Streaming XML transformations using term rewriting

Alain Frisch
INRIA Rocquencourt
Alain.Frisch@inria.fr

Keisuke Nakano
Department of Mathematical Informatics
University of Tokyo
ksk@mist.i.u-tokyo.ac.jp

Abstract

This paper describes XStream, a Turing-complete programming language which allows the programmer to write XML transformations in a functional tree-processing style and have them evaluated in a streaming way: the output is produced incrementally while the input is still being parsed. The programmer does not need to care explicitly about buffering. We introduce the language, describe some techniques used in the implementation and present some performance results.

1. Introduction

This paper describes XStream, a small functional programming language for XML transformations. The XStream compiler produces very efficient code which computes the result and produce the output while the input XML document is still being parsed. This processing model, often referred to as streaming evaluation, has obvious advantages when dealing with big documents or when combining many transformation in a pipeline. Even for documents which would fit in memory, interleaving XML parsing, computation and output is a way to improve the use of three different resources (input channel, processing unit, output channel).

XStream differs from other attempts to streaming XML transformation on several points:

- XStream is Turing-complete and based on the very general formalism of term-rewriting, which makes it easy to translate transformations written in a purely functional style to XStream. As a matter of fact, the λ-calculus and ML-like pattern matching of trees can be directly translated to the core XStream language (the prototype support this features natively). Previous work focused on languages with reduced expressive power, such as XPath [1, 6, 7], a subset of XQuery [15], attributed tree transducers [17], or macro-forest transducers [18]. Notable exceptions include streaming XQuery processors such as FluXQuery [11] or the BEA/XQRL engine [3].
- XStream doesn't require any explicit hint from the programmer to enable streaming. This must be contrasted with e.g. the STX language [25] which let the programmer define and use buffers, with the explicit buffering primitives from [12], and of course with XML stream processors written by hand in a general purpose language. However, it should be noted that different ways to express the same transformation in XStream can yield different steaming behaviors, so in some situations, the programmer might need to understand how XStream works. The good thing is that the behavior of XStream runtime engine is easy to explain.
- XStream does not try to reject transformations which require a large fragment of the input to be available to compute just a small fragment of the output. Instead, it adopts a best effort approach to ensure a real-time behavior (discard the input and compute the output as soon as possible) but it does not *enforce* it.

  The rationale is that some transformations can behave reasonnably on the specific inputs which occur in practice and we don't want to reject them. For instance, a transformation that computes some kind of table of contents and appends it after the input document formally requires an unbounded amount of memory, but one expects the table of contents to be much smaller than the document itself.

  A typical representative of the opposite approach is the work by Kodama, Suenaga and Kobayashi [12]. They propose a static analysis (formulated as a type system based on ordered linear types) to check that a transformation written in functional style actually processes the input in a linear way.
- XStream does not decide statically which parts of the input must be buffered and which parts can be treated in streaming. The rationale is that this choice can depend on the context where the code is used and on the specific inputs to be processed. Suenaga, Kobayashi and Yonezawa [26] propose to extend [12] with a static analysis to insert explicit buffering primitives around parts of the program which cannot be dealt in a purely streamable way. Because these primitives can only materialize a whole subtree of the input, it is not possible to interrupt bufferization and go back to streaming in the middle of the subtree (e.g. because the information collected so far is enough to determine that the subtree will be completely discarded); also, it does not allow to start computing on the subtree before it has been completely parsed (the computation unit remains under exploited). The same comments apply to all query engines based on a statically elaborated query plan, such as FluXQuery [11] or the BEA/XQRL Streaming XQuery Processor [3]. Instead, XStream never (even at runtime) makes a binary decision between materializing a subtree or processing it purely in streaming; the evaluation engine is responsible for deciding dynamically which part of the input can be discarded.

Overview of the paper Section 2 gives an overview of the XStream language, its syntax, high-level semantics, and streaming behavior. Section 3 describes more formally the syntax and semantics for a simplified version of the XStream source language, which is based on the well-known notion of term rewriting.

```
type ev =
  | Open of string | Close of string | End

let rev_iter f l = List.iter f (List.rev l)

let rec buffer buf bufs = match next() with
  | Open "b" as ev ->
      rev_iter (rev_iter out)
        ((ev::buf)::bufs);
      copy ()
  | Open "a" as ev ->
      buffer [ev] (buf::bufs)
  | Close "a" as ev ->
      (match bufs with
        | [] -> copy ()
        | buf::bufs -> buffer buf bufs)
  | ev -> buffer (ev::buf) bufs
and copy () = match next() with
  | End -> ()
  | Open "a" as ev -> buffer [ev] []
  | ev -> out ev; copy ()
```

Listing 1. A hand-written implementation of a transformation

Section 4 introduces evaluations strategies which make streaming processing possible: incremental parsing, incremental evaluation, incremental pretty-printing. Section 6 describes concrete implementation techniques used in our XStream compiler. Section 5 shows step by step how XStream would evaluate the transformation used as a running example in Section 2. Section 7 compares the performance of XStream with other XML transformation technologies (CDuce and various implementations of XQuery, XSLT and STX). In Section 8 we propose possible extensions which we would like to work on in the future. In Section 9, we give a detailed comparison between XStream evaluation model and lazy evaluation. In Section 10, we compare XStream with other approaches proposed to stream XML transformations. XStream is distributed as open source software. It can be downloaded from `http://gallium.inria.fr/~frisch/xstream`.

2. An overview of XStream

A hand-written streaming transformation XML streaming transformers can be written by hand. This is usually done using an XML parser which returns a stream of XML events (or tokens) describing sequentially the structure of the input XML document (the most important events are: opening tag, closing tag, textual content). The SAX API [22] is a well known interface for event parsers in push mode (the parser is given callbacks to be called for each event), but event parsers in pull mode (the parser provides a function to fetch the next event) are usually easier to work with.

To show the benefits of programming with XStream, we will first consider a hand-written implementation of an streaming transformation and then show an equivalent XStream implementation. Listing 1 is an example of a hand-written transformation expressed in OCaml. This program assumes that the input XML document is well-formed. The library function `next` and `out` are used to fetch from the input and send to the output one XML event (the `next` function would be provided by an XML pull parser). The function `copy` is the main entry point; it simply copies the input stream to the output until the opening tag for an `<a>` element is found. The control is then passed to another function `buffer`, which keeps track of a stack of buffers. Each buffer is represented as a list of events and the stack is represented as a list of buffers (its topmost element being distinguished).

```
if(true(),x,_) | if(false(),_,x) -> x
or(true(),_) | or(_,true()) -> true()
or(false(),x) | or(x,false()) -> x

hasb(b[_] _) -> true()
hasb(%t[e1] e2) when << t <> "b" >> ->
  or(hasb(e1),hasb(e2))
hasb(()) -> false()

main(a[e1] e2) ->
  let q = main(e2) in
  if(hasb(e1),a[main(e1)] q, q)
main(%t[e1] e2) when << t <> "a" >> ->
  %t[main(e1)] main(e2)
main(()) -> ()
```

Listing 2. An XStream implementation of the same transformation

The readers are invited to figure out by themselves what the code from Listing 1 does. The answer is given in Listing 2 which is a functional specification of the same transformation, and also an example of an XStream script.

XStream scripts Before explaining Listing 2, we will give a quick overview of the syntax of XStream scripts. A script is a sequence of rewriting rules which collectively define a term rewriting system. The terms are finite trees constructed over a signature made of:

- XML element constructors, written `a[_] _` where `a` is an XML tag; the first argument of the constructor represents the children of the XML element and the second argument represents its folllowing siblings. In other words, XStream represents XML fragments using a classical binary encoding.
- The empty sequence, written `()`.
- A unary symbol **main**(_), which behaves as the entry point of the transformation.
- Other symbols defined implicitly by their occurence in some rewriting rule. They are used to represent both intensional computations (function names) and intermediate results. The script in Listing 2 creates two constant symbols `true()` and `false()`, one unary symbol `hasb(_)`, one binary symbol `or(_,_)` and one ternary symbol `if(_,_,_)`.

Any XML document (and any sequence of XML elements) can be encoded with only the first two kinds of symbols. (In this simplified presentation, we omit textual content and XML attributes.)

The rewriting rules in an XStream script have the form `p -> e` where `p` is a pattern and `e` is an expression. In their simplest form, patterns and expressions are simply terms with variables (the set of variable of `e` being contained in the set of variables of `p`), but some extensions are implemented: or-patterns (separated by `|`), let-expressions, the `%x` notation used to capture arbitrary tag names and guard expressions to add side constraints to rules.

An XStream script denotes an XML transformation. Conceptually, if the input XML document is encoded in a term τ, then the output is obtained by starting from the term **main**(τ) and then applying the rewriting rules from the script until a normal form is obtained (and this normal form must be the encoding of an XML document).

Back to the example Now we can explain the meaning of the script in Listing 2. The `if` symbol is the usual ternary operator; the

`or` symbol implements the Boolean disjunction; the `hasb` symbol checks whether a sequence of XML elements contains an sub-element `<b>`; the **main** symbol performs a deep copy of a sequence of XML elements but discards any sub-element of tag `<a>` which does not contain a sub-element of tag `<b>`.

The XStream script is a functional description of a transformation which deletes any `<a>` element without a `<b>` descendant. Which behavior does one expect from a streaming implementation of this transformation? As long as the input stream does not contain an opening tag event `<a>`, one can copy it verbatim to the output. This is what the `copy` function from Listing 1 does. While parsing the content of the `<a>` element, one needs to keep the input in a buffer because one doesn't know yet if it has to be copied to the output or discarded. As soon as one encounters an opening tag event `<b>`, one can copy the buffer to the output and go back to the original copy mode. Otherwise, if we reach the closing tag event `</a>`, it means that the corresponding XML element `<a>` has no `<b>` descendant. The corresponding part of the buffer must be discarded. In case of nested `<a>` elements, only part of the buffer is concerned. That's why it is necessary to keep a stack of buffers: each buffer corresponds to one of the nested `<a>` elements. If we reach the closing tag event `</a>` corresponding to the opening tag that initiated the buffering, we can go back to the original copy mode. All this logic is implemented in the `buffer` function.

It should be clear that the transformation as implemented in Listing 1 exhibits the optimal streaming behavior: it keeps in memory only the part of the input which is necessary for the rest of the computation and it produces parts the output incrementally as soon as possible. Writing the kind of code by hand, even for a transformation with such a simple specification, is very difficult and maintaining it is a nightmare: a small change in the specification can force a complete rewrite of the implementation. As a matter of fact, a previous version of this paper contained (unintentionally) a wrong implementation of the transformation.

Optimality The XStream compiler takes the source code in Listing 2 and produces a program which also exhibits the optimal steaming behavior. We don't claim any general optimality result (and we don't even propose a formal notion of optimality). Indeed, imagine we had used the following rewriting rules instead of the ones in Listing 2 for the symbol `or`:

```
or(true(),false()) |
or(true(),true()) |
or(false(),true()) -> true()

or(false(),false()) -> false()
```

The functional specification of the transformation would define the same transformation, but the code produced by XStream would keep in memory the whole content of an `<a>` element to determine whether it must be copied or discarded, even if a `<b>` element is found early (unless it appears as an immediate children of the `<a>` element).

The programmer must somewhat be informed on how XStream works in order to write efficient code. The following rule of thumb seems to be enough in practice: write rules that can be triggered as soon as possible, even when some parts of the input terms are yet unknown.

In the example above, the alternative definition of `or` is less efficient than the one in Listing 2 because it waits for its two arguments to be fully evaluated (to a Boolean value). When one of the argument is known to evaluate to `true()`, it is not necessary to continue the evaluation of the other argument. Failing to detect this case is inefficient because of two reasons: (i) some parts of the output could be delayed if they depend on the result of the `or` operator; (ii) the computation for the other argument takes time even if its result is useless. In addition, failing to simplify `or(false(),x)` to `x` (and the symmetric rule) is space-inefficient.

How XStream works The idea behing XStream is pretty simple. The high-level description of the semantics of an XStream script that we mentioned earlier consists in parsing the input document into a term x (that is, computing a substitution which maps the variable x onto a term which encode the input document) and then reducing the term $\mathbf{main}(x)$ until reaching a normal form. The code produced by XStream does the same, but incrementally. It maintains a current term which is initially $\mathbf{main}(x)$ and it reduces it periodically. Initially, no information is known about x, but parsing the input incrementally discovers the structure of x, thus triggering more reduction steps. When parsing is over, the content of x is fully known and the reduction can proceed until a final normal form. In parallel to this incremental parsing and evaluation, parts of the final result are extracted from the root of the current term, sent to the output, and discarded from memory.

This process is described more formally in the following sections and illustrated on our running example in Section 5.

3. The specification language

We are now going to describe formally the syntax and semantics of (a simplified version of) the XStream language. XStream scripts are purely functional programs written in term-rewriting style. For this simplified presentation, we will assume that XML documents are only made of elements. In particular, we don't consider attributes or textual content, though they are supported by the implementation.

Let us consider the terms defined by the following grammar:

$$\tau ::= () \mid \mathtt{a}[\tau]\tau \mid x \mid \sigma^n(\tau_1, \ldots, \tau_n)$$

The meta-variable x ranges over an infinite set of variables. The meta-variable $\mathtt{a}$ ranges over XML tags (to simplify, we assume here that the set of XML tags is finite). The meta-variable σ^n ranges over a set of symbols of arity n (defined by each XStream script). Each symbol σ^n can be thought either as a data constructor or as a function name. We assume that a distinguished unary symbol **main** always exists.

The term $()$ denotes the empty sequence. The term $\mathtt{a}[\tau_1]\tau_2$ denotes the XML fragment starting with an element of tag $\mathtt{a}$, with content τ_1, and followed by τ_2. Any term built only with these two constructions is said to represent an XML document. Any XML document (without attributes or textual content) can be represented by such a term. Note that our notion of XML document allows several root elements (unlike the official XML specification).

A substitution Φ is a finite mapping from variables to terms. We write $\tau\Phi$ for the application of a substitution Φ to a term τ (variables which are not in the domain of Φ are left untouched).

We write $\mathrm{FV}(\tau)$ for the set of free variables of a term τ. A *rewriting rule* is a pair of terms, written $\tau_1 \to \tau_2$ where τ_1 is of the form $\sigma^n(\ldots)$ and such that $\mathrm{FV}(\tau_2) \subset \mathrm{FV}(\tau_1)$. Let $r = (\tau_1 \to \tau_2)$ be a rewriting rule. We write $\tau \xrightarrow{r} \tau'$ if there exists a substitution Φ such that $\tau = \tau_1\Phi$ and $\tau' = \tau_2\Phi$. Note that "function names" such as **main** are allowed to appear nested in left-hand sides. This can be useful to implement high-level optimizations (e.g. `rev(rev(x)) -> x`). In reality, there is absolutely no distinction between symbols used as function names and symbols used as data structures.

A *script* is a finite set of rewriting rules. Let $P = \{r_1, \ldots, r_n\}$ be a script. We write $\tau \xrightarrow{P} \tau'$ if there is some $1 \leq i \leq n$ such that $\tau \xrightarrow{r_i} \tau'$. We write $\overset{P}{\Rightarrow}$ for the closure of $\xrightarrow{P}$ under context (in words: this relation allows reduction to take place in arbitrary position, not only at the root of terms). We write $\tau \overset{P}{\rightsquigarrow} \tau'$ if τ' can be reached

from τ by the reflexive and transitive closure of $\overset{P}{\Rightarrow}$ and τ' is a normal form under $\overset{P}{\Rightarrow}$.

The semantics of a script P on a term τ, written $P(\tau)$ is the set of normal forms for τ (terms τ' such that $\tau \overset{P}{\rightsquigarrow} \tau'$) , plus the element $\bot$ if there exists an infinite $\overset{P}{\Rightarrow}$-derivation starting from τ).

A script is *valid* if, for any term τ which represents an XML document, the set $P(\mathbf{main}(\tau))$ contains only terms representing XML documents (and maybe $\bot$). Our current implementation of XStream assumes that the script is valid without trying to check this property.

A *correct implementation* of a valid script P is a program which, given an XML document represented by τ, produces an XML document represented by some element of $P(\mathbf{main}(\tau))$; the program is allowed not to terminate on the input τ only if $\bot \in P(\mathbf{main}(\tau))$.

Syntactic sugar Listing 2 illustrates some additional sugar on the syntax of scripts. Though the implementation supports the extended syntax directly, it is possible to explain the new constructions conceptually in terms of the simplified language:

- Or-patterns are allowed in left-hand sides. We can eliminate an or-pattern by pushing it to the toplevel of the left-hand side, and then duplicating the rule. E.g. the rule `f(a()|b(),x) -> g(x)` is equivalent to `f(a(),x)|f(b(),x) -> g(x)` and can be decomposed to the two rules `f(a(),x) -> g(x)`, `f(b(),x) -> g(x)`.
- Wildcards are allowed in left-hand side. A wildcard, written as `_`, behaves just like a variable which is not used in the right-hand side.
- The notation $\%\mathtt{t}[\tau_1]\tau_2$ in a rule means that this rule should be duplicated once for each possible value of the tag $\%\mathtt{t}$. We restrict this duplication by putting some constraints on the values for $\%\mathtt{t}$, e.g. $\%\mathtt{t} <> \text{"}\mathtt{a}\text{"}$.
- Let-bindings in right-hand side are eliminated by textual substitution.

It is also possible to add two higher-level features to the specification language:

- A pattern matching `match` τ `with` $[\tau_\mathtt{i} \rightarrow \tau'_\mathtt{i}]_{1\leq \mathtt{i} \leq \mathtt{n}}$ in a right-hand side is translated to $\sigma^{\mathtt{k}+1}(\tau, \mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k})$ where $\{\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k}\} = \bigcup_\mathtt{i} \mathrm{FV}(\tau'_\mathtt{i}) \backslash \mathrm{FV}(\tau_\mathtt{i})$ and σ^{k+1} is a fresh symbol subject to the following rewriting rules ($1 \leq i \leq n$): $\sigma^{\mathtt{k}+1}(\tau_\mathtt{i}, \mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k}) \rightarrow \tau'_\mathtt{i}$
- Similarly, a lambda abstraction `fun` $[\mathtt{x} \rightarrow \tau]$ is translated to $\mathtt{f}^\mathtt{k}(\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k})$ where $\{\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k}\} = \mathrm{FV}(\tau) \backslash \{\mathtt{x}\}$ and $\mathtt{f}^\mathtt{k}$ is a fresh symbol. We consider a unique symbol **apply** of arity 2, and for each such λ-lifting, we add a rewriting rule $\mathbf{apply}(\mathtt{f}^\mathtt{k}(\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k}), \mathtt{x}) \rightarrow \tau$.

Since XStream is based on term rewriting (and can thus encode the λ-calculus), it is clear that the language is Turing-complete.

4. Evaluation strategies

We will now describe several implementation strategies, that is, ways to derive correct implementations from scripts. In the following, we thus assume that P is given. In particular, we will introduce incremental parsing and incremental pretty-printing as two optimizations over a naive global evaluation strategy.

4.1 Global evaluation

The naive global implementation strategy closely follows the semantics of the specification language. It consists in fully parsing the input XML document into an in-memory term τ, reducing $\mathbf{main}(\tau)$ into a normal form (with an arbitrary reduction strategy), and then pretty-printing this normal form into an XML document.

4.2 Incremental evaluation

We now assume that the input XML document is obtained through an XML parser which produces a stream of events. Events are described by the following grammar:

$$\epsilon ::= \texttt{<a>} \mid \texttt{</>}$$

We don't need to keep the tag of the closing events. We also use `</>` to represent the end of the document. For instance, parsing the XML document `<a><b></b></a><c></c>` will produce the sequence of events `<a>`, `<b>`, `</>`, `</>`, `<c>`, `</>`, `</>`.

Incremental parsing From a stream of events, it is possible to reconstruct the XML document as a tree. We will formalize this reconstruction process as the composition of basic substitutions, which give incremental knowledge about the input document. First, we introduce as notion of parsing stack, which stores a list of variables:

$$S ::= x :: S \mid []$$

A parsing stack keeps track of which variables will receive the parts of the document which remain to be parsed (one variable for each level in the nested structure of XML corresponding to the current node being parsed). Then we define a function `Parse` which maps a pair of a parsing stack and a parsing event into a pair of new parsing stack and a substitution. We write $(x_0 := \tau)$ for the substitution which maps x_0 to τ. The function `Parse` is defined by:

$$\begin{array}{lcl} \mathtt{Parse}(x_0 :: S, \texttt{<a>}) & = & (x_1 :: x_2 :: S, (x_0 := \mathtt{a}[x_1]x_2)) \\ \mathtt{Parse}(x_0 :: S, \texttt{</>}) & = & (S, (x_0 := ())) \end{array}$$

We assume that the variables x_1 and x_2 are fresh. Let $(\epsilon_i)_{1\leq i\leq n}$ be the sequence of events produced by the XML parser on a well-formed input XML document. We start with a stack $S_0 = x_0 :: []$ and we define $(S_i, \Phi_i) = \mathtt{Parse}(S_{i-1}, \epsilon_i)$ for $1 \leq i \leq n$. Then we have $S_n = []$ and the term $x_0\Phi_1 \ldots \Phi_n$ represents the input XML document.

Incremental evaluation This incremental reconstruction of the input XML document gives a strategy for evaluating of the transformation while parsing the input. Instead of fully parsing the document into a variable x_0 and then normalizing the term representation instead of $\mathbf{main}(x_0)$, we can instead parse the input incrementally, apply the corresponding substitution and normalize the current term (which is $\mathbf{main}(x_0)$ initially) after each event.

Formally, the state of the streaming transformer is described by a pair (τ, S) of a term in normal form and a parsing stack (which always contains all the free variables of τ and maybe other variables). We write $(\tau, S) \overset{\epsilon}{\rightarrow} (\tau', S')$ if $(S', \Phi) = \mathtt{Parse}(S, \epsilon)$ and $\tau\Phi \overset{P}{\rightsquigarrow} \tau'$.

Again, consider a sequence of events $(\epsilon_i)_{1\leq i\leq n}$ produced by the XML parser on a well-formed input XML document. If we have a sequence $(\tau_i, S_i)_{0\leq i\leq n}$ with $S_0 = x_0 :: []$, $\tau_0 = \mathbf{main}(x_0)$ and $(\tau_{i-1}, S_{i-1}) \overset{\epsilon_i}{\rightarrow} (\tau_i, S_i)$, then the final result τ_n is a correct result (an element of $P(\mathbf{main}(x_0\Phi_1 \ldots \Phi_n))$). If the construction of the sequence stops at some step (τ_i, S_i) because of a non-terminating derivation, then $P(\mathbf{main}(x_0\Phi_1 \ldots \Phi_n))$ contains $\bot$. These properties comes from the following commutation lemma:

Lemma 1. *If $\tau \overset{P}{\Rightarrow} \tau'$, then $\tau\Phi \overset{P}{\Rightarrow} \tau'\Phi$ for any substitution Φ.*

To summarize, the incremental evaluation strategy is as follow. Start from a fresh variable x_0, an initial parsing stack which contains only x_0, and the initial term $\mathbf{main}(x_0)$. For each parsing event, update the parsing stack, rewrite the current term by applying

the substitution given by `Parse`, and normalize this term. This description leaves the normalization strategy itself unspecified. Note that contrary to the global evaluation strategy, we are now reducing terms with free variables.

Discarding the input When the printing stack contains a variable which is not free in term of the current state, this means that the fragment of the input corresponding to this variable will simply be discarded. One can exploit this property and avoid building and applying the useless substitution corresponding to the parsing event which arrives when this variable is in head position on the stack. In theory, we could even inform the XML parser that a fragment of the input must be ignored; the parser could enter a special mode in which it discard the input efficiently without producing any event and without building strings to store attributes and tag names. The parser described in [6] exposes such an interface.

4.3 Incremental pretty-printing

The incremental evaluation strategy can be combined with an incremental pretty-printer. The idea is to produce parts of the output as soon as they become available. We can thus start sending the result e.g. to the network early, and also release some memory.

To formalize the incremental pretty-printer, we introduce a notion of printing stack, defined by the following grammar:

$$O ::= \tau :: O \mid []$$

From a printing stack, we can extract printing events [1] in head position. We consider the relation $O \xrightarrow{\epsilon} O'$ defined by:

$$\begin{array}{lll} () :: O & \xrightarrow{\texttt{</>}} & O \\ \texttt{a}[\tau_1]\tau_2 :: O & \xrightarrow{\texttt{<a>}} & \tau_1 :: \tau_2 :: O \end{array}$$

A printing state is a pair (s, O) where s is the sequence of already printed events (which are not actually kept in memory) and O is a printing stack. We write $(s :: \epsilon)$ for a sequence extended with a new event. We consider the following *extraction* relation on printing states: $(s, O) \rightarrow (s :: \epsilon, O')$ if $O \xrightarrow{\epsilon} O'$. We write $\texttt{Print}(s, O)$ for the unique normal form for (s, O) with respect to extraction.

We can extend the notions of substitution and of reduction (for a given script P) from terms to printing stacks.

We can now describe an implementation strategy which combines incremental evaluation and incremental pretty-printing. The state of the streaming transformer is described by a triple (s, S, O) where s is a sequence of printed events, S is a parsing stack, and O is a printing stack. Let $(\epsilon_i)_{1 \leq i \leq n}$ be the sequence of events produced by the XML parser. A sequence of states $(s_i, S_i, O_i)_{0 \leq i \leq n}$ is correct with respect to $(\epsilon_i)_{1 \leq i \leq n}$ if it is such that:

- Initialization:
  - $\textbf{main}(x_0) :: [] \overset{P}{\rightsquigarrow} O'_0$.
  - $(s_0, O_0) = \texttt{Print}([], O'_0)$.
  - $S_0 = x_0 :: []$.
- For each parsing event $1 \leq i \leq n$:
  - $(S_i, \Phi_i) = \texttt{Parse}(S_{i-1}, \epsilon_i)$.
  - $O_{i-1}\Phi_i \overset{P}{\rightsquigarrow} O'_i$.
  - $(s_i, O_i) = \texttt{Print}(s_{i-1}, O'_i)$.

In other words, the evaluation proceeds as follow. We start from an empty sequence of printed events, a parsing stack S_0 which contains only the variable x_0 and a printing stack which contains only the term $\textbf{main}(x_0)$. Before reading the first parsing event, we normalize this printing stack to obtain a new printing stack O'_0. At this point, some prefix of the output might already be available. It is collected in s_0, and the remaining stack is O_0. When an parsing event ϵ_i arrives, we do the following. First, we update the parsing stack and obtain a substitution Φ_i which represents the new information held by the event. We apply this substitution to the printing stack O_{i-1} and normalize it with respect to the rewriting rules of the script P in order to obtain a new printing stack O'. Finally, we extract from this stack new events in head position.

When parsing is over, the parsing stack S_n is empty (if the input XML document is well-formed). If the script P is valid, the transformer must have computed a complete XML document (or maybe one of the normalization steps did not terminate). As a consequence, the printing stack O_n is also empty and the output s_n is a linear representation of a correct result for P.

Correctness of this implementation strategy relies on two facts:

- Substitution and reduction on printing stacks commute with extraction.
- Because rules in scripts cannot rewrite XML elements (and the empty sequence $()$), we know that parts of the output which are extracted won't need to be substituted nor normalized.

Let us develop the second point above. If we allowed the script made of the two rules `main(x) -> x` and `a[()] y -> y`, then the behavior of XStream would be to copy an opening tag event `<a>` immediatly to the output when parsed (because the term `a[x1] x2` is a normal form), and it would thus transform the document `<a/>` into itself, even if it not a normal form for the rewriting system. Disallowing rules whose root symbol is an XML constructor prevents such a case.

Stopping the evaluation When the printing stack is empty, the computation is over. This might happen well before the end of parsing, if the transformation does not need the rest of the input.

5. An example, step-by-step

Figure 1 shows the successive steps of the evaluation of the XStream program given in Listing 2 on the input XML stream `<a><c><b></></><d></></></>` (the last event indicates the end of the stream).

Each line corresponds to one event in the input stream (the first line represents the state before parsing begins). The input event is shown in the first column, the induced substitution in the second column and the parsing stack in the third column.

The fourth column is the most interesting one. It represents the printing stack. To make the notation shorter, we use `m` and `h` instead of `main` and `hasb`, we expand the let notation, we omit `()`. Also we represent the whole stack and the already output part as a single term. For instance, the term $\underline{\texttt{a[c[b}}\texttt{m(x}_5\texttt{)]m(x}_6\texttt{)]m(x}_4\texttt{)]m(x}_2\texttt{)}$ denotes the printing stack made of the four terms $\texttt{m(x}_5\texttt{)}, \texttt{m(x}_6\texttt{)}, \texttt{m(x}_4\texttt{)}, \texttt{m(x}_2\texttt{)}$; the fragment $\underline{\texttt{a[c[b[}}$ is what has been sent to the output (the corresponding events are shown in the last column) and discarded from memory.

We can observe that while parsing the content of the `<a>` element and before the `<b>` opening tag event has been seen, the input is kept in an inner part of the current term. This corresponds intuitively to a notion of buffer. Reducing the `if` symbol either discards the buffer or makes it visible at the root of the current term (and thus allows it to be sent to the output and discarded from memory).

[1] We use the same notion of events as for the incremental parsing, that is, we forget about the tag of closing events. It is easy to keep this information in another stack, or to modify our formalization of the pretty-printer stack to keep track of these tags, between terms in the stack.

Input	Substitution	Parsing stack	Current term	Output
		$\mathtt{x}_0 :: []$	$\mathtt{m}(\mathtt{x}_0)$	
<a>	$\mathtt{x}_0 := \mathtt{a}[\mathtt{x}_1]\mathtt{x}_2$	$\mathtt{x}_1 :: \mathtt{x}_2 :: []$	$\mathtt{if}(\mathtt{h}(\mathtt{x}_1), \mathtt{a}[\mathtt{m}(\mathtt{x}_1)]\mathtt{m}(\mathtt{x}_2), \mathtt{m}(\mathtt{x}_2))$	
<c>	$\mathtt{x}_1 := \mathtt{c}[\mathtt{x}_3]\mathtt{x}_4$	$\mathtt{x}_3 :: \mathtt{x}_4 :: \mathtt{x}_2 :: []$	$\mathtt{if}(\mathtt{or}(\mathtt{h}(\mathtt{x}_3), \mathtt{h}(\mathtt{x}_4)), \mathtt{a}[\mathtt{c}[\mathtt{m}(\mathtt{x}_3)]\mathtt{m}(\mathtt{x}_4)]\mathtt{m}(\mathtt{x}_2), \mathtt{m}(\mathtt{x}_2))$	
	$\mathtt{x}_3 := \mathtt{b}[\mathtt{x}_5]\mathtt{x}_6$	$\mathtt{x}_5 :: \mathtt{x}_6 :: \mathtt{x}_4 :: \mathtt{x}_2 :: []$	$\mathtt{a}[\mathtt{c}[\mathtt{b}[\mathtt{m}(\mathtt{x}_5)]\mathtt{m}(\mathtt{x}_6)]\mathtt{m}(\mathtt{x}_4)]\mathtt{m}(\mathtt{x}_2)$	<a><c>
	$\mathtt{x}_5 := ()$	$\mathtt{x}_6 :: \mathtt{x}_4 :: \mathtt{x}_2 :: []$	$\mathtt{a}[\mathtt{c}[\mathtt{b}[\,]\mathtt{m}(\mathtt{x}_6)]\mathtt{m}(\mathtt{x}_4)]\mathtt{m}(\mathtt{x}_2)$	
</c>	$\mathtt{x}_6 := ()$	$\mathtt{x}_4 :: \mathtt{x}_2 :: []$	$\mathtt{a}[\mathtt{c}[\mathtt{b}[\,]\,]\mathtt{m}(\mathtt{x}_4)]\mathtt{m}(\mathtt{x}_2)$	</c>
<a>	$\mathtt{x}_4 := \mathtt{a}[\mathtt{x}_7]\mathtt{x}_8$	$\mathtt{x}_7 :: \mathtt{x}_8 :: \mathtt{x}_2 :: []$	$\mathtt{a}[\mathtt{c}[\mathtt{b}[\,]\,]\mathtt{if}(\mathtt{h}(\mathtt{x}_7), \mathtt{a}[\mathtt{m}(\mathtt{x}_7)]\mathtt{m}(\mathtt{x}_8), \mathtt{m}(\mathtt{x}_8))]\mathtt{m}(\mathtt{x}_2)$	
	$\mathtt{x}_7 := ()$	$\mathtt{x}_8 :: \mathtt{x}_2 :: []$	$\mathtt{a}[\mathtt{c}[\mathtt{b}[\,]\,]\mathtt{m}(\mathtt{x}_8)]\mathtt{m}(\mathtt{x}_2)$	
	$\mathtt{x}_8 := ()$	$\mathtt{x}_2 :: []$	$\mathtt{a}[\mathtt{c}[\mathtt{b}[\,]\,]\,]\mathtt{m}(\mathtt{x}_2)$	
</>	$\mathtt{x}_2 := ()$	$[]$	$\mathtt{a}[\mathtt{c}[\mathtt{b}[\,]\,]\,]()$	</>

Figure 1. Step-by-step evaluation of the example transformation

6. Implementation

We have implemented a compiler for XStream which produces an executable from a script describing an XML transformation. The script in evaluated with incremental parsing and pretty-printing. In this section, we describe some aspects of the implementation.

The XStream compiler currently uses Objective Caml as a back-end: from a script, it produces OCaml source code which is then compiled by the OCaml native compiler. The resulting programs use the Expat parser (`http://expat.sourceforge.net/`). The only OCaml feature which is really piggybacked is the pattern matching compiler. In particular the code produced by XStream does not use higher-order functions and it explicitly manages the memory. It would thus be quite feasible to use a C back-end instead.

A tricky point of the implementation is to deal with the fact that OCaml programs are not allowed to recurse very deeply (except for tail calls) because they use the system stack to implement function calls. A solution which completely avoids non-tail calls (such as continuation-passing style or trampolines) would induce a large overhead for some basic operations. We currently use a hybrid technique which consists in keeping a counter that store the current depth of recursion (an approximation of stack usage) and to turn to a trampoline technique when this counter reaches a fixed value. We found out that the overhead compared to normal function calls is small and that we can still deal with very deep recursion.

6.1 Representation of terms

A first naive idea would be to implement the algebra of terms by functional trees. However, we need to be able to find variables, detect redexes, substitute a new term. In order to support these operations efficiently, we use a imperative graph data-structure. The graph is global: it stores all the terms which appear in the printing stack. It is basically a DAG with multiple roots. A node which represents of symbol of arity n comes with a tuple of n ordered children (downlinks). The DAG representation creates some sharing in the terms. We maintain the invariant that each variable appears at most once in the graph (variables can thus be identified by node identity).

When an inner node X is replaced with a node X' by the incremental parser or by a rewriting rule, all the parents of X must be informed. We thus keep for each node X a list of pairs (Y, i) (uplinks) where Y is a parent node and i is the position of i amongst the children of Y. These uplinks makes it possible to redirect (by in-place assignment) the i-th downlink of Y from X to X'.

The parsing stack stores a sequence of variables. We simply represent it as the list of the corresponding nodes in the graph. It is thus immediate to find the node to be substituted when the next parsing event arrives.

When a node is no longer referenced, it can be destroyed. Note that a children of such a node can still be referenced (by another node, or because it is in the parsing stack). Because of uplinks, the node to be destroyed is thus formally accessible from alive data, and so we cannot rely on a garbage collector to free the node. We could use a mechanism of weak-references, but this is tricky: we would have to destroy the uplinks pointing to this node when the node is actually garbage collected, and so we would need finalizers as well. This becomes rather heavy. Also we prefer to destroy nodes as soon as possible, without having to wait for the next GC pass. Indeed, destroying a node might prevent doing useless computation. We thus choose to manually destroy nodes when they are no longer referenced. Because of uplinks, we know if there is any parent node. We also keep an additional reference count to avoid destroying roots of the graph (during the evaluation of right-hand sides of rewriting rules, the freshly created nodes are not linked to the rest of the graph).

When a node is destroyed, one must remove the uplinks from all its children (and check whether this children can in turn be destroyed). In the current implementation, the set of uplinks of a node is represented as a singly linked list. To remove an uplink, we must thus traverse such a list. We could use a doubly linked list instead, with a technique similar to the one described in [23], but the overhead induced by the extra structure would probably makes this solution slower in practice.

6.2 Applying rewriting rules

The core of the evaluation process consists in normalizing the terms in the printing stack. Concretely, one needs to detect a node X where a rewriting rule could apply, produce fresh nodes corresponding to the right-hand side (with a root node X'), and replace X with X'.

Because of sharing, a single reduction step might actually simulate many reduction steps in the terms.

When we replace a node X by a new node X', this might create new redexes. These potential redexes are necessarily found close to X. Let d be the maximal depth of left-hand sides of the script, where we define the depth of a term as the maximal number of nested symbols and constructors. E.g. a left-hand side $\sigma^2(x, \mathtt{a}[y]z)$ has depth 2. If $d = 2$, then a redex can only be rooted at an immediate parent of X. In general, a redex can only appear rooted at k-ancestor of X with $k \leq d - 1$. (We define a 1-ancestor as an immediate parent and a $(k+1)$-ancestor as an immediate parent of a k-ancestor.) The compiler also compute which symbols can trigger rewriting rules (simply by collecting all the non-root symbols from left-hand sides). We could use more refined heuristics but we found the current solution acceptable.

We maintain (in a stack) the set of nodes which need to be inspected and check them in turn. When this set is empty, the graph

has been normalized. Checking whether the node is a redex and computing the corresponding right-hand side basically amounts to ML pattern matching. Our implementation directly uses the pattern matching compiler from OCaml. Since OCaml supports or-patterns, we don't actually expand them and instead we treat them directly.

OCaml enforces a first-match policy for pattern matching and XStream inherits this property. This makes it possible to optimize some patterns. For instance, the guards in the rules of Listing 2 could be eliminated.

6.3 Functional evaluation

It is often the case that a node freshly constructed during the evaluation of a right-hand side can be immediately reduced. In order to avoid the overhead of substitution in the graph, we check whether a node creation would produce a redex. More precisely, when the right-hand side of a rule is evaluated, the symbols (node constructors) are interpreted as function calls. That is, instead of producing a fresh node in the graph, we first evaluate the arguments and then check whether one of the rewriting rules could be applied to the node that would be created. If this is the case, we proceed recursively, as would be the case for a normal strict implementation of a functional language. If no rule can be applied, the node is really created (and it does not need to be marked as a potential redex). In order to avoid a stack overflow, we maintain in a counter the depth of recursion; when this counter reaches a fixed limit, we also stop recursion and create a node in the graph (and mark it as a potential redex).

The optimization mentioned in the previous paragraph is made even more effective by the following observation. It is not necessary to normalize the term between each parsing events. Instead, one can wait for a fixed number of parsing events and only start the normalization process. (The current prototype waits for 20 events.) The effect is to delay a little bit the computation and the output with respect to parsing events, but the overhead of manipulating nodes in the graph is significantly reduced.

6.4 Non-linear patterns, maximal sharing

Our description of the specification language does not restrict the left-hand sides of rewriting rules to be linear. When a variable appear several times, the semantics is to check the equivalence of the corresponding subterms (as usual in term rewriting). Currently, our implementation does not support non-linear patterns. We could support it simply by adding guards to check deep equivalence. Another strategy would be to keep the invariant that equivalent nodes are physically equal (a.k.a. maximal sharing).

Maximal sharing allows not only to evaluate non-linear patterns efficiently, but it can also use less memory to store the graph. Also, sharing nodes does not only mean sharing data, but also computations.

Maximal sharing of immutable terms built in a bottom-up way is classically done by hash-consing. The same technique can actually be used in our setting (graph with in-place modification), and it is only slightly trickier. Here is how to do so. Each node is given a unique identifier. When a fresh node is created and assuming that its children have already been maximally shared, a simple lookup in a hash-table is enough to find whether there already exists in the graph an equivalent node (same symbol, same children). When a node X is replaced in the graph by a node X', one must update the entry in the hash table corresponding to the parents of X (because the unique identifier for X' is not the same as for X), and doing so, we might detect that some parents are actually equivalent to other nodes, thus producing more sharing. This process is propagated up in the DAG.

Some preliminary tests showed that the overhead induced by this technique is high, and are not compensated by the gains, except for specially crafted examples. Also, we believe that non-linear patterns are not so important, so we simply decided not to support them.

We mentioned that the specification language can be extended with a `let x` $=$ τ_1 `in` τ_2 construction in right-hand sides of rewriting rules. It can be seen as just syntactic sugar for the term τ_2 where X has been replaced with τ_1. Our implementation supports the construction directly so as to preserve the trivial sharing it induces when `x` is used several times in τ_2.

6.5 Data, attributes and the interface with OCaml

In our formalization of the specification language, our algebra has a single kind (terms). We thus said that there is one constructor of arity 2 $\mathtt{a}[\_], \_$ for each XML tag, and that rules which inspect the tags must be duplicated once for each possible tag. In reality, there is an infinite number of possible XML tags and we also want to deal with strings and maybe integers or other basic types to define interesting computations.

XStream lets the programmer declare explicitly some symbols together with their signatures. Each argument can be either a term (that is, concretely, a node in the graph) or a value of any OCaml type (which does not mention the term type). For instance, the constructor for XML elements could has been defined as (but it is actually built-in):

```
declare elt(string,
            <<(string*string) list>>,_,_)
```

which means that the first argument is a string, the second is a list of pair of strings (to represent the XML attributes of the element) and the two other arguments are terms. The `<<...>>` notation is used to introduce syntactically OCaml types (we can omit these delimiters for simple types such as `string` or `int`). XStream allows OCaml expressions and patterns to be used in place of these arguments with the same `<<...>>` notation (the delimiters can be omitted around variables and literals). The notation `a[t1],t2` is then just syntactic sugar for `elt`$(\mathtt{"a"}, [], \mathtt{t1}, \mathtt{t2})$, and `%x[t1],t2` really is `elt(x,[],t1,t2)` (the empty list [] is replaced by a wildcard in pattern position). Similarly, `elt("a",x,t1,t2)` can be written `a[@x t1] t2`. The guards on Listing 2 are interpreted as OCaml guard when rules are translated to OCaml pattern matching by the compiler.

In order to support textual content in XML, there is another built-in declaration:

```
declare str(string,_)
```

which represents an XML fragment starting with some text (and followed by another XML fragment, that is, a term).

Listing 3 shows an example of a transformation which uses XML attributes and integers. This transformation returns a single element whose content is a string representing the n-th XML tag found in the input, in document order, where n is the value of the XML attribute `n` of the root element. Of course, the program produced by XStream for this transformation stops as soon as this tag has been parsed.

7. Benchmark

A paper on a language with a new evaluation scheme would not be complete without a benchmark section. This section thus compares the performance of XStream with other technologies for XML transformation.

Transformation specification We choose a specific transformation task which has been previously used as a benchmark between

```
declare nth(int,_)

nth(0,str(t,_)) -> str(t,())
nth(n,str(_,l)) when << n > 0 >> ->
  nth(<< n - 1 >>,l)

tags(%t[e1] e2,q) ->
  str(t,(tags (e1,tags(e2,q))))
tags((),q) -> q
tags(str(_,e),q) -> tags(e,q)

main(_[@attr _]_ as x) ->
  let i =
    <<int_of_string (List.assoc "n" attr)>>
  in
  a[nth(i,tags(x,()))] ()
```

Listing 3. Attributes and integers

CDuce [2] and XSLT [2]. The XStream code is given in Listing 4. The input document is assumed to store a database of the descendants of a number of persons (the root of the document is a `<doc>` element). A person is described as an XML element:

```
<person gender="...">
  <name>...</name>
  <children>...</children>
</person>
```

The gender attribute is either `"F"` or `"M"` according to whether the person is a woman or a man. The `<name>` sub-element contains a single string. The `<children>` sub-element contains one `<person>` element for each child of the person. The task is to transform such an element into:

```
<woman name="...">
  <sons>...</sons>
  <daughters>...</daughters>
</woman>
```

or into `<man>...</man>` according to the gender of the person. The name is moved from a sub-element into an attribute; the gender information is moved from an attribute into a tag name; the children are split according to their gender and transformed recursively.

Tools We wrote the same transformation in the CDuce [2], XSLT [28], XQuery [27] and STX [25] languages. For these four languages, we tried several implementation variants and picked the most efficient one for each tool. All these implementations, and all the tools needed to reproduce the benchmark, can be found in the XStream distribution.

We used four well established XSLT engines: Xalan C++ version 1.10, the XSLTC compiler version 1.4 from the Xalan-Java project (an XSLT-to-Java bytecode compiler), Saxon version 8.7.3 (a Java-based implementation), and the `xsltproc` tool built above the Gnome project's libxml/libxslt libraries (written in C).

We used two Java-based implementations of XQuery which are reputed to be efficient: Qizx/open version 1.1 and the XQuery engine from Saxon version 8.7.3.

STX is a one-pass, event-oriented transformation language for XML documents, with explicit buffering primitives (see Section 10 for a comparison with XStream). We used Joost version 2006-05-05 which is the most efficient STX interpreter available. Joost is implemented in Java.

We used the CDuce compiler version 4.1, which is implemented in OCaml. This compiler produces some kind of intermediate code which is then evaluated by an OCaml interpreter.

Protocol We generated random XML documents of various sizes. They consist of a long sequence of toplevel `<person>` elements, each one containing a number of descendant (the maximum nesting depth of `<person>` elements is 6). For each implementation and each input document, we ran the transformation five times, measured the wall-clock time and took the geometrical mean over the three executions. We excluded compilation time for XStream, CDuce and XSLTC (the other implementations are interpreters). The machine used for this benchmark is a Pentium 4 2.80Ghz with 1Gb of RAM. The Java Virtual Machine (used by Qizx/open, Saxon, Xalan) is Sun's J2RE version 1.5.0 with maximum Java heap size set to 512Mb. Saxon was run with the `-pull` option which gave some noticeable speedup.

We also measured the maximum memory size used by each implementation. To do this, we fetched every second the `VmRSS` field (resident set size) from the pseudo file $/\texttt{proc}/pid/\texttt{status}$ (this is the same information as returned by the `top` command). We ran this benchmark separately from the time measurement in order not to disturb it.

Streaming behavior Of course, XStream is able to process the toplevel `<person>` elements one by one in a streaming fashion. But it does more: within a toplevel `<person>` element, the male children are also processed on the fly, and so are their own male children, and so on.

We were not able to obtain the same behavior with STX. The problem is that the transformation requires one STX buffer at each nesting level (to store the daughters). The STX version used for the benchmark (which is available from XStream's website) has been specialized to deal with bounded-depth documents (depth 6) and the buffering code need to be duplicated for each level. Another version which only tried to stream the toplevel elements (and not their male children) turned out to be much slower.

Results The throughput results are given in Figure 2 in Mb per second (higher is better) with respect to the size of the input. A star indicates that the process was killed either by the OS or by the JVM. The memory results are given in the same figure.

Comments All the Java-based implementations suffers from the JVM initialization costs and the just-in-time compilation. In a realistic framework, the Java code would be preloaded so this runtime overhead would not matter. The tools which are interpreters (and not compilers) usually start by doing some form of internal compilation. This also explain slow startup behavior.

XStream and Joost/STX are the only implementations which use a bounded amount of memory and which can deal with very large XML documents. The most important difference between XStream and STX, except the speed difference, is that XStream allows the programmer to write the transformation in a tree-processing style, whereas STX makes explicit the use of buffers and requires code duplication.

Reverse transformation Some transformations cannot benefit from streaming at all. A typical example is a transformation that reverses the order of top-level elements (children of the root). In order to evaluate the overhead of XStream in a worst-case situation, we ran our benchmark on such a transformation. The source code for the various implementations are given in Listing 5. Note that the XQuery and XSLT 2.0 (Saxon) versions use a built-in reverse operator. The results are given in Figure 3 and 4. Since the transformation require the whole document to be loaded in memory, the tools which can deal better with large documents are those with the more compact in-memory representation of XML documents

```
(* Get elements with a specific tag *)
declare extract(string,_,_)
extract(t, %s[@a x] y,q) when <<s=t>> -> %s[@a x] extract(t,y,q)
extract(t,_[_] y,q) -> extract(t,y,q)
extract(_,(),q) -> q

split(person[@a name[str(n,_)] children[c] _] y,q) ->
  let tag = << match List.assoc "gender" a with "M" -> "man" | _ ->"woman" >> in
  let a' = << ["name",n] >> in   (* The new attribute *)
  let c = split(c,()) in         (* Transform recursively *)
  let s = extract("man",c,()) in (* Split according to gender *)
  let d = extract("woman",c,()) in
  %tag[ @a' sons[s] daughters[d] ] split(y,q)
split(str(_,y),q) -> split(y,q)
split((),q) -> q

main(doc[x] _) -> doc[split(x,())] ()
```

Listing 4. The transformation used as benchmark

Throughput in Mb/s

input size:	1Mb	2Mb	5Mb	10Mb	20Mb	40Mb	80Mb	160Mb	320Mb
XStream	4.45	5.30	6.12	6.01	6.25	6.47	7.24	7.23	7.25
CDuce	3.38	3.45	4.43	4.24	4.06	3.74	3.35	2.61	⋆
Saxon (XQuery)	0.31	0.63	1.03	1.25	1.41	1.51	1.58	⋆	⋆
Qizx/open (XQuery)	0.51	0.76	1.21	1.30	1.43	1.50	1.53	⋆	⋆
XSLTC (XSLT)	0.85	1.41	2.48	3.05	3.41	3.18	2.72	0.88	⋆
Saxon (XSLT)	0.42	0.83	1.53	1.94	2.29	2.55	2.76	⋆	⋆
xsltproc (XSLT)	2.13	2.54	3.43	2.47	1.59	1.04	0.60	⋆	⋆
Xalan (XSLT)	1.07	1.23	1.43	1.19	1.08	0.68	0.46	0.24	⋆
Joost (STX)	0.35	0.39	0.51	0.53	0.55	0.54	0.57	0.56	0.60

Maximum memory in Mb

10Mb	20Mb	40Mb	80Mb
1.0	1.0	1.0	1.0
38.7	69.5	165.8	348.6
76.8	134.1	258.8	499.8
65.8	112.4	205.9	398.7
57.3	102.3	215.0	440.8
79.9	134.7	249.3	496.9
109.5	216.3	430.6	852.8
42.1	77.9	149.4	290.2
16.7	16.7	16.7	16.7

Figure 2. Benchmark results

(CDuce and Xalan). The benchmark suggests two improvements to XStream memory representation: element names could be hash-consed (as in e.g. CDuce), and XML fragments which are fully computed don't require to keep upward pointers.

Conclusion Of course, we should try many more different transformations before drawing any definitive conclusion from the benchmark. XStream will behave well for transformations which it can evaluate really in streaming, keeping very little of the document in memory. Even when memory usage is not a key criterion (medium-sized documents), XStream performs relatively well.

We did not try to tweak any of the garbage collector parameters for the Java- and OCaml-based implementations. Clever choices could give huge speed-ups.

It is unfair to compare implementations with different source languages and different target/implementation languages. Also, it should be noted that XStream's XML data model is simpler than the one used by XSLT or XQuery implementation (e.g. no pointer to the parent element, no namespace). Still, the benchmark suggests that XStream is a promising approach to deal with large XML documents.

8. Extensions, future work

Concatenation XStream's XML data model currently relies on a binary encoding of XML trees. An XML fragment can be either the empty sequence, an XML element with a tag, a sequence of children and a sequence of right-siblings, or some text with a sequence of right-siblings. Modulo syntactic sugar, the data model is pre-defined by the following declarations:

```
declare elt(string,
            <<(string*string) list>>,_,_)
declare str(string,_)
declare nil()
```

Note that this data model does not natively support concatenation of XML fragments. One can of course define a concatenation symbol with the following rules:

```
concat((),e) | concat(e,()) -> e
concat(%t[@a e1] e2,e) ->
   %t[@a e1] concat(e2,e)
concat(str(s,e2),e) -> str(s,concat(e2,e))
```

(The pattern `concat(e,())` is just an extra optimization.) As it well known by functional programmers, this implementation of concatenation gives a quadratic behavior when sequences are used to accumulate results on the right. An alternative is to consider `concat` as uninterpreted symbol (except maybe for simplification rules dealing with empty sequences.) Then, of course, all rules for other symbols which deconstruct XML must be extended to deal with the concatenation. (A variant would be in addition to remove the last argument of the `elt` and `str` symbols. Of course, the translation from one data model to the other can be implemented by the programmer.) A good situation is when concatenation nodes don't need to be deconstructed (the pretty-printer could be extended to deal with concatenation natively.)

Another solution is to get rid of concatenation by adding extra "queue" arguments, when possible. A typical example is the `tags`

```
(* XStream version *)
rev(%t[x] y, a) -> rev(y, %t[x] a)
rev(%t y, a) -> rev(y, %t a)
rev((), a) -> a
main(doc[x] _) -> doc[rev(x,())] ()

(* CDuce version *)
type T = [ (AnyXml | Char)* ]
let rev ((T,T) -> T)  ([],q) -> q  |  ((hd,tl),q) -> rev (tl,(hd,q))
match src with <doc>l -> dump_xml_utf8 <doc>(rev (l,[]))

(* XQuery version *)
<doc>{fn:reverse(./doc/node())}</doc>

(* XSLT 1.0 version *)
<xsl:template match="/doc">
 <doc>
  <xsl:for-each select="child::node()">
  <xsl:sort select="position()" data-type="number" order="descending"/>
  <xsl:apply-templates select="." mode="copy"/>
  </xsl:for-each>
 </doc>
</xsl:template>
<xsl:template match="node()" mode="copy">
 <xsl:copy>
  <xsl:apply-templates select="child::node()" mode="copy"/>
 </xsl:copy>
</xsl:template>

(* XSLT 2.0 version *)
<xsl:template match="doc">
 <doc>
  <xsl:copy>
   <xsl:copy-of select="reverse(node())"/>
  </xsl:copy>
 </doc>
</xsl:template>
```

Listing 5. The reverse transformation

input size:	1Mb	2Mb	5Mb	10Mb	20Mb	40Mb	80Mb	160Mb	320Mb
XStream	3.76	3.84	4.88	4.64	4.54	4.24	3.85	⋆	⋆
CDuce	1.98	2.39	3.38	3.62	3.50	3.48	3.44	2.99	2.34
Saxon (XQuery)	0.74	1.09	1.86	2.29	2.78	3.13	3.44	⋆	⋆
Qizx/open (XQuery)	0.54	1.10	1.81	2.11	2.18	2.09	⋆	⋆	⋆
XSLTC (XSLT 1.0)	0.70	1.27	2.34	2.96	3.34	3.07	2.65	1.00	⋆
Saxon (XSLT 2.0)	0.60	1.06	1.96	2.47	3.09	3.56	3.84	⋆	⋆
xsltproc (XSLT 1.0)	1.82	3.18	4.18	4.14	4.22	4.25	⋆	⋆	⋆
Xalan (XSLT 1.0)	0.39	1.06	1.35	1.19	1.00	0.73	0.44	0.23	⋆

Figure 3. Benchmark results for the reverse transformation: throughput in Mb/s

input size:	10Mb	20Mb	40Mb	80Mb
XStream	71.7	144.5	272.7	540.8
CDuce	32.3	60.2	116.3	226.6
Saxon (XQuery)	76.3	133.6	248.3	477.3
Qizx (XQuery)	89.8	167.6	312.2	529.5
XSLTC (XSLT 1.0)	58.0	103.5	215.6	442.5
Saxon (XSLT 2.0)	76.8	134.1	248.8	498.5
xsltproc (XSLT 1.0)	122.2	241.5	480.7	⋆
Xalan (XSLT 1.0)	42.2	78.1	149.7	290.5

Figure 4. Benchmark results for the reverse transformation: maximum memory in Mb

```
main(%t[e1]e2,e)|main(e,%t[e1]e2) ->
  %t[e1] main(e1,e)
main((),e) | main(e,()) -> e
```

Listing 6. Merging multiple inputs with arbitrary interleaving

```
main(%t[e1] e2, %s[e3] e4) ->
  %t[e1] %s[e3] main(e2,e4)
main((),e) | main(e,()) -> e
```

Listing 7. Merging multiple inputs with fixed interleaving

symbol from Listing 3. It would be interesting to perform such a transformation automatically.

Finally, it might be possible to get some inspiration from purely functional implementation of sequences with efficient concatenation[20] in order to derive a clever treatment of concatenation. It is not clear, however, how to deal in this setting with sharing and imperative updates in terms.

Multiple inputs It is possible to extend our formalism to deal with multiple inputs. We assumed that scripts define a distinguished unary symbol **main**. In a script with n inputs, this symbol would be of arity n. An example of a transformation with two inputs is given in Listing 6. This scripts takes two XML streams and merge the toplevel elements, with arbitrary interleaving [2]. Another example is given in Listing 7. Here, the two XML streams are merged in a synchronized way.

To implement a script with n inputs, we need to run n XML parsers simultaneously and to maintain n parsing stacks. The tricky point is to schedule the n parsers. We can simply run them concurrently and rely on the system's process/thread scheduler; when an event is made available by one of the parser, the corresponding parsing stack is updated and the unique printing stack is normalized (one would probably need a global lock to avoid race-conditions). Such a scenario works well for a script such as in Listing 6. However, for the script in Listing 7, it is desirable to ensure that the toplevel elements of two input streams are read at a comparable rate. Otherwise, the one which is read faster needs to be buffered in memory. If the input streams come from the network, or simply if the size of the toplevel elements are arbitrary, then it is better if the main transformer loop is in charge of scheduling the different parsers. A scheduling heuristics must choose on which input stream to accept the next event. A strategy could be to wait for one event on each stream and choose the one which would minimize the size of the printing stack, after normalization and extraction. We leave to future work the precise design and study of such heuristics.

Controlling the reduction strategy The main evaluation loop of an XStream program consists in normalizing the printing stack. Our current implementation does not try to be clever when it chooses which redex to reduce next. It just picks one from the current set of nodes which need to be inspected. One could imagine adding some heuristics or some programmer-specified annotations to improve the normalization process. Let us consider for instance the `if` symbol as defined in Listing 2. It it is a good idea to reduce the first argument, and then try to apply the rewriting rules for `if` before looking at the other ones. Indeed, if the first argument reduces to either `true`() or `false`(), one can immediately destroy one the two other arguments. For many other symbols, however, there is no point trying to reduce an instance of the symbol before reducing its arguments. The literature on lazy evaluation and strictness analysis might give interesting starting points to design good heuristics for driving the normalization process.

Dynamic instrumentation XStream programs store suspended computations as data. Currently, data is only inspected by rewriting rules and by the pretty-printer. However, the reified aspect of computation opens the door to various kind of dynamic instrumentation. For instance, it is trivial to support checkpointing (the ability to stop a program, store its current state, and restart it later, maybe on a different machine). Instead of storing the whole computation, it is possible to detect dynamically some parts which are stuck (e.g. parts of the result which can not yet be extracted to the output stream) and to store them in order to free some memory. The extra knowledge about the computation could make it possible to do a better job than the OS virtual memory manager.

Encoding existing XML transformation languages into XStream XStream could be used as a back-end for higher-level XML transformation/programming languages, such as XDuce [8, 10], CDuce [2, 5], XSLT [28] or XQuery [27]. In particular, we are planning to investigate in a future work the details of the translation from CDuce to XStream. The CDuce and XDuce languages rely on regular expression patterns [9]. Patterns are usually compiled to tree automata, using static type information as an optimization [4, 13, 14]. It is then straightforward to translate automata into XStream. For instance, consider the following CDuce fragment:

```
match e with [ (x:: <a>_ | _)* ] -> f x
```

The semantics of the pattern is to extract in a new sequence all the elements of tag `a`. This CDuce fragment could be compiled to the XStream fragment `f`(`p`(`e`)) where the new fresh symbol `p` is subject to the following rules, directly obtained from an automaton version of the pattern:

```
p(a[x1] x2) -> a[x1] p(x2)
p(%t[_] x) when << t <> "a" >>  -> p(x)
p(()) -> ()
```

As another example, consider the CDuce fragment:

```
match e with t -> e1 | _ -> e2
```

where `t` is some CDuce type. This pattern matching can be compiled to the XStream fragment `if`(`q`(`e`), `e1`, `e2`) where `q` is a symbol subject to rewriting rules which mimics the automaton associated to `t` (possibly taking static type information about `e` into account as described in [4, 13, 14]). As the name of the current section suggests, we leave the details of the translation from regular expression patterns to XStream for a future publication.

9. Comparison with lazy evaluation

Our evaluation scheme is somewhat reminiscent of lazy evaluation. Lazy computation is pulled by the output: when some part of the output must be printed, the functional expression is reduced enough to compute this part. This might induce, as a side effect, some of the input to be read. The effect is roughly similar to the behavior of an XStream program, but the evaluation of an XStream program is instead pushed by the input. In a sense, XStream is eager: between two parsing events, the functional expression is reduced up to a normal form.

[2] In practice, the interpreter will commit to pick one element from one the two inputs as soon as a root tag is available. An alternative implementation (which the interested reader can try to write as an exercise) would require a whole element to be available before committing to copying it. Also, this transformation is an example of a non deterministic XStream script (that is, a non-confluent rewriting system) which could be useful also in the setting of a single input, e.g. as a library function to merge two sequences, when the choice of interleaving does not matter. Hard-coding a specific interleaving strategy gives less opportunity to get a good streaming behavior.

Let us consider the two requirements for streaming that we mentioned in the introduction: greediness and memory-awareness. We will show that XStream sometimes beats a lazy evaluator on both these criteria.

First, consider the term `or(hasb(x),hasa(x))` where `or` and `hasb` are defined as in Listing 2 and `hasa` is defined as `hasb`, replacing `b` by `a`. A lazy evaluator would choose to reduce first one of the two arguments of the `or` operators into a head normal form, which should be `true()` or `false()`. Imagine that `hasb(x)` is reduced first and that the input document (bound to `x`) contains no `b` element, but an `a` element close to the beginning. In this case, the lazy evaluator will have to parse the whole input, whereas XStream would only parse it until the first `a` element. This example shows that for some situations, the eager approach taken by XStream will beat any sequential evaluator (be it lazy or strict) in terms of greediness.

Now, consider the following script:

```
main(x) -> a[x] last(x)
last((%t[e1] ()) as x) -> x
last(_[_] (_[_] as x)) -> last(x)
```

The transformation basically consists in duplicating the last toplevel element from the input. XStream would evaluate this script as expected: the input is copied on the fly and the program uses a bounded amount of memory. A lazy evaluator, instead, would start evaluating the expression `a[x]`, thus forcing the parsing of the input in `x`. The input would indeed be copied on the fly to the output, but the `last(x)` expression would not be reduced before the end of the input. Since the lazy evaluator would keep a reference to the whole input, its memory usage would be linear in the size of the input. In this example, XStream beats a lazy evaluator in terms of memory-awareness. (For a more realistic scenario, replace `last` by a symbol which computes e.g. a table of contents.)

The advantage of the incremental reduction strategy over the global evaluation strategy is that some computation can be interleaved with the actual parsing of the input XML document. It should be noted, though, that the incremental strategy might globally induces more computation than the global strategy (or a lazy evaluator). As an example, consider a script defined by `main(x) -> if (hasb(x),f(x),g(x))` where `f` and `g` yield some complex computation, and `if`, `hasb` are as in Listing 2. In the global semantics with a call-by-name reduction strategy, one could choose to compute first a normal form for `hasb(x)`, and only compute one of `f(x)` or `g(x)` according to whether this normal form is `true()` or `false()`. With the incremental strategy, one would start to compute both `f(x)` and `g(x)`, and throw away the computation for `g(x)` only if some `<b>` is found. Note that is it not necessarily bad to do more computation since this might allow to use less memory. This would be the case e.g. if the (intermediate) results of `f(x)` and `g(x)` are significantly smaller than `x`.

Note that even with the incremental strategy, it is possible to suspend manually some computation. Indeed, instead of using the ternary `if` symbol as above, the programmer can write `match hasb(x) with[true() → f(x)|false() → g(x)]`, which is just syntactic sugar for $\sigma^2(\texttt{hasb(x)}, \texttt{x})$ where σ^2 is a fresh symbol subject to the two rewriting rules $\sigma^2(\texttt{true()}, \texttt{x}) \rightarrow \texttt{f(x)}$ and $\sigma^2(\texttt{false()}, \texttt{x}) \rightarrow \texttt{g(x)}$. The computation of `f(x)` and `g(x)` is suspended until the result of `hasb(x)` is available.

10. Related work

XStream evaluates in streaming XML transformations given by functional specifications. In this section, we mention other works with the same goal.

While plenty of work has been devoted to the automatic derivation of XML stream processors from declarative programs, most of them deals with query languages, such as XPath [1, 6, 7] and a subset of XQuery [15]. They are not expressive enough to describe some simple transformations such as the structure-preserving transformation renaming all the labels `a` to `b`, which can be expressed naturally in recursive functional style.

The second author previously proposed XTiSP [17], an XML transformation language intended for stream processing. XTiSP programs can be translated into attributed tree transducers (att), and then composed with an XML parser also expressed as an att, using an original composition method [16]. In a later work [18], he proposed to base XTiSP instead on macro-forest transducers (mft) [21] as a model of XML transformation, thus improving the expressive power of the language. Macro-forest transducers are a generalization of top-down tree transducers with accumulators. They can be seen as a subclass of XStream scripts. In an mft, only the first argument of each symbol can be inspected; the other arguments can only be copied to the output. The left-hand side of a rewriting rule can only be of the form $\texttt{f}(\texttt{a}[\texttt{x1}]\texttt{x2}, \texttt{y1}, ..., \texttt{yk})$ or $\texttt{f}((), \texttt{y1}, ..., \texttt{yk})$. The right-hand side of a rule can call other symbols, but only with the first argument being `x1` or `x2`. It is well-known that the expressive power of macro-forest transducers is not closed under composition (unlike XStream). A pushdown machine is proposed in [19] as an implementation strategy for streaming mfts. This idea of using a pushdown is similar to our formalization of incremental parsing with a stack. The advantage of our presentation is to clearly dissociate incremental parsing from the actual reduction engine.

Compared to [18], this paper is based on simple term rewriting, which is more uniform, more expressive, and better known than macro-forest transducers. As a consequence, it does not impose constraints which might seem ad hoc from the point of view of a functional programmer. In particular, XStream naturally supports higher-order functions, deep pattern matching (see Section 3 for an encoding of these features), and Booleans (see e.g. the `if` and `or` symbols in Listing 2) without any special treatment. This paper also details implementation techniques and expands on the comparison with lazy evaluation.

STX [25] is a one-pass, event-oriented transformation language for XML documents. The language is based on a notion of templates as XSLT, but instead of matching subtree, STX templates match and react to parsing events. STX uses different programming constructions for streaming and processing which require buffering (explicit primitives to create, populate and deconstruct buffers). Also, STX is really geared towards a forward processing model and does not allow composition nor look-ahead in the documents. Simple transformations requires the programmer to insert explicit opening/closing tags (with the risk of producing ill-formed XML documents). STX keeps the context of the currently processed node: the stack of ancestors and the position within siblings. This context can be simulated in XStream either by the control flow or by explicitly managed extra arguments.

Kodama, Suenaga and Kobayashi [12] propose a static analysis to check that a transformation written in functional style actually processes the input in a linear way (each node is accessed exactly once in document order), thus guaranteeing that the transformation can be trivially evaluated in streaming. In order to allow non-trivial transformations, the language is extended with primitives to materialize a whole subtree in memory and then work on it without any restriction. Suenaga, Kobayashi and Yonezawa [26] propose a system to automatically insert these buffering primitives where necessary. In this approach, any subtree will be processed either in a purely linear (streaming) way or in a completely buffered way. Moreover, this decision is made statically. This makes it impossible to deal efficiently with a transformation such as in Listing 2, where the decision to copy the input or to buffer it depends on the actual

input and is not scoped on a whole subtree (when a tag `<b>` is found under a `<a>`, it is not necessary to buffer the rest of the subtree rooted at the `<a>`). Let us give another example which would be evaluated efficiently by XStream and not by this approach: a transformation which appends to a document some kind of table of contents (e.g. the sequence of all the `<title>` elements found in the document). XStream would copy the input on the fly and only keep the table of contents in memory. The system from [26] would instead buffer the whole input, because it is not used strictly linearly.

FluXQuery [11] is an XQuery engine that optimizes query evaluation using schema constraints derived from DTDs. FluX, the internal algebra of FluXQuery, is an extension of XQuery with an explicit stream processor construction, built from event handlers. The FluXQuery query optimizer elaborates regular XQuery into FluX, using ordering and cardinality constraints from the DTD. This approach suffers from the same main drawback as [26]: a subtree is either processed sequentially or buffered, but the decision has to be done statically. The idea of using static type information to discover more streaming opportunities could be recasted in XStream as an optimization stage, or would, more realistically be applied during the compilation from a higher-language (such as XQuery, XSLT, or CDuce) into XStream. The CDuce pattern matching compiler actually follows this idea since it uses precise type information to produce efficient one-pass automata version of patterns; by translating these automata to XStream, one would get a type-optimized streaming implementation of patterns.

The BEA/XQRL processor [3] is another XQuery engine that attempts to pipeline XML processing. Its query optimizer generates a query execution plan which is a composition of iterators (stream transformers). The lack of a more detailled description and of an easily available standalone implementation prevents us from doing a more detailled comparison.

11. Conclusion

We have presented XStream, a Turing-complete purely functional language intended to write XML transformations. XStream allows the programmer to write a transformation in a tree-processing style and have it evaluated efficiently in streaming when this is possible. Our preliminary performance comparison indicates that the current prototype is already quite efficient and thus suggests trying to use XStream as a back-end for higher-order XML languages.

References

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ACM International Conference on Functional Programming (ICFP)*, 2003.

[3] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. In *VLDB*, 2003.

[4] A. Frisch. Regular tree language recognition with static information. In *IFIP Conference on Theoretical Computer Science (TCS)*, Toulouse, 2004. Kluwer.

[5] A. Frisch and the CDuce team. CDuce: User's manual, 2004. `http://www.cduce.org/manual.html`.

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

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

[8] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Data bas es (WebDB2000)*, 2000.

[9] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Journal of Functional Programming*, volume 13(4). 2002.

[10] H. Hosoya and B. C. Pierce. A typed XML processing language. In *ACM Transactions on Internet Technology*, volume 3(2), pages 117–148. 2003.

[11] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *VLDB*, 2004.

[12] K. Kodama, K. Suenaga, and N. Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. In *The Second ASIAN Symposium on Programming Languages and Systems (APLAS'04)*, 2004.

[13] M. Y. Levin. Matching automata for regular patterns. In *International Conference on Functional Programming (ICFP)*, 2003.

[14] M. Y. Levin and B. C. Pierce. Type-based optimization for regular patterns. In *First International Workshop on High Performance XML Processing*, 2004.

[15] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *VLDB*, pages 227–238, 2002.

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

[17] K. Nakano. An implementation scheme for XML transformation languages through derivation of stream processors. In *The Second ASIAN Symposium on Programming Languages and Systems (APLAS'04)*, 2004.

[18] K. Nakano. Streamlining functional XML processing. In *The First DIKU-IST Joint Workshop on Foundations of Software*, 2005.

[19] K. Nakano and S.-C. Mu. A pushdown machine for recursive XML processing. In *The Fourth ASIAN Symposium on Programming Languages and Systems (APLAS 2006)*, 2006.

[20] C. Okasaki. *Purely Functional Data Structures*. PhD thesis, Carnegie Mellon University, 1996.

[21] T. Perst and H. Seidl. Macro forest transducers. *Information Processing Letters*, 89(3):141–149, 2004.

[22] SAX; `http://www.saxproject.org/`.

[23] O. Shivers and M. Wand. Bottom-up β-reduction: uplinks and λ-dags. In *Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, 2005.

[24] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, dec 2005.

[25] Streaming Transformations for XML (STX); `http://stx.sourceforge.net/`.

[26] 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 Symposium on Logic-based Program Synthesis and Transformation (LOPSTR2005)*, 2005.

[27] XQuery 1.0: An XML Query Language; `http://www.w3.org/TR/xquery/`.

[28] XSL Transformations (XSLT); `http://www.w3.org/TR/xslt/`.