Avoiding Unnecessary Ordering Operations in XPath

Jan Hidders

University of Antwerp
Dept. of Mathematics and Computer Science
Middelheimlaan 1, 2020 Antwerpen, Belgium
jan.hidders@ua.ac.be
tel: +32 3 218.08.73
fax: +32 3 218 07.77

Philippe Michiels*

University of Antwerp
Dept. of Mathematics and Computer Science
Middelheimlaan 1, 2020 Antwerpen, Belgium
philippe.michiels@ua.ac.be
tel: +32 3 218.08.80
fax: +32 3 218 07.77

Abstract

We present a sound and complete rule set for determining whether sorting and duplicate removal operations in the query plan of XPath expressions are unnecessary. Additionally we define a deterministic finite automaton that illustrates how these rules can be translated into an efficient algorithm. This work is an important first step in the understanding and tackling of XPath/XQuery optimization problems that are related to ordering and duplicate removal.

1 Introduction

The XQuery Formal Semantics [5] provide a full description of both XPath's [2] and XQuery's [3] semantics and an extensive set of rules for the translation of both languages into the XQuery Core language. The semantics of XPath [11] require that the result of an XPath expression (with exception of the sequence operator) is sorted by document order and duplicate-free. In addition, some XPath expressions — for instance, those that contain aggregate functions or element indexes — also require that their input is duplicate-free and sorted. As a consequence many of the implementations that are faithful to the XPath semantics, such as Galax [6], insert an explicit operation for sorting and duplicate elimination after each step. These operations often create bottlenecks in the evaluation of certain XPath expressions on large documents. Therefore many other implementations omit these operations and sacrifice correctness for the sake of efficiency. In many cases however, these time consuming operations are not necessary because (1) after certain steps the result will always be sorted and/or duplicate-free or (2) the context in which this XPath expression occurs does not depend on the ordering or uniqueness of the nodes in the path expression's result.

The main contributions of this work are:

- A sound and complete set of inference rules that deduce whether an XPath expression that is evaluated by a straightforward implementation, that omits all sorting and duplicate elimination operations, always results in a list of unique nodes that is in document order.
- The illustration of how these rules interact and how they can be used for the definition of an efficient algorithm realised by deterministic automata.

To understand why finding such a rule set is not trivial, consider the following two examples. The relative path expression `ancestor::*/follsibl::*/ parent::*` when evaluated for a single, always produces an ordered result. However, its subexpression `ancestor::*/follsibl::*` clearly

*Contact Author

does not have that property. It is quite surprising to see that intermediate results are unordered where the final result is ordered.

One might think that the above only occurs after following certain axes. But this is not the case. Take, for instance, the path `/child::*/parent::*/foll-sibl::*/parent::*`. Once again, this result of the expression always is sorted (which we will explain later). But the subexpression `/child::*/parent::*/foll-sibl::*` sometimes produces a result out of document order.

The remainder of the paper is organized as follows. After defining some essential concepts in Section 2, Section 3 discusses a set of properties that we need for categorizing XPath expressions and for deducing the two essential properties: order and duplicate-freeness. These rules are defined in Section 4. In Section 5 we present deterministic automata that show the interactions between the rules and illustrate how our approach can be translated into an efficient algorithm. In Section 6, we discuss how our work can be applied to improve the performance of the Galax XQuery engine. Section 7 is about related work and we conclude in Section 8.

2 XPath

We begin with a (simplified) formalization of an XML document.

Definition 2.1 (XML Document). An *XML document* is a tuple $D = (N, \lhd, r, \lambda, \prec)$ such that $(N, \lhd)$ is a directed graph that is a tree with root r and $\lhd$ giving the parent-child relationship, $\lambda : N \to T$ is a labeling of the nodes and $\prec$ is a strict total order over on N that defines the document order and orders the nodes of the tree in preorder. The relation $\lhd^+$ denotes the transitive closure of $\lhd$.

Note that we do not consider processing instructions, namespaces, text nodes or attributes here. Next, we define the syntax of the XPath fragment that we will consider.

Definition 2.2 (XPath Expression). The language of *XPath expressions*, denoted as $\mathcal{P}$, is defined by the following abstract grammar

$$\begin{aligned} P &::= A \mid P/A \\ A &::= \uparrow \mid \downarrow \mid \uparrow^+ \mid \downarrow^+ \mid \uparrow^* \mid \downarrow^* \mid \\ &\quad\; \twoheadleftarrow \mid \twoheadrightarrow \mid \dot{\twoheadleftarrow} \mid \dot{\twoheadrightarrow} \end{aligned}$$

where A is the set of all axes defined as follows:

symbol	axis
$\downarrow$	`child`
$\downarrow^+$	`descendant`
$\downarrow^*$	`descendant-or-self`
$\twoheadrightarrow$	`following`
$\dot{\twoheadrightarrow}$	`following-sibling`
$\uparrow$	`parent`
$\uparrow^+$	`ancestor`
$\uparrow^*$	`ancestor-or-self`
$\twoheadleftarrow$	`preceding`
$\dot{\twoheadleftarrow}$	`preceding-sibling`

This notation is an extention of the one used in [1] and is primarily used for compactness reasons.

In XPath, step expressions are of the form `axis::nodetest[predicate]`. Our syntax ignores predicates and node tests. For instance, the path expression $\downarrow/\uparrow$ actually represents the XPath expression `child::*/parent::*`.

Also, the `self` axis is disregarded here, because it represents the identity function and as a consequence preserves all properties of the preceding XPath expression. The grammar does not include the production rule P ::= P/P. This implies that we do not take into account path expressions of the form $p_1/(p_2/p_3)$. However, our theory can be generalized to include such expressions but this would take us too far.

The semantics of an XPath expression p in a document D is denoted by the function

$$[\![p]\!]_D : N \to \mathcal{L}(N)$$

where N is the set of nodes in D and $\mathcal{L}(N)$ the set of all finite lists over N. When it is obvious which document the path expression queries, we will omit D.

In addition to these formal semantics of path expressions, we define a "sloppy" semantics that corresponds to an implementation that does not

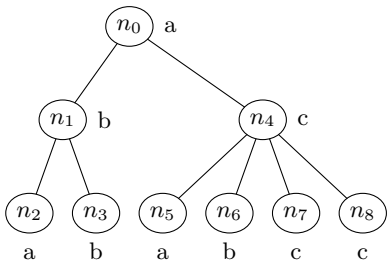

Figure 1: A simple XML document.

eliminate duplicates and does not sort by document order after each step in the path expression. The semantics are defined by giving for each path expression p its "sloppy" implementation $\alpha(p)$ in terms of the XQuery core algebra [5], that contains one free variable `$dot` that represents the context node. For this mapping, we assume that for each axis a in A there is a function $\alpha(a)$ that implements it and maps every node in the document to a sorted and duplicate-free list of nodes.

$$\alpha(p_1/a) \quad = \quad \texttt{for \$dot in } \alpha(p_1) \texttt{ return } \alpha(a)$$

The semantics of the implementation $\alpha(p)$ under a document D is written as a function

$$[\![\alpha(p)]\!]_D : N \to \mathcal{L}(N).$$

It is easy to see that the semantics of $\alpha(p)$ is equal to the formal semantics of p up to sorting and duplicate elimination.

3 Path Properties

In this section, we introduce some properties that XPath expressions can have. These properties will assist us in determining whether a path expression always produces a result that is in document order or free from duplicates. In the next section, we define a set of rules for deriving these properties for each expression $p \in \mathcal{P}$.

The two main properties we want to derive for these path expressions are

- ord - the order property, which says that for all documents D and nodes n in D the result of $[\![\alpha(p)]\!]_D(n)$ is in document order (possibly with duplicates);
- $nodup$ - the *no-duplicates* property that indicates that for all documents D and nodes n in D the result of $[\![\alpha(p)]\!]_D(n)$ contains no duplicates (but may not be in document order).

In order to derive these properties for all path expressions p in $\mathcal{P}$ we need an additional set of properties:

- gen - the generation property tells us whether for all documents D and nodes n in D all of the nodes in $[\![\alpha(p)]\!]_D(n)$ have the same distance to the root;

For instance, in Figure 1, the list [n_2, n_3, n_6] has the gen property, but the list [n_1, n_6, n_8] does not.

This property finds its use in that the notion of nodes being of the same generation is a crucial factor for deciding whether some expressions generate duplicate nodes. For instance, the axes $\downarrow^+$and $\downarrow^*$, in general produce duplicate nodes *unless* their inputs have the gen property.

- $max1$ - this property determines whether for all documents D and nodes n in D the number of nodes $\mid [\![\alpha(p)]\!]_D(n) \mid \leq 1$;

For instance, in Figure 1, the query $\uparrow$, executed from the context node n_1 results in the list [n_0], and clearly has the $max1$ property. However, the query $\downarrow$, executed from the same context node results in [n_2, n_3] and does not have the max_1 property. The $max1$ property is important because, in general, most of the axes evaluated from a list of nodes generate duplicates or produce unordered results. But if an axis is evaluated from one single node, their results will — by definition — be in order and duplicate free.

- $unrel$ - the *unrelated* property tells us whether for all documents D and nodes n in D there are no two nodes in $[\![\alpha(p/\uparrow)]\!]_D(n)$ that have an ancestor-descendant relationship; i.e.,

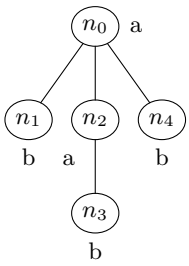

Figure 2: An example XML fragment that shows the relevance of the $unrel$ property.

$$\nexists n_1, n_2 : n_1 \neq n_2 \, and \, n_1 \in anc(n_2) \, and \, n_2 \in anc(n_1)$$

For instance, in Figure 1, the query $\dot{\twoheadrightarrow}$, executed from the context node n_5 results in the list [n_6, n_7, n_8], which has the $unrel$ property. However, the query $\uparrow^+$, executed from the same context node results in [n_0, n_4] and does not have the $unrel$ property. This property is much alike the gen property, but is less restrictive. In general, the $\downarrow$, $\downarrow^+$and $\downarrow^*$ axes can produce duplicates, but the following rule states that if their inputs are ordered and duplicate-free and the $unrel$ property holds for the input, the result will, again, be ordered.

$$\frac{p : ord_0 \qquad p : unrel \qquad p : nodup \qquad a \in \{\downarrow, \downarrow^+, \downarrow^*\}}{p/a : ord_0}$$

Take for instance, the example XML document fragment in figure 2 and consider the query $//a/b$. It is clear that the result of $//a$ is ordered and produces the list $[n_0, n_2]$. However, if we iterate over this list to retrieve the children, we obtain the list $[n_1, n_4, n_3]$, which is not ordered at all! This is because n_0 and n_2 are related. This example also demonstrates the relation between gen property and the $unrel$ property. Consider following the parent axis from the list of (unrelated) nodes $[n_1, n_3, n_4]$. Once again, the result is unordered.

- ord_m - this property tells us that for all documents D and nodes n in D, the nodes in $[\![\alpha(p/\uparrow^m)]\!]_D(n)$ are in document order. This means that if we follow the $\uparrow$ axis m times from $[\![\alpha(p)]\!]_D(n)$, that the result will be in document order again. It is obvious that ord is a special case of ord_m, where $m = 0$.

For instance, in Figure 1, the query $\downarrow/\dot{\twoheadrightarrow}$, executed from the context node n_3 results in the list [n_6, n_7, n_8, n_7, n_8, n_8], which is clearly not in document order. However, if we follow the $\uparrow$ axis once we obtain a list of six times n_3, which is in order. The query $\downarrow/\dot{\twoheadrightarrow}$ has the property ord_1. Once again, we see that, even though a subexpression can produce an unordered result, the entire expression could well be ordered. For instance, the expression $\uparrow^+/\dot{\twoheadrightarrow}$ does not have the ord property, but the result of the expression $\uparrow^+/\dot{\twoheadrightarrow}/\uparrow$ is always ordered.

- lin_m - the *linear* property indicates that for all documents D and nodes n in D, the nodes in $[\![\alpha(p/\uparrow^m)]\!]_D(n)$ are linear, i.e. for all two nodes $n_1, n_2 \in [\![\alpha(p/\uparrow^m)]\!]_D(n)$ it holds that $n_1 \lhd^+ n_2$, $n_2 \lhd^+ n_1$ or $n_1 = n_2$.

For instance, in Figure 1, the query $\uparrow^+$, executed from the context node n_2 results in the list [n_0, n_1], which is linear. The fact that a path expression has the lin property, for instance, tells us that if the $\dot{\twoheadrightarrow}$ axis is followed from that path, the $unrel$ property holds.

- sib_m - the *sibling* property tells us that for all documents D and nodes n in D, if a node is in $[\![\alpha(p/\uparrow^m)]\!]_D(n)$ then its left or right siblings or both will also appear in the result.

It is obvious that if we query the document from context node n_1 for the following siblings, that all siblings added after n_1 will also appear in the result. Thus, $\dot{\twoheadrightarrow}$ has the sib_0 property. This property is essential for proving that certain properties do not hold. It plays a crucial role in the completeness proof of Section 5.

4 Inference Rules

We have chosen for the use of inference rules because it easy to verify their correctness. We define

a set of rules $\mathcal{R}$ for the inference of the $nodup$ and ord properties. The set of rules is given in Figure 3. Not all rules are intuitive. We explain a few of them.

- The gen property is preserved by $\downarrow$, $\uparrow$, $\dot{\twoheadrightarrow}$ and $\dot{\twoheadleftarrow}$

$$\frac{p : gen \qquad a \in \{\downarrow, \uparrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : gen}$$

  This rule states that if a path expression p has the gen property i.e., all the nodes in the result of p have the same distance to the root, then if it is followed by one of the axes $\downarrow$, $\uparrow$, $\dot{\twoheadrightarrow}$, $\dot{\twoheadleftarrow}$, then the entire expression also has the gen property.

- The ord property is preserved by the $\uparrow$ axis if the gen property also holds

$$\frac{p : ord_0 \qquad p : gen}{p/\uparrow : ord_0}$$

  If all nodes in the input of the $\uparrow$ axis have the same distance to the root and if they are in document order, then the result may contain duplicate nodes. But these duplicates are the result of evaluating the $\uparrow$ axis from the children of the same node. Since these children are ordered, the duplicates will always occur in clusters and the result will be ordered. Surprisingly, the gen property is absolutely required and cannot be replaced by the less restrictive $unrel$ property. The above also implies that the removal of duplicates in this situation can be achieved very efficiently.

Theorem 4.1. *The rules in $\mathcal{R}$ are sound for the ord and $nodup$ properties; i.e., if we can derive with the rules in $\mathcal{R}$ in a finite number of steps that $p : ord$ ($p : nodup$) for all XML documents D and nodes $n \in D$, it holds that $[\![\alpha(p)]\!]_D(n)$ is in document order (contains no duplicate nodes).*

Proof. (sketch[1]) To prove the theorem, we can prove soundness individually for each rule in $\mathcal{R}$. □

5 Decision Procedure

The rules in $\mathcal{R}$ allow us to construct a deterministic automaton that decides whether or not the result of a path expression contains duplicates or is out of document order. To indicate that the algorithm can be easily implemented, we consider two separate automata: one for deriving the $nodup$ property (A_{nodup}) and one for deriving the ord_0 property (A_{ord}). Both automata accept expressions p that have the ord ($nodup$) property, in a time that is linear to the length of p; i.e., the number of step expressions in p.

5.1 The A_{ord} Automaton

This infinite automaton (see Figure 4) shows five accept states. Each state is labeled with the properties that hold in that state. The three-dot symbols at the right indicate that the automaton has an infinite number of subsequent states with transitions from and to it that are the same as those of the last state before the symbol. The states are labeled with the same properties unless that property has an index. In this case, the index ascends in the subsequent states.

Note that the prefix of a path that has the ord property does not necessarily have the ord property itself, however it is possible to return from an unordered state back into an ordered one.

Theorem 5.1. *A_{ord} is sound for the ord_0 property in $\mathcal{P}$; i.e., A_{ord} accepts only path expressions in $\mathcal{P}$ that have the ord property.*

Proof. For each transaction τ from state s_1 to state s_2, labeled with axis a, it holds that there is a set of rules in $\mathcal{R}$ that justifies it; i.e., for every property that holds in s_2 the rules in $\mathcal{R}$ derive this property for a. Soundness now follows from the soundness of $\mathcal{R}$. □

[1] For brevity, we will omit the proofs for the separate inference rules.

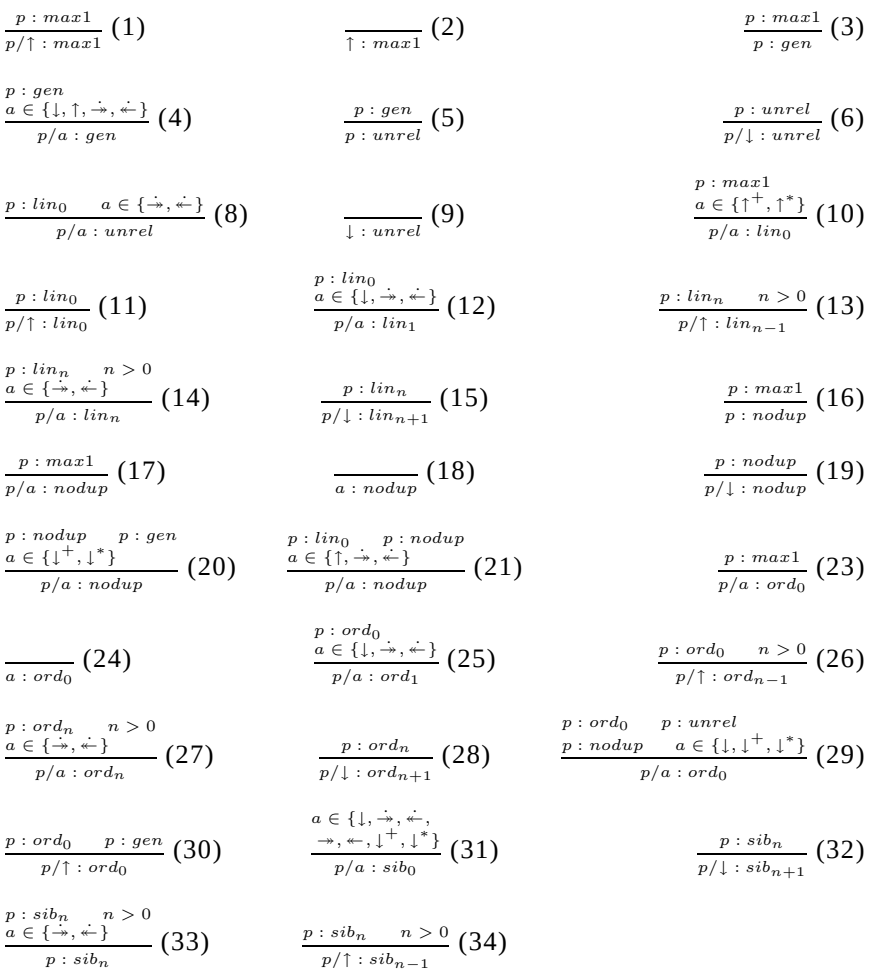

Figure 3: The rules of $\mathcal{R}$ for determining the $nodup$ and ord properties for expressions in $\mathcal{P}$.

Theorem 5.2. *A_{ord} is complete for the ord_0 property in $\mathcal{P}$; i.e., every path expression in $\mathcal{P}$ that has the ord property is accepted by A_{ord}.*

Proof. (sketch) We first extend the automaton in Figure 4 as shown in Figure 7:

1. Add a *sink* state, which indicates that all path expressions that lead to this state have definitively lost their ord property; i.e., they cannot be extended to a path expression that has the ord property;

2. For each state S in A_{ord} and for each axis a that does not have a transition from S, add a transition from S to the sink, labeled with a;

3. Label all states with the negation of the properties that do not hold in that state.

Next, we define an additional set of rules for the negative properties that justify the new transactions inside the new automaton. For each state, we determine the properties that do not hold and label the state accordingly. All non-accepting states are labeled with the $\neg ord$ property.

If we can prove these rules to be sound, then we know that the automaton rejects only those expressions that do *not* have the ord property. Since our automaton (by construction) now defines for every axis a transition from each state to another state, we can conclude that our original automaton accepts *all* path expressions that have the ord property. This implies that our set of principal

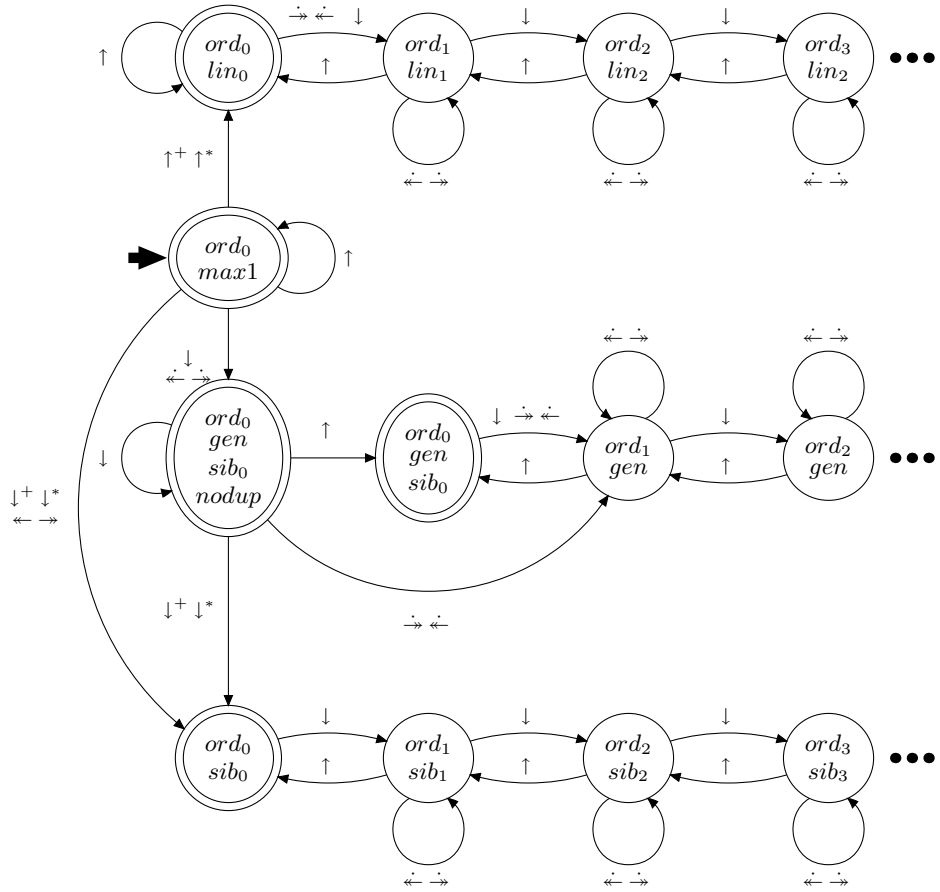

Figure 4: The A_{ord} automaton for the ord_0 property has an infinite number of states.

rules $\mathcal{R}$ is complete for the ord property. □

Figure 7 shows the result of extending our automaton. Note that in all accepting states, the ord_0 property holds and that in every other state, somehow the negation of this property holds. For instance ord_1 and $\neg ord_{\leq 0}$ together imply $\neg ord_0$. The new transitions and properties are justified by the rules given in Appendix B. If a path expressions brings the automaton into a sink state, we now it's result will be unordered, no matter what the remainder of the expression is.
For instance, if a path expression begins with $\downarrow^+/\uparrow$, then — no matter what step expressions follow — the entire expression will never regain the ord_0 property again.

5.2 The A_{nodup} Automaton

This finite automaton (see Figure 5) shows that, unlike the ord_0 property, once the $nodup$ property is lost in $\mathcal{P}$, it never recurs; i.e., if a path expression does not have the $nodup$ property, then neither will any of its prefixes.

Theorem 5.3. *A_{nodup} is sound for the $nodup$ property in $\mathcal{P}$; i.e., A_{nodup} accepts only path expressions in $\mathcal{P}$ that have the $nodup$ property.*

Proof. Analogous to proof of theorem 5.1. □

Theorem 5.4. *A_{nodup} is complete for the $nodup$ property in $\mathcal{P}$, i.e., every path expression in $\mathcal{P}$ that has the $nodup$ property is accepted by A_{nodup}.*

Proof. (sketch) Analogous to the proof of theorem 5.2 (see Figure 8). □

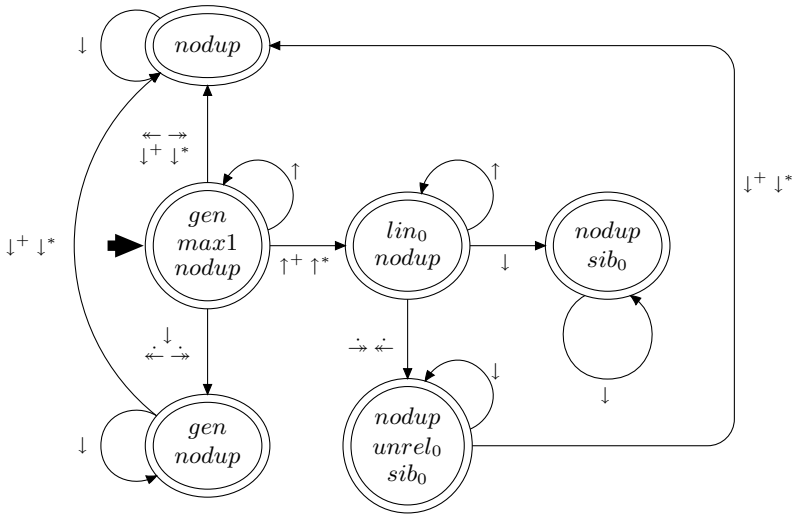

Figure 5: Unlike A_{ord}, the A_{nodup} automaton is finite.

6 Implementation in Galax

We will use the Galax XQuery engine for evaluating our approach. Galax is one of the first implementations of XQuery and one of a few that implements XQuery's static type system. Galax's architecture is based on the XQuery Formal Semantics, making it an ideal platform on which to implement and evaluate novel optimizations for XPath and XQuery.

In the previous sections, we have seen that ordering and duplicate elimination directly influence the evaluation efficiency of XPath. Indeed, unnecessary ordering and duplicate removal can cause a tremendous overhead during evaluation, especially when larger XML documents are involved. In the Galax [6] XQuery engine, for instance, this problem sometimes results in an unacceptable
evaluation performance.

Our approach impacts most XPath expressions and, since XQuery is based on XPath, XQuery expressions can also profit from it. We can generalize our approach, working on the core expressions where we apply the optimizations on the `for`-loops into which XPath expressions are mapped.

Using our approach to determine whether a path expression generates duplicate nodes or nodes that are out of document order, we can optimize almost any XQuery Core mapping by eliminating any obsolete `distinct-docorder` operations. The `distinct-docorder` operation is a meta-operator that is inserted into the core expression during normalization in order to assure correctness regarding document order and no duplicates [5].

Take, for instance, the path expression

```
//b/c/d/following-sibling::d/parent::*
```

which selects all elements `c` that have a parent `b` and that have more than one `d` child. This expression is mapped to the following, slightly simplified [4] core expression.

```
glx:distinct-docorder(
  for $glx:dot in $input
  return glx:distinct-docorder(
    for $glx:dot in descendant-or-self::node()
    return glx:distinct-docorder(
      for $glx:dot in child::b
      return glx:distinct-docorder(
        for $glx:dot in child::c
        return glx:distinct-docorder(
          for $glx:dot in child::d
          return glx:distinct-docorder(
            for $glx:dot in foll-sibl::d
            return parent::*
))))));
```

However, as our automaton shows, this expression results in an ordered sequence of nodes that possibly contains duplicates. Therefore, the expression is equivalent to the following one

```
distinct-nodes(
  for $glx:dot in $input
  return
    for $glx:dot in descendant-or-self::b
    return
        for $glx:dot in child::c
        return
          for $glx:dot in child::d
          return
            for $glx:dot in foll-sibl::d
            return parent::*
```

Note that no sorting options are required in the last query, whereas the original query required no less than six sorting operations. Additionally, since we know that the result of the query is in document order, we can remove duplicate nodes in linear time.

As the example shows, it may be useful to split the `distinct-docorder` operation into two separate instructions (one for sorting and one for eliminating duplicates from an ordered list) for cases where the result is ordered but contains duplicates.

There also seems to be an interesting interaction between our optimization technique and other schema-based optimizations. There are path expressions for which we cannot derive the ord or $nodup$ properties. Nevertheless, when they are rewritten to equivalent expressions based on schema information [7, 9], it can sometimes become possible to derive these properties.

For example, if we consider the path expression `//b/c`, then our algorithm shows that it returns nodes out of document order. But if we know from the schema of the source XML document that an element `b` cannot occur nested inside another element `b` and there is only one path to `b`, then we can substitute `//b` with the path to `b` using only `child` axes. In this way we avoid an ordering operation. This technique can be used for optimizing path expressions that have any axes in them that do not preserve the gen property.

7 Related Work

Galax is not the only implementation facing these problems. In an attempt to pipeline step expressions from an XPath expression, [8] proposes a technique that avoids the generation of duplicate nodes in the first place. This is done by translating XPath expressions into a sequence of algebraic operations such that no duplicate nodes are generated, which is very important because the elimination of duplicates is a pipeline breaker. One of the basics of this approach is the rewriting of XPath expressions into equivalent expressions that do not generate duplicates. These rewriting rules are inspired by [10] where the setup is to translate paths with reverse axis into equivalent ones, containing only forward axes.

8 Conclusion and Future Work

Our approach has focussed primarily on two properties of XPath expressions: order and duplicate-freeness. We have shown for our XPath fragment, that we can efficiently derive whether a query evaluated by the *sloppy* implementation α, returns a result in document order and free from duplicates. This knowledge can be used to remove unnecessary ordering or duplicate removal operations from the query plan or to rewrite certain expressions so that neither ordering nor duplicate removal are required, like the schema based optimizations we discussed in section 6.

We will implement our algorithm into the Galax XQuery engine, where unnecessary `distinct-docorder` operations sometimes cause unacceptable evaluation performance for queries on large documents. The optimizer will be extended with an algorithm that manipulates the abstract syntax tree of XQuery core expressions to remove unnecessary ordering and duplicate removal operations. We expect that our approach will be very helpful in improving the performance of query evaluation however, not all unnecessary ordering or duplicate removal operations are removed. One reason for this is that we fail to take into account possible ordering or duplicate removal operations on a part of a query that influence the entire query. A simple example can illustrate this.

For instance, the expression

$$\downarrow/\downarrow/\dot{\twoheadrightarrow}/\uparrow/\downarrow$$

is normalized into the following simplified core expression:

```
glx:distinct-docorder(
  let $glx:sequence := child::*
  return
    for $glx:dot in $glx:sequence
    return glx:distinct-docorder(
      let $glx:sequence := child::*
      return
        for $glx:dot in $glx:sequence
        return glx:distinct-docorder(
          let $glx:sequence := fol-sibl::*
          return
            for $glx:dot in $glx:sequence
            return glx:distinct-docorder(
              let $glx:sequence := parent::*
              return
               for $glx:dot in $glx:sequence
               return child::*)))))
```

which we represent simplified as ddo(↓/ddo(↓/ddo(⇾̇/ddo(↑/↓)))). We can be simplify this to ddo(↓/ddo(↓/ ddo(⇾̇/↑/↓))). The outermost ddo operation is useless because the one near it removes all duplicate nodes, however we cannot deduce this with our approach because we lack the notion of the ddo operation.
In a next step we will extend our approach to detect these unnecessary operations and enable us to further optimize XPath evaluation.

References

[1] M. Benedikt, W. Fan, and G. Kuper. Structural properties of XPath fragments. In *Proc. of ICDT'03, 2003.*, 2003.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (xpath) 2.0, w3c working draft 02 may 2003, 2003. http://www.w3.org/TR/xpath20.

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0 an xml query language, w3c working draft 02 may 2003, 2003. http://www.w3.org/TR/xquery.

[4] B. Choi, M. Fernández, and J. Siméon. The XQuery formal semantics: A foundation for implementation and optimization. Internal Working Document at AT&T and Lucent. http://www.cis.upenn.edu/~kkchoi/galax.pdf, 2002.

[5] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, w3c working draft 2 may 2003, 2002. http://www.w3.org/TR/query-semantics.

[6] M. Fernández and J. Siméon. *Galax, the XQuery implementation for discriminating hackers.* AT&T Bell Labs and Lucent Technologies, v0.3 edition, 2003. http://www-db-out.bell-labs.com/galax.

[7] M. Fernández and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the International Conference on Data Engineering*, pages 14–24, 1998.

[8] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. of the 3rd International Conference on Web Information Systems Engineering (WISE 2002)*, pages 215–224, Singapore, 2002.

[9] A. Kwong and M. Gertz. Schema-based optimization of XPath expressions. Technical report, Univ. of California, dept. of Computer Science, 2001.

[10] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[11] P. Wadler. Two semantics for XPath, 1999. http://www.cs.bell-labs.com/who/wadler/topics/xml.html.

A Rules for negative properties

Figure 6 shows the ules for the negative properties that are used for justifying the transitions in the extended automata (Figures 7 and 8). Proving the soundness for these rules is essential for proving completeness of the rules in $\mathcal{R}$.

$$\frac{a \in \{\downarrow^+, \downarrow^*, \twoheadrightarrow, \twoheadleftarrow\}}{p/a : \neg ord_{\geq 1}}\ \neg ord_{\geq} \qquad \frac{p : \neg ord_{\geq n}}{p/\downarrow : \neg ord_{\geq n+1}}\ \neg ord_{\geq} \qquad \frac{p : \neg ord_{\geq n} \quad n > 0}{p/\uparrow : \neg ord_{\geq n-1}}\ \neg ord_{\geq}$$

$$\frac{p : \neg ord_{\geq n} \quad n > 0 \quad a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : \neg ord_{\geq n}}\ \neg ord_{\geq} \qquad \frac{p : \neg max1 \quad a \in \{\uparrow^+, \uparrow^*, \twoheadrightarrow, \twoheadleftarrow\}}{p/a : \neg ord_{\geq 0}}\ \neg ord_{\geq} \qquad \frac{p : \neg ord_{\geq 0}}{p/\uparrow : \neg ord_{\geq 0}}\ \neg ord_{\geq}$$

$$\frac{p : \neg ord_{\geq 0}}{p/\downarrow : \neg ord_{\geq 1}}\ \neg ord_{\geq} \qquad \frac{p : \neg ord_{\geq 1} \quad p : \neg ord_{\leq n}}{p : \neg ord_{\geq 0}}\ \neg ord_{\geq} \qquad \frac{p : \neg ord_{\geq 0} \quad a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p : \neg ord_{\geq 0}}\ \neg ord_{\geq}$$

$$\frac{p : \neg nodup \quad a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}, \downarrow, \downarrow^+, \downarrow^*\}}{p/a : \neg ord_{\leq 0}}\ \neg ord_{\leq} \qquad \frac{p : sib_0 \quad a \in \{\dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\}}{p/a : \neg ord_{\leq 0}}\ \neg ord_{\leq} \qquad \frac{p : \neg ord_{\leq 0} \quad a \in \{\downarrow, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}, \downarrow^+, \downarrow^*\}}{p/a : \neg ord_{\leq 0}}\ \neg ord_{\leq}$$

$$\frac{p : \neg ord_{\geq 0}}{p : \neg ord_{\leq 0}}\ \neg ord_{\leq} \qquad \frac{p : \neg unrel_0 \quad a \in \{\downarrow, \uparrow^+, \uparrow^*, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\}}{p/a : \neg ord_{\leq 0}}\ \neg ord_{\leq} \qquad \frac{p : \neg ord_{\leq n}}{p/\downarrow : \neg ord_{\leq n+1}}\ \neg ord_{\leq}$$

$$\frac{p : \neg ord_{\leq n} \quad n > 0}{p/\uparrow : \neg ord_{\leq n-1}}\ \neg ord_{\leq} \qquad \frac{p : \neg ord_{\leq n} \quad n > 0 \quad a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : \neg ord_{\leq n}}\ \neg ord_{\leq} \qquad \frac{p : \neg ord_{\leq n}}{p : \neg ord_{\leq 0}}\ \neg ord_{\leq}$$

$$\frac{a \in \{\uparrow^+, \downarrow^+, \uparrow^*, \downarrow^*, \twoheadrightarrow, \twoheadleftarrow\}}{p/a : \neg gen}\ \neg gen \qquad \frac{p : \neg gen \quad a \in \{\downarrow, \uparrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : \neg gen}\ \neg gen \qquad \frac{a \in \{\downarrow, \downarrow^+, \downarrow^*, \uparrow^+, \uparrow^*, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}, \twoheadrightarrow, \twoheadleftarrow\}}{p/a : \neg max1}\ \neg max1$$

$$\frac{p : \neg max1}{p/\uparrow : \neg max1}\ \neg max1 \qquad \frac{p : \neg gen}{p/a : \neg max1}\ \neg max1 \qquad \frac{a \in \{\downarrow^+, \downarrow^*, \uparrow^+, \uparrow^*, \twoheadrightarrow, \twoheadleftarrow\}}{p/a : \neg unrel_0}\ \neg unrel_n$$

$$\frac{p : \neg unrel_0 \quad a \in \{\uparrow, \downarrow\}}{p/a : \neg unrel_0}\ \neg unrel_n \qquad \frac{p : \neg unrel_0 \quad a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : \neg unrel_1}\ \neg unrel_n \qquad \frac{p : \neg unrel_n \quad n > 0}{p/\uparrow : \neg unrel_{n-1}}\ \neg unrel_n$$

$$\frac{p : \neg unrel_n}{p/\downarrow : \neg unrel_{n+1}}\ \neg unrel_n \qquad \frac{p : \neg unrel_n \quad n > 0 \quad a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : \neg unrel_n}\ \neg unrel_n \qquad \frac{p : sib \quad a \in \{\uparrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : \neg nodup}\ \neg nodup$$

$$\frac{p : \neg nodup}{p/a : \neg nodup}\ \neg nodup \qquad \frac{p : \neg max1 \quad a \in \{\uparrow^+, \uparrow^*, \twoheadleftarrow, \twoheadrightarrow\}}{p/a : \neg nodup}\ \neg nodup \qquad \frac{p : \neg gen \quad p : \neg unrel_0 \quad a \in \{\downarrow^+, \downarrow^*\}}{p/a : \neg nodup}\ \neg nodup$$

$$\frac{}{p/\downarrow : \neg lin_0}\ \neg lin_n$$

Figure 6: The rules for the negative properties justify the transitions in the extended automata.

B Extended Automata

Figure 7 shows the A_{ord} automaton, extended according to the algorithm described in the proof of Theorem 5.2 in Section 5. Figure 8 shows the same extension for the A_{nodup} automaton.

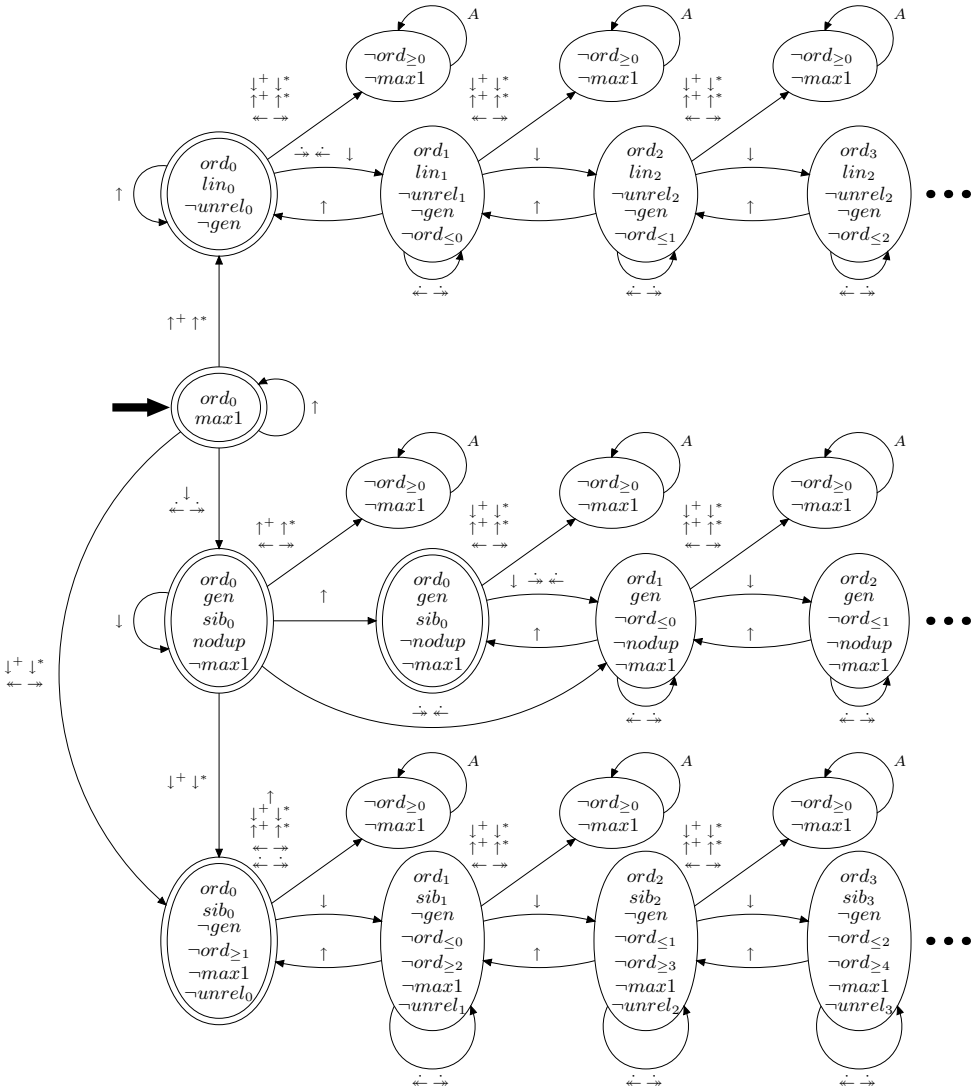

Figure 7: The extended version of the A_{ord} automaton.

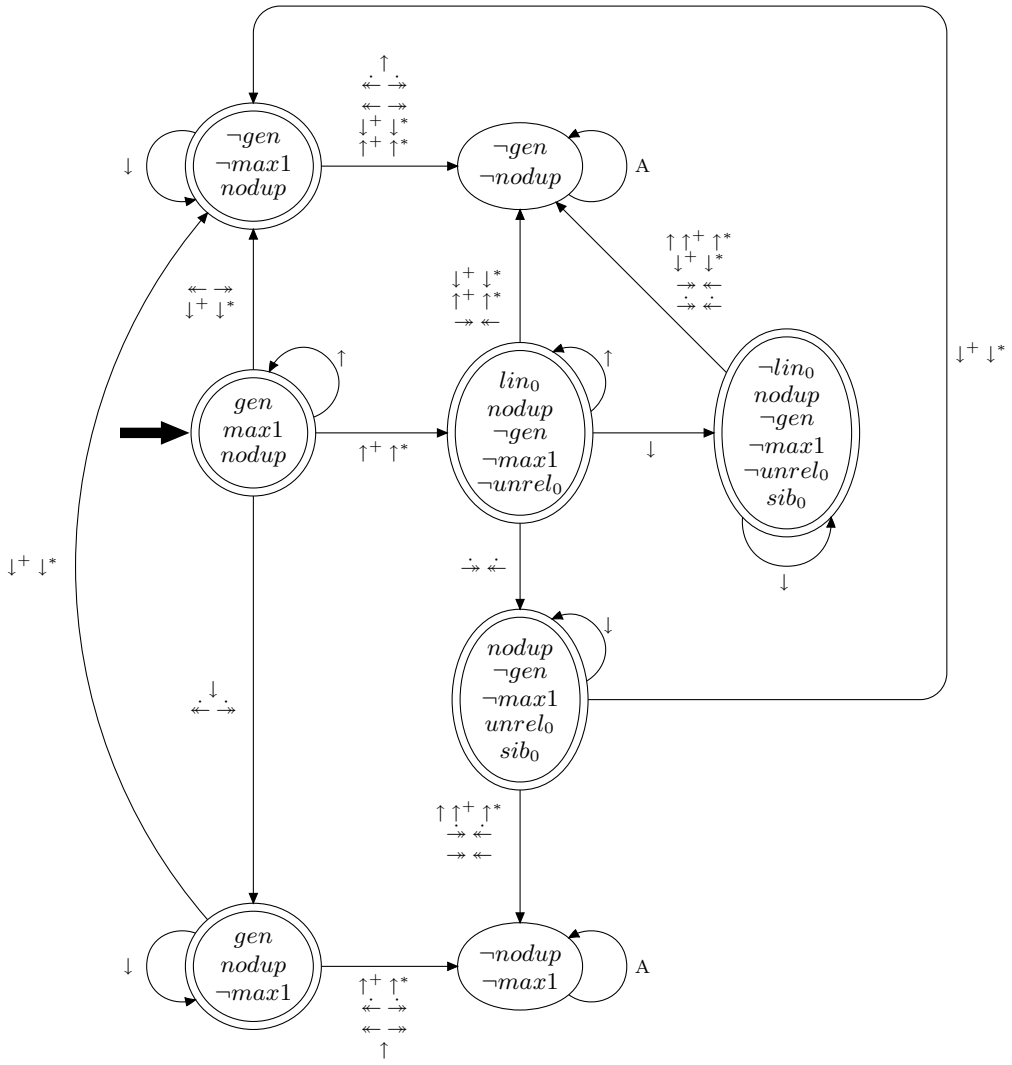

Figure 8: The extended A_{nodup} automaton.