
Pruning Nested XQuery Queries

Bilel Gueni[‡], Talel Abdessalem[‡], Bogdan Cautis[‡], Emmanuel Waller[†]

[‡] *Telecom ParisTech*
LTCI – UMR CNRS 5141
46, rue Barrault-Paris, 75013 – France
First.Last@enst.fr

[†] *Université de Paris-Sud*
LRI – UMR CNRS 8623
Bât. 490-91405 Orsay Cedex – France
Emmanuel.Waller@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 we prove formally its correctness.

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

Table des matières

1	Introduction	3
2	Preliminaries	8
3	The rewriting algorithm	12
3.1	Path analysis	12
3.2	Path projection	15
3.3	Pruning process	19
3.4	Correctness	23
3.4.1	Proof of lemma 3.1	25
3.4.2	Proof of lemma 3.2	26
4	Optimized pruning	33
4.1	Environment changes	34
4.2	Path Projection extensions	34
4.3	Extended algorithm	37
4.3.1	<i>prune</i> function.	38
4.3.2	<i>prune'</i> function.	39
5	Handling Descendent and Attribute axis	40
5.1	Path projection	40
5.2	Pruning	42
6	Experiments	43
7	Conclusion	45
8	Bibliographie	46

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. [MAR 03], of projecting XML documents w.r.t. a given XQuery query. Instead of XML documents, we project XQuery subexpressions with respect to 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 [SCH 02]. 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 follows¹:*

Q_2 :
`<site>`

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

4 Technical Report.

```
{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>
```

```
Q3 :
<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 "`source1 := docC@S1 ∪ Q2 ∪ Q3`" 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 `source1`, which returns the `open_auction` elements that have some person data in common with another document, `docD@S1`:

```
Q1 :
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 `source1`. 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 queries 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 :

```

Q1.2 :
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 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 $Q_{1.2}$.

```

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 :

```

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 potentially 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 heterogeneous XML sources in integration and mediation scenarios are quite common [HAA 05, TAT 04, ABI 04]. The Clio project [HAA 05] 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 [TAT 04], a peer can refer to data held 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 [ABI 04] 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 common to cache and reuse the *definitions* of queries but not necessarily their results. This can for instance guide inexperienced users, allowing them to query XQuery expressions that are already available and well-understood. Finally, our sim-

plification technique can be used to optimize automatically generated queries (e.g., for graphical editors in the style of query-by-example).

Related work. Several works on XQuery processing and optimization adopt an approach based on rewrite rules. In [MAN 01, MIC 07, MIC 03, GRI 04, RAM 02, SHA 01], 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. [BRA 07] introduces rewrite rules for nesting minimization but does not consider the elimination of useless navigation and result construction. In [DEU 04], 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 [DEU 04].

More germane to this work is [MAR 03], 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 [BEN 06]. 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 [TAT 04, DON 04], 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 [KOC 05]. 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, [KOC 05] 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 [CAR 00] and SilkRoute [FER 02]. 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 some preliminary notions. Section 3 details our rule-based algorithm and extensions of the algorithm are presented in Section 4. In Section 5 we describe the experiments we conducted. We conclude in Section 6.

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 grammar of Figure 1.

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

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 ($//$) are presented in the Section 4. 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 [DRA 07], 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 variable or reformulate nested expressions in order to have one variable per clause.

$$\begin{aligned}
e/step_1/\dots/step_n &\Rightarrow \text{let } \$v := e \text{ return } \$v/step_1/\dots/step_n \\
&\quad \text{for } \$v_1 \text{ in } e_1, \dots, \$v_n \text{ in } e_n \text{ return } e \Rightarrow \\
&\quad \text{for } \$v_1 \text{ in } e_1 \text{ return for } \dots \text{ for } \$v_n \text{ in } e_n \text{ return } e \\
&\quad \text{let } \$v_1 := e_1, \dots, \$v_n := e_n \text{ return } e \Rightarrow \\
&\quad \text{let } \$v_1 := e_1 \text{ return let } \dots \text{ let } \$v_n := e_n \text{ return } e \\
&\quad \text{some } \$v_1 \text{ in } e_1, \dots, \$v_n \text{ in } e_n \text{ satisfies } e \Rightarrow \\
&\quad \text{some } \$v_1 \text{ in } e_1 \text{ satisfies some } \dots \text{ some } \$v_n \text{ in } e_n \text{ satisfies } e \\
&\quad \text{every } \$v_1 \text{ in } e_1, \dots, \$v_n \text{ in } e_n \text{ satisfies } e \Rightarrow \\
&\quad \text{every } \$v_1 \text{ in } e_1 \text{ satisfies every } \dots \text{ every } \$v_n \text{ in } e_n \text{ satisfies } e \\
\langle QName \rangle \{ e \} \langle /QName \rangle &\Rightarrow \text{element}\{QName\}\{e\}
\end{aligned}$$

Figure 2. Normalization Rules

Inference rules notation and environment. We present our algorithm via a set of inference rules, and we adopt standard programming languages notation similar to the one used in [MAR 03]. Inference rules are based on *judgements*, which denote statements of the form :

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

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

Inference rules are represented as follows :

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

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

In XQuery, a variable is always associated (by either $\$v \text{ in } exp$ or $\$v := exp$) 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 $\$j$ is bound to a constructed site element wrapping some content returned by Q_2 's FLWR expression.

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

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 $\$v$ to a set of objects. Formally, this is written $\$v \Rightarrow \{o_1, \dots, o_m\}$. 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 write $\$i \Rightarrow \{docA@S2/site\}$ for Q_2 , $\$i \Rightarrow \{docB@S3/site/closed_auctions/closed_auction\}$, $\$l \Rightarrow \{\$i\}$ for Q_3 , or $\$j \Rightarrow \{<site>...</site>\}$ 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 :

$$\begin{aligned} varRes(\textit{for } \$v \textit{ in } e_1 \textit{ (where } e_2\textit{)? return } e_3) &\Rightarrow varRes(e_3) \\ varRes(\textit{let } \$v := e_1 \textit{ (where } e_2\textit{)? return } e_3) &\Rightarrow varRes(e_3) \\ varRes(\textit{if } (e_1) \textit{ then } e_2 \textit{ else } e_3) &\Rightarrow varRes(e_2) \cup varRes(e_3) \\ varRes(e_1, \dots, e_n) &\Rightarrow varRes(e_1) \cup \dots \cup varRes(e_n) \\ varRes(\textit{step}_1 / \dots / \textit{step}_2) &\Rightarrow \{\textit{step}_1 / \dots / \textit{step}_2\} \\ varRes(\textit{element}\{QName\}\{e\}) &\Rightarrow \{\textit{element}\{QName\}\{e\}\} \\ varRes(\$v) &\Rightarrow \{\$v\} \\ varRes(\textit{literal}) &\Rightarrow \{\#text\} \\ varRes(\textit{some } \$v \textit{ in } e_1 \textit{ satisfies } e_2) &\Rightarrow \{\#text\} \\ varRes(\textit{every } \$v \textit{ in } e_1 \textit{ satisfies } e_2) &\Rightarrow \{\#text\} \end{aligned}$$

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 *After obtaining by $varRes$ that $\$l \Rightarrow \i and $\$i \Rightarrow \{docB@S3/site/closed_auctions/closed_auction\}$, we can 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 $\$v$ and its bound expression $exp_b(\$v)$, the addition of $\$v$ to the environment is achieved by the following statement :

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

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

- *getBind*(\$v) : retrieves from the environment the set of objects associated to the input variable \$v.
- *getXPathBind*(\$v) : among the objects to which the variable \$v 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 compute intermediate results that may only be partially useful to Q 's end result, our algorithm identifies and prunes the irrelevant parts in bound expressions. The output is an equivalent query Q' obtained from Q by substituting each subexpression $exp_b(\$v)$ by a subexpression $exp'_b(\$v)$ that has the advantage of computing only the needed intermediate results.

For a given variable $\$v$, the algorithm retrieves from $exp_r(\$v)$ all the XPath expressions that access the result of $exp_b(\$v)$. This task is performed by the *extractPaths* function. These paths are then used to retrieve and project out the useless parts in $exp_b(\$v)$. This is the role of the *projectPaths* function. A simpler subexpression $exp'_b(\$v)$ 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 $\$v$ in Q , the pruning is first applied recursively within its bound and return expressions, then for $\$v$ 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

The *extractPaths* function takes as input a variable $\$v$ and its return expression $exp_r(\$v)$, analyses $exp_r(\$v)$ and extracts the paths that navigate through the variable $\$v$. These paths start with $\$v$ (either explicitly, or via other declared variables), and are denoted the *projection paths* of $\$v$.

Similar to [MAR 03], 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. These are the paths that simply bind a variable $\$v$, appearing only in its bound expression $exp_b(\$v)$.

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 *used* kind are verified.

We now present the inference rules for *extractPaths* function. The result of a rule application will be two sets of paths, \mathcal{P} and $\mathcal{P}^\#$, for the *used* and *returned paths* respectively.

Literal, empty sequence. When the input expression $exp_r(\$v)$ is a literal (rule $ep1$) or an empty sequence (rule $ep2$), no paths can be extracted and the output result is two empty sets.

$$\frac{}{Env \vdash extractPaths(\$v, literal) \Rightarrow \emptyset, \emptyset}^{(ep1)}$$

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

Sequence, conditional, comparison, element construction. In this case (rules $ep3$ to $ep6$), the static analysis of the input expression yields to the static analysis of its subexpressions, and the output sets \mathcal{P} and $\mathcal{P}^\#$ are obtained from the union of the *used* and *returned paths* extracted from the subexpressions.

$$\frac{Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^\# \quad Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^\#}{Env \vdash extractPaths(\$v, (e_1, e_2)) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{P}_1^\# \cup \mathcal{P}_2^\#}^{(ep3)}$$

$$\frac{Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^\# \quad Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^\# \quad Env \vdash extractPaths(\$v, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^\#}{Env \vdash extractPaths(\$v, if(e_1) then e_2 else e_3) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}_1^\# \cup \mathcal{P}_2^\# \cup \mathcal{P}_3^\#}^{(ep4)}$$

$$\frac{Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^\# \quad Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^\#}{Env \vdash extractPaths(\$v, (e_1 (Op) e_2)) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{P}_1^\# \cup \mathcal{P}_2^\#}^{(ep5)}$$

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

Variable reference. When the input expression $exp_r(\$v)$ is a variable reference, we have two alternative cases. If $exp_r(\$v) = \v (rule $ep7$), then the output result is a *returned path* $\$v$ (a one step path). If $exp_r(\$v) = \v' ($\$v' \neq \v), then we extract from the XPath expressions bound to $\$v'$ those who are relative to $\$v$ (i.e., $\$v$ is their first step). These paths constitute the output set of *returned paths* $\mathcal{P}^\#$ (see rule $ep8$). The output set of *used paths* \mathcal{P} is empty in both cases.

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

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

XPath expression. Here, due to the normalization process described in section 2, we have only three alternative cases (rules $ep9$ to $ep11$). If the input expression $exp_r(\$v)$

is a path relative to $\$v$, then this path composes the output set of *returned paths* $\mathcal{P}^\#$ (rule *ep9*). If $exp_r(\$v) = \$v'/\dots/s_n$ ($\$v' \neq \v), then we extract from the XPath expressions bound to $\$v'$ those who are relative to $\$v$ (if any). These are used to substitute $\$v'$ and create the output set of *returned paths* $\mathcal{P}^\#$ (rule *ep10*). Otherwise (Rule *ep11*), no paths relative de $\$v$ can be extracted from the input expression and the output set of *returned paths* is empty. The output set of *used paths* \mathcal{P} is empty in the three cases.

$$\frac{}{Env \vdash extractPaths(\$v, \$v/\dots/s_n) \Rightarrow \emptyset, \{\$v/\dots/s_n\}}^{(ep9)}$$

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

$$\frac{}{Env \vdash extractPaths(\$v, doc(uri)/\dots/s_n) \Rightarrow \emptyset, \emptyset}^{(ep11)}$$

FLWR expression, quantifier. Here, the input expression $exp_r(\$v)$ is a FLWR or quantifier expression : for instance, *for* $\$v'$ *in* e_1 *where* e_2 *return* e_3 .

The first three premises (or the first two, in the case of the quantifier expression) in the inference rules *ep12* to *ep14* will simply apply recursively the path analysis process to the subexpressions e_1 , e_2 and e_3 .

The role of the remaining premises is to transform some of the returned paths from e_1 ($\mathcal{P}_1^\#$) into used paths for $\$v$. (This would obviously allow for more drastic query simplifications in the later stages.) The transformation happens when certain conditions are verified for the projection paths $\mathcal{P}_1^\#$, more precisely when (1) the paths are bound to variable $\$v'$ (this can be checked by testing if they appear in $Env.getXPathBind(\$v')$), and (2) the paths are relative to $\$v$ (i.e., $\$v$ is their first step). The paths verifying these two conditions are moved from the set of *returned paths* to the one of *used paths*.

Consider the following example :

Example 3.1 *Given the query :*

```
for $v in ...
return for $v' in ($v/A/B, ...)
return ...
```

the path $\$v/A/B$ is bound to variable $\$v'$. Although $\$v/A/B$ is initially a returned path, the fact that it is relative to $\$v$ suggests that it can be safely considered as a used path. In this case, the descendants of B elements are not considered to be useful to compute the end result (unless some other path overwrites this fact).

$$\begin{aligned} Env \vdash extractPaths(\$v, e_1) &\Rightarrow \mathcal{P}_1, \mathcal{P}_1^\# \\ Env \vdash extractPaths(\$v, e_2) &\Rightarrow \mathcal{P}_2, \mathcal{P}_2^\# \end{aligned}$$

$$\begin{array}{c}
Env \vdash extractPaths(\$v, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^\# \\
Env.getXPathBind(\$v') \Rightarrow P \\
\mathcal{P}_{1.bis} = \{p \in P, p = \$v/\dots\}, \mathcal{P}_1^\# - \mathcal{P}_{1.bis} \Rightarrow \mathcal{P}_{1.bis}^\# \\
\hline
Env \vdash extractPaths(\$v, \text{for } \$v' \text{ in } e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_{1.bis}, \mathcal{P}_{1.bis}^\# \cup \mathcal{P}_2^\# \cup \mathcal{P}_3^\# \quad (ep12)
\end{array}$$

$$\begin{array}{c}
Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^\# \\
Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^\# \\
Env \vdash extractPaths(\$v, e_3) \Rightarrow \mathcal{P}_3, \mathcal{P}_3^\# \\
Env.getXPathBind(\$v') \Rightarrow P \\
\mathcal{P}_{1.bis} = \{p \in P, p = \$v/\dots\}, \mathcal{P}_1^\# - \mathcal{P}_{1.bis} \Rightarrow \mathcal{P}_{1.bis}^\# \\
\hline
Env \vdash extractPaths(\$v, \text{let } \$v' := e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup \mathcal{P}_{1.bis}, \mathcal{P}_{1.bis}^\# \cup \mathcal{P}_2^\# \cup \mathcal{P}_3^\# \quad (ep13)
\end{array}$$

$$\begin{array}{c}
Env \vdash extractPaths(\$v, e_1) \Rightarrow \mathcal{P}_1, \mathcal{P}_1^\# \\
Env \vdash extractPaths(\$v, e_2) \Rightarrow \mathcal{P}_2, \mathcal{P}_2^\# \\
Env.getXPathBind(\$v') \Rightarrow P \\
\mathcal{P}_{1.bis} = \{p \in P, p = \$v/\dots\}, \mathcal{P}_1^\# - \mathcal{P}_{1.bis} \Rightarrow \mathcal{P}_{1.bis}^\# \\
\hline
Env \vdash extractPaths(\$v, \text{some } \$v' \text{ in } e_1 \text{ satisfies } e_2) \Rightarrow \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_{1.bis}, \mathcal{P}_{1.bis}^\# \cup \mathcal{P}_2^\# \quad (ep14)
\end{array}$$

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.

$$\begin{array}{l}
(ep11) \quad Env \vdash extractPaths(\$j, docD@S_1/site) \Rightarrow \emptyset, \emptyset \\
(ep5, ep9, ep10) \quad Env \vdash extractPaths(\$j, \$j/person = \$k \dots) \Rightarrow \emptyset, \{\$j/person\} \\
(ep6, ep9) \quad Env \vdash extractPaths(\$j, <common-auction>\{\$j/open_auction\} \\
\quad \quad \quad </common-auction>) \Rightarrow \emptyset, \{\$j/open_auction\} \\
(ep12) \quad Env \vdash extractPaths(\$j, \text{for } \$k \dots) \Rightarrow \emptyset, \{\$j/open_auction, \$j/person\}
\end{array}$$

3.2 Path projection

In this section, we present the function *projectPaths* that, for each variable $\$v$, projects out the useless parts of $exp_b(\$v)$ (the bound expression for $\$v$) based on the sets of paths \mathcal{P} and $\mathcal{P}^\#$ extracted from $exp_r(\$v)$ (the return expression for $\$v$). This function takes as input a set of *used paths*, a set of *returned paths* and an XQuery expression, and returns a new XQuery expression. Intuitively, the output expression is obtained by projecting out any subexpression producing an intermediary result that is not in the scope of these paths. For a given path p and a given expression e , *projectPaths* analyzes p and e and determines if a matching is possible between p and the expected result of e (i.e., we expect a possibly non empty result for the evaluation of p on the result of e).

Next, we detail the inference rules.

Literal, comparison, quantifier. When matching an input path p with a literal value, the only cases that yield to a non empty result are when $p = \text{text}()$ (a final step of a *projection path*) or $p = \$v$ (see rule *pp1*). Otherwise (rule *pp2*), the output expression is empty.

We use the same reasoning when the input query is a comparison ($e_1 \text{ Op } e_2$) or a quantifier expression (*some* $\$v$ *in* e_1 *satisfies* e_2), as the result of their evaluation is necessarily a numeric or a boolean literal. The corresponding rules (*pp3* to *pp6*) are similar to *pp1* and *pp2*.

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

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

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

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

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

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

Sequence. The matching of an input path p with a sequence (e_1, e_2) yields to the matching of p with the subexpressions (e_2 and e_3) composing the sequence. Then, the output expression is a sequence composed from the obtained elementary results (see rule *pp7*).

$$\frac{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_1) \Rightarrow e'_1 \quad Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_2) \Rightarrow e'_2}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, (e_1, e_2)) \Rightarrow e'_1, e'_2} \text{ (pp7)}$$

Variable reference. The evaluation of a path p on a variable $\$v$ yields to the evaluation of this path on the objects bound to $\$v$. Then, the output expression is equal to $\$v$ (equal to the input expression), if a matching is possible between at least an input path p and one of the objects bound to $\$v$ (rule *pp8*). Otherwise (rule *pp9*), the output expression is empty.

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

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

XPath expression. Here, the input expression e is an XPath ($e = s_1 / \dots / s_n$). The matching (or not) of an input path p with e depends on the nature of p and the expected result of e . Many useful information on this expected result can be deduced from the last step s_n of e (rule *pp10*).

If $s_n = text()$, the result will be composed of literal values. So, the only case where a matching is possible is when $p = text()$ (a final step of a *projection path*) or $p = \$v$.

If $s_n \neq text()$ and $s_n \neq *$, hence the nodes returned by $e = s_1 / \dots / s_n$ are element nodes, it is sufficient to have one input path in $\mathcal{P} \cup \mathcal{P}^\#$ that starts with $\$v$, $*$ (which corresponds to any element test) or s_n (i.e., the first step of p corresponds to the element returned by e). Since we do not have enough information about the eventual descendants of s_n , we only check if a matching is possible between e and the first step of p .

If $s_n = *$, hence the name of the returned element nodes is unknown, the only case where there is no possible matching is when $p = text()$.

Otherwise (rule *pp11*), no matching is possible and the output expression is empty.

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

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

FLWR expression. According to the XQuery semantics, the result of a *FLWR* expression is computed by its *return* subexpression. So, the matching of an input path p with a *FLWR* expression e yields to the matching of p with the *return* subexpression of e (e_3 in *pp12*). If, for at least one path $p \in \mathcal{P} \cup \mathcal{P}^\#$ the result expression e'_3 is not empty, the composed output is a new *FLWR* expression obtained by substituting the initial *return* expression by e'_3 in the input *FLWR* (rule *pp12*). Otherwise (rule *pp13*), the output expression is empty. The specific rules of *let* expressions (*pp14* and *pp15*) are similar to those of the *for* expressions.

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e_3) \Rightarrow e'_3}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, \text{for } \$v \text{ in } e_1 \text{ where } e_2 \text{ returns } e_3) \Rightarrow \text{for } \$v \text{ in } e_1 \text{ where } e_2 \text{ return } e'_3} \text{ (pp12)}$$

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

$$\frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e_3) \Rightarrow e'_3}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, let \$v := e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow let \$v := e_1 \text{ where } e_2 \text{ return } e'_3} \text{ (pp14)}$$

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

Conditional. The matching of an input path p with a conditional expression (*if* (e_1) *then* e_2 *else* e_3) yields to the matching of p with the subexpressions composing the *true* and the *false* branches. The output expression is then a new conditional expression obtained by substituting in the input expression the initial *true* and *false* branches by two potentially simplified subexpressions (see rule *pp16*).

$$\frac{\begin{array}{l} Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e_2) \Rightarrow e'_2 \\ Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e_3) \Rightarrow e'_3 \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, if (e_1) \text{ then } e_2 \text{ else } e_3) \Rightarrow if (e_1) \text{ then } e'_2 \text{ else } e'_3} \text{ (pp16)}$$

Element construction. Here, the input expression is an element construction expression $expr = element\{QName\}\{e\}$. The matching (or not) of an input path p with $expr$ depends on the nature of p . 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 the input set $\mathcal{P}^\#$ contains a path p such as $p = \$v$ or $p = QName$, then a matching is possible between $expr$ and at least one *returned path*. In this case, nothing can be projected out and the returned expression is identical to the input one.

Rule *pp18* : if there is no input path $p = s_1/\dots$ with a first step $s_1 = \$v$ or $s_1 = QName$, then no matching is possible between $expr$ and the input paths. In this case, the output expression is empty (i.e., the input expression is projected out).

Rule *pp19* : if, for all the input paths $p = s_1/s_2/\dots$ with a first step $s_1 = \$v$ or $s_1 = QName$, there is no matching possible between their remaining parts s_2/\dots and the subexpression e , then no matching is possible between $expr$ and the input paths. The output expression is also empty in this case.

Rule *pp20* : if there is at least one *used path* p such as $p = \$v$ or $p = QName$, and no other input path has a first step equal to $\$v$ or $QName$, then we deduce that the descendent nodes of the expected element $QName$ are useless. In this case, the subexpression e is projected out and the output expression is then a new element construction $QName$ with an empty content.

Rule pp21 : if there is at least one input path $p = s_1/s_2/\dots$ with a first step $s_1 = \$v$ or $s_1 = QName$, and a matching is possible between the remaining part s_2/\dots of p and the subexpression e , then we deduce that a matching is possible with $expr$. In this case the output is a new element construction expression for $QName$ with a new subexpression e' obtained by the recursive application of $projectPaths$ for these s_2/\dots paths over e .

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

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

$$\frac{\begin{array}{l} \exists p \in \mathcal{P}, p = (QName | \$v) \\ \mathcal{P}' = \{p', [QName | \$v]/p' \in \mathcal{P}\} \\ \mathcal{P}'^\# = \{p', [QName | \$v]/p' \in \mathcal{P}^\#\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^\#, e) \Rightarrow () \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow ()} \text{ (pp19)}$$

$$\frac{(\exists p \in \mathcal{P}, p = (QName | \$v)) \wedge (\forall \bar{p} \in (\mathcal{P} \cup \mathcal{P}^\#) - \{p\}, head(\bar{p}) \neq (QName | \$v))}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{\}} \text{ (pp20)}$$

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

Empty sequence. If the input expression is empty, then the output expression is also empty.

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

3.3 Pruning process

The pruning process is applied recursively, in a bottom-up manner, by the *Prune* function. This function takes in the input a query Q and returns a pruned query Q' . It is defined by the following inference rules.

Literal, variable reference, XPath expression and empty sequence. When the input expression Q is a literal, a variable reference, an XPath expression or an empty sequence, the pruning have no effect and the output expression is the same as the input one.

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

$$\overline{Env \vdash Prune(\$v) \Rightarrow \$v} \quad (p2)$$

$$\overline{Env \vdash Prune(s_1 / \dots / s_n) \Rightarrow s_1 / \dots / s_n} \quad (p3)$$

$$\overline{Env \vdash Prune(()) \Rightarrow ()} \quad (p4)$$

Sequence, comparison, element construction. The pruning of a sequence of subexpressions returns as a result the sequence of the pruned subexpressions (rule $p5$). We use the same reasoning to prune comparison expressions and element construction expressions (rules $p6$ and $p7$).

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env \vdash Prune(e_2) \Rightarrow e'_2}{Env \vdash Prune(e_1, e_2) \Rightarrow e'_1, e'_2} \quad (p5)$$

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env \vdash Prune(e_2) \Rightarrow e'_2}{Env \vdash Prune(e_1 Op e_2) \Rightarrow e'_1 Op e'_2} \quad (p6)$$

$$\frac{Env \vdash Prune(e) \Rightarrow e'}{Env \vdash Prune(element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e'\}} \quad (p7)$$

Conditional. In this case, the pruning operation is propagated to the condition subexpression, and to the *true* and *false* branches. The output expression is obtained by substituting the subexpressions by their pruning result.

$$\frac{Env \vdash Prune(e_1) \Rightarrow e'_1 \quad Env \vdash Prune(e_2) \Rightarrow e'_2 \quad Env \vdash Prune(e_3) \Rightarrow e'_3}{Env \vdash Prune(if(e_1) then e_2 else e_3) \Rightarrow if(e'_1) then e'_2 else e'_3} \quad (p8)$$

FLWR expressions, quantifier. When the input Q is a FLWR expression (*for* $\$v$ *in* e_1 *return* e_2), the pruning operation is first applied to the bound expression e_1 . The result is a pruned expression e'_1 . Next, the variable $\$v$ is added to the environment Env with its bound objects computed by *varRes*. The *saturate* function refines the bindings stored in the environment (see Section 2).

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}, \mathcal{P}^\# \\ Env \vdash projectPaths(\mathcal{P} \cup \{\$v\}, \mathcal{P}^\#, e'_1) \Rightarrow e''_1 \end{array}}{Env \vdash Prune(for \$v in e_1 return e_2) \Rightarrow for \$v in e''_1 return e'_2} \quad (p9)$$

Then, the pruning operation is applied to the return subexpression e_2 . The result is a pruned expression e'_2 , from which the *extractPaths* function extracts the paths relative to $\$v$. Finally, extracted paths ($\mathcal{P} \cup \{\$v\}, \mathcal{P}^\#$) are applied on e'_1 in order to remove its useless parts. Here, $\$v$ is added to ensure that the number iterations

remains the same. The resulting expression e_1'' and e_2' replace e_1 and e_2 , respectively, in the FLWR expression.

The pruning of *for* expression can lead to the following interesting special cases (p10 and p11) :

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

Here, the pruning of the bound expression generates an empty result, which means that the number of iterations is equal to 0. So, the pruning result of the whole FLWR expression is empty.

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

This case corresponds to the situation in which the pruning of the return expression e_2 leads to the empty result. This means that whatever the number of iterations is, the result is always empty. In such case, the pruning of the whole *for* expression gives empty result.

When the input *for* expression contains a *where* clause, we follow the same reasoning and treat the *where* clause as we treat the *return* subexpression (see rules p12 to p14).

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e_1' \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e_1')))) \\ Env \vdash Prune(e_2) \Rightarrow e_2' \\ Env \vdash Prune(e_3) \Rightarrow e_3' \\ Env \vdash extractPaths(\$v, e_2') \Rightarrow \mathcal{P}_2, \mathcal{P}_2^\# \\ Env \vdash extractPaths(\$v, e_3') \Rightarrow \mathcal{P}_3, \mathcal{P}_3^\# \\ Env \vdash projectPaths(\mathcal{P}_2 \cup \mathcal{P}_3 \cup \{\$v\}, \mathcal{P}_2^\# \cup \mathcal{P}_3^\#, e_1') \Rightarrow e_1'' \end{array}}{Env \vdash Prune(for \$v in e_1 where e_2 return e_3) \Rightarrow for \$v in e_1'' where e_2' return e_3'} \text{ (p12)}$$

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

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

When the *for* expression contains a *where* clause, an additional rule is used each time the pruning of the *where* gives an empty result (rule p15). In this case, the condition is definitively *false* (empty sequence) and the pruning of the whole *for* expression yields to an empty result.

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\ Env \vdash Prune(e_2) \Rightarrow () \end{array}}{Env \vdash Prune(for \$v in e_1 where e_2 return e_3) \Rightarrow ()} \quad (p15)$$

The pruning process of *let* expressions is similar to the pruning of *for* expressions (see rules p16 to p22). The only difference is that we do not have to add the $\$v$ to the set of used paths \mathcal{P} .

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}, \mathcal{P}^\# \\ Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e'_1) \Rightarrow e''_1 \end{array}}{Env \vdash Prune(let \$v := e_1 return e_2) \Rightarrow let \$v := e''_1 return e'_2} \quad (p16)$$

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

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ RootPath(e'_1) \Rightarrow PathB_1 \\ Env = Env + (\$v \Rightarrow PathB_1) \quad Env = Env.saturate(\$v) \\ Env \vdash Prune(e_2) \Rightarrow () \end{array}}{Env \vdash Prune(let \$v := e_1 return e_2) \Rightarrow ()} \quad (p18)$$

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash Prune(e_3) \Rightarrow e'_3 \\ Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}_2, \mathcal{P}^\#_2 \\ Env \vdash extractPaths(\$v, e'_3) \Rightarrow \mathcal{P}_3, \mathcal{P}^\#_3 \\ Env \vdash projectPaths(\mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}^\#_2 \cup \mathcal{P}^\#_3, e'_1) \Rightarrow e''_1 \end{array}}{Env \vdash Prune(let \$v := e_1 where e_2 return e_3) \Rightarrow let \$v := e''_1 where e'_2 return e'_3} \quad (p19)$$

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

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash Prune(e_3) \Rightarrow () \end{array}}{Env \vdash Prune(let \$v := e_1 where e_2 return e_3) \Rightarrow ()} \quad (p21)$$

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\ Env \vdash Prune(e_2) \Rightarrow () \end{array}}{Env \vdash Prune(let \$v := e_1 where e_2 return e_3) \Rightarrow ()} \quad (p22)$$

For the quantifier expressions, we use the same reasoning. But, for the special cases, instead of returning an empty result we return the $false()$ boolean value, as a quantifier returns necessarily a logical value.

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}, \mathcal{P}^\# \\ Env \vdash projectPaths(\mathcal{P} \cup \{\$v\}, \mathcal{P}^\#, e'_1) \Rightarrow e''_1 \end{array}}{Env \vdash Prune(some \$v in e_1 satisfies e_2) \Rightarrow some \$v in e''_1 satisfies e'_2} \text{ (p23)}$$

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

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ RootPath(e'_1) \Rightarrow PathB_1 \\ Env = Env + (\$v \Rightarrow PathB_1) \quad Env = Env.saturate(\$v) \\ Env \vdash Prune(e_2) \Rightarrow () \end{array}}{Env \vdash Prune(some \$v in e_1 satisfies e_2) \Rightarrow false()} \text{ (p25)}$$

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 $I = \{d_1, \dots, d_k\}$ 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 [MAL 07].*

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[DRA 07] 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 $q = \text{if}(e_1) \text{ then } e_2 \text{ else } e_3$, the pruning process produces $q' = \text{if}(e'_1) \text{ then } e'_2 \text{ else } e'_3$, where $Env \vdash \text{Prune}(e_1) \Rightarrow e'_1$, $Env \vdash \text{Prune}(e_2) \Rightarrow e'_2$, and $Env \vdash \text{Prune}(e_3) \Rightarrow e'_3$. 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 $\text{if}(e_1) \text{ then } e_2 \text{ else } e_3$ and $\text{if}(e'_1) \text{ then } e'_2 \text{ else } e'_3$ 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* (\mathcal{P} and $\mathcal{P}^\#$), 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. [MAR 03]). 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 ($Env \vdash \text{extractPaths}(\$var, e) \Rightarrow \mathcal{P}, \mathcal{P}^\#$) 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 ($Env \vdash \text{ProjectPaths}(\mathcal{P}, \mathcal{P}^\#, e_1) \Rightarrow e_2$), satisfies the following properties :*

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

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 and their proofs are given in the following sections.

3.4.1 Proof of lemma 3.1

–*This proof is under reviewing by its authors. A final version will be available soon.*–

We construct an induction on the inference rule of the *extractPaths* function.

Literal, empty sequence. In this case *extractPaths* returns empty sets whatever the variable, (rules *ep1* and *ep2* in Section 3.1). This is evident because a literal value or an empty sequence does not contain an XPath expression.

Sequence, conditional, comparison, element construction. For instance we take the rule *ep3* : sequences return the union of the sets of their sub-sequences. Assuming the subsequences return all the paths referencing a variable $\$v$, then the union of these sets will contain obviously all the paths that reference $\$v$ in the whole sequence. We use the same logic for the rules *ep4*, *ep5* and *ep6*.

Variable reference. For the rule *ep7* it is obvious that a variable reference references itself. For the rule *ep8* we have the following properties :

Property 3.1 [Path extension] *If a variable $\$v$ is bound to an XPath expression p , then all the nodes that are returned by paths of the form $\$v(/p')?$ are the same nodes returned by the paths of the forme $p(/p')?$.*

Property 3.2 [saturate environment function] *For a given variable $\$v$, the saturate environment function ensures that in the environment we keep track of all the links between variables.*

Property 3.3 [var Res function] *The var Res computes all the XPath expression bound to a given variable.*

According to these properties, to extract paths referencing a given variable $\$v$ from a variable referencing $\$var$, it is sufficient to extract from the environment the paths referencing $\$v$ and that are bound to $\$var$, because :

1) The property 3.1 ensures that the nodes returned by evaluating a $\$var$ and which come from a path p referencing $\$v$, are the same nodes returned by p .

2) The properties 3.2 and 3.3 guaranties that in the environment we find all the paths referencing $\$v$ and that are bound to $\$var$.

To keep all the nodes returned by these paths we have to consider them as *return* paths.

This is exactly what does the rule $ep8$. \square

XPath expression. For the rules $ep9$ and $ep10$ we use the same logic as for the rules $ep7$ and $ep8$ respectively. For the rule $ep11$, it is obvious that a path starting with $doc("uri")$ is not a path referencing a variable, and so the two sets to return are empty. \square

FLWR expression. The prove is the same for the rules $ep12$, $ep13$ and $ep14$. For the rule $ep12$, despite the last premise where we transit some paths from return to used set, the proof is similar to that of the rules $ep3$ to $ep6$. So, what we have to prove in addition is that the descendants of the nodes we migrate from return to used path set are useless. To demonstrate this, we use the following property :

Property 3.4 [*Useless descendants*] *If a path p is directly bound to a variable, then the descendants of the result of p are useless.*

We remarque that in the rule $ep12$ that the paths that transit from return path set to used one are those directly bound to the variable $\$v$ because they are computed by the $getXPathBind$ environment function, and according to the property 3.4 their descendants are useless. \square

3.4.2 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.

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 $e = literal$, $e = e_1 Op e_2$ 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 $e = \text{literal}$, $e = \text{some } \$var \text{ in } e_1 \text{ satisfies } e_2$ or $e = e_1 \text{ Op } e_2$.

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 $\text{text}()$ (see the premise of the rules). In this case, and according to the semantics of XQuery [DRA 07], $\text{eval}(p, e)$ will return an empty sequence. Since $\text{eval}(p, e) = ()$ and $\text{eval}(p, ()) = ()$, the properties are satisfied.

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

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

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 $\{o_1, \dots, o_n\}$ in the considered environment, and when the application of the input paths on these objects returns an empty sequence ($\forall i, 1 \leq i \leq n, \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, o_i) \Rightarrow ()$).

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 $\text{eval}(p, \$var) = ()$ and $\text{eval}(p, ()) = ()$, the properties are satisfied.

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

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

According to the semantics of XQuery, the last step s_n of a path $s = s_1 / \dots / s_n$ determines the nodes result of s . In the case of paths composition, between $p = p_1 / \dots / p_m$ and $s = s_1 / \dots / s_m$ (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 $a/\text{text}()$ 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, $\text{eval}(p, s_1 / \dots / s_n)$ returns an empty sequence and the properties are satisfied.

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

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

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

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

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

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

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

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

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

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

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

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 :

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

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 :

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

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

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

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 :

$$\begin{aligned}
& - \forall p \in \mathcal{P}^\# \\
& eval(p, (for \$v in e_1 where e_2 return e'_3)) \\
& = for \$v in e_1 where e_2 return eval(p, e'_3) \quad - - \text{XQuery} \\
& = for \$v in e_1 where e_2 return eval(p, e_3) \quad - - \text{I. H.} \\
& = eval(p, (for \$v in e_1 where e_2 return e_3)) \quad - - \text{XQuery} \\
& - \forall p \in \mathcal{P} \\
& root(eval(p, (for \$v in e_1 where e_2 return e'_3))) \\
& = root(for \$v in e_1 where e_2 return eval(p, e'_3)) \\
& = for \$v in e_1 where e_2 return root(eval(p, e'_3)) \quad - - \text{Prop. root} \\
& = for \$v in e_1 where e_2 return root(eval(p, e_3)) \quad - - \text{I. H.} \\
& = root(for \$v in e_1 where e_2 return eval(p, e_3)) \quad - - \text{Prop. root} \\
& = root(eval(p, (for \$v in e_1 where e_2 return e_3))) \quad \square
\end{aligned}$$

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

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

In the premise, we assume that :

$$\begin{aligned}
& - \forall p \in \mathcal{P}^\# : eval(p, ()) = eval(p, e_3) \\
& - \forall p \in \mathcal{P} : root(eval(p, ())) = root(eval(p, e_3))
\end{aligned}$$

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

$$\begin{aligned}
& - \forall p \in \mathcal{P}^\# \\
& eval(p, ()) = () \quad = for \$v in e_1 where e_2 return () \quad - - \text{XQuery} \\
& \quad \quad \quad = for \$v in e_1 where e_2 return eval(p, ()) \quad - - \text{XQuery} \\
& \quad \quad \quad = for \$v in e_1 where e_2 return eval(p, e_3) \quad - - \text{I. H.} \\
& \quad \quad \quad = eval(p, (for \$v in e_1 where e_2 return e_3)) \quad - - \text{XQuery} \\
& - \forall p \in \mathcal{P} \\
& root(eval(p, ())) \quad = root(for \$v in e_1 where e_2 return eval(p, ())) \\
& \quad \quad \quad = for \$v in e_1 where e_2 return root(eval(p, ())) \quad - - \text{Prop. root} \\
& \quad \quad \quad = for \$v in e_1 where e_2 return root(eval(p, e_3)) \quad - - \text{I. H.} \\
& \quad \quad \quad = root(for \$v in e_1 where e_2 return eval(p, e_3)) \quad - - \text{Pro. root} \\
& \quad \quad \quad = root(eval(p, (for \$v in e_1 where e_2 return e_3))) \quad - - \text{XQuery} \quad \square
\end{aligned}$$

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 $p = p_1 / \dots / p_m$ on an element $\langle QName \rangle \dots \langle / QName \rangle (element \{QName\}\{e\})$ 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 $\mathcal{P}' = \{p', [QName | \$var]/p' \in \mathcal{P}\}$ and $\mathcal{P}'^\# = \{p', [QName | \$var]/p' \in \mathcal{P}'^\#\}$.

Then, we infer from the following :

- $\forall p \in \mathcal{P}^\#$
- If $p = [QName | \$var]/p'$ 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 $p = [QName | \$var]$ then *No path corresponds to the pattern, pp17 is checked before.*

- $\forall p \in \mathcal{P}$
- If $p = [QName | \$var]/p'$ 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 $p = [QName | \$var]$ 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 :

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

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 :

$$\begin{aligned} & - \forall p \in \mathcal{P}^\# \\ & \text{If } p = [QName | \$var]/p' \text{ then } \textit{No path corresponds} \text{ -- I. H.} \\ & \text{If } p = step_1/p', \text{ } step_1 \neq [QName | \$var] \text{ then} \\ & \quad eval(p, element\{QName\}\{\}) = () \text{ -- XQuery} \\ & \quad = eval(p, (element\{QName\}\{e\})) \text{ -- XQuery} \\ & \text{If } p = [QName | \$var] \text{ then } \textit{No path corresponds} \text{ -- I. H.} \\ & - \forall p \in \mathcal{P} \\ & \text{If } p = [QName | \$var]/p' \text{ then } \textit{No path corresponds} \text{ -- I. H.} \\ & \text{If } p = step_1/p', \text{ } step_1 \neq [QName | \$var] \text{ then} \\ & \quad root(eval(p, element\{QName\}\{\})) = root(()) \text{ -- XQuery} \\ & \quad = root(eval(p, (element\{QName\}\{e\}))) \text{ -- XQuery} \\ & \text{If } p = [QName | \$var] \text{ then -- one path corresponds, I. H.} \\ & \quad root(eval(p, element\{QName\}\{\})) = root(element\{QName\}\{\}) \text{ -- XQuery} \\ & \quad = element\{QName\}\{\} \text{ -- Prop. root} \\ & \quad = root(element\{QName\}\{e\}) \text{ -- Prop. root} \\ & \quad = root(eval(p, (element\{QName\}\{e\}))) \text{ -- XQuery} \end{aligned}$$

Rule *pp21*.

In this case, we have to prove the following properties :

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

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

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

where $\mathcal{P}' = \{p', [QName | \$var]/p' \in \mathcal{P}\}$ and $\mathcal{P}'^\# = \{p', [QName | \$var]/p' \in \mathcal{P}^\#\}$.

So, we conclude from the following that the lemma properties are satisfied :

$-\forall p \in \mathcal{P}^\#$
If $p = [QName | \$var]/p'$ **then**
 $eval(p, element\{QName\}\{e'\}) = eval(p', e') \quad \text{-- XQuery}$
 $= eval(p', e) \quad \text{-- I. H.}$
 $= eval(p, (element\{QName\}\{e\})) \quad \text{-- Prop. 2 of } eval$
If $p = step_1/p', step_1 \neq [QName|\$var]$ **then**
 $eval(p, element\{QName\}\{e'\}) = () \quad \text{-- XQuery}$
 $= eval(p, (element\{QName\}\{e\})) \quad \text{-- XQuery}$
If $p = [QName | \$var]$ **then** *No path corresponds, pp17 checked before*
 $-\forall p \in \mathcal{P}$
If $p = [QName | \$var]/p'$ **then**
 $root(eval(p, element\{QName\}\{e'\})) = root(eval(p', e')) \quad \text{-- XQuery}$
 $= root(eval(p', e)) \quad \text{-- I. H.}$
 $= root(eval(p, (element\{QName\}\{e\}))) \quad \text{-- Prop. 2 eval}$
If $p = step_1/p', step_1 \neq [QName|\$var]$ **then**
 $root(eval(p, element\{QName\}\{e'\})) = root(()) \quad \text{-- XQuery}$
 $= root(eval(p, (element\{QName\}\{e\}))) \quad \text{-- Prop. 2 eval}$
If $p = [QName | \$var]$ **then**
 $root(eval(p, element\{QName\}\{e'\})) = root(element\{QName\}\{e'\}) \quad \text{-- XQuery}$
 $= element\{QName\}\{\} \quad \text{-- Prop. root}$
 $= root(element\{QName\}\{e\}) \quad \text{-- Prop. root}$
 $= root(eval(p, (element\{QName\}\{e\}))) \quad \text{-- XQuery}$

4 Optimized pruning

We present in this section extensions on the inference rules detailed in Section 3.3 that may further simplify the queries.

Path refinements. Let us illustrate a first optimization by the following example. Let q be the following XQuery expression :

```
for $j in (for $i in <A><B/><C/></A> return $i)
return $j/B
```

Applying the pruning rule on q , no simplifications would apply and the query remains unchanged. However, one can easily notice that the C elements constructed by the inner *for* are not necessary for the end result and can thus be projected out.

This kind of pruning is not possible using the bottom-up inference rules of Section 3.3. This is mainly due to the fact that when we prune some inner subexpression, we have no information about the outer subexpressions, that may lead to further refinements. For instance, in the query q , when we prune the inner *for* subexpression, the projection path $\$i$ suggests to keep all the bound expression of $\$i$ (by rule *pp1*). Then, when we apply the path $\$j/B$ on the inner *for*, we apply it in fact only on the variable $\$i$ (by rule *pp12*), and we conclude that we must keep the variable $\$i$ (by rule *pp8*). In this way, we fail to refine the path $\$i$ to $\$i/b$ and to detect that only B elements, children of A s, are needed from $\$i$'s content.

In order to detect this kind of pruning opportunity, we have to use information of the entire query during the pruning process.

We are currently extending our rule-based algorithm to take into account such simplification opportunities, using a two-step pruning. In the first step, we apply the pruning as described previously, starting with empty sets of used and returned paths for each variable. In addition, when we have situations in which we apply a path p on a variable reference $\$v$ or on a path p' starting by a variable name $\$v$, we keep in a separate structure a mapping from the variable to the projection path that is applied, i.e. $(\$v \Rightarrow p)$ or $(\$v \Rightarrow p'/p)$, along with its kind (used or returned). Then, these mappings are used to initialize the \mathcal{P} and $\mathcal{P}^\#$ sets for a second pruning pass.

Going back to the example, in the first step q remains unchanged but we extract a mapping $\$i \Rightarrow \{\$i/B\}$ (as returned path). In the second step, when pruning the inner *for*, this path allows us to prune the C elements.

Elimination of useless XPath expressions. The query $Q'_{1.3}$ presented in Section 1 contains a path, $\$j/open_auction$, whose evaluation is not necessary for the end result. This is because it always returns the empty sequence $()$. A second optimization addresses this issue, eliminating irrelevant navigation by directly substituting such paths by the empty sequence $()$.

The simplified query $Q'_{1.3}$ would be the following :

```

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

```

Intuitively, we can find these paths during the *projectPaths* process, when applying paths over expressions. In the same way we retrieve the expressions parts that do match some path, we can also retrieve the paths that do not match anything in these expressions. In the remaining of this section we give the algorithm changes in order to take into account all the optimizations described above. These changes concern the environment, the *projectPaths* and *prune* functions.

4.1 Environment changes

The optimizations we discuss above require some changes on the handling of the environment. Recall that the environment contains a set of mappings between variables and objects to which they are bound in the query. We add a new set of mappings that map variables to two sets of paths, \mathcal{P} and $\mathcal{P}^\#$, where these paths are the ones extracted by the *extractPaths* function. Formally, these mappings are added with an environment function *addVarRef* and retrieved by the function *getVarRef*. The mappings can be enriched by additional paths detected during the *projectPaths* steps, using the *addNewVarRef* function. The *addNewVarRef* adds first the *used* paths then the *return* ones. When adding the *used* paths, *addNewVarRef* checks for each *used* path to add if there is already an occurrence of this path in the existing *return* paths set. If one such path exists, it is removed from the set. We remove these redundant paths from the previous *return* path set because the paths we add are more global.

4.2 Path Projection extensions

The extended *projectPaths* function takes as input two sets of paths and an XQuery expression ; and returns a new expression and a set of paths representing useless paths that can be replaced by (). We present the extended rules.

Literal, comparison, quantifier. For literals, if there is at least one path in $\mathcal{P} \cup \mathcal{P}^\#$ that matches with $\$v$ or *text()* (rule *pp'1*) the set of useless paths is formed by all the paths in $\mathcal{P} \cup \mathcal{P}^\#$ minus these two paths. Otherwise (rule *pp'2*), it is obvious that no path in $\mathcal{P} \cup \mathcal{P}^\#$ matches with the literal, and so all these paths are useless. In the same way, we process quantifier, logical and comparison expressions, and arithmetic expressions.

$$\begin{array}{c}
\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^\#), p = (\$v \mid \text{text}())}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{literal}) \Rightarrow \text{literal}, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, \text{text}()\}} \text{(pp'1)} \\
\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^\#), p \neq (\$v \mid \text{text}())}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{literal}) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{(pp'2)} \\
\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^\#), p = (\$v \mid \text{text}())}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_1 \text{ Op } e_2) \Rightarrow e_1 \text{ Op } e_2, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, \text{text}()\}} \text{(pp'3)} \\
\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^\#), p \neq (\$v \mid \text{text}())}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_1 \text{ Op } e_2) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{(pp'4)} \\
\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^\#), p = (\$v \mid \text{text}())}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{some } \$var \text{ in } e_1 \text{ satisfies } e_2) \Rightarrow \text{some } \$var \text{ in } e_1 \text{ satisfies } e_2, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, \text{text}()\}} \text{(pp'5)} \\
\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^\#), p \neq (\$v \mid \text{text}())}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{some } \$var \text{ in } e_1 \text{ satisfies } e_2) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{(pp'6)}
\end{array}$$

Sequence. In the case of a sequence, a path is useless for a sequence expression q if it is useless for all subsequences q_i . So, the useless path set of a sequence is the *intersection* of the useless path sets of its subsequences.

$$\frac{
\begin{array}{l}
Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_1) \Rightarrow e'_1, Up_1 \\
Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_2) \Rightarrow e'_2, Up_2
\end{array}
}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, (e_1, e_2)) \Rightarrow (e'_1, e'_2), Up_1 \cap Up_2} \text{(pp'7)}$$

Variable reference. The useless paths to return are those that are common to the useless paths returned by *projectPaths* when applied to the objects bound to the variable reference. In the case where no path matches, *projectPaths* returns the set $\mathcal{P} \cup \mathcal{P}^\#$. In addition, the environment function *addNewVarRef* adds new paths to the mappings of $\$var$. We add only the useful paths obtained by subtracting from \mathcal{P} and $\mathcal{P}^\#$ the set of useless paths.

$$\frac{
\begin{array}{l}
(Env.getBind(\$var) \Rightarrow \{o_1, \dots, o_n\}) \\
\exists i, 1 \leq i \leq n, \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, o_i) \Rightarrow o'_i, Up_i \text{ s.t. } o'_i \neq () \\
Env.addNewVarRef(\$var, [\mathcal{P} - \bigcap_{i=1}^n Up_i], [\mathcal{P}^\# - \bigcap_{i=1}^n Up_i])
\end{array}
}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \$var) \Rightarrow \$var, \bigcap_{i=1}^n Up_i} \text{(pp'8)}$$

$$\frac{
\begin{array}{l}
(Env.getBind(\$var) \Rightarrow \{o_1, \dots, o_n\}) \\
\forall i, 1 \leq i \leq n, \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, o_i) \Rightarrow ()
\end{array}
}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \$var) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{(pp'9)}$$

XPath expression. The set of useless paths represent in this case the set of the paths that do not match with the premises.

The function *addNewVarXPathRef* is a variant of *addNewVarRef*, with the difference that before adding the paths, it checks if the path $s_1/\dots/s_n$ starts

by a variable $\$var$. If this is the case, it takes each path of the sets to add, removes its first step (because it matches with s_n) and concatenates it to $s_1/\dots/s_n$. After changing the paths, $addNewVarXPathRef$ adds them to the mapping of $\$var$ à la $addNewVarRef$.

$$\frac{\begin{array}{l} \exists p \in (\mathcal{P} \cup \mathcal{P}^\#), ((s_n = \text{text}()) \wedge (p = \text{text}()|\$v)) \vee \\ ((s_n \neq \text{text}()) \wedge (s_n \neq *) \wedge \text{head}(p) = (*|s_n|\$v)) \vee ((s_n = "*" \wedge (p \neq \text{text}())) \\ Up = p \in \mathcal{P} \cup \mathcal{P}^\#, p \text{ does not satisfy the condition above} \\ Env.addNewVarXPathRef([\mathcal{P} - Up], [\mathcal{P}^\# - Up]) \end{array}}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, s_1/\dots/s_n) \Rightarrow s_1/\dots/s_n, Up} \text{ (pp'10)}$$

$$\frac{\text{otherwise}}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, s_1/\dots/s_n) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{ (pp'11)}$$

FLWR expression. As with sequences, the set of useless path is returned by applying projectPaths on the return expression.

$$\frac{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_3) \Rightarrow e'_3, Up}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{for } \$var \text{ in } e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow \text{for } \$var \text{ in } e_1 \text{ where } e_2 \text{ return } e'_3, Up} \text{ (pp'12)}$$

$$\frac{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_3) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{for } \$var \text{ in } e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{ (pp'13)}$$

$$\frac{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_3) \Rightarrow e'_3, Up}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{let } \$var := e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow \text{let } \$var := e_1 \text{ where } e_2 \text{ return } e'_3, Up} \text{ (pp'14)}$$

$$\frac{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_3) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{let } \$var := e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{ (pp'15)}$$

Conditional. In the case of conditional expressions, a path is useless if it is useless for both the *then* and *else* expressions.

$$\frac{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_2) \Rightarrow e'_2, Up_2 \quad Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e_3) \Rightarrow e'_3, Up_3}{Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, \text{if } (e_1) \text{ then } e_2 \text{ else } e_3) \Rightarrow \text{if } (e_1) \text{ then } e'_2 \text{ else } e'_3, Up_2 \cap Up_3} \text{ (pp'16)}$$

Element construction. For the rules $pp'17$ and $pp'20$ ($pp'18$ and $pp'19$), we proceed in the same manner as in rule $pp'1$ ($pp'2$). In the case of rule $pp'21$, the set of useless paths is computed as following :

1) We compute the set of the paths that do not match with the constructor element. This set represents the elements of $\mathcal{P} \cup \mathcal{P}^\#$ that not starts with $QName$ or $\$v$. We obtain these paths by doing a set subtraction between $\mathcal{P} \cup \mathcal{P}^\#$ and the set obtained from extending (using the *extend* function) the paths of $\mathcal{P}' \cup \mathcal{P}'^\#$ with $QName$ and $\$v$.

- 2) We extend the set of useless paths Up' with $QName$ and $\$v$.
- 3) We do the union of the two sets obtained in steps 1 and 2.
- 4) Extending paths with $QName$ and $\$v$ can lead sometimes to paths that are not in $\mathcal{P} \cup \mathcal{P}^\#$. To ensure that the useless path set we return contains only paths in $\mathcal{P} \cup \mathcal{P}^\#$ we do intersection with this set.

$$\frac{\exists p \in \mathcal{P}^\#, p = (QName | \$v)}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow (pp'17) \frac{element\{QName\}\{e\}, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, QName\}}{}}$$

$$\frac{\forall p \in \mathcal{P} \cup \mathcal{P}^\#, head(p) \neq (QName | \$v)}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\# (pp'18)}$$

$$\frac{\begin{array}{l} \exists p \in \mathcal{P}, p = (QName | \$v) \\ \mathcal{P}' = \{p', [QName | \$v]/p' \in \mathcal{P}\} \\ \mathcal{P}'^\# = \{p', [QName | \$v]/p' \in \mathcal{P}^\#\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^\#, e) \Rightarrow (), Up \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\# (pp'19)}$$

$$\frac{(\exists p \in \mathcal{P}, p = (QName | \$v)) \wedge (\forall \bar{p} \in (\mathcal{P} \cup \mathcal{P}^\#) - \{p\}, head(\bar{p}) \neq (QName | \$v))}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow (pp'20) \frac{element\{QName\}\{e\}, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, QName\}}{}}$$

$$\frac{\begin{array}{l} \exists p \in \mathcal{P} \cup \mathcal{P}^\#, head(p) = (QName | \$v) \\ \mathcal{P}' = \{p', [(/)?QName | \$v]/p' \in \mathcal{P}\} \\ \mathcal{P}'^\# = \{p', [(/)?QName | \$v]/p' \in \mathcal{P}^\#\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^\#, e) \Rightarrow e', Up' \end{array}}{Up = [((\mathcal{P} \cup \mathcal{P}^\#) - (extend((QName | \$v), \mathcal{P}' \cup \mathcal{P}'^\#))) \cup (extend(QName | \$v), Up')] \cap (\mathcal{P} \cup \mathcal{P}^\#) (pp'21) \frac{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e'\}, Up}{}}$$

Empty sequence. This rule is self explanatory.

$$\frac{}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, ()) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\# (pp'22)}$$

4.3 Extended algorithm

To take into account the path refinements extension we need a two steps algorithm described by the following rule :

$$\frac{\begin{array}{l} Env \vdash prune(exp) \Rightarrow exp' \\ Env \vdash prune'(exp') \Rightarrow exp'' \end{array}}{Env \vdash twoStepsPrune(exp) \Rightarrow exp''}$$

By this rule, the algorithm will first simplify the query according to the optimized *prune* function rules, then it applies a second simplification using the *prune'* function which is a slightly modified version of the *prune* function.

The rest of the rules are similar to the *prune* function rules described in the Section 3.

4.3.1 *prune* function.

We change only the rules of the *FLWR* and quantifier expressions, the rest of the rules are similar to the *prune* function rules described in the Section 3.

FLWR and Quantifier expressions. The main changes in the pruning function are done in the *FLWR* expression rules. For instance, in the rule *p12*, we add two judgments that replace in the *where* and *return* expressions, e'_2 and e'_3 respectively, each occurrence of each path in the detected useless paths set by the empty sequence (). We note that the function *Replace* uses the environment to infer if a path expression is relative to a given useless path or not.

For instance, if $\$v/A/B$ is a useless path and if in the expression we have a path $\$v'/B$ such as $\$v'$ is bound in the environment to $\$v/A$, so the function *Replace* infers that $\$v'/B$ is a useless path. We also add a judgment to keep the *used* and the *return* paths of a given variable in the environment. We use in this judgment the function *addVarRef*. We give in the following the new optimized rule *p'12* :

$$\begin{array}{c}
Env \vdash Prune(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\
Env \vdash Prune(e_2) \Rightarrow e'_2 \quad Env \vdash Prune(e_3) \Rightarrow e'_3 \\
Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}_2, \mathcal{P}^{\#}_2 \quad Env \vdash extractPaths(\$v, e'_3) \Rightarrow \mathcal{P}_3, \mathcal{P}^{\#}_3 \\
Env.addVarRef(\$v, \mathcal{P}_2 \cup \mathcal{P}_3 \cup \{\$v\}, \mathcal{P}^{\#}_2 \cup \mathcal{P}^{\#}_3) \\
Env \vdash projectPaths(\mathcal{P}_2 \cup \mathcal{P}_3 \cup \{\$v\}, \mathcal{P}^{\#}_2 \cup \mathcal{P}^{\#}_3, e'_1) \Rightarrow e''_1, Up \\
Replace(e_2, Up) \Rightarrow e''_2 \quad Replace(e_3, Up) \Rightarrow e''_3 \\
\hline
Env \vdash Prune(for \$v in e_1 where e_2 return e_3) \Rightarrow for \$v in e''_1 where e''_2 return e''_3 \quad (p'12)
\end{array}$$

In the following, we give the remaining of the optimized *prune* rules :

$$\begin{array}{c}
Env \vdash Prune(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\
Env \vdash Prune(e_2) \Rightarrow e'_2 \\
Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}_2, \mathcal{P}^{\#}_2 \\
Env.addVarRef(\$v, \mathcal{P}_2 \cup \{\$v\}, \mathcal{P}^{\#}_2) \\
Env \vdash projectPaths(\mathcal{P}_2 \cup \{\$v\}, \mathcal{P}^{\#}_2, e'_1) \Rightarrow e''_1, Up \\
Replace(e_2, Up) \Rightarrow e'_2 \\
\hline
Env \vdash Prune(for \$v in e_1 return e_2) \Rightarrow for \$v in e''_1 return e'_2 \quad (p'9)
\end{array}$$

$$\begin{array}{c}
Env \vdash Prune(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\
Env \vdash Prune(e_2) \Rightarrow e'_2 \quad Env \vdash Prune(e_3) \Rightarrow e'_3 \\
Env \vdash extractPaths(\$v, e'_2) \Rightarrow \mathcal{P}_2, \mathcal{P}^{\#}_2 \quad Env \vdash extractPaths(\$v, e'_3) \Rightarrow \mathcal{P}_3, \mathcal{P}^{\#}_3 \\
Env.addVarRef(\$v, \mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}^{\#}_2 \cup \mathcal{P}^{\#}_3) \\
Env \vdash projectPaths(\mathcal{P}_2 \cup \mathcal{P}_3, \mathcal{P}^{\#}_2 \cup \mathcal{P}^{\#}_3, e'_1) \Rightarrow e''_1, Up \\
Replace(e_2, Up) \Rightarrow e''_2 \quad Replace(e_3, Up) \Rightarrow e''_3 \\
\hline
Env \vdash Prune(let \$v := e_1 where e_2 return e_3) \Rightarrow let \$v := e''_1 where e''_2 return e''_3 \quad (p'16)
\end{array}$$

$$\begin{array}{c}
Env \vdash Prune(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\
Env \vdash Prune(e_2) \Rightarrow e'_2
\end{array}$$

$$\begin{array}{c}
Env \vdash \text{extractPaths}(\$v, e'_2) \Rightarrow \mathcal{P}_2, \mathcal{P}^\#_2 \\
Env.\text{addVarRef}(\$v, \mathcal{P}_2, \mathcal{P}^\#_2) \\
Env \vdash \text{projectPaths}(\mathcal{P}_2, \mathcal{P}^\#_2, e'_1) \Rightarrow e''_1, Up \\
\text{Replace}(e'_2, Up) \Rightarrow e'_2 \\
\hline
Env \vdash \text{Prune}(\text{let } \$v := e_1 \text{ return } e_2) \Rightarrow \text{let } \$v := e''_1 \text{ return } e'_2 \quad (p'19)
\end{array}$$

$$\begin{array}{c}
Env \vdash \text{Prune}(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.\text{saturate}(\text{varRes}(e'_1)))) \\
Env \vdash \text{Prune}(e_2) \Rightarrow e'_2 \\
Env \vdash \text{extractPaths}(\$v, e'_2) \Rightarrow \mathcal{P}_2, \mathcal{P}^\#_2 \\
Env.\text{addVarRef}(\$v, \mathcal{P}_2, \mathcal{P}^\#_2) \\
Env \vdash \text{projectPaths}(\mathcal{P}_2, \mathcal{P}^\#_2, e'_1) \Rightarrow e''_1, Up \\
\text{Replace}(e'_2, Up) \Rightarrow e'_2 \\
\hline
Env \vdash \text{Prune}(\text{some } \$v \text{ in } e_1 \text{ satisfies } e_2) \Rightarrow \text{some } \$v \text{ in } e''_1 \text{ satisfies } e'_2 \quad (p'23)
\end{array}$$

4.3.2 *prune'* function.

This function has the same rules for all the considered expressions, except for the *FLWR* and quantifier expression. The difference is that all the paths to be applied are kept in the environment, so instead of calling the *extractPaths* function for a given variable, we use only the environment function *getVarRef*. Then, we call as usual the *projectPaths* function. The rules are the following :

$$\begin{array}{c}
Env \vdash \text{Prune}(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.\text{saturate}(\text{varRes}(e'_1)))) \\
Env \vdash \text{Prune}(e_2) \Rightarrow e'_2 \quad Env \vdash \text{Prune}(e_3) \Rightarrow e'_3 \\
Env \vdash \text{getVarRef}(\$v) \Rightarrow \mathcal{P}, \mathcal{P}^\# \\
Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e'_1) \Rightarrow e''_1, Up \\
\text{Replace}(e'_2, Up) \Rightarrow e''_2 \quad \text{Replace}(e'_3, Up) \Rightarrow e''_3 \\
\hline
Env \vdash \text{Prune}(\text{for } \$v \text{ in } e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow \text{for } \$v \text{ in } e''_1 \text{ where } e''_2 \text{ return } e''_3 \quad (p''12)
\end{array}$$

$$\begin{array}{c}
Env \vdash \text{Prune}(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.\text{saturate}(\text{varRes}(e'_1)))) \\
Env \vdash \text{Prune}(e_2) \Rightarrow e'_2 \\
Env \vdash \text{getVarRef}(\$v) \Rightarrow \mathcal{P}, \mathcal{P}^\# \\
Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e'_1) \Rightarrow e''_1, Up \\
\text{Replace}(e'_2, Up) \Rightarrow e''_2 \\
\hline
Env \vdash \text{Prune}(\text{for } \$v \text{ in } e_1 \text{ return } e_2) \Rightarrow \text{for } \$v \text{ in } e''_1 \text{ return } e''_2 \quad (p''9)
\end{array}$$

$$\begin{array}{c}
Env \vdash \text{Prune}(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.\text{saturate}(\text{varRes}(e'_1)))) \\
Env \vdash \text{Prune}(e_2) \Rightarrow e'_2 \quad Env \vdash \text{Prune}(e_3) \Rightarrow e'_3 \\
Env \vdash \text{getVarRef}(\$v) \Rightarrow \mathcal{P}, \mathcal{P}^\# \\
Env \vdash \text{projectPaths}(\mathcal{P}, \mathcal{P}^\#, e'_1) \Rightarrow e''_1, Up \\
\text{Replace}(e'_2, Up) \Rightarrow e''_2 \quad \text{Replace}(e'_3, Up) \Rightarrow e''_3 \\
\hline
Env \vdash \text{Prune}(\text{let } \$v := e_1 \text{ where } e_2 \text{ return } e_3) \Rightarrow \text{let } \$v := e''_1 \text{ where } e''_2 \text{ return } e''_3 \quad (p''16)
\end{array}$$

$$\begin{array}{c}
Env \vdash \text{Prune}(e_1) \Rightarrow e'_1 \\
Env = +(\$v \Rightarrow (Env.\text{saturate}(\text{varRes}(e'_1)))) \\
Env \vdash \text{Prune}(e_2) \Rightarrow e'_2
\end{array}$$

$$\frac{\begin{array}{l} Env \vdash getVarRef(\$v) \Rightarrow \mathcal{P}, \mathcal{P}^\# \\ Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e'_1) \Rightarrow e'_1, Up \\ Replace(e'_2, Up) \Rightarrow e''_2 \end{array}}{Env \vdash Prune(let \$v := e_1 return e_2) \Rightarrow let \$v := e'_1 return e''_2} \text{ (p''19)}$$

$$\frac{\begin{array}{l} Env \vdash Prune(e_1) \Rightarrow e'_1 \\ Env = +(\$v \Rightarrow (Env.saturate(varRes(e'_1))) \\ Env \vdash Prune(e_2) \Rightarrow e'_2 \\ Env \vdash getVarRef(\$v) \Rightarrow \mathcal{P}, \mathcal{P}^\# \\ Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e'_1) \Rightarrow e'_1, Up \\ Replace(e'_2, Up) \Rightarrow e''_2 \end{array}}{Env \vdash Prune(some \$v in e_1 satisfies e_2) \Rightarrow some \$v in e'_1 satisfies e''_2} \text{ (p''23)}$$

5 Handling Descendent and Attribute axis

We present in this section how do we handle descendent and attribute axis in our algorithm, considering the following extensions on the XQuery fragment :

$$\begin{array}{ll} exp & := \langle QName(Att_{exp}) * \rangle \{exp\} \langle /QName \rangle \\ Att_{exp} & := QName \text{ " = " } exp \\ Path & := (doc(uri) | \$QName | exp) (/ | //) Step | doc(uri) \\ Step & := NodeTest((/ | //) Step)? | text() | @NodeTest \end{array}$$

We add the following normalization rule :

$$\langle QName att_i = v_i \rangle \{e\} \langle /QName \rangle \Rightarrow element\{QName\}\{attribute\{att_i\}\{v_i\}, e\}$$

This rule transforms all direct attribute constructions to compute ones, in order to facilitate their processing.

To process descendant and attribute axis we need to enrich the *projectPaths* rules with some conditions, and add an additional *prune* rule for attribute construction.

5.1 Path projection

In this section we present only the modified rules for descendant and attribute axis handling.

Literal, comparison, quantifier. For literals, if there is at least one path in $\mathcal{P} \cup \mathcal{P}^\#$ that matches with $\$v$ or $(/)?text()$ ⁴ (rule *pp''1*) the set of useless paths is formed by all the paths in $\mathcal{P} \cup \mathcal{P}^\#$ minus these two paths. Otherwise (rule *pp''2*), it is obvious that no path in $\mathcal{P} \cup \mathcal{P}^\#$ matches with the literal, and hence these paths are useless. In the same way, we process quantifier, logical or comparison expressions, and arithmetic expressions.

$$\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^\#), p = (\$v | (/)?text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, literal) \Rightarrow literal, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, text()\}} \text{ (pp''1)}$$

4. Here $(/)?$ means that the `text()` test can follow a descendant axis.

$$\begin{array}{c}
\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^\#), p \neq (\$v \mid (/)?text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, literal) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{ (pp''2)} \\
\\
\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^\#), p = (\$v \mid (/)?text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e_1 Op e_2) \Rightarrow e_1 Op e_2, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, text()\}} \text{ (pp''3)} \\
\\
\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^\#), p \neq (\$v \mid (/)?text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, e_1 Op e_2) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{ (pp''4)} \\
\\
\frac{\exists p \in (\mathcal{P} \cup \mathcal{P}^\#), p = (\$v \mid (/)?text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, some \$var \text{ in } e_1 \text{ satisfies } e_2) \Rightarrow some \$var \text{ in } e_1 \text{ satisfies } e_2, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, text()\}} \text{ (pp''5)} \\
\\
\frac{\forall p \in (\mathcal{P} \cup \mathcal{P}^\#), p \neq (\$v \mid (/)?text())}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, some \$var \text{ in } e_1 \text{ satisfies } e_2) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{ (pp''6)}
\end{array}$$

XPath expression. To process the attribute and descendent axis we verify first some conditions : we check if the path to project on does not return an attribute (*i.e.* $s_n \neq @ \dots$) then we apply the conditions we have seen in rule (pp10), to which we add a new condition that process descendent axis. This additional condition checks if there is a path following a descendent axis, $head(p) = / \dots$. In this case we return the path $s_1 / \dots / s_n$ because at this level we have not enough information about the elements returned by $s_1 / \dots / s_n$. In the case when s_n starts by @, the only way to return an attribute is to have paths of the form $(/)?@ (s_n | *)$.

$$\begin{array}{c}
\frac{\begin{array}{l} \exists p \in (\mathcal{P} \cup \mathcal{P}^\#), [(s_n \neq @ \dots) \wedge [(head(p) = (*|s_n|\$var) \wedge s_n \neq text()) \\ \vee (s_n = " * " \wedge p \neq text()) \vee (p = text()|\$var \wedge s_n = text()) \vee (head(p) = / \dots)]] \vee \\ [[(s_n = @s) \wedge p = (/)?(@s|@*)] \vee [(s_n = @*) \wedge (p = (/)?@ \dots)]] \\ Up = p \in \mathcal{P} \cup \mathcal{P}^\#, p \text{ does not satisfy the condition above} \\ Env.addNewVarXPathRef([\mathcal{P} - Up], [\mathcal{P}^\# - Up]) \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, s_1 / \dots / s_n) \Rightarrow s_1 / \dots / s_n, Up} \text{ (pp''10)} \\
\\
\frac{\text{otherwise}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, s_1 / \dots / s_n) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{ (pp''11)}
\end{array}$$

Element construction. To process the descendent axis, when we build the sets \mathcal{P}' and $\mathcal{P}'^\#$, if the path starts with /, which means that it is a descendent axis, we add this path to \mathcal{P}' or $\mathcal{P}'^\#$ (according to its kind) in addition to the path we obtained by taking out its first step. If the path has the form s_1 / s_2 , we add $/s_2$ in order to record that the step s_2 follows a descendent axis.

$$\begin{array}{c}
\frac{\exists p \in \mathcal{P}^\#, p = ((/)?QName \mid \$v)}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e\}, \mathcal{P} \cup \mathcal{P}^\# - \{\$v, (/)?QName\}} \text{ (pp''17)} \\
\\
\frac{\forall p \in \mathcal{P} \cup \mathcal{P}^\#, head(p) \neq ((/)?QName \mid \$v)}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{ (pp''18)}
\end{array}$$

$$\frac{\begin{array}{l} \bar{\Delta} p \in \mathcal{P}, p = (/?QName | \$v) \\ \mathcal{P}' = \{p', [(/?QName | \$v)/p' \in \mathcal{P} \vee p' = /QName \dots \in \mathcal{P}]\} \\ \mathcal{P}'^\# = \{p', [(/?QName | \$v)/p' \in \mathcal{P}^\# \vee p' = /QName \dots \in \mathcal{P}^\#]\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^\#, e) \Rightarrow (), Up \end{array}}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{(pp''19)}$$

$$\frac{(\exists p \in \mathcal{P}, p = (/?QName | \$v)) \wedge (\forall \bar{p} \in (\mathcal{P} \cup \mathcal{P}^\#) - \{p\}, head(\bar{p}) \neq (/?QName | \$v))}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e\}, \mathcal{P} \cup \mathcal{P}^\# - \{p, QName\}} \text{(pp''20)}$$

$$\frac{\begin{array}{l} \exists p \in \mathcal{P} \cup \mathcal{P}^\#, head(p) = (/?QName | \$v) \\ \mathcal{P}' = \{p', [(/?QName | \$v)/p' \in \mathcal{P} \vee p' = /QName \dots \in \mathcal{P}]\} \\ \mathcal{P}'^\# = \{p', [(/?QName | \$v)/p' \in \mathcal{P}^\# \vee p' = /QName \dots \in \mathcal{P}^\#]\} \\ Env \vdash projectPaths(\mathcal{P}', \mathcal{P}'^\#, e) \Rightarrow e', Up' \end{array}}{Up = [((\mathcal{P} \cup \mathcal{P}^\#) - (extend^s((QName | \$v), \mathcal{P}' \cup \mathcal{P}'^\#))) \cup (extend(QName | \$v), Up')] \cap (\mathcal{P} \cup \mathcal{P}^\#)} \text{(pp''21)}$$

$$Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, element\{QName\}\{e\}) \Rightarrow element\{QName\}\{e'\}, Up$$

Attribute construction. The only way to return an attribute is to have a path of the forme $(/?@((QName|*) | \$v))$, and the useless paths in this case are those that do not obey condition. Otherwise we return $()$.

$$\frac{\exists p \in \mathcal{P} \cup \mathcal{P}^\#, p = (/?@((QName|*) | \$v))}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, attribute\{QName\}\{e\}) \Rightarrow attribute\{QName\}\{e\}, \mathcal{P} \cup \mathcal{P}^\# - \{/?@((QName|*) | \$v)\}} \text{(pp''23)}$$

$$\frac{\forall p \in \mathcal{P} \cup \mathcal{P}^\#, p \neq (/?@((QName|*) | \$v))}{Env \vdash projectPaths(\mathcal{P}, \mathcal{P}^\#, attribute\{QName\}\{e\}) \Rightarrow (), \mathcal{P} \cup \mathcal{P}^\#} \text{(pp''24)}$$

5.2 Pruning

Attribute construction. We proceed in the same manner as in the element construction pruning rule.

$$\frac{Env \vdash Prune(e) \Rightarrow e'}{Env \vdash Prune(attribute\{QName\}\{e\}) \Rightarrow attribute\{QName\}\{e'\}} \text{(p26)}$$

6 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' , measuring 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 :

```

let $q := <personInf>
  {for $i in doc("xmark.xml")/site/people/person
  return
    (<name > {test_exp} </name >,
     <age > {test_exp} </age >,
     <gender > {test_exp} </gender >,
     <email > {test_exp} </email >)}
  </personInf>
for $j in $q
return($j/names?, $j/age?, $j/gender?, $j/email?)

```

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 nodes of *personInf* depends on the size of the sequence to which the variable $\$i$ is bound (*person* elements) and varies with the size of the document on which the test is 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 element of the *personInf* element. When testing the gain for 50% of irrelevant intermediate results, we can use two paths, such as $\$j/age$ 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 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[FER 07] (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 Linux PC with 2Gb of memory.

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 savings in evaluation time seems to be linked to the amount of pruned intermediate results. These savings increase 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.

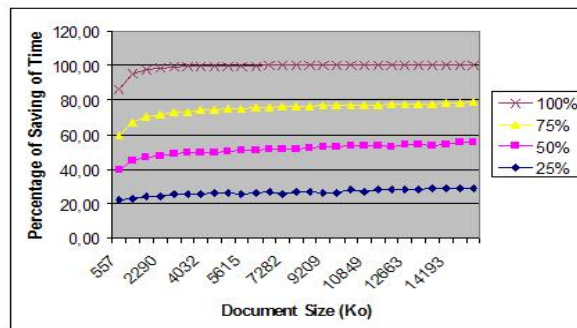


Figure 3. Test results for queries pruning FLWR blocks

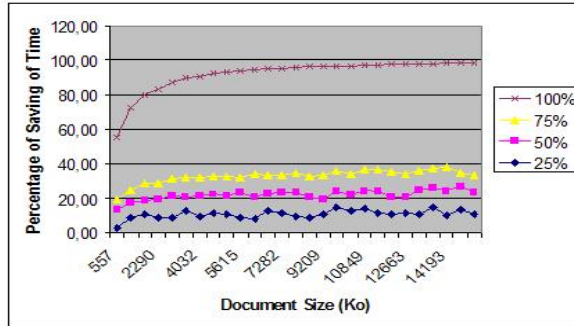


Figure 4. Test results for queries pruning variable XPaths

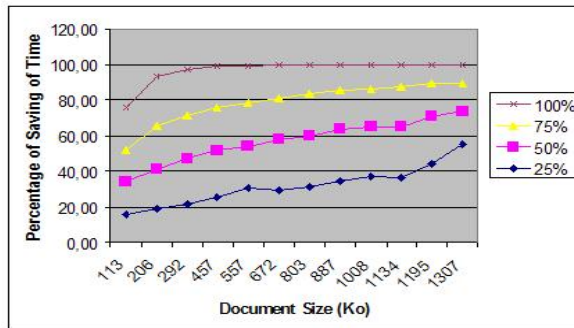


Figure 5. Test results for queries pruning document XPaths

7 Conclusion

We present in this paper a rewriting algorithm for XQuery queries which prunes from subexpressions the computations that are irrelevant for the overall query result. Our algorithm generates for each input query an equivalent output query. We show by extensive experiments the important savings in evaluation time, and we prove formally the correctness of our algorithm.

8 Bibliographie

- [ABI 04] ABITEBOUL S., BENJELLOUN O., CAUTIS B., MANOLESCU I., MILO T., PREDAN N., « Lazy query evaluation for Active XML », *SIGMOD Conf*, 2004.
- [BEN 06] BENZAKEN V., CASTAGNA G., COLAZZO D., NGUYEN K., « Type-Based XML Projection », *VLDB Conf*, 2006.
- [BRA 07] BRANTNER M., KANNE C.-C., MOERKOTTE G., « Let a Single FLWOR Bloom (to improve XQuery plan generation) », *XSym Workshop*, 2007.
- [CAR 00] CAREY M. J., KIERNAN J., SHANMUGASUNDARAM J., SHEKITA E. J., SUBRAMANIAN S. N., « XPERANTO : Middleware for Publishing Object-Relational Data as XML Documents », *VLDB Conf*, 2000.
- [DEU 04] DEUTSCH A., PAPAKONSTANTINOY Y., XU Y., « The NEXT Logical Framework for XQuery. », *VLDB Conf*, 2004.
- [DON 04] DONG X., HALEVY A. Y., TATARINOV I., « Containment of Nested XML Queries », *VLDB Conf*, 2004.
- [DRA 07] DRAPER D., FANKHAUSER P., FERNÁNDEZ M. F., MALHOTRA A., ROSE K., RYS M., SIMÉON J., WADLER P., « XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation », 2007.
- [FER 02] FERNÁNDEZ M. F., KADIYSKA Y., SUCIU D., MORISHIMA A., TAN W. C., « SilkRoute : A framework for publishing relational data in XML », *ACM Trans. Database Syst.*, vol. 27, n° 4, 2002.
- [FER 07] FERNÁNDEZ M. F., SIMÉON J., « The Galax System "The XQuery Implementation for Discriminating Hackers" Version 0.7.2 », 2007.
- [GRI 04] GRINEV M., « XQuery Optimizing Based on Rewriting », *ADBS*, 2004.
- [HAA 05] HAAS L. M., HERNÁNDEZ M. A., HO H., POPA L., ROTH M., « Clio grows up : from research prototype to industrial tool », *SIGMOD Conf*, 2005.
- [KOC 05] KOCH C., « On the role of Composition in XQuery. », *WebDB Workshop*, 2005.
- [MAL 07] MALHOTRA A., MELTON J., WALSH N., « XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation », 2007.
- [MAN 01] MANOLESCU I., FLORESCU D., KOSSMANN D., « Answering XML Queries on Heterogeneous Data Sources », *VLDB Conf*, 2001.
- [MAR 03] MARIAN A., SIMÉON J., « Projecting XML Documents », *VLDB Conf*, 2003.
- [MIC 03] MICHIELS P., « XQuery Optimization. », *VLDB PhD Workshop*, 2003.
- [MIC 07] MICHIELS P., MIHAILA G. A., SIMÉON J., « Put a Tree Pattern in Your Algebra », *ICDE Conf*, 2007.
- [RAM 02] RAMANAN P., « Efficient Algorithms for Minimizing Tree Pattern Queries. », *SIGMOD Conf*, 2002.
- [SCH 02] SCHMIDT A., WAAS F., KIRSTEN M., J.CAREY M., MANOLESCU I., BUSSE R., « XMark : A Benchmark for XML Data Management », *VLDB Conf*, 2002.
- [SHA 01] SHANMUGASUNDARAM J., KIERNAN J., SHEKITA E. J., FAN C., FUNDERBURK J., « Querying XML Views of Relational Data », *VLDB Conf*, 2001.
- [TAT 04] TATARINOV I., HALEVY A. Y., « Efficient Query Reformulation in Peer-Data Management Systems », *SIGMOD Conf*, 2004.