Transforming XML Streams with References

Citation for published version:
Maneth, S, Ordóñez, A & Seidl, H 2015, Transforming XML Streams with References. in C Iliopoulos, S Puglisi & E Yilmaz (eds), String Processing and Information Retrieval: 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9309, Springer International Publishing, pp. 33-45. DOI: 10.1007/978-3-319-23826-5_4

Digital Object Identifier (DOI):
10.1007/978-3-319-23826-5_4

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
String Processing and Information Retrieval

Transforming XML Streams with References

Sebastian Maneth[1], Alberto Ordóñez[2], and Helmut Seidl[3]

1 **School of Informatics, University of Edinburgh**
`smaneth@inf.ed.ac.uk`
2 **Database Laboratory, Universidade da Coruña**
`alberto.ordonez@udc.es`
3 **Institut für Informatik, TU München**
`seidl@in.tum.de`

Abstract

Many useful XML transformations can be formulated through deterministic top-down tree transducers. As soon as transducers process parts of the input repeatedly or in an order deviating from the document order, such transductions cannot be realized over the XML document stream with constant or even depth-bounded memory. Here we show that by enriching streams by *forward references* every such transformation can be compiled into a stream processor with a space consumption depending only on the transducer and the depth of the XML document. This is remarkable because tree transducers have rich restructuring capabilities and can process copies of input subdocuments independently. Also, references allow one to produce output in a compressed form which is guaranteed to be linear in the size of the input (up to the space required for labels). Our model is designed so that without decompression, the output may again serve as the input of a subsequent transducer. In order to reduce the extra overhead incurred by references, we investigate three optimizations: (i) reference reuse to save space, (ii) multi-labels to avoid chains of references and (iii) inlining to limit the use of references to deviations of the document order.

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases XML Stream; Bounded Memory; Tree Transducer

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

In many scenarios data arrives in a stream, e.g., sensor readings, news feeds, or large data that cannot fit in memory. If the streamed data is tree structured, such as XML, further challenges arise because documents have nesting-depth as well as width. One basic question is, which tree transformations can be computed with *constant memory*, i.e., memory only depending on the transformation but not on the size of the input stream. It turns out that even very simple transformations cannot be realized with constant memory, e.g., the transformation that removes all subtrees with a certain root label from a document adhering to a non-recursive DTD (see [24]). Therefore we consider a milder restriction: a tree transformation is *left-depth-bounded memory* (LDBM), if it can be computed with memory only depending on the transformation and on the *left-depth* of the input tree. Similar to the ordinary depth, the left-depth of a ranked tree is defined as the maximal length of a path from the root to a leaf. The only difference is that edges from nodes to their *right-most* (last) child are not counted. Thus, monadic trees have left-depth 0, ordinary lists have left-depth 1, and the left-depth of the binary tree representation of an XML document corresponds to the nesting-depth of the document. This is practically relevant since most XML documents are of small nesting-depth.

There are two fundamental limitations of LDBM tree translations:

- subtrees must be transformed in the order they arrive;
- subtrees may not be transformed multiple times (“copied”).

Conference title on which this volume is based on.

To see this, consider first the transformation that flips the order of two lists: for the input tree $\mathsf{root}(l_1, l_2)$ it outputs the tree $\mathsf{root}(l_2, l_1)$, where l_1, l_2 are arbitrary lists. The translation is not LDBM because the entire list l_1 must be stored in memory. For the same reason, the translation that copies a list l_1, i.e., on input $\mathsf{root}(l_1)$ outputs $\mathsf{root}(l_1, l_1)$, is not LDBM. Translations that flip or copy, frequently appear in practice. For instance consider the transformation which, when applied to a structured document b representing a book, first outputs a table of contents (obtained by extracting the list of section and sub-section headers in b), followed by b.

In this paper we propose to enrich streams by *forward references*. A forward reference is a pointer to a later position in the stream. It is given by a *label* which must be defined later in the stream. For the purpose of this paper, labels are considered as abstract data objects which only can be created and compared for equality. It is for ease of implementation only that we use plain numbers for representing labels. When stored in memory we count the occupied space for 1 — but keep in mind that as few labels should be introduced as possible. Using labels and references, a valid output for flipping the lists on input $\mathsf{root}(l_1, l_2)$ is this enriched output stream

$$\mathsf{root}(\mathsf{ref}\ 1, \mathsf{ref}\ 2)\ \ 2\!:\!l_1\ 1\!:\!l_2.$$

The translation generating this output can be performed in LDBM, because the order of the lists remains unchanged. Similarly, the copy of a list can be realized using two references to the same label, i.e., by outputting $\mathsf{root}(\mathsf{ref}\ 1, \mathsf{ref}\ 1)\ \ 1\!:\!l_1$ on the input $\mathsf{root}(l_1)$. In this case, the output represents a DAG-compressed representation of the corresponding tree.

Allowing many references on the other hand, also incurs an overhead, namely the extra space for the labels representing positions in the tree. Also, references to references allow to create arbitrarily large streams all representing the same small tree. Therefore, we ask: Which classes of tree translations are LDBM if output streams *and input streams* may include forward references? For a given translation, how many distinct labels are required and what is the space overhead caused by the forward references? Can it be avoided to output *reference chains* of the form $\mathsf{ref}\ 1 \ldots 1\!:\!\mathsf{ref}\ 2 \ldots 2\!:\!\mathsf{ref}\ 3 \ldots$? Our contributions are summarized as follows.

1. Any *deterministic top-down tree transducer* (*dtop*) can be transformed into an LDBM stream processor. The memory and the number of references in the output is bounded by the depth of the input, the transducer, and the number of references in the input. Thus, it is independent of the length m of the input stream. The size of the output is in $\mathcal{O}(m)$ (using $\mathcal{O}(m)$ many labels), even if the dtop produces exponentially many copies.
2. We show (for the same model) that the generation of reference chains can be avoided, at the expense of introducing *sets* of labels. The cardinality of those sets is bounded by the maximal sharing in the input and the maximal number of visits of input nodes by the transducer.
3. We identify sufficient conditions where the introduction of references can altogether be avoided. These conditions, e.g., are met by transducers which process every subtree only once and in left-to-right fashion.

We implemented a prototype and use it to experimentally evaluate the impacts of the different optimizations. One important outcome of the experiments is that the extra overhead introduced by references is quite moderate. Even when we run many different dtops in sequence, the output with references is typically below two times the size of the conventional output without references. The experimental results can be found in the Appendix.

Forward vs. Backward References and Garbage

The choice of dtops as class of tree transformations, and forward references as way of achieving LDBM, may seem rather ad hoc. In this section we indicate why these two choices are not only natural, but also (in a certain sense) maximal. Our decisions are directed by the two desires

- never to produce *garbage* (label definitions that are not referenced), and
- to be compositional, i.e., the output of one processor can serve as input to another processor.

A priori, we do not have a preference with respect to forward or backward references. In fact, even allowing both types of references is a viable choice. It is not difficult to see that for realizing the translation of a dtop, the output either may be represented by means of forward references as suggested so far, or alternatively by means of backward references. For instance, for the flip of two lists we can as well produce this stream with backward references, $1 : l_1 \ 2 : l_2 \ 0 : \mathsf{root}(\mathsf{ref}\ 2, \mathsf{ref}\ 1)$ where the label 0 represents the root of the output tree. But with such a representation, are we compositional? Consider a second dtop which translates $\mathsf{root}(l_1, l_2)$ into the tree $\mathsf{root}(l'_1, l_2)$ where l'_1 contains only every odd element of the list l_1. Let τ_{odd} denote this translation. Taking as input the stream displayed above, how can this transformation be realized? This *cannot* be done with LDBM, because each element of l_1 must be kept in memory (until we know where in the input relative to the root it appears). If on the other hand, garbage is allowed, then LDBM is possible by producing both translations l_1 and l'_1, and later inserting a reference to the correct one (leaving the other as garbage).

Fact 1: The dtop translation τ_{odd} cannot be realized in LDBM over streams with backward references (unless garbage is allowed).

We conclude that backward references in the input must be ruled out, if we want to handle arbitrary dtops. On the other hand, can we handle larger classes of transformations than dtops, by using forward references only (and no garbage)? Clearly, the addition of regular look-ahead is not possible: in some cases, it would require the generation of garbage for similar reasons as above. Let us review the two other important generalizations of dtops:

- deterministic top-down tree-to-string (dts) transducers, and
- left-to-right attribute grammars (LR-AG), seen as tree transducers.

LR-AGs have been used in the context of XML streaming [13] and already Bochman [26] showed that all attribute values of an LR-AG can be computed in one left-to-right pass through the input tree. Accordingly, LR-AGs seem a natural candidate in our context. Is it always possible to for an LR-AG to produce with LDBM output streams with forward references? As an example, consider the transformation which reverses a list, given as binary tree, i.e., which translates trees $r(a_1(\bot, a_2(\bot \ldots a_n(\bot, \bot) \ldots)), \bot)$ into $r(a_n(\bot, a_{n-1}(\bot \ldots a_1(\bot, \bot) \ldots)), \bot)$. This translation τ_{reverse} can be realized by an LR-AG: it uses one inherited attribute to compute the reverse of the a_i-path. This translation can also be realized by a dts transducers using these rules (for all possible list labels a):

$$\begin{array}{lcllcl} q_0(r(x_1, x_2)) & \rightarrow & r(q_0(x_1), \bot) & \qquad q(a(x_1, x_2)) & \rightarrow & q(x_2)a(\bot, \\ q_0(a(x_1, x_2)) & \rightarrow & q(x_2)a(\bot, \bot)) & \qquad q(\bot) & \rightarrow & \varepsilon \end{array}$$

Note that the right-hand sides are strings, i.e., "," and the parenthesis are ordinary letters. It should be clear that this translation *cannot* be realized in LDBM using forward references alone. Therefore, we have:

Fact 2: τ_{reverse} cannot be realized in LDBM using streams with forward references.

Facts 1 and 2 imply that LDBM streaming can neither be extended from dtops to dts transducers nor to LR-AGs without introducing backward references and thus loosing composability with dtops.

2 Streams with References

An XML document is modeled as a *ranked tree*, e.g., via the well-known first-child next-sibling (fcns) encoding (see [22]). The fcns encoding produces binary trees. Here we consider arbitrary ranked trees. What is important is that the last child of a node represents the next sibling. Thus, if an element's content consists of three lists, then the element should be represented by a node of rank four. In this way, a ranked tree transducer can change the order of such lists. See [16] where such encodings are called "DTD-based". For the rest of of the paper (except Appendix A) the details of the encoding are not relevant.

A *ranked alphabet* is a finite set Σ together with a mapping from Σ to the non-negative integers; the mapping associates to each symbol in Σ its *rank*. The rank of a symbol determines the number of children of nodes labeled by that symbol. Let Lab be an infinite set of *labels*. We require that the set Lab is equipped with a method new(), which, given a current active set of labels $L \subseteq \mathsf{Lab}$, returns some symbol in $\mathsf{Lab} - L$.

The set $\mathcal{S}_\Sigma$ of *streams* p (over Σ with forward references over Lab) consists of a tree, possibly followed by a sequence of definitions. Trees t, sequences s of definitions, and streams p are defined by the following grammar:

$$\begin{array}{lcl} t & ::= & a\,\overbrace{t\ldots t}^{k \text{ times}} \mid \mathsf{ref}\,l \\ s & ::= & \epsilon \mid L\!:\!t\ s \\ p & ::= & t\,s \end{array}$$

where $a \in \Sigma$ is of rank $k \geq 0$, $l \in \mathsf{Lab}$, $L \subseteq \mathsf{Lab}$ with $L \neq \emptyset$, and, if ref l occurs in s then $L:t$ occurs in s to the right for some t and L with $l \in L$. Thus, a stream is a sequence $L_1:t_1 \cdots L_n:t_n$ of *label definitions*. Note that label definitions $L:t$ only appear at the top-level of the stream and not inside of trees. Note further, that we do *not* use parentheses or commas in our streams. The *size* of a tree t, a sequence s, or a stream p counts the number of occurring symbols where a reference ref l counts for one and a label set L counts for the cardinality of L. The size function is denoted by $|\,.\,|$, i.e., vertical bars. A Σ-tree is a tree t without occurrences of ref l. The set $\mathcal{T}_\Sigma$ is the set of all *trees t over* Σ. The symbol ϵ denotes the empty sequence of definitions. When writing Σ-trees, we always will include parenthesis and commas for better readability. In examples, we syntactically represent the multiple labels in a set $L = \{l_1, \ldots, l_r\}$ of labels as a comma-separated list $l_1, \ldots, l_r$. In particular, a one-element set of labels $L = \{l\}$ is simply denoted by l. As an example, consider a ranked alphabet Σ containing the symbols a and b of rank 2 and the symbol e of rank zero; then

$$a \ \mathsf{ref}\ 1\ a\ \mathsf{ref}\ 2\ \mathsf{ref}\ 1\ \ 1\!:\!b\ \mathsf{ref}\ 2\ \mathsf{ref}\ 2\ \ 2\!:\!e$$

is a stream in $\mathcal{S}_\Sigma$ consisting of the tree t_{ex} followed by the sequence of definitions s_{ex} with

$$t_{\mathsf{ex}} = a\ \mathsf{ref}\ 1\ a\ \mathsf{ref}\ 2\ \mathsf{ref}\ 1 \quad \text{and} \quad s_{\mathsf{ex}} = 1\!:\!b\ \mathsf{ref}\ 2\ \mathsf{ref}\ 2\ \ 2\!:\!e.$$

It should be clear that this stream represents the tree $a(b(e,e), a(e, b(e,e)))$. Formally, we define a function dec_t which for a tree t and a mapping E ("environment") providing trees for labels referenced in t, returns a Σ-tree. This function is mutually recursive with the function decode which, for a sequence of definitions s and an environment E, returns a mapping which provides a Σ-tree for each label defined in s or E. These functions are defined as follows:

$$\begin{array}{lcl} \mathsf{dec\_t}(\mathsf{ref}\ l, E) & = & E(l) \\ \mathsf{dec\_t}(a\ t_1 \ldots t_k, E) & = & a(\mathsf{dec\_t}(t_1, E), \ldots, \mathsf{dec\_t}(t_k, E)) \\ \mathsf{decode}(\epsilon, E) & = & E \\ \mathsf{decode}(L\!:\!t\ s, E) & = & \mathsf{decode}(s, E) \oplus \{l \mapsto \mathsf{dec\_t}(t, \mathsf{decode}(s, E))\} \end{array}$$

where the operation $\oplus$ updates the partial function in the first argument according to the argument-value pairs provided in the second argument. For a stream $p = t\,s$ we define:

$$\mathsf{decode}(p) = \mathsf{dec\_t}(t, \mathsf{decode}(s, \emptyset))$$

where $\emptyset$ denotes the empty assignment. As an example, let us compute $E = \mathsf{decode}(s_{\mathsf{ex}}, \emptyset)$:

$$\begin{array}{lcl}
\mathsf{decode}(2:\ e, \emptyset) & = & \emptyset \oplus \{2 \mapsto e\} = \{2 \mapsto e\} \\
\mathsf{dec\_t}(b\,\mathsf{ref}\ 2\ \mathsf{ref}\ 2, \{2 \mapsto e\}) & = & b(e,e) \\
\mathsf{decode}(s_{\mathsf{ex}}, \emptyset) & = & \{2 \mapsto e\} \oplus \{1 \mapsto b(e,e)\} \\
 & = & \{1 \mapsto b(e,e), 2 \mapsto e\}.
\end{array}$$

Accordingly, the Σ-tree denoted by the stream $t_{\mathsf{ex}}\ s_{\mathsf{ex}}$ is given by:

$$\mathsf{dec\_t}(a\,\mathsf{ref}\ 1\ a\,\mathsf{ref}\ 2\ \mathsf{ref}\ 1, E) = a(b(e,e), e, b(e,e)).$$

Left Depth of a Tree. Consider an XML document consisting of element nodes only. It corresponds to a well-balanced sequence of opening and closing tags. How much memory is required to check whether a given sequence of tags is well-balanced? It is not hard to see that the required amount of memory is proportional to the *depth* of the unranked XML tree: for each opening tag we push its name onto a stack, and for each closing tag we pop if the tag names match (and report an error otherwise). Consider now a ranked encoding of an unranked XML document tree. First of all, consider the first-child next-sibling (fcns) encoding: it is a binary tree in which a node is the left child of its parent if it is the first child in the unranked XML tree, and it is the right-child of its parent if it is the next-sibling in the XML tree. How can we define the nesting-depth of the XML tree, in terms of its fcns-encoded binary tree B? We call this the *left-depth* (or *ldepth*) of B: it is the maximal number of left-edges traversed during any root-to-leaf path in B. We now slightly generalize this idea from binary trees to streams of ranked trees in $\mathcal{S}_\Sigma$. The ldepth of a stream $t\ s$ in $\mathcal{S}_\Sigma$ is defined as:

$$\begin{array}{lcl}
\mathsf{ldepth}'(a) & = & 0 \\
\mathsf{ldepth}'(a\,t_1 \ldots t_k) & = & \max\{1 + \mathsf{ldepth}'(t_1), \ldots, 1 + \mathsf{ldepth}'(t_{k-1}), \mathsf{ldepth}'(t_k)\} \\
\mathsf{ldepth}'(\mathsf{ref}\ l) & = & 1 \\
\mathsf{ldepth}(\epsilon) & = & 0 \\
\mathsf{ldepth}(L{:}t\ s) & = & \max\{\mathsf{ldepth}'(t), \mathsf{ldepth}(s)\} \\
\mathsf{ldepth}(t\ s) & = & \max\{\mathsf{ldepth}'(t), \mathsf{ldepth}(s)\}.
\end{array}$$

The left depth of a tree is meant to measure the maximal depth of the stack when performing a dfs preorder traversal over the tree. This means that we may give up the current stack entry as soon as we descend into the right-most subtree.

3 Top-Down Tree Transducers

For a finite-state automaton, the current state and input symbol determines the new states to transition to. In a finite-state transducer, a rule also carries an output word; when the rule is applied the output word is concatenated to the current output word. Top-down tree transducers generalize this idea from words to ranked trees. They were invented in the 1970's as formal models of compilers, and have recently been used as a model of XML transformations [19, 18, 20, 17]. Formally, a *top-down tree transducer* is a tuple $M = (Q, \Sigma, \Delta, q_0, R)$ where Q is a finite set of states, Σ and Δ are ranked alphabets of input and output symbols, respectively, $q_0 \in Q$ is the initial state, and R is a finite set of rules of the form

$$q(a(x_1, \ldots, x_k)) \quad \to \quad t$$

where t is a tree generated by this grammar:

$$t \quad ::= \quad q(x_i) \mid a(t_1, \ldots, t_m)$$

for $q \in Q$, $i \in \{1, 2, \ldots, k\}$, and $a \in \Delta$ of rank $m \geq 0$. If for each $q \in Q$ and $a \in \Sigma$ there is at most one rule in R with left-hand side $q(a(x_1, \ldots, x_k))$, then M is a *deterministic top-down tree transducer* (dtop). Let M be a dtop, $q \in Q$, $a \in \Sigma$, and $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$. The q-translation $[\![q]\!]$ is the function from $\mathcal{T}_\Sigma$ to $\mathcal{T}_\Delta$ defined recursively as: $[\![q]\!](a(t_1, \ldots, t_k)) = t[q'(x_i) \leftarrow [\![q']\!](t_i) \mid q' \in Q, i \in \{1, \ldots, k\}]$ if $q(a(x_1, \ldots, x_k)) \to t$ is in R. Note that for leaf labels $d_1, \ldots, d_n$ and trees $s_1, \ldots, s_n$, we denote by $[d_j \leftarrow s_j \mid 1 \leq j \leq n]$ the substitution that replaces in a given tree all occurrences of d_j by the tree s_j, for $1 \leq j \leq n$. The translation of M, denoted $[\![M]\!]$ is defined as $[\![q_0]\!]$.

As an example, consider a dtop M_1 that takes input trees $c(t_1, c(t_2, \ldots, c(t_n, e) \ldots))$. Each t_i is of the form $\text{book}(\text{authors}(a_1(e, a_2(e, \ldots a_m(e, e) \ldots))), \text{title}(t), \text{year}(t'))$, where a_j represents an author and t and t' represent the title and year of the book, respectively. The dtop M_1 translates each t_i into

$$\text{book}(\text{main}(a_1(e)), \text{title}(t), \text{year}(t'), \text{co}(a_2(e, a_3(e, \ldots a_m(e, e) \ldots)))).$$

The rules of the dtop M_1 are:

$$\begin{array}{lcl}
q(c(x_1, x_2)) & \to & c(q(x_1), q(x_2)) \\
q(\text{book}(x_1, x_2, x_3)) & \to & \text{book}(q_1(x_1), p(x_2), p(x_3), q_2(x_1)) \\
q_1(\text{authors}(x_1)) & \to & \text{main}(q_1(x_1)) \\
q_1(a(x_1, x_2)) & \to & a(e, e) \\
q_2(\text{authors}(x_1)) & \to & \text{co}(q_2(x_1)) \\
q_2(a(x_1, x_2)) & \to & p(x_2)
\end{array}$$

where a is any author and p is a state realizing the identity, i.e., for every input symbol f of rank $k \geq 0$ there is the rule $p(f(x_1, \ldots, x_k)) \to f(p(x_1), \ldots, p(x_k))$.

4 DTOP to Stream Processor

In general, a stream processor S can be considered as a word transducer $S = (\mathsf{conf}, \bar{\Sigma}, \vdash, \mathsf{init}, \mathsf{Final})$ where conf is a set of *configurations* or states and init and Final are the initial configuration and the set of final configurations, $\bar{\Sigma}$ is the set of characters or tokens in the input or output, and $\vdash$: $\mathsf{conf} \times (\bar{\Sigma} \cup \{\epsilon\}) \to \mathsf{conf} \times \bar{\Sigma}^*$ is a partial function describing the one-step transitions (or computations steps) of the processor. For the processor to operate deterministically, we additionally demand for $\vdash$, that $\vdash (q, a)$ is undefined for every $a \in \bar{\Sigma}$ whenever $\vdash (q, \epsilon)$ is defined. As usual, $\vdash$ is written infix. Thus, a transition: $(q, a) \vdash (q', w)$ means that in configuration q, the processor may consume the token a in the input, while outputting the string w and entering the configuration q'. Now assume that we are given a dtop $M = (Q, \Sigma, \Delta, q_0, R)$. Then we construct a corresponding stream processor S_M for M which is meant to traverse (the string representation of) an input stream denoting a tree t, and produce (the string representation of) an output stream denoting the output tree of M for t. The set $\bar{\Sigma}$ of tokens of the stream processor S_M consists of the alphabet Σ of M extended by tokens $\mathsf{ref}\ l$ with $l \in \mathsf{Lab}$, and L: with L a finite subset of Lab. The configurations of the stream processor are of the following form:

$$\begin{array}{lcl}
\mathsf{conf} & \subseteq & (\mathsf{lmap} \times \mathsf{stack} \times \mathsf{Lab}) \cup \{\mathsf{init}\} \\
\mathsf{lmap} & \subseteq & (Q \times \mathsf{Lab}) \to 2^{\mathsf{Lab}} \\
\mathsf{stack} & \subseteq & \mathsf{state}^* \\
\mathsf{state} & \subseteq & Q \to 2^{\mathsf{Lab}}.
\end{array}$$

Thus, a configuration of the stream processor either equals the initial configuration init, or consists of the following components:

- A mapping lmap which records for each state q of the dtop and label l in the input, the set of labels in the output stream which refer to a representation of the output for the corresponding subtree translated in state q of M. According to this usage, each label may occur at most once in any of the sets lmap $\langle q, l\rangle$.
- A stack of *local configurations* state where each such local configuration ρ provides for each state q of M, the set of output labels whose definition is meant to be the output of the current part of the input for state q of M.
- The next fresh label to be used in the output stream. For convenience in our implementation we assume labels to be natural numbers (starting from 0).

In the following, we use the convention that we only list nontrivial argument-value pairs of a mapping, i.e., where the value is different from $\emptyset$. The empty stack is denoted by [].

Initially (from configuration init), a first label $l_0 \in \mathsf{Lab}$ is produced. According to our convention, $l_0 = 0$. Then a reference to 0 is outputted, and a configuration is created in which the initial state q_0 maps to 0: $(\mathsf{init}, \epsilon) \vdash_M ((\emptyset, \{q_0 \mapsto \{0\}, 1), \mathsf{ref}\ 0)$.

```
let rec do_tree $t\ \phi\ (\lambda, o)$ =
    let $\{q_1 \mapsto L_1, \ldots, q_r \mapsto L_r\} = \phi$ in
    match $t$ with
     | ref $l$        →  for $j = 1, \ldots, r$
                             let $l'_j$  =  if $\lambda(q_j, l) = \emptyset$ then new()
                                            else select $(\lambda(q_j, l))$ in
                         let $\lambda' = \lambda \cup \{(q_j, l) \mapsto \{l'_j\} \mid j = 1 \ldots, r\}$ in
                         let $o' = o.L_1 : \mathsf{ref}\ l'_1\ \ldots L_r : \mathsf{ref}\ l'_r$ in
                         $(\lambda', o')$
     | $a\ t_1 \ldots t_k$  →  for $j = 1, \ldots, r$
                           let $\zeta_j = \mathsf{rhs}(q_j, a)$ in
                         let $N = \{\langle q', i\rangle \mid \exists j.\ q'(x_i)$ occurs in $\zeta_j\}$ in
                         forall $\langle q', i\rangle \in N$
                           let $l'_{q',i} = \mathsf{new}()$ in
                         for $j = 1, \ldots, r$
                           let $\zeta'_j = \zeta_j[q'(x_i) \leftarrow \mathsf{ref}\ l'_{q',i} \mid \langle q', i\rangle \in N]$ in
                         let $o' = o.L_1 : \zeta'_1 \ldots L_r : \zeta'_r$ in
                         let $\phi_1 = \{q' \mapsto \{l'_{q',1}\} \mid \langle q', 1\rangle \in N\}$ in
                            ...
                         let $\phi_k = \{q' \mapsto \{l'_{q',k}\} \mid \langle q', k\rangle \in N\}$ in
                         do_tree $t_k\ \phi_k$ (...
                         do_tree $t_1\ \phi_1\ (\lambda, o')$ ...)
let rec do_stream $s\ (\lambda, o)$ = match $s$ with
     | $\epsilon$      →  $(\lambda, o)$
     | $L : t\, s$  →  let $\phi\ q = \bigcup\{\lambda(q, l) \mid l \in L\}$ in
                    let $\lambda'\ (q, l)$ =  if $l \notin L$ then $\lambda(q, l)$
                                     else $\emptyset$   in
                    do_stream $s$ (do_tree $t\ \phi\ (\lambda', o)$)
```

Figure 1 The functional implementation of S_M

For the next cases we consider a configuration $c = (\lambda, \sigma, l')$. Assume that the next symbol in the input stream is a constructor $a \in \Sigma$, and $\sigma = \phi :: \sigma'$ where $\phi = \{q_1 \mapsto L_1, \ldots, q_r \mapsto L_r\}$ for non-empty sets $L_j \subseteq \mathsf{Lab}$. Furthermore assume that $q_j(a(x_1, \ldots, x_k)) \to t_j \ \in \ R$ for $j = 1, \ldots, r$. Then we consider a set L of fresh labels containing one distinct label $l_{q',i}$ for every $q'(x_i)$ occurring in any of the t_j. For $i = 1, \ldots, k$, let ϕ_i denote the mapping $\phi_i = \{q' \mapsto \{l_{q',i}\} \mid l_{q',i} \in L\}$. Let t'_j be the tree obtained from t_j by replacing $q'(x_i)$ with ref $l_{q',i}$. Then

$$(c, a) \vdash_M (c', L_1 \!:\! t'_1 \ldots L_r \!:\! t'_r) \quad \text{where} \quad c' = (\lambda, \phi_1 :: \ldots :: \phi_k :: \sigma', l' + |L|).$$

Now assume that the next token in the input is a reference ref l to some label l where $\phi = \{q_1 \mapsto L_1, \ldots, q_r \mapsto L_r\}$. Then we define a mapping λ' by $\lambda'(q_j, l) = \{l'_j\}$ for a new distinct label l'_j if $\lambda(q_j, l) = \emptyset$, and otherwise $\lambda'(q', l') = \lambda(q', l')$. Thus, new labels are created for every pair (q_j, l) with $j = 1, \ldots, r$, which has not yet been defined in λ. Assume that the number of these new labels is m. Now let select denote a (partial) function which selects one label from each nonempty set of labels, and define $l'_j = \mathsf{select}(\lambda'(q_j, l))$ for $j = 1, \ldots, r$. Then

$$(c, \mathsf{ref}\ l) \vdash_M (c', L_1 \!:\! \mathsf{ref}\ l'_1 \ldots L_r \!:\! \mathsf{ref}\ l'_r) \quad \text{where} \quad c' = (\lambda', \sigma', l' + m).$$

This means we redirect the references as provided by ϕ to the new references as introduced in λ'.

Finally, assume that the next token in the input is a label set L followed by : indicating a definition. Then new mappings ϕ' and λ' are constructed by $\phi'(q') = \bigcup\{\lambda(q', l) \mid l \in L\}$ and λ' is obtained from λ by setting the values for $(q', l), l \in L$, to $\emptyset$. With that, the transition is

$$(c, L\!:) \vdash_M (c', \epsilon) \quad \text{where} \quad c' = (\lambda', \phi' :: \sigma, l').$$

Note that if none of the $l \in L$ has ever been referred to, then $\lambda(q, l)$ is the empty set. This implies that the mapping ϕ' is the empty mapping as well. Consequently, the stream processor will *ignore* the rest of the definition of the $l \in L$, i.e., the subsequent tree in the stream. The stream processor terminates in configurations $(\emptyset, [], l')$.

The formalization of the operational behavior of S_M by means of a word transducer is convenient when it comes to verify that it processes its input symbol by symbol from left to right with a minimum of extra buffering. For reasoning about the correctness of the implementation of a dtop, though, this formulation is not convenient. For that purpose, we prefer to reformulate the definition of the translation of S_M by means of a functional program, see Figure 1. This functional program consists of a function do_tree for processing trees t together with a function do_stream for processing streams s. The function do_tree processes its argument t by recursive descent over the structure of t using an extra parameter ϕ to record a current mapping $Q \to 2^{\mathsf{Lab}}$. Implementing this function by means of recursion, allows us to get rid of explicitly maintaining stacks of such mappings. The function do_stream then processes sequences $L_1 \!:\! t_1 \ldots L_r \!:\! t_r$ from left to right by successively applying the function do_tree to the trees $t_1, \ldots, t_r$. For their correct operation, both functions refer to a global mapping $\lambda : Q \times \mathsf{Lab} \to 2^{\mathsf{Lab}}$, which might be changed during execution. Therefore, the current version of λ is passed as an extra parameter and returned as an extra result. The same happens with the output: the output so far is passed as a parameter and, possibly extended to the right also returned as a result. For the extension to the right of o_1 by a sequence o_2, we use the notation $o_1.o_2$. Finally, we have left the creation of fresh labels implicit by using the function new which returns a fresh label. The functional program from Figure 1 realizes the same translation as the stream processor specified as as string transducer.

5 Reference Reuse

The implementation so far referred to *fresh* labels whenever new labels are required. Labels therefore are unique throughout whole streams. If the streams are meant to represent XML documents, we thus can use IDs and IDREFs as provided by the XML standard. The number of labels, however, may grow proportional to the *size* of the input stream. In that case, the *sizes* of labels themselves can no longer be ignored both for storing intermediate information and, more importantly, also in the generated output. Even by using a binary encoding of node identities, the required bit size of the output may blow up by a logarithmic factor. In our design, on the other hand, we only require references referring to labels arriving in the future, i.e., are always located to the right. This allows to employ labels which are *not* unique throughout the whole document. In case of several occurrences of the same label l, a reference $\mathsf{ref}\ l$ simply refers to the *next* occurrence of the label l to the right. This idea enables us to reduce the total number of distinct labels and thus also the sizes of their identifiers. In fact, only as many labels are required as are maximally jointly *in scope*, i.e., the maximal number of references which at some point in the stream have already been referred to but which have not yet been defined. Technically, these consist of all labels which are mentioned either in the current stack σ or in one of the output sets of the global mapping λ of the current configuration. Let $L(\sigma, \lambda)$ denote this set. Then the next new label can be generated without referring to a counter which is passed around. Instead, it can be chosen as the smallest natural number not contained in $L(\sigma, \lambda)$. Fresh labels are possibly introduced when processing an input tag a or a reference $\mathsf{ref}\ l$ in the input. On the other hand, a label l' for the output goes *out of scope* as soon as some set L': is produced with $l' \in L'$. The label l' may be used as a fresh label already in the sub-stream succeeding the colon.

▶ **Theorem 1.** *Every dtop M can be compiled into a stream processor S_M such that for every stream p and input document t with $\mathsf{decode}(p) = t$ the following holds:*

1. *If $[\![M]\!](t)$ is undefined then so is $S_M(p)$. If $[\![M]\!](t) = t'$ then $S_M(p) = p'$ for some stream p' with $\mathsf{decode}(p') = t'$.*
2. *The depth of the stack of S_M when processing the input stream p is bounded by $l = (k-1) \cdot (\mathsf{ldepth}(p) + 1)$,*
3. *The size $|p'|$ of the output stream p' is bounded by $n \cdot d \cdot |p|$.*
4. *The maximal number of distinct labels used by S_M is bounded by $c \cdot n \cdot (l + r)$.*
5. *The cardinality of label sets L occurring in the output is at most $\max\{1, c\}$.*

Here k is the maximal rank of an input symbol occurring in p, n is the number of states of M, d is the maximal size of right-hand sides of the rules of M, r is the number of distinct labels in the input stream, and c is the maximal cardinality of label sets in the input.

Since each output token carries at most one set of labels, the total number of occurrences of labels in the output is bounded by $c \cdot n \cdot d \cdot |p|$, for an input stream p. Therefore, the total bit length of the output stream generated by a given stream processor is in $\mathcal{O}(m \cdot \log(m))$ if m is the bit length of the input stream — even if the dtop is unboundedly copying and thus may produce output trees of size exponential in the input tree. This even holds for the stream processor *without* reuse of labels. In case of label reuse, the total number of labels to be stored in any configuration is bounded by the number of the labels stored in the stack plus the number of labels stored in the mapping λ, i.e., proportional to $c \cdot n \cdot (l + r)$. Therefore, the bit length of the output can more precisely be bounded by $\mathcal{O}(m \cdot \log(l + r))$ if sets of labels in the input stream (and thus also in the output stream) are assumed to have constant size. Likewise, the maximal memory consumption in bits used by the stream processor can be bounded by $\mathcal{O}((l + r) \cdot \log(l + r))$.

In the following, we prove that the stream processor S_M of Theorem 1 indeed implements the corresponding dtop M. For simplicity, we only consider the vanilla version of S_M where always fresh labels are created. We proceed in two steps. First, we argue that the program from Figure 1 correctly implements the processor S_M. Then we show that the program realizes the transformation of the dtop M.

The correctness of the functional specification of the stream processor S_M follows from the following two lemmas, where for both lemmas we assume that the new labels conceived during the run of the stream processor agree with the new labels returned by corresponding calls of the function new() in the program. Assume that the partial function $\vdash_M$ on conf $\times$ $(\bar{\Sigma} \cup \{\epsilon\})$ is extended to a partial function $\vdash_M^*$ on conf $\times$ $\bar{\Sigma}^*$ in the obvious way. Lemma 2 states that the function do_tree correctly describes the behavior of the stream processor on trees.

▶ **Lemma 2.** *Assume that $\phi, o, \lambda, \lambda', \sigma, c, c'$ are given. Then for every tree t, the following two statements are equivalent:*
1. $((\lambda, \phi{::}\sigma, c), t) \vdash_M^* ((\lambda', \sigma, c'), o)$*;*
2. do_tree $t\, \phi\, (\lambda, o_1) = (\lambda', o_1 o)$ *for every sequence o_1 of output symbols.* ◻

The proof of this lemma is by induction on the structure of t. The second lemma then formalizes in which sense the function do_stream realizes the behavior of the stream processor on a stream s.

▶ **Lemma 3.** *Assume that c, c', λ, o are given. Then for every input stream s, the following two statements are equivalent:*
1. $(s, (\lambda, [], c)) \vdash_M^* ((\emptyset, [], c'), o)$*;*
2. do_stream $s\, (\lambda, o_1) = (\emptyset, o_1 o)$ *for every sequence o_1 of output symbols.* ◻

Again, the proof is by structural induction, where for each tree occurring in s Lemma 2 is applied.

Let us now turn the proof that the program correctly realizes the transformation specified by a dtop. For that, we extend the partial functions $[\![q]\!]$ of the dtop to trees t possibly containing references to labels $l \in$ Lab by additionally defining $[\![q]\!](\mathsf{ref}\ l) = \langle q, l\rangle$ for new output symbols $\langle q, l\rangle$. Accordingly, we also extend the notion of trees to include single occurrences of $\langle q, l\rangle$ as well. The following technical lemma is proved by induction on the size of t.

▶ **Lemma 4.** *Assume that L is the set of labels referenced in t, and that N is the set of all pairs $\langle q', l'\rangle$ occurring in $[\![q]\!](t)$. Furthermore, assume that $E : L \to \mathcal{T}_\Sigma$. Then $[\![q]\!](\mathsf{dec\_t}(t, E))$ is defined iff $[\![q']\!](E(l'))$ is defined for all $\langle q', l'\rangle \in N$, where $[\![q]\!](\mathsf{dec\_t}(t, E)) = [\![q]\!](t)[\langle q', l'\rangle \leftarrow [\![q']\!](E(l')) \mid \langle q', l'\rangle \in N]$.* ◻

The correctness of the functional specification of S_M with respect to the dtop M follows from Lemmas 5 and 6. Lemma 5 again is proved by structural induction on t.

▶ **Lemma 5.** *Assume that $\phi = \{q_1 \mapsto L_1, \ldots, q_r \mapsto L_r\}$ and $[\![q_j]\!](t)$ is defined for all j iff the call* do_tree $t\ \phi\ (\lambda, o)$ *terminates and returns (λ', o'). In this case, $\lambda' = \lambda \cup \lambda_2$ and $o' = o.o_2$ such that the following holds:*
1. *All labels in the image of λ_2 are fresh, i.e., do not yet occur in the image of λ;*
2. *For all $q' \in Q$, $l \in$ Lab, $\lambda_2(q', l)$ has cardinality at most 1;*
3. *$\lambda_2(q', l) \neq \emptyset$ iff $\lambda(q', l) = \emptyset$ and $\langle q', l\rangle$ occurs in $[\![q_j]\!](t)$ for some j.*
4. *Let $E(l') = \langle q', l\rangle$ whenever $l' \in \lambda_2(q', l)$. Then for every j and $l \in L_j$,* $\mathsf{decode}(o_2, E)(l) = [\![q_j]\!](t)$. ◻

▶ **Lemma 6.** *Assume that $E = \mathsf{decode}(s, \emptyset)$ and consider a mapping λ such that $E(l)$ is defined whenever $\lambda(q, l) \neq \emptyset$ for some q. Let N denote the set of pairs $\langle q, l\rangle$ where $\lambda(q, l) \neq \emptyset$. Then $[\![q]\!](E(l))$ is defined for all $\langle q, l\rangle \in N$ iff* do_stream $s\ (o, \lambda)$ *is defined. Moreover, in this case, for every o,* do_stream $s\ (o, \lambda) = (o.o_2, \emptyset)$ *where $[\![q]\!](E(l)) = \mathsf{decode}(o_2, \emptyset)(l')$ for all $l' \in \lambda(q, l)$.*

Proof. We proceed by structural induction on the sequence of definitions s. If $s = \epsilon$, the assertion is vacuously true. Now assume that $s = L{:}t\ s'$, and consider a mapping λ. Let $\bar{L}$ denote the set of labels defined in s where $\lambda(q', l) \neq \emptyset$ for some q' where $\bar{L} = \bar{L}_1 \cup \bar{L}'$ with $\bar{L}_1 = \bar{L} \cap L$, and $\bar{L}'$ are the labels in $\bar{L}$ defined in s'. If $\bar{L}_1 = \emptyset$, the assertion follows by induction hypothesis on s'. Therefore now assume that $\bar{L}_1 \neq \emptyset$, and let ϕ denote the mapping $\phi(q') = \bigcup\{\lambda(q', l) \mid l \in L\}$, which we write as usual $\phi = \{q_1 \mapsto L_1, \ldots, q_r \mapsto L_r\}$. First assume that $[\![q]\!](E(l))$ is defined for all $\langle q, l\rangle \in N$. Then $[\![q]\!](E(l))$ is defined for all $\langle q, l\rangle \in N$ where $l \in L$.

Let λ_1 be obtained from λ by removing all entries for $l \in L$ Then by Lemma 5, there is a mapping λ_2 and an output o_1, with the following properties:

1. The call do_tree t ϕ (λ_1, o) terminates and returns $(\lambda_1 \cup \lambda_2, o.o_1)$.
2. $\lambda_2(q', l) \neq \emptyset$ iff $\lambda_1(q', l) = \emptyset$ and $\langle q', l\rangle$ occurs in $[\![q_j]\!](t)$ for some j.
3. Let $E_1(l') = \langle q', l\rangle$ whenever $l' \in \lambda_2(q', l)$. Then for every j and $l \in L_j$, decode$(o_1, E_1)(l) = [\![q_j]\!](t)$.

Now let $\lambda' = \lambda_1 \cup \lambda_2$, and $o' = o.o_1$. Then by induction hypothesis for s' and λ', do_stream s' (o', λ') is defined where do_stream s' $(o', \lambda') = (o'.o_2, \emptyset)$ for some o_2 where

$$[\![q']\!](E(l')) = \mathsf{decode}(o_2, \emptyset)(l'')$$

for all $l'' \in \lambda'(q', l')$. Therefore, do_stream s (o, λ) is defined for every o. Moreover, in this case, do_stream s $(o, \lambda) = (o.o_1.o_2, \emptyset)$. Therefore for $l \in \bar{L}_1$, and $l' \in \lambda(q, l)$,

$$\begin{aligned} [\![q]\!](E(l)) &= [\![q]\!](t)[\langle q', l''\rangle \leftarrow \mathsf{decode}(o_2, \emptyset)(l_{q',l''})] \\ &= \mathsf{decode}(o_1, \mathsf{decode}(o_2, \emptyset))(l') \\ &= \mathsf{decode}(o_1.o_2, \emptyset)(l') \end{aligned}$$

while for $l \in \bar{L}'$, the assertion follows by induction hypothesis for s'. This proves the first direction. The reverse direction follows analogously, again by using lemma 5 for the inductive step. ☐

Altogether, we thus have shown that the stream processor S_M when applied to the stream representation of a tree t, produces a stream representation of the output tree returned by M for t.

6 Avoiding Reference Chains

The disadvantage of the construction so far is that it may abundantly generate references which themselves may point to references. In particular, this is the case if the dtop has erasing rules, i.e., rules whose right-hand sides do not produce any output nodes but consist of recursive calls only.

▶ **Example 7.** Consider the dtop with the following transitions:

$$\begin{array}{lllllll} q_0(f(x_1,x_2)) & \to & q_1(x_1) & \quad & q_1(f(x_1,x_2)) & \to & q_0(x_2) \\ q_0(\bot) & \to & a & & q_1(\bot) & \to & b \end{array}$$

where q_0 is the initial state. The corresponding stream processor translates the input stream f ref l_1 $\bot$ l_1 : f $\bot$ ref l_2 l_2 : $\bot$ into the stream ref 0 0 : ref 1 1 : ref 2 2 : a. ☐

In order to avoid such chains of references, we modify the rules for constructor applications and references as follows. Assume that $a \in \Sigma$ and $\sigma = \phi :: \sigma'$ with $\phi = \{q_1 \mapsto L_1, \ldots, q_r \mapsto L_r\}$. Furthermore, assume that for $j = 1, \ldots, r$, $q_j(a(x_1, \ldots, x_k)) \to \zeta_j$. Then we consider the set N_i the set of all states $q' \in Q$ with the following two properties:

- $q'(x_i)$ occurs as a proper subterm in any of the ζ_j;
- $q'(x_i)$ is different from any of the ζ_j.

For $i = 1, \ldots, k$, let ϕ_i denote the mapping defined by

$$\begin{array}{lll} \phi_i(q') & = & \{l_{q',i}\} \qquad \text{if } q' \in N_i \\ \phi_i(q') & = & \bigcup\{L_j \mid \zeta_j = q'(x_i)\} \quad \text{otherwise} \end{array}$$

where the $l_{q',i}$ are fresh labels. Let $j_1, \ldots, j_m$ be the subsequence of the j' where ζ_j produces output nodes, i.e., is *not* just equal to some recursive call $q'(x_{i'})$. For $j' \in \{j_1, \ldots, j_m\}$, let $\zeta'_{j'}$ be obtained from $\zeta_{j'}$ by replacing $q'(x_i)$ with ref $l'_{q',i}$ for some $l'_{q',i} = \mathsf{select}(\phi_i(q'))$. Then

$$o' = L_{j_1} : \zeta'_{j_1} \ldots L_{j_m} : \zeta'_{j_m} \quad \text{and} \quad c' = (\lambda, \phi_1 :: \ldots :: \phi_k :: \sigma').$$

A second modification takes place for processing references in the input. Let ref l be a reference in the input and $\sigma = \phi :: \sigma'$ where $\phi = \{q_1 \mapsto L_1, \ldots, q_r \mapsto L_r\}$. We define a mapping λ' by $\lambda'(q_j, l) = \lambda(q_j, l) \cup L_j$ for $j = 1, \ldots, r$ and $\lambda'(q', l') = \lambda(q', l')$ otherwise. Thus, no new labels are created at all. Also, no output is produced, λ is updated to λ', and σ is popped to σ', i.e., $c' = (\lambda', \sigma')$.

▶ **Example 8.** Consider again the dtop from Example 7 together with the input stream

$$f \ \mathsf{ref}\ l_1 \perp \ l_1 : f \perp \mathsf{ref}\ l_2 \ \ l_2 : \perp$$

Then no new label is ever introduced. Instead, the output is given by ref 0 $0 : a \perp\perp$. ☐

In the previous example, the output stream without reference chains is much simpler than the original output stream. In some cases, though, sets of labels in the output stream grow considerably.

▶ **Example 9.** Consider the following dtop with states q, q':

$$\begin{array}{lcl@{\qquad}lcl} q(a(x_1,x_2)) & \to & b(q(x_2), q'(x_2)) & q'(a(x_1,x_2)) & \to & q'(x_2) \\ q(\perp) & \to & c(\perp,\perp) & q'(\perp) & \to & d(\perp,\perp) \end{array}$$

Assume that q is the initial state. Then the input stream $a \perp a \perp a \perp\perp$ is translated to the stream

$$\mathsf{ref}\ 0 \ \ 0 : b \ \mathsf{ref}\ 1 \ \mathsf{ref}\ 2 \ \ 1 : b \ \mathsf{ref}\ 3 \ \mathsf{ref}\ 4 \ \ 3 : b \ \mathsf{ref}\ 5 \ \mathsf{ref}\ 6 \ \ 5 : c \perp\perp \ \ 2,4,6 : d \perp\perp$$

This means that the labels for the erasing calls of state q' all are collected into one label set. ☐

A second source for the growing of label sets may be sharing present in the input stream.

```
process_tree $t$ $\phi$ $(\lambda, o)$  =  let $\phi = \{q_1 \mapsto L_1, \ldots, q_r \mapsto L_r\}$ in
                                  match $t$ with
                                  | ref $l$       →  let $l' =$ if $\lambda(q,l) \neq \emptyset$ then select$(\lambda(q,l))$
                                                               else new() in
                                                     let $\lambda' = \lambda \cup \{(q,l) \mapsto \{l'\}\}$ in
                                                     let $o' = o$.ref $l'$ in
                                                     $(\lambda', o')$
                                  | $a\, t_1 \ldots t_k$  →  let $j =$ factorize $\phi\, a$ in
                                                     let $o' = o.L_1$ in
                                                     let $(\lambda', o') =$ inline $t$ $q_1$ $(\lambda', o')$ in
                                                         ...
                                                     let $o' = o.L_j$ in
                                                     let $(\lambda', o') =$ inline $t$ $q_j$ $(\lambda', o')$ in
                                                     let $\phi' = \{q_{j+1} \mapsto L_{j+1}, \ldots, q_r \mapsto L_r\}$ in
                                                         process_tree $t$ $\phi'$ $(\lambda', o')$
```

Figure 2 Modification of the function process_tree.

```
inline $t$ $q$ $(\lambda, o)$ = match $t$ with
    | ref $l$        → let $l'$ = if $\lambda(q,l) \neq \emptyset$ then select$(\lambda(q,l))$
                                  else new() in
                       let $\lambda' = \lambda \cup \{(q,l) \mapsto \{l'\}\}$ in
                       let $o' = o$.ref $l'$ in
                       $(\lambda', o')$
    | $a\, t_1 \dots t_k$ → let rhs$(q,a) = \zeta_0 q_1(x_{j_1})\zeta_1 \dots q_2(x_{j_m})\zeta_m$
                           where $\zeta_0, \dots, \zeta_m$ do not contain $x_i$
                       let $o' = o.\zeta_0$ in
                       let $(\lambda', o')$ = inline $t_{j_1}$ $q_1$ $(\lambda, o')$ in
                       let $o' = o'.\zeta_1$ in
                       ...
                       let $(\lambda'.o')$ = inline $t_{j_m}$ $q_m$ $(\lambda', o')$ in
                       let $o' = o'.\zeta_m$ in
                       $(\lambda', o')$
```

Figure 3 The function inline.

▶ **Example 10.** Consider the following dtop with the single state q implementing the identity:

$$q(a(x_1,x_2)) \quad \rightarrow \quad a(q(x_1), q(x_2)) \qquad\qquad q(\bot) \quad \rightarrow \quad \bot$$

together with the input stream a ref l_1 a ref l_1 a ref l_2 $\bot$ l_2 : ref l_1 $l_1 : \bot$ containing four occurrences of references to the same node. This input stream is translated into the output stream

ref 0 $0{:}a$ ref 1 ref 2 $2{:}a$ ref 3 ref 4 $4{:}a$ ref 5 ref 6 $6{:}\bot$ $1,3,5{:}\bot$

This means that all labels introduced for the occurrences of the references ref l_1, ref l_2 are collected into one label set. ☐

In fact, these two mechanisms are the only reasons how large sets of labels in the output set may be accumulated. We have:

▶ **Theorem 11.** *For a dtop M, a streaming processor S'_M can be constructed with the following properties: (1) The output stream does not contain chains of references (i.e., substreams of the form L:*ref *l) even if such subexpressions occur in the input stream. (2) The cardinality of any label set in the output is bounded by $a \cdot b$, where a is the maximal sharing in the input, and b is the number of visits of M to nodes in the tree unfolding of the input.*

Hereby, the *maximal sharing* of a stream p is the maximal number of occurrences of references ref l all of which directly or indirectly point to the same subtree. The maximal number of visits to subtrees of the input, on the other hand, is the maximal number of occurrences of leaves $\langle q, l\rangle$ in the output produced by M for any input tree which contains a single reference ref l.

Beyond the construction for Theorem 11, further optimizations are possible. In some cases, for example, sizes of label sets can be decreased by introducing finite look-ahead into the input stream.

▶ **Example 12.** Consider the dtop implementing the identity from Example 10, now together with the input stream a ref l_1 a ref l_1 a ref l_1 $\bot$ $l_1{:}\bot$ which contains three occurrences of references to the same leaf with label l_1. This input stream is translated into the output stream

ref 0 $0{:}a$ ref 1 ref 2 $2{:}a$ ref 3 ref 4 $4{:}a$ ref 5 ref 6 $6{:}\bot$ $1,3,5{:}\bot$

This means that all labels introduced for the occurrences of the reference ref l_1 are collected into one label set. $\square$

All new labels introduced in the output stream of the Example 12 for the recursive calls refer to a call of state q for the same node with label l_1. In many cases, a k-symbol *look-ahead* into the input stream (for some fixed $k \geq 1$) allows to detect whether the ith child of a node is a reference or not. Note that the *first* child always will be contained in the look-ahead buffer. In case that the ith child is in the look-ahead buffer and a reference ref l, introduction of a fresh label $l'_{q',i}$ can be avoided for a recursive call $q'(x_i)$ given that the pair $\langle q', l\rangle$ is already in the domain of the current λ. Then any label $l' \in \lambda(q', l)$ can be selected to replace the call $q'(x_i)$ with ref l' in the output stream. In the Example 10, the transformation of the input stream with look-ahead 1 results in the output stream:

ref 0 $0\!:\!a$ ref 1 ref 2 $2\!:\!a$ ref 1 ref 3 $3\!:\!a$ ref 1 ref 4 $4\!:\!\bot$ $1\!:\!\bot$

7 Inlining

An important class of stream transformations produces output nodes in the same order as nodes occur in the input stream. This is, e.g., the case for the identity mapping from Example 10. For such transformations, insertion of references is an extra overhead which we may want to avoid. In order to do so, we propose the following optimization to our basic stream processor. We give the modified semantics expressed as a functional program, instead of using the small-step semantics of the stream processor. The starting point is the functional program of Figure 1 for our vanilla stream processor. Within this functional program, we first modify the case for trees $a\, t_1 \dots t_k$. Given a mapping $\phi = \{q_1 \mapsto L_1, \dots, q_r \mapsto L_r\}$, we first determine the maximal prefix of the corresponding sequence $\zeta_1, \dots, \zeta_r$ of right-hand sides for states $q_1, \dots, q_r$ and a, within which all $q'(x_i)$ can be recursively inline-expanded (a precise definition follows below).

Having successively executed this inline-processing, the remaining suffix is produced where all remaining occurrences of subtrees $q'(x_i)$ are replaced by references as before. For simplicity of presentation, our algorithm will perform inlining only for complete right-hand sides ζ_j. Accordingly, let factorize be the auxiliary function which, when called for $\phi = \{q_1 \mapsto L_1, \dots, q_r \mapsto L_r\}$ and a k-ary constructor a, will return the maximal j such that inlining can be applied for the right-hand sides $\zeta_1, \dots, \zeta_j$ of $q_1, \dots, q_j$ for a in sequence. The function inline processes an input tree t for a given state q (cf. fig. 3). If t is equal to a reference ref l, we produce the corresponding output reference. The interesting case is where t equals $a\, t_1 \dots t_k$ for some $t_1, \dots, t_k$. An occurrence of $q'(x_{j'})$ in a sequence $\zeta = L_1\!:\!\zeta_1 \dots L_r\!:\!\zeta_r$ of right-hand sides ζ_i (decorated with sets of labels L_i), can be *inline-expanded* if the following properties are satisfied:

- The state q' can be *recursively inline-expanded*;
- There is no further occurrence of $x_{j'}$ anywhere in the sequence ζ;
- all calls $p(x_j)$ with $j < j'$ occur to the left and can, if present, be inline-expanded.

Furthermore, a state q can be recursively inline-expanded, if each call $q'(x_{j''})$ in every right-hand side for q can be inline-expanded. Given the dtop M, the set of all such states can be determined in polynomial time before-hand. If in particular, the start state q_0 is found to be recursively expandable, even the initial reference in the output stream can be avoided and the function inline directly be called for the main tree in the input.

8 Related Work

We are not aware of works that consider tree streams with references. Filiot, Gauwin, Reynier, and Servais [5] show that two large subclasses of visibly pushdown transducers (VPTs) can be streamed with memory only depending on the height of the unranked input tree and on the transducer. For one class, the memory depends exponentially on the height of the input and for the other class it depends quadratically on the input height. The expressive power of VPTs is incomparable to that of dtops, but includes linear size increase dtop translations. They also show that it is decidable for a given VPT, whether or not it can be streamed with height bounded memory.

Segoufin and Vianu [24] and Segoufin and Sirangelo [23] present subclasses of schemas (expressed in terms of tree automata) that can be validated using finite automata (and hence true constant memory). Kumar, Madhusudan, and Viswanathan [15] show that visibly pushdown automata can be validated over XML streams with height bounded memory.

There is a large body of work on streaming of XPATH. The most fundamental result is by Green, Gupta, Miklau, Onizuka, and Suciu [9]: they translate simple XPATH queries into finite word automata. These automata can be evaluated using height bounded memory. Their approach is practical and gives theoretical guarantees; however, it only works for a rather restricted class of XPATH queries. Grohe, Koch, and Schweikardt show that Core XPATH, namely the full navigational fragment of XPATH 1.0 can be evaluated with height bounded memory [10]. Bar-Yossef, Fontoura, and Josifovski [2] study theoretical bounds as well as practical streaming algorithms for XPATH; more lower bounds for XPATH are given by Ramanan [21]. Shalem and Bar-Yossef [25] investigate twig-join algorithms over XML streams. Some algorithms have been proposed which output nodes selected by an XPATH query at the earliest possible event in the stream, e.g., Benedikt and Jeffrey [3] and Gauwin, Niehren, and Tison [7, 8].

On the practical side there are several best-effort approaches which use “as little memory as possible”, but do not give guarantees. Most notably, there is the Michael Kay’s SAXON system [12]. It can stream XSLT transformations in a best-effort approach. Michael Kay is also the editor of the W3C working draft on XSLT 3.0; the primary purpose of that draft is to change the language in order to enable streamed processing. Note that only very restricted transformations can be formulated through the stream primitives of XSLT 3.0. For XQuery there are several best-effort systems: Raindrop [27], GCX [14], the BEA streaming processor [6], and XTISP [11]. Note that XTISP is based on a more general model that our top-down tree transducers, namely, on macro forest transducers (MFTs) [20]. They do not guarantee memory bounds, but show that their MFT-based system performs on par with the state-of-the art XQuery streaming engine GCX.

9 Conclusion

Through the enhancement of XML streams by forward references, a large and natural class of tree transformations can be realized with bounded memory. The memory only depends on the depth of the XML document (corresponding to the ldepth of the tree encoding), the number of distinct references in the input, and the transducer. The class of transducers, namely, deterministic top-down tree transducers, have many convenient properties: e.g., they are closed under composition [1], they have a canonical normal form and decidable equivalence (even in PTIME for total transducers) [4], and they can be Gold-style learned from examples [16]. Experiments (see Appendix) show that the price for references is not high: the largest blow-ups are by a factor of three. In return, we are able to stream with constant memory transformations that are un-thinkable on conventional XML streams, e.g., and we can interchange large subtrees. Using references has the beneficial side effect that copies of the transducer are represented only once (i.e., DAGs are outputted). It achieves compression proportional to the amount of copying. Thus, for certain transformations we obtain output streams that are much smaller than the corresponding conventional XML streams.

References

1 B. S. Baker. Composition of top-down and bottom-up tree transductions. *Information and Control*, 41(2):186–213, 1979.

2 Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of XPath evaluation over XML streams. *J. Comput. Syst. Sci.*, 73(3):391–441, 2007.

3 M. Benedikt and A. Jeffrey. Efficient and expressive tree filters. In *FSTTCS*, pages 461–472, 2007.

4 J. Engelfriet, S. Maneth, and H. Seidl. Deciding equivalence of top-down XML transformations in polynomial time. *J. Comput. Syst. Sci.*, 75(5):271–286, 2009.

5 E. Filiot, O. Gauwin, P.-A. Reynier, and F. Servais. Streamability of nested word transductions. In *FSTTCS*, pages 312–324, 2011.

6 D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *VLDB J.*, 13(3):294–315, 2004.

7 O. Gauwin, J. Niehren, and S. Tison. Earliest query answering for deterministic nested word automata. In *FCT*, pages 121–132, 2009.

8 O. Gauwin, J. Niehren, and S. Tison. Queries on XML streams with bounded delay and concurrency. *Inf. Comput.*, 209(3):409–442, 2011.

9 T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

10 M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. *Theor. Comput. Sci.*, 380(1-2):199–217, 2007.

11 S. Hakuta, S. Maneth, K. Nakano, and H. Iwasaki. Xquery streaming by forest transducers. In *ICDE*, pages 952–963, 2014.

12 M. Kay. Ten reasons why saxon XQuery is fast. *IEEE Data Eng. Bull.*, 31(4):65–74, 2008.

13 C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. *VLDB J.*, 16(3):317–342, 2007.

14 C. Koch, S. Scherzinger, and M. Schmidt. The GCX system: Dynamic buffer minimization in streaming XQuery evaluation. In *VLDB*, pages 1378–1381, 2007.

15 V. Kumar, P. Madhusudan, and M. Viswanathan. Visibly pushdown automata for streaming XML. In *WWW*, pages 1053–1062, 2007.

16 A. Lemay, S. Maneth, and J. Niehren. A learning algorithm for top-down XML transformations. In *PODS*, pages 285–296, 2010.

17 S. Maneth and F. Neven. Structured document transformations based on XSL. In *DBPL*, pages 80–98, 1999.

18 W. Martens and F. Neven. On the complexity of typechecking top-down XML transformations. *Theor. Comput. Sci.*, 336(1):153–180, 2005.

19 W. Martens, F. Neven, and M. Gyssens. Typechecking top-down XML transformations: Fixed input or output schemas. *Inf. Comput.*, 206(7):806–827, 2008.

20 T. Perst and H. Seidl. Macro forest transducers. *Inf. Process. Lett.*, 89(3):141–149, 2004.

21 P. Ramanan. Memory lower bounds for XPath evaluation over XML streams. *J. Comput. Syst. Sci.*, 77(6):1120–1140, 2011.

22 T. Schwentick. Automata for XML - a survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.

23 L. Segoufin and C. Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *ICDT*, pages 299–313, 2007.

24 L. Segoufin and V. Vianu. Validating streaming XML documents. In *PODS*, pages 53–64, 2002.

25 M. Shalem and Z. Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In *ICDE*, pages 824–832, 2008.

26 G. von Bochmann. Semantic evaluation from left to right. *Commun. ACM*, 19(2):55–62, 1976.

27 M. Wei, E. A. Rundensteiner, M. Mani, and M. Li. Processing recursive XQuery over XML streams: The Raindrop approach. *Data Knowl. Eng.*, 65(2):243–265, 2008.

dataset	size (in GB)	#tags (in M)	depth	#tags/size (in M/GB)
Dblp	1.2	30.2	6	25.16
Medline	0.14	3.0	7	21.13
Treebank	.07	2.4	36	30.07
Xmark-4	0.45	6.6	12	14.25
Xmark-8	0.9	13.3	12	14.26
Xmark-16	1.8	26.7	12	14.25
Xmark-32	3.6	53.4	12	14.26
Xmark-64	7.2	106.9	12	14.25
Xmark-128	15.0	213.8	12	14.25

Table 1 Statistics of the data sets

A Experimental evaluation

We implemented a prototype in C++ using a well known XML parser (xerces). Our prototype takes as input a dtop and an XML file or stream. It executes the dtop in one of the following four modes:

- No-reuse: generates a fresh label for each new() call.
- Reuse: it reuses labels that are out of scope instead of generating new ones as described in Section 5.
- Chain: avoids outputting reference chains by outputting sets of references, see Section 6.
- Inline: inlines the output of a child. It can be combined with any of the previous versions. E.g., Inline-Chain means that we run Chain when inline is not applicable.

Since our stream format is different from raw XML, we do not include comparisons with existing engines (except a mention about memory consumption). We concentrate on comparing the impacts of the different optimizations of our algorithms.

XML Encoding. We consider XML documents consisting of attribute, element and text nodes. We use the first-child next-sibling encoding of XML documents: the document root node is parsed as a unary node, all element nodes are parsed as binary nodes (with first child and next sibling as left and right child, respectively), and a text node is either a unary node or a leaf (depending on whether it has a next sibling in a mixed content or not). Note that there is a schema-based XML encoding for dtops [16] which is supported by our prototype, but which is not used in the experiments here.

Datasets. We use four different datasets: Dblp and Medline (bibliographic databases), Treebank (representation of natural language parse trees), and Xmark (synthetic auction data) of various sizes. Table 1 shows statistics about these sets. The last column shows the number of tags per GB; Treebank has the largest such value, meaning that its markup density is the highest of all our documents. As can be seen in Table 2, this value is inversely proportional to the parsing speed: the denser the markup, the slower the parsing. In that table we show the pure parsing speed, and also the time needed to output the identity of the document (without using a dtop, just following the SAX parse events), called **r&w** in the table. The speed of our processors is often higher than the r&w, but can never be faster than the number for **read**. Dblp consists of a long list of bibliographic entries. Each entry is a subtree where the root label describes the type of the entry and is one of article, inproceedings, masters, www, ...

Queries. We evaluate our stream processors using these dtops:

- toc-*, where * is in $\{\text{article}, \text{masters}\}$, is a dtop that generates a list of all *-rooted subtrees in the Dblp document.
- toc-article-all same as toc-* but generates another top-level list with all entries in the Dblp document except the article entries.

$$\begin{array}{lcl} p(*(x_1,x_2)) & \to & *(p(x_2),p(x_1)) \\ p(*(x_1)) & \to & *(p(x_1)) \\ p(*) & \to & * \end{array}$$

Figure 4 The dtop REV

$$\begin{array}{lcl} q_0(*(x_1)) & \to & *(q_1(x_1)) \\ q_1(\text{article}(x_1,x_2)) & \to & \text{article}(q_{\text{id}}(x_1),q_1(x_2)) \\ q_1(*(x_1,x_2)) & \to & q_1(x_2) \\ q_1(*(x_1)) & \to & q_1(x_1) \\ q_1(*) & \to & * \\ p_0(*(x_1)) & \to & *(p_1(x_1)) \\ p_1(\text{article}(x_1,x_2)) & \to & \text{aList}(\text{article}(q_{\text{id}}(x_1),q_1(x_2)),p_1(x_2)) \\ p_1(*(x_1,x_2)) & \to & p_1(x_2) \\ p_1(*(x_1)) & \to & p_1(x_1) \\ p_1(*) & \to & * \\ r_0(*(x_1)) & \to & *(\text{Out}(q_1(x_1),r_1(x_1))) \\ r_1(*(x_1,x_2)) & \to & *(q_{\text{id}}(x_1),r_1(x_2)) \\ r_1(*(x_1)) & \to & r_1(x_1) \\ r_1(*) & \to & * \end{array}$$

Figure 5 The dtops toc-article, toc-article-DAG, and toc-article-all

- toc-article-DAG generates a list of all the articles in Dblp. With each article it outputs a list of all article subtrees that follow in the file.
- REV recursively inverts the order of first child and next sibling.

The rules of these dtops are shown in Figures 4 and 5. All dtops run over the first-child next-sibling encoding of the input XML document, thus in dtop rules, x_1 refers to the first child and x_2 refers to the next sibling of the current node. In the left-hand sides of rules $*$ refers to a wildcard symbol; it matches if no other rule for that state and the given arity matches. Such rules cannot test the text values in the document. It is straightforward, however, to extend these rules so that, e.g., word transducer rules are used to process data values. Similarly, attribute values could be tested (our prototype currently simply copies them over from the input element node).

Test machine. Our tests are run on an isolated 64-bit Intel Xeon -E5520@2.26GHz with 72 GB DDR3@800 MHz RAM was used. It ran Ubuntu 9.10 (kernel 2.6.31-19-server), using gcc version 4.4.1 with `-O9` option. Time results refer to CPU user time. The parsing of XML is done using Xerces-C++[1] version 3.1.1.

Memory Consumption. Table 2 shows the memory consumption of the whole system (Total), the memory strictly used by the parser (Parser), and the memory used by the stream processor (Processor). As can be seen, the memory consumption of the processor is proportional to the depth of the unranked XML tree (thus, Treebank needs more memory than Dblp). The memory stays the same across all different dtops and for all the different versions, except for the Chain version. — Note our processor's memory consumption is minuscule compared to conventional processors not using references; e.g., for REV their memory consumption may be half of document size.

Reusing References. Table 3 shows a comparison between No-reuse, Reuse, and Inline (-reuse) for the reverse (REV) transformation. We do not show the numbers for Chain because for REV no reference chains are ever introduced. As can be seen, when references are reused then only a small

[1] http://xerces.apache.org/xerces-c/

dataset	total	parser	processor	r&w	read
Dblp	5,124	5,012	112	12.06	42.37
Medline	5,502	5,388	114	13.84	48.83
Treebank	4,880	4,732	148	12.17	46.89
Xmark-*	4,820	4,700	120	19.59	71.05

Table 2 Memory (in **KB**) and parsing speeds (in **MB/s**)

dataset	algo	#ref	#dist.ref	output	over	MB/s
Xmark-128	No-re	427M	427M	30GB	2.00	8.17
	Reuse	427M	12	24GB	1.60	8.46
	Inline	213M	11	19GB	1.27	8.69
Medline	No-re	6M	6M	342MB	2.33	6.07
	Reuse	6M	7	275MB	1.87	6.26
	Inline	3M	6	213MB	1.45	6.38
Treebank	No-re	4.8M	4.8M	243MB	2.93	4.83
	Reuse	4.8M	36	198MB	2.39	4.87
	Inline	2.4M	35	141 MB	1.70	5.35
Dblp	No-re	60.3M	60.3M	3.3GB	2.75	4.75
	Reuse	60.3M	6	2.5GB	2.08	4.95
	Inline	30M	5	1.8GB	1.50	5.23

Table 3 Comparison of running REV on different documents

number of distinct references is needed. For these simple one-state dtops, the number of distinct references is just the depth of the document. Reusing references also produces smaller output files due to the smaller numbers printed in the output: 15%-24% smaller than for No-reuse. When we run the Chain version then no difference is observed in any of the numbers, except for the throughput. This is because there are no chains of references for REV, but, checking for them induces a slight overhead. These numbers are omitted.

Note that if we inline a dtop that realizes the identity transformation (same as REV but x_1 and x_2 interchanged in the first rule), then we obtain exactly the input file as output, i.e., there are no references and labels in the output. For REV the situation is more interesting: as can be seen in Table 3, the number of reference is exactly halved. This is because the call $p(x_2)$ in the right hand side of the first rule in Figure 4 cannot be inlined. Note that our current prototype does not introduce labels for text nodes; thus, the $p(x_1)$-call in the $p(*, 1)$-rule is always (in all versions) inlined and no reference is introduced.

Let us consider the dtop toc-article-all. Recall that toc-article-all produces as output a copy of the original input of Dblp, plus, a list of all article subtrees in the file. When producing the article subtrees, chains of references are produced at input nodes on the paths to the article nodes. As can be seen on the top of Table 4 the total number of references is only 60.3M for Chain, as compared to 64M for Reuse (and No-reuse). Thus, the additional numbers printed at those reference chain nodes incur quite a number of extra references. This is also reflected in the output size: 2.5GB when chains are avoided, and 2.5GB otherwise.

Reference Chain Avoidance. To further investigate the impact of removing reference chains, we test the toc-* dtop. It generates a list of all *-subtrees. We run it over Dblp for $*$ =article and $*$ =masters. Note that there are far fewer masters-subtrees in Dblp than there are article-subtrees (9 versus ca. 1 million). As can be seen in Table 4, the impact of reference chain avoidance is much more drastic: for toc-masters the number of references goes from 3.6 million down to 106 references

dtop	algo	#ref	output	over	MB/s
toc-article-all	Reuse	64M	2.6GB	2.17	4.75
	Chain	60.3M	2.5GB	2.08	4.13
	Inline	7.3M	1.4GB	1.17	6.08
toc-article	Reuse	26M	1.0GB	2.23	9.56
	Chain	23M	966MB	2.10	8.72
	Inline	1	459MB	1	12.84
toc-masters	Reuse	3.6M	81MB	26K	18.11
	Chain	106	5.48KB	1.75	22.44
	Inline	1	3.13KB	1	25.91

Table 4 Toc-article versus toc-masters over Dblp

and the size shrinks from 81MB to 5.48KB; this is further improved by inlining. Unlike for toc-article, for toc-masters the impact of ref-chain avoidance is much larger than the impact of inlining. The table also show in column **over**, the factor by which the output is larger as compared to the conventional XML output without references. As we can see, with inlining the factor goes down to one. This is discussed the section "DAG Compression" below.

We have seen that avoiding reference chains can have drastic effect on the number of references and size of output produced. The price to be payed, is that we produce output sets of references at a definition. Let us now study the sizes of these sets. In the examples seen so far which produce reference chains (toc-*, toc-*-all, and toc-*-DAG) each reference set has size one. Let us consider a dtop that traverses the input and at each article runs the dtop of Example 9 over the children list. We obtain as many lists of references in the output as there are articles (roughly one million). The average size of these sets is 10.4. This means that the average number of children of an article is 10.4. Then the number of distinct references needed is 3 where the largest set has size 358. Consider now another dtop which produces separate lists of article, master, www, and PhD entries. Now there are four distinct references and four sets of references appear in the output. The sizes of these sets correspond to the numbers of article, etc. entries. Thus, the set for article has roughly above one million entries. Here, memory consumption goes up considerably due to reference chain avoidance from 5MB to 18MB of total memory. The output size on the other hand goes down from 570MB to 242MB. Different overheads are also obtained depending on the size of what is summarized by each set of references.

DAG compression. If a dtop copies, i.e., x_1 or x_2 appears multiple times in the right-hand side of a rule, then the output with references can be *smaller* than the conventional XML output. This is the case in toc-article-all: the subtrees of article nodes are produced twice, once in the special article list, and once in the copy of all items. Since these trees are however relatively small, the output with references is not smaller than the conventional XML output: the overhead is 1.17, i.e., there is still additional overhead produced by the references (see top row of Table 4). A more drastic example is toc-article-DAG. Here the size of the XML output (without references) is quadratic in the number of articles. For the output of our stream processor, however, the size is just linear. Thus, we get dramatically small overhead numbers: the output of our processor is 5 to 6 orders of magnitude smaller, see Table 5.

Running over inputs with references. We analyze the performance of our system when processing input streams that contain references already. To do this, we run several dtops in sequence. Our first experiment consists of iteratively applying the REV dtop. Clearly, after any even number of applications, we obtain an output stream that represents the original document; however, this stream contains references for each node, as those will never be removed anymore, even not by the

algo	#ref	output	over	MB/s
No-re	29.6M	1.56GB	$6.09e^{-6}$	9.27
Reuse	29.6M	1.18GB	$4.5e^{-6}$	9.19
Chain	24.3M	1.07GB	$4.1e^{-6}$	8.58
Inline	3M	629MB	$2.38e^{-6}$	10.79
Uncompressed		251TB		

Table 5 Toc-article-DAG over Dblp

dtop	algo	#ref	output	over	MB/s
2-copies	Chain	23M	0.97GB	1.09	9.66
m-2-copies	Chain	49M	1.90GB	2.12	3.18
	Inline	46M	1.83GB	2.04	3.29
3-copies	Chain	23M	0.99GB	0.74	9.53
m-3-copies	Chain	72M	2.84GB	2.11	2.76
	Inline	67M	2.78GB	2.07	2.72

Table 6 Copying articles and processing each copy in a different state.

Inline mode. For No-reuse and Reuse, we get increasing amounts of overhead, with each additional application of REV. This is because reference chains are introduced for the sequence of flippings of x_1 and x_2 induced by the composition of REVs. If we run Chain or Inline, then these chains are removed and the overhead stays constant for any number of application of REV. The experiment shows that reference chain removal is a crucial ingredient for obtaining constant memory streaming!

The second experiment consists of first running a dtop that makes two copies of each article subtree. This means that the output stream of this dtop is a DAG where each article is referenced twice. Then we run another dtop on this stream, which processes the two article copies in distinct states. Thus, the DAG sharing of articles present in the input stream is broken by the second dtop, by introducing distinct labels for the two copies. Clearly, the overhead is increased, because we do not have sharing anymore. However, the overhead is not large (around 2, see Table 6). If we run the same experiment but with 3 instead of 2 copies, then the overhead stays essentially identical. If we compare the numbers with the unfolding of the DAG, i.e., we produce the output of the first translation as ordinary XML without references, and then run the second dtop m-2-copies, then we obtain only 10M references and an overhead of 1.27. Interestingly, this difference stays constant when iterating the idea several times (selecting one of the articles copies to be copied in the next round). The experiment shows that introducing and then breaking DAG sharing, introduces additional overhead caused by references, but this overhead is not large. In fact, we have tried various combinations of compositions in the quest to get large overhead numbers, but were not able to find anything that produced an overhead larger than 3. This strengthens our belief that adding references in streams is beneficial: it introduces little overhead, while allowing bounded height memory streaming of a large class of transformations.