Math. Struct. in Comp. Science (2015), *vol.* 25, *pp.* 916–941. © Cambridge University Press 2014
doi:10.1017/S096012951300008X First published online 10 November 2014

Context-preserving XQuery fusion

H. KATO[†], S. HIDAKA[†], Z. HU[†], K. NAKANO[‡] and Y. ISHIHARA[§]

[†]*National Institute of Informatics, Japan*
[‡]*The University of Electro-Communications, Japan*
[§]*Osaka University, Japan*
Email: kato@nii.ac.jp

Received 7 May 2011; revised 3 February 2012

This paper solves the known problem of elimination of unnecessary internal element construction as well as variable elimination in XML processing with (a subset of) XQuery without ignoring the issues of document order. 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 through the system.

1. Introduction

Fusion (Chin 1992; Fegaras and Maier 2000; Wadler 1988) 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 (Daniels *et al.* 1991) and object query languages (Fegaras and Maier 2000), it remains as a challenge to implement fusion for XQuery optimization. This is because XQuery has more complicated semantics (Hidders *et al.* 2004); *it is context-sensitive and requires preservation of document order*. One may consider, for example, the following naive fusion transformation[†] (as studied in (Deutsch *et al.* 2004)).

$$<e>\{E_1, \ldots, E_n\}</e>/c \mapsto \sigma_c(E_1), \ldots, \sigma_c(E_n). \tag{F}$$

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 (Amano *et al.* 2009; Fernández *et al.* 2005), and various index structures for ordered trees have been developed for XML documents (Lu *et al.* 2005; Tatarinov *et al.* 2002; Xu *et al.* 2009). 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.

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

```
<na>
  <lhs>
    <item><a/></item>
    <item><b/></item>
  </lhs>
  <rhs>
    <item><c/></item>
    <item><d/></item>
  </rhs>
</na>
```

```
<sa>
   {(<lhs>{/na/rhs/item}</lhs>,
     <rhs>{/na/lhs/item}</rhs>)}
</sa>
```

```
<sa>
  <lhs>
    <item><c/></item>
    <item><d/></item>
  </lhs>
  <rhs>
    <item><a/></item>
    <item><b/></item>
  </rhs>
</sa>
```

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

Consider the simple case illustrated in Figure 1, where the query **Qm** (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 **Q1** to the serialized T,

Q1. let $v := (/sa/rhs, /sa/lhs) **return** $v/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:

```
<item><c/></item>,
<item><d/></item>,
<item><a/></item>,
<item><b/></item>.
```

On the other hand, consider the composite query of **Qm** and **Q1**, that is a 'let'-expression, in which first a variable is bound to the result of **Qm** then the path expression referring to the root element in **Q1** in 'return'-clause is replaced by the variable. We write this composite query as (**Qm**; **Q1**), and the following expression is obtained:

let $t := < sa >{(<lhs>{/na/rhs/item}</lhs>, <rhs>{/na/lhs/item}</rhs>)}< /sa >
return let $v := ($t/rhs, $t/lhs) **return** $v/item.

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

```
   Qm; Q1
→    {(variable elimination for $t); (F)}
   let $v := (<rhs>{/na/lhs/item}</rhs>, <lhs>{/na/rhs/item}</lhs>)
   return $v/item
→    {(variable elimination for $v); (F)}
   (/na/lhs/item, /na/rhs/item)
```

[†] In this rewriting, the variable $v is eliminated by replacing it with its value. This elimination does not work correctly either in XQuery processing. This problem will be described later.

then evaluating the transformed query (/na/lhs/item, /na/rhs/item) on S gives

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

whose order of 'item' elements is different from the previous expected result. Furthermore, if we consider the query **Q2** on T:

Q2. let $\$v$:= /sa/rhs/item **return** $\$v$/..

then, although the expected result of **Q2** to T is the 'rhs' element, the result of the transformed query from (**Qm**; **Q2**) via similar steps above is the 'lhs' element. In both examples, due to the disregard of the context, a tree navigation over the newly constructed XML fragment using <sa>{...}</sa> in **Qm** is ignored.

The problem of the existing fusion transformation lies in that the naive elimination of internal 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 expressions /na/rhs/item and /na/lhs/item in **Qm** 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 expressions, for given (**Qm**; **Q1**), we can formulate the correct XQuery expression (/na/rhs/item, /na/lhs/item) from this information, which is captured as the context in our fusion.

In this paper, we propose a novel context-preserving XQuery fusion for when an XML document is modelled as an ordered tree. 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. Dewey order encoding of XML nodes is a lossless representation of a position in the document order and it has been used for index structure of XML documents (Lu *et al.* 2005; Tatarinov *et al.* 2002). We use extended Dewey codes as the context representation of XQuery expressions. We extend the Dewey code to be suitable for the context of XQuery expressions, especially for 'for'-expressions and sequence expressions.

Our twofold main contributions can be summarized as follows. First, to keep track of context, we carefully design the context representation (extended Dewey code and its order) of XQuery expressions to reflect the properties of element constructions. This enables us to statically emulate newly created XML fragments – created by element constructors – in the XML store. Second, 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. This keeps 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 by using Dewey code to represent nodes in XML fragments and introduces value-equivalent expressions to show our fusion concisely. In Section 3, to design the context of XQuery expressions by extending Dewey code and its order to suite the semantics of XQuery expressions, we establish the correspondence between dynamic operations on XML store and static property of XQuery expressions. Section 4 presents the algorithm of context-preserving fusion using the extended Dewey code and its order. Also, the correctness of the algorithm is shown. Section 5 describes one typical application, which is XML data integration, and its experimental results. We discuss related work in Section 6 and conclude the paper in Section 7.

2. XQuery semantics

To give our XQuery fusion concisely and show that it is semantics-preserving, we briefly review the semantics of the core part of XQuery that is based on (Hidders *et al.* 2004). Our target XQuery expressions, a subset of XQuery, are as follows:

$$\begin{aligned} e ::= {}& \$v \mid (e, e, \ldots, e) \mid () \mid e/\alpha::\tau \mid \textbf{for}\ \$v\ \textbf{in}\ e\ \textbf{return}\ e \\ \mid {}& \textbf{let}\ \$v\ := e\ \textbf{return}\ e \mid \text{<}t\text{>}\{e\}\text{</}t\text{>}. \end{aligned}$$

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 any of all 13 XPath axes, which can be child, self, parent (..) and so on , 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 $\text{<}t\text{>}\{e\}\text{</}t\text{>}$. Since we focus on newly constructed trees that consist of XML nodes, to simplify the presentation, we use 'empty-element tags' like <c/> to represent constant c. Although constants themselves are not nodes, they become a (text) node when they occur in an element constructor. For example, a constant 'b' is not a node i.e. this constant does not populate any ordered trees. On the other hand, consider <a>{'b'}</a>. In this expression, the constant 'b' is a text node because the constant occurs in the element construction of <a>{(...)}</a>, i.e. this constant is a child node of the element node of a. We could define the semantics of constants with such behaviour, but this would make our presentation unnecessarily complex.

While location step expressions can be translated into 'for'-expressions (World Wide Web Consortium 2010b), our target includes location step expressions because of the following three reasons: (1) path expressions have better chances to exploit efficient evaluation

[†] For simplifying our presentation, we impose this syntactic restriction. Since this restriction is not essential, our algorithm can be extended straightforwardly to treat nested sequence expressions. Our prototype system can deal with nested sequence expressions.

algorithm through alternative semantics (Gottlob *et al.* 2005; Parys 2009) which would have been impossible if these expressions were translated into theoretically equivalent 'for'-expressions; (2) since the feature of our fusion is based on a partial evaluation of location steps, if location steps are translated into 'for'-expressions, we have to prepare a specific rule for such 'for' -expressions translated from location steps in addition to one for other 'proper' 'for'-expressions; and (3) previous work on XQuery dealt with location steps (Grust *et al.* 2004; Grust *et al.* 2010; Koch 2005).

2.1. *Sequence: data model in XQuery*

The data model of XQuery is *sequences* (World Wide Web Consortium 2010a). 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 $s_1 \circ s_2$. We use $\in$ for sequence membership in addition to set membership, $(d|d \in D \wedge \phi(d))$ for a sequence of d obtained by selecting them from D such that all items satisfy $\phi(d)$, and $|s|$ for the length of a sequence s.

2.2. *Dewey order encoding and XML store*

An XML document is modelled 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 (Hidders *et al.* 2004).

2.2.1. *Dewey code and axis relation.*

Dewey order encoding of XML nodes is a lossless representation of a position in the document order (Lu *et al.* 2005; Tatarinov *et al.* 2002).

Definition 2.1 (Dewey code). In Dewey order, each node is represented by a Dewey code which is defined as follows:

$$\begin{aligned} \mathtt{d} &::= n\ \mathtt{x} \\ \mathtt{x} &::= \epsilon \mid .\ \mathtt{d}. \end{aligned}$$

Where $n \in (\mathcal{R} \cup \mathcal{I})$ with $\mathcal{R}$ being a set of special codes and $\mathcal{I}$ being the set of integers.

A Dewey code is 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 nth child of a node b in the document order; 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 non-determinism in XQuery. Therefore, if two Dewey codes begin with different codes which are in $\mathcal{R}$, it implies that the two nodes are in different ordered trees. The special code represents the root of the ordered tree because we cannot define order among roots. By using Dewey order encoding, one can easily compute axis relations in the document order. For example, **ancestor**(d_1, d_2) holds when d_1 has the form $d_2.n_1.n_2.\cdots.n_k$. Therefore, Dewey codes have been used as index structure for XML documents (Lu *et al.* 2005; Tatarinov *et al.* 2002).

2.2.2. *Simple XML store.* 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 2.2 (simple XML store). A simple XML store is a pair $St = (D, v)$, where (a) D is a finite subset of $\mathcal{D}$ and (b) v is a total function $v : D \rightarrow \mathcal{T}$ that maps a Dewey code to its element name.

For instance, the store of the source S in Figure 1 is defined as $St_0 = (D_0, v_0)$, 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 $v_0(s) = \texttt{na}$, $v_0(s.1) = \texttt{lhs}$, $v_0(s.2) = \texttt{rhs}$, $v_0(s.1.1) = v_0(s.1.2) = v_0(s.2.1) = v_0(s.2.2) = \texttt{item}$, $v_0(s.1.1.1) = \texttt{a}$, $v_0(s.1.2.1) = \texttt{b}$, $v_0(s.2.1.1) = \texttt{c}$, $v_0(s.2.2.1) = \texttt{d}$. In what follows, we will refer to a simple XML store as an XML store.

Definition 2.3 (disjoint union of stores). Two stores $St_1 = (D_1, v_1)$ and $St_2 = (D_2, v_2)$ are said to be disjoint when $D_1 \cap D_2 = \emptyset$. For two disjoint stores St_1 and St_2, the disjoint union of the two stores, denoted as $St_1 \cup St_2$, is defined as $St_1 \cup St_2 = (D_1 \cup D_2, v')$ where $v' : (D_1 \cup D_2) \rightarrow \mathcal{T}$ with $v'(d_1) = v_1(d_1)$ when $d_1 \in D_1$ and $v'(d_2) = v_2(d_2)$ when $d_2 \in D_2$.

Definition 2.4 (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 $h : D_1 \rightarrow D_2$ with $D_1 = \{d | d \in D_{St_1} \wedge \textbf{ancestor-or-self}(d, d_1)\}$ and $D_2 = \{d | d \in D_{St_2} \wedge \textbf{ancestor-or-self}(d, d_2)\}$, such that for each d and $d' \in D_1$, it holds that (1) $h(d) \in D_2$, (2) $v(d) = v(h(d))$, 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*

Figure 2 shows the semantics of our target XQuery using a set of inference rules based on (Hidders *et al.* 2004). 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 (**for** $\$v$ **in** e_1 **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 ($<t>\{e\}</t>$) and location step ($e/\alpha :: \tau$) are worth further attention because they are evaluated using the document order. The semantics of

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

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

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

$$\frac{\begin{array}{c} St; En \vdash e \Rightarrow (St_1, s_1) \qquad \text{a fresh } r \in \mathcal{R} \\ \forall d (d \in D_{St_2} \rightarrow \textbf{descendant-or-self}(d, r)) \qquad \nu_{St_2}(r) = t \\ \textbf{ddo}_{St_2}(d' | d' \in D_{St_2} \wedge \textbf{child}(d', r)) = s_2 \qquad s_1 \equiv_{(St_1, St_2)} s_2 \end{array}}{St; En \vdash \texttt{<}t\texttt{>}\{e\}\texttt{</}t\texttt{>} \Rightarrow (St \cup St_2, r)}$$

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

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

$<t>\{e\}</t>$ 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 created root node is returned. We use '$\rightarrow$' for logical implication. Note that an element constructor has a side effect in a sense that it creates a new ordered tree. 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 **ddo** (distinct-doc-order), which is implemented easily when its domain is Dewey codes (Tatarinov *et al.* 2002).

While document order plays an important role in XQuery semantics, the serialized result of a query expression is not associated with the document order which is in the store used in the XQuery processing. For example, assuming identical bindings of the externally defined variable $\$v$, the serialized result of $<t>\{(\$v/c, \$v/a)\}</t>/c$ cannot be distinguished from one of $\$v/c$. This enables us to introduce *value-equivalent expressions*, which will be used to prove the correctness of our fusion later.

Definition 2.5 (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 $s_1 \equiv_{(St_1,St_2)} s_2$.

2.4. *The problem of context insensitiveness*

As described in the formal semantics of XQuery, an element constructor has a side effect on the document order of nodes: it constructs a new ordered tree. This side effect is a barrier to query optimization based on two standard query rewriting techniques – fusion and variable elimination – because these standard techniques are developed for side effect free languages. These techniques cannot handle a location step expression being applied to an element constructor to extract contents of the newly constructed ordered tree, since the axis access used in the location step expression requires sorting in the document order without duplicates. As will be seen in Section 5, such an expression that a location step expression is applied to an element constructor is often used in data integration, which is a typical application using XML documents and XQuery expressions.

An example of the problem of fusion based on eliminating unnecessary internal element constructors is described in the introduction. Here, a simple example of the variable elimination in XQuery is shown.

Example 2.1 (variable elimination). Consider the following expression,

$$\textbf{let}\ \$v\ := \ <a/>\ \textbf{return}\ (\$v, \$v)/\,\textbf{self}\ ::a.$$

This expression first constructs a new '*a*' element with an empty content. The expression in the return clause performs sorting without duplicates in the document order over two occurrences of the variable $\$v$. Since the two occurrences of the variable $\$v$ have the same identity, the result of the evaluation of this expression is $<a/>$. On the other hand, when variable elimination for $\$v$ is applied to this expression, we get $(<a/>,<a/>)/\,\textbf{self}\ ::a$. Now, since two '*a*' elements have distinct identities, the result of this expression is $(<a/>,<a/>)$.

The problem solved in this paper is that both the existing fusion for eliminating internal element constructors and standard variable elimination cannot convey their contexts but just convey their values. The solution will be described in the next section.

Before moving to the next section, we demonstrate that this complicated but practical semantics of XQuery may have a wrong rewriting rule. For example, Fegaras states in (Fegaras 2010) that many XQuery optimizers use the following rule.

$$(<A>\{e\}</A>)/\,\textbf{child}\ ::B = e/\,\textbf{self}\ ::B.$$

However, this rule is not correct when the following expression is considered

$$\textbf{let}\ \$v\ := \ <B/>\ \textbf{return}\ <A>\{(\$v, \$v)\}</A>/\,\textbf{child}\ ::B.$$

This expression first constructs a new 'B' element with an empty content. The expression in the return clause constructs a new 'A' element containing two copies of the 'B' element – that is, the element bound to the variable $\$v$ and the two elements contained in the 'A' element have distinct identities. Therefore, the result of the evaluation of this expression is ($<B/>, <B/>$). However, by applying the above rule, we get the following expression:

$$\textbf{let } \$v := <B/> \textbf{ return } (\$v, \$v)/\,\textbf{self}::B.$$

As described in Example 2.1, the result of the evaluation of this expression is $<B/>$.

3. Emulating XML stores with extended Dewey codes

Since element constructors have side effects, eliminating internal element constructors is a known difficult problem (Brundage 2004). 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 XQuery expression e, its result s has a common shape according to the structure of the expression. We denote this relation by $e : s$ if there exist St, En, St' such that $St\,; En \vdash e \Rightarrow (St', s)$.

Property 3.1 (Dewey code correspondence in element construction). For an element construction, $<t>\{e\}</t>$, the semantics of element constructors implies that e is an expression that is value equivalent to $<t>\{e\}</t>/\,\textbf{child}::*$, and the following properties hold from the XQuery semantics.

(i) If $<t>\{e\}</t> : r$ then $r \in \mathcal{R}$ and r is not in the input store. Note that this r is a single item, which cannot be distinguished from the singleton sequence, (r), in the data model of XQuery as described in Section 2.

(ii) If $<t>\{e\}</t> : r$ and $(<t>\{e\}</t>/\,\textbf{child}::*) : s$ then $d \in s$ implies $d = r.n$ for $n \in \mathcal{I}$.

(iii) If $(<t>\{(e_1, \ldots, e_N)\}</t>/\,\textbf{child}::*) : (s_1 \circ \ldots \circ s_N)$ then $d_i \in s_i$ and $d_j \in s_j$ for $1 \leqslant i < j \leqslant N$ imply $d_i < d_j$.

(iv) If $(<t>\{(e_1, \ldots, e_N)\}</t>/\,\textbf{child}::*) : s$ then there is a unique decomposition $(s_1 \circ \ldots \circ s_N)$ of s such that for all i $(1 \leqslant i \leqslant N)$, s_i is value equivalent to the result of the expression of e_i.

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. We will give the property of this code for XQuery expressions in Property 3.2. Now we introduce annotated XQuery expressions.

Definition 3.1 (annotated XQuery expressions). For an XQuery expression e, annotated XQuery expressions are defined as follows:

$$\begin{aligned} e^d ::= \$v^d \mid (e^d, e^d, \ldots, e^d)^d \mid (e^d/\alpha::\tau)^d \mid (\textbf{for}\ \$v\ \textbf{in}\ e^d\ \textbf{return}\ e^d)^d \\ \mid\ (\textbf{let}\ \$v\ :=\ e^d\ \textbf{return}\ e^d)^d \mid (\text{<}t\text{>}\{e^d\}\text{</}t\text{>})^d. \end{aligned}$$

Where d is a context information, to establish such association between expressions and their context information based on Dewey codes.

For instance, for the element construction $\text{<}t\text{>}\{(\$v/c, \$v/a)\}\text{</}t\text{>}$, we may give the following Dewey order encoding to the expression:

$$(\text{<}t\text{>}\{(\$v/c)^{r.1}, (\$v/a)^{r.2}\}\text{</}t\text{>})^r$$

where e^d denotes that d is the Dewey order encoding of the expression e.

One difficulty, however, remains in associating Dewey codes to expressions to keep the context information: how do we deal with the 'for'or ('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.*

Definition 3.2 (extended Dewey code). In extended Dewey order encoding of XQuery expressions, each expression is annotated with an extended Dewey code which is defined as follows:

$$\begin{aligned} \mathtt{d} &::= n\ \mathtt{x} \mid \underline{\epsilon} \mid \underline{(\mathtt{d}, \mathtt{d}, \ldots, \mathtt{d})} \\ \mathtt{x} &::= \epsilon \mid .\ \mathtt{d} \mid \underline{\#\ \mathtt{d}} \end{aligned}$$

where $n \in (\mathcal{R} \cup \mathcal{I})$ with $\mathcal{R}$ being a set of special codes which are used for topmost element constructors, and $\mathcal{I}$ being the set of integers.

The extended Dewey code has a hierarchical structure, the same as in XQuery expressions, because it is an annotation for an XQuery expression. Here, the underlined parts are our extension, and ϵ is used for a termination, so, every extended Dewey code ends with ϵ. Intuitively, the form of this code is as follows. ϵ is annotated to an expression, which does not occur inside an element constructor. For a sequence construction, the form of sequence[†] 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 sorting with duplicate elimination for multiple 'for' or 'let' expressions that are derived from identical 'for' or 'let' expressions.

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

Q3. $<a>\{\textbf{for}\ \$u\ \textbf{in}\ e\ \textbf{return}\ (\$u/c, \$u/d)\}</a>$.

To show the idea behind the design of our delimiter '#', let us consider the fusion transformation for the expression $(\mathbf{Q3}/d, \mathbf{Q3}/c)/\,\textbf{self} :: *$. For the expressions $\mathbf{Q3}/d$ and $\mathbf{Q3}/c$, we can get the *value-equivalent* expressions **Q4** and **Q5**, respectively, from the XQuery semantics.

Q4. $\textbf{for}\ \$u\ \textbf{in}\ e\ \textbf{return}\ \$u/d$.

Q5. $\textbf{for}\ \$u\ \textbf{in}\ e\ \textbf{return}\ \$u/c$.

Now consider the following expression **Q6**.

Q6. $((\mathbf{Q4}), (\mathbf{Q5}))/\,\textbf{self} :: *$.

As described in the previous section, since axis access by '/' requires sorting and duplicate elimination in document order, the correct transformation of **Q6** should result in **Q7**, in which two 'for' -expressions **Q4** and **Q5** are merged.

Q7. $\textbf{for}\ \$u\ \textbf{in}\ e\ \textbf{return}\ (\$u/c, \$u/d)$.

Here, we can capture the order of the two expressions in the 'return' expressions by using '#'. Thus, by encoding **Q3** into

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

and encoding **Q4** and **Q5** 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 **Q7** (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 d in a way similar 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 $r.1\#(1,2)$.

Now, extended Dewey order encoding of XQuery expressions has to have the following property to exploit both of Property 3.1 and the above discussion.

Property 3.2 (extended Dewey order encodeing of element constructors). For an element constructor $<t>\{e\}</t>$, the annotated element constructor is $<t>\{e^{d_2}\}</t>^{d_1}$ such that d_1 is a new Dewey code and the axis relation $\textbf{child}(d_2, d_1)$ holds from Properties 3.1(i) and (ii), respectively. Furthermore, the following properties hold when the content expression is a sequence expression or a 'for'-expression.

(i) When e is a sequence expression $(e_1, \ldots, e_N)$, the annotated expression is

$$(e_1^{d_1}, \ldots, e_N^{d_N})^{(d_1,\ldots,d_N)}$$

such that for $1 \leqslant i < j \leqslant N$, $d_i < d_j$ holds from Properties 3.1(iii) and (iv).

(ii) When e is a 'for'-expression **for** $\$v$ **in** e_b **return** $(e_1, \ldots, e_N)$, the annotated expression is

$$\textbf{for}\ \$v\ \textbf{in}\ e_b\ \textbf{return}\ (e_1^{d_1}, \ldots, e_N^{d_N})^{r.1\#(d_1,\ldots,d_N)}$$

such that for $1 \leqslant i < j \leqslant N$, $d_i < d_j$ holds from the above discussion.

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. 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

$$getExpGlobal(d)$$

that can be used to extract the set of the expressions whose codes are d from the global environment.

4.1. *Fusion transformation*

Figure 3 summarizes our fusion transformation on XQuery expressions. The fusion transformation is defined by a partial evaluation function peval:

$$\mathsf{peval} \quad :: \quad e \to \Theta \to e^d$$

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

$$\Theta :: Var \to (e^d, \textbf{let} \mid \textbf{for})$$

and produces an 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 normally, we can ignore all the annotations and get a normal XQuery expression as the final result. Otherwise, we end fusion by returning the input expression.

The definition of peval in Figure 3 is straightforward. For a variable, if it is bound 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 group them into a new sequence annotated with Dewey codes from the results of each element expression. Note

$$\mathsf{peval}\ ()\ \ \Theta = ()^{()}$$

$$\mathsf{peval}\ \$v\ \ \Theta = \begin{cases} e & \text{if } \Theta(\$v) = (e, \mathbf{let}) \\ \$v & \text{otherwise} \end{cases} \qquad (\text{PEVR})$$

$$\begin{aligned} \mathsf{peval}\ (e_1, ..., e_N)\ \ \Theta = \ & \underline{\text{for each}}\ i \in [1, N] & (\text{PESEQ}) \\ & \underline{\text{let}}\ e_{i'}^{d_i} = \mathsf{peval}\ \ e_i\ \ \Theta \\ & \underline{\text{in}}\ \mathsf{flatten}\ \ ((e_{1'}, ..., e_{N'})^{(d_1,..,d_N)}) \end{aligned}$$

$$\mathsf{peval}\ \ (e/\alpha :: \tau)\ \ \Theta = \mathsf{axis\_fusion}\,(\mathsf{peval}\ \ e\ \ \Theta)\ \alpha\ \tau \qquad (\text{PESTP})$$

$$\begin{aligned} \mathsf{peval}\ \ (\mathbf{let}\ \$v := e_1\ \mathbf{return}\ e_2)\ \ \Theta = \ & \underline{\text{let}}\ \ e_{1'} = \mathsf{peval}\ \ e_1\ \ \Theta \\ & \quad e_{2'} = \mathsf{peval}\ \ e_2\ \ (\Theta \cup \{\$v \mapsto (e_{1'}, \mathbf{let})\}) \\ & \underline{\text{in}}\ e_{2'} & (\text{PELET}) \end{aligned}$$

$$\begin{aligned} \mathsf{peval}\ \ (\mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2)\ \ \Theta = \ & \underline{\text{let}}\ \ e_{1'} = \mathsf{peval}\ \ e_1\ \ \Theta \\ & \quad e_{2'}^{d} = \mathsf{peval}\ \ e_2\ \ (\Theta \cup \{\$v \mapsto (e_{1'}, \mathbf{for})\}) \\ & \underline{\text{in}}\ (\mathbf{for}\ \$v\ \mathbf{in}\ e_{1'}\ \mathbf{return}\ e_{2'})^{\#d} & (\text{PEFOR}) \end{aligned}$$

$$\begin{aligned} \mathsf{peval}\ \ (<\!t\!>\!\{e_1\}\!<\!/t\!>)\ \ \Theta = \ & \underline{\text{let}}\ \ e_{1'} = \mathsf{peval}\ \ e_1\ \ \Theta & (\text{PEEC}) \\ & \quad \text{a fresh } r \in \mathcal{R} \\ & \underline{\text{in}}\ \mathsf{dc\_assign}\ <\!t\!>\!\{e_{1'}\}\!<\!/t\!>\ r \end{aligned}$$

Fig. 3. Fusion by partial evaluation.

that we use flatten to remove nested sequences (e.g. $\mathsf{flatten}((e_{11}^{d_1}, e_{12}^{d_2})^{(d_1,d_2)}, e_3^{d_3})^{((d_1,d_2),d_3)} = (e_{11}^{d_1}, e_{12}^{d_2}, e_3^{d_3})^{(d_1,d_2,d_3)}$). 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 important fusion functions axis_fusion, 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 (**Qm**; **Q2**) in the introduction.

4.1.1. *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 4 defines a function dc_assign $e^-\ d$:

$$\mathsf{dc\_assign} :: e^d \to d \to e^d$$

which is to propagate the Dewey code d 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 'do not care'.

$$
\begin{aligned}
\mathsf{dc\_assign}\;()^{\text{-}}\;d &= ()^{()} \\
\mathsf{dc\_assign}\;\$v^{\text{-}}\;d &= \$v^{d} \\
\mathsf{dc\_assign}\;(e/c)^{\text{-}}\;d &= (e/c)^{d} && (\textsc{dcpStp}) \\
\mathsf{dc\_assign}\;(e_1,\ldots,e_N)^{\text{-}}\;d &= \underline{\text{let}}\;\; d_1 = d && (\textsc{dcpSeq}) \\
&\qquad\quad e_{i'} = \mathsf{dc\_assign}\;e_i\;d_i \\
&\qquad\quad d_{i+1} = \mathsf{succ}(d_i) \\
&\quad\;\; \underline{\text{in}}\;(e_{1'},\ldots,e_{N'})^{(d_1,\ldots,d_N)} \\
\mathsf{dc\_assign}\;(\texttt{<}t\texttt{>}\{e\}\texttt{</}t\texttt{>})^{\text{-}}\;d &= \underline{\text{let}}\;e^{d_1} = \mathsf{dc\_assign}\;e\;d.1 && (\textsc{dcpEc}) \\
&\quad\;\; \underline{\text{in}}\;\texttt{<}t\texttt{>}\{e^{d_1}\}\texttt{</}t\texttt{>}^{d} \\
\mathsf{dc\_assign}\;(\textbf{for}\;\$v\;\textbf{in}\;e_0\;\textbf{return}\;e)^{\text{-}}\;d &= \underline{\text{let}}\;\; e^{d_1} = \mathsf{dc\_assign}\;e\;1 && (\textsc{dcpFor}) \\
&\quad\;\; \underline{\text{in}}\;(\textbf{for}\;\$v\;\textbf{in}\;e_0\;\textbf{return}\;e)^{d\#d_1}
\end{aligned}
$$

Fig. 4. Dewey code propagation.

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 $d.1 = d.2$). (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 (DCPSTP) and (DCPFOR). In (DCPSTP), 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, 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. d_1 that reflects the horizontal structure produced by the return clause is combined with the # sign to produce $r\#d_1$ as the top level code allocated to the 'for'-expression.

Lemma 4.1. From the definition of dc_assign, which is invoked by peval with an element constructors, given an XQuery expression e, the extended Dewey code assigned by dc_assign e^{-} d satisfies Property 3.2.

Proof. We assume that d satisfies Property 3.2 for e when dc_assign e d is invoked. For an empty sequence, there has no Dewey codes correspondence. For a variable, from the definition of peval this variable is bound in a 'for'-clause and d satisfies these properties from the assumption. For a step expression, it also satisfies these properties from the assumption. For a sequence expression $(e_1, \ldots, e_N)$, for each $i \in [1, N]$, e_i has an extended Dewey code succ^{i-1} d as its context by (DCPSEQ) and these codes satisfy Property 3.2(i). For an element constructor $<t>\{e\}</t>$, dc_assign e $d.1$ is invoked by (DCPEC) and the assigned code satisfies Property 3.2. For a 'for'-expression, the extended Dewey code assigned by (DCPFOR) satisfies Property 3.2(ii). □

Definition 4.1 (correct context information). For a given annotated XQuery expression e^d, d is said to be correct context information when e^d is a result of dc_assign e^{-} d.

$$
\begin{aligned}
&\mathsf{axis\_fusion} :: e^d \to \alpha \to \tau \to e^d \\
&\mathsf{axis\_fusion}\ e^d\ \alpha\ \tau = \begin{cases} \mathsf{remove\_dup}\ (e'_1, \dots, e'_N) & \text{if } \mathsf{dc\_sort} \text{ succeeds} \\ \bot & \text{otherwise} \end{cases} \\
&\quad \text{where } (e'_1, \dots, e'_N) = \mathsf{dc\_sort}(\mathsf{filter}(\mathsf{equal\_to}\,\tau)(\mathsf{get\_axis}\ e^d\ \alpha)) \qquad \text{(CPFUSION)} \\
&\mathsf{get\_axis} :: e^d \to \alpha \to e^d \\
&\mathsf{get\_axis}\ e^d\ \alpha = (e_1^{d_i} | \alpha(d_i, d) \wedge e_1^{d_i} \in getExpGlobal(d_i))
\end{aligned}
$$

Fig. 5. Fusion rules for location step expressions.

4.1.2. *Fusion rules.* Our fusion transformation on $e/\alpha::\tau$ is based on a function axis_fusion. The definition of axis_fusion is given by fusion rules in Figure 5, where each fusion rule corresponds to an axis type. The basic procedure is as follows:

1. extract (get) subexpressions from the global environment according to the axis α by using get_axis function;
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 with $\bot$, which indicates the fusion has failed. The reason why we use $\bot$ in case dc_sort fails is that when dc_sort fails, the result expression should be the initial expression, not e^d because peval is called recursively.

More concretely, let us consider the definition of axis_fusion. We use get_axis e^d to get a sequence of expressions that contribute to producing the axis relation in the document order of the XML fragments that can be obtained by evaluating e, and use the filter (equal_to τ) function to keep those expressions, the result of which is a sequence of node n_i such that $v(n_i) = \tau$, 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. Whether dc_sort succeeds or not is detected by using the predicate given in Definition A.3. If the sorting succeeds, we return a sequence expression by removing all duplicated element subexpressions; otherwise, we end fusion by returning the original expression $e/\alpha :: \tau$.

It is worth remarking that we could merge two algorithms, peval and dc_assign, so that the combined algorithm could handle the fusion in a lazy way, since both of the algorithms have the same structure. However, developing this combined algorithm is in our future work.

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

Lemma 4.2. For a 'let'-expression **let** $\$v := e_1$ **return** e_2, when e_1 and $e_{1'}$ are value-equivalent expressions, then e_2 and $e_{2'}$ are value-equivalent expressions where $e_{2'}$ is constructed from e_2 with replacing the occurrence of $\$v$ by $e_{1'}$ ($e_{2'} = e_2[\$v/e_{1'}]$).

Proof. This is trivial from the semantics of 'let'-expressions. □

Theorem 4.1 (correctness of fusion). For an XQuery expression e, if $\mathsf{peval}\ \ e\ \ \Theta = e_1^d$ then e and e_1 are value-equivalent expressions.

Proof. It is sufficient to show the correctness when our fusion succeeds since when our fusion fails, the result is the input expression. The proof uses structural induction on the expressions. It is worth remarking that syntactical base case shows only an empty sequence expression, however, the meaningful case is in an 'empty-element tag' which represents a constant as described in Section 2.

Base case. For an empty sequence expression (), $\mathsf{peval}\ \ ()\ \ \Theta = ()^0$.

Induction step. For a sequence expression $(e_1, \ldots, e_N)$, we assume that e_i and $e_{i'}$ are value-equivalent expressions where $\mathsf{peval}\ \ e_i\ \ \Theta = e_{i'}^{d_i}$ for each $i \in [1, N]$. From the definition of peval for sequence expression, $(e_1, \ldots, e_N)$ and $(e_{1'} \ldots, e_{N'})$ are value-equivalent expressions.

For a 'let'-expression **let** $\$v\ := e_1$ **return** e_2, we assume that e_1 and $e_{1'}$ are value-equivalent expressions where $\mathsf{peval}\ \ e_1\ \ \Theta = e_{1'}^{d_1}$ with correct context information d_1. From Lemma 4.2, we can assume that e_2 and $e_{2'}$ are value-equivalent expressions where $\mathsf{peval}\ \ e_2\ \ (\Theta \cup \{\$v \mapsto (e_{1'}, \textbf{let})\}) = e_{2'}^{d_2}$ with correct context information d_2. From both of the semantics and peval for 'let'-expression, **let** $\$v\ := e_1$ **return** e_2 and $e_{2'}$ are value-equivalent expressions.

For a 'for'-expression **for** $\$v$ **in** e_1 **return** e_2, we assume that e_1 and $e_{1'}$ are value-equivalent expressions where $\mathsf{peval}\ \ e_1\ \ \Theta = e_{1'}^{d_1}$ with correct context information d_1. Also, assume that e_2 and $e_{2'}$ are value-equivalent expressions where $\mathsf{peval}\ \ e_2\ \ (\Theta \cup \{\$v \mapsto (e_{1'}, \textbf{for})\}) = e_{2'}^{d_2}$ with correct context information d_2. From both of the semantics and peval for 'for'-expression, **for** $\$v$ **in** e_1 **return** e_2 and **for** $\$v$ **in** $e_{1'}$ **return** $e_{2'}$ are value-equivalent expressions.

For a step expression $e/\alpha :: \tau$, we assume e and e_1 are value-equivalent expressions where $\mathsf{peval}\ \ e\ \ \Theta = e_1^d$, and d is a correct context information on e_1. To show the correctness for step expressions, we have to show $e_1^d/\alpha :: \tau$ and $e_2^{d'}$ are value-equivalent expressions with correct context information d and d' where $\mathsf{axis\_fusion}\ \ e_1^d\ \alpha\ \tau = e_2^{d'}$. This correctness is implied by the definition of $\mathsf{axis\_fusion}$ and Lemma A.3 in Appendix A together with the semantics of the location step expressions. Note that since *getExpGlobal*(d) used in $\mathsf{axis\_fusion}$ results in the expressions which is already processed by our fusion algorithm, this does not interfere with the structural argument.

For an element constructor $<t>\{e\}</t>$, we assume e and e_1 are value-equivalent expressions when $\mathsf{peval}\ \ e\ \ \Theta = e_1^{d'}$. From the semantics of element constructors, $<t>\{e\}</t>$ and $<t>\{e_1\}</t>$ are value-equivalent expressions where $\mathsf{peval}\ \ <t>\{e\}</t>\ \ \Theta = (<t>\{e_1^{d_1}\}</t>)^d$. Moreover, d and d_1 are a correct context information by the Lemma 4.1.

For a variable $\$v$, we assume the variable binding for $\$v$. When the variable $\$v$ is bound in a 'for'-clause, peval results in $\$v$, whereas when $\$v$ is bound in a 'let'-clause, peval eliminates the variable by replacing it with the expression, of which result binds to the variable in the 'let'-clause. In both cases, peval results in the value-equivalent expressions.

□

4.2. *Examples*

For (**Qm**; **Q1**) described in the introduction, our fusion function peval works as follows:

$$
\begin{array}{ll}
 & \mathsf{peval}\ \ (\mathbf{Qm};\mathbf{Q1})\ \ \{\} \\
\rightsquigarrow & \{(\textsc{peStp});(\textsc{peLet});(\textsc{peEc})\} \\
 & \mathbf{let}\ \$t\ :=\ <\mathsf{sa}>\{(\ \mathsf{<lhs>}\{\mathsf{/na/rhs/item}^{r.1.1}\}\mathsf{</lhs>}^{r.1}, \\
 & \qquad\qquad\qquad\qquad \mathsf{<rhs>}\{\mathsf{/na/lhs/item}^{r.2.1}\}\mathsf{</rhs>}^{r.2})^{(r.1,r.2)}\}<\mathsf{/sa}>^{r} \\
 & \mathbf{return}\ \ \mathbf{let}\ \$v\ :=\ (\$t\mathsf{/rhs}, \$t\mathsf{/lhs})\ \mathbf{return}\ \$v\mathsf{/item} \\
\rightsquigarrow & \{(\textsc{peLet});(\textsc{peSeq});(\textsc{peStp});(\textsc{peStp})\} \\
 & \mathbf{let}\ \$v\ :=\ (\ \mathsf{<rhs>}\{\mathsf{/na/lhs/item}^{r.2.1}\}\mathsf{</rhs>}^{r.2}, \\
 & \qquad\qquad \mathsf{<lhs>}\{\mathsf{/na/rhs/item}^{r.1.1}\}\mathsf{</lhs>}^{r.1})^{(r.2,r.1)} \\
 & \mathbf{return}\ \$v\mathsf{/item} \\
\rightsquigarrow & \{(\textsc{peStp})\} \\
 & \mathsf{remove\_dup}\ \ (\mathsf{dc\_sort}\ \ (\mathsf{/na/lhs/item}^{r.2.1}, \mathsf{/na/rhs/item}^{r.1.1})) \\
\rightarrow & \\
 & (\mathsf{/na/rhs/item}^{r.1.1}, \mathsf{/na/lhs/item}^{r.2.1}).
\end{array}
$$

Similarly, for (**Qm**; **Q2**), our fusion function peval works as follows:

$$
\begin{array}{ll}
 & \mathsf{peval}\ \ (\mathbf{Qm};\mathbf{Q2})\ \ \{\} \\
\rightsquigarrow & \{(\textsc{peLet});(\textsc{peSeq});(\textsc{peStp});(\textsc{peStp})\} \\
 & \mathbf{let}\ \$t\ :=\ <\mathsf{sa}>\{(\ \mathsf{<lhs>}\{\mathsf{/na/rhs/item}^{r.1.1}\}\mathsf{</lhs>}^{r.1}, \\
 & \qquad\qquad\qquad\qquad \mathsf{<rhs>}\{\mathsf{/na/lhs/item}^{r.2.1}\}\mathsf{</rhs>}^{r.2})^{(r.1,r.2)}\}<\mathsf{/sa}>^{r} \\
 & \mathbf{return}\ \ \mathbf{let}\ \$v\ :=\ \$t\mathsf{/rhs/item}\ \mathbf{return}\ \$v\mathsf{/..} \\
\rightsquigarrow & \{(\textsc{peLet});(\textsc{peStp});(\textsc{peStp});(\textsc{peVr})\} \\
 & \mathsf{/na/lhs/item}^{r.2.1}\mathsf{/..} \\
\rightsquigarrow & \{(\textsc{CPFusion})\} \\
 & \mathsf{<rhs>}\{\mathsf{/na/lhs/item}^{r.2.1}\}\mathsf{</rhs>}^{r.2}.
\end{array}
$$

Next, consider the following expression where the expression of the let-bound variable itself is a non-element composed expression such as a ‘for’-expression.

$$
\begin{array}{l}
\mathbf{let}\ \$u\ :=\ \mathbf{for}\ \$v\ \mathbf{in}\ \mathsf{/a/b}\ \mathbf{return}\ (\$v\mathsf{/c}, \$v\mathsf{/d}) \\
\mathbf{return}(\$u/\,\mathbf{self}\ \mathsf{::d}, \$u/\,\mathbf{self}\ \mathsf{::c}).
\end{array}
$$

Our algorithm results in the same expression as this expression since dc_sort fails for both expressions $\$u/\,\mathbf{self}\ \mathsf{::d}$ and $\$u/\,\mathbf{self}\ \mathsf{::c}$. This is because the ‘for’-expression binding to $\$u$ has ϵ as its context information.

5. Application

Integrating data coming from different sources is a very important task. Since XML is developed for data exchange format, an XML data integration system is one solution to the task. In this section, we will show how our XQuery fusion technique is useful in XML data integration systems and report our experimental results using synthetic queries and XML documents.

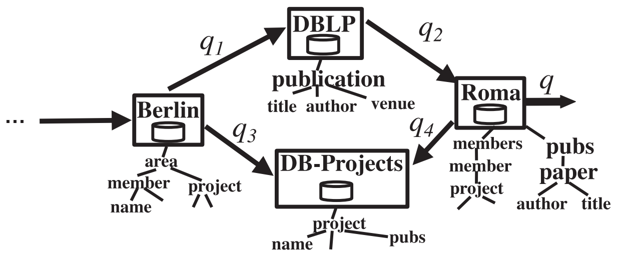

Fig. 6. An XML data integration system based on P2P approaches.

5.1. *XML data integration*

XML data integration systems offer an architecture for data sharing in which the data is queried through peers' schema (Tatarinov and Halevy 2004). Consider an XML data integration based on P2P approaches shown in Figure 6, which is adapted from (Tatarinov and Halevy 2004). In Figure 6, each rectangle shows a peer, which manages data under its own schema and each arrow denoted by q_i shows a schema mapping in XQuery. A direction of each arrow shows a mapping direction. For example, the schema mapping q_2 defines **Roma** as a query over **DBLP**.

```
<Roma>
  <pubs>
    {for $v in /DBLP/publication
    return <paper>{($v/author, $v/title)}</paper>}
  </pubs>
</Roma>
```

One can share data among multiple peers in the data integration setting. For example, a query q to a peer **Roma** can get data not only stored in **Roma** but also stored in **DBLP** by using q_2. Note that the person who writes the query q knows only schema information for **Roma**. She does not have to know all the schema information connected by schema mappings. A schema mapping bridges a gap of schema differences. A schema mapping in XQuery typically has element constructions, since it changes the schema information from one to another by using element constructions.

In XML data integration systems, a query to a peer is reformulated the query over its immediate neighbours by expanding the schema mapping. Such queries have redundant internal element constructors because schema mappings has element constructors. In this situation, the proposed technique based on the eliminating internal element constructors without ignoring the order issues is useful.

5.2. *Experimental results*

While actual evaluation times are predictable, for example from (Michiels *et al.* 2008), we have tested two kinds of queries **Q8(n)** and **Q9(n)** using two XQuery engines, `Galax`

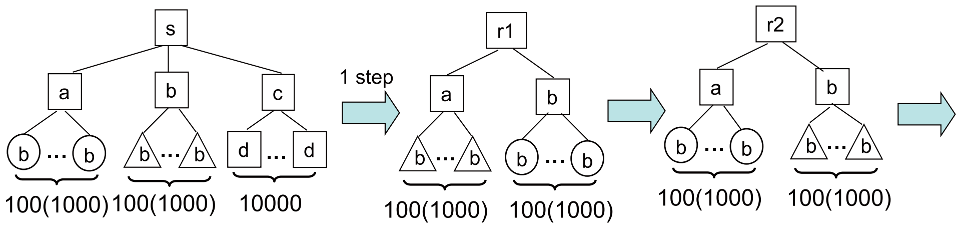

Fig. 7. Schema mappings in **Q8**.

version 1.0.1[†] and Saxon-HE version 9.4.0.2[‡] on 2.6GHz Intel Core2 Duo with 4GB RAM, running MacOS 10.5.6. The reason why we have chosen Galax and Saxon-HE as XQuery engines is that both engines closely track the definition of XQuery 1.0 as specified by the W3C. Both queries are extreme cases for document order and they are synthetic for XML data integration systems with *n* steps as schema mappings inspired by (Tatarinov and Halevy 2004).

The query **Q8(*n*)**, which is for a document 'd1.xml', is defined as follows:

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

In the 'd1.xml', the root node s has three child nodes a, b and c shown as the left-most tree in Figure 7. We prepared two documents, in which the number of b elements at level 3 under the a and b elements at level 2 (where the root is at level 1) is 100 (1000). In **Q8**, each step of schema mapping swaps b elements at level 3 under a element with ones under b element. This mapping is shown in Figure 7.

The query **Q9(*n*)**, which is for a document 'd2.xml', is defined as follows:

[†] http://galax.sourceforge.net/, default optimization option turned on.
[‡] http://saxon.sourceforge.net/, Java implementation.

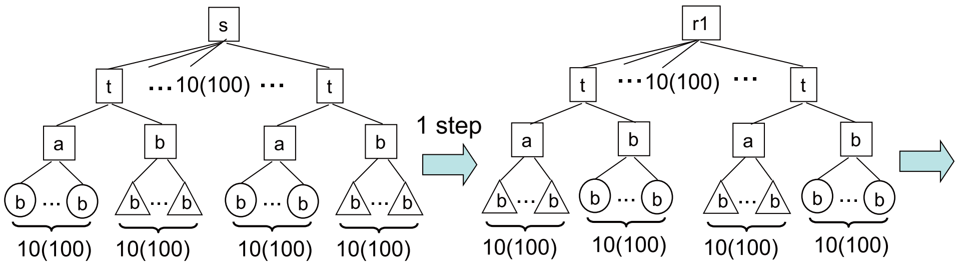

Fig. 8. Schema mappings in **Q9**.

```
let  $r1 := <r1>{for $t1 in doc('d2.xml')/s/t
                     return <t>{(<a>{$t1/b/b}</a>, <b>{$t1/a/b}</b>)}</t>}
              </r1>
return
let  $r2 := <r2>{for $t2 in $r1/t
                     return <t>{(<a>{$t2/b/b}</a>, <b>{$t2/a/b}</b>)}</t>}
              </r2>
return
...
let $rn := <rn>{for $tn in $r(n-1)/t
                    return <t>{(<a>{$tn/b/b}</a>, <b>{$tn/a/b}</b>)}</t>}
             </rn>
return
let $v := ($rn/t/b,$rn/t/a)
return $v/b.
```

For 'for'-expressions, we prepared the two documents 'd2.xml' shown in the left tree in Figure 8. In this document, the root node `s` has 10 (100) `t` elements, and each `t` element has two elements `a` and `b` at level 3. Under both of the `a` and `b` elements, there are 10 (100) `b` elements at level 4. In **Q9**, each step of schema mapping swaps `b` elements at level 4 under the `a` elements with ones under the `b` elements. This mapping is shown in Figure 8.

Figures 9 and 10 show the execution times for naive queries(`N`), optimized queries(`O`) and query rewriting costs plus optimized queries(`R+O`) for **Q8** and **Q9**, respectively. Since naive queries produce redundant intermediate results in proportional to the number of steps, the execution times are increasing with respect to steps. Whereas, since optimized queries rewritten by our prototype system always degenerate to queries to the extensional DB only, the execution time remain constant. For an even number of steps, our prototype system rewrites **Q8**(n) into the following optimized query:

(`doc`('d1.xml')/`s`/`a`/`b`, `doc`('d1.xml')/`s`/`b`/`b`, ()).

For an odd number of steps, our prototype system rewrites **Q9**(n) into the following optimized query:

for \$$t1$ **in** `doc`('d2.xml')/`s`/`t` **return** (\$$t1$/`b`/`b`, (\$$t1$/`a`/`b`, ())).

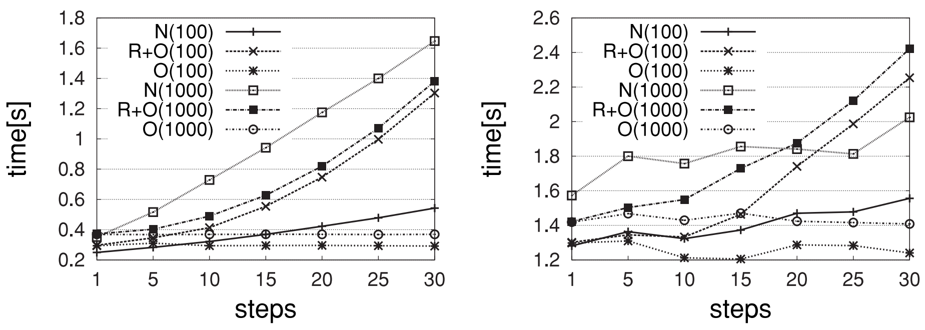

Fig. 9. Times for **Q8** using Galax(left) and Saxon-HE(right).

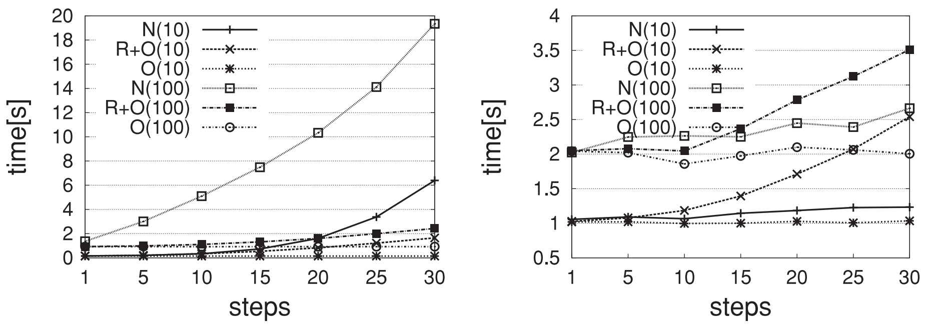

Fig. 10. Times for **Q9** using Galax(left) and Saxon-HE(right).

Both Figures 9 and 10 show that the rewriting costs are not neglectable when the steps are increased. However, in the data integration settings, we can assume that schema mappings do not change frequently because the schema on each peer does not change frequently. This assumption implies that since the time for optimization can be done statically at compile time, this time is not necessarily to be included. Most of the rewriting cost comes from the global environment that is kept in memory. Since the global environment is only used in solving reverse axis, it can be safely discarded when input queries include forward axis only. This optimization will be incorporated in the next version of our prototype system.

6. Related work

There are many studies on rewriting XQueries into other XQueries (Gueni *et al.* 2008; Koch 2005; Page *et al.* 2005; Tatarinov and Halevy 2004). The study most related to ours in the sense of eliminating redundant expressions is (Gueni *et al.* 2008). The authors of (Gueni *et al.* 2008) 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 (Koch 2005) and Page *et al.* (Page *et al.* 2005) introduced some classes for composite XQuery and proposed XQuery-to-XQuery transformations over the classes of XQuery they defined. Their target queries do not 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 (Tatarinov and Halevy 2004). In this system, the composition of the element construction is typical because the schema mapping that maps one element to another 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 (Chin 1992; Gill *et al.* 1993; Ohori and Sasano 2007; Wadler 1988). 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 (Ohori 1990), but our focus here is to perform XQuery-to-XQuery transformations, and the node identity is not a first class object[†] in XQuery.

7. Concluding remarks

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.pl.nii.ac.jp/fusion`.

The proposed technique is also useful in two directions. One is in 'unordered mode' in XQuery. Order of data is not important in many database applications. Our approach is also useful in these cases because the 'unordered mode' still requires eliminating duplicates on nodes. The other is in checking non-determinism on order of a result of an expression. By using our approach it becomes possible to detect that the order of user's intention may differ from the order of the result of an expression.

Future work includes extending the framework to graph structures through the use of ID/IDREF attributes, refining context information by using schema information, developing context-preserving common subexpression elimination, and more practical applications of the system for eliminating internal element constructions.

[†] While node identity is not 'first class' in XQuery, one can compare node identities explicitly with 'is' and nodes can be assigned an ID reference.

Acknowledgement

The authors thank anonymous reviewers for their thorough comments on earlier versions of the paper.

Appendix 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:

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

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

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

Definition A.1 (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 hold;

— $n_1, n_2 \in \mathcal{R}$ 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 A.1 (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 A.2 (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 $ds_1 \circ ds_2$ is *sortable.*

Definition A.3 (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 A.2 (irreflexivity of $\prec_d$). $\prec_d$ is irreflexive from its definition.

Theorem A.1 (reflexive partial order of $\precsim$). $\precsim$ is a reflexive partial order.

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

Definition A.4 (duplicate elimination and merging). Given two xD codes 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:

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

$$\frac{}{(\epsilon \oplus_x \epsilon) \to \epsilon} \qquad \frac{(d_1 \oplus_d d_2) \to d_3}{(.d_1 \oplus_x .d_2) \to .d_3} \qquad \frac{\mathbf{xDDO}(ds_1 \circ ds_2) \to ds_3}{(\#ds_1 \oplus_x \#ds_2) \to \#ds_3}$$

Definition A.5 (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 ($\mathbf{xDDO}\ ds_1$) is defined straightforwardly by using $\precsim$ and $\oplus_d$.

Lemma A.3. For a given sortable list of xD ds_1, the result of the sorting without duplicate on ds_1 ($\mathbf{xDDO}\ \ ds_1$) is strictly ordered under $\precsim$ from its definition.

References

Amano, S., Libkin, L. and Murlak, F. (2009) XML schema mappings. In: *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, ACM 33–42. http://doi.acm.org/10.1145/1559795.1559801.

Brundage, M. (2004) *XQuery: The XML Query Language*, Addison-Wesley.

Chin, W. (1992) Safe fusion of functional expressions. In: *Proceedings of the Conference on Lisp and Functional Programming*, San Francisco, California. ACM Press 11–20.

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

Deutsch, A., Papakonstantinou, Y. and Xu, Y. (2004) The NEXT framework for logical XQuery opimization. In: *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, Morgan Kaufmann. 168–179. http://www.vldb.org/conf/2004/RS4P5.PDF.

Fegaras, L. (2010) Propagating updates through XML views using lineage tracing. In: *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, IEEE Computer Society 309–320.

Fegaras, L. and Maier, D. (2000) Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems (TODS)* **25** (4) 457–516.

Fernández, M., Hidders, J., Michiels, P., Siméon, J. and Vercammen, R. (2005) Optimizing sorting and duplicate elimination in XQuery path expressions. In: *Proceedings of the 16th International Conference on Database and Expert Systems Applications (DEXA)*, Springer 554–563. http://dx.doi.org/10.1007/11546924_54.

Gill, A., Launchbury, J. and Jones, S. L. P. (1993) A short cut to deforestation. In: *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture (FPCA)*, ACM Press 223–232.

Gottlob, G., Koch, C. and Pichler, R. (2005) Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems (TODS)* **30** (2) 444–491.

Grust, T., Mayr, M. and Rittinger, J. (2010) Let SQL drive the XQuery workhorse (XQuery join graph isolation). In: *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, ACM Press 147–158. http://doi.acm.org/10.1145/1739041.1739062.

Grust, T., Sakr, S. and Teubner, J. (2004) XQuery on SQL hosts. In: *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, Morgan Kaufmann 252–263. http://www.vldb.org/conf/2004/RS7P1.PDF.

Gueni, B., Abdessalem, T., Cautis, B. and Waller, E. (2008). Pruning nested XQuery queries. In: *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM)*, ACM 541–550. http://doi.acm.org/10.1145/1458082.1458154.

Hidders, J., Paredaens, J., Vercammen, R. and Demeyer, S. (2004). A light but formal introduction to XQuery. In: Proceedings of the Second International XML Database Symposium (XSym). *Springer Lecture Notes in Computer Science* **3186** 5–20. http://dx.doi.org/10.1007/978-3-540-30081-6_2.

Koch, C. (2005) On the role of composition in XQuery. In: *Proceedings of the 8th International Workshop on the Web and Databases (WebDB)* 37–42.

Lu, J., Ling, T. W., Chan, C.-Y. and Chen, T. (2005) From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In: *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, ACM 193–204. http://www.vldb2005.org/program/paper/tue/p193-lu.pdf.

Michiels, P., Manolescu, I. and Miachon, C. (2008) Toward microbenchmarking XQuery. *Information Systems* **33** (2) 182–202. http://dx.doi.org/10.1016/j.is.2007.05.003.

Ohori, A. (1990) Representing object identity in a pure functional language. In: Proceedings of the 3rd International Conference on Database Theory (ICDT). *Springer-Verlag Lecture Notes in Computer Science* **470** 41–55. http://dx.doi.org/10.1007/3-540-53507-1_69.

Ohori, A. and Sasano, I. (2007) Lightweight fusion by fixed point promotion. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* 143–154. http://doi.acm.org/10.1145/1190216.1190241.

Page, W. L., Hidders, J., Michiels, P., Paredaens, J. and Vercammen, R. (2005) On the expressive power of node construction in XQuery. In: *Proceedings of the 8th International Workshop on the Web and Databases (WebDB)* 85–90.

Parys, P. (2009) XPath evaluation in linear time with polynomial combined complexity. In: *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART. Symposium on Principles of Database Systems (PODS)*, ACM 55–64. http://doi.acm.org/10.1145/1559795.1559805.

Tatarinov, I. and Halevy, A. (2004) Efficient query reformulation in peer data management systems. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD)* 539–550. http://doi.acm.org/10.1145/1007568.1007629.

Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E. and Zhang, C. (2002) Storing and querying ordered XML using a relational database system. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)* 204–215. http://doi.acm.org/10.1145/564691.564715.

Wadler, P. (1988) Deforestation: Transforming programs to eliminate trees. In: Proceedings of the 2nd European Symposium on Programming (ESOP). *Springer Lecture Notes in Computer Science* **300** 344–358. http://dx.doi.org/10.1007/3-540-19027-9_23.

World Wide Web Consortium (2010a) XQuery1.0: An XML query language. http://www.w3.org/TR/xquery/. W3C Recommendation.

World Wide Web Consortium (2010b) XQuery1.0 and XPath2.0 formal semantics. http://www.w3.org/TR/xquery-semantics/. W3C Recommendation.

Xu, L., Ling, T. W., Wu, H., and Bao, Z. (2009) DDE: From Dewey to a fully dynamic XML labelling scheme. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* 719–730. http://doi.acm.org/10.1145/1559845.1559921.