DDO-Free XQuery

Hiroyuki Kato
National Institute of Informatics
kato@nii.ac.jp

Yasunori Ishihara
Osaka University
ishihara@ist.osaka-u.ac.jp

Torsten Grust
Universität Tübingen
torsten.grust@uni-tuebingen.de

ABSTRACT

XQuery has an order-sensitive semantics in the sense that it requires nodes to be sorted in document order without duplicates (or in *Distinct Document Order*, DDO for short). This paper shows that for a given XQuery expression and a nested-relational DTD, the input expression can be transformed into an expression that can be evaluated without—potentially costly—ordering operations even if the input query requires its result to be in DDO. To this end, we propose an XQuery transformation algorithm that consists of simple rewriting rules. The basic idea is inspired by a *generate-and-test* approach as commonly used for solving search problems. We apply this approach when constructing the transformed expression: first, a *skeleton query* is prepared for the *generate* phase. This skeleton query can be evaluated without DDO, but it has the ability to return all nodes in DDO for all XML documents that conform to the input DTD. Second, an output expression is generated by injecting conditions for the *test* phase, which are extracted from the input expression, into the skeleton query. The key to performing both the extraction and injection of conditions in a systematic way is to utilize XQuery transformations that preserve equivalence *up to DDO*.

KEYWORDS

XQuery, Optimization, Static Analysis, Document Order

1 INTRODUCTION

XQuery is a functional query language for processing ordered trees, XML documents in particular. Order is, indeed, central to the semantics of XQuery and its sublanguage XPath. In an XML document, the *document order* is a total order defined over all nodes in the tree, determined by a preorder tree traversal. Reflecting this, the semantics of XQuery/XPath step expressions requires the resulting *nodes to be sorted in document order without duplicates* (or in *Distinct Document Order, DDO* for short): for a given step expression $e/\alpha{::}\tau$, the semantics of the step expression is defined using the `distinct-doc-order` (`ddo`) function, which performs sorting in document order and eliminates duplicates based on node identity [22]:

$$e/\alpha{::}\tau = \mathtt{ddo}(\textbf{for}\ \mathtt{\$fs{:}dot}\ \textbf{in}\ e\ \textbf{return}\ \alpha{::}\tau)$$

where the variable `$fs:dot` binds an implicit context node and the **return** part, $\alpha{::}\tau$, is evaluated over this context node.

One of the typical use cases of XQuery is the so-called twig queries. A twig query extracts subtrees that satisfy tree patterns described in terms of XQuery or XPath [19]. Since we particularly focus on the order-based semantics here, let us make this aspect explicit in the following definition:

Definition 1 (Twig query with DDO)**.** For a given XQuery expression e, a *twig query with DDO specified by* e extracts nodes that satisfy e and are sorted in DDO. To obtain the results of such a query, $\mathtt{ddo}(e)$ is evaluated.

In the context of XQuery/XPath, twig queries with DDO are the norm and not the exception: the generation of DDO-sorted results is the default behavior in these language (that can be explicitly disabled using the **fn:unordered** function).

Since DDO is so central to the semantics of XQuery, it is important to be aware of its cost. Chains of step expressions—or: paths—require multiple applications of the `ddo` function. The repeated node sorting and deduplication effort is a source of inefficiency in XQuery processing. Indeed, to date, a variety of optimization techniques have been proposed to avoid the need for DDO operations [5, 6, 9, 13, 15, 21].

Moreover, DDO hinders the application of classical techniques for query rewriting. For example, consider the following expression:

$$\textbf{for}\ \$v\ \textbf{in}\ \textbf{doc}(\text{“foo.xml”})/\textbf{child}{::}a\ \textbf{return}\ \underline{(\$v,\$v)/\textbf{child}{::}b}$$

One might suppose that the underlined **return** part of this expression could be rewritten as $(\$v/\textbf{child}{::}b, \$v/\textbf{child}{::}b)$ by pushing the axis access into the sequence construction (“projection pushdown”). However, this rewrite is *not* valid because the semantics of step expressions requires DDO. The underlined **return** part should instead be rewritten as $\$v/\textbf{child}{::}b$.

In the present work, we study a class of XQuery expressions, *DDO-free XQuery*, which may be correctly evaluated without invoking DDO operations at all:

Definition 2 (DDO-free XQuery)**.** A step expression $e/\alpha{::}\tau$ is *DDO-free* if the following equation holds:

$$[[e/\alpha{::}\tau]] = [[\textbf{for}\ \mathtt{\$fs{:}dot}\ \textbf{in}\ e\ \textbf{return}\ \alpha{::}\tau]].$$

An XQuery expression e is DDO-free if all step expressions contained in e are DDO-free. Note that DDO is required only in the semantics of step expressions.

The decision problem of DDO-freeness would be hard to solve because of the above semantic definition. We can, however, use the following syntactic restriction which we share with a whole family of theoretical work on XQuery [4, 7, 17]:

Definition 3 (Single-node child-traversal expression)**.** A step expression $\$v/\textbf{child}{::}\tau$ is a *single-node child-traversal expression* if variable $\$v$ is bound through a **for**-expression.

DBPL 2017, Munich, Germany

978-1-4503-5354-0/17/09…$15.00
DOI: 10.1145/3122831.3122832

A single-node child-traversal expression $\$v/\mathbf{child}::\tau$ is DDO-free: **for** binds variable $\$v$ to a single node at a time and navigating to **child** nodes from a single node does not require DDO: under the assumption that XML documents are stored in a serialized (i.e., preorder-based) fashion, evaluation will encounter the children in document order. XML document storage in serialization order is, indeed, prevalent among XPath and XQuery implementations: BaseX [8], MonetDB/XQuery [3], or DB2/pureXML [2], among a variety of others. The same is true for streaming XQuery processors [18]. These systems take advantage of guaranteed node order and evaluate **child** axis steps in a single scan over the document, avoiding any DDO overhead [11, 14, 18]. The present work enables such systems to operate in this particularly efficient DDO-free mode, even if the original input query suggests otherwise. The experiment in Appendix A provides a taste of the—indeed substantial—runtime savings that we can expect from DDO-freeness.

Now, let us consider the following tree pattern in a twig query:

$$\begin{array}{l}\mathbf{for}\ \$b\ \mathbf{in}\ \mathbf{doc}(\text{“foo.xml”})/\mathbf{child}::\mathrm{a}/\mathbf{child}::\mathrm{b}\ \mathbf{return}\\ \quad \mathbf{for}\ \$a\ \mathbf{in}\ \$b/\mathbf{ancestor}::*\ \mathbf{return}\ (\$b,\$a)/\mathbf{child}::\mathrm{c}\end{array} \tag{A}$$

Suppose that an XML document stored as “foo.xml” has a root element “a”, which has several “b” children and several “c” children. To extract nodes that satisfy (A) with DDO, evaluation of ddo(A) suffices. However, naïve evaluation of ddo(A) requires multiple applications of ddo (once per step expression) and may lead to significant sorting overhead. Can we use schema information—as given for the “a”, “b”, and “c” elements above—to obtain a DDO-free expression equivalent to ddo(A)?

The objective of this paper is to prove the following theorem:

Theorem 1 (Main Theorem)**.** For given schema information (a nested-relational DTD) and an XQuery e over an XML document that conforms to the schema, e can be transformed into e' such that $e' = \mathsf{ddo}(e)$ and e' is DDO-free.

Our basic idea to prove the main theorem is inspired by two-phase *generate-and-test* strategies, as they are commonly used in search problems. We adapt this approach to construct a DDO-free XQuery for given schema information (a nested-relational DTD, see below) and an input XQuery, which may not be DDO-free. Conceptually, our approach is as follows:

- *Generate* phase:
  Prepare a DDO-free *skeleton query* s, which has the ability to *generate all nodes in DDO* for any XML document that conforms to the schema.
- *Test* phase:
  Formulate $s[cond]$ by injecting node test conditions *cond* extracted from the input query e into the skeleton query s.

This leaves three questions to answer: (1) how to prepare the skeleton query s, (2) how to extract appropriate conditions from the input query, and (3) how to inject those conditions into the skeleton query. In the sequel, we offer the following solutions:

- We show that the skeleton query s can be derived if schema information is given as a *nested-relational DTD*, which is a type of DTD that is often used in practice [1] (as presented in Section 2.2).
- We show that the input query e can be transformed into e', which has a structure *similar* to that of the skeleton query. Keys are transformations that *preserve equivalence up to DDO*, that is, $\mathsf{ddo}(e) = \mathsf{ddo}(e')$ (as presented in Sections 3 and 4).
- Since e' has a structure similar to that of the skeleton query s, we can obtain the query $s[cond]$ by injecting the conditions *cond* from e' into s in a systematic way (as presented in Section 5).

The obtained query $s[cond]$ is DDO-free because of the definition of s. In addition, $s[cond]$ is equivalent to e up to DDO because the distinct nodes generated by $s[cond]$ are also generated by e. The converse is also true since *cond* is extracted from e', which is equivalent to e up to DDO.

Our main contributions are summarized as follows:

- We design an approach to proving the main theorem inspired by generate-and-test strategies.
- We develop a method of implementing our approach that consists of the following three phases:
  - The *split* phase, in which an input XQuery expression is split into a flat sequence expression such that each component of the sequence expression does not itself contain sequence expressions.
  - The *map* phase, in which each component expression of the sequence expression obtained above is transformed into a **for**-expression with a structure similar to that of the skeleton query to allow conditions to be extracted from the transformed expression.
  - The *inject* phase, in which the conditions extracted above are injected into the skeleton query to obtain a DDO-free expression that is equivalent to the input expression up to DDO.
- XQuery transformation rules up to DDO are developed. The most interesting rule among them is a rule for removing *duplicate-generating* **for**-expressions to obtain DDO-free expressions.
- We carefully design the order of rule application in the split and inject phases.

The remainder of the paper is organized as follows. Section 2 gives an overview of the proposed method as well as a brief introduction to XQuery. Sections 3 and 4 present input query transformation rules that allow conditions to be extracted from transformed queries. In Section 3, the split phase introduced above is described. In Section 4, the map phase is described. Section 5 describes a method of injecting the conditions extracted from the input query into the skeleton query. Section 6 discusses related work before Section 7 wraps up.

2 OVERVIEW

In this section, we begin with a brief introduction to XQuery with a focus on the data model, and then review the target dialect of XQuery that is output by the transformation. We continue with a complete walk-through that clarifies the two-phase *generate-and-test* strategy. To this end, Section 2.2 describes how to derive the skeleton query from schema information and discusses structural features that the skeleton query exhibits. These features allow conditions to be injected in a systematic way. Section 2.3 presents a transformation that rewrites the input query such that we can “read off” the conditions to be placed in the skeleton query. Section 2.4 exercises both phases in terms of a complete example transformation.

$$\overline{D; \mathbf{cxt} \vdash () \rightarrow ()}$$

$$\frac{D; \mathbf{cxt} \vdash r_1 \rightarrow e_1 \quad \cdots \quad D; \mathbf{cxt} \vdash r_N \rightarrow e_N}{D; \mathbf{cxt} \vdash (r_1, \ldots, r_N) \rightarrow (e_1, \ldots, e_N)}$$

$$\frac{\begin{array}{l}((r = l) \vee (r = l^*) \vee (r = l^+) \vee (r = l^?)) \\ \text{a fresh variable } \$u \in \mathit{Var} \qquad D; \$u \vdash D.\mu(l) \rightarrow e\end{array}}{D; \$v \vdash r \rightarrow \mathbf{for}\ \$u\ \mathbf{in}\ \$v/\mathbf{child}::l\ \mathbf{return}\ (\mathbf{if}\ ()\ \mathbf{then}\ \$u\ \mathbf{else}\ (), e)}$$

Figure 1: Skeleton query derivation for NRDTD D**.**

$$\begin{array}{lcl} s & ::= & \mathbf{for}\ \$v\ \mathbf{in}\ \$v/\mathbf{child}::\tau\ \mathbf{return}\ sr \mid () \\ sr & ::= & ((\mathbf{if}\ ()\ \mathbf{then}\ \$v\ \mathbf{else}\ ()), s, \ldots, s) \end{array}$$

Figure 2: The syntax of a skeleton query

2.1 XQuery

2.1.1 The data model of XQuery. XQuery's data model is based on *sequences*, namely, ordered collections of zero or more items. One important characteristic of the data model is that all sequences are *flat*: a sequence never contains other sequences; nested sequence expressions are implicitly flattened by the XQuery processor. In addition, there is no distinction between an item x and the singleton sequence (x) containing that item. Sequences are assigned an *effective Boolean value*: an empty sequence, denoted by $()$, represents **false** while any non-empty sequence represents **true**.

2.1.2 Input XQuery expressions. The subset of XQuery expressions that comprise our input dialect is represented in Figure 3a. Note that the input XQuery form does not include element constructors because we focus on twig queries, which extract subtrees that satisfy given tree patterns. The absence of element constructors renders the target dialect purely functional (constructors in XQuery induce side effects) so that **let**-expressions can be eliminated by replacing bound variables with their defining expressions [15]. Note also that we use a special variable $\$R$ instead of the **doc** function to denote the document node of an input XML document. We use **dos** and **aos** to abbreviate the **descendant-or-self** and **ancestor-or-self** axes, respectively. In addition, we use *Var* and *Label* to denote an infinite set of variables and an infinite set of element tag names (short: labels), respectively.

2.2 The skeleton query

2.2.1 Derivation of skeleton queries. We design skeleton queries based on the following principles, which later enable us to inject test conditions that we extract from the input query:

- A skeleton query is a DDO-free expression,
- the query encodes the schema that the input documents adhere to, and
- stub **if**-conditionals (to be replaced later) are placed in appropriate positions.

Our skeleton queries are formulated based on *nested-relational DTDs*, which are very common in practice [1]. Nested-relational DTDs are a proper subclass of non-recursive, disjunction-free DTDs.

Definition 4 (Document Type Definition (DTD))**.** A DTD over a finite alphabet Σ is a triple $D = (\Sigma, l_0, \mu)$, where l_0 is the root label and μ is a function from Σ to the set of regular expressions over Σ. $\mu(l) = ()$ (the empty sequence) if label l denotes an element leaf node. A regular expression r over Σ is defined as follows:

$$\begin{array}{lcll} r & ::= & l & (*\text{ label, } l \in \Sigma *) \\ & \mid & (r, r, \ldots, r) & (*\text{ sequence } *) \\ & \mid & r^* & (*\text{ zero or more occurrences } *) \\ & \mid & r^+ & (*\text{ one or more occurrences } *) \\ & \mid & r? & (*\text{ zero or one occurrence } *) \\ & \mid & r \mid r \mid \cdots \mid r & (*\text{ disjunction } *) \end{array}$$

Definition 5 (Nested-relational DTD (NRDTD))**.** A DTD $D = (\Sigma, l_0, \mu)$ is an NRDTD if D is non-recursive, and $\mu(l)$ is a sequence $(r_1, \ldots, r_N)$ such that each r_i has the form l_i, l_i^*, l_i^+, or $l_i?$, and all l_is are distinct labels.

The algorithm for deriving a skeleton query from an NRDTD is defined in terms of a set of inference rules, as shown in Figure 1. In these rules, a judgment of the form

$$D; \mathbf{cxt} \vdash r \rightarrow e$$

indicates that, given an NRDTD D and a variable **cxt** of XQuery representing the context position in all the XML documents conforming to D, the regular expression r is transformed into skeleton XQuery e. For a given NRDTD D and a variable $\$R$ representing the root nodes of all the XML documents conforming to D, the skeleton XQuery e is obtained by means of the following judgment:

$$D; \$R \vdash D.l_0 \rightarrow e.$$

The resulting skeleton query e has the syntactic form shown in Figure 2. The query is DDO-free since every step expression it contains is a single-node child-traversal expression.

Example 1. Consider an NRDTD $D_1 = (\Sigma_1, \mathsf{a}, \mu_1)$, where $\Sigma_1 = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and $\mu_1(\mathsf{a}) = (\mathsf{b}*, \mathsf{c}+)$, $\mu_1(\mathsf{c}) = \mathsf{d}?$, and $\mu_1(\mathsf{b}) = \mu_1(\mathsf{d}) = ()$. Then, the skeleton query for D_1 is as follows (for readability, we omit concatenations with the empty sequence and follow the convention to abbreviate $\$v/\mathbf{child}::\tau$ as $\$v/\tau$):

$$\begin{array}{l} \mathbf{for}\ \$a\ \mathbf{in}\ \$R/\mathsf{a}\ \mathbf{return} \\ \quad (\mathbf{if}\ ()\ \mathbf{then}\ \$a\ \mathbf{else}\ (), \\ \quad\ \mathbf{for}\ \$b\ \mathbf{in}\ \$a/\mathsf{b}\ \mathbf{return}\ (\mathbf{if}\ ()\ \mathbf{then}\ \$b\ \mathbf{else}\ ()), \\ \quad\ \mathbf{for}\ \$c\ \mathbf{in}\ \$a/\mathsf{c}\ \mathbf{return} \\ \quad\quad (\mathbf{if}\ ()\ \mathbf{then}\ \$c\ \mathbf{else}\ (), \\ \quad\quad\ \mathbf{for}\ \$d\ \mathbf{in}\ \$c/\mathsf{d}\ \mathbf{return}\ (\mathbf{if}\ ()\ \mathbf{then}\ \$d\ \mathbf{else}\ ()))) \end{array}$$

Note how the stub conditionals **if** () **then** ... are placed to control whether an element is produced or not—these will be replaced in the sequel.

2.2.2 Structural features of skeleton queries. The skeleton query serves as a query template whose stub conditions will be instantiated in the second phase. The two following definitions help to make properties of this template precise:

Definition 6 (Output variable)**.** A variable is said to be an *output variable* when that variable is bound to nodes that may be output.

$$\begin{array}{lcl} e & ::= & \$v \mid (e, e, \ldots, e) \mid () \mid e/\alpha{::}\tau \mid \textbf{for } \$v \textbf{ in } e \textbf{ return } e \\ & \mid & \textbf{if } e \textbf{ then } e \textbf{ else } () \\ \alpha & ::= & \textbf{child} \mid \textbf{parent} \mid \textbf{self} \mid \textbf{descendant} \mid \textbf{ancestor} \\ & \mid & \textbf{descendant-or-self (dos)} \mid \textbf{ancestor-or-self (aos)} \\ \tau & ::= & \text{label} \mid * \end{array}$$

(a) Our input syntax

$$\begin{array}{lcl} es & ::= & (e, e, \ldots, e) \\ e & ::= & \$v \mid \$v/\textbf{child}{::}\tau \mid \textbf{if } e \textbf{ then } e \textbf{ else } () \\ & \mid & \textbf{for } \$v \textbf{ in } \$v/\textbf{child}{::}\tau \textbf{ return } e \\ \tau & ::= & \text{label} \mid * \end{array}$$

(b) Output syntax in the split phase

$$\begin{array}{lcl} es & ::= & (ef, ef, \ldots, ef) \\ ef & ::= & \$v \mid \textbf{for } \$v \textbf{ in } \$v/\textbf{child}{::}\tau \textbf{ return } er \\ er & ::= & ef \mid \textbf{if } cond \textbf{ then } \$v \textbf{ else } () \\ cond & ::= & ep \mid \textbf{if } cond \textbf{ then } e \textbf{ else } () \\ ep & ::= & \$v \mid ep/\textbf{child}{::}\tau \\ \tau & ::= & \text{label} \mid * \end{array}$$

(c) Output syntax in the map phase

$$\begin{array}{lcl} e & ::= & \textbf{for } \$v \textbf{ in } \$v/\textbf{child}{::}\tau \textbf{ return } er \\ er & ::= & ((\textbf{if } conds \textbf{ then } \$v \textbf{ else } ()), e, \ldots, e) \\ conds & ::= & (cond, \ldots, cond) \\ cond & ::= & ep \mid \textbf{if } cond \textbf{ then } e \textbf{ else } () \mid \langle\rangle \\ ep & ::= & \$v \mid ep/\textbf{child}{::}\tau \\ \tau & ::= & \text{label} \mid * \end{array}$$

(d) Output syntax in the inject phase

Figure 3: Input and output syntaxes in each phase

Definition 7 (Consecutive-child-axis **for**-expression)**.** A **for**-expression ($\textbf{for}\ \$v_1\ \textbf{in}\ e_2\ \textbf{return}\ e_3$) is said to be a *consecutive-child-axis* expression when
(1) e_2 has the form $\$v_0/\textbf{child}{::}\tau_1$, and
(2) if e_3 contains a **for**-expression, then in the outermost **for**-expression ($\textbf{for}\ \$v_2\ \textbf{in}\ e_4\ \textbf{return}\ e_5$) of e_3, e_4 has the form ($\$v_1/\textbf{child}{::}\tau_2$).

Intuitively, a consecutive-child-axis **for**-expression is a nested **for**-expression in which the **in** part is a step expression ($\$v/\textbf{child}{::}\tau$) and $\$v$ is defined in the innermost outer **for**.

Property 1. A skeleton query exhibits the following three structural properties:

(a) If a node is output it has been previously bound to an output variable,
(b) all occurrences of **for** are *consecutive-child-axis* **for**-expressions, and
(c) a (stub) **if**-conditional is located in the **return** part of each **for**. These conditionals have the form

$$\textbf{if } \langle\rangle \textbf{ then } \$v \textbf{ else } ().$$

The $\langle\rangle$ conditions are placeholders (or holes) that will be filled with test conditions extracted from the input query. Note that a skeleton query returns *all nodes in DDO* for any XML document that conforms to the input NRDTD if we replace the conditions with **true**. This DDO-property is preserved when we place more restrictive conditions in the holes.

2.3 Transforming input queries and injecting conditions

The transformation of the input queries is the core of the proposed method. By properly transforming an input query, we can obtain an expression with a structure similar to that of a skeleton query. It then becomes possible to "read off" the conditions specified in the input query and to inject those conditions into the skeleton query holes. To facilitate this, we rewrite the transformed input query to take on a specific form:

Property 2. The target form of a transformed input query is a sequence expression $(e_1, \ldots, e_K)$ in which each component expression e_i in $(e_1, \ldots, e_K)$ is a **for**-expression or the variable $\$R$. When e_i is a **for**-expression, it exhibits the following three structural properties:

(a) If a node is output it has been previously bound to an output variable,
(b) all occurrences of **for** are *consecutive-child-axis* **for**-expressions, and
(c) **if**-conditionals that appear in the innermost **return** part of a **for** have the following form:

$$\textbf{if } cond \textbf{ then } \$v \textbf{ else } () \ .$$

The conditions *cond* that appear in these **if**-expressions can be extracted and placed to fill the associated holes in the skeleton query's stub conditionals. Note that the transformed input and skeleton queries exhibit a nearly identical structure (compare Properties 1 and 2).

We structure the transformation of the input query as follows. *Split* and *map* are preparatory; the actual extraction and injection of conditions happens in the final *inject* phase:

- *Split* is described in Section 3. In this phase, an input query that conforms to the syntax shown in Figure 3a is split such that there are no sequence expressions except for the topmost expression. In addition, non-child axes are eliminated. The expressions obtained in this phase conform to the syntax shown in Figure 3b.
- *Map* is described in Section 4. Here, for each expression *e* in the topmost sequence expression *es* obtained in the *split* phase (see Figure 3b), *e* is rewritten into a **for**-expression that is equivalent *up-to-DDO*. Each of these **for**-expression satisfies Property 2. The expressions obtained in this phase conform to the syntax shown in Figure 3c.
- *Inject* is discussed in Section 5. This phase finally extract conditions from the transformed input query and places them in the skeleton's holes. Since the skeleton query and the transformed query share shapes, this injection can be performed in a straightforward fashion.

Key to the input query transformation are rewriting rules that preserve equivalence *up-to-DDO*. We have marked these rules by ($*$) to aid the discussion. In an effort to make the following longer chain of rewriting phases more digestible, we characterize the intermediate XQuery dialects obtained after a phase has completed its work.

2.4 A complete example

For the DTD D_1 given in Example 1, consider the XQuery expression (A) presented in the introduction. In the *split* phase, expression (A) is rewritten into the following form (additionally, standard simplifications [12] have been applied to eliminate empty sequences):

```
(for $v in $R/a return
   for $b in $v/b return
     for $a in $R/a return $b/c,
 for $v in $R/a return
   for $b in $v/b return
     for $a in $R/a return $a/c)
```

Note that topmost syntactic construct is a sequence. Non-child axes have been eliminated. Next, in the *map* phase, the above expression is transformed as follows:

```
(for $v in $R/a return
   for $b in $v/b return
     for $o in $b/c return
       if (if $R/a then $o else ()) then $o else (),
 for $v in $R/a return
   for $o in $v/c return
     if (if (if $R/a then $R/a/b else ()) then $o else ())
     then $o
     else ())
```

If this query outputs a node, it has previously been bound to an output variable (here: \$o). Finally, in the *inject* phase, we extract the conditions from the above expression and place them in the skeleton query's holes (recall Example 1). We obtain the following:

```
for $a in $R/a return
  (if () then $a else (),
   for $b in $a/b return (if () then $b else ()),
   for $c in $a/c return
     (if (if (if $R/a then $R/a/b else ()) then $c else ())
      then $c else (),
      for $d in $c/d return (if () then $d else ())))
```

Note that the condition (with dashed underline) in the first **for**-expression in the sequence expression is not injected since there are no places to inject it in the skeleton query. Again, the application of existing techniques for the elimination of empty sequence expressions [12] leads to a simplified variant of the above:[1]

```
for $a in $R/a return
  for $c in $a/c return
    if (if (if $R/a then $R/a/b else ()) then $c else ())
    then $c else ()
```

Evaluation of this query invokes no DDO operations at all. If we wrap (A) in a ddo() call, it is equivalent to the above expression.

3 SPLIT PHASE

This section describes how a given XQuery expression e—conforming to the input syntax shown in Figure 3a—is transformed into an expression e' that contains no sequence expressions except for the topmost expression. Non-**child** axes are also eliminated in this phase. The obtained expression conforms to the syntax shown in Figure 3b. To this end, six transformation rules are presented. Each transformation is relatively simple.

3.1 Eliminating long-distance axes

A *long-distance axis* step may extract nodes that are not directly adjacent to the step's context node. Steps along these axes, such as **dos**, **descendant**, **aos** and **ancestor**, can be eliminated by translating them into finite sequences of **child** or **parent** axis steps: there is a maximum height of input XML documents that conform to a given NRDTD. The maximum height of trees that conform to NRDTD D can be easily calculated using $\mathtt{MaxH}(D.l_0)$, which is defined as follows:

$$\begin{array}{lcl} \mathtt{MaxH}((r_1,\ldots,r_N)) & = & \text{maximum}(\mathtt{MaxH}(r_1),\ldots,\mathtt{MaxH}(r_N),1) \\ \mathtt{MaxH}(r) & = & \mathtt{MaxH}(D.\mu(l))+1 \quad \text{if } r \in \{l, l^*, l^+, l?\} \end{array}$$

We use H to denote the maximum height of the input trees. For example, the maximum height for XML documents that conform to the NRDTD D_1 presented in Example 1 is $\mathsf{H} = 4$. Each long-distance axis can be eliminated using the following transformation rules that "unroll" the long-distance axis:

$$\frac{e/\mathbf{dos}::\tau}{(e/\mathbf{self}::\tau,\; e/\mathbf{child}::\tau,\; e/\mathbf{child}::*/\mathbf{child}::\tau,\; \ldots,\; \underbrace{e/\mathbf{child}::*/\mathbf{child}::*/\ldots/\mathbf{child}::\tau}_{\mathsf{H}\text{ steps}})}\,(*)$$

$$\frac{e/\mathbf{aos}::\tau}{(e/\mathbf{self}::\tau,\; e/\mathbf{parent}::\tau,\; e/\mathbf{parent}::*/\mathbf{parent}::\tau,\; \ldots,\; \underbrace{e/\mathbf{parent}::*/\mathbf{parent}::*/\ldots/\mathbf{parent}::\tau}_{\mathsf{H}\text{ steps}})}\,(*)$$

Similar transformation rules can be applied to eliminate **descendant**- and **ancestor**-axis step expressions. The generated path expressions "probe" the vertical vicinity of the context node (up to H steps away) for elements with label τ. Some of these probing paths will always yield the empty sequence (). For a path of **parent**-axis steps, such meaningless expressions can be eliminated as described in Section 3.4. For a path of **child**-axis steps, such expressions can be eliminated in the *inject* step, as described in Section 5. The expressions that are obtained after the application of the above rules will have the following syntactic form:

$$\begin{array}{lcl} e & ::= & \$v \mid (e, e, \ldots, e) \mid () \mid e/\alpha::\tau \mid \textbf{for } \$v \textbf{ in } e \textbf{ return } e \\ & \mid & \textbf{if } e \textbf{ then } e \textbf{ else } () \\ \alpha & ::= & \textbf{child} \mid \textbf{parent} \mid \textbf{self} \\ \tau & ::= & \text{label} \mid * \end{array}$$

3.2 Simplifying step expressions

We rely on two transformations to simplify step expressions. Once these two transformations are applied, we obtain single-step expressions that originate in a variable.

[1] We could further unfold the nested **if**-conditional but optimizations along these lines are well-known and not the focus of the present paper.

$$\frac{\textbf{for } \$v \textbf{ in } () \textbf{ return } e}{()} \qquad \frac{\textbf{for } \$v \textbf{ in } \$u \textbf{ return } e}{e[\$v \Leftarrow \$u]}$$

$$\frac{\textbf{for } \$v \textbf{ in } (e_1, \ldots, e_N) \textbf{ return } e}{(\textbf{for } \$v \textbf{ in } e_1 \textbf{ return } e, \ldots, \textbf{for } \$v \textbf{ in } e_N \textbf{ return } e)}$$

$$\frac{\textbf{for } \$v \textbf{ in } (\textbf{for } \$u \textbf{ in } e_1 \textbf{ return } e_2) \textbf{ return } e}{\textbf{for } \$u \textbf{ in } e_1 \textbf{ return } (\textbf{for } \$v \textbf{ in } e_2 \textbf{ return } e)}$$

$$\frac{\textbf{for } \$v \textbf{ in } (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } ()) \textbf{ return } e_3}{\textbf{if } e_1 \textbf{ then } (\textbf{for } \$v \textbf{ in } e_2 \textbf{ return } e_3) \textbf{ else } ()}$$

Figure 4: Known rewriting rules for for-expressions

3.2.1 Pushing axis access. The first transformation is done by applying the following four rules to $e/\alpha{::}\tau$. The rules push the step inside e: in the result, all steps originate in a variable (but not in a sequence, **for**-, or **if**-expression):

$$\frac{(e_1, \ldots, e_N)/\alpha{::}\tau}{(e_1/\alpha :: \tau, \ldots, e_N/\alpha{::}\tau)}(*) \qquad \frac{()/\alpha{::}\tau}{()}$$

$$\frac{(\textbf{for } \$v \textbf{ in } e_1 \textbf{ return } e_2)/\alpha{::}\tau}{\textbf{for } \$v \textbf{ in } e_1 \textbf{ return } e_2/\alpha{::}\tau}(*) \qquad \frac{(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } ())/\alpha{::}\tau}{\textbf{if } e_1 \textbf{ then } e_2/\alpha{::}\tau \textbf{ else } ()}$$

After application, the expression adheres to the following syntax (note non-terminal ep in particular):

$$\begin{array}{lcl} e & ::= & ep \mid (e, e, \ldots, e) \mid () \mid \textbf{for } \$v \textbf{ in } e \textbf{ return } e \\ & \mid & \textbf{if } e \textbf{ then } e \textbf{ else } () \\ ep & ::= & \$v \mid ep/\alpha{::}\tau \\ \alpha & ::= & \textbf{child} \mid \textbf{parent} \mid \textbf{self} \\ \tau & ::= & \text{label} \mid * \end{array}$$

3.2.2 Decomposing multiple steps. Multi-step path expressions are decomposed (this simply follows the standard XQuery semantics):

$$\frac{ep/\alpha_1{::}\tau_1/\alpha_2{::}\tau_2 \qquad \text{a fresh variable } \$v \in \mathit{Var}}{\textbf{for } \$v \textbf{ in } ep/\alpha_1{::}\tau_1 \textbf{ return } \$v/\alpha_2{::}\tau_2}$$

Decomposition leaves us with expressions of this form:

$$\begin{array}{lcl} e & ::= & \$v \mid (e, e, \ldots, e) \mid () \mid \$v/\alpha{::}\tau \mid \textbf{for } \$v \textbf{ in } e \textbf{ return } e \\ & \mid & \textbf{if } e \textbf{ then } e \textbf{ else } () \\ \alpha & ::= & \textbf{child} \mid \textbf{parent} \mid \textbf{self} \\ \tau & ::= & \text{label} \mid * \end{array}$$

3.3 Normalizing "for"-expressions

In an additional preparatory step, we simplify the **in** (or: generator) part of **for**-expressions. The aim is to produce generators $\$v/\alpha{::}\tau$, that are closer to the form required by *consecutive-child-axis* (recall Definition 7). We can call on established rewriting rules for **for**-expressions [17, 20], shown in Figure 4. Here, $e[\$v \Leftarrow \$u]$ represents e with all free occurrences of $\$v$ replaced by $\$u$. These rules bring the query expression into the form defined by Figure 5.

3.4 Eliminating parent and self axes

With Figure 5, we have now reached an intermediate expression form that can be characterized as:

- Every axis step expression originates in a variable,

$$\begin{array}{lcl} e & ::= & \$v \mid (e, e, \ldots, e) \mid () \mid \$v/\alpha{::}\tau \\ & \mid & \textbf{for } \$v \textbf{ in } \$v/\alpha{::}\tau \textbf{ return } e \mid \textbf{if } e \textbf{ then } e \textbf{ else } () \\ \alpha & ::= & \textbf{child} \mid \textbf{parent} \mid \textbf{self} \\ \tau & ::= & \text{label} \mid * \end{array}$$

Figure 5: Syntax after the normalization of for-expressions

- every variable is defined in a **for**-expression where the **in** part is a step expression, and
- every axis is of the **child**, **parent** or **self** type.

These three features imply the following property:

Property 3. For an axis step expression $\$v/\alpha{::}\tau$, all nodes bound to variable $\$v$ will always be found *at the same level* of their input tree if $\$v$ has been defined in an enclosing **for**-expression with a generator of the form $\$u/\textbf{child}{::}\tau'$. Below, we will see that it is reasonable to assume that $\$v$ is defined like this.

We build on this property to develop transformation rules that eliminate **parent** and **self** axis steps. Given a step expression $\$v/\alpha{::}\tau$, the basic idea is the following: we use static analysis to track the levels of the nodes that will be bound to $\$v$. If these levels all agree, we call them the level of $\$v$. Once we know the level of $\$v$, we use it to rewrite $\$v/\alpha{::}\tau$ into a suitable expression of **child**-axis steps.

To implement this static analysis, we introduce two environments: L and Γ. L maps variables to levels in terms of natural numbers.

$$L :: \mathit{Var} \to \mathbf{N}$$

For the special variable $\$R$ that is bound to the document node, $L(\$R) \stackrel{\text{def}}{=} 0$. When the nodes that are bound to $\$v$ are children of the nodes that are bound to $\$u$, $L(\$v) = L(\$u) + 1$. When the node bound to $\$v$ is the document's root element, $L(\$v) = 1$. Γ is an environment for mapping variables to **child**-axis step expressions:

$$\Gamma :: \mathit{Var} \to \{\$v/\textbf{child}{::}\tau, \ldots\} \ .$$

For a given $\$v$, $\Gamma(\$v) = \$u/\textbf{child}{::}\tau$ indicates that variable $\$v$ is defined by the following **for**-expression:

$$\textbf{for } \$v \textbf{ in } \$u/\textbf{child}{::}\tau \textbf{ return } \ldots$$

With these in place, an algorithm for eliminating **self** and **parent** axes is presented in terms of a set of inference rules as shown in Figure 6. According to these rules, a judgment of the form

$$\Gamma; L \vdash e \to e'$$

indicates that for a given Γ and L, an XQuery expression e that conforms to the syntax shown in Figure 5 is transformed into e'; if e' contains step expressions, these will exclusively use the **child** axis. The transformation starts with the top-level expression, $\Gamma = \{\}$ and $L = \{\$R \mapsto 0\}$. In this algorithm, the rules for variables, the empty sequence (), sequence expressions, **if**-conditionals and **child**-axis steps are straightforward. Note that for a sequence expression, nested sequences are not constructed.

Here, we pay particular attention to **for**-expressions to demonstrate that the assumption of Property 3 is reasonable. For a **for**-expression **for** $\$v_1$ **in** $\$v_0/\alpha{::}\tau_1$ **return** e_1, its generator $\$v_0/\alpha{::}\tau_1$ is transformed first. In this transformation, any **self**- or **parent**-axis steps are eliminated (if present). The transformed generator either is a **child**-axis step $\$u/\textbf{child}{::}\tau_2$ or an empty sequence expression

$$\frac{}{\Gamma; L \vdash \$v \to \$v} \qquad \frac{}{\Gamma; L \vdash () \to ()} \qquad \frac{\Gamma; L \vdash e_1 \to e'_1 \qquad \cdots \qquad \Gamma; L \vdash e_1 \to e'_N}{\Gamma; L \vdash (e_1, \ldots, e_N) \to (e'_1, \ldots, e'_N)} \qquad \frac{\Gamma; L \vdash e_1 \to e_3 \qquad \Gamma; L \vdash e_2 \to e_4}{\Gamma; L \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } () \to \textbf{if } e_3 \textbf{ then } e_4 \textbf{ else } ()}$$

$$\frac{\Gamma; L \vdash \$v_0/\alpha{::}\tau_1 \to \$u/\textbf{child}{::}\tau_2 \qquad \Gamma + \{\$v \mapsto \$u/\textbf{child}{::}\tau_2\}; L + \{\$v \mapsto L(\$u) + 1\} \vdash e_1 \to e_2}{\Gamma; L \vdash \textbf{for } \$v \textbf{ in } \$v_0/\alpha{::}\tau_1 \textbf{ return } e_1 \to \textbf{for } \$v \textbf{ in } \$u/\textbf{child}{::}\tau_2 \textbf{ return } e_2} (*)$$

$$\frac{\Gamma; L \vdash \$v_0/\alpha{::}\tau_1 \to ()}{\Gamma; L \vdash \textbf{for } \$v_1 \textbf{ in } \$v_0/\alpha{::}\tau_1 \textbf{ return } e \to ()}$$

$$\frac{}{\Gamma; L \vdash \$v/\textbf{child}{::}\tau \to \$v/\textbf{child}{::}\tau} \qquad \frac{L(\$v_1) = 0}{\Gamma; L \vdash \$v_1/\textbf{self}{::}\tau_1 \to ()} \qquad \frac{L(\$v_1) \geq 1 \qquad \Gamma(\$v_1) = \$v_2/\textbf{child}{::}\tau_2 \qquad (\tau_2 \sqcap \tau_1) = \tau'}{\Gamma; L \vdash \$v_1/\textbf{self}{::}\tau_1 \to \$v_2/\textbf{child}{::}\tau'} (*)$$

$$\frac{L(\$v_1) \leq 1}{\Gamma; L \vdash \$v_1/\textbf{parent}{::}\tau_1 \to ()} \qquad \frac{L(\$v_1) > 1 \qquad \Gamma(\$v_1) = \$v_2/\textbf{child}{::}\tau_2 \qquad \Gamma(\$v_2) = \$v_3/\textbf{child}{::}\tau_3 \qquad (\tau_3 \sqcap \tau_1) = \tau'}{\Gamma; L \vdash \$v_1/\textbf{parent}{::}\tau_1 \to \$v_3/\textbf{child}{::}\tau'} (*)$$

Figure 6: Algorithm for eliminating self and parent axes

()—we use the latter to signal that no unique level could be assigned to $\$v$. Then, the **return** part e_1 is transformed using the two updated environments, yielding e_2. Finally, this transformation results in the **for**-expression

$$\textbf{for } \$v \textbf{ in } \$u/\textbf{child}{::}\tau_2 \textbf{ return } e_2$$

(or `()` if the transformation fails).

For a **self**-axis step expression $\$v_1/\textbf{self}{::}\tau_1$, if the nodes bound to $\$v_1$ are document nodes (denoted by $L(\$v_1) = 0$), then the transformation results in () because document nodes do not have element names; otherwise, the **self**-axis step is transformed into a suitable **child**-axis step. More concretely, suppose that variable $\$v_1$ is defined in the following **for**-expression (denoted by $\Gamma(\$v_1) = \$v_2/\textbf{child}{::}\tau_2$):

$$\textbf{for } \$v_1 \textbf{ in } \$v_2/\textbf{child}{::}\tau_2 \textbf{ return } \ldots$$

Then, if $\tau_1 = *$ or $\tau_1 = \tau_2$ (resp. $\tau_2 = *$ or $\tau_1 = \tau_2$), the transformation results in $\$v_2/\textbf{child}{::}\tau_2$ (resp. $\$v_2/\textbf{child}{::}\tau_1$); otherwise (i,e., if $\tau_1 \neq \tau_2$), the transformation results in (). To see why this meets the XQuery semantics, consider the following path expression obtained by replacing $\$v_1$ with $\Gamma(\$v_1)$:

$$\$v_2/\textbf{child}{::}\tau_2/\textbf{self}{::}\tau_1 \quad .$$

To implement the above strategy, we introduce binary operator $\sqcap$ on labels (element names) as follows:

$$\tau_2 \sqcap \tau_1 = \begin{cases} \tau_1 & ((\tau_2 = *) \lor (\tau_1 = \tau_2)) \\ \tau_2 & ((\tau_1 = *) \lor (\tau_1 = \tau_2)) \\ \tau_3 & ((\tau_1 \nsim \tau_2) \land (\text{a fresh } \tau_3 \in \mathit{Label})) \end{cases}$$

where $\tau_1 \sim \tau_2$ holds iff $\tau_1 = \tau_2$, $\tau_1 = *$, or $\tau_2 = *$. Note that we use $\$v_2/\textbf{child}{::}\tau_3$ with a fresh label τ_3 not used in the DTD to represent ().

Similarly, for a **parent**-axis step $\$v_1/\textbf{parent}{::}\tau_1$, if the nodes that are bound to $\$v_1$ are either document or root nodes, the transformation results in () because neither have parent nodes with element names;[2] otherwise, the transformation results in $\$v_3/\textbf{child} :: \tau'$, as obtained using $\sqcap$ and Γ.

[2]Note that in this paper, τ is either a label (an element name) or $*$ (an arbitrary element name). If τ can be node(), then the bounds for L in the premises of the inference rules for the **self**- and **parent**-axis steps need to be adapted.

The transformed expressions adhere to following dialect (since we have reached a **child**-axis-only intermediate form norm, we use $\$v/\tau$ instead of $\$v/\textbf{child}{::}\tau$ from now on):

$$\begin{array}{rcl} e & ::= & \$v \mid (e, e, \ldots, e) \mid () \mid \$v/\tau \\ & \mid & \textbf{for } \$v \textbf{ in } \$v/\tau \textbf{ return } e \mid \textbf{if } e \textbf{ then } e \textbf{ else } () \\ \tau & ::= & \text{label} \mid * \end{array}$$

3.5 Decomposing sequence expressions

To wrap up the *split* phase, a final set of rules decomposes sequences expression that do not appear at the top-level. We obtain expressions that conform to the non-terminal *es* defined in Figure 3b:

$$\frac{\textbf{for } \$v \textbf{ in } \$u/\tau \textbf{ return } (e_1, \ldots, e_N)}{\begin{array}{c}((\textbf{for } \$v \textbf{ in } \$u/\tau \textbf{ return } e_1), \\ \ldots, (\textbf{for } \$v \textbf{ in } \$u/\tau \textbf{ return } e_N))\end{array}} (*)$$

$$\frac{\textbf{if } (e_1, \ldots, e_N) \textbf{ then } e \textbf{ else } ()}{((\textbf{if } e_1 \textbf{ then } e \textbf{ else } ()), \ldots, (\textbf{if } e_N \textbf{ then } e \textbf{ else } ()))} (*)$$

$$\frac{\textbf{if } e \textbf{ then } (e_1, \ldots, e_N) \textbf{ else } ()}{((\textbf{if } e \textbf{ then } e_1 \textbf{ else } ()), \ldots, (\textbf{if } e \textbf{ then } e_N \textbf{ else } ()))}$$

4 MAP PHASE

Phase *split* (previous section) emits a sequence expression $es = (e_1, \ldots, e_N)$, recall Figure 3b and Property 2 in Section 2.3. The goal of the *map* phase is to rewrite each e_i into a **for**-expression that adheres to Properties 2(a) through (c). In particular, these **for**-expressions (1) explicitly reveal conditions specified in the input query and (2) are in *consecutive-child-axis* form. We give five transformation rules towards this goal. The most interesting among these, presented in Section 4.2.2, transforms *duplicate-generating* **for**-*expressions* into **if**-conditionals. The other four rules are relatively straightforward.

4.1 Introducing output variables

To satisfy Property 2(a), each expression e in *es* is rewritten to introduce explicit output variables:

$$\frac{e}{\textbf{for } \$o \textbf{ in } e \textbf{ return if } \$o \textbf{ then } \$o \textbf{ else } ()}$$

In the **return** part, note that we write **if** $\$o$ **then** $\$o$ **else** () instead of the equivalent $\$o$. As will be shown in Section 5, this notational trick renders condition injection simpler.

$$
\begin{array}{lcl}
es & ::= & (e, e, ..., e) \\
e & ::= & \$v \mid \textbf{for } \$v \textbf{ in } \$v/\tau \textbf{ return } e \\
 & \mid & \textbf{if } cond \textbf{ then } e \textbf{ else } () \\
cond & ::= & \$v \mid \$v/\tau \mid \textbf{if } cond \textbf{ then } e \textbf{ else } () \\
\tau & ::= & \text{label} \mid *
\end{array}
$$

Figure 7: Output syntax after simplification of the "if" part with the normalization of "for"-expressions

We reach the following intermediate syntactic form:

$$
\begin{array}{lcl}
es & ::= & (ef, ef, ..., ef) \\
ef & ::= & \textbf{for } \$v \textbf{ in } e \textbf{ return if } \$v \textbf{ then } \$v \textbf{ else } () \\
e & ::= & \$v \mid \$v/\tau \mid \textbf{if } e \textbf{ then } e \textbf{ else } () \\
 & \mid & \textbf{for } \$v \textbf{ in } \$v/\tau \textbf{ return } e \\
\tau & ::= & \text{label} \mid *
\end{array}
$$

4.2 Transformations to obtain consecutive-child-axis "for"-expressions

Establishing Property 2(b) calls for two transformation rules. The first rule simplifies the condition of a **for**-expression. The second rewrites a *duplicate-generating* **for**-expression into a *consecutive-child-axis* **for** loop. Both rules preserve equivalence *up-to-DDO*.

4.2.1 Simplifying conditions. Conditions expressed in terms of **for**-expressions are simplified by the following rewrite:

$$
\frac{\textbf{if (for } \$v \textbf{ in } \$v/\tau \textbf{ return } e_1\textbf{) then } e_2 \textbf{ else } ()}{\textbf{for } \$v \textbf{ in } \$v/\tau \textbf{ return (if } e_1 \textbf{ then } e_2 \textbf{ else } ())} \;(*)
$$

In this transformation, the input conditional is evaluated once, whereas the **if**-expression in the output is evaluated k times, where k is the length of the sequence of the result of $\$v/\tau$. The resulting duplicates of e_2 render this rule equivalence-preserving *up-to-DDO*. Once this rule has been applied, we are left with an expression of the following shape:

$$
\begin{array}{lcl}
es & ::= & (ef, ef, ..., ef) \\
ef & ::= & \textbf{for } \$v \textbf{ in } e \textbf{ return if } \$v \textbf{ then } \$v \textbf{ else } () \\
cond & ::= & \$v \mid \$v/\tau \mid \textbf{if } cond \textbf{ then } e \textbf{ else } () \\
e & ::= & cond \mid \textbf{for } \$v \textbf{ in } \$v/\tau \textbf{ return } e \\
\tau & ::= & \text{label} \mid *
\end{array}
$$

We re-apply the normalization of **for**-expressions (see Figure 4) and end up with the intermediate XQuery dialect of Figure 7.

4.2.2 Removing duplicate-generating for-expressions. Perhaps the most interesting rule presented here transforms *duplicate-generating* **for**-*expressions* into **if**-conditionals. This, in turn, facilitates the generation of *consecutive-child-axis* **for**-expressions as required by Property 2(b).

Definition 8 (Duplicate-generating **for**-expression)**.** A **for**-expression **for** $\$v$ **in** $\$u/\tau$ **return** e is *duplicate-generating* if variable $\$v$ does not appear in any generator (or **in** part) in e and is not an output variable.

Consider the following duplicate-generating **for**-loop to see why such expressions may yield duplicate nodes:

$$\textbf{for } \$v \textbf{ in } \$u/\tau \textbf{ return } e$$

- If variable $\$v$ does not appear freely in e, e does not depend on $\$v$. Instead, e will yield the same value in each iteration of the **for**-loop. Hence duplicates are generated.
- If variable $\$v$ is free in e, then it should appear in the condition *cond* of an **if**-expression in the form $\$v$ or $\$v/\tau'$ (see the syntax shown in Figure 7), because $\$v$ is not an output variable. While $\$v$ determines whether e generates output at all, the value of e's output variable does *not* depend on $\$v$. Iterated evaluation of e in the **for**-loop may thus generate duplicate nodes.

Duplicate-generating **for**-expressions are eliminated by applying the following transformation rule:

$$
\frac{\begin{array}{l}\textbf{for } \$v \textbf{ in } \$u/\tau \textbf{ return } e \\ \$v \text{ does not appear in any generator of } e \\ \$v \text{ is not an output variable}\end{array}}{\textbf{if } \$u/\tau \textbf{ then } e[\$v \Leftarrow (\$u/\tau)] \textbf{ else } ()} \;(*)
$$

where $e[\$v \Leftarrow (\$u/\tau)]$ represents e with all free occurrences of $\$v$ replaced with $(\$u/\tau)$.

The soundness of the above rule can be proved as follows:

PROOF. Consider the input expression **for** $\$v$ **in** $\$u/\tau$ **return** e.

- If variable $\$v$ is not free in e, the proof is trivial.
- If variable $\$v$ is free in e, $\$v$ appears in the condition of an **if**-expression in the form $\$v$ or $\$v/\tau'$, as described above. Consider the case in which $\$v$ appears in the form $\$v/\tau'$. More specifically, let us discuss the following input **for**-expression without loss of generality:

$$\textbf{for } \$v \textbf{ in } \$u/\tau \textbf{ return (if } \$v/\tau' \textbf{ then } e' \textbf{ else } ())$$

Suppose that $\$u/\tau$ evaluates to $(n_1, ..., n_k)$. We then see that evaluation of the **for**-expression leads to the following sequence of **if**-conditionals:

$$\big(\textbf{if } n_1/\tau' \textbf{ then } e' \textbf{ else } (), \cdots, \textbf{if } n_k/\tau' \textbf{ then } e' \textbf{ else } ()\big) \;.$$

Now, transform the above sequence expression equivalently *up-to-DDO*:

$$
\begin{array}{l}
\Rightarrow \{ \text{ value of } e' \text{ does not change } \} \\
\quad \textbf{if } (n_1/\tau' \textbf{ or} \ldots \textbf{or } n_k/\tau') \textbf{ then } e' \textbf{ else } () \\
\Rightarrow \{ \textbf{ or } \text{ can be replaced by sequence construction } \} \\
\quad \textbf{if } (n_1/\tau', ..., n_k/\tau') \textbf{ then } e' \textbf{ else } () \\
\Rightarrow \{ \text{ preserves effective Boolean value of condition } \} \\
\quad \textbf{if } (n_1, ..., n_k)/\tau' \textbf{ then } e' \textbf{ else } () \\
\Rightarrow \{ \text{ assumption: } \$u/\tau = (n_1, \ldots, n_k) \} \\
\quad \textbf{if } \$u/\tau/\tau' \textbf{ then } e' \textbf{ else } () \\
\Rightarrow \{ \text{ apply replacement } e[\$v \Leftarrow (\$u/\tau)] \} \\
\quad (\textbf{if } \$v/\tau' \textbf{ then } e' \textbf{ else } ())[\$v \Leftarrow (\$u/\tau)]
\end{array}
$$

A similar proof holds for the case in which the condition of the **if**-expression takes the form $\$v$.

□

All **for**-expressions in the resulting expression are now of the *consecutive-child-axis* type. The dialect reads:

$$
\begin{array}{lcl}
es & ::= & (e, e, ..., e) \\
e & ::= & \$v \mid \textbf{for } \$v \textbf{ in } \$v/\tau \textbf{ return } e \\
 & \mid & \textbf{if } cond \textbf{ then } e \textbf{ else } () \\
cond & ::= & ep \mid \textbf{if } cond \textbf{ then } e \textbf{ else } () \\
ep & ::= & \$v \mid ep/\tau \\
\tau & ::= & \text{label} \mid *
\end{array}
$$

Note that replacing the occurrences of $\$v$ with $\$u/\tau'$ may introduce multi-step paths *ep*. These appear only in **if**-conditions, however.

Example 2. Consider the following input **for**-expression:

$$
\begin{array}{l}
\textbf{for } \$a \textbf{ in } \$R/a \textbf{ return} \\
\quad \textbf{for } \$b \textbf{ in } \$a/b \textbf{ return} \\
\quad\quad \textbf{for } \$c \textbf{ in } \$R/c \textbf{ return } \$c \ .
\end{array}
$$

First, since **for** $\$b$ **in** $\$a/b$ **return** ... is duplicate-generating, the above transformation rule is applied. We obtain:

$$
\begin{array}{l}
\textbf{for } \$a \textbf{ in } \$R/a \textbf{ return} \\
\quad \textbf{if } \$a/b \\
\quad \textbf{then } (\textbf{for } \$c \textbf{ in } \$R/c \textbf{ return } \$c) \\
\quad \textbf{else } () \ .
\end{array}
$$

Here, **for** $\$a$ **in** $\$R/a$ **return** ... also is duplicate-generating. We apply the above transformation rule again, yielding the following expression, which finally is of the desired *consecutive-child-axis* type:

$$
\begin{array}{l}
\textbf{if } \$R/a \\
\textbf{then if } \$R/a/b \\
\quad\quad \textbf{then } (\textbf{for } \$c \textbf{ in } \$R/c \textbf{ return } \$c) \\
\quad\quad \textbf{else } () \\
\textbf{else } () \ .
\end{array}
$$

4.3 Revealing the conditions

To satisfy the final Property 2(c), two simple transformation rules are applied. One moves **if**-conditionals to the innermost **return** part of a query, the other normalizes nested conditionals.

4.3.1 Moving conditionals to the innermost return. We apply the following transformation (which is standard XQuery lore if read from bottom to top):

$$
\frac{\textbf{if } e_1 \textbf{ then } (\textbf{for } \$v \textbf{ in } e_2 \textbf{ return } e_3) \textbf{ else } ()}{\textbf{for } \$v \textbf{ in } e_2 \textbf{ return } (\textbf{if } e_1 \textbf{ then } e_3 \textbf{ else } ())}
$$

We obtain expressions of the form:

$$
\begin{array}{lcl}
es & ::= & (ef, ef, ..., ef) \\
ef & ::= & \$v \mid \textbf{for } \$v \textbf{ in } \$v/\tau \textbf{ return } er \\
er & ::= & ef \mid \textbf{if } cond \textbf{ then } cond \textbf{ else } () \\
cond & ::= & ep \mid \textbf{if } cond \textbf{ then } e \textbf{ else } () \\
ep & ::= & \$v \mid ep/\tau \\
\tau & ::= & \text{label} \mid *
\end{array}
$$

4.3.2 Normalizing nested conditionals. To prepare condition injection, it is necessary to normalize conditionals such that (1) the **then** part of the outermost **if**-expression consists of an output variable only, and (2) the conditions to be extracted occur in the outermost conditional:

$$
\frac{\textbf{if } e_1 \textbf{ then } (\textbf{if } e_2 \textbf{ then } e_3 \textbf{ else } ()) \textbf{ else } ()}{\textbf{if } (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } ()) \textbf{ then } e_3 \textbf{ else } ()}
$$

The input expression is now in a form in which conditions can be extracted for injection into the skeleton query. We have presented that final dialect already in Figure 3c of Section 2.

5 INJECT PHASE

In this final phase, the conditions that we have just isolated in the input query are extracted to be placed in their associated holes in the skeleton query (recall the definition in Section 2.2). We reap the benefits of the substantial preparatory work done in the *split* and *map* phases: since the skeleton query and the transformed input expression both satisfy Properties 1 and 2, the injection algorithm is relatively simple. It first locates appropriate positions (holes) in the skeleton query to fill. Conditions are then injected with appropriate variable renaming.

The injection algorithm is, again, specified in terms of a set of inference rules (see Figure 8). In these rules, a judgment of the form

$$(e_1, \ldots, e_N) \uplus s \to s_N$$

indicates that for an expression $(e_1, \ldots, e_N)$ obtained as described in Section 4 and a skeleton query s, a DDO-free XQuery s_N can be obtained by injecting the conditions from $(e_1, ..., e_N)$ into s. The actual injection of the conditions in the individual e_i—whose syntax conforms to the dialect of Figure 3c—is then performed by the judgment

$$e_i \oplus s \to s' \ .$$

Skeleton query s, whose holes may be partially filled already, is further completed by the conditions extracted from e_i to yield the skeleton s'. As we said before, s' will be a DDO-free XQuery. Note that if e_i is a variable, it must be the special variable $\$R$ that represents the input tree's document node. In this case, the final result of the *inject* phase is $(\$R, s_N)$.

Let us thus consider the case in which e_i is not a variable: the conditions of e_i = **for** $\$v_1$ **in** $\$v_2/$**child** $:: \tau_1$ **return** e_r are to be injected into the skeleton query **for** $\$v_3$ **in** $\$v_4/$**child** $:: \tau_2$ **return** s_r. If the queries do *not* traverse the same nodes (denoted by $\tau_1 \nsim \tau_2$), no injection takes place. Otherwise, if $\tau_1 \sim \tau_2$, then the condition in the **return** expression $e_r[\$v_1 \Leftarrow \$v_3]$ is injected into its skeleton counterpart s_r. (Recall that $\tau_1 \sim \tau_2$ holds iff $\tau_1 = \tau_2$, $\tau_1 = *$, or $\tau_2 = *$.) When the conditions in e_r are injected into $(s_1, \ldots, s_N)$, we individually inject them into each s_j (to yield s'_j) which are then grouped into a new sequence expression $(s'_1, ..., s'_N)$. No injection is performed if the expression kinds do not match (**for** vs. **if**). The injection of a condition *cond* into a skeleton **if**-conditional with condition $cond_s$, leads to a merge of both conditions $(cond, cond_s)$. (Note that this also works if $cond_s = ()$ is a hole.)

6 MORE RELATED WORK

The inherent cost of DDO operations has long been acknowledged by the XML and XQuery community, including the W3C itself. [3] On the language level, this led to the inclusion of primitives like **unordered** { } which, however, gives up on establishing document order and retains duplicate nodes [10]. The present work preserves these features of the XQuery data model.

[3] *"... a performance advantage may be realized by [...] granting the system flexibility to return the result in the order that it finds most efficient." — https://www.w3.org/TR/xquery-31/*

$$\frac{e_1 \oplus s \to s_1 \qquad e_2 \oplus s_1 \to s_2 \qquad \dots \qquad e_N \oplus s_{N-1} \to s_N}{(e_1, e_2 \dots, e_N) \uplus s \to s_N}$$

$$\frac{\tau_1 \nsim \tau_2}{(\textbf{for } \$v_1 \textbf{ in } \$v_2/\tau_1 \textbf{ return } e_r) \oplus (\textbf{for } \$v_3 \textbf{ in } \$v_4/\tau_2 \textbf{ return } s_r) \to \textbf{for } v_3 \textbf{ in } \$v_4/\tau_2 \textbf{ return } s_r}$$

$$\frac{\tau_1 \sim \tau_2 \qquad e_r[\$v_1 \Leftarrow \$v_3] \oplus s_r \to e'}{(\textbf{for } \$v_1 \textbf{ in } \$v_2/\tau_1 \textbf{ return } e_r) \oplus (\textbf{for } \$v_3 \textbf{ in } \$v_4/\tau_2 \textbf{ return } s_r) \to \textbf{for } v_3 \textbf{ in } \$v_4/\tau_2 \textbf{ return } e'}$$

$$\frac{e_r \oplus s_1 \to s_1' \qquad \dots \qquad e_r \oplus s_N \to s_N'}{e_r \oplus (s_1, \dots, s_N) \to (s_1', \dots, s_N')}$$

$$\frac{}{(\textbf{for } \$v_1 \textbf{ in } \$v_2/\tau_1 \textbf{ return } e_r) \oplus (\textbf{if } cond_s \textbf{ then } \$v_3 \textbf{ else } ()) \to \textbf{if } cond_s \textbf{ then } \$v_3 \textbf{ else } ()}$$

$$\frac{}{(\textbf{if } cond \textbf{ then } \$v_1 \textbf{ else } ()) \oplus (\textbf{if } cond_s \textbf{ then } \$v_2 \textbf{ else } ()) \to \textbf{if } (cond, cond_s) \textbf{ then } \$v_2 \textbf{ else } ()}$$

Figure 8: Algorithm for injecting the conditions extracted from transformed expressions into skeleton queries

A variety of XML document storage formats and associated path index structures—both native and relational—have been designed to represent and exploit XML serialization order [2, 5, 9, 21]. These systems take care to scan storage and indexes in document order to save on node sorting effort. Axis step evaluation algorithms over these storage structures have been design to operate in a *scan-once-only* fashion to avoid the generation of duplicate nodes [6, 11, 13, 14]. Prevalent of the **child** axis—as proposed by DDO-free XQuery—supports these approaches.

The above dynamic (or: runtime) approaches are complemented by static analyses that build on query transformation [15]. This work is closest to our strategy. The authors of [15] study the transformation of XQuery programs that feature element construction. This is complimentary to the present work and we conjecture that both could be fused to yield queries that save on subtree copying as well as DDO operations.

7 WRAP UP

We show that for a given nested-relational DTD and an XQuery e for an XML document that conforms to the DTD, e can be transformed into a DDO-free XQuery e' such that the resulting nodes still adhere to DDO. DDO-free XQuery is useful since we save on node sorting and de-duplication effort. The runtime savings can be substantial as Appendix A already demonstrates. Future work will complement the present theoretical study with a comprehensive assessment of the benefits of DDO-freeness.

REFERENCES

[1] Marcelo Arenas, Pablo Barceló, Leonid Libkin, and Filip Murlak. 2010. *Relational and XML Data Exchange*. Morgan & Claypool Publishers.

[2] Andrey Balmin, Kevin S. Beyer, Fatma Özcan, and Matthias Nicola. 2006. On the Path to Efficient XML Queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*.

[3] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. 2006. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[4] Giuseppe Castagna, Hyeonseung Im, Kim Nguyen, and Véronique Benzaken. 2015. A Core Calculus for XQuery 3.0 - Combining Navigational and Pattern Matching Approaches. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015*.

[5] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. 2003. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[6] Mary Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. 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.

[7] Pierre Genevès and Nils Gesbert. 2015. XQuery and static typing: tackling the problem of backward axes. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[8] Christian Grün, Alexander Holupirek, Marc Kramis, Marc H. Scholl, and Marcel Waldvogel. 2006. Pushing XPath Accelerator to its Limits. In *First International Workshop on Performance and Evaluation of Data Management Systems (ExpDB)*.

[9] Torsten Grust. 2002. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*.

[10] Torsten Grust, Jan Rittinger, and Jens Teubner. 2007. eXrQuy: Order Indifference in XQuery. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*.

[11] Torsten Grust, Maurice van Keulen, and Jens Teubner. 2003. Staircase Join: Teach a Relational DBMS to Watch its Axis Steps. In *Proceedings of the International Conference on Very Large Databases (VLDB)*.

[12] Bilel Gueni, Talel Abdessalem, Bogdan Cautis, and Emmanuel Waller. 2008. Pruning Nested XQuery Queries. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM)*.

[13] Jan Hidders and Philippe Michiels. 2003. Avoiding Unnecessary Ordering Operations in XPath. In *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*.

[14] Vanja Josifovski, Marcus Fontoura, and Attila Barta. 2005. Querying XML Streams. *VLDB Journal* 14, 2 (2005).

[15] Hiroyuki Kato, Soichiro Hidaka, Zhenjiang Hu, Keisuke Nakano, and Yasunori Ishihara. 2015. Context-preserving XQuery Fusion. *Mathematical Structures in Computer Science (MSCS)* 25 (2015), 916–941.

[16] M. Kay. *The Saxon XSLT and XQuery Processor*. http://www.saxonica.com.

[17] Christoph Koch. 2006. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. Database Syst.* 31, 4 (2006), 1215–1256.

[18] Xiaogang Li and Gagan Agrawal. 2005. Efficient Evaluation of XQuery over Streaming Data. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*.

[19] Ling Liu and M. Tamer Özsu (Eds.). 2009. *Encyclopedia of Database Systems*. Springer.

[20] Mary Fernandez and Jerome Simeon and Philip Wadler. 2001. A Semi-monad for Semi-structured Data. In *Proceedings of 8th International Conference on Database Theory* .

[21] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. 2002. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*.

[22] World Wide Web Consortium. 2010. XQuery1.0 and XPath2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/. (December 2010). W3C Recommendation.

A DO XQUERY PROCESSORS BENEFIT FROM DDO-FREENESS?

While its individual rewritings are simple, the DDO-freeness transformation constitutes a whole-query multi-step transformation that incurs effort at compile time. Are we, then, actually rewarded with reduced query runtime? This brief appendix answers this question for the paper's running example query (A) of Section 2.3. For convenience, we have reproduced the actual XQuery text of the original and transformed queries in Figure 9.

A comprehensive experimental assessment that accompanies the present theoretical study is still due but the following already provides a clear indication of the potential of DDO-freeness. It is a salient feature of DDO-free XQuery that it is implementable *on top* of any existing language implementation—no changes to the XQuery engines' core are required. The following experiments use the XQuery processors BaseX 8.4 [8] and Saxon-HE 9.7.0.18J [16].

The larger the intermediate results of path step evaluation, the more impact we expect to see from a transformation that removes the—possibly many—implicit calls to `ddo()` [10]. To this end, we synthesized a series of XML documents that

(1) validate against the nested-relational DTD D_1 of Example 1 (also see Figure 10a) and
(2) grow in size: a document of size n contains about $1 + 2\,\frac{1}{2} \times n$ elements, arranged in a node hierarchy as depicted in Figure 10b. (In serialized form, this document amounts to $\approx 20 \times n$ bytes of XML text.)

```
(let $R := doc("⟨document of size n⟩") return
   for $b in $R/a/b return
     for $a in $b/ancestor::* return
       ($b,$a)/c)/self::node()
```

(a) Original twig query incurring DDO overhead

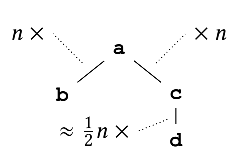
```
let $R := doc("⟨document of size n⟩") return
  for $a in $R/a return
    for $c in $a/c return
      if (if (if $R/a then $R/a/b else ()) then $c
                                           else ()) then $c
                                                    else ()
```

(b) DDO-free query after transformation

Figure 9: Timed XQuery expressions (original vs. DDO-free)

```
<!DOCTYPE a [
  <!ELEMENT a (b*, c+)>
  <!ELEMENT b EMPTY>
  <!ELEMENT c (d?)>
  <!ELEMENT d (#PCDATA)
]>
```

(a) Nested-relational DTD



(b) Sketch of the element hierarchy in document of size n

Figure 10: Generated input XML documents

Table 1: Wall-clock times (measured in milliseconds)[4] for the evaluation of the twig query over different input document sizes n (OOM: no measurement due to out of memory condition)

	BaseX		Saxon	
doc. size n	original	DDO-free	original	DDO-free
1	1.78	1.06	0.56	1.02
10	7.03	2.17	2.69	3.12
100	40.43	5.30	10.70	6.05
1 000	454.20	17.44	287.67	13.53
10 000	OOM	30.69	79646.54	62.15
100 000	OOM	72.80	OOM	217.07
1 000 000	OOM	404.79	OOM	1531.95

Looking at the original twig query of Figure 9a, its evaluation over a document of size n will incur

$$\underbrace{n}_{\text{\# of b nodes}} \times (\underbrace{1}_{\text{ancestor::*}} + \underbrace{1}_{\text{child::c}}) + \underbrace{1}_{\text{self::node()}}$$

invocations of `ddo()` (the initial path $R/a/b does not lead to a `ddo()` operation). The `ddo()` call implicit in the final self::node() step will remove duplicates among n^2 `c` elements, leaving us with an document-ordered result sequence of length n. We instrumented the code of BaseX 8.4 and found its engine to perfectly follow this breakdown of the predicted `ddo()` runtime effort.

Table 1 reports on the evaluation times we observed when the original query and its DDO-free equivalent are evaluated over documents of size $n = 1, 10, \ldots, 1\,000\,000$. We list the average time of 10 runs; for BaseX we include *evaluation* and *printing* (serialization) time. Starting with $n = 1000$, the DDO-free query exhibits a substantial performance advantage of at least an order of magnitude. The gap dramatically widens with growing document size as the original `ddo()`-intensive variants start to struggle with the intermediate node sequences of length n^2 (in fact, both BaseX and Saxon fail to process the larger document instances within a JVM heap budget of 4 GB). We also learn that the DDO-free transformation is safe to be used as the engine's default processing mode since the system always benefits (BaseX) or pays a negligible price for tiny to small document sizes only (Saxon).

A look at BaseX' query plans discloses that the system has to evaluate the original query variant in terms of its `CachedStep` operators which allocate and fill buffers of nodes that are then passed to `ddo()`. Instead, the DDO-free equivalent exclusively relies on the `IterStep` primitive, a path evaluation algorithm that does not use any intermediate node storage.

[4] Intel Core i7 CPU clocked at 3.3 GHz supported by 16 GB RAM. BaseX and Saxon are both implemented in Java.

B STEP BY STEP: THE COMPLETE TRANSFORMATION

For reference and reader convenience, this appendix pulls together the entire series of steps that the input query (A) (see the Introduction) goes through to reach DDO-freeness. We add nothing new to the mix here—the entire approach is documented in the main paper.

B.1 Transformation example in each step in the split phase

```
for $b in $R/child::a/child::b return
  for $a in $b/ancestor::* return
    ($b,$a)/child::c
⇒ { eliminating long-distance axes }
for $b in $R/child::a/child::b return
  for $a in ($b/parent::*,
             $b/parent::*/parent::*,
             $b/parent::*/parent::*/parent::*) return
    ($b,$a)/child::c
⇒ { pushing axis access }
for $b in $R/child::a/child::b return
  for $a in ($b/parent::*,
             $b/parent::*/parent::*,
             $b/parent::*/parent::*/parent::*) return
    ($b/child::c,$a/child::c)
⇒ { decomposing multiple steps }
for $b in (for $v1 in $R/child::a return $v1/child::b) return
  for $a in ($b/parent::*,
             for $v2 in $b/parent::* return
               $v2/parent::*,
             for $v3 in $b/parent::* return
               for $v4 in $v3/parent::* return
                 $v4/parent::*
            ) return
    ($b/child::c,$a/child::c)
⇒ { normalizing for-expressions }
for $v1 in $R/child::a return
  for $b in $v1/child::b return
    (for $a in $b/parent::* return
       ($b/child::c,$a/child::c),
     for $v2 in $b/parent::* return
       for $a in $v2/parent::* return
         ($b/child::c,$a/child::c),
     for $v3 in $b/parent::* return
       for $v4 in $v3/parent::* return
         for $a in $v4/parent::* return
           ($b/child::c,$a/child::c)
    )
⇒ { eliminating parent and self axes }
for $v1 in $R/child::a return
  for $b in $v1/child::b return
    (for $a in $R/child::a return
       ($b/child::c,$a/child::c),
     for $v2 in $R/child::a return
       (),
     for $v3 in $R/child::a return
       ()
    )
⇒ { decomposing sequence expressions }
(for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $a in $R/child::a return
       $b/child::c,
 for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $a in $R/child::a return
       $a/child::c,
 for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $v2 in $R/child::a return
       (),
 for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $v3 in $R/child::a return
       (),
)
⇒ { XQuery folklore: eliminate empty sequences }
(for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $a in $R/child::a return
       $b/child::c,
 for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $a in $R/child::a return
       $a/child::c,
)
```

B.2 Transformation example in each step in the map phase

```
⇒ { introducing output variables }
(for $o in (for $v1 in $R/child::a return
              for $b in $v1/child::b return
                for $a in $R/child::a return
                  $b/child::c) return
   if $o then $o
         else (),
 for $o in (for $v1 in $R/child::a return
              for $b in $v1/child::b return
                for $a in $R/child::a return
                  $a/child::c) return
   if $o then $o
         else ()
)
⇒ { simplifying conditions and normalizing for-expressions}
```

```
(for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $a in $R/child::a return
       for $o in $b/child::c return
         if $o then $o
               else (),
 for $v1 in $R/child::a
 return for $b in $v1/child::b
        return for $a in $R/child::a
               return for $o in $a/child::c
                      return if $o then $o
                                   else ()
)
```

⇒ { removing duplicate-generating expressions }

```
(for $v1 in $R/child::a return
   for $b in $v1/child::b return
     if $R/child::a then
       for $o in $b/child::c return
         if $o then $o
               else ()
                    else (),
 if $R/child::a then
   if $R/child::a/child::b then
     for $a in $R/child::a return
       for $o in $a/child::c return
         if $o then $o
               else ()
                           else ()
                else()
)
```

⇒ { moving conditions to the innermost **return**}

```
(for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $o in $b/child::c return
       if $R/child::a then
         if $o then $o
               else ()
                      else (),
for $a in $R/child::a return
  for $o in $a/child::c return
    if $R/child::a then
      if $R/child::a/child::b then
        if $o then $o
              else ()
                             else ()
                   else()
)
```

⇒ { normalizing nested conditionals }

```
(for $v1 in $R/child::a return
   for $b in $v1/child::b return
     for $o in $b/child::c return
       if (if $R/child::a then $o
                           else ()) then $o
                                    else (),
for $a in $R/child::a return
  for $o in $a/child::c return
    if (if (if $R/child::a then $R/child::a/child::b
                           else ()) then $o
                                    else ()) then $o
                                             else ()
)
```

B.3 Injecting conditions into the skeleton query in the inject phase

⇒ { injecting conditions into the skeleton query }

```
for $a in $R/child::a return
  (if () then $a
         else (),
   for $b in $a/child::b return
     (if () then $b
            else ()),
   for $c in $a/child::c return
     (if (if (if $R/child::a then $R/child::a/child::b
                             else ()) then $c
                                      else ()) then $c
                                               else (),
      for $d in $c/child::d return
        (if () then $d
               else ())
     )
  )
```

⇒ { XQuery folklore: eliminate empty sequences }

```
for $a in $R/a return
  for $c in $a/c return
    if (if (if $R/a then $R/a/b else ()) then $c
                                         else ()) then $c
                                                  else ()
```