ISSN 1884-0760

GRACE TECHNICAL REPORTS

Context-Preserving XQuery Fusion

Hiroyuki Kato Soichiro Hidaka Zhenjiang Hu
Keisuke Nakano Yasunori Ishihara

GRACE-TR 2010–07 September 2010

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.

Context-Preserving XQuery Fusion

H. Kato[1], S. Hidaka[1], Z. Hu[1], K. Nakano[2], and Y. Ishihara[3]

[1] National Institute of Informatics, Japan,
{kato, hidaka, hu}@nii.ac.jp
[2] The University of Electro-Communications, Japan,
ksk@cs.uec.ac.jp
[3] Osaka University, Japan,
ishihara@ist.osaka-u.ac.jp

Abstract. XQuery is a DBPL for querying XML databases. The semantics of XQuery is context sensitive and requires preservation of document order. In this paper, we propose, as far as we are aware, the first XQuery fusion that can deal with both the document order and the context of XQuery expressions. More specifically, we carefully design a context representation of XQuery expressions based on the Dewey order encoding, develop a context-preserving XQuery fusion for ordered trees by static emulation of the XML store, and prove that our fusion is correct. Our XQuery fusion has been implemented, and all the examples in this paper have passed the system.

1 Introduction

Fusion [21, 2, 5] is a well-known technique for improving efficiency by removing unnecessary intermediate data from the computation. Although it has been applied to optimize query languages such as SQL [3] and object query languages [5], it remains a challenge to implement fusion for XQuery optimization. This is because XQuery has more complicated semantics [12]; *it is context sensitive and requires preservation of document order*. One may consider, for example, the following naive fusion transformation[4] (as studied in [4]).

$$\langle e \rangle E_1, \ldots, E_n \langle /e \rangle / c \mapsto \sigma_c(E_1), \ldots, \sigma_c(E_n) \;^{5} \quad \text{(F)}$$

This transformation works correctly only if the order of the XML document and the context can be ignored. However, order is an important issue in XML documents [6, 1], and various index structure for ordered trees have been developed for XML documents [20, 14, 24]. When we view an XML document as an ordered tree, an existing fusion transformation like (F) by naive elimination of element constructors does not work correctly because the context, which is a navigation of newly constructed trees, is missing during the transformation.

---

[4] Analogous to relational algebra operators, σ_c is used as a selection, which extracts data with their element name being c.

[5] We use "narrow" angle brackets for XML tags. For example, we use $\langle e \rangle$ instead of <e>.

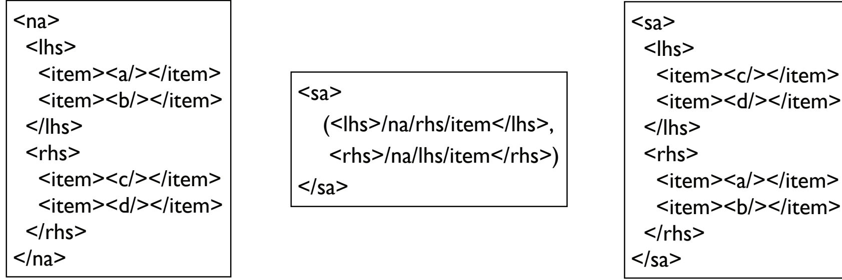

Fig. 1. Source XML: S (left). XQuery expression: e_m (middle) and the serialized result: T (right).

Consider the simple case illustrated in Figure 1, where the query e_m (the middle) is applied to the source S (the left), and the target T (the right) is obtained as the serialized result. Let us apply the following query e_1 to the serialized T,

$$e_1 : \textbf{let } \$v := (/\mathsf{sa}/\mathsf{rhs}, /\mathsf{sa}/\mathsf{lhs}) \textbf{ return } \$v/\mathsf{item}.$$

Since the semantics of "axis access" by using "/" in XQuery (and XPath) requires sorting without duplicates in the document order, the correct result is the following sequence of "item" elements:

$$\langle\mathsf{item}\rangle\langle\mathsf{c}/\rangle\langle/\mathsf{item}\rangle, \langle\mathsf{item}\rangle\langle\mathsf{d}/\rangle\langle/\mathsf{item}\rangle, \langle\mathsf{item}\rangle\langle\mathsf{a}/\rangle\langle/\mathsf{item}\rangle, \langle\mathsf{item}\rangle\langle\mathsf{b}/\rangle\langle/\mathsf{item}\rangle.$$

However, for the composite query $e_1 \circ e_m$, by unfolding the expression e_m, we can get

$$\begin{array}{l}\textbf{let } \$t := \langle\mathsf{sa}\rangle\{(\langle\mathsf{lhs}\rangle/\mathsf{na}/\mathsf{rhs}/\mathsf{item}\langle/\mathsf{lhs}\rangle, \langle\mathsf{rhs}\rangle/\mathsf{na}/\mathsf{lhs}/\mathsf{item}\langle/\mathsf{rhs}\rangle)\}\langle/\mathsf{sa}\rangle \\ \textbf{return let } \$v := (\$t/\mathsf{rhs}, \$t/\mathsf{lhs}) \textbf{ return } \$v/\mathsf{item}.\end{array}$$

Now if we perform the calculation according to the context-insensitive fusion rule (F):

$$\begin{array}{ll} & e_1 \circ e_m \\ \rightarrow & \quad \{(\text{variable expansion for } \$t); (\text{F})\} \\ & \textbf{let } \$v := (\langle\mathsf{rhs}\rangle/\mathsf{na}/\mathsf{lhs}/\mathsf{item}\langle/\mathsf{rhs}\rangle, \langle\mathsf{lhs}\rangle/\mathsf{na}/\mathsf{rhs}/\mathsf{item}\langle/\mathsf{lhs}\rangle) \\ & \textbf{return } \$v/\mathsf{item} \\ \rightarrow & \quad \{(\text{variable expansion for } \$v); (\text{F})\} \\ & (/\mathsf{na}/\mathsf{lhs}/\mathsf{item}, /\mathsf{na}/\mathsf{rhs}/\mathsf{item}) \end{array}$$

then evaluating the transformed query $(/\mathsf{na}/\mathsf{lhs}/\mathsf{item}, /\mathsf{na}/\mathsf{rhs}/\mathsf{item})$ on S gives

$$\langle\mathsf{item}\rangle\langle\mathsf{a}/\rangle\langle/\mathsf{item}\rangle, \langle\mathsf{item}\rangle\langle\mathsf{b}/\rangle\langle/\mathsf{item}\rangle, \langle\mathsf{item}\rangle\langle\mathsf{c}/\rangle\langle/\mathsf{item}\rangle, \langle\mathsf{item}\rangle\langle\mathsf{d}/\rangle\langle/\mathsf{item}\rangle$$

whose order of "item" elements is different from the expected result. Furthermore, if we consider the query e_2 on T:

$$e_2 : \textbf{let } \$v := /\mathsf{sa}/\mathsf{rhs}/\mathsf{item} \textbf{ return } \$v/..$$

then although the expected result of e_2 to T is the "rhs" element, the result of the transformed query from $e_2 \circ e_m$ via similar steps above is the "lhs" element. In both examples, the problem is caused by not having the context, which is a tree navigation over the newly constructed XML fragment using $\langle \mathsf{sa} \rangle ... \langle / \mathsf{sa} \rangle$ in e_m.

The problem of the existing fusion transformation lies in that the naive elimination of element constructors during the transformation does not preserve the (computation) context because element constructors construct ordered trees. This implies that eliminating element constructors in XQuery expressions and preserving the context of the expressions are conflicting requirements. The purpose of our work is to propose a new fusion mechanism to meet these two requirements. To this end, we should find a way to manage the context of the original expressions in developing a correct fusion transformation.

While we will show the concrete solution to both examples at the end of this paper, we shall give an intuitive idea of our solution to the first example here. For two step expressions /na/rhs/item and /na/lhs/item in e_m which constructs the ordered tree T, there is a fact that the items of the sequence generated by /na/rhs/item always precede ones generated by /na/lhs/item in the ordered tree T for an arbitrary XML store. By adding this information to these two step expressions, for given $e_1 \circ e_m$, we can formulate the correct XQuery expression (/na/rhs/item, /na/lhs/item) from this information. We call this information, context.

We propose a novel context-preserving XQuery fusion for when an XML document is modeled as an ordered tree. Our basic idea is *to lift dynamic operations on XML store to the static level of expression*. Our main contributions can be summarized as follows.

- To keep track of context, we carefully design the context representation of XQuery expressions to reflect the properties of element constructions. This enable us to statically emulate newly created XML fragments — created by element constructors — in the XML store.
- We develop a context-preserving fusion for XQuery by partial evaluation and prove the correctness of our fusion. Our fusion introduces an annotated XQuery, which is an XQuery expression with the context as an annotation, to preserve the context of the input expressions even when the element constructors are eliminated during our fusion transformation.

The paper proceeds as follows. Section 2 reviews the XQuery semantics and introduces value equivalent expressions to show our fusion concisely. In Section 3, we carefully design the context of XQuery expressions by extending Dewey code and its order to suite the semantics of XQuery expressions. Section 4 presents the algorithm of context-preserving fusion using the extended Dewey code and its order. We discuss related work in Section 5 and conclude the paper in Section 6.

2 XQuery Semantics

To show our XQuery fusion concisely and that it is semantics-preserving, we briefly review the semantics of the core part of XQuery that is based on [12]. Our target XQuery expressions, a subset of XQuery, are as follows.

$$e ::= \$\mathtt{v} \mid (e, e, ..., e) \mid () \mid e/\alpha{::}\tau \mid \textbf{for } \$\mathtt{v} \textbf{ in } e \textbf{ return } e$$
$$\mid \quad \textbf{let } \$\mathtt{v} := e \textbf{ return } e \mid \langle t \rangle e \langle /t \rangle$$

A query expression can be a variable $\$v$, a sequence expression $(e_1, \ldots, e_n)$ where each subexpression e_i is not a sequence expression, an empty sequence (), a location step expression $e/\alpha{::}\tau$ where α is an axis which can be child, self, parent (..), and τ is a name test which can be a tag name t or $*$ (an arbitrary tag), a "for"-expression, a "let"-expression, or an element construction expression $\langle t \rangle e \langle /t \rangle$. Since we focus on newly constructed trees that consist of XML nodes, to simplify the presentation, a constant c is represented by "empty-element tags" like $\langle \mathsf{c} / \rangle$. Although constants themselves are not nodes, they become a (text) node when they occur in an element constructor. For example, a constant "abc" is not a node i.e., this constant does not populate any ordered trees. On the other hand, consider $\langle \mathsf{a} \rangle$"abc"$\langle /\mathsf{a} \rangle$; in this expression, the constant "abc" is a text node because the constant occurs in the element construction of $\langle \mathsf{a} \rangle(...)\langle /\mathsf{a} \rangle$, i.e., this constant is a child node of the element node of a. We could define the semantics of constants with such behavior, but this would make our presentation unnecessarily complex.

2.1 Sequence: Data Model in XQuery

The data model of XQuery is *sequences* [22]. A sequence is an ordered collection of zero or more items. One important characteristic of the data model is that sequences are *flat* in the sense that a sequence never contains other sequences; if sequences are combined, the result is always a flattened sequence. In addition, there is no distinction between an item and a singleton sequence containing that item, i.e., we often write $[a]$ as a or vice versa.

We denote the empty sequence as [], non-empty sequences for example as [a,b,c], and the concatenation of two sequences s_1 and s_2 as $s_1 +\!\!+ s_2$. We use $\in$ for sequence containment in addition to set containment and $[d | d \in D \wedge \phi(d)]$ for a sequence of d obtained by selecting them from D, all items that satisfy $\phi(d)$.

2.2 Dewey Order Encoding and XML Store

An XML document is modeled as an ordered tree. *Document order* in an XML document is a total order defined over the nodes in a tree, and this order is determined by a preorder traversal of the tree. This order plays an important role in the semantics of XQuery, especially in node creation and axis accesses. An XQuery expression is evaluated against an XML store which contains XML fragments with their document order. This store contains fragments that are created as intermediate results, in addition to the initial XML documents [12].

Dewey order encoding of XML nodes is a lossless representation of a position in the document order [14, 20]. In Dewey order, each node is represented by a path from a root using '.' : (1) a root node is encoded by $r \in \mathcal{R}$, where $\mathcal{R}$ is a countably infinite set of special codes; (2) say that a node a is the n-th child of a node b; then the Dewey code of a, $did(a)$, is $did(b).n$. The fact that the relative order of nodes in distinct trees is implementation-dependent leads to nondeterminism in XQuery. Therefore, if two

Dewey codes begin with different codes, it implies that the two nodes are in different ordered trees. By using Dewey order encoding, one can easily compute axis relations. For example, $\texttt{ancestor}(d_1, d_2)$ holds when d_1 has the form $d_2.n_1.n_2.\cdots.n_k$.

Let $\mathcal{T}$ be a set of symbols for element names, and $\mathcal{D}$ be a countably infinite set of Dewey codes on which a strict partial order $<$ and the equality $=$ is defined.

Definition 1 (Simple XML Store). *A simple XML store is a pair $St = (D, \nu)$, where (a) D is a finite subset of $\mathcal{D}$ and (b) ν is a partial function $\nu : \mathcal{D} \mapsto \mathcal{T}$ that maps a Dewey code to its element name.*

For instance, the store of the source S in Figure 1 is defined as $St_0 = (D_0, \nu_0)$, where $D_0 = \{s, s.1, s.1.1, s.1.1.1, s.1.2, s.1.2.1, s.2, s.2.1, s.2.1.1, s.2.2, s.2.2.1\}$ and $\nu_0(s) = \texttt{na}$, $\nu_0(s.1) = \texttt{lhs}$, $\nu_0(s.2) = \texttt{rhs}$, $\nu_0(s.1.1) = \nu_0(s.1.2) = \nu_0(s.2.1) = \nu_0(s.2.2) = \texttt{item}$, $\nu_0(s.1.1.1) = \texttt{a}$, $\nu_0(s.1.2.1) = \texttt{b}$, $\nu_0(s.2.1.1) = \texttt{c}$, $\nu_0(s.2.2.1) = \texttt{d}$. In what follows, we will refer to a simple XML store as an XML store. We denote the disjoint union of two stores St_1 and St_2 as $St_1 \cup St_2$ (combining D and ν independently).

Definition 2 (Value Equivalence, $\equiv_{(St_1,St_2)}$). *Given two stores St_1, St_2, and two nodes, d_1 in St_1 and d_2 in St_2, d_1 and d_2 are said to be value equal, denoted as $d_1 \equiv_{(St_1,St_2)} d_2$, if d_1 and d_2 refer to two isomorphic trees, i.e., there is a one-to-one function $h : D_1 \mapsto D_2$ with $D_1 = \{d | d \in D_{St_1} \wedge \texttt{ancestor-or-self}(d, d_1)\}$ and $D_2 = \{d | d \in D_{St_2} \wedge \texttt{ancestor-or-self}(d, d_2)\}$, such that for each d and $d' \in D_1$, it holds that (1) $h(d) \in D_2$, (2) $\nu(d) = \nu(h(d))$, and (3) $d < d'$ iff $h(d) < h(d')$. This definition can be extended to the value equivalence over two sequences, straightforwardly.*

2.3 Formal Semantics

The formal semantics of XQuery established by W3C is defined over XQuery Core, which is a subset of XQuery [23]. While XQuery Core does not have a location step expression, the reason why our target has is that (1) evaluating path expressions is more efficient than "for"-expressions [8, 18], although theoretically, it can be translated into "for"-expressions; and (2) previous work on XQuery dealt with location steps [13, 10, 9]. Figure 2 shows the semantics of our target XQuery using a set of inference rules. In these rules, a judgment of the form $St; En \vdash e \Rightarrow (St', s)$ indicates that the evaluation of expression e against the store St and environment En (mapping variables to values) results in a (new) store St' and value s. The semantics of sequence expressions, "let"-expressions and variables are straightforward. The semantics of a "for"-expression ($\textbf{for } \$v \textbf{ in } e_1 \textbf{ return } e_2$) is the concatenation of the results of e_2 evaluated N times for each item in the result of e_1 but with v in the environment bound to the item in question in the result of e_1, where N is the length of the sequence of the result of e_1. The semantics of the element constructor ($\langle t \rangle e \langle /t \rangle$) and location step ($e/\alpha :: \tau$) are worth futher attention because they are evaluated using the document order. The semantics of $\langle t \rangle e \langle /t \rangle$ is as follows. A new store St_2 that contains a new root node having t as its name and having contents is created. The contents are the value-equivalent sequence to the result of e. St_2 is added to the input store, and the newly

$$\frac{}{St; En \vdash () \Rightarrow (St, [])} \qquad \frac{St; En \vdash e_1 \Rightarrow (St_1, s_1) \quad \cdots \quad St_{N-1}; En \vdash e_N \Rightarrow (St_N, s_N)}{St; En \vdash (e_1, \ldots, e_N) \Rightarrow (St_N, s_1 +\!\!+ \ldots +\!\!+ s_N)}$$

$$\frac{\begin{array}{c} St; En \vdash e_1 \Rightarrow (St_0, [d_1, \cdots, d_N]) \\ St_0; En + \{\$v \mapsto d_1\} \vdash e_2 \Rightarrow (St_1, s_1) \\ \cdots \\ St_{N-1}; En + \{\$v \mapsto d_N\} \vdash e_2 \Rightarrow (St_N, s_N) \end{array}}{St; En \vdash \textbf{for } \$v \textbf{ in } e_1 \textbf{ return } e_2 \Rightarrow (St_N, s_1 +\!\!+ \cdots +\!\!+ s_N)}$$

$$\frac{\begin{array}{c} St; En \vdash e_1 \Rightarrow (St_1, s_1) \\ St_1; En + \{\$v \mapsto s_1\} \vdash e_2 \Rightarrow (St_2, s_2) \end{array}}{St, En \vdash \textbf{let } \$v := e_1 \textbf{ return } e_2 \Rightarrow (St_2, s_2)} \qquad \frac{}{St; En \vdash \$v \Rightarrow (St, En(\$v))}$$

$$\frac{\begin{array}{c} St; En \vdash e \Rightarrow (St_1, s_1) \qquad \text{a fresh } r \in \mathcal{R} \\ d \in D_{St_2} \Rightarrow d \text{ begins with } r \qquad \nu_{St_2}(r) = t \\ \mathbf{ddo}_{St_2}[d' | d' \in D_{St_2} \wedge \mathbf{child}(d', r)] = s_2 \qquad s_1 \equiv_{(St_1, St_2)} s_2 \end{array}}{St; En \vdash \langle t \rangle e \langle /t \rangle \Rightarrow (St \cup St_2, [r])}$$

$$\frac{\begin{array}{c} St; En \vdash e \Rightarrow (St_0, [d_1, \cdots, d_N]) \\ [d'_1 | d'_1 \in D_{St_0} \wedge \alpha(d'_1, d_1) \wedge \nu_{St_0}(d'_1) = \tau] = s_1 \\ \cdots \\ [d'_N | d'_N \in D_{St_0} \wedge \alpha(d'_N, d_N) \wedge \nu_{St_0}(d'_N) = \tau] = s_m \end{array}}{St; En \vdash e/\alpha :: \tau \Rightarrow (St_0, \mathbf{ddo}_{St_0}(s_1 +\!\!+ \cdots +\!\!+ s_m))}$$

Fig. 2. Semantics of XQuery using the simple XML store

created root node is returned. The semantics of $e/\alpha :: \tau$ is as follows. First, e is evaluated. Then, for each node d_i in its result, construct a sequence s_i such that for each content d'_i in s_i, d'_i is contained in St_0, and α-relation holds for d_i and d'_i, and the element name of d'_i is τ. The results of these sequences are concatenated. Finally, this sequence is sorted in the document order and duplicates are removed from it because an axis access by "/" requires sorting and duplicate elimination in the document order. This sorting without duplicates is performed by using the function $\mathbf{ddo}$ (distinct-doc-order).

Value equivalent expressions are introduced in order to prove the correctness of our fusion later.

Definition 3 (Value Equivalent Expressions). *Given a store St, an environment En, and two XQuery expressions e_1 and e_2, e_1 and e_2 are said to be value equivalent, if the following conditions hold; $St; En \vdash e_1 \Rightarrow (St_1, s_1)$, $St; En \vdash e_2 \Rightarrow (St_2, s_2)$ and $s_1 \equiv_{(St_1, St_2)} s_2$.*

3 Emulating XML Stores with Extended Dewey Codes

The problem of the existing fusion transformation is that the naive elimination of element constructors during the transformation does not preserve the context. To give a

correct fusion transformation, we should be able to emulate (keep track of) the context information (i.e., XML store) during the static transformation when an element is constructed. Our idea is to *lift dynamic operations on XML store to the static level of expression*, and it is based on the observation that Dewey order encoding of the result of the evaluation of an expression corresponds well to the structure of the expression.

3.1 XML Store Emulation on Expression

First, we show an important property for element constructors in terms of Dewey code: The Dewey order encoding of the result of an evaluation of an expression corresponds to the structure of the expression. This enables us to associate the static transformation world with the dynamic evaluation world by using Dewey code.

Given an element construction $\langle t \rangle e \langle /t \rangle$, we denote its relation with its result by $\langle t \rangle e \langle /t \rangle \sim r$ if there exist St, En, St' such that $St; En \vdash \langle t \rangle e \langle /t \rangle \Rightarrow (St', r)$.

Property 1 (Dewey code correspondence in element construction). For an element construction, $\langle t \rangle e \langle /t \rangle$, the following properties hold.

(i) $\langle t \rangle e \langle /t \rangle \sim r$, where $r \in \mathcal{R}$ and r is not in the input store.
(ii) $\langle t \rangle e \langle /t \rangle \sim r$ and $e \sim [r_1, \cdots, r_n]$ imply $r_i = r.i$.
(iii) For $\langle t \rangle (e_1, e_2) \langle /t \rangle$, $(e_1, e_2) \sim [r_1, r_2]$ and $d_1 \in r_1$ and $d_2 \in r_2$ imply $d_1 < d_2$.
(iv) $\langle t_1 \rangle e_1 \langle /t_1 \rangle \sim r_1$ and $\langle t_2 \rangle e_2 \langle /t_2 \rangle \sim r_2$ imply $r_1 \neq r_2$, where $r_1, r_2 \in \mathcal{R}$.

The above correspondence property hints that we should associate each expression with a Dewey code, so that these codes can be used to keep track of context information during the fusion transformation. For instance, for the element construction $\langle t \rangle (\$v/c, \$v/a) \langle /t \rangle$, we may give the following Dewey order encoding to the expression:

$$(\langle t \rangle (\$v/c)^{r.1}, (\$v/a)^{r.2} \langle /t \rangle)^r$$

where e^d denotes that d is the Dewey order encoding of the expression e (we will define this formally in Section 4.)

One difficulty, however, remains in associating Dewey codes to expressions to keep the context information: how do we deal with the "for" ("let") expressions in XQuery? We have to extend Dewey code for this purpose.

3.2 Extended Dewey Code

To be able to associate XQuery expressions with suitable context information, we propose an *extended Dewey code* (xD), defined by

$$\begin{aligned} \hat{d} &::= n\ \hat{x} \mid \underline{\epsilon} \mid \underline{[\hat{d}, \hat{d}, \ldots, \hat{d}]} \\ \hat{x} &::= \epsilon \mid \text{"."}\ \hat{d} \mid \underline{\text{"\#"}\ \hat{d}} \end{aligned}$$

where $n \in (\mathcal{R} \cup \mathcal{I})$ with $\mathcal{R}$ being a set of special codes[6] and $\mathcal{I}$ being a set of integers. It has a hierarchical structure, the same as in XQuery expressions, because xD is an

[6] The special code is used to exploit *Property* 1 (i).

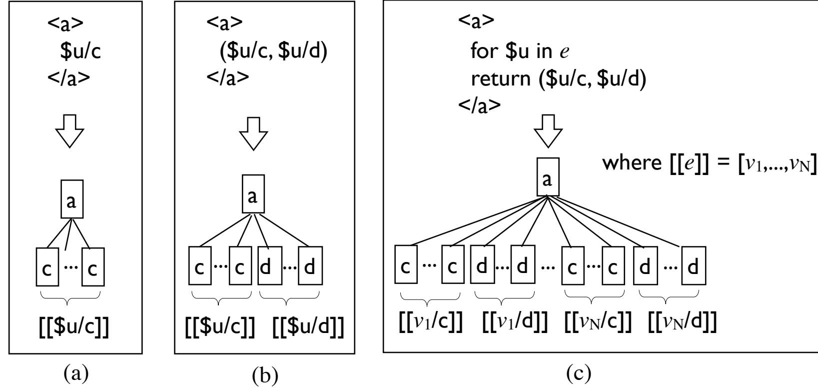

Fig. 3. A simple example for the document order in element creations

annotation for an XQuery expression. Here, the underlined parts are our extension, and ϵ is used for a termination, so, every xD ends with ϵ. Intuitively, the form of xD is as follows. ϵ is annotated to an expression, which does not occur inside an element constructor. A sequence construction has the form of sequence[7] is used. The delimiter ".", which plays the same role as in the original Dewey codes, is used to represent parent-child relationships.

The delimiter "#", which is our extension, represents the association of a "return" clause with a "for" or "let" expression and is used to resolve sortings with duplicate elimination for multiple "for" or "let" expressions that are derived from identical "for" or "let" expressions. Figure 3 (c) shows how an element is constructed with the "for" expression `(Q1)`.

Q1: $\langle a \rangle$ **for** $\$u$ **in** e **return** $(\$u/c, \$u/d)\langle /a \rangle$

To show the idea behind the design of our delimiter "#", let us consider the fusion transformation for the expression $(\texttt{(Q1)}/d, \texttt{(Q1)}/c)/$**self** $:: *$. For the expressions `(Q1)`$/d$ and `(Q1)`$/c$, we can get the *value equivalent* expressions `(Q2)` and `(Q3)`, respectively, from the XQuery semantics.

Q2: **for** $\$u$ **in** e **return** $\$u/d$

Q3: **for** $\$u$ **in** e **return** $\$u/c$

Now consider the following expression `(Q4)`.

Q4: $((\texttt{(Q2)}), (\texttt{(Q3)}))/$**self** $:: *$

[7] This sequence is the same as the data model of XQuery. So, it is flattened, and singleton and its element cannot be distinguished.

$$e^{\hat{d}} ::= \$v^{\hat{d}} \mid (e^{\hat{d}}, e^{\hat{d}}, ..., e^{\hat{d}})^{\hat{d}} \mid (e^{\hat{d}}/\alpha{::}\tau)^{\hat{d}} \mid (\textbf{for}\ \$\mathtt{v}\ \textbf{in}\ e^{\hat{d}}\ \textbf{return}\ e^{\hat{d}})^{\hat{d}}$$
$$\mid \quad (\textbf{let}\ \$\mathtt{v} := e^{\hat{d}}\ \textbf{return}\ e^{\hat{d}})^{\hat{d}} \mid (\langle t \rangle e^{\hat{d}} \langle /t \rangle)^{\hat{d}}$$

Fig. 4. Annotated XQuery

As described in the previous section, since axis access by "/" requires sorting and duplicate elimination in document order, the correct transformation of `(Q4)` should result in `(Q5)`, in which two "for" expressions `(Q2)` and `(Q3)` are merged.

Q5: $\textbf{for}\ \$u\ \textbf{in}\ e\ \textbf{return}\ (\$u/c, \$u/d)$

Here, we can capture the order of the two expressions in the "return" expressions by using "#". Thus, by encoding `(Q1)` into

$$(\langle a \rangle (\textbf{for}\ \$u\ \textbf{in}\ e\ \textbf{return}\ (\$v/c, \$v/d))^{r.1\#[1,2]} \langle /a \rangle)^{r}$$

and encoding `(Q2)` and `(Q3)` into

$$(\textbf{for}\ \$u\ \textbf{in}\ e\ \textbf{return}\ \$v/d)^{r.1\#2} \text{ and } (\textbf{for}\ \$u\ \textbf{in}\ e\ \textbf{return}\ \$v/c)^{r.1\#1}$$

we can apply the transformation to `(Q5)` (See Section 4), thanks to sorting on subsequences produced by the "for" expressions.

Returning to our extend Dewey codes, we can introduce the context position of sorting and duplicate elimination over $\hat{d}$ in a similar way to the original Dewey code (See Appendix A for details). Therefore, we can use the functions dc_sort and remove_dup for sorting and duplicate elimination, respectively. The difference from the sorting of the original Dewey code is in merging two extended codes sharing the same prefix until they reach $\#$. For instance, sorting $[r.1\#2, r.1\#1]$ results in $[r.1\#[1, 2]]$.

4 XQuery Fusion

This section describes our algorithm for automatic fusion of XQuery expressions so that unnecessary element constructions can be correctly eliminated. Basically, we will focus on fusing the following subexpression,

$$e/\alpha{::}\tau$$

so that unnecessary element constructions in the query expression in e are eliminated under the context of "selection" by $\alpha{::}\tau$.

We add annotations of the extended Dewey codes to the XQuery expression (Figure 4). 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.

$$
\begin{array}{rll}
\mathsf{peval}\ ()\ \Theta = & ()^{[]} & \\
\mathsf{peval}\ \$v\ \Theta = & \begin{cases} \Theta(\$v) & \text{if \$v is letvar} \\ \$v & \text{otherwise} \end{cases} & (\text{PEVR}) \\
\mathsf{peval}\ (e_1, ..., e_N)\ \Theta = & \underline{\text{let}}\, e'_i = \mathsf{peval}\ e_i\ \Theta & (\text{PESEQ}) \\
& \quad d_i = \mathsf{extract_dc}(e'_i) & \\
& \underline{\text{in}}\ \mathsf{flatten}\ ((e'_1, ..., e'_N)^{[d_1, .., d_N]}) & \\
\mathsf{peval}\ (e/\,\mathbf{child} :: \tau)\ \Theta = & \mathcal{F}_{\mathsf{c}}\,(\mathsf{peval}\ e\ \Theta)\ \tau & (\text{PECSTP}) \\
\mathsf{peval}\ (e/\,\mathbf{self} :: \tau)\ \Theta = & \mathcal{F}_{\mathsf{s}}\,(\mathsf{peval}\ e\ \Theta)\ \tau & (\text{PESSTP}) \\
\mathsf{peval}\ (e/\,\mathbf{parent} :: \tau)\ \Theta = & \mathcal{F}_{\mathsf{p}}\,(\mathsf{peval}\ e\ \Theta)\ \tau & (\text{PEPSTP}) \\
\mathsf{peval}\ (\mathbf{let}\ \$\mathtt{v} := e_1\ \mathbf{return}\ e_2)\ \Theta = & \underline{\text{let}}\, e'_1 = \mathsf{peval}\ e_1\ \Theta & \\
& \quad e'_2 = \mathsf{peval}\ e_2\ (\Theta \cup \{\$\mathtt{v} \mapsto (e'_1, \mathbf{let})\}) & \\
& \underline{\text{in}}\ e'_2 & (\text{PELET}) \\
\mathsf{peval}\ (\mathbf{for}\ \$\mathtt{v}\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2)\ \Theta = & \underline{\text{let}}\, e'_1 = \mathsf{peval}\ e_1\ \Theta & \\
& \quad e'_2 = \mathsf{peval}\ e_2\ (\Theta \cup \{\$\mathtt{v} \mapsto (e'_1, \mathbf{for})\}) & \\
& \quad d = \mathsf{extract_dc}\ e'_2 & \\
& \underline{\text{in}}\ (\mathbf{for}\ \$\mathtt{v}\ \mathbf{in}\ e'_1\ \mathbf{return}\ e'_2)^{\#d} & (\text{PEFOR}) \\
\mathsf{peval}\ (\langle t\rangle e\langle/t\rangle)\ \Theta = & \underline{\text{let}}\, e' = \mathsf{peval}\ e\ \Theta & (\text{PEEC}) \\
& \quad \text{a fresh } r \in \mathcal{R} & \\
& \underline{\text{in}}\ \mathsf{dc_assign}\ \langle t\rangle e'\langle/t\rangle\ r &
\end{array}
$$

Fig. 5. Fusion by partial evaluation

4.1 Fusion Transformation

Figure 5 summarizes our fusion transformation on XQuery expressions. Fusion is defined by a partial evaluation function peval:

$$\mathsf{peval}\ ::\ e \rightarrow \Theta \rightarrow e^{\hat{d}}$$

which accepts an XQuery expression and an environment Θ (mapping variables bound by "let" or "for" to expressions):

$$\Theta :: \mathit{Var} \rightarrow (e^{\hat{d}}, \mathbf{let}\,|\,\mathbf{for})$$

and produces a more efficient XQuery expression in which subexpressions are annotated by the extended Dewey codes. 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 annotations and get a normal XQuery expression as the final result.

The definition of peval in Figure 5 is straightforward. For a variable, if it is bounded by the outside "let", we retrieve its corresponding expression from the environment; otherwise, it must be a variable bound by the outside "for", and we leave it as is. For a sequence expression, we partially evaluate each element expression and

$$
\begin{array}{rcll}
\mathsf{dc_assign}\ ()^{\text{-}}\ r & = & ()^{[]} & \\
\mathsf{dc_assign}\ \$v^{\text{-}}\ r & = & \$v^{r} & \\
\mathsf{dc_assign}\ (e/c)^{\text{-}}\ r & = & (e/c)^{r} & \text{(DCSTP)} \\
\mathsf{dc_assign}\ (e_1, \ldots, e_n)^{\text{-}}\ r & = & \underline{\text{let}}\, r_0 = r & \text{(DCPSEQ)} \\
 & & \quad e'_i = \mathsf{dc_assign}\, e_i\ r_{i-1} & \\
 & & \quad r_i = \mathsf{succ}(\mathsf{extract_dc}\, e'_i) & \\
 & & \underline{\text{in}}\ (e_1, \ldots, e_n)^{[r_1, \ldots, r_n]} & \\
\mathsf{dc_assign}\ (<t>e</t>)^{\text{-}}\ r & = & \underline{\text{let}}\ e' = \mathsf{dc_assign}\, e_i\ r.1 & \text{(DCPEC)} \\
 & & \underline{\text{in}}\ <t>e'</t>^{r} & \\
\mathsf{dc_assign}\ (\textbf{for}\ \$v\ \textbf{in}\ e_0\ \textbf{return}\ e)^{\text{-}}\ r & = & \underline{\text{let}}\, e' = \mathsf{dc_assign}\, e\ 1 & \text{(DCPFOR)} \\
 & & \quad bs = \mathsf{extract_dc}\, e' & \\
 & & \underline{\text{in}}\ (\textbf{for}\ \$v\ \textbf{in}\ e_0\ \textbf{return}\ e')^{r\#bs} &
\end{array}
$$

Fig. 6. Dewey code propagation

group them into a new sequence annotated with Dewey codes from the results of each element expression. Note that we use flatten to remove nested sequences (e.g., $\mathsf{flatten}((e_{11}^{r_1}, e_{12}^{r_2})^{[r_1,r_2]}, e_3^{r_3})^{[[r_1,r_2],r_3]} = (e_{11}^{r_1}, e_{12}^{r_2}, e_3^{r_3})^{[r_1,r_2,r_3]}$), and extract_dc to get annotated Dewey code from an expression (i.e., $\mathsf{extract_dc}\ \ e^d = d$). For a location step expression $e/\alpha{::}\tau$, we perform fusion transformation to eliminate unnecessary element constructions in e after partially evaluating e. We will discuss the definitions of the three important fusion functions $\mathcal{F}_{\mathtt{c}}$, $\mathcal{F}_{\mathtt{s}}$, and $\mathcal{F}_{\mathtt{p}}$, later. 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. We do similarly for a "for" 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 into e', we create a new Dewey code for annotating this element and propagate this Dewey code information to all subexpressions in e' (with the function dc_assign) so that we can access (recover) this element constructor when processing the subexpressions of e'. It is this trick that helps to solve the problem of $e_2 \circ e_m$ in the Introduction.

Dewey Code Propagation Propagating the Dewey code of an element construction to its subexpressions plays an important role in constructing our fusion rules, described later, for correct fusion transformation.

Figure 6 defines a function $\mathsf{dc_assign}\ \ e^{-}\ r$:

$$\mathsf{dc_assign} :: e^{\hat{d}} \rightarrow \hat{d} \rightarrow e^{\hat{d}}$$

which is to propagate the Dewey code r into an annotated expression e by assigning proper new Dewey codes to e and its subexpressions. In what follows, we will explain some of the 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) horizontally numbers sequence expressions. The function succ is used to enforce numberings using a strictly greater value relative to previously

processed expressions (e.g., succ $r.1 = r.2$). (DCPEC) introduces a vertical structure to the numbering by initiating dc_assign for the subexpression e by adding ".1" to its second parameter. The equations that needs additional attention are (DCSTP) and (DCPFOR). In (DCSTP), it may seem unusual for dc_assign not to recurse subexpression e. However, considering that the path expression itself does not introduce an additional parent-child relationship and that dc_assign always handles expressions already partially evaluated expressions, there is no additional chance to simplify the path expression further by using the Dewey code allocated to the subexpression. In particular, the characteristic equation (DCPFOR), which introduces # structure to the Dewey code, numbers the expression e at the return clause. Note that the second parameter of the recursive call for e is reset to 1. bs that reflects the horizontal structure produced by the return clause is combined with the # sign to produce $r\#bs$ as the top level code allocated to the "for" expression.

Lemma 1. *From the definition of* dc_assign*, given an XQuery expression e, the extended Dewey code assigned by* dc_assign e^- r *satisfies* Property *1.*

Fusion Rules Our fusion transformation on $e/\alpha{::}\tau$ is based on the three fusion rules (functions) $\mathcal{F}_{\mathsf{c}}$, $\mathcal{F}_{\mathsf{s}}$ and $\mathcal{F}_{\mathsf{p}}$ in Figure 7 that respectively correspond to three axis types. The basic procedure is as follows:

1. Extract (get) subexpressions according to the axis α;
2. Select those that produce nodes whose name is equal to the tag name τ by using a filter;
3. Sort the remaining subexpressions according to their Dewey codes;
4. If the above sort step succeeds, remove the duplicated subexpressions and return its sequence as the result; otherwise, end fusion.

More concretely, let us consider the definition of $\mathcal{F}_{\mathsf{c}}$. 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 evaluating e, and use the filter(equal_to τ) function to keep those that are equal to τ, 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. This sorting may fail since not all of the Dewey codes are comparable. If the sorting succeeds, we return a sequence expression by removing all duplicated element subexpressions; otherwise, we end fusion by returning the original expression $e/$ **child** $::\ \tau$.

Our fusion transformation always terminates and is correct, as summarized by the following theorem.

Theorem 1 (Correctness of Fusion). *For an XQuery expression e, if* peval $e\ \{\} = e'^{d'}$ *then e and e′ are value equivalent expressions.*

Proof. (sketch): It is sufficient to show the correctness for location step expressions. For other expressions, it is straightforward to show the correctness by using structural induction on the expressions. For location step expressions, the correctness is implied by **Lemma** 1 and **lemma** 4 in the Appendix A together with the semantics of the location step expressions. □

Simple Example For $e_1 \circ e_m$ described in the introduction, our fusion function peval works as follows.

$$
\begin{array}{ll}
 & \mathsf{peval}\ \ e_1 \circ e_m\ \ \{\} \\
\rightsquigarrow & \{(\text{PECSTP}); (\text{PELET}); (\text{PEEC})\} \\
 & \mathbf{let}\ \$t\ :=\ \langle\mathsf{sa}\rangle\{(\ \langle\mathsf{lhs}\rangle/\mathsf{na}/\mathsf{rhs}/\mathsf{item}^{r.1.1}\langle/\mathsf{lhs}\rangle^{r.1}, \\
 & \qquad\qquad\qquad \langle\mathsf{rhs}\rangle/\mathsf{na}/\mathsf{lhs}/\mathsf{item}^{r.2.1}\langle/\mathsf{rhs}\rangle^{r.2})^{[r.1,r.2]}\}\langle/\mathsf{sa}\rangle^{r} \\
 & \mathbf{return}\ \ \mathbf{let}\ \$v\ :=\ (\$t/\mathsf{rhs}, \$t/\mathsf{lhs})\ \mathbf{return}\ \$v/\mathsf{item} \\
\rightsquigarrow & \{(\text{PELET}); (\text{PESEQ}); (\text{PECSTP}); (\text{PECSTP})\} \\
 & \mathbf{let}\ \$v\ :=\ (\ \langle\mathsf{rhs}\rangle/\mathsf{na}/\mathsf{lhs}/\mathsf{item}^{r.2.1}\langle/\mathsf{rhs}\rangle^{r.2}, \\
 & \qquad\qquad \langle\mathsf{lhs}\rangle/\mathsf{na}/\mathsf{rhs}/\mathsf{item}^{r.1.1}\langle/\mathsf{lhs}\rangle^{r.1})^{[r.2,r.1]} \\
 & \mathbf{return}\ \$v/\mathsf{item} \\
\rightsquigarrow & \{(\text{PECSTP})\} \\
 & \mathsf{remove_dup}\ (\mathsf{dc_sort}\ (/\mathsf{na}/\mathsf{lhs}/\mathsf{item}^{r.2.1}, /\mathsf{na}/\mathsf{rhs}/\mathsf{item}^{r.1.1})) \\
\rightarrow & \\
 & (/\mathsf{na}/\mathsf{rhs}/\mathsf{item}^{r.1.1}, /\mathsf{na}/\mathsf{lhs}/\mathsf{item}^{r.2.1})
\end{array}
$$

For $e_2 \circ e_m$, which is also from the introduction, peval performs the correct transformation.

$$
\begin{array}{ll}
 & \mathsf{peval}\ \ e_2 \circ e_m\ \ \{\} \\
\rightsquigarrow & \{(\text{PELET}); (\text{PESEQ}); (\text{PECSTP}); (\text{PECSTP})\} \\
 & \mathbf{let}\ \$t\ :=\ \langle\mathsf{sa}\rangle\{(\ \langle\mathsf{lhs}\rangle/\mathsf{na}/\mathsf{rhs}/\mathsf{item}^{r.1.1}\langle/\mathsf{lhs}\rangle^{r.1}, \\
 & \qquad\qquad\qquad \langle\mathsf{rhs}\rangle/\mathsf{na}/\mathsf{lhs}/\mathsf{item}^{r.2.1}\langle/\mathsf{rhs}\rangle^{r.2})^{[r.1,r.2]}\}\langle/\mathsf{sa}\rangle^{r} \\
 & \mathbf{return}\ \ \mathbf{let}\ \$v\ :=\ \$t/\mathsf{rhs}/\mathsf{item}\ \mathbf{return}\ \$v/.. \\
\rightsquigarrow & \{(\text{PELET}); (\text{PECSTP}); (\text{PEPSTP}); (\text{PEVR})\} \\
 & /\mathsf{na}/\mathsf{lhs}/\mathsf{item}^{r.2.1}/.. \\
\rightsquigarrow & \{(\text{PFUSION})\} \\
 & \langle\mathsf{rhs}\rangle/\mathsf{na}/\mathsf{lhs}/\mathsf{item}^{r.2.1}\langle/\mathsf{rhs}\rangle^{r.2}
\end{array}
$$

5 Related work

There are many studies on rewriting XQueries into other XQueries [11, 17, 13, 19]. The study most related to ours in the sense of eliminating redundant expressions is [11]. The authors of [11] proposed a rewriting optimization that replaces expressions which return empty sequences with () by using an emptiness detection based on static analysis. In contrast, our rewriting eliminates redundant element constructors as well.

Koch [13] and Page et al. [17] 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 the real world, however, practical expressions such as schema mapping always return newly constructed elements.

Tatarinov et al. proposed an efficient query reformulation in data integration systems, in which XML and XQuery are used for the data model and schema mapping, respectively [19]. In this system, the composition of the element construction is typical

because the schema mapping that maps one element to an other element involves element construction. They treat the actual reformulation algorithm as a black box. Our work attempts to open the box and exploit some of its properties.

Fusion has been extensively studied in the functional programming (FP) community [21, 2, 7, 16]. Referentially transparent FP languages allow naive fusion rules (F), as we saw in the Introduction, if the element constructor behaves like the constructors in FP. However, since the element constructor introduces a new node identity in each evaluation, thereby breaking the referential transparency, it is not directly applicable. It would be interesting to promote the identity as a first class object by using the technique described in [15], but our focus here is to perform XQuery-to-XQuery transformations, and the node identity is not a first class object in XQuery.

6 Conclusion

We proposed a new rewriting technique for XQuery fusion to eliminate unnecessary element constructions in the expressions while preserving the document order. The prominent feature of our framework is its static emulation of the XML store and assignment of extended Dewey codes to the expressions. The result is easy construction of correct fusion transformations.

We implemented a prototype system in Objective Caml. It consists of about 4600 lines of code. Currently it works stand-alone by reading XQuery expressions from standard input and produces rewritten XQueries to standard outputs. The system is available at `http://www.biglab.org/fusion`.

We believe that our approach can be generalized straightforwardly to handle the other axes including "transitive" axes like **ancestor**.

Acknowledgments We would like to thank the anonymous reviewers for their extensive and extremely helpful comments. Part of this work was supported by Grant-in-Aid for Scientific Research No. 22300012, No. 20500043, and No. 20700035.

$$\mathcal{F}_{\mathsf{c}} :: e^{\hat{d}} \to \tau \to e^{\hat{d}}$$

$$\mathcal{F}_{\mathsf{c}}\ e^{d}\ \tau = \begin{cases} \mathsf{remove_dup}\ (e'_1, ..., e'_N) & \text{if } \mathsf{dc_sort} \text{ succeeds} \\ (e^{d} / \mathbf{child} :: \tau)^{\epsilon} & \text{otherwise} \end{cases}$$

$$\text{where } (e'_1, ..., e'_N) = \mathsf{dc_sort}(\mathsf{filter}(\mathsf{equal_to}\ \tau)(\mathsf{get_children}\ e^{d})) \qquad \text{(cFUSION)}$$

$$\mathcal{F}_{\mathsf{s}} :: e^{\hat{d}} \to \tau \to e^{\hat{d}}$$

$$\mathcal{F}_{\mathsf{s}}\ e^{d}\ \tau = \begin{cases} \mathsf{remove_dup}\ (e'_1, ..., e'_N) & \text{if } \mathsf{dc_sort} \text{ succeeds} \\ (e^{d} / \mathbf{self} :: \tau)^{\epsilon} & \text{otherwise} \end{cases}$$

$$\text{where } (e'_1, ..., e'_N) = \mathsf{dc_sort}(\mathsf{filter}(\mathsf{equal_to}\ \tau)(\mathsf{get_self}\ e^{d})) \qquad \text{(sFUSION)}$$

$$\mathcal{F}_{\mathsf{p}} :: e^{\hat{d}} \to \tau \to e^{\hat{d}}$$

$$\mathcal{F}_{\mathsf{p}}\ e^{d}\ \tau = \begin{cases} \mathsf{remove_dup}\ (e'_1, ..., e'_N) & \text{if } \mathsf{dc_sort} \text{ succeeds} \\ (e^{d} / \mathbf{parent} :: \tau)^{\epsilon} & \text{otherwise} \end{cases}$$

$$\text{where } (e'_1, ..., e'_N) = \mathsf{dc_sort}(\mathsf{filter}(\mathsf{equal_to}\ \tau)(\mathsf{get_parent}\ e^{d})) \qquad \text{(pFUSION)}$$

$$\mathsf{get_children} :: e^{\hat{d}} \to e^{\hat{d}}$$

$$\mathsf{get_children}\ \$v^{\text{-}} = (\$v / \mathbf{child} :: *)^{\epsilon} \quad \mathsf{get_children}\ ()^{[]} = ()^{[]}$$

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

$$\text{where } e'_i = \mathsf{get_children}\ e_i \quad d_i = \mathsf{extract_dc}(e'_i) \qquad \text{(gcSEQ)}$$

$$\mathsf{get_children}\ (e / \mathbf{child} :: en)^{\text{-}} = (e / \mathbf{child} :: en / \mathbf{child} :: *)^{\epsilon}$$

$$\mathsf{get_children}\ (\langle en \rangle e^{d} \langle / en \rangle)^{\text{-}} = e^{d} \qquad \text{(gcEC)}$$

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

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

$$\text{where } (e_{i1}, \ldots, e_{in_i}) = \mathsf{get_children}\, e_i \qquad r_{ij} = \mathsf{extract_dc}\, e'_{ij}$$

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

$$\mathsf{get_self}, \mathsf{get_parent} :: e^{\hat{d}} \to e^{\hat{d}}$$

$$\mathsf{get_self}\ e^{r} = e^{r} \quad \mathsf{get_parent}\ e^{r.n} = \mathit{getExpGlobal}(r)$$

Fig. 7. Fusion rules for three kinds of $axis$

Bibliography

[1] S. Amano, L. Libkin, and F. Murlak. XML Schema Mappings. In *PODS*, pages 33–42, 2009.

[2] W. Chin. Safe Fusion of Functional Expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

[3] S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt, and B. Vance. Query optimization in revelation, an overview. *Data Eng.*, 14(2):58–62, 1991.

[4] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Framework for Logical XQuery Opimization. In *Proc of VLDB*, pages 168–179, 2004.

[5] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[6] M. Fernamdez, J. Hidders, P. Michiels, J. Simeon, and R. Vercammen. Optimizing sorting and duplicate elimination in xquery path expressions. In *Proceedings of 16th International Conference on Database and Expert Systems Applications (DEXA 2005)*, 2005.

[7] A. Gill, J. Launchbury, and S. L. P. Jones. A short cut to deforestation. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, New York, NY, USA, 1993. ACM Press.

[8] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM TODS*, June 2005.

[9] T. Grust, M. Mayr, and J. Rittinger. Let SQL drive the XQuery workhorse (XQuery join graph isolation). In *EDBT*, pages 147–158, 2010.

[10] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB*, pages 252–263, 2004.

[11] B. Gueni, T. Abdessalem, B. Cautis, and E. Waller. Pruning Nested XQuery Queries. In *CIKM*, pages 541–550, 2008.

[12] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A Light but Formal Introduction to XQuery. In *Second International XML Database Symposium,(XSym2004)*, pages 5–20, 2004.

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

[14] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig pattern Matching. In *Proc of VLDB*, 2005.

[15] A. Ohori. Representing object identity in a pure functional language. In *ICDT '90: Proceedings of the third international conference on database theory on Database theory*, pages 41–55, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[16] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. *SIGPLAN Not.*, 42(1):143–154, 2007.

[17] W. L. Page, J. Hidders, P. Michiels, J. Paredaens, and R. Vercammen. On the expressive power of node construction in XQuery. In *Proceedings of Eighth International Workshop on the Web and Databases (WebDB 2005)*, 2005.

[18] P. Parys. XPath evaluation in linear time with polynomial combined complexity. In J. Paredaens and J. Su, editors, *PODS*, pages 55–64. ACM, 2009.

[19] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *Proceedings of the ACM International Conference on Management of Data*, pages 539–550, 2004.

[20] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc of SIGMOD*, 2002.

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

[22] World Wide Web Consortium. XQuery1.0 : An XML Query Language, January 2007. W3C Recommendation.

[23] World Wide Web Consortium. XQuery1.0 and XPath2.0 Formal Semantics, January 2007. W3C Recommendation.

[24] L. Xu, T. W. Ling, H. Wu, and Z. Bao. DDE: From Dewey to a Fully Dynamic XML Labeling Scheme. In *SIGMOD Conference*, pages 719–730, 2009.

A Sorting without Duplicates on Extended Dewey Code

In this appendix, we use the standard list representation for the extended Dewey code to simplify our presentation. First, our extended Dewey code(xD) is redefined as follows.

$$\begin{array}{lll} ds & ::= [\,] \mid d : ds & \\ d & ::= \epsilon \mid n\ x & \text{where } n \in (\mathcal{R} \cup \mathcal{I}) \\ x & ::= \epsilon \mid "." \, d \mid "\#" \, ds & \end{array}$$

To show sorting without duplicates on xD, we define ordering and equivalence relation on xD.

A.1 Ordering on xD

We use $\prec_d$ and $\prec_x$ for ordering on d and x, respectively. We define partial order on xD.

Definition 4 (xD Order). *For ordering on d, $n_1\ x_1 \prec_d n_2\ x_2$ if and only if one of the following three conditions holds;*

- $n_1, n_2 \in \mathcal{R}$ *and* $n_1 = n_2$, $x_1 \prec_x x_2$.
- $n_1, n_2 \in \mathcal{I}$ *and* $n_1 < n_2$.
- $n_1, n_2 \in \mathcal{I}$ *and* $n_1 = n_2$, $x_1 \prec_x x_2$.

For ordering on x,

- $.\ d_1 \prec_x .\ d_2$ *if and only if* $d_1 \prec_d d_2$ *holds.*
- $\epsilon \prec_x x_1$ *if and only if* $x_1 \neq \epsilon$ *holds.*

Lemma 2 (Transitivity of xD Order). *If $d_1 \prec_d d_2$ and $d_2 \prec_d d_3$ then $d_1 \prec_d d_3$.*

Proof. Structural induction on d is used.

We use $\sim_d$ and $\sim_x$ for equivalence relation on d and x, respectively. We define equivalence relation on xD.

Definition 5 (Equvalence Relation). *For equivalence relation on d, $n_1\ x_1 \sim_d n_2\ x_2$ if and only if $n_1 = n_2$ and $x_1 \sim_x x_2$.*
For equivalence relation on x,

- $\epsilon \sim_x \epsilon$.
- $.\ d_1 \sim_x .\ d_2$ *if and only if* $d_1 \sim_d d_2$ *holds.*
- $\#\ ds_1 \sim_x \#\ ds_2$ *if and only if* $ds_1 +\!\!+ ds_2$ *is* sortable.

Definition 6 (Sortable). *For given a list of xD ds_1, ds_1 is sortable if and only if one of the following three conditions holds;*

- $ds_1 = [\,]$
- $ds_1 = d_1 : [\,]$
- $ds_1 = d_2 : ds_2$ *and* $\forall d' \in ds_2 (d_2 \prec_d d' \vee d' \prec_d d_2 \vee d' \sim_d d_2)$

Lemma 3 (Irreflexivity of $\prec_d$). *$\prec_d$ is irreflexive from its definition.*

Theorem 2 (Reflexive partial order of $\precsim$). *$\precsim$ is a reflexive partial order.*

Surprisingly, both duplicate eliminating and merging two xD codes can be defined as the following one algorithm.

Definition 7 (Duplicate Elimination and Merging). *Given two xD code n_1x_1 and n_2x_2 where $n_1x_1 \sim_d n_2x_2$, both duplicate eliminating and merging, $n_1x_1 \oplus_d n_2x_2$ is defined by the following inference rules;*

$$\frac{(x_1 \oplus_x x_2) \rightarrow x_3}{(n_1x_1 \oplus_d n_2x_2) \rightarrow n_1\ x_3}$$

$$\frac{}{(\epsilon \oplus_x \epsilon) \rightarrow \epsilon} \qquad \frac{(d_1 \oplus_d d_2) \rightarrow d_3}{(.d_1 \oplus_x .d_2) \rightarrow .d_3} \qquad \frac{\mathbf{xDDO}(ds_1 \mathbin{+\!\!+} ds_2) \rightarrow ds_3}{(\#ds_1 \oplus_x \#ds_2) \rightarrow \#ds_3}$$

Definition 8 (Sorting without Duplicates on xD, xDDO). *For a given list of xD ds_1 where ds_1 is sortable, sorting without duplicates on ds_1 (**xDDO** ds_1) is defined straightforwardly by using $\precsim$ and $\oplus_d$.*

Lemma 4. *For a given sortable list of xD ds_1, the result of the sorting without duplicate on ds_1 (**xDDO** ds_1) is strictly ordered under $\precsim$ from its definition.*