Research Paper

Rewriting XQuery by child-path folding

Hiroyuki KATO[1], Soichiro HIDAKA[2] and Masatoshi YOSHIKAWA[3]
[1,2]National Institute of Informatics
[3]Kyoto University

ABSTRACT

An XQuery optimization by rewriting based on a partial evaluation using folding expressions is proposed. It consists of two parts: one is the main algorithm, which is a recursive algorithm based on an inductive definition of XQuery expressions. The other is invoked by the main algorithm with an expression whose subexpressions are already folded as its input, and it applies the expression-specific folding function. The main contributions of this paper is to propose an algorithm of an expression-specific folding called "child-path folding". For a given query to the child axis over an element constructed by another XQuery, that is, a composite XQuery, this folding eliminates redundant element construction operators and expressions. These kinds of queries are typical in Global-As-View approach to data integration systems. We also show that all redundant element construction operators and expressions in child-path expressions are eliminated by applying auxiliary transformations. In addition to child-path folding, we discuss folding of major XQuery constructors including FLWOR and quantified expressions. Moreover, we improve the rewriting algorithm based on pruning by adding two annotations to the parsed trees of XQuery expressions.

KEYWORDS

Query optimization, functional language, database, XQuery

1 Introduction

An XQuery is often mapped to the queries supported by the underlying models and it can use their optimization features. However, optimizing XQuery by rewriting is very important because rewriting queries at a high level can dramatically improve the query complexity [1].

In this paper, we present a partial evaluation algorithm for XQuery expressions. This algorithm exploits the functional aspects of XQuery, and it is universal in nature, so many other partial evaluation techniques for functional languages in general (including those not specifically mentioned in this paper) can be easily combined with it.

To make our algorithm universal, it consists of two parts: the main function, *peval*, is a recursive algorithm based on the inductive definition of the XQuery expression; the *fold* function is invoked by *peval* with an expression whose subexpressions are already folded as its input, and it invokes an expression-specific folding function. Consequently, in general, for an expression e which is defined inductively as

$$e ::= \mathsf{op}(e_1, ...e_N),$$

the partial evaluation of e — i.e., $peval(e)$ — can be applied by our algorithm as

$$peval(e) \stackrel{\text{def}}{=} fold(\mathsf{op}(peval(e_1), ..., peval(e_N))),$$

where op is an operator that represents XQuery constructors such as FLWOR.

One of the main contributions in this paper is an algorithm of an expression-specific folding function, *child-path folding*. For a given query to a child axis

Received August 29, 2006; Revised January 19, 2007; Accepted February 8, 2007.
[1)]*kato@nii.ac.jp*, [2)]*hidaka@nii.ac.jp*, [3)]*yoshikawa@i.kyoto-u.ac.jp*
DOI: 10.2201/NiiPi.2007.4.3

over an element constructed by another XQuery — that is, a composite XQuery — this folding eliminates redundant element construction operators and redundant expressions. These composite queries are typical in a Grobal-As-View-based approach [2], which is analogous to view expansion in traditional database systems, adopted in data integration systems [3] because element construction operators correspond to schema definitions in XQuery expressions used as schema mapping [4]. The performance improvement of input XQuery expressions enabled by our rewriting algorithm is equal, at most, to the evaluation cost of redundant element construction operators because applying the *child-path folding* to input expressions eliminates those redundant element constructions. The evaluation cost of element construction operators is the evaluation cost of the expressions enclosed by the operators in addition to the copy cost of evaluated values of the enclosed expressions [5]. The actual cost of element construction operators for the existing famous XQuery engine *Galax* [6] is consistent with the above estimation [7].

Let us consider the following XQuery expression (**Q1**);

```
(for $b in doc("bib.xml")/bib/book,
    $a in $b/author
return <pub>{$a}, {$b/title}</pub>)/author
```

This XQuery forms *expr*/*en*, where *expr* is an XQuery expression, a FLWOR expression in this case, and *en* is an element name, "author". By referring to the XQuery specification [5] the following equation holds.

$$expr/en \equiv expr/\texttt{child::node()/self::}en \quad \cdots (1)$$

By referring to the XQuery formal semantics [8], we can distinguish two properties: one for *expr*/`child::node()`,

(a) `<a>`*expr*′`</a>/child::node()` $=^{1)}$ *expr*′

and the other for *expr*`/self::`*en*;

(b) *expr*′`/a/self::a` = *expr*′`/a`

Through these properties, *child-path folding* results in the following XQuery (**Q2**);

```
for $b in doc("bib.xml")/bib/book,
    $a in $b/author
return $a
```

Query **Q2** is more efficient than **Q1** because of the removal of both the unnecessary element construction operator for `pub` and the path expression $*b*`/title`.

[1)] This equality is actually value-based. The accommodation of node-id equality is part of our ongoing work.

We will show that by applying auxiliary transformation rules, we can eliminate all redundant element construction operators which exploit property (a) and all redundant expressions which exploit property (b). Moreover, the *child-path folding* algorithm is improved through pruning by adding two annotations to parsed trees of XQuery expressions.

In this paper, we focus on the child and self axes for the axis steps because the main purpose of this paper is to explain *child-path folding* algorithm by using properties (a) and (b) described above. However, as we will show, our algorithm can be easily extended for new expressions including other axes such as descendant, attribute, or parent axes.

This paper is structured as follows. After related works are described in the next section, our target XQuery expressions and their notation are shown in Section 3. Section 4 provides an overview of our algorithm. Section 5 describes one of the main contributions, *child-path folding*. In Subsection 5.1, function *cpf* is defined for an expression-specific *child-path folding* as a composite function to exploit the two properties mentioned above. In Subsection 5.2, auxiliary transformation functions to support *child-path folding* are shown. Some of these functions are already known. In Subsection 5.3, we introduce two annotations to parsed trees (XQuery expressions) for pruning recursions. In Section 6, our algorithm is constructed from the functions defined in the previous sections, and its correctness and termination are shown. We conclude in Section 7 and consider our future work.

2 Related works

In the database world, most studies on XQuery optimizations have been for the underlying engines; that is, for relational engines [1] [9], or for their own engines [10]–[12]. In contrast, our optimization is universal in that it applies to any engine.

XQuery Core [8] is a subset of XQuery. While there have been studies on optimizing XQuery Core [13] [14], none mention the XQuery folding that we propose.

An ad hoc enumeration of rewriting rules has been done for XQuery [15] [12]. In particular [12], deals with the folding of element construction operators like ours. However, compared with the technique presented there, ours is much simpler and is powerful enough to be extensible. Thus, it is more general and more widely applicable than the folding described in [12].

3 Target XQuery

This section describes the XQuery expressions that are to be taken as input for our algorithm and the notation we use for these expressions.

XQ ::=

`FLWOR`((*bt*,*vn*,**XQ**)+, **XQ**, **XQ**)	FLWOR
| `QE`((*qf*,*vn*,**XQ**)+, **XQ**)	quantified
| `PE`(**XQ**+)	parenthesized
| `EC`(*en*, **XQ**)	element constructors
| `SPE`(**XQ**,*axis*,*en*)	simplified path
| `IF`(**XQ**,**XQ**,**XQ**)	if
| `VR`(*vn*)	variable references
| `d-v`(`SPE`(**XQ**,*axis*,*en*))	*distinct-values*
| `doc`(*arg*)	*doc*
| `NAryOp`(**XQ**)	*n*-ary operators
| ϵ	an empty sequence ()
| `Literal`	literals

Fig. 1 Abstract syntax for XQuery with our notation.

Fig. 1 shows an abstract syntax for our target XQuery according to our notation. This syntax tree is almost the same as the original XQuery syntax, except for minor changes. Thus, we believe that it will be easy to incorporate our optimizing technology into existing XQuery engines made for the original XQuery syntax.

"`Literal`" represents literals including the Boolean values "TRUE" and "FALSE". In this paper, Boolean values as XQuery expressions are represented by $\top$ for "TRUE" and $\bot$ for "FALSE". Note that we use "TRUE" and "FALSE" in the meta-language to explain our algorithm.

Function calls whose function bodies are defined by XQuery can be naturally expanded into XQuery expressions. For a given function name, the function expansion can be the function body with the formal parameters bound by the argument expressions of the function. We assume that each XQuery-defined function call is expanded in this way.

By introducing *n*-ary operators, our algorithm can take almost all XQuery expressions as inputs. However, expressions accommodated by using these *n*-ary operators cannot benefit from our optimization techniques proposed in this paper. For example, our algorithm deals with a FLWOR expression having an *order by* clause as an *n*-ary operator. A *distinct-values* function call with a simplified path expression as its argument can enjoy our optimization techniques, whereas this function call with expressions other than simplified path expressions are dealt with as *n*-ary operators and cannot benefit from our techniques.

Note that, if operands of *n*-ary operators are not *n*-ary operators these expressions can benefit from our optimization techniques. For example, if a FLWOR expression having *order by* clauses has an element constructor as a "where" expression, the "where" expression can enjoy our optimization techniques. A limitation of our input expressions, though, is that for simplified path expressions only the "child axis" and the "self axis" are allowed, because the main purpose of this paper is to explain the *child-path folding* algorithm.

We want to emphasize that other expressions including other axes, such as descendant, attribute, or parent axes, cannot gain from our optimization techniques because these expressions are treated as *n*-ary operators. However, as we will describe, our algorithm can be easily extended for these new expressions.

3.1 Notation for sequence manipulation

The following notation is used in the meta-language to explain our algorithm. We use [] for sequence constructors.

Let $S = [s_1, \cdots, s_n]$ be a sequence of s_i $(1 \le i \le n)$;

- $|S|$ is the length of $S (= n)$.
- for some k $(1 \le k \le |S|)$
  - $S[k]$ denotes the k-th element from the head of S (s_k).
  - $S[< k]$ denotes a subsequence of S with length $k-1$ where for each i $(1 \le i \le k-1)$, $S[< k][i] = S[i]$.
  - $S[>= k]$ denotes a subsequence of S with length $|S| - k + 1$ where for each j $(k \le j \le |S|)$, $S[>= k][j] = S[j]$.
- **append** s_l **into** $S \stackrel{\text{def}}{=} [s_1, \cdots, s_n, s_l]$

3.2 Notation of XQuery expressions

`VR`(v) represents a variable reference, where v denotes a variable name. The semantics of a variable reference is the value of the expression to which the relevant variable is bound. We call this expression a *binding expression*. We refer to two types of XQuery binding variable, *in-binding* and *let-binding*. In in-binding, a variable is bound to each element of the sequence, which is an evaluated value of the binding expression; in let-binding, a variable is bound to the whole value of the binding expression. An *in-binding expression* denotes an expression that follows "in".

A static environment Γ which maps a variable to the corresponding binding expression *expr* is introduced. For a given variable name v, the variable expansion $ve(v, \Gamma)$ is defined as follows;

$$ve(\texttt{VR}(v), \Gamma) \equiv \begin{cases} expr & \text{if } v \text{ is let-binding} \\ each(expr) & \text{if } v \text{ is in-binding} \end{cases}$$

`FLWOR`(fbs, e_w, e_r) represents a FLWOR expression, where;

- fbs is a binding sequence that consists of let-clauses and/or for-clauses, which are denoted by $(btype, var, expr)$.

- *btype* is a binding type, i.e., "in" for a for-clause or "let" for a let-clause.
- *var* is a variable name.
- *expr* is an expression of **XQ**.
- $fbs[1]$ corresponds to the outer-most binding clause.
- $fbs[|fbs|]$ corresponds to the inner-most binding clause.
- $fbs[i]$ is farther out than $fbs[j]$ iff. $i < j$.

- e_w is an expression preceded by "where" (a where-expression) which is an expression of **XQ**.
- e_r is an expression preceded by "return" (a return-expression) which is an expression of **XQ**.

$\Gamma + fbs$ represents a static environment by updating Γ with the variable bindings in *fbs*.

The following equation can be verified by the formal semantics of FLWOR expressions [8]. For each $i(1 \leq i \leq |fbs|)$,

$$\boxed{\begin{array}{l}\texttt{FLWOR}(fbs, e_w, e_r) \\ \equiv \texttt{FLWOR}(fbs[< i], \top, \texttt{FLWOR}(fbs[>= i], e_w, e_r))\end{array}} \cdots (EF1)$$

where, $\texttt{FLWOR}(fbs[< 1], \top, \texttt{FLWOR}(fbs[>= 1], e_w, e_r)) \equiv \texttt{FLWOR}(fbs, e_w, e_r)$.

$\texttt{QE}(qfr, qbs, e_s)$ represents a quantified expression, where;

- *qfr* is a quantifier; i.e., "some" or "every".
- *qbs* is a sequence of binding clauses denoted by $(var, expr)$ in the quantified expressions
  - *var* is a variable name.
  - *expr* is an expression of **XQ**.
  - $qbs[1]$ corresponds to the outer-most binding clause.
  - $qbs[|qbs|]$ corresponds to the inner-most binding clause.
  - $qbs[i]$ is farther out than $qbs[j]$ iff. $i < j$.
- e_s is an expression preceded by "satisfies" (called a satisfies-expression), which is an expression of **XQ**.

The following equation can be verified by the formal semantics of quantified expressions. For each $i(1 \leq i \leq |qbs|)$,

$$\boxed{\begin{array}{l}\texttt{QE}(qfr, qbs, e_s) \\ \equiv \texttt{QE}(qfr, qbs[< i], \texttt{QE}(qfr, qbs[>= i], e_s))\end{array}} \cdots (EC1)$$

where

$\texttt{QE}(qfr, qbs[< 1], \texttt{QE}(qfr, qbs[>= 1], e_s)) \equiv \texttt{QE}(qfr, qbs, e_s)$.

$\texttt{EC}(en, e)$ represents an element constructor, where *en* is an element name and *e* is an expression of **XQ**.

$\texttt{SPE}(e, axis, en)$ represents a simplified path expression, where *e* is an expression of **XQ**, *axis* is either `child` or `self`, and *en* denotes element names. We call $\texttt{SPE}(e, \texttt{child}, en)$ a *child-path expression*.

$\texttt{PE}[e_1, ..., e_N]$ represents parenthesized expressions, where each e_i is an expression of **XQ**.

$\texttt{IF}(e_c, e_t, e_f)$ represents an if-expression, where e_c, e_t, and e_f are expressions of **XQ** that respectively represent the "test" expression, "then" expression, and "else" expression.

4 Overview of our algorithm

Our algorithm that returns a partially evaluated XQuery expression for an input XQuery expression consists of three functions: *peval*, *fold*, and *cpf*.

4.1 Main function: peval

peval is the main procedure, and it is a recursive algorithm based on the inductive definition of XQuery expressions shown in Fig. 1.

Property 1 *An XQuery expression e is processed by peval after its subexpressions have been processed because of the recursive definition of peval according to the inductive definition of XQuery expressions. For example, when a simplified path expression* $\texttt{SPE}(e', axis, en)$ *is to be processed by peval, the subexpression* e' *will have already been processed by peval.* □

As we explain in subsection 5.3, thanks to Property 1 our algorithm can employ recursion pruning by adding annotation to the parsed trees.

Property 2 *When peval processes expressions having variable binding such as FLWOR expressions and quantified expressions, the binding expressions are processed before processing of the expressions including the variable references bound by these binding expressions. For example, a given* $\texttt{FLWOR}(fbs, e_w, e_r)$*, for all i from 1 to* $|fbs|$*,* $fbs[< i].expr$ *is processed by peval before* $fbs[>= i].expr$*,* e_w *or* e_r *are processed.* □

Property 2 implies that for each variable reference $\texttt{VR}(v)$, its *binding expression* obtained by using the variable expansion function $ve(v)$ has already been processed by *peval*. Thus, *peval* does not need to process variable references. Fig. 2 shows function *peval*. Note that we improve the part where the binding expression is processed in Section 6.1.

```
function peval(e : XQ, Γ : Env) result XQ {
  var Γ′ : Env;

  switch(e){
    case FLWOR(fbs, e_w, e_r): /* improving binding part later */
      copy Γ to Γ′;
      foreach fbs[i] from i := 1 to |fbs|{
        replace fbs[i].expr by peval(fbs[i].expr, Γ′);
        append fbs[i] into Γ′;
      } /* end foreach, binding part first (property 2) */
      return fold(FLWOR(fbs, peval(e_w, Γ′), peval(e_r, Γ′)));
    case QE(qfr, qbs, e_s): /* improving binding part later */
      copy Γ to Γ′;
      foreach qbs[i] from i := 1 to |qbs|{
        replace qbs[i].expr by peval(qbs[i].expr, Γ′);
        append qbs[i] into Γ′;
      } /* end foreach, binding part first (property 2)*/
      return fold(QE(qfr, qbs, peval(e_s));
    case PE[e_1, ..., e_N]:
      return fold(PE[peval(e_1, Γ), ..., peval(e_N, Γ)]);
    case EC(en, e′):
      return fold(EC(en, peval(e′, Γ)));
    case SPE(e′, axis, en):
      return fold(SPE(peval(e′, Γ), axis, en), Γ);
    case IF(e_c, e_t, e_f):
      return fold(IF(peval(e_c, Γ), peval(e_t, Γ), peval(e_f, Γ));
    case d-v(e′):
      return fold(d-v(peval(e′, Γ)));
    case VR(v): case doc(arg): case Literal:
      return fold(e);
    case NAryOp(e_1, ..., e_N):
      return fold(NAryOp(peval(e_1, Γ), ..., peval(e_N, Γ)));
    case ϵ:
      return ϵ;
  } /* end switch
}
```

Fig. 2 Partial evaluation function *peval*.

4.2 Expression-specific folding function; fold

The function *fold*, shown in Fig. 3, is invoked by *peval* with an expression whose subexpressions are already folded as its input, and applies expression-specific folding. For example, for an "if" expression IF(e_c, e_t, e_f), if e_t is equvalent to e_f, the "if" expression could be e_t without evaluating e_c. This is a classical partial evaluation technique. Also, for a FLWOR expression having an *in-binding expression* equvalent to an empty expression, *fold* returns an empty expression. The function *anno* in *fold* is placed just before the exit and is used for adding the annotations described later. At this point in our discussion, we can view *anno* to be an identity function.

Theorem 1 *Except for simplified path expressions, the function "fold" returns expressions having*

```
function fold(e : XQ, Γ : Env) result XQ{
  switch(e){
    case SPE(e′, axis, en):
      if e′ is ϵ
        return ϵ;
      if (e.ind is TRUE) and (axis == "child")
        return fold(cpf(e, en,"c", Γ));
      elseif axis == "self"
        return fold(cpf(e, en,"s", Γ));
      else
        return anno(e);
    case IF(e_c, e_t, e_f):
      if (e_c is ϵ) or (e_t == e_f)
        return e_t;
      return anno(e);
    case FLWOR(fbs, e_w, e_r):
      if there is a ("in", v,ϵ) in fbs /* (cf1) */
          or (e_w is ϵ) or (e_r is ϵ)
        return ϵ;
      return anno(e);
    case QE(qfr, qbs, e_s):
      if there is a (v, ϵ) in qbs /* (cq1) */
        {qfr is "some"} ? {return ⊥;} : {return ⊤;}
      return anno(e);
    case PE[e_1, ..., e_N]:
      reform e by eliminating ϵ expressions
        from e_1, ..., e_N into e′;
      if e′ is PE[nil]
        return ϵ;
      if e′ has a single expression like PE[e_k];
        return e_k;
      return anno(e′);
    case EC(en, [e_1, ..., e_N]):
      return anno(EC(en, fold(PE[e_1, ..., e_N])));
    case d-v(e):
      {e is ϵ} ? {return ϵ;} : {return anno(d-v(e);)}
    case VR(v):
      {ve(v, Γ) is ϵ} ? {return ϵ;} : {return anno(e);}
    case doc(arg): case Literal:
      return anno(e);
  } /* end switch */
}
```

Fig. 3 Expression-specific folding function *fold*.

- *the same semantics as the input expressions, and*
- *the same or more efficient evaluating costs than those of the input expressions.*

The former can be verified through the formal semantics [8], and the latter is trivial because the size of an output expression is reduced or remains the same compared with the corresponding input. A precise discussion of these costs is given in [7]. □

4.3 Child-path expression folding function; cpf

The function cpf is invoked by the $fold$ with a child-path expression and it returns a folded expression by partial evaluation. This is described in the next section.

5 Child-path folding

In this section, we describe folding of a child-path expression $\texttt{SPE}(e, \texttt{child}, en)$ based on partial evaluation.

To begin with, function *cpf* is defined for an expression-specific *child-path folding* as a composite function to exploit the two properties mentioned in Subsection 5.1. Next, auxiliary transformation functions to support *child-path folding* are shown in Subsection 5.2. After that, we introduce two annotations to parsed tree (XQuery expressions) for pruning recursions in Subsection 5.3.

5.1 Composite function to fold child-path expressions

In this subsection, an expression-specific folding function *cpf* is presented for child-path expressions. Function *cpf* eliminates redundant element construction operators and expressions according to the following equation and properties.

The following equation can be verified by the XQuery specification [5]. This is the same as equation (1) with our notation.

$$\boxed{\begin{array}{l}\texttt{SPE}(e, \texttt{child}, en)\\ \equiv \texttt{SPE}(\texttt{SPE}(e, \texttt{child}, \texttt{node()}), \texttt{self}, en)\end{array}} \cdots (2)$$

Properties 3 and 4 are the form of $\texttt{SPE}(e, \texttt{child}, \texttt{node})$ and the form of $\texttt{SPE}(e, \texttt{self}, en)$, each of which appear on the right-hand side of equation (2), respectively.

Property 3 *For XQuery expressions with the form of* $\texttt{SPE}(e, \texttt{child}, \texttt{node()})$*, the following equations hold. Note that the first equation can hold if a value-based equality is used.*

- $\texttt{SPE}(\texttt{EC}(en, expr), \texttt{child}, \texttt{node()}) = expr$
- $\texttt{SPE}(\texttt{QE}(qfr, qbs, e_s), \texttt{child}, \texttt{node()}) = \epsilon$
- $\texttt{SPE}(\texttt{Literal}, \texttt{child}, \texttt{node()}) = \epsilon$
- $\texttt{SPE}(\epsilon, \texttt{child}, \texttt{node()}) = \epsilon$

□

Property 4 *For XQuery expressions with the form of* $\texttt{SPE}(e, \texttt{self}, en)$*, the following equations hold:*

- $\texttt{SPE}(\texttt{EC}(en, e), \texttt{self}, en')$

$$= \begin{cases} \texttt{EC}(en, e); & \text{if } en = en' \\ \epsilon; & \text{else} \end{cases}$$

- $\texttt{SPE}(\texttt{SPE}(e, axis, en), \texttt{self}, en')$

$$= \begin{cases} \texttt{SPE}(e, axis, en); & \text{if } en = en' \\ \epsilon; & \text{else} \end{cases}$$

□

Definition 1 *For each XQuery expression, Fig. 4 shows the definitions of two folding functions: cnode which exploits the property 3, and* σ_{en} *which exploits the property 4.* □

Theorem 2 *For an input expression e, functions cnode and* σ_{en} *return expressions having the same semantics as* $\texttt{SPE}(e, \texttt{child}, \texttt{node()})$ *and* $\texttt{SPE}(e, \texttt{self}, en)$*, respectively. This can be verified from the formal semantics [8].* □

Definition 2 *By using equation (2) for a given child-path expression* $\texttt{SPE}(e, \texttt{child}, en)$*, the folding of the expression through Properties 3 and 4,* $[\![\ \texttt{SPE}(e, \texttt{child}, en), \Gamma]\!]_{fold}$ *is defined as a composite function of cnode and* σ_{en}*:*

$$[\![\texttt{SPE}(e, \texttt{child}, en), \Gamma]\!]_{fold} \stackrel{\text{def}}{=} \sigma_{en}(cnode(e, \Gamma), \Gamma)$$

□

Property 5 *The composite function defined in Definition 2 can be implemented using the following function,* $cpf(e, en, mode, \Gamma)$*, where*

$$cpf(e, en, mode, \Gamma) \stackrel{\text{def}}{=} \begin{cases} cpf(cnode(e, \Gamma), en, \text{“}s\text{”}, \Gamma) & (mode = \text{“}c\text{”}) \\ \sigma_{en}(e, \Gamma) & (mode = \text{“}s\text{”}) \end{cases}$$

Fig. 5 shows function cpf *according to this definition.*

This property is easily verified through Properties 3 and 4. □

Theorem 3 *For all child-path expressions,* cpf *returns expressions which have the same semantics as the input expressions, these expressions have the same or lower costs than the evaluating costs of the inputs, and* cpf *terminates.*

The first part can be verified by referring to the formal semantics [8]. The second can be verified by noting that if input expressions exploit Property 3 or 4, the number of subexpressions composing the output expressions will be reduced, otherwise the number remains unchanged. A precise discussion of these costs is given in [7]. The third is verified by structural induction on XQuery expressions. □

$e(\in \mathbf{XQ})$	$cnode(e,\Gamma)$	$\sigma_{en}(e,\Gamma)$
VR(v) (let-bindig)	$cnode(ve(v,\Gamma),\Gamma)$	$\sigma_{en}(ve(v,\Gamma),\Gamma)$
VR(v) (in-binding)	SPE(e,child,node())	SPE(e,self, en)
PE[e_1, ...e_N]	PE[$cnode(e_1,\Gamma)$, ..., $cnode(e_N,\Gamma)$]	PE[$\sigma_{en}(e_1,\Gamma)$, ..., $\sigma_{en}(e_N,\Gamma)$]
FLWOR(fbs, e_w, e_r)	FLWOR($fbs, e_w, cnode(e_r, \Gamma + fbs)$)	FLWOR($fbs, e_w, \sigma_{en}(e_r, \Gamma + fbs)$)
QE(qfr, qbs, e_s)	ϵ	ϵ
EC(en', e')	e'	e (if $en = en'$) ϵ ($else$)
SPE($e', axis, en'$)	SPE(e,child,node())	e (if $en = en'$) ϵ ($else$)
IF(e_c, e_t, e_f)	IF($e_c, cnode(e_t,\Gamma), cnode(e_f,\Gamma)$)	IF($e_c, \sigma_{en}(e_t,\Gamma), \sigma_{en}(e_f,\Gamma)$)
d-v(e')	SPE(e,child,node())	d-v($\sigma_{en}(e',\Gamma)$)
doc(arg)	SPE(e,child,node())	ϵ
NAryOp(e')	SPE(e,child,node())	SPE(e,self,en)
Literal	ϵ	ϵ
ϵ	ϵ	ϵ

Fig. 4 Definitions of two functions, $cnode$ and σ_{en}, each of which exploits the property 3 and 4, respectively.

```
function cpf(e : XQ, en : QName, mode : "c" | "s", Γ : Env)
result XQ{
  switch(e){
    case VR(v):
      if v is a let-bound variable
        return cpf(ve(v,Γ), en, mode, Γ);
      else /* v is an in-bound variable */
        if mode ="c" /* cnode */
          return SPE(e,child,node())
        else /* σ_en */
          return SPE(e,self, en)
    case EC(en', e'):
      if mode = "c" /* cnode */
        return cpf(e', en,"s", Γ);
      else /* σ_en */
        {en = en'} ? {return e;} : {return ϵ;}
    case SPE(e', axis, en'):
      if mode = "c" /* cnode */
        return SPE(e,child, en);
      else /* σ_en */
        {en = en'} ? {return e;} : {return ϵ;}
    case FLWOR(fbs, e_w, e_r):
      return FLWOR(fbs, e_w, cpf(e_r, en, mode, Γ));
    case PE[e_1, ..., e_N]:
      return PE[cpf(e_1, en, mode, Γ), ..., cpf(e_N, en, modeΓ)];
    case QE(qbs, e_s):
      return ϵ;
    case IF(e_c, e_t, e_f):
      return IF(e_c, cpf(e_t, en, mode, Γ), cpf(e_f, en, mode, Γ));
    case d-v(e'): case doc(arg): case NaryOp(e_1, ..., e_N) :
      return SPE(e,child,en);
    case Literal: case ϵ:
      return ϵ;
  } /* end switch */
```

Fig. 5 Function cpf as a composite function of $cnode$ and σ_{en}.

5.2 Auxiliary transformations to support child-path folding for in-bound variables

So far, the algorithm we've built up in this discussion seems unable to fold in-bound variable references. However, there is a case in which it can fold such variable references. Consider the following query:

for $\$v_1$ **in for** $\$v_{11}$ **in** e_{11}
  return (<a>e_{13}, <c>$\$v_{11}$</c>
    <d>e_{12}</d>),
$\$v_2$ **in** $\boxed{\$v_1/\mathtt{b}}$
return <result>$\boxed{\$v_1/\mathtt{c}}$, $\$v_2$</result>

In this query expression, it is unnecessary to evaluate the element constructor <d>e_{12}</d> because it does not contribute to the answer of this query. However, because the variable expansion $ve(v_1)$ cannot distinguish the redundant expression <d>e_{12}</d>, cpf as it is defined can not be applied to $\$v_1$/b or $\$v_1$/c.

This situation is due to in-bound variables whose run-time values would be the result of evaluating "parenthesized" expressions. When the following expressions occur as *in-binding expressions*, the same problems will arise.

- "parenthesized" expressions.
- variable references in which the corresponding variable expansion expressions are "parenthesized" expressions.
- FLWOR expressions in which the "return" expressions are "parenthesized" expressions.
- "if" expressions in which "true" expressions and/or "false" expressions are "parenthesized" expressions.

fbs(binding sequence of FLWOR)	function definition
(“in”, v, PE[e_1, ...e_N]) $\oplus$ *fbs*′	*fpe*(FLWOR(*fbs*, e_w, e_r)) $\stackrel{\text{def}}{=}$ PE[FLWOR((“in”, v, e_1) $\oplus$ *fbs*′, e_w, e_r), ..., FLWOR((“in”, v, e_N) $\oplus$ *fbs*′, e_w, e_r)]
(“in”, v, FLWOR(*fbs*′, e'_w, e'_r)) $\oplus$ *fbs*″	*ffl*(FLWOR(*fbs*, e_w, e_r)) $\stackrel{\text{def}}{=}$ FLWOR(*fbs*′, e'_w,FLWOR((“in”, v, e'_r) $\oplus$ *fbs*″, e_w, e_r))
(“in”,v, IF(e_c, e_t, e_f)) $\oplus$ *fbs*′	*fif*(FLWOR(*fbs*, e_w, e_r)) $\stackrel{\text{def}}{=}$ IF(e_c,FLWOR((“in”,v, e_t) $\oplus$ *fbs*′, e_w, e_r),FLWOR((“in”,v, e_f) $\oplus$ *fbs*′, e_w, e_r))
(“in”,v, VR(v')) $\oplus$ *fbs*′	$fvr(\text{FLWOR}(fbs, e_w, e_r) \stackrel{\text{def}}{=} \begin{cases} \text{FLWOR}((\text{“in”}, v, ve(v')) \oplus fbs', e_w, e_r) \\ \quad \text{if } v' \text{ is a let-bound variable} \\ \text{FLWOR}((\text{“let”}, \text{VR}(v')) \oplus fbs', e_w, e_r) \\ \quad \text{if } v' \text{ is an in-bound variable} \end{cases}$
(“in”, v, e) $\oplus$ fbs' where e is a unit type.	*fun*(FLWOR((“in”, v, e) $\oplus$ *fbs*, e_w, e_r)) $\stackrel{\text{def}}{=}$FLWOR((“let”, v, e) $\oplus$ *fbs*, e_w, e_r)
qbs(binding sequence of quantified)	function definition
(v,PE[e_1, ..., e_N]) $\oplus$ *qbs*′	$qpe(\text{QE}(qfr, qbs, e_s)) \stackrel{\text{def}}{=} \begin{cases} \text{Or}(\text{QE}(qfr, (v, e_1) \oplus qbs', e_s), ..., \text{QE}(qfr, (v, e_N) \oplus fbs', e_s)) \\ \quad \text{if } qfr \text{ is “some”} \\ \text{And}(\text{QE}(qfr(v, e_1) \oplus qbs', e_s), ..., \text{QE}((qfr, (v, e_N) \oplus fbs', e_s)) \\ \quad \text{if } qfr \text{ is “every”} \end{cases}$
(v, FLWOR(*fbs*, e_w, e_r)) $\oplus$ *qbs*′	$qfl(\text{QE}(qfr, qbs, e_s)) \stackrel{\text{def}}{=} \begin{cases} \text{QE}(qfr, qbs'', \text{And}(e_w, \text{QE}(qfr, (v, e_r) \oplus qbs', e_s))) \\ \quad \text{if } qfr \text{ is “some”} \\ \text{QE}(qfr, qbs'', \text{Or}(\text{Not}(e_w), \text{QE}(qfr, (v, e_r) \oplus qbs', e_s))) \\ \quad \text{if } qfr \text{ is “every”} \end{cases}$ where for all $i(1 \le i \le \lvert fbs \rvert)$, $qbs''[i].vname = fbs[i].vname \wedge qbs''[i].expr = fbs[i].expr$
(v, IF(e_c, e_t, e_f)) $\oplus$ *qbs*′	*qif*(QE(*qfr*, *qbs*, e_s)) $\stackrel{\text{def}}{=}$ IF(e_c,QE(*qfr*, (v, e_t) $\oplus$ *qbs*′, e_s), QE(*qfr*, (v, e_t) $\oplus$ *qbs*′, e_s))
qbs = (v, VR(v')) $\oplus$ *qbs*′	$qvr(\text{QE}(qfr, qbs, e_s)) \stackrel{\text{def}}{=} \begin{cases} \text{QE}(qfr, (v, ve(v')) \oplus qbs', e_s) \\ \quad \text{if } v' \text{ is a let-bound variable} \\ \text{QE}((v, e') \oplus qbs', e_s) \\ \quad \text{if } v' \text{ is a in-bound variable} \\ \quad \text{where } ve(v') = each(e') \end{cases}$
(v, e) $\oplus$ *qbs*′ where e is a unit type.	$qun(\text{QE}(qfr, qbs, e_s)) \stackrel{\text{def}}{=} \begin{cases} \text{FLWOR}([(\text{“let”}, v, e)], nil, \text{QE}(qfr, qbs', e_s)) \\ \quad \text{if } \lvert qbs' \rvert \ge 1 \\ \text{FLWOR}([(\text{“let”}, v, e)], nil, e_s) \\ \quad \text{if } \lvert qbs' \rvert = 0 \end{cases}$

Fig. 6 Auxiliary transformation functions for FLWOR and quantified expressions.

In this subsection, we present transformation rules which eliminate the above expressions, including “parenthesized”, FLWOR, and “if” expressions, and variable references from *in-binding expressions*. Some of these are already known [12] [13] [16]. By transforming expressions through these rules, function *cpf* can eliminate all redundant element construction operators and all redundant expressions in child-path expressions.

Only FLWOR expressions and quantified expressions can have *in-binding expressions* in their *binding sequences*. Note that it is sufficient that only the following FLWOR and quantified expressions are considered without loss of generality by using equation (EF1) and (EQ1).

- FLWOR expressions such as FLWOR(*fbs*, e_w, e_r) with *fbs*=(“in”, v, e) $\oplus$ *fbs*′
- quantified expressions such as QE(qfr, qbs, e_s) with $qbs = (v, e) \oplus qbs'$.

Here, $\oplus$ inserts an element at the top of a sequence.

Example 1 *For a given expression* FLWOR(*fbs*′, e'_w, e'_r) *such that* $fbs' = [(\text{“let”}, v_1, e_1), (\text{“let”}, v_2, e_2), (\text{“in”}, v_3, e_3)]$, *we can get the expression*
FLWOR([(“in”, v_3, e_3)], e'_w, e'_r) *by equation (EF1)*
FLWOR(*fbs*′, e'_w, e'_r) ≡
FLWOR(*fbs*′[< 3], ⊤, FLWOR([(“in”,v_3, e_3)], e'_w, e'_r)) □

$e(\in \mathbf{XQ})$	$ind(e,\Gamma)$	$ptype(e,\Gamma)$
VR(v)	$ind(ve(v,\Gamma))$	$ptype(ve(v,\Gamma))$
PE[$e_1,...e_N$]	$ind(e_1)\vee,...,\vee ind(e_N)$	$ptype(e_1)\cup,...,\cup ptype(e_N)$
FLWOR(fbs,e_w,e_r)	$ind(e_r,\Gamma+fbs)$	$ptype(e_r,\Gamma+fbs)$
QE(qfr,qbs,e_s)	TRUE	$\emptyset$
EC(en',e')	TRUE	$\{en'\}$
SPE($e',axis,en'$)	FALSE	$\{en'\}$
IF(e_c,e_t,e_f)	$ind(e_t,\Gamma)\vee ind(e_f,\Gamma)$	$ptype(e_t,\Gamma)\cup ptype(e_f,\Gamma)$
d-v(e')	FALSE	$ptype(e',\Gamma)$
doc(arg)	FALSE	$\emptyset$
NAryOp(e')	FALSE	ANY
Literal	TRUE	$\emptyset$
ϵ	TRUE	$\emptyset$

Fig. 7 Definition of annotations *ind* and *ptype* for each XQuery expression.

Fig. 6 shows these transformation functions which eliminate "parenthesized", FLWOR, and "if" expressions, as well as variable references, from *in-binding expressions*. Fig. 6 also shows two functions, *fun* and *qun*, which turn an *in-binding clause* whose *binding expression* would be evaluated to a singleton[2]. at runtime into a *let-binding (clause)*. Such expressions, namely element constructors, quantified expressions and literals, are called unit-type expressions. Changing *in-binding* clauses to *let-binding* ones makes it easy to treat a variable reference by using simple variable expansion.

Some of these transformations are from [13] [16], or [12]. For example, function *fll* stems from the "associative law" of for-expressions in XQuery Core [13], and is almost the same as (R2) of the "Standard XQuery Rewriting Rules" in [12]. The correctness of these functions — that is, that the resultant expressions have the same semantics as the inputs — is verified through the formal semantics [8].

Note that a FLWOR expression which is handled by function qfl is limited to a (nested) for-expression in XQuery Core for the sake of easy transformation. We omit the rationale for this minor limitation because of space constraints.

By applying these functions in Fig. 6, we have only to consider simplified path expressions, "distinct-values" functions, "doc" functions, and n-ary operators as in-binding expressions.

5.3 Recursion pruning by two annotations

Composite function cpf defined in Definition 2 invokes cpf recursively according to its definition shown in Fig. 5. This recursion can be avoided by having two annotations with respect to the exploitability of Properties 3 and 4 for an input expression.

[2] A sequence containing exactly one item is called a singleton [5]

```
function anno(e : XQ, Γ : Env) result XQ{
  switch(e){
    case VR(v):
      e.ind := (ve(v,Γ)).ind;
      e.ptype := (ve(v,Γ)).ptype;
    case SPE(e',axis,en'):
      e.ind := FALSE ;
      e.ptype := {en'};
    case IF(e_c,e_t,e_f):
      e.ind := e_t.ind ∨ e_f.ind;
      e.ptype := e_t.ptype ∪ e_f.ptype;
    case FLWOR(fbs,e_w,e_r):
      e.ind := e_r.ind;
      e.ptype := e_r.ptype;
    case QE(qfr,qbs,e_s):
      e.ind := TRUE;
      e.ptype := ∅;
    case PE[e_1,...,e_N]:
      e.ind := e_1.ind∨,...,∨e_N.ind;
      e.ptype := e_1.ptype∪,...,∪e_N.ptype;
    case EC(en',e'):
      e.ind := TRUE;
      e.ptype := {en'};
    case d-v(e'):
      e.ind := FALSE;
      e.ptype := e'.ptype;
    case NAryOp(e):
      e.ind := FALSE;
      e.ptype := { ANY };
    case doc(arg): case Literal: case ε:
      e.ind := FALSE;
      e.ptype := ∅ ;
  } /* end switch */
  return e;
}
```

Fig. 8 Annotation adding function *anno*.

$e(\in \mathbf{XQ})$	$cpf(e, en,\text{“c”},\Gamma)$	$cpf(e, en,\text{“s”},\Gamma)$
VR(v) (let-bound)	$cpf(ve(v,\Gamma), en,\text{“c”},\Gamma)$ (if $e.ind$ =TRUE) SPE(e,child,en) (if $e.ind$ =FALSE)	ϵ (if $en \notin e.ptype$) $cpf(ve(v,\Gamma), en,\text{“s”},\Gamma)$ (if $en \in e.ptype$)
VR(v) (in-bound for SPE)	SPE(e,child,en) ($e.ind$ is always FALSE)	ϵ (if $en \notin e.ptype$) e (if $en \in e.ptype$)
VR(v) (in-bound for d-v)	SPE(e,child,en) ($e.ind$ is always FALSE)	ϵ (if $en \notin e.ptype$) e (if $en \in e.ptype$)
VR(v) (in-bound for doc)	SPE(e,child,en) ($e.ind$ is always FALSE)	ϵ ($en \notin e.ptype$ because of $e.ptype = \emptyset$)
VR(v) (in-bound for NAryOp)	SPE(e,child,en) ($e.ind$ is always FALSE)	SPE(e,self,en)
EC(en', e')	$cpf(e', en,\text{“s”},\Gamma)$	ϵ (if $en \notin ptype$) e (if $en \in ptype$)
SPE(e', $axis$, en')	SPE(e,child,en) ($e.ind$ is always FALSE)	ϵ (if $en \notin ptype$) e (if $en \in ptype$)
FLWOR(fbs, e_w, e_r)	SPE(e,child,en) (if $e.ind$ =FALSE) FLWOR($fbs, e_w, cpf(e_r, en,\text{“c”},\Gamma)$) (if $e.ind$ =TRUE)	ϵ (if $en \notin ptype$) FLWOR($fbs, e_w, cpf(e_r, en,\text{“s”},\Gamma)$) (if $en \in ptype$)
QE(qbs, e_s)	ϵ	ϵ
IF(e_c, e_t, e_f)	SPE(e,child,en) (if $e.ind$ =FALSE) IF($e_c, cpf(e_t, en,\text{“c”},\Gamma), cpf(e_f, en,\text{“c”},\Gamma)$) (if $e.ind$ =TRUE)	ϵ (if $en \notin ptype$) IF($e_c, cpf(e_t, en,\text{“s”},\Gamma), cpf(e_f, en,\text{“s”},\Gamma)$) (if $en \in ptype$)
d-v(e')	SPE(e,child,en) ($e.ind$ is always FALSE)	ϵ (if $en \notin ptype$) e (if $en \in ptype$)
doc(arg)	SPE(e,child,en) ($e.ind$ is always FALSE)	ϵ ($en \notin e.ptype$ because of $e.ptype = \emptyset$)
NaryOp($e_1, ..., e_N$)	SPE(e,child,en) ($e.ind$ is always FALSE)	SPE(e,self,en)
Literal	ϵ	ϵ
ϵ	ϵ	ϵ

Fig. 9 Definition of cpf using two annotations *ind* and *ptype* after applying auxiliary transformation functions.

For example, consider the case where function cpf is invoked with FLWOR(fbs, e_w, e_r) and *mode* =“c” as its arguments. In this case, cpf is recursively invoked with e_r and *mode* =“c” as its arguments according to the definition in Fig. 5. If e_r is a simplified path expression, the first invocation cpf can result in SPE(FLWOR(fbs, e_w, e_r), child, en) without recursive invocation of cpf because simplified path expressions can not exploit Property 3. Similarly, cpf with FLWOR(fbs, e_w, e_r) and *mode* =“s” can result in ϵ without recursion if e_r is an element constructor with element name en' and $en \neq en'$ because this element constructor does not exploit Property 4.

Our algorithm can be improved naturally by making use of the above annotations. When function *peval* processes an expression e, all subexpressions of e are processed first by *peval* according to Property 1. By adding annotations when expressions are folded, our algorithm can use, when processing an expression, annotations of subexpressions of the expression. Function call *anno* in function $fold$ just before exit adds these annotations.

Definition 3 *We now introduce the two annotations ind and ptype to parsed trees (expressions). For a given expression (a parsed tree) e,*

- *e.ind is a Boolean value which denotes whether e can exploit Property 3.*
- *e.ptype is a set of element names which can be used to determine whether e can exploit Property 4.*

□

Definition 4 *For a given expression e, e.ind is trivial from the definition of cnode. However, this annotation can also be defined by recursive function ind, which results a Boolean value. In addition, e.ptype is defined by recursive function ptype(e) which returns a set of element names that can occur when evaluating the expression e. Fig. 7 shows the definition of these two functions.* □

According to Property 1, recursion to subexpressions when annotating an expression is not necessary. Annotations which are already associated with these subexpressions can be used instead. Fig. 8 shows function *anno* which adds two annotations *ind* and *ptype* this way. By using these two annotations, each result of

```
function cpf(e : XQ, en : QName, mode : "c" | "s", Γ : Env)
result XQ{
  if (mode ="c") and (e.ind is FALSE) /* cnode-pruning */
    return anno(SPE(e,child,en));
  if (mode ="s") and (en ∉ e.ptype) /* σ_en-pruning */
    return ϵ;
  switch(e)
    case VR(v):
      if mode = "c" /* let-bound and e.ind is TRUE */
        return cpf(ve(v,Γ),en,mode,Γ);
      else /* mode = "s" and en ∈ e.ptype */
        if v is an in-bound variable
          if e.ptype = {ANY} /* in-bound for NAryOp */
            return SPE(e,self,en);
          elseif .e.ptype = ∅ /* in-bound for doc */
            return ϵ;
          else /* in-bound for SPE or d-v */
            return e;
        else /* v is a let-bound variable */
          return cpf(ve(e,Γ),en,mode,Γ);
    case EC(en′,e′):
      if mode ="c"
        return cpf(e′,en,"s",Γ);
      else /* mode ="s" and en ∈ e.ptype */
        return e;
    case SPE(e′,axis,en′): case d-v(e′):
      return e; /* mode ="s" and en ∈ ptype(e) */
    case IF(e_c,e_t,e_f):
      return IF(e_c,cpf(e_t,en,mode,Γ),cpf(e_f,en,mode,Γ));
    case FLWOR(fbs,e_w,e_r):
      return FLWOR(fbs,e_w,cpf(e_r,en,mode,Γ+fbs));
    case PE[e_1,...,e_N]:
      return PE[cpf(e_1,en,mode,Γ),...,cpf(e_N,en,mode,Γ)];
    case NaryOp(e′):
      return SPE(e,self,en);
    case QE(qfr,qbs,e_s): case Literal: case ϵ:
      return ϵ;
  } /* end switch */
}
```

Fig. 10 Revised "cpf" function using annotations.

function cpf, for each XQuery expression, is re-defined as shown in Fig. 9. We assume that an input expression has already had auxiliary transformation functions applied to it.

Property 6 *Each annotation ind of all in-bound variable references is FALSE after application of the auxiliary transformation functions described in subsection 5.2.* □

Function cpf can now be revised as shown in Fig. 10 by using these two annotations. Note that because function $fold$, with a simplified path expression SPE(e, child, en) as its argument, invokes cpf with e as its argument, e is already folded. In the beginning of revised function cpf shown in Fig. 10, two recursion prunings — *cnode-pruning* using annotation *ind* and σ_{en}*-pruning* using annotation *ptype* — are applied.

- *cnode-pruning*
  A function invocation of cpf with an expression e and *mode* ="c" as its arguments results in SPE(e, child, en) if e.*ind* is FALSE. All in-bound variable references are processed through this pruning because of Property 6.

- σ_{en}*-pruning*
  A function invocation of cpf with an expression e and *mode* ="s" as its arguments results in ϵ if $en \notin e.ptype$, where for all element names en, $en \notin e.ptype$ does not hold if ANY $\in e.ptype$.

6 The algorithm

In this section, we present our algorithm. We show its correctness and its termination. We also show the extensibility of the algorithm.

6.1 Improving binding parts

Because function *peval* may change the forms of input expressions through the invocation of $fold$ or cpf, the auxiliary transformation functions mentioned in Subsection 5.2 are applied *after* applying *peval* to *in-expressions*.

Figs. 11 and 12 respectively show the processing for FLWOR expressions and quantified expressions in function *peval* with the processing of binding parts improved by applying the auxiliary transformation functions. Note that (f1) and (q1), which are expression-specific foldings in function $fold$ are moved to these binding parts because of pruning for the processing of "where" , "return", and "satisfies" expressions.

We can now show our algorithm as follows:

Algorithm 1 *Rewriting Algorithm:*

Input: an XQuery expression shown in Fig. 1

Output: an XQuery expression having the same or less query complexity [17] compared with the input

Method: Call function peval in Fig. 2 with the input. Note that

- *the processing for FLWOR and quantified expressions are replaced as shown by Fig. 11 and Fig. 12, respectively.*
- *function cpf invoked by function fold is shown in Fig. 10*

```
case FLWOR(fbs, e_w, e_r):
  copy Γ to Γ′;
  foreach fbs[i] from i := 1 to |fbs|{
    replace fbs[i].expr by peval(fbs[i].expr, Γ′);
    if (fbs[i].var is a let-bound variable) or (fbs[i].expr.aux is FALSE) or (fbs[i].expr.ind is FALSE)
      append fbs[i] into Γ′;
    else /* (fbs[i].var is an in-bound variable) and (fbs[i].expr.aux is TRUE) */
      switch(fbs[i].expr){ /* applying auxiliary transformation functions */
        case VR(v):
          return fold(FLWOR(fbs[<i], nil, peval(fvr(FLWOR(fbs[>=i], e_w, e_r)), Γ′)));
        case PE(e_1, ..., e_N):
          return fold(FLWOR(fbs[<i], nil, peval(fpe(FLWOR(fbs[>=i], e_w, e_r)), Γ′)));
        case FLWOR(fbs′, e′_w, e′_r):
          return fold(FLWOR(fbs[<i], nil, peval(ffl(FLWOR(fbs[>=i], e_w, e_r)), Γ′)));
        case IF(e_c, e_t, e_f):
          return fold(FLWOR(fbs[<i], nil, peval(fif(FLWOR(fbs[>=i], e_w, e_r)), Γ′)));
        case EC(en, e): case QE(qfr, qbs, e_s): case Literal:
          return fold(FLWOR(fbs[<i], nil, peval(fun(FLWOR(fbs[>=i], e_w, e_r)), Γ′)));
        case ϵ: /* (f1) this condition is moved from fold */
          return ϵ;
        default:
          append fbs[i] into Γ′; } /* end switch */ } /* end foreach */
  return fold(FLWOR(fbs, peval(e_w, Γ′), peval(e_r, Γ′)));
```

Fig. 11 Process for FLWOR expressions with the binding part improved in function *peval*.

```
case QE(qfr, qbs, e_s):
  copy Γ to Γ′;
  foreach qbs[i] from i := 1 to |qbs|{
    replace qbs[i].expr by peval(qbs[i].expr, Γ′);
    if mathitqbs[i].expr.mathitaux is FALSE
      append qbs[i] into Γ′;
    else
      switch (qbs[i].expr){
        case VR(e_1, ..., e_N):
          return fold(QE(qfr, qbs[<i], peval(qvr(QE(qfr, qbs[>=i], e_s)), Γ′)));
        case PE(e_1, ..., e_N):
          return fold(QE(qfr, qbs[<i], peval(qpe(QE(qfr, qbs[>=i], e_s)), Γ′)));
        case FLWOR(fbs′, e′_w, e′_r):
          return fold(QE(qfr, qbs[<i], peval(qfl(QE(qfr, qbs[>=i], e_s)), Γ′)));
        case IF(e_c, e_t, e_f):
          return fold(QE(qfr, qbs[<i], peval(qif(QE(qfr, qbs[>=i], e_s)), Γ′)));
        case EC(en, e): case QE(qfr′, qbs′, e′_s): case Literal:
          return fold(QE(qfr, qbs[<i], peval(qun(QE(qfr, qbs[>=i], e_s)), Γ′)));
        case ϵ: /* (q1) this condition is moved from fold */
          {qfr ==“some”} ? {return ⊥;} : {return ⊤;}
        default:
          append qbs[i] into Γ′;
      } /* end switch ((qbs[i].expr))*/
  } /* end foreach */
  return fold(QE(qfr, qbs, peval(e_s));
```

Fig. 12 Process for quantified expressions with the binding part improved in function *peval*.

□

Theorem 4 *For an XQuery expression our algorithm*

- *results in an XQuery expression having the same semantics as the input expression, and;*
- *results in an XQuery expression with the same or less query complexity than the input, and then*
- *terminates.*

The first and second statements are verified by Theorem 1, Theorem 3 and the correctness of the auxiliary functions mentioned in Subsection 5.2. The third is verified by structural induction on XQuery expressions. □

6.2 Extensibility of the algorithm

Our algorithm can be extended for a new expression $\mathsf{op}(e_1, ..., e_N)$ as follows:

1. add a branch in *peval* in which *peval* is recursively called for operands e_1,...,e_N
2. add a branch in $fold$ which describes expression-specific folding rules
3. add auxiliary transformation rules for the expression if necessary.

7 Conclusion and future work

In this paper, we have proposed an algorithm for the partial evaluation of XQuery. This algorithm exploits the property that among XQuery expressions only element constructors define a "child axis". Our algorithm is capable of eliminating all redundant element constructors from input expressions. We have begun to implement the rewriting algorithm.

Our rewriting algorithm runs in polynomial time in terms of the query size in the worst case because the auxiliary rewriting may require repeatedly visiting the same subexpressions. This rewriting cost is feasible compared with the well-known query rewriting techniques such as magic set rewriting [18] and Predicate Move-Around [19].

This work is still in progress. Future work will include

- further improvement through the introduction of additional annotations,
- the inclusion of *axes* of other directions, such as descendant, attribute, parent axes in simplified path expressions,
- more formal discussion, and
- accommodation of node-ID-based equality.

References

[1] R. Krishnamurthy, P. Kaushik, and J. Naughton, "Efficient XML-to-SQL Query Translation: Where to Add the Intelligence?," in *Proceedings of the Thirtieth International Conference on Vary Large Data Bases*, pp.144–155, 2004.

[2] M. Lenzerini, "Data Integration: A Theoretical Perspectives," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium of Principles of database systems*, pp.233–246, 2002. Tutorial.

[3] A. Y. Halevy, "Structures, semantics and statistics," in *Proceedings of the Thirtieth International Conference on Vary Large Data Bases*, pp.4–6, 2004. Keynote.

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

[5] World Wide Web Consortium. XQery1.0 : An XML Query Language. http://www.w3.org/TR/xquery, Sept. 2005.

[6] The Galax team. The Galax XQuery and XPath 2.0 interpretor, version 0.3.0. http://db.bell-labs.com/galax/.

[7] S. Hidaka, H. Kato, and M. Yoshikawa, "An XQuery Cost Model in Relative Form," Technical report, National Institute of Informaitcs, NII-2005-016E, 2005. http://research.nii.ac.jp/TechReports/05-016E.html

[8] World Wide Web Consortium. XQery1.0 and XPath2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics, Sept. 2005.

[9] A. Deutsch and V. Tannen, "Reformulation of XML Queries and Constraints," in *Proceedings of 8th International Conference on Database Theory*, pp.225–241, 2003.

[10] H.V. 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*, vol.11, no.4, pp.274–291, 2002.

[11] 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*, vol.13, no.3, pp.294–315, 2004.

[12] A. Deutsch, Y. Papakonstantinou, and Y. Xu, "The NEXT Framework for Logical XQuery Opimization," in *Proceedings of the Thirtieth International Conference on Vary Large Data Bases*, pp.168–179, 2004.

[13] M. Fernandez, J. Simeon, and P. Wadler, "A Semi-monad for Semi-structured Data," in *Proceedings of 8th International Conference on Database Theory*, pp.263–300, Jan. 2001.

[14] M. Fernandez and J. Simeon, "Building an Extensible XQuery Engine: Experiences with Galax," in *Second International XML Database Symposium, (XSym2004)*, pp.1–4, 2004.

[15] M. Grinev and S. D. Kuznetsov, "Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience," in *Proceedings of 6th East-European Conference on Advances in Databases and Information Systems (ADBIS '02)*, pp.340–345, Sept. 2002.

[16] A. Poulovassilis and C. Small, "Algebraic query qoptimization for database programming languages," *The VLDB Journal*, vol.5, pp.119–132, 1996.

[17] G. Gottlob, C. Koch, and R. Pichler, "Efficient Algorithms for Processing XPath Queries," in *Proceedings of the 28th International Conference on Vary Large Data Bases*, pp.95–106, 2002.

[18] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan, "Magic is relevant," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp.247–258, Atlantic City, N.J., May 1990.

[19] A. Y. Levy, I. S. Mumick, and Y. Sagiv, "Query optimization by predicate move-around," in *Proceedings of the Twentieth International Conference on Very Large Databases*, pp.96–107, Santiago, Chile, 1994.

Hiroyuki KATO

received his degree in doctorate from Nara Institute of Technology in 1999. He is now an assistant professor at the National Institute of Informatics. His research interests are in XML query languages and Topic Maps query languages.

Soichiro HIDAKA

received his bachelor's degree in engineering and doctorate from the University of Tokyo in 1994 and 1999. He is now an assistant professor at the National Institute of Informatics and The Graduate University for Advanced Studies, studying XML query processing and parallel processing.

Masatoshi YOSHIKAWA

received B.E., M.E. and Ph.D. degrees from Department of Information Science, Kyoto University in 1980, 1982 and 1985, respectively. He was on the faculty of Kyoto Sangyo University from 1985 until 1993. From 1989 to 1990, he was a visiting scientist at Computer Science Department, University of Southern California. In 1993, he joined Nara Institute of Science and Technology (NAIST) as a faculty member. From April 1996 to January 1997, he has stayed at Department of Computer Science, University of Waterloo as a visiting associate professor. From June 2002 to March 2006, he served as a professor at Nagoya University. From April 2006, he has been a professor at Kyoto University. His general research interests are in the area of databases. His current interests include XML databases and index structures for text and multimedia data.