ISSN 1884-0760

GRACE TECHNICAL REPORTS

An Order-Sensitive Fusion for XQuery

Hiroyuki Kato Soichiro Hidaka Zhenjiang Hu
Keisuke Nakano Yasunori Ishihara

GRACE-TR 2009–04 September 2009

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

The GRACE technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

An Order-Sensitive Fusion for XQuery

Hiroyuki Kato Soichiro Hidaka Zhenjiang Hu
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo 101-8430, Japan
{kato,hidaka,hu}@nii.ac.jp

Keisuke Nakano
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi
Tokyo 182-8585, Japan
ksk@cs.uec.ac.jp

Yasunori Ishihara
Osaka University
1-5 Yamadagaoka, Suita-shi
Osaka 565-0871, Japan
ishihara@ist.osaka-u.ac.jp

September 2, 2009

Abstract

In XQuery, composite expressions with node creation are typical in practice, for example, in data integration systems for XML with XQuery as schema mapping in addition to the classical view resolution. We propose a fusion algorithm for this kind of composite XQuery expressions. In developing the XQuery fusion, there is a problem that naive elimination of node creations does not preserve the semantics of XQuery with constraints of the order of nodes. We solve this problem by introducing an adornment code called extended Dewey's assigned to the occurrences of expressions. In this paper, we show that XML fragments created dynamically as intermediate results in a store can be emulated statically in such a way that rewriting XQuery expressions to avoid redundant parts is enabled using the extended Dewey's. The experimental results show that under a multi-step schema mapping scenario, our prototype system successfully eliminates execution cost of redundant node creations produced as intermediate results.

1 Introduction

An XML document is modeled as an ordered tree based on document order which is the preorder in the tree. Document order is a total order defined over the nodes

in a tree. This order plays an important role in the semantics of XQuery, especially in node creations and axis accesses. An XQuery expression is evaluated against an XML store which contains XML fragments with their document order. This store contains the fragments that are created as intermediate results, in addition to initial XML documents Hidders et al. [2004]. A node creation by an element constructor generates a new node which is placed at an arbitrary position in document order between the already existing trees. An axis access by a step expression returns its result in document order and without duplicates.

Rewriting composite expressions based on eliminating intermediate results generated by redundant expressions is a traditional optimization technique (known as fusion) Wadler [1988], Chin [1992], Fegaras and Maier [2000] in both programming languages community and database community. In XQuery, composite expressions for node creation are typical in practice, for example, in data integration systems for XML with XQuery as schema mapping Tatarinov and Halevy [2004]. We propose, in this paper, a fusion algorithm for this kind of composite XQuery.

In developing the XQuery fusion, there is a problem that naive elimination of node creations does not preserve the semantics of XQuery with constraints of the order of nodes. The XQuery fusion is more difficult than the existing fusion Wadler [1988], because naive elimination of node creations does not preserve document order. For example, it is incorrect to transform $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c$ into $\$v/c$. For an arbitrary store — assuming identical bindings of the externally defined variable $\$v$ — both expressions always return a value equivalent data, in the sense that they produce the same results when they are serialized and output by the query processor as a final result. However, as intermediate results in a query processor, two data evaluated by these expressions populate in different document order. When $\$v/c$ does not result in an empty sequence [1], the nodes produced by the former populate in the new document order created by the element constructor $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle$, whereas the nodes returned by the latter populate in the document order existing in the input store. Consequently, if we take a further step along parent axis for both expressions, namely, $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c/..$ and $\$v/c/..$, now it is easy to see the differences since the former results in a node created by the t element, whereas the latter results in a sequence of nodes bound to $\$v$. Therefore, eliminating redundant expressions including node construction and preserving document order are conflicting requirements. The purpose of our work is to meet these two conflicting requirements.

Although expressions that contain element constructors are non-deterministic with respect to document orderPage et al. [2005], we notice two properties that (1) a node generated by an element constructor is placed at the first position of the document order defined by the element constructor, (2) nodes in a sequence generated by expressions occurring inside the element constructor are copied deeply and

[1] To simplify the discussion, we do not consider in this paper, the case that $\$v/c$ results in an empty sequence. This is included in our future work.

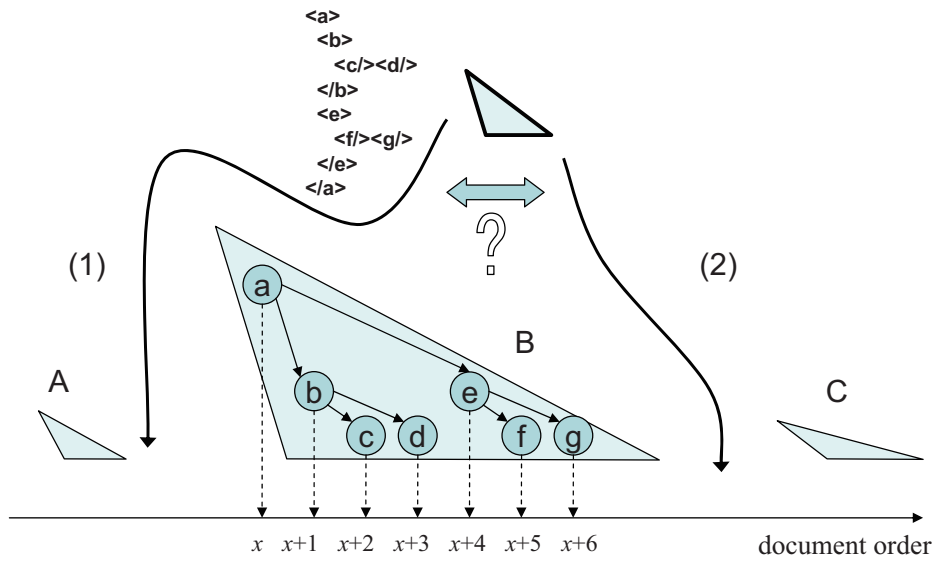

Figure 1: Node creation in the document order

placed following the node in (1) above with preserving the order in the sequence. These properties enable us to emulate newly created document order, statically.

In this paper, we propose deforestation techniques for XQuery. The kind of queries on which those techniques apply are common in practice due to the use of XQuery both as a language for producing XML views from legacy sources and to query XML such as XML data integration systems with XQuery as schema mapping. In addition, constructing trees in XQuery is expensive due to the complexity of the XML data model, making the proposed optimization particularly significant. The fundamental idea is to support static-time rewritings for queries which perform XML navigation over newly constructed trees. Preserving the semantics of the original code is particularly challenging in the XQuery context due to the importance of node identify and document order in the XML data model and in XPath. To address those issues, we show that XML fragments created dynamically as intermediate results in a store can be emulated statically in such a way that rewriting XQuery to avoid redundant expressions is enabled. This emulation is achieved by using an adornment code called the extended Dewey's assigned to the occurrences of expressions. The Dewey encoding has been used in index structure for XML documents Lu et al. [2005], Tatarinov et al. [2002]. We have extended the Dewey code to be suitable for the semantics of XQuery, especially for "for" expressions. Note that no schema information is required in doing this rewriting.

Our main contributions can be summarized as follows.

- We show that a static emulation of XML store can be achieved by using an extended Dewey code, which preserves the document order in terms of

expressions.

- By using this static emulation, we propose an XQuery fusion so that unnecessary element constructions are avoided while preserving the document order in XML.
- We show actual evaluation numbers by experimental results using a prototype system to give an understanding of the performance benefits.

This paper is organized as follows. After explaining our static emulation of store in Section 2, we show how fusion transformation can be correctly performed by partial evaluation of expression based on three fusion rules in Section 3. Some properties on the extended Dewey code are described in Section 4. The revised version of our fusion which handles failure cases is described in Section 5. Our experimental results are shown in Section6. We conclude the paper in Section 7.

Related work There are several studies on rewriting XQuery into XQueryGueni et al. [2008], Page et al. [2005], Koch [2005], Tatarinov and Halevy [2004]. In these, the most related is Gueni et al. [2008] in a sense of eliminating redundant expressions. In Gueni et al. [2008], the authors have proposed a rewriting optimization that replace the expressions, which return empty sequences, with () by the emptiness detection based on static analysis. Compared with this, our rewriting is to eliminate redundant element constructors as well as to detect emptiness. KochKoch [2005] and Page et al.Page et al. [2005] introduced some classes for composite XQuery and proposed XQuery-to-XQuery transformations over the classes of XQuery they defined. Their target queries don't contain newly constructed nodes. In real world, however, practical expressions such as schema mapping always returns newly constructed elements. Tatarinov and Halevy proposed an efficient query reformulation in data integration systems, in which XML and XQuery are used for data model and schema mapping, respectively Tatarinov and Halevy [2004]. In this system, composition of element construction is typical because the schema mapping that maps some element to other element involves element construction. They treat actual reformulation algorithm as a black box. Our work attempts to open the box and exploit some properties in this box.

2 Static Emulation of Store

This section describes how to achieve a static emulation of XML store. First, we introduce a notion of simple XML store using Dewey code and its order in Section 2.1. Then, we explain our static emulation of store by using the extended Dewey code and its order.

Before explaining our static emulation of store, we describe the relationship of node creation in XQuery and the document order. Figure 1 shows the treatment of newly created nodes by an element constructor relative to existing nodes in the

store. An element constructor that is depicted in the upper center part of the figure produces tree structure just below the expression (B) within which nodes are given order in one-dimensional document order axis. For example, if the topmost node named "a" is given order x, then its first child node named "b" is given order that is strictly greater than x, say, x+1, which is also strictly less than the order given to its children named "c" and "d". These ordering is guaranteed to be consistent between elements created in a common element constructor.

On the other hand, order between nodes that are separately created by different element constructors in a query is implementation dependent. For example, consider the following expression `(Q1)` in XQuery.

Q 1. $(\langle h\rangle\langle i/\rangle\langle/h\rangle, \langle j\rangle\langle k/\rangle\langle/j\rangle)$

In this query, the document order between the tree with root node named "h" and the one with root node named "j" is implementation dependent. So, no one can decide the order of these two nodes by static analysis. In addition, the document order between the existing nodes – like A and C in the figure – and a newly created node is also implementation dependent, thus static analysis can not decide this order either. However, overlap along document order axis never happens between these nodes. Extended Dewey order defined in this section is designed to respect all these properties, namely, (a) no order is predefined statically across nodes that are separately created in different element constructors in a query, (b) preorder is defined between nodes inside an element constructor, (c) orders given to elements that belong to different roots of trees are pair-wise disjoint.

XML store is used in the semantics of XQuery Hidders et al. [2004] [2] while our algorithm is based on a static analysis. In this section we show that a static emulation of XML store can be achieved by using an extended Dewey order, which preserves the document order in terms of expressions.

2.1 Simple XML Store using Dewey Order

Dewey Order encoding of XML nodes is a lossless representation of a position in document order Lu et al. [2005], Tatarinov et al. [2002]. In Dewey Order, each node is represented by a path from a root using ".", which is depicted by D in Figure 2: (1) a root is encoded by $r \in \mathcal{S}$ where $\mathcal{S}$ is a countably infinite set of special codes; (2) when a node a is the n-th child of a node b, the Dewey code of a, $did(a)$, is $did(b).n$. Note that ϵ in Figure 2 is used for a termination, so every Dewey code ends with ϵ.

Using Dewey encoding, sorting and duplicate elimination in document order can be achieved by a straightforward way. Now, simple XML store, in which nodes are restricted to element nodes — other nodes such as attributes are disregarded here — is defined by an ordered tree representation using Dewey codes instead of

[2]The semantics of XQuery is formally given by World Wide Web Consortium [2007]. However, due to not being self containedMelton [2008] and to simplify the discussion, we refer Hidders et al. [2004] instead.

$$
\begin{array}{lcll}
D & ::= & r\ X & r \in \mathcal{S}, \mathcal{S} \text{ is a set of special codes.} \\
X & ::= & \epsilon \mid .B & \\
B & ::= & n\ X & n \in \mathcal{I}, \mathcal{I} \text{ is a set of integers.}
\end{array}
$$

Figure 2: Pure Dewey code

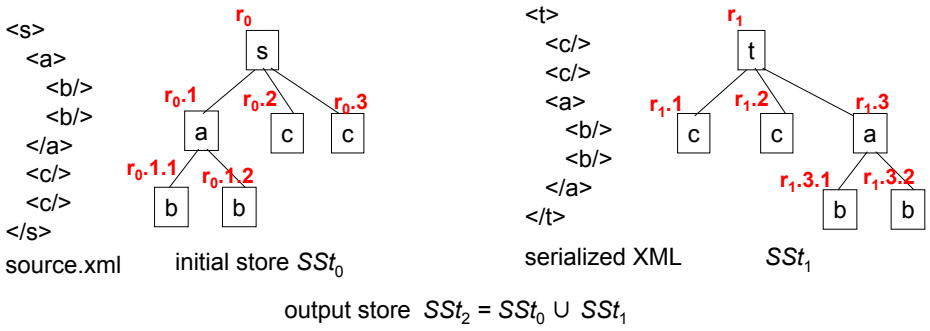

Figure 3: An input document source.xml, SSt_0 and output store by `(Q2)`.

nodes and edges in Hidders et al. [2004]. We assume a set of names $\mathcal{N}$ used for element names and a countably infinite set of Dewey Code $\mathcal{D}$, which is depicted by D in Figure 2. Both the strict partial order $<$ and the equality $=$ on $\mathcal{D}$ are straightforward.

DEFINITION 2.1 (Simple XML Store). *A simple XML store is a 3-tuple* $SSt = (D, \nu)$ *where, (a)* D *is a finite subset of* $\mathcal{D}$*; (b)* $\nu : D \mapsto \mathcal{N}$ *maps the Dewey codes to their node name.*

Evaluating an element constructor against an input simple store will add a tree into the input store. Consider the following XQuery expression `(Q2)` when given the input document `source.xml` shown in Figure 3.

Q 2.
```
let $v := doc("source.xml")/s
          return <t>$v/c,$v/a</t>
```

For an initial store $SSt_0 = (D_0, \nu_0)$ where,

- $D_0 = \{r_0, r_0.1, r_0.1.1, r_0.1.2, r_0.2, r_0.3\}$ where $r_0 \in \mathcal{S}$;
- $\nu_0(r_0) = \mathtt{s},\ \nu_0(r_0.1) = \mathtt{a},\ \nu_0(r_0.1.1) = \nu_0(r_0.1.2) = \mathtt{b}$, $\nu_0(r_0.2) = \nu_0(r_0.3) = \mathtt{c}$ where $\{\mathtt{s}, \mathtt{a}, \mathtt{b}, \mathtt{c}\} \subset \mathcal{N}$,

evaluating `(Q2)` updates SSt_0 into $SSt_2(D_2, \nu_2)$ where,

- $D_2 = D_0 \cup \{r_1, r_1.1, r_1.2, r_1.3, r_1.3.1, r_1.3.2\}$ where $r_1 \in \mathcal{S} \wedge r_1 \neq r_0$

- $\nu_2 = \nu_0 + \{r_1 \mapsto \texttt{t},\ r_1.1 \mapsto \texttt{c},\ r_1.2 \mapsto \texttt{c},$
  $r_1.3 \mapsto \texttt{a},\ r_1.3.1 \mapsto \texttt{b},\ r_1.3.2 \mapsto \texttt{b}\}$
  where $\texttt{t} \in \mathcal{N}$

This updating is achieved by the following steps in a recursive way for nested element constructors. (i) Generate a new root code $r \in \mathcal{S}$ for an element constructor. (ii) Reassign Dewey codes for values produced by evaluated expressions occurring inside the element constructor. Note that once the data is serialized, the information about document order associated with nodes is lost.

2.2 Emulating Simple Store

In this subsection, we will show that static emulation of newly created XML fragments in simple store is achieved by using the extended Dewey Order encoding of expressions. The purpose of this encoding is to allow operation like sorting, axis access and duplicate elimination on expression rather than on the dynamic store.

When expressions contain element constructors, the semantics of XQuery requires; (1) a node generated by an element constructor is placed at the first position of the document order defined by the element constructor, (2) nodes in a sequence generated by expressions occurring inside the element constructor are copied deeply and placed following the node in (1) above with preserving the order in the sequenceHidders et al. [2004]. This requirement leads to the following properties. Note that for an expression e we use $[\![e]\!]$ for Dewey Order encoding of evaluated data against an arbitrary store (D, ν).

PROPERTY 2.2. *For an element constructor, $\langle en \rangle e \langle /en \rangle$, where en is an element name and e is an expression,*

(i) $[\![\langle en \rangle e \langle /en \rangle]\!] = r$ *where* $r \in \mathcal{S} \wedge r \notin D$

(ii) $\forall d \in [\![e]\!], d = [\![$ <en>e</en> $]\!].n$[3] *where n is an integer.*

(iii) when e is a sequence constructor (e_1, e_2),
$\forall d_1 \in [\![e_1]\!] \forall d_2 \in [\![e_2]\!], d_1 < d_2$

Figure 4 shows this property using concrete examples. This property enables us to statically emulate newly created XML fragments — created by element constructors — in simple store. This emulation is achieved by Dewey encoding of expressions which exploits PROPERTY 2.2.

In this paper, as will be seen in next section we extend Dewey code and its order by introducing new delimiter "#" to be suitable for the semantics of "for" expressions in XQuery. From now on to the end of this section, we will see the property of the "for" expressions occurring inside element constructors and describe the role of the new delimiter "#". Figure 4 shows such a property of the "for" expression `(Q3)`, bellow.

[3] We use $\in$ for sequence containment. And we treat an item identically to a sequence containing only that item as in the semantics of XQuery.

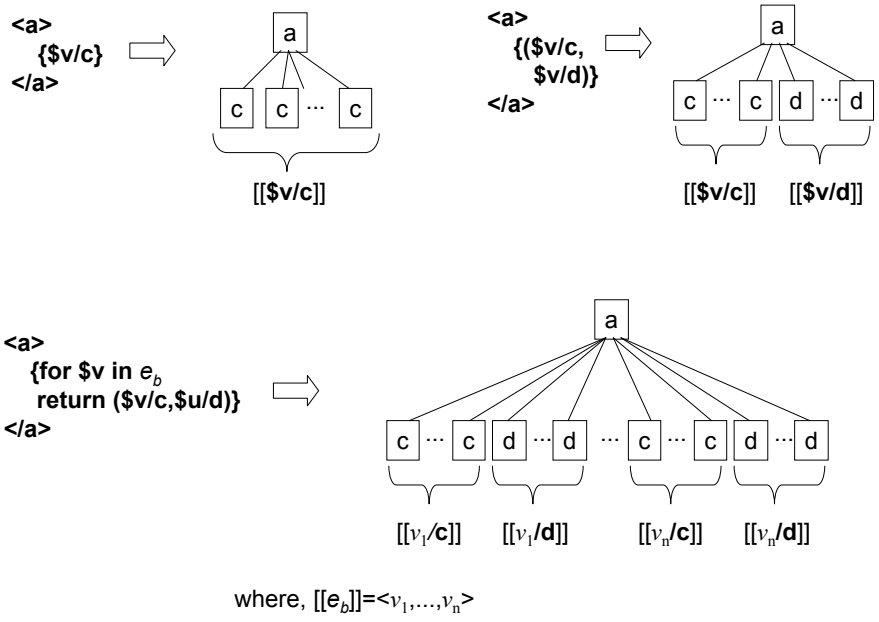

Figure 4: A simple example for the document order in element creations

Q 3. $\langle a \rangle$ for $\$v$ in e_b return $(\$v/c, \$v/d)\langle/a\rangle$

The semantics of a "for" expression is to evaluate the "return" expression k times where k is the length of the sequence, which is the result of the expression followed by "in". So, for the ordered tree which is the result of `(Q3)`, the child nodes of the root are the sequence of elements, which is the deep copied sequence of the result of evaluating $(\$v/c, \$v/d)$ zero or more times.

For the expressions `(Q3)`$/d$ and `(Q3)`$/c$ we can get easily the *value equivalent* expressions `(Q4)` and `(Q5)`, respectively.

Q 4. for $\$u$ in e_b return $\$v/d$

Q 5. for $\$u$ in e_b return $\$v/c$

Then, consider the expression $((\texttt{(Q4)}), (\texttt{(Q5)}))/\texttt{self} :: *$. As described in the previous subsection, since axis access by "/" requires the sorting and duplicate elimination in document order, the correct transformation of this expression should result in `(Q6)`, in which two "for" expressions `(Q4)` and `(Q5)` are merged, sorting expressions appeared in the "return" expression.

Q 6. for $\$u$ in e_b return $(\$v/c, \$v/d)$

To this end, we extend the Dewey codes and its order with new structure "#" to represent the order of expressions occurring in the "return" expressions in "for"

e	::=	c	constants
	|	\$v	variables
	|	$(e, e, ..., e)$	sequence constructions
	|	$e/\alpha::en$	location step expressions
	|	for \$v in e return e	for-exp.
	|	let \$v := e return e	let-expressions
	|	$\langle en \rangle e \langle /en \rangle$	element constructor

Figure 5: XQuery

B	::=	$(n\|?)X$	$n \in \mathcal{I}$
X	::=	$\epsilon \mid .B \mid \#[B, \ldots, B]$	
D	::=	$B \mid \epsilon \mid r\ X \mid \#\ [D, \ldots, D]$	$r \in \mathcal{S}$

Figure 6: Abstract syntax of the extended Dewey code

expressions. As will be seen the next section, when we have a same prefix until "#" delimiter for given two extended Dewey codes, the sorting and duplicate elimination on this codes requires merging into one code to merge two "for" expressions into one "for" expression.

For example, the extended Dewey encoding of the "for" expression in `(Q3)` is $r.1\#[1, 2]$, where r is the extended Dewey encoding of `(Q3)`. And the extended Dewey encoding of `(Q4)` and `(Q5)` are $r.1\#2$ and $r.1\#1$, respectively.

3 Algorithm Overview

In this section, we briefly overview our algorithm for automatic fusion of XQuery expressions so that unnecessary element constructions can be correctly eliminated. The purpose of this section is on conveying the essential idea we use. The detailed and precise algorithm will be given in Section 5. Basically, we will focus on fusing the following subexpression

$$e/\alpha::en$$

so that unnecessary element construction in the query expression in e is eliminated under the context of "selection" by $\alpha::en$.

Note that the XQuery fusion does not always succeed. We will describe the algorithm handling failure cases in Section 5.

3.1 Annotated XQuery Expressions

We consider the XQuery expressions defined in Figure 5. A query expression can be a constant c, a variable $\$v$, a sequence expression $(e_1, \ldots, e_n)$ where each subexpression e_i is not a sequence expression, a location step expression $e/\alpha::en$ where α is an axis which can be *child*, *self*, or .. (parent), and en is a name

$$
\begin{array}{rcl}
e^d & ::= & \mathsf{c}^d \mid \$\mathtt{v}^d \mid (e^d, e^d, ..., e^d)^d \mid (e^d/\alpha{::}en)^d \\
& \mid & (\mathsf{for}\ \$\mathtt{v}\ \mathsf{in}\ e^d\ \mathsf{return}\ e^d)^d \\
& \mid & (\mathsf{let}\ \$\mathtt{v} := e^d\ \mathsf{return}\ e^d)^d \\
& \mid & (\langle en\rangle e^d \langle/en\rangle)^d
\end{array}
$$

Figure 7: Annotated XQuery

$$\frac{}{\Gamma \vdash \mathsf{c} \to \mathsf{c}^{\epsilon}} \quad (\textsc{Pcst})$$

$$\frac{\Gamma(v).btype = \text{"let"}}{\Gamma \vdash \$v \to \Gamma(v).expr} \quad (\textsc{Plvar})$$

$$\frac{\Gamma(v).btype = \text{"for"}}{\Gamma \vdash \$v \to \$v^1} \quad (\textsc{Pfvar})$$

$$\frac{\Gamma \vdash e_1 \to e_1'^{d_1} \qquad \ldots \qquad \Gamma \vdash e_N \to e_n'^{d_N}}{\Gamma \vdash (e_1, \ldots, e_N) \to \mathsf{flatten}\ (e_1'^{d_1}, \ldots, e_N'^{d_N})^{[d_1,\ldots,d_N]}} \quad (\textsc{Pseq})$$

$$\frac{\Gamma \vdash e_1 \to e_1'^{d_1} \qquad \Gamma \cup \{v \mapsto (e_1'^{d_1}, \text{"let"})\} \vdash e_2 \to e_2'^{d_2}}{\Gamma \vdash \mathsf{let}\ \$v := e_1\ \mathsf{return}\ e_2 \to e_2'^{d_2}} \quad (\textsc{Plet})$$

$$\frac{\Gamma \vdash e_1 \to e_1'^{d_1} \qquad \Gamma \cup \{v \mapsto (e_1'^{d_1}, \text{"for"})\} \vdash e_2 \to e_2'^{d_2}}{\Gamma \vdash \mathsf{for}\ \$v\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \to (\mathsf{for}\ \$\mathtt{v}\ \mathsf{in}\ \mathsf{e}_1'^{\mathtt{d}_1}\ \mathsf{return}\ \mathsf{e}_2'^{\mathtt{d}_2})^{\#\mathtt{d}_2}} \quad (\textsc{Pfor})$$

$$\frac{e \to e'^{d} \qquad e''^{d'} = \mathsf{child\_fusion}\, e'^{d}\ en}{\Gamma \vdash e/\,\mathtt{child} :: en \to e''^{d'}} \quad (\textsc{Pcstp})$$

$$\frac{e \to e'^{d} \qquad e''^{d'} = \mathsf{self\_fusion}\, e'^{d}\ en}{\Gamma \vdash e/\,\mathtt{self} :: en \to e''^{d'}} \quad (\textsc{Psstp})$$

$$\frac{e \to e'^{d} \qquad e''^{d'} = \mathsf{parent\_fusion}\, e'^{d}\ en}{\Gamma \vdash e/\,\mathtt{parent} :: en \to e''^{d'}} \quad (\textsc{Ppstp})$$

$$\frac{\Gamma \vdash e \to e'^{d_1} \qquad d_2 = new\ rootD \qquad e'^{d_3} = \mathsf{dc\_assign}\ e'\ d_2.1}{\Gamma \vdash \langle en\rangle e \langle/en\rangle \to \langle en\rangle e'^{d_3} \langle/en\rangle^{d_2}} \quad (\textsc{Pelm})$$

Figure 8: An Overview of Our XQuery Fusion

test which can be a tag name or $*$ (an arbitrary tag), a "for" expression, a "let" expression, or an element construction expression $\langle en\rangle e\langle/en\rangle$.

As seen in the introduction, to guarantee the correct transformation, we should pay attention to the context and the order of subexpressions. To this end, we would like to associate all expressions, old and new in the later transformation, with an extended Dewey code. Recall that the usual Dewey code is basically in the form of a path encoding such as $r.3.2$ (which denotes a subexpression which is the second subexpression of the third subexpression of the expression with code r.) The extension is the code of the form $r\#[d_1, \ldots, d_n]$ for the "for" expression, where d_i's are again the extended Dewey code. The formal definition of the extended Dewey

code is given in Figure 6. Informally, we may consider it as

$$d ::= \ \epsilon \mid r.d \mid r\#[d_1, \ldots, d_m]$$

where ϵ denotes the unknown code. This code is assigned to both expressions which are occurred in outside of element constructors and ones which cannot be partial evaluated. Again note that the XQuery fusion does not always succeed. The algorithm handling the fail case is shown in Section 5.

The partial order on the extended Dewey codes are essentially the dictionary order. For example, $r.1.2 < r.1.3$, $r.1 < r.1.2$ hold. But the following pairs of codes are incomparable: (ϵ, r) is incomparable because ϵ is the unknown code; (r,r') is incomparable if $r \neq r'$; and ($r.1\#[3]$, $r.1\#[1,2]$) is incomparable because they represent interleaved document orders of the elements produced by a "for" expression. Whereas, as will be seen later, sorting and duplicate elimination on $[r.1\#[3], r.1\#[1,2]]$ results in $r.2\#[1,2,3]$ to merge two "for" expressions into one "for" expression because they have the same prefix until $\#$, say $r.1$. This operation will be appeared as remove_duplicate (dc_sort $[r.1\#[3], r.1\#[1,2]]$) in Section 3.2 and as $[] \sim_D \uplus^{\oplus_D}_{\prec_D} [r.1\#[3], r.1\#[1,2]]$ in Section 4.

Now we can add annotations of the extended Dewey codes to XQuery expression as in Figure 7. We sometimes omit the annotation if it is clear from the context. To simplify our presentation, we will assume that there is a global environment for storing all annotated expressions during our fusion transformation, and a function

$$getExpGlobal(r)$$

that can be used to extract the expression whose code is r from the global environment.

3.2 Fusion Transformation

Figure 8 summarizes our fusion transformation on XQuery expressions. This fusion is defined in terms of a set of inference rules. In these rules, a judgment of the form[4]

$$\Gamma \vdash e \rightarrow e^d$$

indicate that, for a given environment Γ (mapping XQuery variables bound by "let" or "for" to annotated expressions), the XQuery expression e complies into the annotated expressions e^d. As will be seen later, the annotation is used to keep track of information of the order and the context among expressions, and it plays an important role in our fusion transformation. When the fusion transformation is finished, we can ignore all the annotation and give a normal XQuery expression as the final result.

The definition of our fusion in Figure 8 is rather straightforward. For a constant expression c, we return itself but annotate it with the Dewey code ϵ. For a

[4]This form gives the intuition of our fusion algorithm. This will be changed later.

$$\mathsf{dc\_assign}\ c^{\text{-}}\ r = c^r \qquad \mathsf{dc\_assign}\ \$v^{\text{-}}\ r = \$v^r$$

$$\mathsf{dc\_assign}\ (e/c)^{\text{-}}\ r = (e/c)^r \tag{DCSTP}$$

$$\mathsf{dc\_assign}\ (e_1, \ldots, e_n)^{\text{-}}\ r = (e_1, \ldots, e_n)^{[r_1,\ldots,r_n]}$$

$$\text{where } r_0 = r\ \ e'_i = \mathsf{dc\_assign}\ e_i\ r_{i-1}\ \ r_i = \mathsf{succ}(\mathsf{extract\_dc}\ e'_i) \tag{DCPSEQ}$$

$$\mathsf{dc\_assign}\ (\texttt{<}t\texttt{>}e\texttt{</}t\texttt{>})^{\text{-}}\ r = \texttt{<}t\texttt{>}e'\texttt{</}t\texttt{>}^r$$

$$\text{where } e' = \mathsf{dc\_assign}\ e_i\ r.1 \tag{DCPEC}$$

$$\mathsf{dc\_assign}\ (\mathsf{for}\ \$v\ \mathsf{in}\ e_0\ \mathsf{return}\ e)^{\text{-}}\ r = (\mathsf{for}\ \$v\ \mathsf{in}\ e_0\ \mathsf{return}\ e')^{r\#bs}$$

$$\text{where } e' = \mathsf{dc\_assign}\ e\ 0\ \ bs = \mathsf{extract\_dc}\ e' \tag{DCPFOR}$$

Figure 9: Dewey code propagation

variable, if it is bounded by the outside "let", we retrieve its corresponding annotated expression from the environment, otherwise it must be a variable bound by the outside "for" and we put the extended Dewey code 1 to the variable when its corresponding expression has an extended Dewey code except for ϵ. Otherwise we put ϵ to the variable bound by the outside "for". Note that the reason why the annotation of the variable bound by "for" always is 1 is described later. for a sequence expression, we partially evaluate each element expression, and then group them to a new sequence annotated with a Dewey that are gathered from the result of each element expression. Note that we use $\mathsf{flatten}$ to remove nested sequences (e.g., $\mathsf{flatten}((e_{11}^{r_1}, e_{12}^{r_2})^{[r_1,r_2]}, e_3^{r_3})^{[[r_1,r_2],r_3]} = (e_{11}^{r_1}, e_{12}^{r_2}, e_3^{r_3})^{[r_1,r_2,r_3]}$). For a location step expression $e/\alpha{::}en$, we perform fusion transformation to eliminate unnecessary element construction in e after partially evaluating e. We will discuss the definitions of the three important rules (PCSTP),(PSSTP) and (PPSTP) in Section 3.2.2. For a "let" expression, we first partially evaluate the expression e_1, and then partially evaluate e_2 with an updated environment and return it as the result. Note that this rule eliminate variables bound by "let" by expanding variables using Γ. For a "for" expression, we do similarly as for a "let" expression except that we finally produce a new "for" expression by gluing partially evaluated results together. For an element construction, after partially evaluating its content expression e to e', we create a new Dewey code for annotating this element, and propagate this Dewey code information to all subexpression in e' (with function $\mathsf{dc\_assign}$) so that we can access (recover) this element constructor when processing subexpressions of e'. It is this trick that helps solving the problem in $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c/..$ in Introduction. We will discuss this Dewey code propagation in Section 3.2.1.

3.2.1 Dewey Code Propagation

Propagating the Dewey code of an element construction to its subexpressions(content expressions) plays an important role in constructing our rules (Section 3.2.2) for correct fusion transformation.

Figure 9 defines a function dc_assign $e\ r$:

$$\mathsf{dc\_assign} :: XQuery^D \rightarrow D \rightarrow XQuery^D$$

which is to propagate the Dewey code r into an annotated expression e by assigning proper new Dewey codes to e and its subexpressions. We will explain some important equations in this definition. Note that we write e^- to denote that the Dewey code of e is "don't care".

The equation (DCPSEQ) places horizontal numbering to sequence expressions. Function succ is used to enforce numbering using strictly greater value relative to previously processed expressions (e.g., succ $r.1 = r.2$). (DCPEC) introduces vertical structure to the numbering by initiating dc_assign for subexpression e by adding ".1" to its second parameter. The equations that needs additional attention is (DCSTP) and (DCPFOR) above. In (DCSTP), it may seem unusual for dc_assign not to recurse subexpression e. However, considering that path expression itself do not introduce additional parent-child relationship, and that dc_assign always handle expressions that is already partially evaluated, there is no additional chance to simplify the path expression further using Dewey code allocated to the subexpression. Particularly characteristic equation (DCPFOR), which introduces # structure to the Dewey code, numbers the expression e at return clause. Note that the second parameter to recursive call for e is reset to 0. bs that reflects the horizontal structure produced by the return clause is combined by the # sign to produce $r\#bs$ as the top level code allocated to the "for" expression.

3.2.2 Fusion Rules

Our fusion transformation on $e/\alpha::en$ is based on the three fusion rules (functions) (PCSTP), (PSSTP) and (PPSTP) in Figure 10, which correspond to three types of axis. The basic procedure is as follows: (1) Extract (get) subexpressions according to the axis α; (2) Select those who produce nodes whose name is equal to the tag name en using a filter; (3) Sort the remained subexpressions according to their Dewey codes; (4) If the above sort step succeeds, we remove the duplicated subexpressions and return its sequence as the result, otherwise we give up fusion.

More concretely, consider the definition of child_fusion. We use get_children e to get a sequence of subexpressions that contribute to producing children of the XML document that can be obtained by evaluation of e, and use filter(equal_to en) function to keep those that are equal to en where filter $p\ xs = [x \mid x \leftarrow xs, p\ x]$. The resulting sequence expression is sorted according to their Dewey codes by dc_sort. Since not all Dewey codes are comparable, we may fail in this sorting. If the sorting succeeds, we return a sequence expression by removing all duplicated

element subexpressions (remove_duplicate), otherwise we give up fusion by returning the original expression $e/\,\texttt{child}::en$. We will show the precise algorithm handling this fail case in Section 5.

3.2.3 Examples

We demonstrate our fusion transformation by using some examples. For readability, we use "/" for "child::" and "/.." for "parent::".

First, Figure 11 shows how an example of our fusion transformation for $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c$ shown in Section 1. Note that for a space limitation, we use (A), (B), (C) and (D) instead of (PCSTP), (PELM), (PSEQ) and (PFVAR), respectively in Figure 11. So, for $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c/..$, which is also from Section 1, We can get the correct expression by using the following transformation;

$$\frac{\begin{array}{c}\Gamma \vdash \langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c \rightarrow (\$v/c)^{d_1.1} \\ \langle t\rangle(\$v/c, \$v/a)\langle/t\rangle^{d_1} = \mathsf{parent\_fusion}\ (\$v/c)^{d_1.1} * \end{array}}{\Gamma \vdash \langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c/.. \rightarrow \langle t\rangle(\$v/c, \$v/a)\langle/t\rangle^{d_1}} \quad \text{(PPSTP)}$$

Next, consider the following expression `(Q7)`,

Q 7. let $\$v := \langle a\rangle()\langle/a\rangle$ return $(\$v, \$v)/\,\texttt{self}::a$

In `(Q7)`, the subexpression $(\$v, \$v)/\,\texttt{self}::a$ is redundant because duplicate elimination is needed for this subexpression. Figure 12 shows this transformation.

For more complicated case, we show that our extended Dewey order encoding of "for" expressions occurring inside an element constructor requires to append "#" to the extended Dewey code. This is the prominent feature of our extension to the extended Deweys' which is explained using $((\texttt{(Q4)}), (\texttt{(Q5)}))/\,\texttt{self}::*$ described in Section 2.2. Before we explain this query, consider `(Q3)` which is also described in Section 2.2. Partial evaluation of `(Q3)` assigns the extended Dewey code shown in Figure 13. So, $(\texttt{(Q3)}/d)$ is transformed in the way shown in Figure 14. Also, $(\texttt{(Q3)}/c)$ is transformed in the way shown in Figure 15. Now, return to $(\texttt{(Q3)}d/, \texttt{(Q5)}/c)/\,\texttt{self}::*$. With the above results, the partial evaluation performs shown in Figure 16

4 Properties on Extended Dewey Order

This section describes an algebraic structure of sorting and duplicate elimination in the extended Dewey Order.

Both sorting by dc_sort and duplicate elimination by remove_duplicate take place at the same time. This is achieved on a sequence of Dewey codes $[d_1, d_2, \ldots, d_n]$ by

$$[] \sim_D \uplus^{\oplus_D}_{\prec_D} [d_1, d_2, \ldots d_n]$$

$$\mathsf{child\_fusion} :: XQuery^D \to QName \to XQuery^D$$

$$\mathsf{child\_fusion}\ e^d\ en = \begin{cases} \mathsf{remove\_duplicate}\ (e'_1, ..., e'_N) & \text{if dc\_sort succeeds} \\ (e^d/\,\texttt{child} :: en)^\epsilon & \text{otherwise} \end{cases}$$

$$\text{where } (e'_1, ..., e'_N) = \mathsf{dc\_sort}(\mathsf{filter}(\mathsf{equal\_to}\, en)(\mathsf{get\_children}\ e)) \qquad \text{(cFusion)}$$

$$\mathsf{self\_fusion} :: XQuery^D \to QName \to XQuery^D$$

$$\mathsf{self\_fusion}\ e^d\ en = \begin{cases} \mathsf{remove\_duplicate}\ (e'_1, ..., e'_N) & \text{if dc\_sort succeeds} \\ (e^d/\,\texttt{self} :: en)^\epsilon & \text{otherwise} \end{cases}$$

$$\text{where } (e'_1, ..., e'_N) = \mathsf{dc\_sort}(\mathsf{filter}(\mathsf{equal\_to}\, en)(\mathsf{get\_self}\ e)) \qquad \text{(sFusion)}$$

$$\mathsf{parent\_fusion} :: XQuery^D \to QName \to XQuery^D$$

$$\mathsf{parent\_fusion}\ e^d\ en = \begin{cases} \mathsf{remove\_duplicate}\ (e'_1, ..., e'_N) & \text{if dc\_sort succeeds} \\ (e^d/\,\texttt{parent} :: en)^\epsilon & \text{otherwise} \end{cases}$$

$$\text{where } (e'_1, ..., e'_N) = \mathsf{dc\_sort}(\mathsf{filter}(\mathsf{equal\_to}\, en)(\mathsf{get\_parent}\ e)) \qquad \text{(pFusion)}$$

$$\mathsf{get\_children} :: XQuery^D \to XQuery^D$$

$$\mathsf{get\_children}\ c = ()^{[]} \quad \mathsf{get\_children}\ \$v^{\text{-}} = (\$v/\,\texttt{child} :: *)^\epsilon$$

$$\mathsf{get\_children}\ ()^{[]} = ()^{[]}$$

$$\mathsf{get\_children}\ (e_1, ..., e_N)^{\text{-}} = \mathsf{flatten}\ ((e'_1, ..., e'_N)^{[d_1,..,d_N]})$$

$$\text{where } e'_i = \mathsf{get\_children}\ e_i \quad d_i = \mathsf{extract\_dc}(e'_i) \qquad \text{(gcSeq)}$$

$$\mathsf{get\_children}\ (e/\,\texttt{child} :: en)^{\text{-}} = (e/\,\texttt{child} :: en/\,\texttt{child} :: *)^\epsilon$$

$$\mathsf{get\_children}\ (\langle en\rangle e^d \langle/en\rangle)^{\text{-}} = e^d \qquad \text{(gcEc)}$$

$$\mathsf{get\_children}\ (\mathsf{for}\ \$v\ \mathsf{in}\ e\ \mathsf{return}\ (e_1, ..., e_N))^{r\#[b_1,...,b_N]}$$

$$= \begin{pmatrix} \mathsf{for}\ \$v\ \mathsf{in}\ e\ \mathsf{return} & (e_{11}, e_{12}, \ldots, e_{1n_1}, \\ & e_{21}, e_{22}, \ldots, e_{2n_2}, \\ & \cdots \\ & e_{N1}, e_{N2}, \ldots, e_{Nn_n}) \end{pmatrix}^{r'}$$

$$\text{where} \quad (e_{i1}, \ldots, e_{in_i}) = \mathsf{get\_children}\, e_i \qquad r_{ij} = \mathsf{extract\_dc}\, e'_{ij}$$

$$r' = r\#\ [b_1.r_{11}, \ldots, b_1.r_{1n_1}, \quad b_2.r_{21}, \ldots, b_2.r_{2n_2}, \quad ... \quad b_N.r_{N1}, \ldots, b_N.r_{Nn_n}] \qquad \text{(gcFor)}$$

$$\mathsf{get\_self}, \mathsf{get\_parent} :: XQuery^D \to XQuery^D$$

$$\mathsf{get\_self}\ e^r = e^r \qquad \mathsf{get\_parent}\ e^{r.(n|?)} = getExpGlobal(r)$$

Figure 10: Fusion rules for three kinds of $axis$

$$\dfrac{\dfrac{\dfrac{\Gamma(v).btype = \text{"for"}}{\Gamma \vdash \$v \to \$v^1}\ (\text{D})}{\dfrac{(\$v/c)^\epsilon = \texttt{cf}\ \$v^1\ c}{\Gamma \vdash \$v/c \to (\$v/c)^\epsilon}\ (\text{A})} \cdots}{\dfrac{\Gamma \vdash (\$v/c, \$v/a) \to (\$v/c, \$v/a)^{[\epsilon,\epsilon]}}{\Gamma \vdash \langle t\rangle(\$v/c, \$v/a)\langle/t\rangle \to (\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle)^{d_1}}\ (\text{B})}\ (\text{C})\quad \begin{array}{l} d_1 = new\ rootD \\ (\$v/c, \$v/a)^{d'} \\ \quad = \texttt{da}\ (\$v/c, \$v/a)\ d_1.1 \\ d' = [d_1.1, d_1.2] \end{array}}{\Gamma \vdash \langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c \to (\$v/c)^{d_1.1}}\ (\text{A})$$

$$\texttt{cf}\ (\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle)^{d_1}\ c = (\$v/c)^{d_1.1}$$

where we use `cf` for child_fusion and `da` for dc_assign.

Figure 11: Our fusion transformation for $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c$.

$$\dfrac{\dfrac{d_1 = new\ rootD}{\Gamma \vdash \langle a\rangle\,()\,\langle/a\rangle \to \langle a\rangle\,()\,\langle/a\rangle^{d_1}}\ (\text{PELM}) \quad \dfrac{\dfrac{\dfrac{\Gamma'(v).expr = \langle a\rangle\,()\,\langle/a\rangle^{d_1}}{\Gamma' \vdash \$v \to \langle a\rangle\,()\,\langle/a\rangle^{d_1}}\ (\text{PLVAR}) \quad \dfrac{\Gamma'(v).expr = \langle a\rangle\,()\,\langle/a\rangle^{d_1}}{\Gamma' \vdash \$v \to \langle a\rangle\,()\,\langle/a\rangle^{d_1}}\ (\text{PLVAR})}{\Gamma' \vdash (\$v, \$v) \to (\langle a\rangle\,()\,\langle/a\rangle, \langle a\rangle\,()\,\langle/a\rangle)^{[d_1,d_1]}}\ (\text{PSEQ})}{\Gamma' \vdash (\$v, \$v)/\,\texttt{self} :: a \to \langle a\rangle\,()\,\langle/a\rangle^{d_1}}\ (\text{PSSTP})}{\Gamma \vdash \text{let}\ \$v := \langle a\rangle()\langle/a\rangle\ \text{return}\ (\$v, \$v)/\,\texttt{self} :: a \to \langle a\rangle\,()\,\langle/a\rangle^{d_1}}\ (\text{PLET})$$

where we use Γ' for $\Gamma \cup \{v \mapsto (\langle a\rangle\,()\,\langle/a\rangle^{d_1}, \text{"let"})\}$.

Figure 12: Our fusion transformation of let $\$v := \langle a\rangle()\langle/a\rangle$ return $(\$v, \$v)/\,\texttt{self} :: a$.

where binary operator $\sim_D \uplus^{\oplus_D}_{\prec_D}$ is defined below. Compatibility test between the members of a sequence of Dewey codes — failure of the test causes a failure of the partial evaluation (which is recovered at the caller of this operation by restoring the original expression) — and the unification of two Dewey codes (possibly leads to unification of two for expressions into one) are implemented using orderable and

$$\dfrac{\dfrac{\cdots}{\Gamma \vdash \text{for}\ \$v\ \text{in}\ e_b\ \text{return}\ (\$v/c, \$v/d)}\ (\text{PFOR})}{\begin{array}{l} \to (\text{for}\ \$v\ \text{in}\ e_b\ \text{return}\ (\$v/c, \$v/d))^\epsilon \\ d_1 = new\ rootD \\ \text{dc\_assign}\ (\text{for}\ \$v\ \text{in}\ e_b\ \text{return}\ (\$v/c, \$v/d))\ d_1.1 \\ = (\text{for}\ \$v\ \text{in}\ e_b\ \text{return}\ (\$v/c, \$v/d))^{r_1.1\#[1,2]} \end{array}}{\Gamma \vdash \texttt{(Q3)} \to \langle a\rangle\,\text{for}\ \$v\ \text{in}\ e_b\ \text{return}\ (\$v/c, \$v/d)\langle/a\rangle^{d_1}}\ (\text{PELM})$$

Figure 13: Partial evaluation of `(Q3)`.

$$\dfrac{\dfrac{\cdots}{\Gamma \vdash \texttt{(Q3)} \rightarrow \langle a \rangle \,\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ (\$v/c, \$v/d)\langle/a\rangle^{d_1}}\ (\textsc{Pelm}) \quad (\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ \$v/d)^{d_1.1\#2} = \mathsf{child\_fusion}\ \langle a \rangle \,\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ (\$v/c, \$v/d)\langle/a\rangle^{d_1}\ d}{\Gamma \vdash \texttt{(Q3)}/d \rightarrow (\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ \$v/d)^{d_1.1\#2}}\ (\textsc{PCStp})$$

Figure 14: Partial evaluation of $\texttt{(Q3)}/d$

$$\dfrac{\dfrac{\cdots}{\Gamma \vdash \texttt{(Q3)} \rightarrow \langle a \rangle \,\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ (\$v/c, \$v/d)\langle/a\rangle^{d_1}}\ (\textsc{Pelm}) \quad (\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ \$v/d)^{d_1.1\#1} = \mathsf{child\_fusion}\ \langle a \rangle \,\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ (\$v/c, \$v/d)\langle/a\rangle^{d_1}\ c}{\Gamma \vdash \texttt{(Q3)}/c \rightarrow (\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ \$v/c)^{d_1.1\#1}}\ (\textsc{PCStp})$$

Figure 15: Partial evaluation of $\texttt{(Q3)}/c$

$\oplus_D$, respectively.

DEFINITION 4.1 (distinctly ordered sequences). *For a given sequence* $S = \langle y_1, y_2, \cdots, y_n \rangle$*,* S *is distinctly ordered under* $\precsim$ *when the following conditions hold*.

- *All elements of* S *are in a total order under* $\precsim$*, i.e.,* $\forall y, z, w \in S$*,*

$$y \precsim z \wedge z \precsim y \Rightarrow y \sim z \tag{1}$$
$$y \precsim z \wedge z \precsim w \Rightarrow y \precsim w \tag{2}$$
$$y \precsim z \vee z \precsim y \tag{3}$$

  and

- *That* S *is strictly monotonic which is defined as the followings,*

  1. $[]$ *is a strictly monotonic, and*

$$\dfrac{\dfrac{\cdots}{\Gamma \vdash \texttt{(Q3)}/d \rightarrow e_1^{d_1.1\#2}}\ (\textsc{PCStp}) \quad \dfrac{\cdots}{\Gamma \vdash \texttt{(Q3)}/c \rightarrow e_2^{d_1.1\#1}}\ (\textsc{PCStp})}{\Gamma \vdash (\texttt{(Q3)}/d, \texttt{(Q3)}/c) \rightarrow (e_1^{d_1.1\#2}, e_2^{d_1.1\#1})}\ (\textsc{PSeq})$$
$$(\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ (\$v/c, \$v/d))^{d_1.1\#[1,2]} = \mathsf{self\_fusion}\ (e_1^{d_1.1\#2}, e_2^{d_1.1\#1})^{[d_1.1\#2, d_1.1\#1]}\ *$$
$$\overline{\Gamma \vdash (\texttt{(Q3)}/d, \texttt{(Q3)}/c)/\,\texttt{self} :: * \rightarrow (\mathsf{for}\ \$v\ \mathsf{in}\ e_b\ \mathsf{return}\ (\$v/c, \$v/d))^{d_1.1\#[1,2]}}\ (\textsc{PSStp})$$

where we use e_1 for for $\$v$ in e_b return $\$v/d$ and e_2 for for $\$v$ in e_b return $\$v/c$

Figure 16: Partial evaluation of $((\texttt{(Q3)}/d), (\texttt{(Q3)}/c))/\,\texttt{self} :: *$.

2. *for a strictly monotonic sequence ys, y:ys is also strictly monotonic iff.*

$$\forall y' \in ys(y \prec y').$$

PROPERTY 4.2. *For a given distinctly ordered sequence y:ys, the following properties hold by* DEFINITION *4.1.*

(i) x:ys *is a distinctly ordered where* $x \sim y$.

(ii) x:y:ys *is a distinctly ordered where* $x \prec y$.

DEFINITION 4.3 (Preservation of order). *Binary operator $\oplus$ defined over a total order set under $\precsim$ preserves the order if for any elements y_1, y_2 in the total order set,*

$$\frac{y_1 \sim y_2}{(y_1 \oplus y_2) \sim y_1} \qquad \textit{(PRESO)}$$

holds.

Ordered insertion (one to many)($\sim\triangleleft^{\oplus}_{\prec}$)

DEFINITION 4.4 (Ordered insertion $\sim\triangleleft^{\oplus}_{\prec}$). *Binary operator $\sim\triangleleft^{\oplus}_{\prec}$ returns, for a list on the left operand, a new list in which y on the right operand is inserted by the following inference rules.*

$$\frac{|y| \rightarrow y'}{([] \sim\triangleleft^{\oplus}_{\prec} y) \rightarrow [y']}$$

$$\frac{z \sim y \qquad (z \oplus y) \rightarrow v}{((z{:}zs) \sim\triangleleft^{\oplus}_{\prec} y) \rightarrow v{:}zs}$$

$$\frac{y \prec z}{(z{:}zs) \sim\triangleleft^{\oplus}_{\prec} y) \rightarrow (y{:}z{:}zs)}$$

$$\frac{z \prec y \qquad (zs \sim\triangleleft^{\oplus}_{\prec} y) \rightarrow zs'}{((z{:}zs) \sim\triangleleft^{\oplus}_{\prec} y) \rightarrow z{:}zs'}$$

THEOREM 4.5 (Ordered insertion). *For any distinctly ordered sequence S under $\precsim$, $S\sim\triangleleft^{\oplus}_{\prec}$ y is also distinctly ordered under $\precsim$ where $\oplus$ satisfies (*PRESO*).*

Proof. Induction on the sequence S is used for the following cases;

(i) S is []: $[]\sim\triangleleft^{\oplus}_{\prec} y$, which is $[y']$ by the first definition of $\sim\triangleleft^{\oplus}_{\prec}$, is also distinctly ordered by DEFINITION 4.1

(ii) S is $z{:}zs$: It is sufficient to examine the following each case in binary relation between z and y, because both z and y are in a total order set under $\precsim$.

(a) $z \sim y$: $S {}_\sim\!\lhd^{\oplus}_{\prec} y$, which is $(z \oplus y){:}zs$ by the second definition of ${}_\sim\!\lhd^{\oplus}_{\prec}$, is also distinctly ordered by PROPERTY 4.2 (i) and (PRESO).

(b) $y \prec z$: $S {}_\sim\!\lhd^{\oplus}_{\prec} y$, which is $y{:}z{:}zs$ by the third definition of ${}_\sim\!\lhd^{\oplus}_{\prec}$, is also distinctly ordered by PROPERTY 4.2 (ii).

(c) $z \nsim y \wedge y \not\prec z$: This implies $z \prec y$ by totality(3). So, the fourth definition is applied. The fourth definition is rewritten as follows;

$$\frac{z_1 \prec y \qquad ((z_2{:}zs) \; {}_\sim\!\lhd^{\oplus}_{\prec} y) \rightarrow z'{:}zs'}{((z_1{:}z_2{:}zs) \; {}_\sim\!\lhd^{\oplus}_{\prec} y) \rightarrow (z_1{:}z'{:}zs)}$$

In the above inference rule, $(z'{:}zs)$ is distinctly ordered by the inductive hypothesis. So to see that $(z_1{:}z'{:}zs)$ is distinctly ordered, we have to show $z_1 \prec z'$ because of PROPERTY 4.2 (ii). Now, the following cases in relation between y and z_2 are examined.

i. $y \precsim z_2$: By the second and the third definition of ${}_\sim\!\lhd^{\oplus}_{\prec}$, $z' \sim y$ holds. So, $(z_1 \prec y \wedge z' \sim y)$ implies $z_1 \prec z'$ because of z_1, y, z' in a total order set under $\precsim$.

ii. $z_2 \prec y$: By the fourth definition of ${}_\sim\!\lhd^{\oplus}_{\prec}$, $z' \sim z_2$ holds. So, $(z' \sim z_2 \wedge z_1 \prec z_2)$ implies $z_1 \prec z'$.

□

Ordered union (many to many)(${}_\sim\!\uplus^{\oplus}_{\prec}$)

DEFINITION 4.6 (Ordered union (many to many)). *For sequences in which all elements are in a total order under $\precsim$ where $\oplus$ satisfies (*PRESO*), binary operator ${}_\sim\!\uplus^{\oplus}_{\prec}$ is defined as the following inference rules.*

$$\frac{}{(zs \; {}_\sim\!\uplus^{\oplus}_{\prec} []) \rightarrow zs}$$

$$\frac{(zs \; {}_\sim\!\lhd^{\oplus}_{\prec} y) \rightarrow zs' \qquad zs' \; {}_\sim\!\uplus^{\oplus}_{\prec} ys \rightarrow vs}{zs \; {}_\sim\!\uplus^{\oplus}_{\prec} (y{:}ys) \rightarrow vs}$$

THEOREM 4.7 (Ordered union). *For any distinctly ordered sequence S_1 under $\precsim$, $(S_1 \; {}_\sim\!\uplus^{\oplus}_{\prec} S_2)$ is also distinctly ordered under $\precsim$ where $\oplus$ satisfies (*PRESO*).*

Proof. Induction on the sequence S_2 is used for the following cases;

(i) S_2 is []: $(S_1 \; {}_\sim\!\uplus^{\oplus}_{\prec} [])$, which is S_1 by the first definition of ${}_\sim\!\uplus^{\oplus}_{\prec}$, is also distinctly ordered.

(ii) S_2 is $y{:}ys$: For any distinctly ordered sequence S_1 under $\precsim$, $(S_1 \sim\uplus^{\oplus}_{\prec} ys)$ is also distinctly ordered sequence under $\precsim$ by the inductive hypothesis. By THEOREM 4.5, $(S_1 \precsim\!\!\lhd^{\oplus}_{\prec} y)$ is a distinctly ordered sequence. So, $(S_1 \sim\uplus^{\oplus}_{\prec} y{:}ys)$ is a distinctly ordered sequence by the second definition of $\sim\uplus^{\oplus}_{\prec}$.

□

Now, we can define the three important operators used in ordered union on $\mathcal{D}$, strict partial order $\prec_{\mathcal{D}}$, unifiable $\sim_{\mathcal{D}}$ and unification $\oplus_{\mathcal{D}}$.

DEFINITION 4.8 (Strict Partial Order on $\mathcal{D}$ $(\prec)$). *The strict partial order on $\mathcal{D}$ is defined as follows;*

$$\frac{r_1 = r_2 \qquad x_1 \prec_x x_2}{r_1\ x_1 \prec_D r_2\ x_2}$$

$$\frac{(x \neq \epsilon) \vee (x \neq .?\ x')}{\epsilon \prec_X x}$$

$$\frac{b_1 \prec_B b_2}{.b_1 \prec_X .b_2}$$

$$\frac{(n_1 < n_2) \vee (n_1 = n_2 \wedge x_1 \prec_X x_2)}{n_1 x_1 \prec_B n_2 x_2}$$

DEFINITION 4.9 (Unifiable on $\mathcal{D}(\sim)$). *The binary operator $\sim$ on $\mathcal{D}$ is defined as follows;*

$$\frac{r_1 = r_2 \qquad x_1 \sim_X x_2}{r_1\ x_1 \sim_D r_2\ x_2}$$

$$\frac{n_1 = n_2 \qquad x_1 \sim_X x_2}{n_1 x_1 \sim_B n_2 x_2}$$

$$\frac{}{\epsilon \sim_X \epsilon}$$

$$\frac{b_1 \sim_B b_2}{.b_1 \sim_X .b_2}$$

$$\frac{\mathsf{orderable}_{bs}\ \ bs_1 +\!\!+ bs_2}{\#bs_1 \sim_X \#bs_2}$$

DEFINITION 4.10 (Unification on $\mathcal{D}(\oplus)$). *The binary operator $\oplus$ on $\mathcal{D}$ is defined as follows;*

$$\frac{r_1 x_1 \sim_D r_2 x_2 \qquad (x_1 \oplus_X x_2) \to x}{(r_1 x_1 \oplus_D r_2 x_2) \to r_1\ x}$$

$$\frac{n_1 x_1 \sim_B n_2 x_2 \qquad (x_1 \oplus_X x_2) \to x}{(n_1 x_1 \oplus_B n_2 x_2) \to n_1 x}$$

$$\frac{}{(\epsilon \oplus_X \epsilon) \to \epsilon}$$

$$\frac{.b_1 \sim_X .b_2 \qquad (b_1 \oplus_B b_2) \to b}{(.b_1 \oplus_X .b_2) \to .b}$$

$$\frac{\#bs_1 \sim_X \#bs_2 \qquad [] \sim_B \uplus_{\prec_B}^{\oplus_B} (bs_1 +\!\!+ bs_2) \to bs}{(\#bs_1 \oplus_X \#bs_2) \to \#bs}$$

DEFINITION 4.11 (Orderable on a sequence of the extended Dewey code). *For a sequence of the extended Dewey code ds, unary predicate* orderable *is defined as the following inference rule*.

$$\frac{\mathsf{all}_{\mathsf{ord}}\ ds}{\mathsf{orderable}\ ds}$$

where

$$\frac{}{\mathsf{all}_{\mathsf{ord}}\ []}$$

$$\frac{}{\mathsf{all}_{\mathsf{ord}}\ d{:}[]}$$

$$\frac{\forall d' \in ds, \mathsf{ord}\ d\ d'}{\mathsf{all}_{\mathsf{ord}}\ d{:}ds}$$

$$\frac{((d_1 \prec d_2) \vee (d_2 \prec d_1) \vee (d_1 \sim d_2))}{\mathsf{ord}\ d_1\ d_2}$$

5 The Revised Algorithm

In this section, we will show the revised algorithm with handling failure cases and with more on fusing "for" expressions using auxiliary transformations. As described in Section 3, our fusion does not always succeed. When the dc_sort, which is used in the three kinds of fusion function (PCSTP), (PSSTP) and (PPSTP) in Figure 10, does not succeed, our fusion does not succeed. We show our fusion algorithm handing this failure case in Section 5.1.

5.1 Handling failure cases

To propagate success or failure of our fusion, we extend the return values of the three kinds of fusion functions by adding boolean values which indicate the status of dc_sort. If dc_sort succeeds every fusion function returns with `true`, otherwise with `false`.

$$\begin{aligned}&\mathsf{child\_fusion} :: XQuery^D \to QName \to (Bool, XQuery^D)\\ &\mathsf{self\_fusion} :: XQuery^D \to QName \to (Bool, XQuery^D)\\ &\mathsf{parent\_fusion} :: XQuery^D \to QName \to (Bool, XQuery^D)\end{aligned}$$

Figure 17 shows the revised version of our fusion transformation on XQuery expressions handling the failure cases. Now, the form of the judgement is changed into

$$\Gamma \vdash e \to (\texttt{true} \,/\, \texttt{false}, e^d)$$

which indicates that for a given environment Γ (mapping XQuery variables bound by "let" or "for" to annotated expressions), the XQuery expression e compiles into the annotated expressions e^d with Boolean value which indicates that the fusion succeeds (`true`) or not (`false`). This Boolean values play an important role in fusing "let" and "for" expressions. For both a constant expression and a variable, our fusion functions return with `true`. For a sequence expression, when one or more subexpressions failed, there are two candidates; (1) recovering all the element expressions, including ones fused successfully, back to input expressions; or (2) doing nothing, namely there are fused expressions and nonfused expressions mixed together because when the dc_sort fails the input expressions are returned with annotation ϵ. We chose (2) because for a sequence expressions as a top-level expression, (2) is more efficient than (1). This choice makes the treatment of "let" and "for" expressions in our fusion be intricate. For a "let" expression, when the fusion of e_1 failed, fusing e_2 should be performed under an updated environment with not the result of fusing e_1 but the input e_1 with annotating ϵ because of our choice of (2) in fusing sequence expressions. We use a trinary operator,

$$b \,?\, e_1 \;:\; e_2$$

for a Boolean value b, two expressions e_1 and e_2. This operator results e_1 when b is `true` and results e_2 when b is `false`. For a "for" expression, we do similarly as for a "let" expression. For a location step expression $e/\alpha{::}en$, we use the three extended functions, child_fusion, self_fusion and parent_fusion, with returning Boolean values. In these functions, if dc_sort succeeds then the returned Boolean value is `true`, otherwise `false`. For an element construction, we do similarly as presented in Section 3 except the treatment of Boolean values.

$$\frac{}{\Gamma \vdash \mathsf{c} \rightarrow (\mathtt{true}, \mathsf{c}^{\epsilon})} \quad \text{(PCST')}$$

$$\frac{\Gamma(v).btype = \text{"let"}}{\Gamma \vdash \$v \rightarrow (\mathtt{true}, \Gamma(v).expr)} \quad \text{(PLVAR')}$$

$$\frac{\Gamma(v).btype = \text{"for"}}{\Gamma \vdash \$v \rightarrow (\mathtt{true}, \$v^{1})} \quad \text{(PFVAR')}$$

$$\frac{\Gamma \vdash e_1 \rightarrow (b_1, e_1'^{d_1}) \quad \ldots \quad \Gamma \vdash e_N \rightarrow (b_N, e_n'^{d_N}) \quad e''^{d'} = \mathsf{flatten}\ ((e_1'^{d_1}, \ldots, e_n'^{d_N})^{[d_1,..,d_N]})}{\Gamma \vdash (e_1, \ldots, e_N) \rightarrow (b_1 \wedge \ldots \wedge b_N, e''^{d'})} \quad \text{(PSEQ')}$$

$$\frac{\Gamma \vdash e_1 \rightarrow (b_1, e_1'^{d_1}) \quad \Gamma \cup \{v \mapsto (b_1\ ?\ e_1'^{d_1}\ :\ e_1^{\epsilon})\} \vdash e_2 \rightarrow (b_2, e_2'^{d_2})}{\Gamma \vdash \mathsf{let}\ \$v := e_1\ \mathsf{return}\ e_2 \rightarrow (\mathtt{true}, (b_2\ ?\ e_2'^{d_2}\ :\ \mathsf{let}\ \$v := e_1\ \mathsf{return}\ e_2)} \quad \text{(PLET')}$$

$$\frac{\Gamma \vdash e_1 \rightarrow (b_1, e_1'^{d_1}) \quad \Gamma \cup \{v \mapsto (b_1\ ?\ e_1'^{d_1}\ :\ e_1^{\epsilon})\} \vdash e_2 \rightarrow (b_2, e_2'^{d_2})}{\Gamma \vdash \mathsf{for}\ \$\mathsf{v}\ \mathsf{in}\ \mathsf{e}_1\ \mathsf{return}\ \mathsf{e}_2 \rightarrow (\mathtt{true}, (\mathsf{for}\ \$\mathsf{v}\ \mathsf{in}\ (\mathsf{b}_1\ ?\ \mathsf{e}_1'^{\mathsf{d}_1}\ :\ \mathsf{e}_1^{\epsilon})\ \mathsf{return}\ (\mathsf{b}_2\ ?\ \mathsf{e}_2'^{\mathsf{d}_2}\ :\ \mathsf{e}_2^{\epsilon}))^{(\mathsf{b}_2\ ?\ \#\mathsf{d}_2\ :\ \epsilon)})} \quad \text{(PFOR')}$$

$$\frac{\Gamma \vdash e \rightarrow (b_1, e'^{d}) \quad (b_2, e''^{d'}) = \mathsf{child\_fusion}\, e'^{d}\ en}{\Gamma \vdash e/\,\mathtt{child} :: en \rightarrow (b_2, e''^{d'})} \quad \text{(PCSTP')}$$

$$\frac{\Gamma \vdash e \rightarrow (b_1, e'^{d}) \quad (b_2, e''^{d'}) = \mathsf{self\_fusion}\, e'^{d}\ en}{\Gamma \vdash e/\,\mathtt{self} :: en \rightarrow (b_2, e''^{d'})} \quad \text{(PSSTP')}$$

$$\frac{\Gamma \vdash e \rightarrow (b_1, e'^{d}) \quad (b_2, e''^{d'}) = \mathsf{parent\_fusion}\, e'^{d}\ en}{\Gamma \vdash e/\,\mathtt{parent} :: en \rightarrow (b_2, e''^{d'})} \quad \text{(PPSTP')}$$

$$\frac{\Gamma \vdash e \rightarrow (b_1, e'^{d_1}) \quad d_2 = new\_rootD \quad e'^{d_3} = \mathsf{dc\_assign}\ e'\ d_2.1}{\Gamma \vdash \langle en \rangle e \langle /en \rangle \rightarrow (\mathtt{true}, \langle en \rangle e'^{d_3} \langle /en \rangle^{d_2})} \quad \text{(PELM')}$$

Figure 17: XQuery fusion handling failure

6 Experimental Results

6.1 System overview

We have implemented a prototype system in Objective Caml. It consists of about 4600 lines of code. Although the framework has been represented using simple function definitions, actual implementation uses more complex structure to achieve static emulation of the store more precisely. Main enhancements in the actual implementation are:

- achieving both sorting and duplicate elimination in the extended Dewey order simultaneously using one higher-order function exploiting the algebraic structure shown in Section 4.
- keeping track of the success or failure of the partial evaluation in order to recover original expression when subexpression fails to simplify.
- maintaining the global environment for storing all annotated expressions during our fusion transformation as 4-ary relation of (e, o, c, d) where,
  - e denotes an XQuery expression,

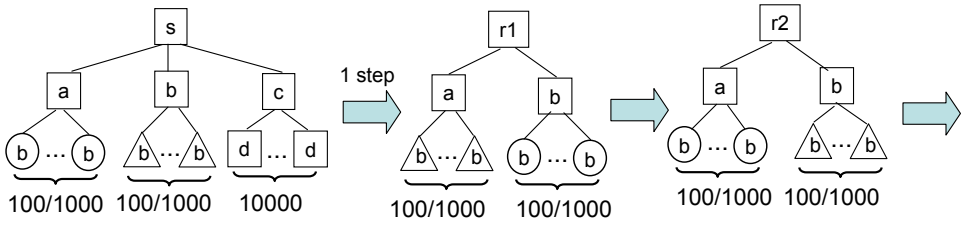

Figure 18: Schema mappings in Qa

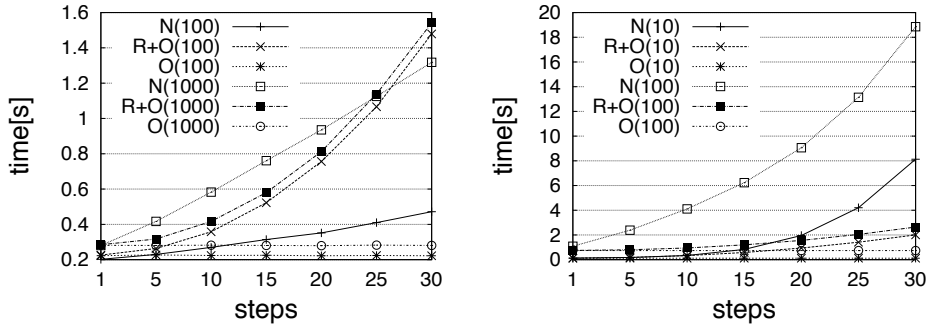

Figure 19: Times for **Qa**(left) and **Qb**(right).

- since annotations for the extended Deweys' are associated with nodes of abstract syntax trees, the node-id o needs to be maintained,
- c denotes a context of an expression e. For example, when e occurs in a `return` expression of a `for` expression, we need to keep this context as Deweys' prefix.
- d denotes a Dewey code of e.

Our fusion algorithm adds information for annotated XQuery to this relation. The function $getExpGlobal(r)$ is implemented by using this relation.

Currently it works stand-alone reading XQuery expression from standard input and produces rewritten XQuery to standard output.

6.2 Settings and Results

While actual evaluation numbers are predictable, for example from Manolescu et al. [2006], we have tested two kinds of queries **Qa**(n) and **Qb**(n) using `Galax`[5] version 1.0 on 2.6GHz Intel Core2 Duo with 4GB RAM, running MacOS 10.5.6. Both queries are synthetic for XML data integration systems with n steps as schema mappings inspired by Tatarinov and Halevy [2004].

Qa(n), which is for a document "d1.xml" is shown in Figure 20. In the "d1.xml", the root node `s` has three child nodes `a`, `b` and `c` shown as the left-most

[5]http://www.galaxquery.org/, default optimization option turned on.

```
let  $r1 := <r1>{let $s := doc("d1.xml")/s
                 return (<a>{$s/b/b}</a>,
                         <b>{$s/a/b}</b>)}</r1>
return
let  $r2 := <r2>{(<a>{$r1/b/b}</a>,
                  <b>{$r1/a/b}</b>)}</r2>
return
...
let  $rn := <rn>{(<a>{$r(n-1)/b/b}</a>,
                  <b>{$r(n-1)/a/b}</b>)}</rn>
return
let $v := ($rn/b,$rn/a)
return $v/b
```

Figure 20: Test queries Qa(n)

tree in Figure 18. We prepared two documents, in which the number of b elements at level 3 under the a and b elements at level 2 (where the root is at level 1) is 100/1000. In **Qa**, each step of schema mapping swaps b elements at level 3 under a element with ones under b element. This mapping is shown in Figure 18. The left graph in Figure 19 shows the execution times for naive queries(N(100) and N(1000), optimized queries(O(100) and O(1000)) and query rewriting costs plus optimized queries(R+O(100) and R+O(1000)) for two documents, respectively. Note that since naive queries produce redundant intermediate results in proportional to the number of steps, the execution times are linear with respect to steps. Whereas, since optimized queries rewritten by our prototype system always degenerate to queries to the extensional DB only, the execution time remain constant. This figure shows that the rewriting costs are not neglectable when the steps are increased. Most of this overhead comes from the global environment that is kept in memory. Since the global environment is only used in solving reverse axis, it can be safely discarded when input queries include forward axis only. This optimization will be incorporated in the next version of our prototype system. For an even number of steps, our prototype system rewrites **Qa(**n**)** into the optimized query, (doc("d1.xml")/s/a/b,doc("d1.xml")/s/b/b,()).

Qb(n**)**, which is for a document "d2.xml" is shown in Figure 21

For "for" expressions, we prepared the two documents "d2.xml" shown in the left tree in Figure 22. In this document, the root node s has 10/100 t elements, and each t element has two elements a and b at level 3. Under both of the a and b elements, there are 10/1000 b elements at level 4. In **Qb**, each step of schema mapping swaps b elements at level 4 under the a elements with ones under the b elements. This mapping is shown in Figure 22. The right graph in Figure 19 shows the execution times for naive queries, optimized queries and rewriting plus optimized queries, where the symbols N, O and R+O have the same meanings as

```
let  $r1 := <r1>{for $t1 in doc("d2.xml")/s/t
                 return <t>{(<a>{$t1/b/b}</a>,
                             <b>{$t1/a/b}</b>)}</t>}
            </r1>
return
let  $r2 := <r2>{for $t2 in $r1/t
                 return <t>{(<a>{$t2/b/b}</a>,
                             <b>{$t2/a/b}</b>)}</t>}
            </r2>
return
...
let $rn := <rn>{for $tn in $r(n-1)/t
                 return <t>{(<a>{$tn/b/b}</a>,
                             <b>{$tn/a/b}</b>)}</t>}
            </rn>
return
let $v := ($rn/t/b,$rn/t/a)
return $v/b
```

Figure 21: Test queries Qb(n)

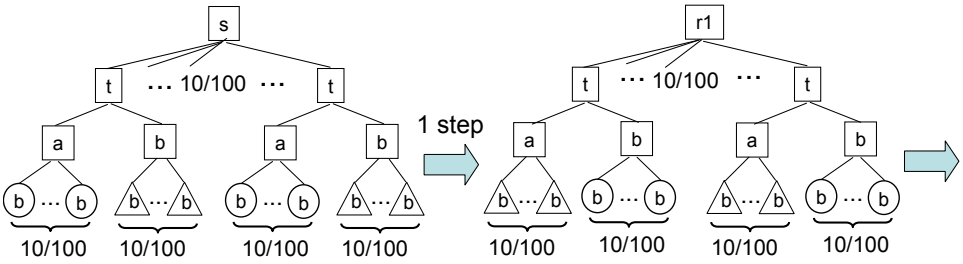

Figure 22: Schema mappings in Qb

in **Qa**. Compared to **Qa**, R+O always outperform N showing that eliminated redundant evaluation costs of intermediate results that are (repeatedly) created by the "for" expressions always exceeded the rewriting overhead. For an odd number of steps, our prototype system rewrites **Qb**(n) into the following optimized query;
for $t1 in doc("d2.xml")/s/t return ($t1/b/b,($t1/a/b,()))

7 Conclusion

In this paper, we proposed a new rewriting technique for XQuery fusion to eliminate unnecessary element construction in the expressions while guaranteeing preservation of document order. The prominent feature of our framework is in

its static emulation of XML store and assignment of extended Deweys' to the expressions, which leads to easy construction of correct fusion transformation.

References

W.N. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/377674.377676.

Bilel Gueni, Talel Abdessalem, Bogdan Cautis, and Emmanuel Waller. Pruning Nested XQuery Queries. In *CIKM*, pages 541–550, 2008.

Jan Hidders, Jan Paredaens, Roel Vercammen, and Serge Demeyer. A Light but Formal Introduction to XQuery. In *Second International XML Database Symposium,(XSym2004)*, pages 5–20, 2004.

Christoph Koch. On the role of composition in XQuery. In *Proceedings of Eighth International Workshop on the Web and Databases (WebDB 2005)*, 2005.

Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan, and Ting Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig pattern Matching. In *Proc of VLDB*, 2005.

Ioana Manolescu, Cedric Miachon, and Philippe Michiels. Towards micro-benchmarking xquery. In *Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems (EXPDB 2006)*, 2006.

Jim Melton. Writing Wrongs: How Not To Build A Standard. In *XIME-P (Keynote)*, 2008.

Wim Le Page, Jan Hidders, Philippe Michiels, Jan Paredaens, and Roel Vercammen. On the expressive power of node construction in XQuery. In *Proceedings of Eighth International Workshop on the Web and Databases (WebDB 2005)*, 2005.

Igor Tatarinov and Alon Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *Proceedings of the ACM International Conference on Management of Data*, pages 539–550, 2004.

Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc of SIGMOD*, 2002.

P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP (LNCS 300)*, pages 344–358, 1988.

World Wide Web Consortium. XQery1.0 and XPath2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics, January 2007. W3C Recommendation.