent steps, we terminate the computations of the branches that get rejected until there only one left. At this point, we can insert the remaining subrecord into the output buffer.

Parameter reference — y. we simply output a record containing the buffer for the hedge variable y, which is present in the environment.

Output reprocessing — $\mathbf{compose}\, x : A \,\mathbf{in}\, A'$. We create a record for storing the output of A and create new instances of the XSMs, which will read from the record. The output buffer will store the output of evaluating A' in the modified environment, where x is bound to the root of the tree produced by A.

Shredding — $shred(A, T)$. We create a new record for holding the evaluation of A. However, this record is not stored in the main output. Instead, we create a new output co-routine, which will run in parallel to the main one and write the output of this record onto a new stream. In the output buffer, we only output a symbolic reference to the main stream. The parameter T gives the name of the output file. If value of T is not immediately available (it depends on the input), we store the stream in a temporary file and rename it after the name becomes available.

6. Implementation and Experiments

We have implemented a part of the algorithm from Section 5 in the Java programming language. For the implementation of XSM we used the open source tool FXP 1.1[6, 7], with some custom modifications. In the compiler from XSLT to X-Fun, we also used the open source version of the QUIXPATH tool [7], which can compile XPATH expression into FXP terms that can then be evaluated by the FXP tool.

Customization of* FXP *interface. Although mainly motivated by the need for answering (unary) XPATH queries, FXP already contained support for answering binary queries of the type we described in Section 5.2, together with reporting aliveness information. However, it lacked the possibility to control the binding of the first variable, which we needed to avoid answering very general queries. Also, because of the need for elaborate control mechanism between XSM instances as illustrated in Example 3, we split the processing of one input event into two sub-steps. I.e., we chose the model with two tapes instead of the (more natural) one, where there is only one tape and the input events are annotated with the information whether one can bind the variable.

Implementation of X-Fun. We have implemented a significant fragment of the full X-Fun evaluation algorithm. The features which are lacking the current implementation are: support for multiple node variables (i.e., in a loop of type $\mathbf{for}\, x' \,\mathbf{in}\, x/P \,\mathbf{do}\, A$, the body of the loop A must not contain the variable x'), speculative evaluating of the loop body for cases when the the path expression selects a predecessor of the current node, and the support for re-processing a shredded tree (re-processing of a normal tree, is fully implemented). However, even with these limitations, our implementation is capable of streaming a much larger class of transformations than SAXON, including many almost-real-world examples.[4]

Besides completing the support for all language features, the implementation can be improved also in terms of performance, both running time and memory. For example, recently we were able to halve the amount of memory needed to buffer a fixed amount of output by storing the output buffers more compactly — by using a singly instead of a doubly linked list for storing the list of items, or even using arrays if the buffers were of fixed, known size.

XSLT *compiler.* The compiler is implemented by applying the rewriting rules from Section 4. However, to support real-world XSLT stylesheet, it needs support for additional XSLT features, some of which do not require any additional support from the core language (like `xsl:import` or the computation of the default priority of template rules), while some need more support from X-Fun (e.g., XSLT 3.0 accumulators) or FXP (XPATH with joins).

QUIXPATH. Since QUIXPATH compiled XPATH expressions into unary FXP terms, we needed to provide a wrapper interface over it, which added the second variable to the term, enabling the correct usage of path XSMs in X-Fun.

6.1 Experiments

To evaluate the streaming performance of our implementation, we have first compared it with the leading industry tool, the SAXON XSLT processor, which supports a subset of the streaming features in the upcoming XSLT 3.0 standard. To the authors' knowledge, there is no official and publicly available benchmark for XSLT. Furthermore, any real-world XSLT stylesheet would either fall out of the streamable fragment supported by SAXON, or it would run into the limitations of our implementation of the XSLT compiler and the FXP tool. Therefore, we have decided to do the comparison on a synthetic, but real-world motivated example.

We have run both implementations on a 1 gigabyte address book document. The transformation in question was a simple one,

[4] We say only *almost* real world, since practically used stylesheet usually contain at least a couple of XPATH expressions referencing template parameters (XPATH joins) or rely on the finer points of XSLT namespace handling and similar features, which are not supported by our implementation.

```
<xsl:stylesheet version="3.0" xmlns:xsl=
    "http://www.w3.org/1999/XSL/Transform">

  <xsl:mode streamable="yes"/>

  <xsl:template match="/">
    <Addresses>
      <xsl:for-each select="*/Address">
        <xsl:copy>
          <xsl:copy-of select=
              "node()[not (self::Note)]"/>
        </xsl:copy>
      </xsl:for-each>
    </Addresses>
  </xsl:template>

</xsl:stylesheet>
```

Figure 8. Stylesheet removing `Note` elements used in the test.

motivated by the examples from Requirements and Use Cases for XSLT 2.1 document[26]. The transformation takes an address book and removes the `Note` field from every `Address` element. The stylesheet for the transformation is given in Figure 8.

Running the transformation in SAXON required 206 seconds on average, while our own implementation took 479 seconds, which is an approximately 2.3-fold increase. These results are already good, given that our implementation can handle a much greater fragment of XSLT and XPATH than SAXON, and we believe that with further optimization we could reach the running time of SAXON. The transformation was run on computer with an Intel Core i7 processor running at 2.8GHz, with 4GB of RAM and a SATA hard drive, running a 32-bit version of Windows 7.

To show the versatility of our tool, we have also tested it using the transformation publishing an address book in HTML. The transformation in question is a more elaborate version of the program in Figure 4, and it includes 33 path expressions and 5 filters. It is not possible to perform this transformation in streaming mode in SAXON, therefore we tested it on a 400MB document, largest we could safely fit into memory. SAXON performed the transformation in 39.9 seconds, while our X-Fun implementation did it in 236. While this is a slowdown of factor almost 6, we do not consider it a completely fair comparison as SAXON had access to the whole input tree during the entire transformation, while our algorithm performed it all in one pass. Indeed, while SAXON barely fit the input tree in the memory, our algorithm has evaluated the same transformation in significantly less space, which is why we consider this result adequate.

Conclusion and Future Work

We have presented the functional core language X-Fun for hyperstreaming transformations on data trees. We have shown that X-Fun subsumes the core of XSLT, and shown that hyperstreaming for XSLT can be done without any a priori restrictions or manual code rewriting, also for transformations that are not streamable without shredding. We believe that this approach offers a valuable alternative to the current approach of the XSLT working group of the W3C, where streaming support is offered only for a smaller fragment.

We believe that X-Fun has much more to offer to XML processing, since it may also serve as a core language of XQUERY and XPROC. When it comes to XQUERY, we need to extend the underlying XPATH expressions with joins. For defining XPROC pipelines, we have to extend the language with operators for the dynamic creation of XPATH expressions. One might also be tempted to turn X-Fun into a core for NOSQL language, providing distribution and streaming support for JSON.

References

[1] M. Benedikt and A. Jeffrey. Efficient and expressive tree filters. In *FSTTCS*, volume 4855 of *LNCS*, pages 461–472. 2007.

[2] M. Benedikt, A. Jeffrey, and R. Ley-Wild. Stream Firewalling of XML Constraints. In *ACM SIGMOD* 487–498. 2008.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003.

[4] V. Benzaken, G. Castagna, K. Nguyen, and J. Siméon. Static and dynamic semantics of NoSQL languages. In *POPL*, 101–114. 2013.

[5] D. Crockford. Introducing JSON, 1999. http://www.json.org/.

[6] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, and M. Zergaoui. Early nested word automata for XPath query answering on xml streams. Technical report, INRIA Lille, Mar. 2013.

[7] D. Debarbieux, J. Niehren, T. Sebastian, and M. Zergaoui. FXP and QuiXPath tools for XML streams, 2012. http://fxp.lille.inria.fr/.

[8] J. Dvoráková and B. Rovan. A Transducer-Based Framework for Streaming XML Transformations. In *Current Trends in Theory and Practice of Computer Science*, pages 50–60. 2007.

[9] J. Engelfriet and H. Vogler. Macro tree transducer. *Journal of Computer and System Science*, 31:71–146, 1985.

[10] M. Fernandez, P. Michiels, J. Siméon, and M. Stark. XQuery streaming à la carte. In *23nd International Conference on Data Engineering*, pages 256–265, 2007.

[11] C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.

[12] A. Frisch and K. Nakano. Streaming XML transformation using term rewriting. In *Programming Language Technologies for XML (PLAN-X)*, 2007.

[13] O. Gauwin and J. Niehren. Streamable fragments of forward XPath. In *International Conference on Implementation and Application of Automata*, volume 6807 of *LNCS*, pages 3–15. 2011.

[14] X. W. Group. XSL transformations (XSLT) version 3.0, 2012. `http://www.w3.org/TR/xslt-30`.

[15] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *ACM SIGMOD*, pages 419–430. 2003.

[16] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[17] M. Kay. A streaming XSLT processor. In *Balisage: The Markup Conference. Balisage Series on Markup Technologies*, vol 5, 2010.

[18] P. Labath. XSLT streamability analysis with recursive schemas. In *RCIS*, pages 1–6. IEEE, 2012.

[19] P. Madhusudan and M. Viswanathan. Query automata for nested words. In *MFCS*, volume 5734 of *LNCS*, pages 561–573. 2009.

[20] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *ACM PODS*, pages 283–294. June 2005.

[21] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *International Conference on Database Technology*, volume 4353 of *LNCS*, pages 254–268. 2007.

[22] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over XML streams. In *ACM SIGMOD*, pages 253–264. 2012.

[23] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *TCS*, 364(3):338–356, 2006.

[24] D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. on Know. Data Eng.*, 19(7):934–949, 2007.

[25] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *23rd IEEE International Conference on Data Engineering*, pages 236–245, 2007.

[26] W3C. Requirements and Use Cases for XSLT 2.1, 2010. http://www.w3.org/TR/xslt-21-requirements/.

Appendix

We present the pseudocode for the evaluation functions for **for** loops. `for_global` listens to events $(\bullet, v, \text{ALIVE})$ events from XSM and creates record templates. `eval_for` is called when we reach the **for** loop while unfolding the program. It instantiates record templates with the context and returns the new record. `for_event` is called in subsequent steps and it manages the creation or deletion of sub-records.

```
// let T be a global list of loop body templates

procedure for_global(node v, state S)
  if S == REJECT then
    remove the loop body for v from T, if it exists
  else // S == ALIVE
    // S cannot be SELECT, as we do not select
    // an incomplete candidate

    D = (x=v)
    R = evaluation of the loop body in D
    put R into T
end

function eval_for(node v, environment D)
  W = new empty working buffer
  if u_v == 0 then
    u_v = 1
    run the evaluation of XSM for current event
    store the notifications from XSM in L
  else
    // someone else has already run the XSM
    // the notifications are waiting for us in L
  end

  for each notification l in L do
    let l = (v, v', S)
    if S != REJECT then
      if v == v' then
        D' = D&x=v
        R' = evaluation of the loop body in D'
      else
        R' = loop body for v' from T
        instantiate missing variables in R from D
      end
      put R' in W
  end

  R = ((→), for)
  move the selected prefix of W to the buffer of R
  store the environment D in R
  let for_event listen to v-events from XSM
  return R
end

procedure for_event(node v', state S)
  if S == REJECT then
    if v' == •
      mark R as completed
    else
      remove record for v' from W
  end
  if S == ALIVE or S == SELECT then
    D' = D&x=v'
    R' = evaluation of the loop body in D'
    put R' in W
  end
  move the selected prefix of W to the output buffer
end
```