Pruning Nested XQuery Queries

Bilel Gueni, Talel Abdessalem, Bogdan Cautis
ENST Paris, UMR CNRS 5141
first.last@enst.fr

Emmanuel Waller
LRI - Université de Paris-Sud
first.last@lri.fr

Abstract

We present in this paper an approach for XQuery optimization that exploits minimization opportunities raised in composition-style nesting of queries. More precisely, we consider the simplification of XQuery queries in which the intermediate result constructed by a subexpression is queried by another subexpression. Based on a large subset of XQuery, we describe a rule-based algorithm that recursively prunes query expressions, eliminating useless intermediate results. Our algorithm takes as input an XQuery expression that may have navigation within its subexpressions and outputs a simplified, equivalent XQuery expression, and is thus readily usable as an optimization module in any existing XQuery processor. We demonstrate by experiments the impact of our rewriting approach on query evaluation costs, and prove formally its correctness.

Keywords: XQuery language, query rewriting, performance analysis, semi-structured, XML.

1 Introduction

XML is by now the de facto standard format for data exchange on the Web. It is also used as a data model for native XML databases and as a common language in systems that integrate data coming from heterogenous sources. It is thus essential to have effective and efficient tools for querying and manipulating XML data. Consequently, query languages such as XPath and XQuery have been receiving a great deal of attention from the research community lately. And, unsurprisingly, query optimization, one of the most important (and most studied) topics in relational databases, has seen a revival in the semi-structured context.

The XQuery language plays a key role in XML data management and has many powerful features such as nesting and composition of *for-let-where-return (FLWR)* query blocks, the construction of hierarchical XML results and the navigation in documents by means of XPath expressions. Unfortunately, its expressive power and operational semantics make the reasoning about query optimization quite difficult and have been the main obstacles in establishing a comprehensive framework for query optimization, although significant progress has been made in this direction.

We study in this paper a novel aspect of XQuery optimization that exploits minimization opportunities raised in a composition-style nesting of XQuery queries. More precisely, we consider the simplification of XQuery expressions in which the intermediate result constructed by a subexpression is queried by another subexpression. In other words, given an XQuery expression with navigation over some documents, we consider a setting in which some of these documents may in fact be *intentional*, defined as the result of other XQuery subexpressions. Our approach is similar in spirit to the one of Simeon et al. [15], of projecting XML documents w.r.t. a given XQuery query. Instead of XML documents, we project XQuery subexpressions w.r.t. other subexpressions querying them.

This kind of composition is common in many scenarios of data exchange, mediation or integration, or in view-based security. Before discussing

in more detail these scenarios, and several others, let us first illustrate the problem we study and the main challenges by a data integration example. The example deals with the reformulation of queries over heterogenous, interconnected sources.

Example 1.1 *Our example is based on the XMark benchmark data [19]. Let us consider three interconnected XML sources S_1, S_2 and S_3. Assuming that S_1 is somehow complemented by the other two sources, mappings between the schema of S_1 and the ones of S_2 and S_3 are defined, by means of some transformation XQuery queries Q_2 and Q_3, as follows*[1]*:*

Q_2 :
```
<site>
{for $i in (docA@S2/site)
 where $i/people/person/@id = "X"
 return ($i/open_auctions/open_auction,
         $i/closed_auctions/closed_auction,
         $i/people/person)
}
</site>
```

Q_3 :
```
<site>
{let $l := for $i in docB@S3/site/closed_auctions/closed_auction
           where ($i/itemref/@item = "car" or
                  $i/buyer/@id = "X") and
                  $i/seller/@id = "Y"
           return $i
 return $l
}
</site>
```

The query Q_2 returns all $open\_auction, closed\_auction$ and $person$ data from the sites containing a person identified by "X". The query Q_3 computes the sequence of the $closed\_auction$ elements having either a buyer identified by "X" or an item car, and a seller identified by "Y". In the two queries, the result is wrapped in a $site$ element.

In this data integration scenario, the role of Q_2 and Q_3 is to define the relationship between S_1, on one hand, and S_2 and S_3, on the other hand. Note that this transfer is only virtual, and the data remains at the sources S_2 and S_3. Moreover, the source S_1 may have its own data, which is complemented by the one produced by Q_2 and Q_3. Intuitively, S_1 could be defined by a virtual document "$source_1 := docC@S_1 \cup Q_2 \cup Q_3$" having an extensional component (for S_1's own data) and two intentional ones (for S_2's and S_3's data) . Let us consider now the following query Q_1, specified over $source_1$, which returns the $open\_auction$ elements that have some $person$ data in common with another document, $docD@S_1$:

Q_1 :
```
for $j in source1
return
 for $k in docD@S1/site
 where $j/person = $k/people/person
 return
  <common-auction>{$j/open_auction}</common-auction>
```

S_1's wrapper module would have no difficulty in executing Q_1 over the extensional part of $source_1$. For the intentional ones, there are two possible approaches: (a) Q_2 and Q_3 are executed at S_2 and S_3, their results are transferred to S_1 and then Q_1 is evaluated over them, or (b) Q_1 is "pushed" to both S_2 and S_3, which evaluate it locally over their respective transformation query and send back to S_1 the result. Unsurprisingly, the latter approach can have significant advantages, especially when Q_1 uses only a small portion from the output of the transformation queries.

Now, by the second approach, S_2's wrapper module has to execute Q_1 over Q_2. This could be done by first evaluating Q_2, then evaluating Q_1 over the intermediate result. But since XQuery is compositional, it is more preferable to interpret this step as a single XQuery expression $Q_{1.2} = Q_1 \circ Q_2$, by simply substituting the virtual variable by its XQuery definition. Importantly, this would allow the query optimizer module to chose the best execution strategy. For instance, $Q_{1.2}$ could be the following XQuery expression:

$Q_{1.2}$:
```
for $j in (the definition of Q2)
return
 for $k in docD@S1/site
 where $j/person = $k/people/person
 return
  <common-auction>{$j/open_auction}</common-auction>
```

At this point, instead of the straightforward execution plan, an efficient query optimizer

[1] We use $doc@S_i$ as short notation for a document URL at S_i.

module should detect that Q_2 is only partially useful in $Q_{1.2}$, since only $open\_auction$ and $person$ elements are queried. Hence the following equivalent yet less expensive query can be executed instead of $Q_{1.2}$.

$\mathbf{Q'_{1.2}}$:

```
for $j in (<site>
            {for $i in (doc2@S2/site)
              where $i/people/person/@id = "X"
              return
             ($i/open_auctions/open_auction, $i/people/person)
            }</site>)
return
 for $k in doc1@S1/site
 where $j/person = $k/people/person
 return
  <common-auction>{$j/open_auction}</common-auction>
```

In the case of S_3, the efficient optimizer would have even greater impact, since the equivalent yet simplified query should in fact replace the content of the Q_3's $site$ element by the empty sequence, because only $closed\_auction$ elements are outputted in the Q_3's $site$ element :

$\mathbf{Q'_{1.3}}$:

```
for $j in <site>{()}</site>
return
 for $k in doc1@S1/site
 where $j/person = $k/people/person
 return
  <common-auction>{$j/open_auction}</common-auction>
```

Our contribution. We study in this paper the simplification of queries that have a composition-style nesting as the one illustrated in Example 1.1. We adopt a static-analysis approach, based on detecting and projecting out the useless parts in subexpressions, keeping only what is needed in order to compute the end result. This task is made difficult by complex relationships between the query blocks. We describe a set of rewrite rules that apply such pruning steps recursively over the blocks of an XQuery query, not only at the uppermost level but at any nesting level in the query. Each rule application will output a strictly simpler (i.e., with less navigation steps) yet equivalent XQuery expression. Our rule-based algorithm applies to a large subset of XQuery and we formally prove its correctness. Importantly, the algorithm takes as input an XQuery that may have navigation within subexpressions and outputs a simplified, equivalent XQuery expression. It can be thus easily plugged as an optimization module in any existing XQuery processor. We demonstrate by experiments the impact of our rewriting approach on evaluation costs in the Galax engine.

In the remaining of this section we further motivate our work and we discuss related research. Queries with this composition-style of nesting are very useful in practice. Transformation XQuery queries for mapping between heterogenous XML sources in integration and mediation scenarios are quite common [11, 21, 1]. The Clio project [11] provides a graphical editor for defining schema mapping definition, generating complex XSLT or XQuery transformations. In peer-to-peer settings, such as the Piazza PDMS [21], a peer can refer to data hold by another peer by means of an XQuery mapping. In this setting, it is crucial to minimize the amount of actual data that is transferred between peers. The Active XML system [1] introduces a flexible framework for peer-to-peer XML integration, by combining in one *active* document materialized (extensional) XML parts with intentional parts defined by calls to Web services. Importantly these services can be defined by XQuery expressions and evaluating a query over an active document amounts precisely to query pushing and composition.

Another important use is in queries posed on security views. In many applications that rely on sensitive data, like medical or juridic applications, access to XML documents may be granted only by querying views over these documents. The views define what data the user can access, and the system may accept only queries formulated over these views. It can either evaluate the global query (i.e., the composition of user query and the views) or can first materialize the views and then evaluate the user query. Obviously, in the case of a large number of views, materializing and maintaining these views can be too costly.

It is also very common to cache and re-use the *definitions* of queries but not necessarily their results. This can for instance guide inexperienced users by allowing them to query XQuery expressions that are already available and well-understood. Finally, our simplification technique

can be used to optimize queries that are automatically generated by some graphical editor, in the style of query-by-example.

Related work. Several works on XQuery processing and optimization adopt an approach based on rewrite rules. In [14, 17, 16, 10, 18, 20], the authors discuss various rules for XQuery normalization or for transformation tasks such as XML-to-SQL translation, elimination of unnecessary ordering operations or introduction of a tree-pattern operator in query plans. These approaches are orthogonal to the query simplification technique presented here. [3] introduces rewrite rules for nesting minimization but does not consider the elimination of useless navigation and result construction. In [5], the authors introduce a logical framework for optimization in the OptXQuery subset of XQuery, the Nested XML Tableaux. They present a set of rewrite rules for normalization and the elimination of repeated navigation steps by means of a group-by operator. The XQuery fragment we consider in this study is strictly more expressive than the one of [5].

More germane to this work is [15], which introduces XML document projection for query optimization. They give a set of rewrite rules for the following task: starting from an XQuery expression Q over a document D, identify and project out the parts of D that are not useful for the evaluation of Q. This is very effective to reduce in-memory computations such as node construction. The technique was later refined and extended to take into account the schema of the document in [2]. Although very close in spirit, our approach subsumes the idea of projecting XML documents since we consider the projection-based simplification of arbitrary XQuery blocks, and not only plain XML documents.

In [21, 6], the authors consider the minimization of queries obtained by following semantic paths (mappings) in the Piazza system. To this end, they study the complexity of query containment for a restricted XQuery flavor, that of conjunctive XML queries (c-XQueries). The role of composition in XQuery evaluation was considered in [12]. For an XQuery fragment strictly smaller than the one we consider here, a formal study of the computational complexity of XQuery without composition is provided. Moreover, [12] shows that, under restrictions, composition can be eliminated and describes a set of rewrite rules to this end.

A problem similar to ours was also studied in the context of publishing relation data in XML format, in projects such as XPeranto [4] and SilkRoute [8]. In Silkroute, the composition of XQuery expressions represented by so called *view forests* over relational sources was considered, where a view forest is a mix of XML structure and SQL expressions representing XQuery-to-SQL translations. These techniques are specific to the XML-over-relational setting and do not transfer to XQuery minimization.

The paper is organized as follows. In Section 2 we present preliminary notions. Section 3 details our rule-based algorithm for XQuery simplification. Section 4 presents the experiments we conducted and we conclude in Section 5.

2 Preliminaries

We describe in this section the data model and XQuery expressions we consider, as well as additional assumptions.

Data model. For the sake of simplicity we present our techniques using a slightly simplified version of the XQuery data model. We consider an XML document as an unranked rooted tree t modeled by a set of edges $\text{EDGES}(t)$, a set of nodes $\text{NODES}(t)$, a distinguished root node $\text{ROOT}(t)$, a labeling function over nodes λ_t assigning to each node a label (or text value) from an infinite alphabet Σ, and a typing function τ_t assigning to each node one of the following kinds: *{document, element, text}*. The *document* type can only be given to the root of the XML document and *text* nodes can only appear as leaves. This simplified model can be extended in straightforward manner to other components of the XQuery data model such as attributes.

XQuery fragment. We focus our study on a

significant subset of XQuery, described by the following grammar.

$$
\begin{array}{lcl}
exp & := & () \\
 & & \mid literal \\
 & & \mid exp, exp \\
 & & \mid exp\ Op\ exp \\
 & & \mid Path \\
 & & \mid \$QName \\
 & & \mid (forClause \mid letClause)+ \\
 & & \ \ (where\ exp)?\ return\ exp \\
 & & \mid (some \mid every)\ \$QName\ in\ exp\ return\ exp \\
 & & \mid if(exp)\ then\ exp\ else\ exp \\
 & & \mid <QName>\{exp\}</QName> \\
 & & \mid element\{QName\}\{exp\} \\
forClause & := & for\ \$QName\ in\ exp \\
letClause & := & let\ \$QName := exp \\
Path & := & (doc(uri) \mid \$QName \mid exp)/Step \mid doc(uri) \\
Op & := & < \mid > \mid = \mid + \mid - \mid * \mid << \mid >> \mid \text{“}is\text{”} \\
Step & := & NodeTest(/Step)? \mid text() \\
NodeTest & := & QName \mid \text{”} * \text{”}
\end{array}
$$

Figure 1: *XQuery fragment*

This grammar captures the main XQuery constructs used in practice, such as literal values, sequence construction, variables, FLWR blocks, conditionals, quantifiers, comparisons operators, logical or arithmetic operations, element constructions. For clarity and space reasons, we consider in this paper XPath navigation only along the `child axis` (/). Extensions to other navigation axis such as attribute (/@) and descendant (//) will be presented in an extended version of this work. We also ignore path qualifiers, which can always be reformulated away using *where* clauses.

XQuery normalization. Before applying our technique for query simplification, we assume that some of the standard normalization steps, usually employed to reduce XQuery expressions to equivalent expressions in the simpler language XQuery Core [7], are first applied. This normalization phase will allow us to present our inference algorithm based on a uniform syntactic formulation. We give in Figure 2 the set of normalization rules we consider, each of them being self-explanatory. In short, they either facilitate the extraction of XPath expressions referencing a given variable or reformulate nested expressions in order to have one variable per clause.

Inference rules notation and environment. We present our algorithm via a set of inference

$$
\begin{array}{c}
e/step_1/\ldots/step_n \Rightarrow \\
let\ \$var := e\ return\ \$var/step_1/\ldots/step_n \\
\\
for\ \$var_1\ in\ e_1, \ldots, \$var_n\ in\ e_n\ return\ e \Rightarrow \\
for\ \$var_1\ in\ e_1\ return\ for \ldots\ for\ \$var_n\ in\ e_n\ return\ e \\
\\
let\ \$var_1 := e_1, \ldots, \$var_n := e_n\ return\ e \Rightarrow \\
let\ \$var_1 := e_1\ return\ let \ldots\ let\ \$var_n := e_n\ return\ e \\
\\
some\ \$var_1\ in\ e_1, \ldots, \$var_n\ in\ e_n\ satisfies\ e \Rightarrow \\
some\ \$var_1\ in\ e_1\ satisfies\ some \ldots\ some\ \$var_n\ in\ e_n\ satisfies\ e \\
\\
every\ \$var_1\ in\ e_1, \ldots, \$var_n\ in\ e_n\ satisfies\ e \Rightarrow \\
every\ \$var_1\ in\ e_1\ satisfies\ every \ldots\ every\ \$var_n\ in\ e_n\ satisfies\ e \\
\\
<QName>\{e\}</QName> \Rightarrow element\{QName\}\{e\}
\end{array}
$$

Figure 2: *Standard normalization Rules*

rules, and we adopt standard programming languages notation similar to the one used in [15]. Inference rules are based on *judgements*, which denote statements of the form:

$$Env \vdash f(p_1, \ldots, p_n) \Rightarrow res.$$

Such a statement reads as follows: *the judgement holds iff in the environment Env, by calling the function f with parameters $p_1, \ldots, p_n$ we obtain the result res.*

Inference rules are represented as follows:

$$\frac{premise_1 \ldots premise_n}{Env \vdash f(p_1, \ldots, p_n) \Rightarrow res}$$

where each premise is a judgement. Such a rule reads as follows: *the judgement ($Env \vdash f(p_1, \ldots, p_n) \Rightarrow res$) holds if the premises $premise_1 \ldots premise_n$ hold.* The functions we consider in our inference rules will be defined in Section 3.

In XQuery, a variable is always associated (by either $\$var\ in\ exp$ or $\$var := exp$) to a subexpression, in this way being bound to the intermediate XML values returned by the subexpression.

Example 2.1 *For instance, in the query Q_2 of the running example, variable $\$i$ is bound to elements produced by the XPath $docA@S2/site$. Similarly, in the query Q_3, variable $\$l$ is bound to some elements produced by variable $\$i$. This is because the FLWR block to which $\$l$ is bound returns some elements over which $\$i$ iterates, those that satisfy certain conditions. In $Q_{1,2}$, the variable $\$j$ will be bound to a constructed $site$ element wrapping some content returned by Q_2's FLWR expression.*

For a variable $\$var$, by the *bound expression* associated with $\$var$ (in short, $exp_b(\$var)$) we denote the expression exp appearing in either a $for\ \$var\ in\ exp$, $let\ \$var\ :=\ exp$ or $some\ \$var\ in\ exp$ statement that declares $\$var$. Also, by the *return expression* of $\$var$ (in short, $exp_r(\$var)$) we denote the associated $where$, $return$ or $satisfies$ part.

In the presentation of our rule-based algorithm, we will rely on a memory space (denoted *environment*) that records for each variable the kind of intermediary results to which it is bound. The environment will contain a set of variable/value mappings, where each mapping binds a variable $\$var$ to a set of objects. Formally, this is written $\$var \Rightarrow \{o_1, \ldots, o_m\}$. We distinguish three possible kinds of such objects: (i) results of an XPath expression (represented in the environment by the XPath expression itself), (ii) element constructors with some element content (can be any XQuery subexpression), (iii) text values (denoted simply `#text`).

Going back to the example, we can thus write $\$i \Rightarrow \{docA@S2/site\}$ for Q_2, $\$i \Rightarrow \{docB@S3/site/closed\_auctions/closed\_auction\}$ and $\$l \Rightarrow \{\$i\}$ for Q_3, or $\$j \Rightarrow \{$<*site*>$\ldots$</*site*>$\}$ for $Q_{1.2}$.

For the construction of the environment, we determine by a static analysis for each variable the objects returned as intermediate XML values by its bound expression. This is done using the function $varRes()$, which infers the output kind of a subexpression by the following exhaustive and straightforward case analysis:

$varRes(for\ \$var\ in\ e_1\ (where\ e_2)?\ return\ e_3) \Rightarrow varRes(e_3)$
$varRes(let\ \$var := e_1\ (where\ e_2)?\ return\ e_3) \Rightarrow varRes(e_3)$
$varRes(if(e_1)\ then\ e_2\ else\ e_3) \Rightarrow varRes(e_2) \cup varRes(e_3)$
$varRes(e_1, \ldots, e_n) \Rightarrow varRes(e_1) \cup \ldots \cup varRes(e_n)$
$varRes(step_1/\ldots/step_2) \Rightarrow \{step_1/\ldots/step_2\}$
$varRes(element\{QName\}\{e\}) \Rightarrow \{element\{QName\}\{e\}\}$
$varRes(\$var) \Rightarrow \{\$var\}$
$varRes(literal) \Rightarrow \{\#\texttt{text}\}$
$varRes(some\ \$var\ in\ e_1\ satisfies\ e_2) \Rightarrow \{\#\texttt{text}\}$
$varRes(every\ \$var\ in\ e_1\ satisfies\ e_2) \Rightarrow \{\#\texttt{text}\}$

Since an XPath expression can be relative to a named variable (i.e., starting with a variable name), the environment will also allow us to keep track of the relationship between variables within a query (e.g., the fact that $\$l$ is bound to $\$i$). For convenience, for the manipulation of the environment we also define a function called $saturate()$, which refines the bindings by making explicit all the XPath navigation.

Example 2.2 *Since we obtain using* $varRes$ *that* $\$l \Rightarrow \$i$ *and* $\$i \Rightarrow \{docB@S3/site/closed\_auctions/closed\_auction\}$, *we refine the information on variable* $\$l$, *using the* $saturate$ *function, as* $\$l \Rightarrow \{\$i, docB@S3/site/closed\_auctions/closed\_auction\}$.

Finally, for a variable $\$var$ and its bound expression $exp_b(\$var)$, the addition of $\$var$ to the environment is achieved by the following statement:

$$Env = +(\$var \Rightarrow (Env.saturate(varRes(exp_b(\$var))))).$$

The following additional functions will be used in the algorithm to access the pre-computed environment:

- $getBind(\$var)$: retrieves from the environment the set of objects associated to the input variable $\$var$.

- $getXPathBind(\$var)$: among the objects to which the variable $\$var$ is bound, it retrieves those corresponding to XPath expressions, if any exist.

3 The rewriting algorithm

Algorithm overview. We give first an overview of our rule-based algorithm, which takes as input an XQuery expression Q and outputs an equivalent simplified XQuery expression Q'.

As the various bound expressions in Q produce intermediate results that may only be partially useful to Q's end result, our algorithm identifies the useful parts in bound expressions and simplifies each of them accordingly. The output is an equivalent query Q' obtained from Q by

substituting each subexpression $exp_b(\$var)$ by a subexpression $exp'_b(\$var)$ that has the advantage of producing only the needed intermediate results.

For a given variable $\$var$, the algorithm retrieves from $exp_r(\$var)$ all the XPath expressions that access the result of $exp_b(\$var)$. This task is performed by the *extractPaths* function. These paths are then used to obtain and project out the irrelevant parts in $exp_b(\$var)$. This is the role of the *ProjectPaths* function. A simpler subexpression $exp'_b(\$var)$, producing only the needed intermediate results, is obtained in this way.

This process is applied recursively, in bottom-up manner, by the *Prune* function over Q. More precisely, for a given variable $\$var$, the pruning is first applied recursively within its bound and return expressions, then it is applied on $\$var$ itself, as described above.

We continue the presentation of the algorithm, starting with the rule-based functions for path analysis (*extractPaths*) and query projection (*ProjectPaths*). We wrap-up the presentation with the *Prune* function, that applies in bottom-up manner the steps for path extraction and projection.

3.1 Path analysis

In this section we present the inference rules that extract for each variable $\$var$ and its return expression $exp_r(\$var)$ the set of paths that navigate through the variable $\$var$. These paths are denoted the *projection paths* of $\$var$. Due to the normalization step, these will be the paths that start with $\$var$ (either explicitly, or via other declared variables).

Similar to [15], in our analysis we will distinguish between two kinds of projection paths: (i) *used paths* and (ii) *returned paths*. The former kind denotes paths for which the descendants of the returned nodes are not necessarily relevant for the end result and no navigation in the subtrees of these nodes is required. Informally, these are paths that bind a variable, following either the in keyword in a for , $every$ or $some$ clauses or the $:=$ keyword in a let clause.

The latter kind denotes paths for which descendants of the nodes reached by the path must be kept in the end result. Paths are by default considered of the returned kind, unless some conditions for the binding kind are verified.

We now present the inference rules for the path extraction function $extractPaths$. The result of a rule application will be two sets of paths, $\mathcal{P}$ and $\mathcal{P}^{\#}$, for the used and returned paths respectively. The $extractPaths$ function takes as input a variable $\$var$ and the XQuery expression $exp_r(\$var)$ that is analyzed, and outputs these two sets of paths.

Literal, empty sequence. Literal expressions and empty sequences do not contain any XPath expressions, so the $extractPaths$ function will return empty sets.

$$\frac{}{Env \vdash extractPaths(\$var, literral) \Rightarrow \emptyset, \emptyset}\ (ep1)$$

$$\frac{}{Env \vdash extractPaths(\$var, ()) \Rightarrow \emptyset, \emptyset}\ (ep2)$$

Sequence, conditional, comparison, element construction. Computing $extractPaths$ for a sequence of two expressions amounts to computing the union of the $\mathcal{P}, \mathcal{P}^{\#}$'s of the subexpressions appearing in the sequence. We extract the paths for a given variable in the same fashion from conditional, arithmetic, logic, comparison and element construction expressions.

$$\frac{\begin{array}{c}Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#}\end{array}}{Env \vdash extractPaths(\$var, (e_1, e_2)) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#}}\ (ep3)$$

$$\frac{\begin{array}{c}Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \\ Env \vdash extractPaths(\$var, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^{\#}\end{array}}{\begin{array}{c}Env \vdash extractPaths(\$var, if(e_1)\ then\ e_2\ else\ e_3) \Rightarrow \\ \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#}\end{array}}\ (ep4)$$

$$\frac{\begin{array}{c}Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#}\end{array}}{\begin{array}{c}Env \vdash extractPaths(\$var, (e_1\ (Op)\ e_2)) \Rightarrow \\ \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#}\end{array}}\ (ep5)$$

$$\frac{Env \vdash extractPaths(\$var, exp) \Rightarrow \mathcal{P}, \mathcal{P}^{\#}}{Env \vdash extractPaths(\$var, element\{QName\}\{exp\}) \Rightarrow \mathcal{P}, \mathcal{P}^{\#}}\ (ep6)$$

Variable reference. The first case is self-explanatory. For the second case, when the variable reference ($\$v$) is different from $\$var$, we obtain from the pre-computed environment the paths

starting by $\$var$ among the items that are associated to $\$v$. These paths will form the set $\mathcal{P}^{\#}$ of returned paths for $\$var$. The set $\mathcal{P}$ of used paths will be empty. Intuitively, this choice is motivated by the fact that these paths must appear in the scope of the variable (i.e. some *where*, *return*, or *satisfies* clause of the query).

$$\frac{}{Env \vdash extractPaths(\$var, \$var) \Rightarrow \emptyset, \{\$var\}}\ (ep7)$$

$$\frac{\$v \neq \$var, \quad Env.getXPathBind(\$v) \Rightarrow B \quad B' = \{p \in B, p = \$var/\ldots\}}{Env \vdash extractPaths(\$var, \$v) \Rightarrow \emptyset, B'}\ (ep8)$$

XPath expression. The first case is self-explanatory. In the second case, when the first step is a variable different from $\$var$, we retrieve from the pre-computed environment the set of the paths associated to $\$v$ that start with $\$var$ (if any). These will be used to create the set of returned paths by substituting the step $\$v$ by each element of the set.

$$\frac{}{Env \vdash extractPaths(\$var, \$var/\ldots/s_n) \Rightarrow \emptyset, \{\$var/\ldots/s_n\}}\ (ep9)$$

$$\frac{Env.getXPathBind(\$v) \Rightarrow B \quad B' = \{p' = p/s_1/\ldots/s_n, p \in B \wedge p = \$var/\ldots\}}{Env \vdash extractPaths(\$var, \$v/s_1/\ldots/s_n) \Rightarrow \emptyset, B'}\ (ep10)$$

FLWR expression, quantifier. The rule will first extract used and returned paths referencing $\$var$ from e_1, e_2 and e_3. The basic approach is quite straightforward: the set of used paths (resp. returned paths) of the entire FLWR block will be obtained as the union of the used (resp. returned) paths of e_1, e_2 and e_3. We use the same logic in order to extract paths from a let or a quantifier expression. After this basic approach, an additional optimization for this step will be also described.

$$\frac{Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \quad Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \quad Env \vdash extractPaths(\$var, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^{\#}}{Env \vdash extractPaths(\$var, for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#}}\ (ep11)$$

$$\frac{Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \quad Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \quad Env \vdash extractPaths(\$var, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^{\#}}{Env \vdash extractPaths(\$var, let\ \$v\ :=\ e_1\ where\ e_2\ return\ e_3) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#}}\ (ep12)$$

$$\frac{Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \quad Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#}}{Env \vdash extractPaths(\$var, some\ \$v\ in\ e_1\ satisfies\ e_2) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{P}_1^{\#} \cup \mathcal{P}_2^{\#}}\ (ep13)$$

An additional optimization that can be applied at this point consists in transforming some of the returned paths coming from e_1 into used paths in the final $\mathcal{P}$ set for $\$var$. This may allow us to simplify even more the query.

This is possible for returned paths p_k for which the variable $\$v$ is bound directly to the result of p_k. When this happens, we can safely conclude that only the nodes selected by p_k will be useful at the higher nesting levels in the query, instead of the entire subtrees rooted at those nodes in the XML document.

Example 3.1 *Consider for instance a query of the form*

$for\ \$var\ in\ \ldots$
$return\ for\ \$v\ in\ (\texttt{<a>}p_1\texttt{}, p_2)$
$\quad return\ \ldots$
with p_1 *and* p_2 *both of the form* $\$var/\ldots$

The variable $\$v$ *will be bound directly to elements produced by the path* p_2*. Although the path is initially a returned one, this suggests that it can be safely considered a used one, thus allowing for more drastic simplifications in the later stages.*

While $\$v$ *is also bound to constructed a elements, we cannot make the link between* $\$v$ *and* $\$p_1$ *, so we adopt the conservative approach of keeping this path as a returned one.*

We obtain the returned paths whose results are directly linked to $\$v$ by using the $getXPathBind$ function and selecting the ones that reference $\$var$. These paths are then removed from $\mathcal{P}_1^{\#}$, and the remaining ones are denoted by $\mathcal{P}_{bis}$. We rely on the following rule:

$$\frac{Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \quad Env.getXPathBind(\$v) \Rightarrow P \quad P' = \{p_k \in P, p_k = \$var/\ldots\}, \mathcal{P}_1^{\#} - P' \Rightarrow \mathcal{P}_{bis}^{\#} \quad Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \quad Env \vdash extractPaths(\$var, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^{\#}}{Env \vdash extractPaths(\$var, for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup P', \mathcal{P}_{bis}^{\#} \cup \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#}}\ (ep14)$$

$$\frac{\begin{array}{c} Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env.getXPathBind(\$v) \Rightarrow P \\ P' = \{p_k \in P, p_k = \$var/\ldots\}, \\ \mathcal{P}_1^{\#} - P' \Rightarrow \mathcal{P}_{bis}^{\#} \\ Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \\ Env \vdash extractPaths(\$var, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^{\#} \end{array}}{\begin{array}{c} Env \vdash extractPaths(\$var, let\ \$v\ :=\ e_1\ where\ e_2\ return \\ e_3) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup P', \mathcal{P}_{bis}^{\#} \cup \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#} \end{array}}\ (ep15)$$

$$\frac{\begin{array}{c} Env \vdash extractPaths(\$var, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^{\#} \\ Env.getXPathBind(\$v) \Rightarrow P \\ P' = \{p_k \in P, p_k = \$var/\ldots\}, \\ \mathcal{P}_1^{\#} - P' \Rightarrow \mathcal{P}_{bis}^{\#} \\ Env \vdash extractPaths(\$var, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^{\#} \end{array}}{\begin{array}{c} Env \vdash extractPaths(\$var, some\ \$v\ in\ e_1\ satisfies\ e_2) \\ \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup P', \mathcal{P}_{bis}^{\#} \cup \mathcal{P}_2^{\#} \end{array}}\ (ep16)$$

Example 3.2 *We give below the outcome of the* $extractPaths$ *function on the variable* $\$j$ *and its return expression in* $Q_{1.2}$*. The prefix indicates the rules that were applied.*

$(ep9)$ $Env \vdash extractPaths(\$j, docD@S_1/site) \Rightarrow \emptyset, \emptyset$

$(ep4, ep9, ep9)$ $Env \vdash extractPaths(\$j, \$j/person = \$k\ldots) \Rightarrow \emptyset, \{\$j/person\}$

$(ep9, ep5)$ $Env \vdash extractPaths(\$j, <common\text{-}auction>\{\$j/open\_auction\}</common\text{-}auction>) \Rightarrow \emptyset, \{\$j/open\_auction\}$

$(ep10)$ $Env \vdash extractPaths(\$j, for\ \$k\ \ldots) \Rightarrow \emptyset, \{\$j/open\_auction, \$j/person\}$

3.2 Path projection

In this section we present the inference rules that, for each variable $\$var$, will project out the useless parts of $exp_b(\$var)$ (the bound expression for $\$var$) based on the paths obtained from $exp_r(\$var)$ (the return expression for $\$var$).

This operation takes as input a set of XPath expressions and an XQuery expression, and returns a new XQuery expression. Intuitively, the new expression is obtained by projecting out any intermediary results that are not in the scope of these paths. For a given path p and an expression exp, the function $projectPaths$ checks the matching between the result of exp and p. Each matched step will indicate a result part that is necessary and must be kept in the query.

We next detail the inference rules that define the projection function $projectPaths$. The function will take the two sets of paths produced by the path analysis step (i.e., a set of used paths and a set of returned paths).

Literal, comparison, quantifier. When matching a literal expression with a path p the only case that returns a non empty result is when p is equal either to $text()$ or to $\$var$ (which returns any expression on which it is matched). In any other case, we can conclude that the result of p on the literal is empty.

We deal in similar manner with quantifier, logical and comparison expressions, or arithmetic expressions, as they return numeric or boolean literals.

$$\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^{\#}), p = (\$var \mid text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, literal) \Rightarrow literal}\ (pp1)$$

$$\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^{\#}),\ p \neq (\$var \mid text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, literal) \Rightarrow ()}\ (pp2)$$

$$\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^{\#}), p = (\$var \mid text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_1\ Op^2\ e_2) \Rightarrow e_1\ Op\ e_2}\ (pp3)$$

$$\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^{\#}),\ p \neq (\$var \mid text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_1\ Op\ e_2) \Rightarrow ()}\ (pp4)$$

$$\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^{\#}), p = (\$var \mid text())}{\begin{array}{c} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, some\ \$var\ in\ e_1\ satisfies\ e_2) \\ \Rightarrow some\ \$var\ in\ e_1\ satisfies\ e_2 \end{array}}\ (pp5)$$

$$\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^{\#}), p \neq (\$var \mid text())}{\begin{array}{c} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, some\ \$var\ in\ e_1\ satisfies\ e_2) \\ \Rightarrow () \end{array}}\ (pp6)$$

Sequence. For a sequence of two expressions e_1, e_2, the result of the $projectPaths$ function will simply be the sequence obtained by applying $projectsPaths$ on each e_1 and e_2 individually.

$$\frac{\begin{array}{c} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_1) \Rightarrow e_1' \\ Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_2) \Rightarrow e_2' \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, (e_1, e_2)) \Rightarrow e_1', e_2'}\ (pp7)$$

Variable reference. Evaluating a path on a variable amounts to evaluating this path on the elements that are bound to the variable. The result of the projection can be the variable itself, if it is bound to at least one object (in the precomputed environment) for which the matching does not lead to an empty result. The variable is projected out if none of the objects to which it is bound can be matched by some path of $\mathcal{P}$ or $\mathcal{P}^{\#}$.

$$\frac{(Env.getBind(\$var) \Rightarrow \{o_1, \ldots, o_n\}) \quad \exists i, 1 \leq i \leq n, projectPaths(\mathcal{P}, \mathcal{P}^{\#}, o_i) \Rightarrow o_i' \; s.t. \; o_i' \neq ()}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, \$var) \Rightarrow \$var} \; (pp8)$$

$$\frac{(Env.getBind(\$var) \Rightarrow \{o_1, \ldots, o_n\}) \quad \forall i, 1 \leq i \leq n, projectPaths(\mathcal{P}, \mathcal{P}^{\#}, o_i) \Rightarrow ()}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, \$var) \Rightarrow ()} \; (pp9)$$

XPath expression. Note that the elements returned by an XPath expression $s_1/\ldots/s_n$ are defined by the last step of the path s_n. If $s_n = text()$, this means that the path will return a literal, so the only way to retrieve this literal is to have in $\mathcal{P} \cup \mathcal{P}^{\#}$ at least one path p which corresponds to the text-test $text()$ or to a variable reference $\$var$.

If $s_n \neq text()$, hence the nodes returned by $s_1/\ldots/s_n$ are element nodes, it is sufficient to have one path in $\mathcal{P} \cup \mathcal{P}^{\#}$ that starts with $\$var$, $*$ (which corresponds to any element test) or s_n (i.e. the first step of p corresponds to the element returned by $s_1/\ldots/s_n$). We compare only with the first step in this case, because we do not have at this level enough information about the descendants of s_n.

If $s_n = *$, this means that the path returns any element node, so we can retrieve those elements with any path except those of the kind $p = text()$.

In the remaining cases, the application of the paths will always return the empty sequence $()$, and in this case we replace the path $s_1/\ldots/s_n$ by $()$.

$$\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^{\#}), (head^{3}(p) = (*|s_n|\$var) \wedge s_n \neq text()) \vee (s_n = \text{"}*\text{"} \wedge p \neq text()) \vee (p = text()|\$var \wedge s_n = text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, s_1/\ldots/s_n) \Rightarrow s_1/\ldots/s_n} \; (pp10)$$

[3] $head(p)$ is a function that retrieves from a path p its first step.

$$\frac{otherwise}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, s_1/\ldots/s_n) \Rightarrow ()} \; (pp11)$$

FLWR expression. Following the XQuery semantics, the result of a for expression is computed by the subexpression following the $return$ keyword. Hence applying a path on a for expression amounts to applying it on the $return$ subexpression, even if this for expression contains a $where$ clause[4]. The result of this step will be the for expression with a new $return$ block. This logic is clear in the first rule. Moreover, when the projection on the $return$ expression generates an empty sequence, the entire for expression can be a projected out (rule $pp13$). Similar transformations are performed on let expressions.

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_3) \Rightarrow e_3'}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, for \; \$var \; in \; e_1 \; where \; e_2 \; return \; e_3) \Rightarrow for \; \$var \; in \; e_1 \; where \; e_2 \; return \; e_3'} \; (pp12)$$

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_3) \Rightarrow ()}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, for \; \$var \; in \; e_1 \; where \; e_2 \; return \; e_3) \Rightarrow ()} \; (pp13)$$

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_3) \Rightarrow e_3'}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, let \; \$var \; := \; e_1 \; where \; e_2 \; return \; e_3) \Rightarrow let \; \$var \; := \; e_1 \; where \; e_2 \; return \; e_3'} \; (pp14)$$

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_3) \Rightarrow ()}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, let \; \$var \; := \; e_1 \; where \; e_2 \; return \; e_3) \Rightarrow ()} \; (pp15)$$

Conditional. When applying a set of paths on a conditional expression, we have the choice to apply them on either the $true$ branch or the $false$ branch. Since both branches may be followed at runtime, we apply the paths on both. The result is a new conditional expression with the same condition expression e_1, having potentially simplified expressions in the $true$ and $false$ branches.

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_2) \Rightarrow e_2' \quad Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_3) \Rightarrow e_3'}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, if \; (e_1) \; then \; e_2 \; else \; e_3) \Rightarrow if \; (e_1) \; then \; e_2' \; else \; e_3'} \; (pp16)$$

[4] Note that the role of a $where$ clause in a $FLWR$ expression is merely to limit the size of the result according to some condition.

Element construction. The projection of element construction expressions has several cases. In order to simplify the presentation, we assume that the application of the rules is attempted according to the order in which they are presented below :

Rule $pp17$: if $\mathcal{P}^{\#}$ contains a path p that is either the variable reference $\$var$ or a one-step path whose label is identic to that of the constructed element, then $projectPaths$ returns that element.

Rule $pp18$: if there is no path in both used and returned sets that matches with the constructed element, then $projectPaths$ returns the empty sequence.

Rule $pp19$: if a) $\mathcal{P}$ does not contain the paths $\$var$ or $QName$, and b) $\mathcal{P} \cup \mathcal{P}^{\#}$ do not contain any path that can follow the content of the constructed element, even if there are paths whose first step can match with the element name $QName$, then $projectPaths$ returns the empty sequence.

Rule $pp20$: if there exists a used path in $\mathcal{P}$ that matches with the element (i.e., it is either $\$var$ or $QName$), and no other path starts by $\$var$ or $QName$, then $projectPaths$ returns the element $QName$ with an empty content.

Rule $pp21$: in the remaining case, when there exists some paths p (of either kind) starting with $\$var$ or $QName$. In this case, $projectPaths$ will match this step and continue with the remaining steps (other used and returned paths) over the content of the constructed element. It creates new used and returned path sets, $\mathcal{P}'$ and $\mathcal{P}'^{\#}$ from $\mathcal{P}$ and $\mathcal{P}^{\#}$ respectively, by keeping only the paths whose first step matches with the name of the constructed element. The sets $\mathcal{P}'$ and $\mathcal{P}'^{\#}$ are applied on the content of the element. Note that the element will be returned in this case, even if the result of applying the paths over e_2 returns the empty sequence.

$$\frac{\exists p \in \mathcal{P}^{\#},\ p = (QName \mid \$var)}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e\}}\ (pp17)$$

$$\frac{\forall p \in \mathcal{P} \cup \mathcal{P}^{\#}, head(p) \neq (QName \mid \$var)}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow ()}\ (pp18)$$

$$\frac{\begin{array}{c}\nexists p \in \mathcal{P},\ p = (QName \mid \$var) \\ \mathcal{P}' = \{p', [QName \mid \$var]/p' \in \mathcal{P}\} \\ \mathcal{P}'^{\#} = \{p', [QName \mid \$var]/p' \in \mathcal{P}^{\#}\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^{\#}, e) \Rightarrow ()\end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow ()}\ (pp19)$$

$$\frac{\begin{array}{c}(\exists p \in \mathcal{P},\ p = (QName \mid \$var)) \wedge \\ (\forall \overline{p} \in (\mathcal{P} \cup \mathcal{P}^{\#}) - \{p\}, head(\overline{p}) \neq (QName \mid \$var))\end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{\}}\ (pp20)$$

$$\frac{\begin{array}{c}\exists p \in \mathcal{P} \cup \mathcal{P}^{\#}, head(p) = (QName \mid \$var) \\ \mathcal{P}' = \{p', [QName \mid \$var]/p' \in \mathcal{P}\} \\ \mathcal{P}'^{\#} = \{p', [QName \mid \$var]/p' \in \mathcal{P}^{\#}\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^{\#}, e) \Rightarrow e'\end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e'\}}\ (pp21)$$

Empty sequence. When the input expression is an empty sequence, $projectPaths$ returns the empty sequence whatever $\mathcal{P}, \mathcal{P}^{\#}$ contain.

$$\frac{}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, ()) \Rightarrow ()}\ (pp22)$$

3.3 Pruning process

The pruning process is applied recursively, in a bottom-up manner, by the $Prune$ function over Q. For each variable $\$var$ in Q, the pruning is applied recursively within its bound and return expressions, before it is applied to $\$var$ itself, as shown in the following inference rules.

Literal, variable reference, XPath expression and empty sequence. When the input expression Q is a literal, variable reference, XPath expression or just an empty sequence expression, no variable declarations can be found. In this case, the output expression is the same as the input.

$$\frac{}{Env \vdash Prune(literal) \Rightarrow Literal}\ (p1)$$

$$\frac{}{Env \vdash Prune(\$var) \Rightarrow \$var}\ (p2)$$

$$\frac{}{Env \vdash Prune(s_1/\ldots/s_n) \Rightarrow s_1/\ldots/s_n}\ (p3)$$

$$\frac{}{Env \vdash Prune(()) \Rightarrow ()}\ (p4)$$

Sequence, comparison, element construction. The pruning of a sequence of subexpressions returns as a result the sequence of the pruned subexpressions. We use the same logic to prune arithmetic, logic comparison and constructor expressions.

$$\frac{Env \vdash Prune(e_1) \Rightarrow e_1' \quad Env \vdash Prune(e_2) \Rightarrow e_2'}{Env \vdash Prune(e_1, e_2) \Rightarrow e_1', e_2'} (p5)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow e_1' \quad Env \vdash Prune(e_2) \Rightarrow e_2'}{Env \vdash Prune(e_1\ Op\ e_2) \Rightarrow e_1'\ Op\ e_2'} (p6)$$

$$\frac{Env \vdash Prune(e) \Rightarrow e'}{\begin{array}{c}Env \vdash Prune(element\{QName\}\{e\}) \Rightarrow \\ element\{QName\}\{e'\}\end{array}} (p7)$$

Conditional. In this case, the pruning operation is propagated to the if subexpression and to the $then$ and $else$ returned subexpressions. The output expression is obtained by substituting the subexpressions by their pruning result.

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env \vdash Prune(e_2) \Rightarrow e_2' \quad Env \vdash Prune(e_3) \Rightarrow e_3'\end{array}}{\begin{array}{c}Env \vdash Prune(if(e_1)\ then\ e_2\ else\ e_3) \Rightarrow \\ if(e_1')\ then\ e_2'\ else\ e_3'\end{array}} (p8)$$

FLWR expressions, quantifier. When the input Q is a FLWR expression, the pruning operation is first applied to the bound expression of the variable $\$var$ declared in the for clause. Then, the variable $\$var$ is added to the environment Env with its bound objects computed by $varRes$. The $saturate$ function is applied to refine the bindings stored in the environment (see Section 2).

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1')))) \\ Env \vdash Prune(e_2) \Rightarrow e_2' \\ Env \vdash extractPaths(\$var, e_2') \Rightarrow \mathcal{P}, \mathcal{P}^{\#} \\ Env \vdash projectPaths(\mathcal{P} \cup \{\$var\}, \mathcal{P}^{\#}, e_1') \Rightarrow e_1''\end{array}}{\begin{array}{c}Env \vdash Prune(for\ \$var\ in\ e_1\ return\ e_2) \Rightarrow \\ for\ \$var\ in\ e_1''\ return\ e_2'\end{array}} (p9)$$

Next, the pruning operation is applied to the return subexpression e_2, and the $extractPaths$ function is called to extract the paths referencing $\$var$ from the obtained result e_2'. The $projectPaths$ function is used to apply the extracted paths ($\mathcal{P} \cup \{\$var\}$, $\mathcal{P}^{\#}$) on the pruned bound expression e_1' in order to remove its useless parts. The obtained results e_1'' and e_2' will replace the subexpressions e_1 and e_2 respectively in the output result. Here we add the binding path $\$var$ to ensure that the number of for iterations remains the same.

The pruning of for expression can lead to the following interesting special cases:

Case 1:

$$\frac{Env \vdash Prune(e_1) \Rightarrow ()}{Env \vdash Prune(for\ \$var\ in\ e_1\ return\ e_2) \Rightarrow ()} (p10)$$

In this case the pruning of the bound expression generates an empty sequence, which means that the number of the iterations of the for is equal to 0, so the whole for expression generates an empty sequence. For this reason the pruning generates an empty sequence.

Case 2:

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ RootPath(e_1') \Rightarrow PathB_1 \\ Env = Env + (\$var \Rightarrow PathB_1) \quad Env = Env.saturate(\$var) \\ Env \vdash Prune(e_2) \Rightarrow ()\end{array}}{Env \vdash Prune(for\ \$var\ in\ e_1\ return\ e_2) \Rightarrow ()} (p11)$$

This case correspond to the situation in which the pruning of the return expression e_2 leads to the empty sequence $()$. This means that whatever is the number of the for iterations, the result is empty, which is equivalent to counting over an empty sequence. This is why, in a such cases the pruning of the whole for expression gives the empty sequence $()$.

We note that there is a variant of the for pruning rules and its special cases with a $where$ clause. We do the same thing by considering the $where$ clause as a return expression: we prune its condition expression, and we extract from it the paths referencing $\$var$. These path are added to those of the return expression, before being used to prune $\$var$'s bound expression.

$$\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1')))) \\ Env \vdash Prune(e_2) \Rightarrow e_2' \\ Env \vdash Prune(e_3) \Rightarrow e_3' \\ Env \vdash extractPaths(\$var, e_2') \Rightarrow \mathcal{P}_2, {\mathcal{P}^{\#}}_2 \\ Env \vdash extractPaths(\$var, e_3') \Rightarrow \mathcal{P}_3, {\mathcal{P}^{\#}}_3\end{array}$$

$$\frac{Env \vdash projectPaths(\mathcal{P}_2 \cup \mathcal{P}_3 \cup \{\$var\}, \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#}, e_1') \Rightarrow e_1''}{Env \vdash Prune(for\ \$var\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow for\ \$var\ in\ e_1''\ where\ e_2'\ return\ e_3'}\ (12)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow ()}{Env \vdash Prune(for\ \$var\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p13)$$

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1'))) \\ Env \vdash Prune(e_2) \Rightarrow e_2' \\ Env \vdash Prune(e_3) \Rightarrow ()\end{array}}{Env \vdash Prune(for\ \$var\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p14)$$

When pruning a for expression with $where$ clause an additional special case arise. It happens when the pruning of the $where$ clause leads to empty sequence, then the whole for expression is reduced to the empty sequence as a condition with empty sequence in XQuery is considered the $false$ value.

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1'))) \\ Env \vdash Prune(e_2) \Rightarrow ()\end{array}}{Env \vdash Prune(for\ \$var\ in\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p15)$$

For let expressions we apply similar transformations. The only difference is that we do not add the path $\$var$ because the return of a let is executed exactly one time whatever the expression e_1 is.

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1'))) \\ Env \vdash Prune(e_2) \Rightarrow e_2' \\ Env \vdash extractPaths(\$var, e_2') \Rightarrow \mathcal{P}, \mathcal{P}^{\#} \\ Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_1') \Rightarrow e_1''\end{array}}{Env \vdash Prune(let\ \$var\ :=\ e_1\ return\ e_2) \Rightarrow let\ \$var\ :=\ e_1''\ return\ e_2'}\ (p16)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow ()}{Env \vdash Prune(let\ \$var\ :=\ e_1\ return\ e_2) \Rightarrow ()}\ (p17)$$

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ RootPath(e_1') \Rightarrow PathB_1 \\ Env = Env + (\$var \Rightarrow PathB_1)\quad Env = Env.saturate(\$var) \\ Env \vdash Prune(e_2) \Rightarrow ()\end{array}}{Env \vdash Prune(let\ \$var\ :=\ e_1\ return\ e_2) \Rightarrow ()}\ (p18)$$

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1'))) \\ Env \vdash Prune(e_2) \Rightarrow e_2' \\ Env \vdash Prune(e_3) \Rightarrow e_3' \\ Env \vdash extractPaths(\$var, e_2') \Rightarrow \mathcal{P}_2, \mathcal{P}^{\#}{}_2 \\ Env \vdash extractPaths(\$var, e_3') \Rightarrow \mathcal{P}_3, \mathcal{P}^{\#}{}_3 \\ Env \vdash projectPaths(\mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}_2^{\#} \cup \mathcal{P}_3^{\#}, e_1') \Rightarrow e_1''\end{array}}{Env \vdash Prune(let\ \$var\ :=\ e_1\ where\ e_2\ return\ e_3) \Rightarrow let\ \$var\ :=\ e_1''\ where\ e_2'\ return\ e_3'}\ (19)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow ()}{Env \vdash Prune(let\ \$var\ :=\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p20)$$

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1'))) \\ Env \vdash Prune(e_2) \Rightarrow e_2' \\ Env \vdash Prune(e_3) \Rightarrow ()\end{array}}{Env \vdash Prune(let\ \$var\ :=\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p21)$$

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1'))) \\ Env \vdash Prune(e_2) \Rightarrow ()\end{array}}{Env \vdash Prune(let\ \$var\ :=\ e_1\ where\ e_2\ return\ e_3) \Rightarrow ()}\ (p22)$$

For the quantifier, we use the same logic, but for the special cases, instead of returning empty sequence, we return the boolean value as a quantifier returns a logic value. If the pruning of the binding expression e_1 leads to an empty sequence, the result of the quantifier is obviously $false()$. If the pruning of the condition leads to an empty sequence, there is no element that satisfies it, so the quantifier will return $false()$ as well.

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$var \Rightarrow (Env.saturate(varRes(e_1'))) \\ Env \vdash Prune(e_2) \Rightarrow e_2' \\ Env \vdash extractPaths(\$var, e_2') \Rightarrow \mathcal{P}, \mathcal{P}^{\#} \\ Env \vdash projectPaths(\mathcal{P} \cup \{\$var\}, \mathcal{P}^{\#}, e_1') \Rightarrow e_1''\end{array}}{Env \vdash Prune(some\ \$var\ in\ e_1\ satisfies\ e_2) \Rightarrow some\ \$var\ in\ e_1''\ satisfies\ e_2'}\ (p23)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow ()}{Env \vdash Prune(some\ \$var\ in\ e_1\ satisfies\ e_2) \Rightarrow false()}\ (p24)$$

$$\frac{\begin{array}{c}Env \vdash Prune(e_1) \Rightarrow e_1' \\ RootPath(e_1') \Rightarrow PathB_1 \\ Env = Env + (\$var \Rightarrow PathB_1)\quad Env = Env.saturate(\$var) \\ Env \vdash Prune(e_2) \Rightarrow ()\end{array}}{Env \vdash Prune(some\ \$var\ in\ e_1\ satisfies\ e_2) \Rightarrow false()}\ (p25)$$

3.4 Correctness

We prove in this section that our rule-based algorithm is correct, i.e., its input and output queries are *equivalent* (the evaluation of the output query yields the same result as the evaluation of the initial query). The algorithm described in Section 3.3 verifies the following theorem.

Theorem 3.1 *[Equivalence] Let q be an XQuery expression, let $I = \{d_1, \ldots, d_k\}$ be the set of XML documents used in q, let Env be the evaluation environment, and let q' be the XQuery expression obtained from the pruning of q (i.e., $Env \vdash Prune(q) \Rightarrow q'$). Then, the results of q and q' over I are equal: $q_{[Env]}(I) = q'_{[Env]}(I)$, where "=" denotes the deep equality defined for XML values in [13].*

A proof for this theorem can be constructed by induction on the inference rule for each expression.

Proof details

As it is shown in Section 3.3, when the input query q is a literal value, a variable name, an XPath expression or an empty sequence (rules $p1$ to $p4$), the pruning process will produce an output query q' identical to its input q. So, $\forall I, \forall Env, q_{[Env]}(I) = q'_{[Env]}(I)\square$.

When the input query q is a conditional expression, a non empty sequence, a comparison expression, an arithmetic or a logical expression (pruning rules $p5$ to $p8$ in Section 3.3), then the pruning process is simply applied recursively to the subexpressions of q before substituting them by the obtained pruned expressions. Assume that the pruned subexpressions are equivalent to the initial ones, then according to the semantics of XQuery[7] the output query q' is equivalent to q (i.e., $\forall I, \forall Env, q_{[Env]}(I) = q'_{[Env]}(I)$).$\square$

For instance, consider a conditional expression $q = if(e_1)\ then\ e_2\ else\ e_3$, the pruning process produces $q' = if(e'_1)\ then\ e'_2\ else\ e'_3$, where $Env \vdash Prune(e_1) \Rightarrow e'_1, Env \vdash Prune(e_2) \Rightarrow e'_2$, and $Env \vdash Prune(e_3) \Rightarrow e'_3$. Assume that e_1 is equivalent to e'_1, e_2 is equivalent to e'_2 and e_3 is equivalent to e'_3 (i.e., $\forall\ I,\ \forall\ Env, e_{1[Env]}(I) = e'_{1[Env]}(I), e_{2[Env]}(I) = e'_{2[Env]}(I)$ and $e_{3[Env]}(I) = e'_{3[Env]}(I)$). Then, the expressions $if(e_1)\ then\ e_2\ else\ e_3$ and $if(e'_1)\ then\ e'_2\ else\ e'_3$ are equivalent, i.e., $\forall\ I,\ \forall\ Env\ q_{[Env]}(I) = q'_{[Env]}(I)$

$FLWR$ **expressions.** When the input query q is a FLWR expression (pruning rules $p9$ to $p22$ in Section 3.3), the pruning process can be summarized in three main steps : (1) the pruning is applied recursively to the subexpressions of q before substituting them by the obtained pruned expressions (e'_1, e'_2 and e'_3), (2) $extractPaths()$ is called to extract from the *return* and *where* subexpressions (e'_2 and e'_3) the *used* and *return paths* (P and $P^{\#}$), and (3) $projectPaths()$ is called to apply the extracted paths on the bound subexpression (e'_1) in order to minimize it. The first step preserves the equivalence: Assume that e_1 is equivalent to e'_1, e_2 is equivalent to e'_2 and e_3 is equivalent to e'_3, then $for\ \$var\ in\ e_1\ [where\ e_2]\ return\ e_3$ is equivalent to $for\ \$var\ in\ e'_1\ [where\ e'_2]\ return\ e'_3$ and it is the same for let expressions $let\ \$var\ :=\ e_1\ [where\ e_2]\ return\ e_3$ and $let\ \$var\ :=\ e'_1\ [where\ e'_2]\ return\ e'_3$. Then, to prove the equivalence between the input q and the output query q' we need the following result: the projected subexpression e''_1, obtained by applying the *used* and *return paths* on the bound subexpression e'_1, generates all the nodes that are part of the evaluation of e'_2 and e'_3 (this is similar to the *Return Paths lemma* of Simeon et al. [15]). More precisely, we need to prove the following two properties.

Lemma 3.1 *[Paths Extraction] Let e be an XQuery expression and $\$var$ be a variable name. Then, the sets of paths $\mathcal{P}$ and $\mathcal{P}^{\#}$ extracted from e ($Env \vdash extractPaths(\$var, e) \Rightarrow \mathcal{P}, \mathcal{P}^{\#}$) satisfy the following properties:*

- *$\mathcal{P}$ and $\mathcal{P}^{\#}$ contain all the paths in e that reference $\$var$, and nothing else.*
- *only* used paths *are contained in $\mathcal{P}$.*

Lemma 3.2 *[Paths Projection] Let e_1 be an XQuery expression, let $\mathcal{P}$ be a set of* used paths *and $\mathcal{P}^{\#}$ a set of* return paths. *Then, the XQuery expression e_2, obtained by the application of $\mathcal{P}$ and $\mathcal{P}^{\#}$ paths on e_1 ($Env \vdash ProjectPaths(\mathcal{P}, \mathcal{P}^{\#}, e_1) \Rightarrow e_2$), satisfies the following properties:*

- $\forall p \in \mathcal{P}$: root$(eval(p, e_2))$ = root$(eval(p, e_1))$
- $\forall p \in \mathcal{P}^{\#} : eval(p, e_2) = eval(p, e_1)$

where $root$ function retrieves the root nodes of its input XML data, and function $eval$ is defined as follows:

Definition 3.1 (eval) *Let q be an XQuery expression, and p an XPath expression. Then, $eval(p, q)$ denotes the following XQuery expression:*

- *q/p_1, if $p = \$v/p_1$ (i.e., p starts with a variable reference);*
- *$q/self :: p$, otherwise.*

The following properties of function $eval$ are useful in the rest of this section.

1. $eval(p, (e_1, e_2)) = eval(p, e_1), eval(p, e_2)$
2. $eval([QName|\$var]/p, <QName>\{exp\}</QName> = eval(p, exp)$.

The two lemmas presented above can be proven by induction over each expression. The proof of lemma 3.1 is straightforward, since the distinction between *used paths* and *return paths* is clear (see, Section 3.1). For space reason, we where not able to include it in this paper. We chose to detail the proof of Paths Projection lemma 3.2, which is central in our algorithm.

Proof of lemma 3.2

First, for the inference rules who state that the output of $projectPaths$ is equal to its input (rules $pp1$, $pp3$, $pp5$, $pp8$, $pp10$, $pp17$ and $pp22$ in Section 3.2), the proof is evident ($e_2 = e_1$ in this case). For the rules where the output expression of $projectPaths$ is different from its input, the proof details are given in the following.

Now we give the proof for the cases where the output is different from the input.

For literal values, quantifiers, arithmetic expressions, logical and comparison expressions (rules $pp2$, $pp4$ and $pp6$), when the output expression is different from the input one, it is an empty sequence ($Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^{\#}, e) \Rightarrow ()$, where $e = litteral$, $e = e_1\ Op\ e_2$ or $e = some\ \$var\ in\ e_1\ satisfies\ e_2$). To prove the lemma in this case, we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p, ()) = eval(p, e)$
- $\forall p \in \mathcal{P}, root(eval(p, ())) = root(eval(p, e))$

where $e = litteral$, $e = some\ \$var\ in\ e_1\ satisfies\ e_2$ or $e = e_1\ Op\ e_2$.

For the considered rules, the judgement holds when the input paths are all different from a simple variable name $\$var$ or a text-test kind $text()$(see the premise of the rules). In this case, and according to the semantics of XQuery[7], $eval(p, e)$ will return an empty sequence. Since $eval(p, e) = ()$ and $eval(p, ()) = ()$, the properties are satisfied.

Variable reference (rule $pp9$). Here, we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p, ()) = eval(p, \$var)$
- $\forall p \in \mathcal{P}, root(eval(p, ())) = root(eval(p, \$var))$

For the considered rule, the judgement holds when the nodes generated by the subexpression bound to $\$var$ are not part of the evaluation of the input paths. In other words (see the premise), the judgement holds when $\$var$ is bound to a sequence of objects $\{o_1, \ldots, o_n\}$ in the considered environment, and when the application of the input paths on these objects returns an empty sequence ($\forall i, 1 \leq i \leq n, projectPaths(\mathcal{P}, \mathcal{P}^{\#}, o_i) \Rightarrow ()$).

According to the XQuery semantics, the evaluation of a path on a variable $\$v$ yields to the union of the evaluations of this path on each object bound to $\$v$. In our case, all the evaluations lead to an empty sequence. So their union is an empty sequence too. Since $eval(p, \$var) = ()$ and $eval(p, ()) = ()$, the properties are satisfied.

XPath expression (rule $pp11$). Here, we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p, ()) = eval(p, s_1/\ldots/s_n)$
- $\forall p \in \mathcal{P}, root(eval(p, ())) = root(eval(p, s_1/\ldots/s_n))$

According to the semantics of XQuery, the last step s_n of a path $s = s_1/\ldots/s_n$ determines the nodes result of s. In the case of paths composition, between $p = p_1/\ldots/p_m$ and $s = s_1/\ldots/s_m$ (s/p in short), the result is an empty

sequence if no matching is possible between the first step of path p and the last step of path s. For instance, a matching is possible between a/b and $b/c/d$ or between a/b and $*/c/d$ and it is not possible between a/b and c/d or between $a/tex()$ and $b/c/d$. The cases where a matching is possible are given in the premise of rule $pp10$, otherwise (rule $pp11$) no matching is possible. Since no matching is possible, $eval(p, s_1/\ldots/s_n)$ returns an empty sequence and the properties are satisfied.

Sequence (rule $pp7$). we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p,(e'_1,e'_2)) = eval(p,(e_1,e_2))$
- $\forall p \in \mathcal{P}, root(eval(p,(e'_1,e'_2))) = root(eval(p,(e_1,e_2)))$

In the premise of the considered rule, we assume that:

- $\forall p \in \mathcal{P}^{\#}$:
  $eval(p,e'_1) = eval(p,e_1)$ and
  $eval(p,e'_2) = eval(p,e_2)$
- $\forall p \in \mathcal{P}$:
  $root(eval(p,e'_1)) = root(eval(p,e_1))$ and
  $root(eval(p,e'_2)) = root(eval(p,e_2))$

Then, we can infer from the following that the properties are satisfied:

- $\forall p \in \mathcal{P}^{\#}$
  $eval(p,(e'_1,e'_2))$
  $= (eval(p,e'_1), eval(p,e'_2))$ - - Property 1 of $eval$
  $= (eval(p,e_1), eval(p,e_2))$ - - Inference Hypothesis (I. H.)
  $= eval(p,(e_1,e_2))$ - - Prop. 1 $eval$
- $\forall p \in \mathcal{P}$
  $root(eval(p,(e'_1,e'_2)))$
  $= root(eval(p,e'_1), eval(p,e'_2))$ - - Prop. 1 $eval$
  $= (root(eval(p,e'_1)), root(eval(p,e'_2)))$ - - Prop. $root$
  $= (root(eval(p,e_1)), root(eval(p,e_2)))$ - - I. H.
  $= root(eval(p,e_1), eval(p,e_2))$ - - Prop. $root$
  $= root(eval(p,(e_1,e_2)))$ - - Prop. 1 $eval$
  □

Conditional (rule $pp16$). Here, we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p,(if(e_1)\ then\ e'_2\ else\ e'_3)) = eval(p,(if(e_1)\ then\ e_2\ else\ e_3))$
- $\forall p \in \mathcal{P}, root(eval(p,(if(e_1)\ then\ e'_2\ else\ e'_3))) = root(eval(p,(if(e_1)\ then\ e_2\ else\ e_3)))$

In the premise of the considered rule, we assume that:

- $\forall p \in \mathcal{P}^{\#}$:
  $eval(p,e'_2) = eval(p,e_2)$ and
  $eval(p,e'_3) = eval(p,e_3)$
- $\forall p \in \mathcal{P}$:
  $root(eval(p,e'_2)) = root(eval(p,e_2))$ and
  $root(eval(p,e'_3)) = root(eval(p,e_3))$

The XQuery semantics defines the evaluation of a path on a conditional expression as the evaluation of that path on the result of the $then$ or $else$ subexpression. So, we can infer from the following that the properties of the lemma are satisfied:

- $\forall p \in \mathcal{P}^{\#}$
  $eval(p,(if(e_1)\ then\ e'_2\ else\ e'_3))$
  $= if(e_1)\ then\ eval(p,e'_2)\ else\ eval(p,e'_3)$ - - XQuery
  $= if(e_1)\ then\ eval(p,e_2)\ else\ eval(p,e_3)$ - - I. H.
  $= eval(p,(if(e_1)\ then\ e_2\ else\ e_3))$ - - XQuery
- $\forall p \in \mathcal{P}$
  $root(eval(p,(if(e_1)\ then\ e'_2\ else\ e'_3)))$
  $= root(if(e_1)\ then\ eval(p,e'_2)\ else\ eval(p,e'_3))$ - - XQuery
  $= if(e_1)\ then\ root(eval(p,e'_2))$
  $\quad else\ root(eval(p,e'_3))$ - - Prop. $root$
  $= if(e_1)\ then\ root(eval(p,e_2))$
  $\quad else\ root(eval(p,e_3))$ - - I. H.
  $= root(if(e_1)\ then\ eval(p,e_2)$
  $\quad else\ eval(p,e_3))$ - - Prop. $root$
  $= root(eval(p,(if(e_1)\ then\ e_2\ else\ e_3)))$ - - XQuery
  □

FLWR expressions (rules $pp12$ to $pp15$). We give here the proof details for for expressions (rules $pp12$ and $pp13$). The proof details for let expressions (rules $pp14$ and $pp15$) can be easily deduced using the same reasoning.

Rules $pp12$ ***and*** $pp14$: we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p,(for\ \$v\ in\ e_1\ where\ e_2\ return\ e'_3)) = eval(p,(for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3))$
- $\forall p \in \mathcal{P}, root(eval(p,(for\ \$v\ in\ e_1\ where\ e_2\ return\ e'_3))) = root(eval(p,(for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3)))$

In the premise of the considered rule, we assume that:

- $\forall p \in \mathcal{P}^{\#} : eval(p,e'_3) = eval(p,e_3)$
- $\forall p \in \mathcal{P} : root(eval(p,e'_3)) = root(eval(p,e_3))$

The XQuery semantics defines the evaluation of a path on a FLWR expression as the evaluation of that path on the result of its $return$ subexpression. So, we can infer from the following that the properties of the lemma are satisfied:

- $\forall p \in \mathcal{P}^{\#}$
  $eval(p, (for\ \$v\ in\ e_1\ where\ e_2\ return\ e'_3))$
  $= for\ \$v\ in\ e_1\ where\ e_2\ return\ eval(p, e'_3)$ - - XQuery
  $= for\ \$v\ in\ e_1\ where\ e_2\ return\ eval(p, e_3)$ - - I. H.
  $= eval(p, (for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3))$ - - XQuery

- $\forall p \in \mathcal{P}$
  $root(eval(p, (for\ \$v\ in\ e_1\ where\ e_2\ return\ e'_3)))$
  $= root(for\ \$v\ in\ e_1\ where\ e_2\ return\ eval(p, e'_3))$
  $= for\ \$v\ in\ e_1\ where\ e_2$
  $return\ root(eval(p, e'_3))$ - - Prop. $root$
  $= for\ \$v\ in\ e_1\ where\ e_2\ return\ root(eval(p, e_3))$ - - I. H.
  $= root(for\ \$v\ in\ e_1\ where\ e_2$
  $return\ eval(p, e_3))$ - - Prop. $root$
  $= root(eval(p, (for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3)))$
  □

Rules $pp13$ ***and*** $pp15$: we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p, ()) = eval(p, (for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3))$

- $\forall p \in \mathcal{P}, root(eval(p, ())) = root(eval(p, (for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3)))$

In the premise, we assume that:

- $\forall p \in \mathcal{P}^{\#} : eval(p, ()) = eval(p, e_3)$

- $\forall p \in \mathcal{P} : root(eval(p, ())) = root(eval(p, e_3))$

The following shows that the properties of the lemma are satisfied:

- $\forall p \in \mathcal{P}^{\#}$
  $eval(p, ()) = ()$
  $= for\ \$v\ in\ e_1\ where\ e_2\ return\ ()$ - -XQuery
  $= for\ \$v\ in\ e_1\ where\ e_2\ return\ eval(p, ())$ - -XQuery
  $= for\ \$v\ in\ e_1\ where\ e_2\ return\ eval(p, e_3)$ - - I. H.
  $= eval(p, (for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3))$ - -XQuery

- $\forall p \in \mathcal{P}$
  $root(eval(p, ()))$
  $= root(for\ \$v\ in\ e_1\ where\ e_2\ return\ eval(p, ()))$
  $= for\ \$v\ in\ e_1\ where\ e_2$
  $return\ root(eval(p, ()))$ - - Prop. $root$
  $= for\ \$v\ in\ e_1\ where\ e_2\ return\ root(eval(p, e_3))$ - - I. H.
  $= root(for\ \$v\ in\ e_1\ where\ e_2$
  $return\ eval(p, e_3))$ - - Pro. $root$
  $= root(eval(p, (for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3)))$ XQuery
  □

Constructors (rules $pp18$ to $pp20$).
Rule $pp18$.
In this case, we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p, ()) = eval(p, element\ \{QName\}\{e\})$

- $\forall p \in \mathcal{P}, root(eval(p, ())) = root(eval(p, element\ \{QName\}\{e\}))$

In XQuery, the evaluation of a path $p = p_1/\ldots/p_m$ on an element $<QName>\ldots</QName>$ $(element\ \{QName\}\{e\})$ yields to an empty sequence when the first step p_1 of the path does not match $QName$. In the premise of this rule, we assume that $p_1 \neq QName$ and p_1 is not a variable name $\$var$. So, according to the definition of $eval$ function, $eval(p, element\ \{QName\}\{e\})$ will return an empty sequence in this case. Since $eval(p, element\ \{QName\}\{e\}) = ()$ and $eval(p, ()) = ()$, the lemma properties are satisfied.

Rule $pp19$.
In this case we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p, ()) = eval(p, element\ \{QName\}\{e\})$

- $\forall p \in \mathcal{P}, root(eval(p, ())) = root(eval(p, element\ \{QName\}\{e\}))$

In the premise of the considered rule, we assume that:

- $\forall p' \in \mathcal{P}'^{\#} : eval(p', ()) = eval(p', e)$

- $\forall p' \in \mathcal{P}' : root(eval(p', ())) = root(eval(p', e))$

where $\mathcal{P}' = \{p', [QName \mid \$var]/p' \in \mathcal{P}\}$ and $\mathcal{P}'^{\#} = \{p', [QName \mid \$var]/p' \in \mathcal{P}^{\#}\}$.
Then, we infer from the following:

- $\forall \in \mathcal{P}^{\#}$
  If $p = [QName \mid \$var]/p'$ then
  $eval(p, ()) = () = eval(p', ())$ - - XQuery
  $= eval(p', e)$ - - I. H.
  $= eval(p, (element\{QName\}\{e\}))$ - - Prop. 2 of $eval$

  If $p = step_1/p',\ step_1 \neq [QName|\$var]$ then
  $eval(p, ()) = ()$
  $= eval(p, (element\{QName\}\{e\}))$ - - XQuery

  If $p = [QName \mid \$var]$ then
  No path corresponds to the pattern, $pp17$ is checked before.

- $\forall p \in \mathcal{P}$
  If $p = [QName \mid \$var]/p'$ then
  $root(eval(p, ())) = root(()) = root(eval(p', ()))$ - - XQuery
  $= root(eval(p', e))$ - - I. H.
  $= root(eval(p, (element\{QName\}\{e\})))$ - - Prop. 2 $eval$

  If $p = step_1/p',\ step_1 \neq [QName|\$var]$ then
  $root(eval(p, ())) = root(())$ - - XQuery
  $= root(eval(p, (element\{QName\}\{e\})))$ - - Prop. 2 $eval$

  If $p = [QName \mid \$var]$ then
  No path corresponds to the pattern - - I. H.

So, we conclude that the lemma properties are satisfied.

Rule $pp20$.
In this case, we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p, (element\{QName\}\{\})) = eval(p, (element\{QName\}\{e\}))$
- $\forall p \in \mathcal{P}, root(eval(p, (element\{QName\}\{\}))) = root(eval(p, (element\{QName\}\{e\})))$

In the premise of the considered rule, we assume that there is only one path $p = [QName|\$var]$ in $\mathcal{P}$, and there is no other path in $\mathcal{P}$ and $\mathcal{P}^{\#}$ that starts with $QName$ or $\$var$.
We conclude, from the following that the lemma properties are satisfied:

- $\forall p \in \mathcal{P}^{\#}$
  If $p = [QName \mid \$var]/p'$ then
  No path corresponds - - I. H.

  If $p = step_1/p'$, $step_1 \neq [QName|\$var]$ then
  $eval(p, element\{QName\}\{\}) = ()$ - - XQuery
  $= eval(p, (element\{QName\}\{e\}))$ - - XQuery

  If $p = [QName \mid \$var]$ then
  No path corresponds - - I. H.

- $\forall p \in \mathcal{P}$
  If $p = [QName \mid \$var]/p'$ then
  No path corresponds - - I. H.

  If $p = step_1/p'$, $step_1 \neq [QName|\$var]$ then
  $root(eval(p, element\{QName\}\{\})) = root((())$ - - XQuery
  $= root(eval(p, (element\{QName\}\{e\})))$ - - XQuery

  If $p = [QName \mid \$var]$ then - - one path corresponds, I. H.
  $root(eval(p, element\{QName\}\{\}))$
  $= root(element\{QName\}\{\}))$ - - XQuery
  $= element\{QName\}\{\})$ - - Prop. root
  $= root(element\{QName\}\{e\}))$ - - Prop. root
  $= root(eval(p, (element\{QName\}\{e\})))$ - - XQuery

Rule $pp21$.
In this case, we have to prove the following properties:

- $\forall p \in \mathcal{P}^{\#}, eval(p, (element\{QName\}\{e'\})) = eval(p, (element\{QName\}\{e\}))$
- $\forall p \in \mathcal{P}, root(eval(p, (element\{QName\}\{e'\}))) = root(eval(p, (element\{QName\}\{e\})))$

In the premise of the considered rule, we assume that:

- $\forall p' \in \mathcal{P}'^{\#} : eval(p', e') = eval(p', e)$
- $\forall p' \in \mathcal{P}' : root(eval(p', e')) = root(eval(p', e))$

where $\mathcal{P}' = \{p', [QName \mid \$var]/p' \in \mathcal{P}\}$ and $\mathcal{P}'^{\#} = \{p', [QName \mid \$var]/p' \in \mathcal{P}^{\#}\}$.
So, we conclude from the following that the lemma properties are satisfied:

- $\forall p \in \mathcal{P}^{\#}$
  If $p = [QName \mid \$var]/p'$ then
  $eval(p, element\{QName\}\{e'\})$
  $= eval(p', e')$ - - XQuery
  $= eval(p', e)$ - - I. H.
  $= eval(p, (element\{QName\}\{e\}))$ - - Prop. 2 of $eval$

  If $p = step_1/p'$, $step_1 \neq [QName|\$var]$ then
  $eval(p, element\{QName\}\{e'\}) = ()$ - - XQuery
  $= eval(p, (element\{QName\}\{e\}))$ - - XQuery

  If $p = [QName \mid \$var]$ then
  No path corresponds, pp17 checked before

- $\forall p \in \mathcal{P}$
  If $p = [QName \mid \$var]/p'$ then
  $root(eval(p, element\{QName\}\{e'\}))$
  $= root(eval(p', e'))$ - - XQuery
  $= root(eval(p', e))$ - - I. H.
  $= root(eval(p, (element\{QName\}\{e\})))$ - - Prop. 2 $eval$

  If $p = step_1/p'$, $step_1 \neq [QName|\$var]$ then
  $root(eval(p, element\{QName\}\{e'\}))$
  $= root((()))$ - - XQuery
  $= root(eval(p, (element\{QName\}\{e\})))$ - - Prop. 2 $eval$

  If $p = [QName \mid \$var]$ then
  $root(eval(p, element\{QName\}\{e'\}))$
  $= root(element\{QName\}\{e'\})$ - - XQuery
  $= element\{QName\}\{\}$ - - Prop. root
  $= root(element\{QName\}\{e\})$ - - Prop. root
  $= root(eval(p, (element\{QName\}\{e\})))$ - - XQuery

4 Experiments

In this section, we analyze the impact of our approach by comparing the difference between the evaluation time for the input query q and the one for the output query q'. In this way, we mesure the gain obtained by eliminating the computation of irrelevant intermediate results. In our experiments, we varied the nature and complexity of the pruned subexpressions. More precisely, we considered three kinds of subexpressions widely used in practice : FLWR blocks, XPath expressions relative to a given document or XPath expressions relative to a variable. For each kind of subexpression, we varied the amount of intermediate results produced by the pruned subexpression: 25%, 50%, 75% or 100%

of the total intermediate results. We used in our experiments the following template for test queries:

$$
\begin{array}{l}
let\ \$q := <personInf> \\
\quad \{for\ \$i\ in\ doc(\text{"}xmark.xml\text{"})/site/people/person \\
\quad\ return \\
\quad\quad (< name > \{test\_exp\} < /name >, \\
\quad\quad\ < age > \{test\_exp\} < /age >, \\
\quad\quad\ < gender > \{test\_exp\} < /gender >, \\
\quad\quad\ < email > \{test\_exp\} < /email >)\} \\
\quad <\!/personInf> \\
for\ \$j\ in\ \$q \\
return(\$j/names?, \$j/age?, \$j/gender?, \$j/email?)
\end{array}
$$

where the question mark indicates optional parts that could be missing from one test query to another.

By the first let clause in the template we create a set of intermediate results. The let binds the variable $\$q$ to a $personInf$ element that contains four child elements $name$, age, $gender$ and $email$. The four elements have the same content, produced by a $test\_exp$ expression (to be defined for each test query).

The number of children of $personInf$ depends on the size of the sequence to which the $\$i$ variable is bound ($person$ elements) and varies with the size of the document on which the test was performed. The percentage of useless intermediate results is simply tuned by deciding which XPath expressions appear in the query, among the four expressions given in the $return$ of the outer for clause. For example, when testing the gain for 100% of irrelevant intermediate results, we can use the path $\$j/names$, because it does not follow any child of the $personInf$ element. When testing the gain for 50% of irrelevant intermediate results, we can use two paths, $\$j/age$ and $\$j/gender$.

Finally, the kind of expression that is pruned along with its wrapping element was also varied ($test\_exp$).

We show in Figures 3, 4 and 5 the percentage of gain in evaluation time when $test\_exp$ is a FLWR block, an XPath expressions relative to a given document or an XPath expression relative to a variable.

These measures were obtained on the query processor Galax[9] (version 0.7.2). Our choice was motivated by the robustness of this processor and its conformance with the W3C XQuery specifications. The measures were conducted on a Pentium D 3.2 GHz PC machine, with 2Gb of memory and a Linux Debian operating system.

Results & Discussion. The experiments show that our approach ensures a gain of time whatever is the nature of the pruned subexpressions. The gain varies according to the amount of pruned intermediate results and the complexity of the irrelevant subexpression.

In Figure 3, where the pruned subexpressions correspond to FLWR blocks, the saving of time seams to be linked to the amount of pruned intermediate results. This saving increases slightly when the document size increases. It increases significantly when the pruned subexpressions correspond to XPath expressions relative to a document (Figure 4). We believe that this is mainly due to the specificity of the XQuery processor we used. In Figure 5, the pruned subexpressions correspond to XPath expressions relative to a variable. In this case, we measured savings of time less important than in the two previous cases. It seems that in this kind of scenarios we save only the time needed to retrieve the element followed by the path, which is normally done in main memory.

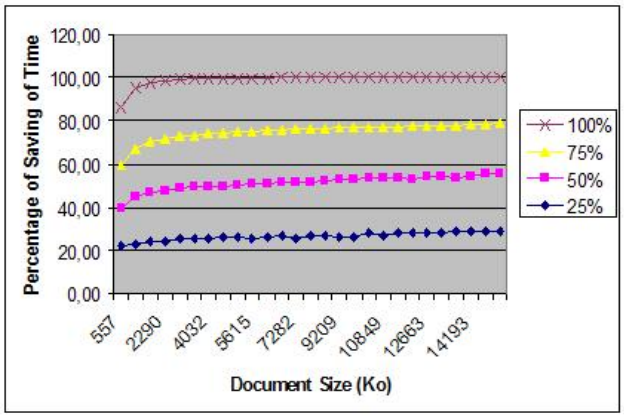

Figure 3: Test Results for Queries pruning FLWR Blocks

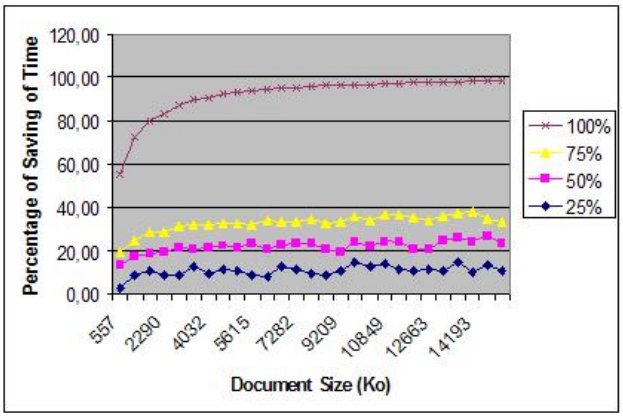

Figure 4: Test Results for Queries pruning Variable XPaths

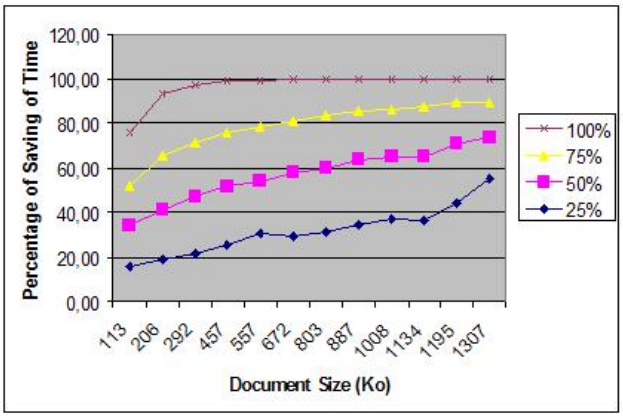

Figure 5: Test Results for Queries pruning Document XPaths

5 Conclusion

We present in this paper a rewriting algorithm for XQuery queries. The underlying approach consists in pruning from subexpressions the computations that are irrelevant for the overall query result. Our algorithm generates for each input query q, an output query q' that is equivalent to q. We show by extensive experiments the important saving of evaluation time, and we prove formally the correctness of our algorithm.

References

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *SIGMOD Conf*, 2004.

[2] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based xml projection. In *VLDB Conf*, 2006.

[3] M. Brantner, C-C. Kanne, and G. Moerkotte. Let a Single FLWOR Bloom (to improve XQuery plan generation). In *XSym Workshop*, 2007.

[4] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. Xperanto: Middleware for publishing object-relational data as xml documents. In *VLDB Conf*, 2000.

[5] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Logical Framework for XQuery. In *VLDB Conf*, 2004.

[6] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of nested xml queries. In *VLDB Conf*, 2004.

[7] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation*, 2007.

[8] M. F. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. Silkroute: A framework for publishing relational data in xml. *ACM Trans. Database Syst.*, 27(4), 2002.

[9] M. F. Fernández and J. Siméon. *The Galax System "The XQuery Implementation for Discriminating Hackers" Version 0.7.2*, 2007.

[10] M. Grinev. XQuery Optimizing Based on Rewriting. In *ADBIS*, 2004.

[11] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD Conf*, 2005.

[12] C. Koch. On the role of Composition in XQuery. In *WebDB Workshop*, 2005.

[13] A. Malhotra, J. Melton, and N. Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation*, 2007.

[14] I. Manolescu, D. Florescu, and D. Kossmann. Answering xml queries on heterogeneous data sources. In *VLDB Conf*, 2001.

[15] A. Marian and J. Siméon. Projecting XML Documents. In *VLDB Conf*, 2003.

[16] P. Michiels. XQuery Optimization. In *VLDB PhD Workshop*, 2003.

[17] P. Michiels, G. A. Mihaila, and J. Siméon. Put a tree pattern in your algebra. In *ICDE Conf*, 2007.

[18] P. Ramanan. Efficient Algorithms for Minimizing Tree Pattern Queries. In *SIGMOD Conf*, 2002.

[19] A. Schmidt, F. Waas, M. Kirsten, M. J.Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB Conf*, 2002.

[20] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *VLDB Conf*, 2001.

[21] I. Tatarinov and A. Y. Halevy. Efficient Query Reformulation in Peer-Data Management Systems. In *SIGMOD Conf*, 2004.