XPath: Looking Forward

Dan Olteanu, Holger Meuss, Tim Furche, François Bry

Insitute for Computer Science and Center for Information and Language Processing
University of Munich, Germany
olteanu@informatik.uni-muenchen.de

Abstract

The location path language XPath is of particular importance for XML applications since it is a core component of many XML processing standards such as XSLT or XQuery. In this paper, based on axis symmetry of XPath, equivalences of XPath 1.0 location paths involving reverse axes, such as `ancestor` and `preceding`, are established. These equivalences are used as rewriting rules in an algorithm for transforming location paths with reverse axes into equivalent reverse-axis-free ones. Location paths without reverse axes, as generated by the presented rewriting algorithm, enable efficient SAX-like streamed data processing of XPath.

1 Introduction

Query languages for XML and semistructured data rely on location paths for selecting nodes in data items. In particular, XQuery [21] and XSLT [19] are based on XPath [18]. XPath takes a navigational approach for specifying the nodes to be selected, hence a large number of navigational axes (e.g. `child`, `descendant`, `preceding`) have been defined in XPath. The number as well as the relevance of these navigational axes for querying XML has been challenged in [8, 21, 23].

The random access to XML data that is enabled by the various navigational axes of XPath has proven particularly difficult for an efficient stream-based processing of XPath queries. Processing of XML has seen the widespread use of the W3C document object model (DOM) [20], where an in-memory representation of the entire XML data is used. As DOM has been developed with focus on document processing in user agents (e.g. browsers), this approach has several shortcomings for other application areas:

First, a considerable amount of XML applications, in particular data-centric applications, handle documents too large to be processed in memory. Such documents are often encountered in natural language processing [11], in biological [13] and astronomical [1] projects.

Second, the need for progressive processing (also referred to as sequential processing) of XML has emerged: Stream-based processing generating partial results as soon as they are available gives rise to a more efficient evaluation in certain contexts, e.g.:

- For selective dissemination of information (SDI), documents have to be filtered according to complex requirements specified as XPath queries before being distributed to the subscribers [7, 4]. The routing of data to selected receivers is also becoming increasingly important in the context of web service architectures.
- To integrate data over the Internet, in particular from slow sources, it is desirable to progressively process the input before the full data is retrieved [14, 10].
- As a general processing scheme for XML, several solutions for pipelined processing have been suggested, where the input is sent through a chain of processors each of which taking the output of the preceding processor as input, e.g. Apache Cocoon [2] and XPipe [15].

- Progressive rendering of large documents, e.g. by means of XSL(T) (cf. Requirement 19 of [23]). There have been several attempts to solve this problem [3].

There is a great interest in the identification of a subset of XPath that allows efficient progressive or stream-based processing (cf. [8] and Requirement 19 of [23]).

For stream-based processing of XML data, the Simple API for XML (SAX) [16] has been specified. Of particular concern for progressive SAX-like processing are the reverse axes of XPath, i.e. those navigational axes (e.g. `parent`, `preceding`) that select nodes occuring before the context node in document order. A restriction to forward axes (i.e. axes selecting only nodes after the context node) in location paths is a straightforward consideration for an efficient stream-based evaluation of XPath queries [8].

There are three principal options how to evaluate reverse axes in a stream-based context:

- Storing in memory sufficient information that allows to access past events when evaluating a reverse axis. This amounts to keeping in memory a (possibly pruned) DOM representation of the data [3].
- Evaluating an XPath expression in more than one run. With this approach, it is also necessary to store additional information to be used in successive runs. This information can be considerably smaller than what is needed in the first approach.
- Replacing XPath expressions by equivalent ones without reverse axes.

In this paper it is shown that the third approach is possible. It is less time consuming than the second approach and does not require the in-memory storage of fragments of the input as the first approach does. Hence, XPath can be evaluated without restriction on the use of reverse axes.

Section 2 specifies the location path language considered in the rest of the paper. Then, the notion of equivalence between location paths is defined in Section 3 using a formal model and a denotational semantics for XPath based on [24, 25]. Furthermore, two sets of equivalences (with rather different properties) are established. These equivalences are used as rewriting rules in an algorithm, called "rare", for transforming absolute XPath location paths with reverse axes into equivalent reverse-axis-free ones (Section 4). Two rewritings, based on the two rule sets, are considered. In Section 5, related work is discussed. Section 6 is a conclusion.

Due to space limitations, parts of this work have been omitted, most notably the proofs for the equivalences. They can be found in the full version [17] of this paper.

Some familiarity with XPath 1.0 is assumed.

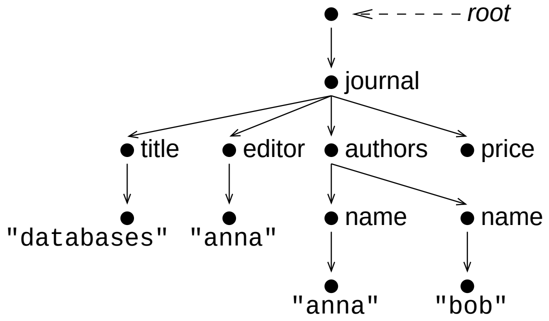
2 Preliminaries

In this paper, specificities of XML that are irrelevant to the issue of concern are left out. Thus, namespaces, comments, processing instructions, attributes, attribute values, document collections, schema types, references, and white space processing are not considered. The results given in this paper extend straightforwardly to unrestricted XML documents.

The root node of a document corresponds to the document node of DOM and of the XQuery 1.0 and XPath 2.0 Data Model [22] – i.e. it is none of the document elements. A leaf is an empty element or a text node – cf. Figure 1.

The mathematical model used in this paper is adapted from [24, 25]. The full formal model as well as the denotational semantics can be found in the full version [17] of this paper. It consists of mathematical functions that can be seen as (formal specifications of) elementary procedures.

Figure 1 Tree and XML data it represents




```
<journal>
  <title>databases</title>
  <editor>anna</editor>
  <authors>
    <name>anna</name>
    <name>bob</name>
  </authors>
  <price />
</journal>
```

2.1 Location Path Language

The location path language considered in the following is unabbreviated XPath without those constructs (such as those needed for processing attributes) irrelevant to the issue of concern. For convenience, this language will be referred to as xPath. Recall that every abbreviated XPath expression can easily be translated into an unabbreviated XPath expression. It is worth stressing that the results given below for xPath extend to XPath 1.0 [18]. The (abstract) syntax of xPath is as follows:

$$
\begin{array}{rcl}
\mathit{path} & ::= & \mathit{path}\ \mathtt{|}\ \mathit{path} \mid \mathtt{/}\ \mathit{path} \mid \mathit{path}\ \mathtt{/}\ \mathit{path} \mid \mathit{path}\ \mathtt{[}\ \mathit{qualif}\ \mathtt{]} \mid \mathit{axis}\ \mathtt{::}\ \mathit{nodetest} \mid \perp \ . \\
\mathit{qualif} & ::= & \mathit{qualif}\ \mathtt{and}\ \mathit{qualif} \mid \mathit{qualif}\ \mathtt{or}\ \mathit{qualif} \mid \mathtt{(}\ \mathit{qualif}\ \mathtt{)} \mid \\
 & & \mathit{path}\ \mathtt{=}\ \mathit{path} \mid \mathit{path}\ \mathtt{==}\ \mathit{path} \mid \mathit{path}\ . \\
\mathit{axis} & ::= & \mathit{reverse\_axis} \mid \mathit{forward\_axis}\ . \\
\mathit{reverse\_axis} & ::= & \mathtt{parent} \mid \mathtt{ancestor} \mid \mathtt{ancestor\text{-}or\text{-}self} \mid \\
 & & \mathtt{preceding} \mid \mathtt{preceding\text{-}sibling}\ . \\
\mathit{forward\_axis} & ::= & \mathtt{self} \mid \mathtt{child} \mid \mathtt{descendant} \mid \mathtt{descendant\text{-}or\text{-}self} \mid \\
 & & \mathtt{following} \mid \mathtt{following\text{-}sibling}\ . \\
\mathit{nodetest} & ::= & \mathit{tagname} \mid \mathtt{*} \mid \mathtt{text()} \mid \mathtt{node()}\ .
\end{array}
$$

As XPath 2.0 and in contrast to XPath 1.0, xPath allows the union $p_1\ \mathtt{|}\ p_2$ of two paths at every level. Such paths can easily be transformed into paths with unions at top level only. Note also that while we do not consider functions in the following sections, the results almost immediately apply to location paths with functions. The only class of functions that needs special treatment are functions for accessing the context position or size of a node.

$\perp$ is convenient for simplifying proofs. It is used as a canonical equivalent path to the xPath expressions that select no nodes whatever the context node and document are, e.g. `/parent::*`.

p_1 `==` p_2 expresses node equality based on identity. Thus, if p_1 and p_2 are two paths, then p_1 `==` p_2 holds if there is a node selected by p_1 which is *identical* to a node selected by p_2. `==` corresponds to built-in node equality operator (==) in XPath 2.0 and XQuery 1.0, but it can also be used for comparing node sets similar to general comparisons in XPath 2.0. As XPath 1.0 has built-in support for equality based on node values only, the XPath 1.0 expression `count(`p_1 `|` p_2`) < count(`p_1`) + count(`p_2`)` can be used for expressing `==`.

A *path* expression will be called a "location path", or "path" for short. A *qualif* expression is a "qualifier" (or pattern). Expressions *axis*`::`*nodetest* and *axis*`::`*nodetest*`[`*qualif*`]` are "steps", also called "location steps". The length of a location path is the number of location steps it contains outside and inside qualifiers. Note that every location path is a qualifier, but the converse is false.

Absolute location paths are recursively defined as follows: A disjunctive path, i.e. a path of the form $p_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ p_i\ \mathtt{|}\ \ldots\ \mathtt{|}\ p_k$, is an absolute path if for all $i = 1, \ldots, k$, p_i is an absolute path.

A non-disjunctive path is an absolute path if it is of the form $/p$, where p is a path. A location path, which is not an absolute path, is a "relative path". A step is a "forward step", if its axis is a forward axis, or a "reverse step", if its axis is a reverse axis.

The axes of the following pairs are "symmetrical" of each other: `parent` – `child`, `ancestor` – `descendant`, `descendant-or-self` – `ancestor-or-self`, `preceding` – `following`, `preceding-sibling` – `following-sibling`, and (useful in proofs) `self` – `self`.

[24] and [25] give a denotational semantics for XPath, which is slightly modified for our purpose in [17]. The semantics defines a function $\mathcal{S}$ that assigns a set of nodes to a location path and a context node: $\mathcal{S}[\![p]\!]x$ is the set of nodes selected by p from node x.

3 Location Path Equivalences

A set of simple equivalences is first established. These are then used to prove equivalences of paths with reverse axes. We distinguish between general equivalences that can be applied to remove any reverse axis, and specific equivalences, each of them being applicable to a certain case. Making use of the semantics of xPath given in the full version [17] of this paper, the equivalence of location paths can be formally defined as follows.

Definition 3.1 (Path equivalence). Two location paths p_1 and p_2 are *equivalent,* noted $p_1 \equiv p_2$, if $\mathcal{S}[\![p_1]\!] = \mathcal{S}[\![p_2]\!]$, i.e. if $\mathcal{S}[\![p_1]\!]x = \mathcal{S}[\![p_2]\!]x$ for all nodes x (from any document).

Intuitively, two location paths are equivalent if they select the same set of nodes for every document and every context node in this document.

Lemma 3.1. *Let p, p_1, and p_2 be location paths, q, q_1, and q_2 qualifiers, n a node test, and $\theta \in \{\texttt{==}, \texttt{=}\}$.*

1. **Right step adjunction:** *If $p_1 \equiv p_2$ and p relative, then $p_1/p \equiv p_2/p$.*
2. **Left step adjunction:** *If $p_1 \equiv p_2$ and p_1, p_2 relative, then $p/p_1 \equiv p/p_2$.*
3. **Qualifier adjunction:** *If $p_1 \equiv p_2$, then $p_1[q] \equiv p_2[q]$ and $p[p_1] \equiv p[p_2]$.*
4. **Relative/absolute path conversion:** *If $p_1 \equiv p_2$, then $/p_1 \equiv /p_2$.*
5. **Qualifier flattening:** $p[p_1/p_2] \equiv p[p_1[p_2]]$.
6. **Ancestor-or-self axis decomposition:**
   $\texttt{ancestor-or-self::}n \equiv \texttt{ancestor::}n \mid \texttt{self::}n$.
7. **Descendant-or-self axis decomposition:**
   $\texttt{descendant-or-self::}n \equiv \texttt{descendant::}n \mid \texttt{self::}n$.
8. **Qualifiers with joins:** $p[p_1\ \theta\ /p_2] \equiv p[p_1[\texttt{self::node()}\ \theta\ /p_2]]$.

Equivalences involving complex qualifiers and unions can be found in the full version [17] of this paper.

Recall that $\perp$ is a location path never selecting any node whatever the context node and document are. Since the root node has no parents and therefore no siblings, the following holds:

Lemma 3.2. *Let m and n be node tests, i.e. m and n are tag names or one of the xPath constructs* `*`, `node()`, *or* `text()`.

- *Let a be one of the axes* `parent`, `ancestor`, `preceding`, `preceding-sibling`, `self`, `following`, *or* `following-sibling`. *Then the following holds:*

$$/a{::}n \equiv \begin{cases} / & \text{if } a = \texttt{self} \text{ and } n = \texttt{node()} \\ \perp & \text{otherwise} \end{cases}$$

- *Let a be the* `preceding` *or* `ancestor` *axis. Then the following equivalences hold:*

$$\texttt{/child::}m/a{::}n \equiv \begin{cases} \texttt{/self::node()[child::}m\texttt{]} & \text{if } a = \texttt{ancestor} \text{ and } n = \texttt{node()} \\ \perp & \text{otherwise} \end{cases}$$

$$\texttt{/child::}m[a{::}n] \equiv \begin{cases} \texttt{/child::}m & \text{if } a = \texttt{ancestor} \text{ and } n = \texttt{node()} \\ \perp & \text{otherwise} \end{cases}$$

3.1 General equivalences

The nodes selected by a reverse step within a location path are necessarily descendants of the document root. The following equivalences show how for any reverse axis only those descendants of the root can be selected that are also matched by the original reverse step.

Proposition 3.1. *Let p and s be relative location paths, n and m node tests, a_m a reverse axis, a_n a forward axis, and b_m the symmetrical axis of a_m. Then the following holds*

$$p[a_m{::}m/s] \equiv p\texttt{[/descendant::}m[s]/b_m\texttt{::node() == self::node()]} \tag{1}$$

$$/p/a_n{::}n/a_m{::}m \equiv \texttt{/descendant::}m[b_m{::}n \texttt{ == } /p/a_n{::}n] \tag{2}$$

$$/a_n{::}n/a_m{::}m \equiv \texttt{/descendant::}m[b_m{::}n \texttt{ == } /a_n{::}n] \tag{2a}$$

Equivalence (1) shows that it is possible to remove the first step in a location path within a qualifier. With help of Lemma 3.1.5 this result is generalized to reverse steps having an arbitrary position within a qualifier.

The key idea of Equivalence (1) is that, instead of looking back from the context node specified by path p for matching a certain node (a_m`::`m), one can look forward from the beginning of the document for matching the node (`/descendant::`m) and then, still forward, for reaching the initial context node (b_m`::node()`). Hence, e.g. instead of checking whether the context node specified by path p has a preceding m (p`[preceding::`m`]`), one rather looks for an m node and then for a following node that is identical to the context node:

$$p\texttt{[/descendant::}m\texttt{/following::node() == self::node()]}.$$

Equivalence (2) removes the first reverse step from an absolute location path using the same underlying idea.

Note that the equality occurring in these equivalences is based on node identity. The equivalent paths might remain expensive to evaluate, but no evaluation of the a_m`::`m reverse step is needed anymore.

Example 3.1. Consider the example of Figure 1 and a query asking for all `name`s that appear before a `price`. A way to select these nodes is using the following location path:

```
/descendant::price/preceding::name
```

By Equivalence (2a), the `preceding` axis can be removed yielding to the following equivalent location path:

```
/descendant::name[following::price == /descendant::price]
```

While the initial location path selects all `name` nodes preceding a `price` node, the equivalent location path selects all `name` nodes, that have a following `price` node, if that node is also a descendant of the root. It is obvious, that there is a considerably simpler equivalent location path (dropping the join), `/descendant::name[following::price]`. The need for the join arises, as the location path selecting the context nodes, relative to which the reverse step is evaluated, (in this case the `price` nodes) can be arbitrarily complex:

Consider a slightly modified case of the previous one, where only `price`s, that are inside a `journal` with a `title`, should be considered. A possible location path for this query with reverse axis is:

```
/descendant::journal[child::title]/descendant::price/preceding::name
```

Again, by Equivalence (2) this is equivalent to

```
/descendant::name[following::price == /descendant::journal[child::title]/descendant::price].
```

As argued above, it is impossible in this case to remove the introduced join. Note that the join in the first example can be removed by additional equivalence rules for simplifying location paths that are outside the scope of this paper.

Using the equivalences above, it is possible to replace reverse steps in xPath expressions. Nonetheless, in the following section specific equivalences for reverse axes are given, that yield to location paths without joins.

3.2 Specific Equivalences

In this section the interaction of the reverse axes (`ancestor`, `ancestor-or-self`, `parent`, `preceding`, and `preceding-sibling`) with forward axes is treated, i.e. equivalences are given, that (if read as rewriting rules from left to right), depending on the location step L_f before a reverse location step L_r, either replace the reverse location step L_r or rewrite the location path into one, where the reverse step L_r is "pushed leftwise". For every reverse step the interaction with every forward step is shown.

In general, the equivalences have the following structure

$$p/L_f/L_r \equiv p' \quad \text{or} \quad p/L_f[L_r] \equiv p',$$

where p is an absolute path, L_f a forward location step, L_r a reverse location step, and p' the equivalent location path. Sometimes the equivalences can be formulated without the leading path p.

Note that interaction with reverse axes, e.g. interaction of `parent` with `preceding-sibling`, is not necessary to investigate in these equivalences due to the way our algorithm works (removing reverse steps from left to right of the location path in question). Also, equivalences involving `ancestor-or-self` and `descendant-or-self` are not necessary since these location steps can be replaced using Equivalences (3.1.6) and (3.1.7).

Some of the following equivalences do still contain reverse steps on the right-hand side, but these reverse steps are either more on the left of the location path, or the right-hand side is of a form, where other equivalences can be applied to fully remove the reverse location steps as elaborated in Section 4.

3.2.1 Parent

The equivalences in the following proposition are divided in two sets. The first set (Equivalences (3) to (7)) covers the case of `parent` location steps outside, the second inside a qualifier. Note that there is a strong structural similarity between the equivalences of the two sets.

Proposition 3.2 (parent axis). *Let m and n be node tests and p a location path.*

$$\begin{aligned}
\texttt{descendant::}n\texttt{/parent::}m &\equiv \texttt{descendant-or-self::}m\texttt{[child::}n\texttt{]} && (3)\\
\texttt{child::}n\texttt{/parent::}m &\equiv \texttt{self::}m\texttt{[child::}n\texttt{]} && (4)\\
p\texttt{/self::}n\texttt{/parent::}m &\equiv p\texttt{[self::}n\texttt{]/parent::}m && (5)\\
p\texttt{/following-sibling::}n\texttt{/parent::}m &\equiv p\texttt{[following-sibling::}n\texttt{]/parent::}m && (6)\\
p\texttt{/following::}n\texttt{/parent::}m &\equiv p\texttt{/following::}m\texttt{[child::}n\texttt{]} && (7)\\
&\mid p\texttt{/ancestor-or-self::*[following-sibling::}n\texttt{]}\\
&\quad\texttt{/parent::}m
\end{aligned}$$

$$\begin{aligned}
\texttt{descendant::}n\ \texttt{[parent::}m\texttt{]} &\equiv \texttt{descendant-or-self::}m\texttt{/child::}n && (8)\\
\texttt{child::}n\texttt{[parent::}m\texttt{]} &\equiv \texttt{self::}m\texttt{/child::}n && (9)\\
p\texttt{/self::}n\texttt{[parent::}m\texttt{]} &\equiv p\texttt{[parent::}m\texttt{]/self::}n && (10)\\
p\texttt{/following-sibling::}n\texttt{[parent::}m\texttt{]} &\equiv p\texttt{[parent::}m\texttt{]/following-sibling::}n && (11)\\
p\texttt{/following::}n\texttt{[parent::}m\texttt{]} &\equiv p\texttt{/following::}m\texttt{/child::}n && (12)\\
&\mid p\texttt{/ancestor-or-self::*[parent::}m\texttt{]}\\
&\quad\texttt{/following-sibling::}n
\end{aligned}$$

Example 3.2. Consider the data of Figure 1. The following location path selects all `editor`s of `journal`s:

`/descendant::editor[parent::journal].`

According to Equivalence (8), this path is equivalent to:

`/descendant-or-self::journal/child::editor.`

3.2.2 Ancestor

The following proposition gives equivalences that either move an `ancestor` step to the left of a path or remove it completely. Equivalences (13a) and (18a) are special cases of Equivalences (13) and (18), respectively.

Proposition 3.3 (`ancestor` axis). *Let m and n be node tests and p a location path.*

$$\begin{aligned}
p\texttt{/descendant::}n\texttt{/ancestor::}m &\equiv p\texttt{[descendant::}n\texttt{]/ancestor::}m && (13)\\
&\mid p\texttt{/descendant-or-self::}m\texttt{[descendant::}n\texttt{]}\\
\texttt{/descendant::}n\texttt{/ancestor::}m &\equiv \texttt{/descendant-or-self::}m\texttt{[descendant::}n\texttt{]} && (13a)\\
p\texttt{/child::}n\texttt{/ancestor::}m &\equiv p\texttt{[child::}n\texttt{]/ancestor-or-self::}m && (14)\\
p\texttt{/self::}n\texttt{/ancestor::}m &\equiv p\texttt{[self::}n\texttt{]/ancestor::}m && (15)\\
p\texttt{/following-sibling::}n\texttt{/ancestor::}m &\equiv p\texttt{[following-sibling::}n\texttt{]/ancestor::}m && (16)\\
p\texttt{/following::}n\texttt{/ancestor::}m &\equiv p\texttt{/following::}m\texttt{[descendant::}n\texttt{]} && (17)\\
&\mid p\texttt{/ancestor-or-self::*}\\
&\quad\texttt{[following-sibling::*/descendant-or-self::}n\texttt{]}\\
&\quad\texttt{/ancestor::}m
\end{aligned}$$

$$\begin{aligned}
p\texttt{/descendant::}n\texttt{[ancestor::}m\texttt{]} &\equiv p\texttt{[ancestor::}m\texttt{]/descendant::}n && (18)\\
&\mid p\texttt{/descendant-or-self::}m\texttt{/descendant::}n\\
\texttt{/descendant::}n\texttt{[ancestor::}m\texttt{]} &\equiv \texttt{/descendant-or-self::}m\texttt{/descendant::}n && (18a)\\
p\texttt{/child::}n\texttt{[ancestor::}m\texttt{]} &\equiv p\texttt{[ancestor-or-self::}m\texttt{]/child::}n && (19)\\
p\texttt{/self::}n\texttt{[ancestor::}m\texttt{]} &\equiv p\texttt{[ancestor::}m\texttt{]/self::}n && (20)\\
p\texttt{/following-sibling::}n\texttt{[ancestor::}m\texttt{]} &\equiv p\texttt{[ancestor::}m\texttt{]/following-sibling::}n && (21)\\
p\texttt{/following::}n\texttt{[ancestor::}m\texttt{]} &\equiv p\texttt{/following::}m\texttt{/descendant::}n && (22)\\
&\mid p\texttt{/ancestor-or-self::*[ancestor::}m\texttt{]}\\
&\quad\texttt{/following-sibling::*/descendant-or-self::}n
\end{aligned}$$

3.2.3 Preceding-sibling

In the following proposition the `preceding-sibling` axis is treated. Note that the right-hand side of equivalences for `preceding-sibling` (and `preceding`) contains more union terms than the other equivalences, since there is no `-or-self` variant of these axes.

Proposition 3.4 (`preceding-sibling` axis). *Let m and n be node tests and p a location path. The following equivalences hold:*

$$\texttt{descendant::}n\texttt{/preceding-sibling::}m \equiv \texttt{descendant::}m\texttt{[following-sibling::}n\texttt{]} \tag{23}$$

$$\texttt{child::}n\texttt{/preceding-sibling::}m \equiv \texttt{child::}m\texttt{[following-sibling::}n\texttt{]} \tag{24}$$

$$p\texttt{/self::}n\texttt{/preceding-sibling::}m \equiv p\texttt{[self::}n\texttt{]/preceding-sibling::}m \tag{25}$$

$$\begin{aligned} p\texttt{/following-sibling::}n\texttt{/preceding-sibling::}m \equiv\ & p\texttt{[self::}m\texttt{/following-sibling::}n\texttt{]} \\ \mid\ & p\texttt{[following-sibling::}n\texttt{]/preceding-sibling::}m \\ \mid\ & p\texttt{/following-sibling::}m\texttt{[following-sibling::}n\texttt{]} \end{aligned} \tag{26}$$

$$\begin{aligned} p\texttt{/following::}n\texttt{/preceding-sibling::}m \equiv\ & p\texttt{/following::}m\texttt{[following-sibling::}n\texttt{]} \\ \mid\ & p\texttt{/ancestor-or-self::*[following-sibling::}n\texttt{]} \\ & \texttt{/preceding-sibling::}m \\ \mid\ & p\texttt{/ancestor-or-self::}m\texttt{[following-sibling::}n\texttt{]} \end{aligned} \tag{27}$$

$$\texttt{descendant::}n\texttt{[preceding-sibling::}m\texttt{]} \equiv \texttt{descendant::}m\texttt{/following-sibling::}n \tag{28}$$

$$\texttt{child::}n\texttt{[preceding-sibling::}m\texttt{]} \equiv \texttt{child::}m\texttt{/following-sibling::}n \tag{29}$$

$$p\texttt{/self::}n\texttt{[preceding-sibling::}m\texttt{]} \equiv p\texttt{[self::}n\texttt{]/following-sibling::}m \tag{30}$$

$$\begin{aligned} p\texttt{/following-sibling::}n\texttt{[preceding-sibling::}m\texttt{]} \equiv\ & p\texttt{[self::}m\texttt{]/following-sibling::}n \\ \mid\ & p\texttt{/following-sibling::}m\texttt{/following-sibling::}n \\ \mid\ & p\texttt{[preceding-sibling::}m\texttt{]/following-sibling::}n \end{aligned} \tag{31}$$

$$\begin{aligned} p\texttt{/following::}n\texttt{[preceding-sibling::}m\texttt{]} \equiv\ & p\texttt{/following::}m\texttt{/following-sibling::}n \\ \mid\ & p\texttt{/ancestor-or-self::*[preceding-sibling::}m\texttt{]} \\ & \texttt{/following-sibling::}n \\ \mid\ & p\texttt{/ancestor-or-self::/following-sibling::}n \end{aligned} \tag{32}$$

3.2.4 Preceding

The following proposition describes the interaction of `preceding` with other axes.

Proposition 3.5 (`preceding` axis). *Let m and n be node tests and p a location path.*

$$\begin{aligned} p\texttt{/descendant::}n\texttt{/preceding::}m \equiv\ & p\texttt{[descendant::}n\texttt{]/preceding::}m \\ \mid\ & p\texttt{/child::*} \\ & \texttt{[following-sibling::*/descendant-or-self::}n\texttt{]} \\ & \texttt{/descendant-or-self::}m \end{aligned} \tag{33}$$

$$\texttt{/descendant::}n\texttt{/preceding::}m \equiv \texttt{/descendant::}m\texttt{[following::}n\texttt{]} \tag{33a}$$

$$\begin{aligned} p\texttt{/child::}n\texttt{/preceding::}m \equiv\ & p\texttt{[child::}n\texttt{]/preceding::}m \\ \mid\ & p\texttt{/child::*[following-sibling::}n\texttt{]} \\ & \texttt{/descendant-or-self::}m \end{aligned} \tag{34}$$

$$p\texttt{/self::}n\texttt{/preceding::}m \equiv p\texttt{[self::}n\texttt{]/preceding::}m \tag{35}$$

$$\begin{aligned} p\texttt{/following-sibling::}n\texttt{/preceding::}m \equiv\ & p\texttt{[following-sibling::}n\texttt{]/preceding::}m \\ \mid\ & p\texttt{/following-sibling::*[following-sibling::}n\texttt{]} \\ & \texttt{/descendant-or-self::}m \\ \mid\ & p\texttt{[following-sibling::}n\texttt{]/descendant-or-self::}m \end{aligned} \tag{36}$$

$$\begin{aligned} p\texttt{/following::}n\texttt{/preceding::}m \equiv\ & p\texttt{[following::}n\texttt{]/preceding::}m \\ \mid\ & p\texttt{/following::}m\texttt{[following::}n\texttt{]} \\ \mid\ & p\texttt{[following::}n\texttt{]/descendant-or-self::}m \end{aligned} \tag{37}$$

$$p/\texttt{descendant::}n\texttt{[preceding::}m\texttt{]} \equiv p\texttt{[preceding::}m\texttt{]/descendant::}n \tag{38}$$
$$\mid p\texttt{/child::*[descendant-or-self::}m\texttt{]}$$
$$\texttt{/following-sibling::*/descendant-or-self::}n$$
$$\texttt{/descendant::}n\texttt{[preceding::}m\texttt{]} \equiv \texttt{/descendant::}m\texttt{/following::}n \tag{38a}$$
$$p\texttt{/child::}n\texttt{[preceding::}m\texttt{]} \equiv p\texttt{[preceding::}m\texttt{]/child::}n \tag{39}$$
$$\mid p\texttt{/child::*[descendant-or-self::}m\texttt{]}$$
$$\texttt{/following-sibling::}n$$
$$p\texttt{/self::}n\texttt{[preceding::}m\texttt{]} \equiv p\texttt{[preceding::}m\texttt{]/self::}n \tag{40}$$
$$p\texttt{/following-sibling::}n\texttt{[preceding::}m\texttt{]} \equiv p\texttt{[preceding::}m\texttt{]/following-sibling::}n \tag{41}$$
$$\mid p\texttt{/following-sibling::*[descendant-or-self::}m\texttt{]}$$
$$\texttt{/following-sibling::}n$$
$$\mid p\texttt{[descendant-or-self::}m\texttt{]/following-sibling::}n$$
$$p\texttt{/following::}n\texttt{[preceding::}m\texttt{]} \equiv p\texttt{[preceding::}m\texttt{]/following::}n \tag{42}$$
$$\mid p\texttt{/following::}m\texttt{/following::}n$$
$$\mid p\texttt{[descendant-or-self::}m\texttt{]/following::}n$$

Example 3.3. Consider the location path

```
/descendant::price/preceding::name
```

of Example (3.1). With Rule 33a it can be rewritten to

```
/descendant::name[following::price].
```

This result is more compact and closer to the original than the result of Example (3.1) using Equivalence (2a).

4 Location Path Rewriting

Each Equivalence (i) $p_1 \equiv p_2$ of Section 3 gives rise to a rewriting rule: A path matching with the left-hand side p_1 can be rewritten into a path corresponding to the right-hand side p_2. In the following, Rule (i) denotes the rewriting rule $p_1 \rightarrow p_2$ induced by Equivalence (i) $p_1 \equiv p_2$.

The equivalences of Lemma 3.1 and Lemma 3.2 induce rewriting rules, denoted Rules (3.1.1) to (3.1.8) and (3.2).

The equivalences of Section 3 are splitted in two sets of rules for use in a rewriting algorithm:

1. RuleSet_1, containing the general Rules (1), (2), (2a) and (3.2).
2. RuleSet_2, containing the specific Rules (3) to (42) and (3.2).

A rule can be applied to a location path in the following manner:

Definition 4.1 (Rule application). Let p be a non-disjunctive location path, and let $p_l \rightarrow p_r$ be a rule either from RuleSet_1 or RuleSet_2. If p is of the form p_l/p', then let q denote the path p_r/p'. If p_l is a relative path and if p is of the form $p_1/p_l/p_2$, then let q denote the path $p_1/p_r/p_2$. In both cases q is called the *result* of the application of rule $p_l \rightarrow p_r$ to p.

An algorithm, called "rare" (sketched in Figure 2) for computing a reverse-axis-free path equivalent to an absolute path is considered below. The input for the algorithm is restricted to paths without qualifiers containing so-called "RR joins":

Definition 4.2 (RR join). An RR join is an expression of the form $p_1\ \theta\ p_2$ where $\theta \in \{\texttt{==}, \texttt{=}\}$, and both p_1 and p_2 are **R**elative paths such that at least one of them contains a **R**everse step.

For the consideration of termination and correctness of the algorithm, some important properties of the application of the rewriting rules to a location path are required:

Lemma 4.1 (Properties of rule application). *Let p be an absolute location path with no qualifier containing RR joins.*

1. *If p contains a reverse step, then a rule from RuleSet$_1$ and a rule from RuleSet$_2$ is applicable to p. Possibly, Rules (3.1.1) to (3.1.8) have to be applied first.*
2. *The result of a rule application to the first reverse step in p is an absolute path with no qualifiers containing RR joins.*
3. *If q is the result of a rule application to p, then $p \equiv q$.*

Proof. **(1):** Let L be the first reverse location step.

First consider RuleSet$_1$: If L occurs outside a qualifier, Rules (2), (2a) or (3.2) can be applied, since p is an absolute location path. If L occurs as the first location step inside a qualifier Rule (1) can be applied. If L appears at any other position inside a qualifier, Rule (3.1.5) can be applied in order to construct a qualifier with L as first location step. Rule (1) can be applied now.

RuleSet$_2$ provides rules for interaction between each reverse step and an arbitrary forward step p, so there is always a rule, that can be applied to the first reverse step in p.
(2) Only Rules (1), (2), and (2a) introduce a binary relation (namely `==`), if they are applied to a location path. But always one of the two paths related by `==` is absolute. Hence, in any case the result of the rule application contains no RR join. Furthermore, since p is an absolute path, the result of applying a rule to p is also an absolute path.
(3) This holds due to Lemma 3.1.1−4. □

"rare" Algorithm. The "rare" Algorithm, outlined in Figure 2, can be used for RuleSet$_1$ as well as for RuleSet$_2$. The algorithm takes as input a location path which is absolute, since some rules from RuleSet$_1$ and RuleSet$_2$ are applicable to absolute location paths only.

Theorem 4.1 (Removal of reverse location steps using RuleSet$_1$). *Let p be an absolute path with no qualifier in which RR joins occur. There exists an absolute path p' with no reverse steps such that $p \equiv p'$. Using "rare" and RuleSet$_1$, this path p' has a length and can be computed in a time linear in the length of p.*

Proof. A path equivalent to p is constructed as sketched in Figure 2. All reverse location steps are rewritten, one by one. Lemma 4.1 guarantees that a rule of RuleSet$_1$ can be applied to any path containing a reverse location step. The resulting path p' contains no reverse location steps and is equivalent to p.

The location path p' is of linear size and constructed in linear time, since each rule application removes one reverse step, adds at most two forward location steps and no reverse ones. □

Theorem 4.2 (Removal of reverse location steps using RuleSet$_2$). *Let p be an absolute path with no qualifiers in which RR joins occur. There exists an absolute path p' with no reverse steps such that $p \equiv p'$. Using "rare" and RuleSet$_2$, this path p' has a length and can be computed in a time exponential in the length of p.*

Proof. An application of a rule from RuleSet$_2$ can have three different result types:

1. removes completely a reverse step (e.g. Rules (3.2) or (3));
2. pushes the reverse step from right to left in the path (e.g. Rule (5));

Figure 2 Algorithm ***rare*** (**r**everse **a**xis **re**moval)

Let ξ = $\mathrm{RuleSet}_1$ or $\mathrm{RuleSet}_2$.

Auxiliary functions:

match(p): returns the result of a rule application from ξ to the first reverse location step in p.

apply-lemmas(p): returns p if Rules (3.1.1-8) are not applicable to p. Otherwise, it returns the result of the repeated application of Rules (3.1.1-8) to p.

union-flattening(p): returns a path equivalent to p with unions at top level only.

***rare*(p)**
Input: p {absolute location path without qualifiers containing RR joins}.

$p \leftarrow$ *apply-lemmas*(p).
$p \leftarrow$ *union-flattening*(p) $= U_1 \mid \ldots \mid U_n$ $(n \geq 1)$.
$S \leftarrow$ empty stack.
for $i \leftarrow 1$ to n **do**
  push(U_i, S).
end for
$p' \leftarrow \perp$. {initialization}
while *not*(*empty*(S)) **do**
  $U \leftarrow$ *pop*(S).
  while U contains a reverse step **do**
    $U \leftarrow$ *match*(U).
    $U \leftarrow$ *apply-lemmas*(U).
    $U \leftarrow$ *union-flattening*(U) $= V_1 \mid \ldots \mid V_n$ $(n \geq 1)$.
    for $i \leftarrow 2$ to n **do**
      push(V_i, S).
    end for
    $U \leftarrow V_1$.
  end while
  $p' \leftarrow p' \mid U$.
end while
Output: p' {location path without reverse axes equivalent to p}.

3. for the interaction between a `following` and a reverse step L_r, i.e. $\texttt{following::}n/L_r$ or $\texttt{following::}n\texttt{[}L_r\texttt{]}$, a union of several other paths is obtained (e.g. Rule (7)); the resulting union terms have reverse steps at positions less than or equal in the original path and they do not contain anymore the interaction between the initial `following` step and a reverse step.

Since the path has a finite length, the procedure of pushing reverse steps leftwise terminates. Also, the number of interactions between `following` and reverse steps is finite. Hence, the algorithm terminates.

Each rule application having the first result type removes a reverse step and does not change the number of union terms. Hence, in the best case, i.e. using only rule applications with the first result type, the algorithm has a linear time complexity in the length of the input path p.

The last two result types are significant for the worst-case complexity of the algorithm, since each rule application can produce intermediate rewritten paths with more than one union term (up to three union terms). Hence, each rule application can increase the order of the input for the next rule application, yielding to an exponential time complexity in the length of the input path p. □

Example runs of the algorithm for both set of rules are presented in Figure 3 and Figure 4.

Comparison. Both RuleSet_1 and RuleSet_2 have advantages and it is an open issue which one is preferable. The path rewriting using RuleSet_2 has in the worst case an exponential time complexity and output size in the length of the input location path. As location paths are in practice small (less than ten steps), the exponential worst-case complexity of RuleSet_2 does not necessarily generate longer paths than RuleSet_1. In addition, since they do not contain joins, the location paths generated using RuleSet_2 are simpler (as can be seen in the examples), hence more convenient to evaluate, than those generated using RuleSet_1, which contain the same number of joins as there are reverse steps in the input location path. It remains to decide in practical tests, up to which size of an input path RuleSet_2 generates simpler paths than RuleSet_1.

Rewriting location paths using variables. There are two classes of location paths not covered by the rules given so far: relative location paths and location paths with RR joins (cf. Definition 4.2), e.g. $p\texttt{[self::*} = \texttt{preceding::*]}$. Any attempt to remove the reverse location steps in these cases results in losing the context given by p.

In the full version [17] of this paper an approach is proposed to solve this problem by remembering the context in a variable. It is based on a `for` construct for variable binding, as provided by XPath 2.0, XQuery, and XSLT. Using this approach *every* location path can be rewritten to an equivalent reverse-axis-free one.

5 Related Work

Several methods have been proposed for rewriting XPath expressions taking integrity constraints or schemas into account [6, 26], and the equivalence and containment problems for XPath expressions have been investigated [9, 27]. Furthermore, a growing interest in query optimization for XML databases, including optimization of XPath expressions, recently emerged. To the best of our knowledge, however, no other approach has been proposed for removing reverse steps from XPath expressions relying upon XPath symmetry. Note that using equivalence preserving rewriting rules for removing reverse steps from XPath expressions, as it is proposed in the present paper, is not closely related to the general equivalence problem for XPath expressions.

In [5] redundancies in XPath expressions based on a "model-oriented" approach are investigated. Such an approach relies on an abstract model of XPath that views XPath expressions as

Figure 3 Example run of ***rare*** algorithm with RuleSet_1

Consider the example of Figure 1 and a query asking for all `title`s that appear before a `name` and are inside `journal`s. This query can be expressed as the following location path:

`/descendant::name/preceding::title[ancestor::journal]`

Note that p is an absolute path with no qualifier containing RR-joins.

Step 1: $p \leftarrow \textit{apply-lemmas}(p) = p$.
Step 2: $U_1 \leftarrow$ `/descendant::name/preceding::title[ancestor::journal]`.
Step 3: $\textit{push}(U_1, S)$.
Step 4: $p' \leftarrow \perp$.
Step 5: $U \leftarrow \textit{pop}(S)$.
Step 6: U contains a reverse step (`preceding::title`).
Step 7: $U \leftarrow \textit{match}(U) =$ `/descendant::title[ancestor::journal]` `[following::name == /descendant::name]`. {Rule (2)}
Step 8: $U \leftarrow \textit{apply-lemmas}(U) = U$.
Step 9: U contains a reverse step (`ancestor::journal`).
Step 10: $U \leftarrow \textit{match}(U) =$ `/descendant::title` `[/descendant::journal/descendant::node() == self::node()]` `[following::name == /descendant::name]`. {Rule (1)}
Step 11: $U \leftarrow \textit{apply-lemmas}(U) = U$.
Step 12: U does not contain reverse steps.
Step 13: $p' \leftarrow U$.
Step 14: S is empty.

Output: $p =$ `/descendant::title` `[/descendant::journal/descendant::node() == self::node()]` `[following::name == /descendant::name]`.

Figure 4 Example run of ***rare*** algorithm with RuleSet_2

Consider the example query in Figure 3. For conciseness the union terms in Step 8 are not pushed to the stack.

Step 1: $p \leftarrow \textit{apply-lemmas}(p) = p$.
Step 2: $U_1 \leftarrow$ `/descendant::name/preceding::title[ancestor::journal]`.
Step 3: $\textit{push}(U_1, S)$.
Step 4: $p' \leftarrow \perp$.
Step 5: $U \leftarrow \textit{pop}(S)$.
Step 6: U contains a reverse step (`preceding::title`).
Step 7: $U \leftarrow \textit{match}(U) =$ `/descendant::title[ancestor::journal][following::name]` {Rule (33a)}
Step 8: U contains a reverse step (`ancestor::journal`).
Step 9: $U \leftarrow \textit{match}(U) =$ `/descendant-or-self::journal/descendant::title[following::name]`. {Rule (18a)}
Step 10: $U \leftarrow \textit{apply-lemmas}(U) = U$.
Step 11: U does not contain reverse steps.
Step 12: $p' \leftarrow U$.
Step 13: S is empty.

Output: $p' =$ `/descendant-or-self::journal/descendant::title[following::name]`.

tree patterns. [5] shows that redundant branches of a tree pattern can be eliminated in polynomial time. Tree patterns are more abstract than XPath expressions in a way which is relevant to the work described in the present paper: A same tree pattern represents multiple equivalent XPath expressions. In particular, the symmetries in XPath exploited in the present paper are absent from tree patterns. Tree patterns do not consider the document order and therefore the concept of forward and reverse steps. In some sense the present work shows in which cases this simplified view upon an XPath expression can be justified.

Stream-based query processing has gained considerable interest in the past few years, e.g. due to its application in data integration [10, 14] and in publish-subscribe architectures [4, 7]. They all consider a navigational approach (XML-QL or XPath) consisting of a restricted subset of forward axes from XPath. This fact contrasts with the present work, which enables the use of the unrestricted set of XPath axes in a stream-based context.

6 Conclusion

The main result of this paper consists in two rule sets, RuleSet_1 and RuleSet_2, used in an algorithm for transforming XPath 1.0 expressions containing reverse axes into reverse-axis-free equivalents. Both RuleSet_1 and RuleSet_2 have advantages and it is an open issue which one is preferable. The location paths generated using RuleSet_2 do not contain joins, in contrast with those generated using RuleSet_1, which contain the same number of joins as there are reverse steps in the input location path. However, the path rewriting using RuleSet_2 has an exponential complexity in the length of the input location path, in contrast with rewriting using RuleSet_1 which has only a linear complexity.

Closely related to the comparison of RuleSet_1 and RuleSet_2 we plan to investigate the notion of "minimality" or "simplicity" of XPath expressions. We are focusing on defining a notion of a minimal XPath expression that can be evaluated more efficiently in a stream-based context than its equivalents. A notion of minimality will allow for well-founded optimization techniques for XPath expressions.

We believe that the equivalences proposed in this paper are an important step towards an efficient progressive evaluation of full XPath. In particular, we show that it is not necessary to restrict the use of axes in XPath for progressive processing, as suggested in [8]. Based on the results of this paper we are designing and implementing a progressive XPath processor, that supports unrestricted XPath [12].

References

[1] *Astronomical Data Center*, Home page, `http://adc.gsfc.nasa.gov/`.

[2] *Cocoon 2.0: XML publishing framework*, `http://xml.apache.org/cocoon/index.html`.

[3] *Xalan-Java Version 2.2*, Apache Project, `http://xml.apache.org/xalan-j/index.html`.

[4] M. Altinel and M. Franklin, *Efficient Filtering of XML Documents for Selective Dissemination of Information*, Proc. of 26th Conference on Very Large Databases (VLDB), 2000.

[5] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava, *Minimization of tree pattern queries*, SIGMOD, 2001.

[6] K. Boehm, K. Gayer, T. Oezsu, and K. Aberer, *Query optimization for structured documents based on knowledge on the document type definition*, Proc. of the Advances in Digital Libraries Conference, 1998.

[7] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi, *Efficient Filtering of XML Documents with XPath Expressions*, Proc. of International Conference on Data Engineering (ICDE), 2002.

[8] A. Desai, *Introduction to Sequential XPath*, Proc. of IDEAlliance XML Conference, 2001, http://www.idealliance.org/papers/xml2001/papers/html/05-01-01.html.

[9] A. Deutsch and V. Tannen, *Containment for classes of XPath expressions under integrity constraints*, Knowledge Representation meets Databases (KRDB), 2001.

[10] T. J. Green, M. Onizuka, and D. Suciu, *Processing XML Streams with Deterministic Automata and Stream Indexes*, Tech. report, University of Washington, 2001.

[11] N. Ide, P. Bonhomme, and L. Romary, *XCES: An XML-based standard for linguistic corpora*, Proc. of the Second Annual Conference on Language Resources and Evaluation, 2000.

[12] T. Kiesling, *Towards a streamed XPath evaluation*, 2002, http://www.pms.informatik.uni-muenchen.de/lehre/projekt-diplom-arbeit/streamedxpath.html.

[13] P. Kroeger, *Modeling of Biological Data*, Tech. report, University of Munich, 2001.

[14] A. Levy, Z. Ives, and D. Weld, *Efficient Evaluation of Regular Path Expressions on Streaming XML Data*, Tech. report, University of Washington, 2000.

[15] S. McGrath, *XPipe*, http://xpipe.sourceforge.net/.

[16] D. Megginson, *SAX: The Simple API for XML*, http://www.saxproject.org/.

[17] D. Olteanu, H. Meuss, T. Furche, and F. Bry, *XPath: Looking Forward*, Tech. Report PMS-FB-2001-16, University of Munich, 2001, http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2001-16.pdf.

[18] W3C, *XML Path Language (XPath) Version 1.0*, W3C Recommendation, 1999, http://www.w3.org/TR/xpath.

[19] W3C, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, 1999.

[20] W3C, *Document Object Model (DOM) Level 2 Core Specification*, W3C Recommendation, 2000.

[21] W3C, *XQuery 1.0: An XML query language*, W3C Working Draft, 2001, http://www.w3.org/TR/xquery/.

[22] W3C, *XQuery 1.0 and XPath 2.0 data model*, W3C Working Draft, 2001, http://www.w3.org/TR/query-datamodel/.

[23] W3C, *XSL Transformations (XSLT) Version 2.0*, W3C Working Draft, 2001, http://www.w3.org/TR/xslt20.

[24] P. Wadler, *A formal semantics of patterns in XSLT*, Proc. of Conference on Markup Technologies, 1999.

[25] P. Wadler, *Two semantics of XPath*, Tech. report, 2000.

[26] P. T. Wood, *Optimising web queries using document type definitions*, 2nd ACM Workshop on Web Information and Data Management (WIDM'99), 1999.

[27] P. T. Wood, *On the equivalence of XML patterns*, Proc. 6th Int. Conf. on Rules and Objects in Databases (DOOD), 2000.