Rewriting XQuery to Avoid Redundant Expressions based on Static Emulation of XML Store

Hiroyuki Kato
kato@nii.ac.jp

Soichiro Hidaka
hidaka@nii.ac.jp

Zhenjiang Hu
hu@nii.ac.jp

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

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

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

ABSTRACT

Rewriting composite expressions based on eliminating intermediate results generated by redundant expressions is a traditional optimization technique (known as fusion) 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. We propose a fusion algorithm for this kind of composite XQuery. The XQuery fusion is more difficult than the existing fusion, because naive elimination of node creations does not preserve document order. The document order plays an important role in XQuery. An XQuery expression is evaluated against an XML store which contains XML fragments that are created as intermediate results, in addition to initial XML documents with their document order. So, the XML store is updated as the expression with node creations is evaluated. 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 to avoid redundant expressions is enabled. This emulation is achieved by using an adornment code called extended Dewey's assigned to the occurrences of expressions. By using this static emulation, our XQuery fusion avoids unnecessary expressions including node creations while preserving the document order in XML.

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 [11]. 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) [21, 2, 6] 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 [19]. We propose, in this paper, a fusion algorithm for this kind of composite XQuery.

The XQuery fusion is more difficult than the existing fusion [21], because naive elimination of node creations does not preserve document order. For example, consider the following two expressions `(Q1)` and `(Q2)` in XQuery.

`(Q1)`: $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle/c$

`(Q2)`: $\$v/c$

For an arbitrary store — assuming identical bindings of the externally defined variable $\$v$ — both `(Q1)` and `(Q2)` always return a value equivalent data, which is precisely equal when they are serialized and output by the query processor as a final result. So, `(Q1)` has redundant expressions, the node construction for t and the path expression $\$v/a$. However, as intermediate results in a query processor, two data evaluated by `(Q1)` and `(Q2)` populate in different document order. When $\$v/c$ does not result in an empty sequence [1], the nodes produced by `(Q1)` populate in the new document order created by the element constructor $\langle t\rangle(\$v/c, \$v/a)\langle/t\rangle$ in `(Q1)`, whereas the nodes returned by `(Q2)` populate in the document order existing in the input store. Consequently, if you take further step along parent axis for both queries, namely, `(Q1)/..` 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.

Copyright is held by the author/owner.
International Workshop on Programming Language Techniques for XML (PLAN-X 2009),
Jan. 24, 2009, Savannah, Georgia, U.S.A.

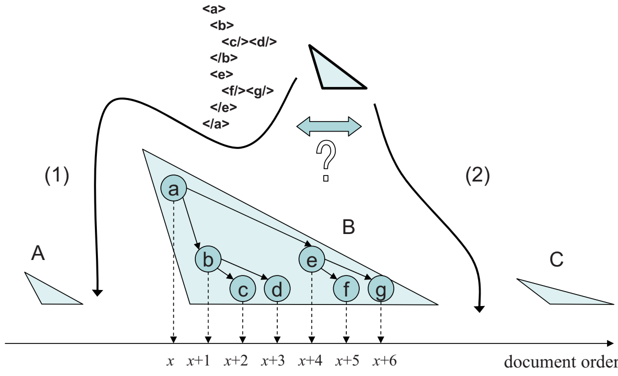


Figure 1: Node creation in the document order

`(Q2)/..` , now it is easy to see the differences as intermediates 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.

In XQuery, node construction is one source of non-determinism. Although expressions that contain element constructors are non-deterministic with respect to document order, (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 sequence. This property enables us to emulate newly created document order, statically.

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 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 [15, 20]. 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 have implemented our XQuery fusion in Objective Caml with about 4,600 loc, and all the examples in this paper have been passed by the system.

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. We discuss implementation issues in Section 4, and related work in Section 5. We conclude the paper in Section 6.

2. STATIC EMULATION OF STORE

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 `(Q3)` in XQuery.

$$\texttt{(Q3):} \quad (\langle h\rangle\langle i/\rangle\langle/h\rangle, \langle j\rangle\langle k/\rangle\langle/j\rangle)$$

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 [11] [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 [15, 20]. 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.

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

Figure 2: Pure Dewey code

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 nodes and edges in [11]. We assume a set of names $\mathcal{N}$ used for element

[2]The semantics of XQuery is formally given by [22]. However, due to not being self contained[17] and to simplify the discussion, we refer [11] instead.

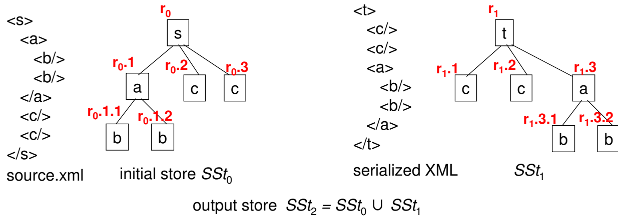


Figure 3: An input document source.xml, SSt_0 and output store by `(Q4)`.

names and a countably infinite set of Dewey Code $\mathcal{D}$, which is depicted by D in Figure 2.

DEFINITION 2.1 (Simple XML Store). *A simple XML store is a 3-tuple* $SSt = (D, <, \nu)$ *where*

- *D is a finite subset of $\mathcal{D}$;*
- *$<$ is a strict partial order on D;*
- *$\nu : D \mapsto \mathcal{N}$ maps the Dewey codes to their node name.*

Both the strict partial order $<$ and the equality $=$ on $\mathcal{D}$ are straightforward.

Evaluating an element constructor against an input simple store will add a tree into the input store. Consider the following XQuery expression `(Q4)` when given the input document `source.xml` shown in Figure 3.

```
(Q4): 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 $\{\mathtt{s}, \mathtt{a}, \mathtt{c}\} \subset \mathcal{N}$,

evaluating `(Q4)` 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 $r_1 \in \mathcal{S} \wedge r_1 \neq r_0$
- $\nu_2 = \nu_0 + \{r_1 \mapsto \mathtt{t},\ r_1.1 \mapsto \mathtt{c},\ r_1.2 \mapsto \mathtt{c},$ $r_1.3 \mapsto \mathtt{a},\ r_1.3.1 \mapsto \mathtt{b},\ r_1.3.2 \mapsto \mathtt{b}\}$ where $\mathtt{t} \in \mathcal{N}$

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.

Axis Access in Store. We will see how a sorting and duplicate elimination in document order is performed using illustrative examples step by step. Consider the following expression `(Q5)`.

```
(Q5): let $u := (Q3)
         return ($u/a,$u/c,$u/a)
```

Since `(Q4)` returns a node encoded by r_1, child nodes of r_1 in SSt_2 are $r_1.1$, $r_1.2$ and $r_1.3$. Among these nodes $r_1.3$ is the only node which is mapped to `a` by ν_2. So, `$u/a` in `(Q5)` returns $r_1.3$. Likewise, `(Q5)` returns a sequence of nodes $r_1.3\ r_1.1\ r_1.2$ and $r_1.3$ in this order. So, the serialized data `(Q5)` returns is as follows;

```
<a><b/><b/></a>,<c/>,<c/>,<a><b/><b/></a>
```

Now consider the following expression `(Q6)`

```
(Q6): (Q4)/self::*
```

Since the semantics of axis access "/" requires the sorting and duplicate elimination in document order, `(Q6)` returns a sequence of nodes $r_1.1\ r_1.2$ and $r_1.3$ in this order. So, the serialized data `(Q6)` returns is as follows;

```
<c/>,<c/>,<a><b/><b/></a>
```

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 sequence[11]. 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 $(D, <, \nu)$.

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) $\forall d \in [\![e]\!], d = [\![\ \texttt{<}en\texttt{>}e\texttt{</}en\texttt{>}]\!].n$[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. For an element constructor, `<t>$v/c,$v/a</t>`, which is in `(Q1)` as a subexpression, the Dewey encoding of the expression results in

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

where e^d denotes that d is the Dewey encoding of the expression e, and we will define this as "annotated XQuery expressions" in the next section.

Now, consider `(Q1)` again. Since the element constructor is encoded by r, child axis of r in this expression are $(\$v/c)^{r.1}$ and

[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.

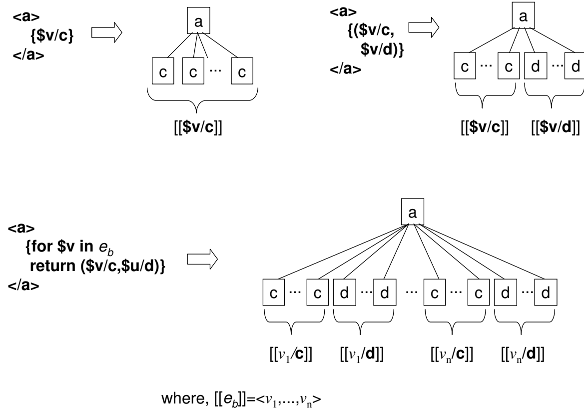


Figure 4: A simple example for the document order in element creations

$(\$v/a)^{r.2}$. Among these expressions $(\$v/c)^{r.1}$ is the only expression which generates c elements. So, we can get $(\$v/c)$ as a rewritten result. This manipulation is done by using PROPERTY 2.2. For (Q1)/.., we can get $\langle t\rangle \$v/c, \$v/a\langle /t\rangle$ by using Dewey encoding of the $\$v/c$, namely, $r.1$.

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 (Q7), bellow.

(Q7): $\langle a\rangle$ for $\$u$ in e_b return $(\$v/c, \$v/d)\langle /a\rangle$

For the expressions (Q7)$/d$ and (Q7)$/c$ we can get easily the *value equivalent* expressions (Q8) and (Q9), respectively by using PROPERTY 2.2.

(Q8): for $\$u$ in e_b return $\$v/d$

(Q9): for $\$u$ in e_b return $\$v/c$

Then, consider the following expression (Q10).

(Q10): $(((\text{Q8})),((\text{Q9})))/$ self $:: *$

As described in the previous subsection, since axis access by "/" requires the sorting and duplicate elimination in document order, the correct transformation of (Q10) should result in (Q11), in which two "for" exrpessions (Q8) and (Q9) are merged according to sort expressions appeared in the "return" expression.

(Q11): for $\$u$ in e_b return $(\$v/c, \$v/d)$

The order of expressions occurring in the "return" expressions is represented by delimiter "#". So, by encoding (Q7) to

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

and encoding (Q8) and (Q9) to

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

and

$$(\text{for } \$u \text{ in } e_b \text{ return } \$v/c)^{r.1\#1}$$

respectively, we can get (Q11). To achieve sorting on subsequences produced by "for" expressions, the extended Dewey code has "#" structure to constitute branches on it.

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

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

Figure 6: Abstract syntax of the extended Dewey code

3. XQUERY FUSION

In this section, we propose 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

$$e/\alpha::en$$

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

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 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. The partial order on the extended Dewey codes are essentially the dictionary order. For ex-

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

Figure 7: XQuery^D

ample, $r.1.2 < r.1.3$, $r.1 < r.1.2$ 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 different data structures of two "for" expressions.

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. Fusion is defined by a partial evaluation function peval:

$$\mathsf{peval} \ :: \ XQuery \rightarrow \Theta \rightarrow XQuery^D$$

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

$$\Theta :: \ Var \rightarrow (XQuery^D, \mathsf{let} \mid \mathsf{for})$$

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 annotation and give a normal XQuery expression as the final result.

The definition of peval in Figure 8 is rather straightforward. For a constant expression c, we return itself but annotate it with the Dewey code ϵ. 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 it is. 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 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 $e/\alpha{::}en$, we perform fusion transformation to eliminate unnecessary element construction in e after partially evaluating e. We will discuss the definitions of the three important fusion functions peval $(e/\,\texttt{child} :: en)$, peval $(e/\,\texttt{self} :: en)$, peval $(e/\,\texttt{parent} :: en)$ 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. For a "for" expression, we do similarly as for a "let" expression except that we finally produce a new "for" expression by glueing 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 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 `(Q1)/..` 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 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 relarive 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) child_fusion, self_fusion and parent_fusion 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, otherwise we give up fusion by returning the original expression $e/\,\texttt{child} :: en$.

3.2.3 Examples

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

$$\text{peval } c \ \Theta = c^{\epsilon}$$

$$\text{peval } \$v \ \Theta = \begin{cases} \Theta(\$v) & \text{if \$v is letvar} \\ \$v & \text{otherwise} \end{cases} \quad (\text{PEVR})$$

$$\text{peval } (e_1, ..., e_N) \ \Theta = \underline{\text{let}} \ e'_i = \text{peval } e_i \ \Theta \quad d_i = \text{extract_dc}(e'_i) \quad \underline{\text{in}} \ \text{flatten } ((e'_1, ..., e'_N)^{[d_1,..,d_N]}) \quad (\text{PESEQ})$$

$$\text{peval } (e/\texttt{child} :: en) \ \Theta = \text{child_fusion } (\text{peval } e \ \Theta) \ en \quad (\text{PECSTP})$$

$$\text{peval } (e/\texttt{self} :: en) \ \Theta = \text{self_fusion } (\text{peval } e \ \Theta) \ en \quad (\text{PESSTP})$$

$$\text{peval } (e/\texttt{parent} :: en) \ \Theta = \text{parent_fusion } (\text{peval } e \ \Theta) \ en \quad (\text{PEPSTP})$$

$$\text{peval } (\text{let } \$\text{v} := e_1 \text{ return } e_2) \ \Theta = \underline{\text{let}} \ e'_1 = \text{peval } e_1 \ \Theta \quad e'_2 = \text{peval } e_2 \ (\Theta \cup \{\$\text{v} \mapsto (e'_1, \text{let})\}) \quad \underline{\text{in}} \ e'_2 \quad (\text{PELET})$$

$$\text{peval } (\text{for } \$\text{v in } e_1 \text{ return } e_2) \ \Theta = \underline{\text{let}} \ e'_1 = \text{peval } e_1 \ \Theta \quad e'_2 = \text{peval } e_2 \ (\Theta \cup \{\$\text{v} \mapsto (e'_1, \text{for})\}) \quad d = \text{extract_dc } e'_2 \quad \underline{\text{in}} \ (\text{for } \$\text{v in } e'_1 \text{ return } e'_2)^{\#d} \quad (\text{PEFOR})$$

$$\text{peval } (\langle en \rangle e \langle /en \rangle) \ \Theta = \underline{\text{let}} \ e' = \text{peval } e \ \Theta \quad r = new\ rootD \quad e'' = \text{dc_assign } e' \ r.1 \quad \underline{\text{in}} \ (\langle en \rangle e'' \langle /en \rangle)^r \quad (\text{PEEC})$$

Figure 8: Fusion by partial evaluation

$$\text{dc_assign } c^- \ r = c^r$$

$$\text{dc_assign } \$v^- \ r = \$v^r$$

$$\text{dc_assign } (e/c)^- \ r = (e/c)^r \quad (\text{DCSTP})$$

$$\text{dc_assign } (e_1, \ldots, e_n)^- \ r = \underline{\text{let}} \ r_0 = r \quad e'_i = \text{dc_assign } e_i \ r_{i-1} \quad r_i = \text{succ}(\text{extract_dc } e'_i) \quad \underline{\text{in}} \ (e_1, \ldots, e_n)^{[r_1, \ldots, r_n]} \quad (\text{DCPSEQ})$$

$$\text{dc_assign } (\texttt{<}t\texttt{>}e\texttt{</}t\texttt{>})^- \ r = \underline{\text{let}} \ e' = \text{dc_assign } e_i \ r.1 \quad \underline{\text{in}} \ \texttt{<}t\texttt{>}e'\texttt{</}t\texttt{>}^r \quad (\text{DCPEC})$$

$$\text{dc_assign } (\text{for } \$v \text{ in } e_0 \text{ return } e)^- \ r = \underline{\text{let}} \ e' = \text{dc_assign } e \ 0 \quad bs = \text{extract_dc } e' \quad \underline{\text{in}} \ (\text{for } \$v \text{ in } e_0 \text{ return } e')^{r\#bs} \quad (\text{DCPFOR})$$

Figure 9: Dewey code propagation

First, for `(Q1)` our fusion function peval computes as follows.

$$\begin{aligned} & \text{peval } (((\langle t \rangle (\$v/c, \$v/a) \langle /t \rangle))/c) \ \Theta \\ = & \quad \{(\text{PECSTP}); (\text{PEEC})\} \\ & \text{child_fusion } (\langle t \rangle ((\$v/c)^{r.1}, (\$v/a)^{r.2}) \langle /t \rangle)^r \ c \\ = & \quad \{\text{definition of child_fusion}\} \\ & (\$v/c)^{r.1} \end{aligned}$$

So for `(Q1)`/.., which is from the introduction, peval performs the correct transformation.

$$\begin{aligned} & \text{peval } (\texttt{(Q1)}/..) \ \Theta \\ = & \quad \{(\text{PEPSTP}); (\text{PECSTP}); (\text{PEEC})\} \\ & \text{parent_fusion } (\$v/c)^{r.1} \ * \\ = & \quad \{\text{definition of parent_fusion}\} \\ & (\langle t \rangle ((\$v/c)^{r.1}, (\$v/a)^{r.2}) \langle /t \rangle)^r \end{aligned}$$

Next, consider the following expression `(Q12)`,

$$\texttt{(Q12)}: \ \text{let } \$v := \langle a \rangle () \langle /a \rangle \text{ return } (\$v, \$v)/\texttt{self} :: a$$

In `(Q12)` subexpression $(\$v, \$v)/\texttt{self} :: a$ is redundant because duplicate elimination is needed for this subexpression. The function peval eliminates duplicate.

$$\begin{aligned} & \text{peval } (\texttt{(Q12)}) \ \Theta \\ = & \quad \{(\text{PELET}); (\text{PEEC}); (\text{PESEQ}); (\text{PEVR})\} \\ & \text{self_fusion } ((\langle a \rangle () \langle /a \rangle)^r, (\langle a \rangle () \langle /a \rangle)^r)^{[r,r]} \ a \\ = & \quad \{\text{definition of self_fusion}\} \\ & (\langle a \rangle () \langle /a \rangle)^r \end{aligned}$$

For more complicated case, we show that a for expression occurring inside an element constructor appends "#" to the extended Dewey code. This is the prominent feature of our extension to the extended Deweys' which is explained using `(Q10)` described in Section 2.2. Before we explain `(Q10)`, consider `(Q13)` which is the subexpression of `(Q10)`.

$$\text{child_fusion} :: XQuery^D \rightarrow QName \rightarrow XQuery^D$$

$$\text{child_fusion}\ e\ en = \underline{\text{let}}\ (e'_1, ..., e'_N) = \text{dc_sort}(\text{filter}(\text{equal_to}\ en)(\text{get_children}\ e)) \quad \text{(cFusion)}$$
$$\underline{\text{in}} \begin{cases} \text{remove_duplicate}\ (e'_1, ..., e'_N) & \text{if dc_sort succeeds} \\ (e/\texttt{child} :: en)^\epsilon & \text{otherwise} \end{cases}$$

$$\text{self_fusion} :: XQuery^D \rightarrow QName \rightarrow XQuery^D$$

$$\text{self_fusion}\ e\ en = \underline{\text{let}}\ (e'_1, ..., e'_N) = \text{dc_sort}(\text{filter}(\text{equal_to}\ en)(\text{get_self}\ e)) \quad \text{(sFusion)}$$
$$\underline{\text{in}} \begin{cases} \text{remove_duplicate}\ (e'_1, ..., e'_N) & \text{if dc_sort succeeds} \\ (e/\texttt{self} :: en)^\epsilon & \text{otherwise} \end{cases}$$

$$\text{parent_fusion} :: XQuery^D \rightarrow QName \rightarrow XQuery^D$$

$$\text{parent_fusion}\ e\ en = \underline{\text{let}}\ (e'_1, ..., e'_N) = \text{dc_sort}(\text{filter}(\text{equal_to}\ en)(\text{get_parent}\ e)) \quad \text{(pFusion)}$$
$$\underline{\text{in}} \begin{cases} \text{remove_duplicate}\ (e'_1, ..., e'_N) & \text{if dc_sort succeeds} \\ (e/\texttt{parent} :: en)^\epsilon & \text{otherwise} \end{cases}$$

$$\text{get_children} :: XQuery^D \rightarrow XQuery^D$$
$$\text{get_children}\ c = ()^{[]}$$
$$\text{get_children}\ \$v^{-} = (\$v/\texttt{child} :: *)^\epsilon$$
$$\text{get_children}\ ()^{[]} = ()^{[]}$$
$$\text{get_children}\ (e_1, ..., e_N)^{-} = \underline{\text{let}}\ e'_i = \text{get_children}\ e_i,\ d_i = \text{extract_dc}(e'_i)\ \underline{\text{in}}\ \text{flatten}\ ((e'_1, ..., e'_N)^{[d_1,..,d_N]}) \quad \text{(gcSeq)}$$
$$\text{get_children}\ (e/\texttt{child} :: en)^{-} = (e/\texttt{child} :: en)^\epsilon$$
$$\text{get_children}\ (\text{for } \$v \text{ in } e \text{ return } (e_1, ..., e_N))^{r\#[b_1,...,b_N]} = \underline{\text{let}}\ (e_{i1}, \ldots, e_{in_i}) = \text{get_children}\ e_i,\ r_{ij} = \text{extract_dc}\ e'_{ij}$$
$$\underline{\text{in}} \left(\text{for } \$v \text{ in } e \text{ return } \begin{matrix} (e_{11}, e_{12}, \ldots, e_{1n_1}, \\ e_{21}, e_{22}, \ldots, e_{2n_2}, \\ \cdots \\ e_{N1}, e_{N2}, \ldots, e_{Nn_n}) \end{matrix} \right)^{r\# \begin{matrix}[b_1.r_{11}, \ldots, b_1.r_{1n_1}, \\ b_2.r_{21}, \ldots, b_2.r_{2n_2}, \\ \ldots \\ b_N.r_{N1}, \ldots, b_N.r_{Nn_n}]\end{matrix}} \quad \text{(gcFor)}$$

$$\text{get_self} :: XQuery^D \rightarrow XQuery^D$$
$$\text{get_self}\ e^r = e^r$$

$$\text{get_parent} :: XQuery^D \rightarrow XQuery^D$$
$$\text{get_parent}\ e^{r.(n|?)} = getExpGlobal(r)$$

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

$\texttt{(Q13)}$: $\langle a \rangle$ for $\$v$ in e_b return $(\$v/c, \$v/d)\langle/a\rangle$

Partial evaluation of $\texttt{(Q13)}$ assigns the extended Dewey code.

$$\begin{aligned} & \text{peval}\ (\texttt{(Q13)})\ \Theta \\ = & \quad \{(\text{peEc}); (\text{peFor}); (\text{dcpFor})\} \\ & (\langle a \rangle (\text{for } \$v \text{ in } e_b \text{ return } ((\$v/c)^1, (\$v/d)^2))^{r.1\#[1,2]} \langle /a \rangle)^r. \end{aligned}$$

So, $(\texttt{(Q13)}/d)$ is transformed in the following way.

$$\begin{aligned} & \text{peval}\ (\texttt{(Q13)}/d)\ \Theta \\ = & \quad \{(\text{peCStp})\} \\ & \text{child_fusion}\ (\text{peval}\ (\texttt{(Q13)})\ \Theta)\ d \\ = & \quad \{(\text{cFusion})\} \\ & (\text{for } \$v \text{ in } e_b \text{ return } ((\$v/d)^2))^{r.1\#[2]}. \end{aligned}$$

Also, $(\texttt{(Q13)}/c)$ is transformed in the following way.

$$\begin{aligned} & \text{peval}\ (\texttt{(Q13)}/c)\ \Theta \\ = & \quad \{(\text{peCStp})\} \\ & \text{child_fusion}\ (\text{peval}\ (\texttt{(Q13)})\ \Theta)\ c \\ = & \quad \{(\text{cFusion})\} \\ & (\text{for } \$v \text{ in } e_b \text{ return } ((\$v/c)^1))^{r.1\#[1]}. \end{aligned}$$

Now, return to $\texttt{(Q10)}$. With the above results, partial evaluation of peval performs as follow:

$$\begin{aligned} & \text{peval}\ (\texttt{(Q10)})\ \Theta \\ = & \quad \{(\text{peLet}); (\text{peEc}); (\text{peSeq})\} \\ & \text{self_fusion}\ (e_1, e_2)^{[r.1\#[2], r.1\#[1]]}\ * \\ = & \quad \{\text{definition of self_fusion}\} \\ & (\text{for } \$v \text{ in } e_b \text{ return } ((\$v/c)^1, (\$v/d)^2))^{r.1\#[1,2]} \end{aligned}$$

where, $e_1 = (\text{for } \$v \text{ in } e_b \text{ return } ((\$v/d)^2))^{r.1\#[2]}$ and
$e_2 = (\text{for } \$v \text{ in } e_b \text{ return } ((\$v/c)^1))^{r.1\#[1]}$.

4. IMPLEMENTATION

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 Appendix A.
- 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,
  - 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.

  The fusion function peval adds information for annotated XQuery to this relation. The function $getExpGlobal(r)$ is implemented by using this relation.
- applying auxiliary transformation rules such as monad laws[16] on "for" expressions. These transformation help to fuse the location step expressions.

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

For input expression (Q12), our system generates the following output `expr=elem a {()}` with the global environment as 4-ary relation. Note that `elem a {()}` is the internal representation of $\langle a \rangle\ ()\ \langle /a \rangle$ in our system. The pair (`Vtx`,`Edg`) represents abstract syntax trees of XQuery expressions by set of nodes (`Vtx`) and set of directed labeled edges (`Edg`) between nodes (for example, `e(9,8)` denotes an edge labeled `e` from node 9 to 8, meaning that an element construction expression is located at node 9 and has content represented by node 8).

```
#############
RESULT peval
ture
th=
Rel=
 oidDS=25, cxtDS=Nil, ds=[1], esDS=(elem a {()})
 oidDS=16, cxtDS=Nil, ds=[1], esDS=(elem a {()})
 oidDS=1, cxtDS=Nil, ds=[], esDS=()
 oidDS=8, cxtDS=Nil, ds=[], esDS=()
 oidD=9, cxtD=Nil, d=1, eD=elem a {()}
 oidDS=11, cxtDS=Nil, ds=[], esDS=()
 oidDS=12, cxtDS=Nil, ds=[1], esDS=(elem a {()})
 oidDS=13, cxtDS=Nil, ds=[1,1], esDS=(elem a {()},elem a {()})
 oidDS=18, cxtDS=Nil, ds=[], esDS=()
 oidDS=19, cxtDS=Nil, ds=[1], esDS=(elem a {()})
 oidDS=20, cxtDS=Nil, ds=[1,1], esDS=(elem a {()},elem a {()})
 oidDS=21, cxtDS=Nil, ds=[], esDS=()
 oidDS=22, cxtDS=Nil, ds=[1], esDS=(elem a {()})
 oidDS=23, cxtDS=Nil, ds=[1,1], esDS=(elem a {()},elem a {()})
 oidDS=24, cxtDS=Nil, ds=[1], esDS=(elem a {()})

Vtx=[25,16,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,17,18,19,20,21,22,23,24]
Edg={h(25,9),t(25,1),h(16,9),t(16,11),e(2,1),h(5,3),t(5,4),s(6,5),b(7,2),
     r(7,6),e(9,8),h(10,9),t(10,9),h(12,9),t(12,11), h(13,9),t(13,12),
     s(14,13),h(15,9),t(15,11),h(17,9),t(17,9), h(19,9),t(19,18),h(20,9),
     t(20,19),h(22,9),t(22,21),h(23,9),t(23,22),h(24,9),t(24,1)}
RootD=[1]
expr=elem a {()}
oid=9
- : unit = ()
```

As a demonstration of the recovery from the failure of the partial evaluation described above, consider the following (Q14) and its partial evaluation sequence:

$$(\text{Q14}): \quad \text{let } \$v := \langle a \rangle()\langle /a \rangle \text{ return } (\$v, \langle b \rangle \$v \langle /b \rangle / \texttt{child} :: a) / \texttt{self} :: a$$

$$= \begin{array}{l} \text{peval (Q14) } \Theta \\ \quad \{(\text{PELET}); (\text{PEEC}); (\text{PESSTP})\} \\ \text{self_fusion } ((((\langle a \rangle()\langle /a \rangle)^{r_1}, (\langle a \rangle()\langle /a \rangle)^{r_2.1})^{[r_1, r_2.1]})\ a \end{array}$$

In this case, $\text{dc_sort}((((\langle a \rangle()\langle /a \rangle)^{r_1}, (\langle a \rangle()\langle /a \rangle)^{r_2.1})^{[r_1, r_2.1]})$ fails because r_1 and $r_2.1$ are incomparable. Our system recovers the input expression with the following global environment.

```
#############
RESULT peval
false
th=
Rel=
 oidD=9, cxtD=Nil, d=Undef,
     eD=let $u := elem a {()} return ($u,elem b {$u}/child::a)/self::a

Vtx=[1,2,3,4,5,6,7,8,9]
Edg={e(2,1),e(5,4),s(6,5),h(7,3),t(7,6),s(8,7),b(9,2),r(9,8)}
RootD=[]
expr=let $u := elem a {()} return ($u,elem b {$u}/child::a)/self::a
oid=9
- : unit = ()
```

5. RELATED WORK

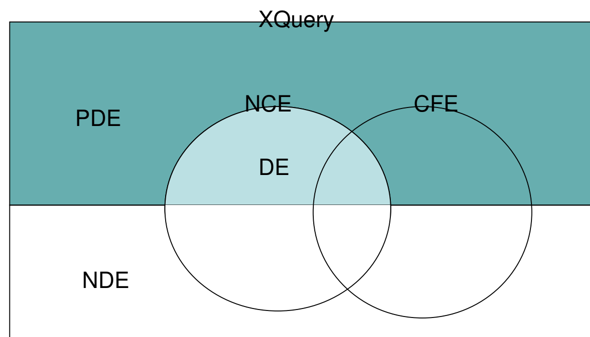
Studies on rewriting XQuery are summarized as two categories: one is rewriting it into XQuery[10, 18, 13, 19], another is rewriting it into SQL[9, 14, 4, 7, 1] or other languages[12, 8, 3]. For XQuery-to-XQuery transformations, the most related is [10] in a sense of eliminating redundant expressions. In [10], 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.

Koch[13] and Page et al.[18] introduced some classes for composite XQuery and proposed XQuery-to-XQuery transformations over the classes of XQuery they defined. In [13], Koch defined composition-free expressions (CFE), in which all variables only range over nodes in the input trees and never over nodes from intermediate query results. Page et al.[18] also defined the following class of XQuery.

- Node-Conservative Expressions (NCE)
  Expressions whose results do not contain newly constructed nodes.
- Deterministic Expressions (DE)
  Expressions which do not have element constructors as their subexpressions.
- Non-Deterministic Expressions (NDE)
  Expressions which have element constructors as their subexpressions.

They have proposed a rewriting method for node-conservative and deterministic expressions.

In real world, however, practical expressions such as schema mapping always returns newly constructed elements and such



NCE: Node-Conservative Expressions
CFE: Composition-Free Expressions (Koch, WebDB05)
DE: Deterministic Expressions (Page, et.al.,WebDB05)
PDE: Partially Deterministic Expressions
NDE: Non-Deterministic Expressions

Figure 11: Partially deterministic expressions (PDE)

queries are not in CFE nor NCE. Our target expressions defined in Figure 5 subsume NCE and CFE. Figure 11 shows a classification of XQuery expressions from our point of view. In Figure 11, surrounded by square denotes XQuery expressions defined in Figure 5, which are our target. For both DE and NDE, our fusion returns the input expressions intact. We define *partially deterministic expressions (PDE)*, for which our fusion does work to eliminate redundant expressions that include element constructors while preserving equivalence of expressions. This equivalence comes from "data exchange equivalence"[5]. The partiality comes from the property in which expressions that have subexpressions in PDE could be DE or NDE.

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 [19]. 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.

6. 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. The prototype system indicates that our framework is not only useful for real life applications including data integration system using XQuery such as schema mapping, but also important for defining novel class of XQuery in expressiveness, namely, partially deterministic expressions (PDE).

This work is still ongoing. In the future, we wish to prove soundness of our method by using algebraic structure shown in Appendix.

7. REFERENCES

[1] M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian. SilkRoute: A Framework for Publishing Relational Data in XML. In *VLDB*, pages 646–648, 2000.

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

[4] A. Deutsch and V. Tannen. Reformulation of XML Queries and Constraints. In *ICDT*, pages 225–241, 2003.

[5] R. Fagin, P. G. Kolaitis, A. Nash, and L. Popa. Towards a Theory of Schema-Mapping Optimization. In *PODS*, pages 33–42, 2008.

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

[7] M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents. *ACM TODS*, 27:438–493, 2002.

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

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

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

[11] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A Light but Formal Introduction to XQuery. In *XSym*, pages 5–20, 2004.

[12] H. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4):274–291, 2002.

[13] C. Koch. On the role of composition in XQuery. In *WebDB*, 2005.

[14] R. Krishnamurthy, P. Kaushik, and J. Naughton. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? In *VLDB*, pages 144–155, 2004.

[15] 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.

[16] M. Fernandez, J Simeon and P. Wadler. A Semi-monad for Semi-structured Data. In *ICDT*, pages 263–300, January 2001.

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

[18] W. L. Page, J. Hidders, P. Michiels, J. Paredaens, and R. Vercammen. On the expressive power of node construction in XQuery. In *WebDB*, 2005.

[19] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *SIGMOD*, 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 *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. XQery1.0 and XPath2.0

Formal Semantics. http://www.w3.org/TR/xquery-semantics, January 2007. W3C Recommendation.

APPENDIX

A. PROPERTIES ON EXTENDED DEWEY ORDER

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

We wrote in Section 4 that 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 $[d_1, d_2, \dots, d_n]$ by

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

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 $\oplus_D$, respectively.

DEFINITION A.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., $\forall y, z, w \in S$,*

$$y \precsim z \wedge z \precsim y \Rightarrow y \sim z \quad (1)$$
$$y \precsim z \wedge z \precsim w \Rightarrow y \precsim w \quad (2)$$
$$y \precsim z \vee z \precsim y \quad (3)$$

and

- *S is strictly monotonic, i.e.,*
  1. *$[]$ is a strictly monotonic*
  2. *for a strictly monotonic sequence ys, y:ys is also strictly monotonic iff.*
$$\forall y' \in ys (y \prec y').$$

PROPERTY A.2. *For a given distinctly ordered sequence y:ys, the following properties hold by* DEFINITION *A.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 A.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,*

$$\frac{y_1 \sim y_2}{(y_1 \oplus y_2) \sim y_1} \quad \text{(PRESO)}$$

holds.

Ordered insertion (one to many)($\sim\!\lhd^{\oplus}_{\precsim}$)

DEFINITION A.4 (Ordered insertion $\sim\!\lhd^{\oplus}_{\precsim}$). *Binary operator $\sim\!\lhd^{\oplus}_{\precsim}$ 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.*

$$\frac{|y| \to y'}{([] \sim\!\lhd^{\oplus}_{\precsim} y) \to [y']} \qquad \frac{z \sim y \quad (z \oplus y) \to v}{((z{:}zs) \sim\!\lhd^{\oplus}_{\precsim} y) \to v{:}zs}$$

$$\frac{y \prec z}{(z{:}zs) \sim\!\lhd^{\oplus}_{\precsim} y) \to (y{:}z{:}zs)} \qquad \frac{z \prec y \quad (zs \sim\!\lhd^{\oplus}_{\precsim} y) \to zs'}{((z{:}zs) \sim\!\lhd^{\oplus}_{\precsim} y) \to z{:}zs'}$$

THEOREM A.5 (Ordered insertion). *For any distinctly ordered sequence S under $\precsim$, $S \sim\!\lhd^{\oplus}_{\precsim} y$ is also distinctly ordered under $\precsim$ where $\oplus$ satisfies* (PRESO).

PROOF. Induction on the sequence S is used. □

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

DEFINITION A.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 $\sim\!\uplus^{\oplus}_{\precsim}$ is defined as the following inference rules.*

$$\frac{}{(zs \sim\!\uplus^{\oplus}_{\precsim} []) \to zs} \qquad \frac{(zs \sim\!\lhd^{\oplus}_{\precsim} y) \to zs' \quad zs' \sim\!\uplus^{\oplus}_{\precsim} ys \to vs}{zs \sim\!\uplus^{\oplus}_{\precsim} (y{:}ys) \to vs}$$

THEOREM A.7 (Ordered union). *For any distinctly ordered sequence S_1 under $\precsim$, $(S_1 \sim\!\uplus^{\oplus}_{\precsim} S_2)$ is also distinctly ordered under $\precsim$ where $\oplus$ satisfies* (PRESO).

PROOF. Induction on the sequence S_2 is used. □

DEFINITION A.8 (Strict Partial Order on $\mathcal{D}(\prec)$).

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

$$\frac{(x \neq \epsilon) \vee (x \neq .?\, x_1)}{\epsilon \prec_X x} \qquad \frac{b_1 \prec_B b_2}{.b_1 \prec_X .b_2}$$

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

unifiable($\sim$)

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

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

$$\frac{}{\epsilon \sim_X \epsilon} \qquad \frac{b_1 \sim_B b_2}{.b_1 \sim_X .b_2} \qquad \frac{\mathsf{orderable}_{bs}\ bs_1 +\!\!+ bs_2}{\#bs_1 \sim_X \#bs_2}$$

unify($\oplus$)

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

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

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

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

distinctable(dst)

$$\frac{\mathtt{dst}_X\ x_1}{\mathtt{dst}_D\ r_1 x_1}$$

$$\frac{}{\mathtt{dst}_X \epsilon} \qquad \frac{\mathtt{dst}_B\ b_1}{\mathtt{dst}_X\ .b_1} \qquad \frac{\mathtt{orderable}\ bs_1}{\mathtt{dst}_X\ \#bs_1}$$

$$\frac{\mathtt{dst} x_1}{\mathtt{dst}_B\ n_1 x_1}$$

distinct($||$)

$$\frac{|x_1|_X \to x}{|r_1 x_1|_D \to r_1 x}$$

$$\frac{}{|\epsilon|_X \to \epsilon} \qquad \frac{|b_1|_B \to b}{|.b_1|_X \to .b} \qquad \frac{[] \sim_B \uplus^{\oplus_B}_{\prec_B} bs_1 \to bs}{|\#bs_1|_X \to \#bs}$$

$$\frac{|x_1|_X \to x}{|n_1 x_1|_B \to n_1 x}$$

orderable

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

where

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

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