Efficient XPath Axis Evaluation for DOM Data Structures

Jan Hidders Philippe Michiels
University of Antwerp
Dept. of Math. and Comp. Science
Middelheimlaan 1, BE-2020 Antwerp, Belgium,
{jan.hidders,philippe.michiels}@ua.ac.be

Abstract

In this article we propose algorithms for implementing the axes for element nodes in XPath given a DOM-like representation of the document. Each algorithm assumes an input list that is sorted in document order and duplicate-free and returns a sorted and duplicate-free list of the result of following a certain axis from the nodes in the input list. The time complexity of all presented algorithms is at most $O(l + m)$ where l is the size of the input list and m the size of the output list. This improves upon results in [4] where also algorithms with linear time complexity are presented, but these are linear in the size of the entire document whereas our algorithms are linear in the size of the intermediate results which are often much smaller.

1 Introduction

The XQuery Formal Semantics [3] requires that the result of an XPath path expression returns a list of document nodes that is sorted in document order and duplicate free. This can be achieved by always sorting the list at the end of the evaluation of the path expression or even sorting after each step in the path expression. The first approach may seem more efficient but as shown in [4] can lead to an exponential blow-up of the intermediate results in the size of the query. The second approach, however, has the drawback that the sorting operations become a major bottleneck in the evaluation of the expression. One way to improve this situation is presented in [8] where given a straightforward implementation of the axes unnecessary sorting operations are detected and removed. In this paper we investigate the possibilities for alternative implementations of the axes such that these use the fact that the previous intermediate result is sorted and return a result that is always sorted and duplicate-free. For this purpose we will assume that the document is stored in a DOM-like pointer structure [1] and that the nodes are numbered with their so-called pre-numbers and post-numbers, i.e., their position in a preorder and postorder tree-walk, respectively.

2 Related Work

Previous research on the complexity of XPath evaluation has shown us that it is possible to construct efficient algorithms for the evaluation of XPath [5, 9]. In fact, there even exists a fragment of XPath (Core XPath) that can be evaluated in the linear combined complexity $O(|D| * |Q|)$, where $|D|$ is the size of the instance and $|Q|$ the size of the query [4]. We improve these results in the sense that our approach leads to linear evaluation in the size of the intermediate results, which can be much smaller than the size of the document.

This work is mostly inspired by the results presented in [6] and [7]. There it is shown that if the document nodes have prenumbers and postnumbers associated with them then it possible to efficiently retrieve the results of certain axes in document order and without retrieving nodes that are not in the result. For example, if we let $\mathsf{pre}(n)$ and $\mathsf{post}(n)$ be the pre- and postnumber of n then we can efficiently test their relative positions because then n' is a descendant of n iff $\mathsf{pre}(n') > \mathsf{pre}(n) \wedge \mathsf{post}(n') < \mathsf{post}(n)$, and n' is a follower of n' iff $\mathsf{pre}(n) < \mathsf{pre}(n') \wedge \mathsf{post}(n) < \mathsf{post}(n')$.

3 The Data Model

The *logical data model* that we will use is a simplification of the XML data model where a *document* is an ordered node-labelled tree and for such a tree the *document order* is defined as the strict total order over its nodes that is defined by the preorder tree-walk over the tree.

The *physical data model* describes in an abstract way how we assume that documents are stored. The first assumption that we make is that the following partial functions are available for document nodes (if undefined the result as assumed to be **null**) and can be evaluated in $O(1)$ time:

- $\mathsf{fc}(n)$ returns the first child of n
- $\mathsf{ns}(n)$ returns the next sibling of n
- $\mathsf{ps}(n)$ returns the previous sibling of n
- $\mathsf{pa}(n)$ returns the parent of n
- $\mathsf{ff}(n)$ returns the first follower of n
- $\mathsf{lp}(n)$ returns the last predecessor of n

Note that except for the last two functions these are all existing pointers in the Document Object Model.

The second assumption that we make is that there are functions such that we can retrieve the pre- and postnumbers in $O(1)$ time:

- $\mathsf{pre}(n)$ returns the prenumber of n
- $\mathsf{post}(n)$ returns the postnumber of n

The reasonableness of these assumptions is demonstrated by the fact that this physical data model can be generated from a SAX representation [2] that consists of a string of opening and closing tags in LogSpace. This can be shown by an extension of the proof given in [9] for the original DOM data model.

The data type we will use the most is that of *List*. We use the following operations:

- *newList*() returns a new list
- *first*(L) returns first element of L
- *last*(L) returns last element of L
- *empty*(L) determines if L is empty
- *addAfter*(L, n) adds n at end of L
- *addBefore*(L, n) adds n at begin of L
- *delFirst*(L) removes and returns the first element of L
- *delLast*(L) removes and returns the last element of L
- *isList*(L) determines if L is a list

The lists are assumed to be represented as a reference to a pair that consists of a reference to the beginning and the end, respectively, of a doubly linked list. Therefore we can assume that all the operations above and assignments and parameter passing can be done in $O(1)$ time. Furthermore this means that if an argument of a function or procedure is a *List* then it is passed as a reference and therefore all the operations applied to the formal argument are in fact applied to the original list that was passed as an argument.

We now proceed by giving for each axis the corresponding algorithm.

4 Descendant Axis

4.1 Informal Description

Given a list of document nodes that is in document order and without duplicates we cannot compute a sorted list of descendants by simply concatenating the lists of descendants. The reason for this is that if n_1 and n_2 appear in the list in that order and n_2 is a descendant of n_1 then the descendants of n_2 will appear twice in the result of the concatenation. However, if n_2 is *not* a descendant of n_1 then all descendants of n_2 will follow in document order the descendants of n_1. It follows that we only have to skip the nodes in the input list that are preceded by an ancestor, to get a result that is in document order and without duplicates.

4.2 The Algorithm

We first present a procedure that adds the descendants of a document node n behind a list L in document order.

```
1 proc addDesc(L, n)
2   n' := fc(n);
3   while n' ≠ null do
4         addAfter(L, n');
5         addDesc(L, n');
6         n' := ns(n')
7   od
8 end
```

The procedure iterates over all children of n in document order and for each (1) adds the child to L and (2) adds its descendants to L. Since the descendants of a node precede in document order the descendants of the following siblings, it follows that the result is indeed that all descendants of n are added to L. Furthermore, it is easy to see that the time complexity is $O(m)$ where m is the number of added elements to L.

Next, we present the function that given a sorted and duplicate-free list L_{in} returns the sorted and duplicate-free list of descendants of the nodes in L_{in}.

```
1  funct allDescOrd(L_in)
2    L_out := newList();
3    while ¬empty(L_in) do
4          n := delFirst(L_in);
5          addDesc(L_out, n));
6          while ¬empty(L_in) ∧
7                post(first(L_in)) < post(n)
8                do
9                delFirst(L_in)
10         od
11   od;
12   L_out
13 end
```

The function iterates over the elements in L_{in} and adds their descendants to L_{out} unless they are preceded in the list by an ancestor. In line 7 this is tested by comparing the post numbers of $first(L_{in})$ and n. Since n appeared in L_{in} before $first(L_{in})$ if follows that $\mathsf{pre}(n) < \mathsf{pre}(first(L_{in}))$ and therefore that $first(L_{in}))$ is a descendant of n iff this condition is true. The time complexity of the function is $O(l+m)$ where l is the size of L_{in} and m the size of L_{out}.

5 Descendant-or-self Axis

The algorithm for this axis is identical to that of the descendant axis except that for each node in L_{in} that is not preceded by an ancestor we retrieve not only the descendants but also the node itself. The time complexity is therefore the same as the previous algorithm.

6 Ancestor Axis

6.1 Informal Description

The problem for this axis is similar to the descendant axis because two distinct nodes can have common ancestors. Moreover, this can not only happen for nodes that have an ancestor-descendant relationship, but also for nodes that do not. The solution for this problem is to retrieve for each node in the input list only those ancestors that were not already retrieve before. Because the input list is sorted in document order we can do this by walking up the tree and stopping if we find a node that is an ancestor of the previous node in the input list.

6.2 The Algorithm

We first present two helper procedures. The first retrieves all ancestors of a document node n and appends them in document order after a list L.

```
1 proc addAnc(L, n)
2   n' := pa(n);
3   if n' ≠ null
4     then addAnc(L, n');
5          addAfter(L, n')
6   fi
7 end
```

If the number of ancestor nodes in m then the time complexity if this procedure is in $O(m)$.

The next helper procedure will, given a list L, a document node n and a document node n' that precedes n in document order, retrieve the ancestors n that are not ancestors of n' and append them in document order after L.

```
1 proc addAncUntilLeft(L, n, n')
2   n'' := pa(n);
3   if n'' ≠ null ∧ pre(n'') ≥ pre(n')
4     then addAncUntilLeft(L, n'', n');
5          addAfter(L, n'')
6   fi
7 end
```

Note that since n' precedes n in document order it holds that the condition $\mathsf{pre}(n'') \geq \mathsf{pre}(n')$ indeed checks if an ancestor n'' of n is an ancestor of n'. Also here the time complexity is $O(m)$ where m is the number of retrieved ancestors.

Finally, we present the function that given a sorted and duplicate-free list of document nodes L_{in} returns a sorted duplicate-free list of all their ancestors.

```
1 funct allAncOrd(L_in)
2   L_out := newList();
3   if ¬empty(L_in)
4     then n := delFirst(L_in);
5          addAnc(L_out, n)
6   fi;
7   while ¬empty(L_in) do
8          n' := delFirst(L_in);
```

```
9           addAncUntilLeft(L_out, n', n);
10          n := n'
11    od;
12    L_out
13  end
```

In the first part of the algorithm (line 3-6) all ancestors of the first node in L_{in} are retrieved. After this a while loop (line 7-11) iterates over the remaining nodes in L_{in} and retrieves for each node n' all ancestors of n' that are not ancestors of n, the node that preceded n' in L_{in}. Since n also precedes n' in document order it follows that all the ancestors of n' that are retrieved indeed follow those that were retrieved for n. As a result all ancestors that are retrieved are appended in document order. The time complexity of this function is $O(l+m)$ if l is the size of L_{in} and m is the size of the result.

7 Ancestor-or-self Axis

The algorithm for this axis as similar to the one for the ancestor axis except that we we retrieve the ancestors of a node we also add the node itself. The time complexity is therefore also the same.

8 Child Axis

8.1 Informal Description

We cannot use the approach of the previous axes here. Consider for example the fragment in Figure 1. If, for example, we only retrieve for each node the children that we know to precede in document order the children of the next node then for the list $L_{in} = [1,3]$ we only obtain $[2,3,4]$. To solve this we introduce a stack on which we store the children of node 1 which were not retrieved already such that we can return to them when we are finished with the children of node 3.

```
<a id="1">
  <b id="2"/>
  <b id="3"> <c id="4"/> </b>
  <b id="5"/>
</a>
```

Figure 1: An XML fragment

8.2 The Algorithm

Before we present the actual algorithm we present a helper function that results in a list of all children of a document node n in document order.

```
1 funct allChildren(n)
2   L := newList();
3   n' := fc(n);
4   while n' ≠ null do
5         addAfter(L, n');
6         n' := ns(n')
7   od;
8   L
9 end
```

The function simply goes to the first child of n and then follows the following-sibling reference until there is no more following sibling. The time complexity of this function is $O(m)$ if m is the number of retrieved children.

Next, we present the actual algorithm that given a sorted and duplicate-free list of document nodes L_{in} returns a sorted duplicate-free list of all their children.

```
1  funct allChildOrd(L_in)
2    L_out := newList();
3    L_st := newList();
4    while ¬empty(L_in) do
5          n := first(L_in);
6          if empty(L_st)
7            then L' := allChildren(n);
8                 addBefore(L_st, L');
9                 delFirst(L_in);
10         elsif empty(first(L_st))
11               then delFirst(L_st)
12         elsif pre(first(first(L_st))) > pre(n)
13               then L' := allChildren(n);
14                    addBefore(L_st, L');
15                    delFirst(L_in)
16                else n' := delFirst(first(L_st));
17                    addAfter(L_out, n')
18         fi
19   od;
20   while ¬empty(L_st) do
21         if empty(first(L_st))
22           then delFirst(L_st)
23            else n' := delFirst(first(L_st));
24                 addAfter(L_out, n')
25         fi
26   od;
27   L_out
28 end
```

The algorithm consists of two while loops. The first (line 4-19) iterates over the nodes in L_{in} and retrieves the children that it knows it can send to the output list L_{out} and stores the

others on the stack L_{st}. The second while loop (line 20-26) iterates over the remaining children on the stack L_{st} and appends those behind L_{out}. In the following we discuss each while loop in more detail.

The first loop stores unprocessed children on the stack L_{st} where the beginning of L_{st} is the top of the stack. Each position on the stack contains a sorted list of siblings that were not yet transferred to L_{out}. The loop maintains an invariant that states that the nodes in lists that are higher on the stack precede in document order those that are lower on the stack. This is mainly achieved by the **if** statement on line 12 that tests if the node at the beginning of L_{in} precedes the first child node on top of the stack. If this is true then the list of children of n are pushed on the stack and n is removed from L_{in}, otherwise the first child node on top of the stack is moved to the end of L_{out}. Note that in the latter case it indeed holds that all the children of the remaining nodes in L_{in} indeed succeed this child in document order.

The second loop simply flushes the stack which indeed results in adding the remaining nodes to L_{out} in document order because of the invariant that was described for the previous loop.

Since the algorithm iterates over all the nodes in L_{in} and retrieves only those nodes that are added to L_{out} it follows that the time complexity is $O(l+m)$ where l is the size of L_{in} and m the size of L_{out}.

9 Parent Axis

9.1 Informal Description

The fundamental property that will be used for this axis is that if for a duplicate-free sorted list of document nodes we retrieve the parent nodes we obtain a sublist of the list of nodes that we meet when we follow the contour of the tree. For example, if we follow the contour of the nodes in the tree for the fragment in Figure 1 then we obtain the list $[1,2,1,3,4,3,1,5,1]$. If we start with the list $[2,3,4,5]$ and we retrieve the list of parents, then we obtain $[1,1,3,1]$ which is indeed a sublist of the first list.

This information can be used by the algorithm because when it iterates over the list of parents and encounters a parent n that precedes the last parent in the output list then it is walking up the tree in the contour walk. As a consequence it knows that after it inserts n in the output list the tail of the output list that starts with n will not change anymore because all the following nodes in the input list will either be after or before this tail in document order. Therefore the algorithm can simply summarize this tail and pretend it corresponds to the node n. It does this by replacing it with a nested list that contains this tail.

As an illustration consider the following possible list of parents: $[1,2,5,4,9,8,2]$. For reasons of homogeneity we represent the output list as a list of lists and if we add a single node it is represented as a singleton list. Therefore after processing the nodes 1, 2 and 5 we obtain the list $[[1],[2],[5]]$. Since the next node 4 precedes 5 the algorithm represents the tail as a nested list that starts with 4 and obtains $[[1],[2],[4,[5]]]$. From this point on the nested list $[4,[5]]$ will be considered as if equal to $[4]$, i.e., the algorithm considers only the first node of the nested lists. Since the next node 9 follows 4 it is simply added, giving $[[1],[2],[4,[5]],[9]]$. The next node is 8 which precedes 9 but follows 4, so we obtain $[[1],[2],[4,[5]],[8,[9]]]$. Also here the nested list $[8,[9]$ is considered as equivalent to $[8]$. Finally the node 2 is added and since it precedes node 4 the two lists starting with 4 and 8 are nested in a list starting with 2 and we obtain $[[1],[2],[2,[4,[5]],[8,[9]]]]$.

As will be clear from the previous example, the result is a nested list that when flattened gives the sorted list of parents but may still contain duplicates. Since the list is sorted these can be eliminated easily.

9.2 The Algorithm

We first present a helper function and a helper procedure. The following function will, given a sorted list L, return a sorted list that contains all the elements in L but no duplicates.

```
1  funct dupElimSort(L)
2    L_out := newList();
3    if ¬empty(L) then n := delFirst(L) fi;
4    while ¬empty(L) do
5          n' := delFirst(L);
6          if n' ≠ n
7            then addAfter(L_out, n')
8                 n := n'
9          fi
10   od;
11   L_out
12 end
```

The time complexity of this function is clearly $O(l)$ if l is the size of L.

Te following procedure will, given a list L and a nested list L_{tr} of document nodes, flatten the list L_{tr} and append it to L.

```
1 proc addFlatList(L, L_tr)
2   while ¬empty(L_tr) do
3         n := delFirst(L_tr);
4         if isList(n)
5           then addFlatList(L, L_tr)
6            else addAfter(L, n)
7         fi
8   end
9 end
```

If at each level every nested list in L_{tr} contains at least one document node then the time complexity if this procedure is $O(m)$ if m is the number of document nodes in the result.

Finally, we present the actual algorithm that given a duplicate-free sorted list of document nodes will return a duplicate-free sorted list of their parents.

```
1  funct allParOrd(L_in)
2    L_tr := newList();
3    while ¬empty(L_in) do
4          n := pa(delFirst(L_in));
5          L := newList();
6          while ¬empty(L_tr) ∧
7                  pre(first(last(L_tr))) > pre(n)
8                  do
9                  n' := delLast(L_tr);
10                 addBefore(L, n')
11         od;
12         addBefore(L, n);
13         addAfter(L_tr, L)
14   od;
15   L_dup := newList();
16   addFlatList(L_dup, L_tr);
17   L_out := dupElimSort(L_dup);
18   L_out
19 end
```

The list L_{tr} is used to represent the list of nested lists. Note that in L_{tr} every nested list will always start with a document node. The crucial part is the while loop on lines 6-11 that determines the tail L of L_{tr} where the first nodes of the lists in this tail follow n in document order and removes this tail from L_{tr}. On line 12 this tail is extended with n and finally on line 13 the tail is put back as a nested list at the end of L_{tr}. At the end of the algorithm, when all the nodes of L_{in} have been processed, the resulting nested list L_{tr} is flattened and duplicates are removed from it.

The time complexity if this algorithm is $O(l)$ where l is the size of Lin. To understand this consider the number of times the pre-numbers of two nodes are compared in the while condition starting on line 6. The number of equations that were false are at most l, one for each parent that is considered. The number of successful equations is also at most l because a successful comparison means that the node is from then on nested and will no longer be considered, so in the final L_{tr} every document node has been successfully compared at most once.

10 Following Axis

10.1 Informal Description

To find all the followers of the nodes in a duplicate-free sorted list of document nodes it is sufficient to retrieve the followers of the first node in the list that is not an ancestor of the next node in the list. To understand this consider the following. Let n be this node and n' a node that is in the list after n. Since the node in the list immediately after n is not its descendant n' and its followers are also not descendants of n. Therefore it follows that (1) n' is a follower of n and (2) all followers of n' are also followers of n. Since n' is not a follower of itself, it holds that the set of followers of n' is a proper subset of those of n. On the other hand it can be shown that if n' is an ancestor of n then the set of followers of n' is a subset of those of n.

10.2 The Algorithm

We first present a helper procedure that, given a list L and a document node n, appends to L all followers of n.

```
1 proc addFoll(L, n)
2   if ff(n) ≠ null
3     then addAfter(L, ff(n));
4          addDesc(L, ff(n));
5          addFoll(L, ff(n))
6   fi
7 end
```

The correctness of this procedure follows from the fact that $\mathsf{ff}(n)$ returns the smallest node (in document order) that is a follower of n, and that the followers of a node n are defined as those nodes that are larger in document order but not a descendant of n. Its time complexity is $O(m)$ where m is the number of followers added to L.

Next we present the actual algorithm that given a duplicate-free sorted list L_{in} of document nodes returns a duplicate-free sorted list of all the followers of these nodes.

```
1  funct allFollOrd(L_in)
2    L_out := newList();
3    if ¬empty(L_in)
4      then n := delFirst(L_in);
5           while ¬empty(L_in) ∧
6                 pre(first(L_in)) > pre(n) ∧
7                 post(first(L_in)) < post(n)
8                 do
9                 n := delFirst(L_in)
10          od;
11          addFoll(L_out, n);
12   fi;
13   L_out
14 end
```

The correctness of this function follows from what was said before and the fact that the while condition indeed tests that the first node in L_{in} is a descendant of n. Because the algorithm iterates over all the nodes in L_{in}, determines a single node n and then applies *addFoll*, it follows that the time complexity is $O(l+m)$ if l is the size of L_{in} and m the size of L_{out}.

11 Preceding Axis

11.1 Informal Description

To find the preceding nodes of a sorted list of document nodes we only have to retrieve the preceding nodes of the last node in the list. If this is node n we can retrieve its preceding nodes in document order as follows. We first apply to n the function ff repeatedly until there is no more immediate predecessor. Let the nodes we encounter be $n_0 = n, n_1, \ldots, n_k$. Then n_k is the first predecessor of n in document order. For this node we first retrieve all its ancestors in document order that are not ancestors of n and n_k itself. After this we return to n_{k-1} and retrieve all its ancestors in document order that are not ancestors of n_k and also not ancestors of n, and we retrieve n_{k-1} itself. We repeat this for each n_i with $0 < i < k$ by retrieving all ancestors of n_i in document order that are not ancestors of n_{i+1} or n, and n_i itself. It is easy to see that for each n_i the retrieved nodes follow in document order those of n_{i+1} and that those nodes are predecessors of n. Conversely all predecessors of n are either in $n_1, \ldots, n_k$ or one of their ancestors.

11.2 The Algorithm

Before we present the actual algorithm we present three helper algorithms. The first algorithm will, given a list L and three document nodes n, n' and n'' such that n' is a predecessor of n and n is a predecessor of n'', adds after L in document order the ancestors of n that are not ancestors of n' or n''.

```
1  proc addAncBetween(L, n, n', n'')
2    n''' := pa(n);
3    if n''' ≠ null ∧
4      (pre(n''') ≥ pre(n')) ∧
5      (post(n''') ≤ post(n''))
6      then addAncUntil(L, n''', n');
7           addAfter(L, n''')
8    fi
9  end
```

Note that to test if n''' is not an ancestor of n' it must be tested whether $\neg(\mathsf{pre}(n''') < \mathsf{pre}(n') \wedge \mathsf{post}(n''') > \mathsf{post}(n'))$ or equivalently $\mathsf{pre}(n''') \geq \mathsf{pre}(n') \vee \mathsf{post}(n''') \leq \mathsf{post}(n')$. However, since n is a follower of n' it holds that $\mathsf{post}(n) > \mathsf{post}(n')$ and since n''' is the parent of n it holds that $\mathsf{post}(n''') > \mathsf{post}(n)$, from which it follows that $\mathsf{post}(n''') > \mathsf{post}(n')$. A similar argument shows that the test for n''' and n'' in the if-expression is also sufficient to test whether n''' is not an ancestor of n''. The time complexity of this procedure is $O(m)$ if m is the number of nodes added to L.

The second helper function is similar and will, given a list L and document nodes n and n' such that n is a predecessor of n', add all the ancestors of n that are not ancestors of n' to L in document order.

```
1  proc addAncUntilRight(L, n, n')
2    n'' := pa(n);
3    if n'' ≠ null ∧ post(n'') ≤ post(n')
4      then addAncUntilRight(L, n'', n');
5           addAfter(L, n'')
6    fi
7  end
```

The correctness of this procedure can be shown in a way similar to that of the previous one. Also here the time complexity is $O(m)$ if m is the number of added document nodes.

The third helper procedure will, given a list L and two document nodes n and n' such that n is a predecessor of n', adds all those nodes to L in document order which are (1) predecessors of n but not ancestors of n', (2) ancestors of n but not ancestors of n' or (3) n itself.

```
1  proc addLeftUntil(L, n, n')
```

```
2    if lp(n) ≠ null
3      then n'' := lp(n);
4           addLeftUntil(L, n'', n');
5           addAncBetween(L, n, n'', n')
6       else addAncUntilRight(L, n, n')
7    fi;
8    addAfter(L, n)
9  end
```

The correctness of this procedure follows from the correctness of the previous procedures. The time complexity is $O(m)$ if m is the number of added document nodes.

Finally, we present the algorithm itself which, given a list L_{in} of document nodes returns a duplicate-free sorted list of all their predecessors.

```
1  funct allPredOrd(L_in)
2    L_out := newList();
3    if ¬empty(L_in)
4      then n := last(L_in);
5           if lp(n) ≠ null
6             then addLeftUntil(L_out, lp(n), n)
7           fi
8    fi;
9    L_out
10 end
```

The correctness follows from the correctness of the helper procedures. The time complexity is $O(m)$ if m is the size of L_{out}.

12 Following-Sibling Axis

12.1 Informal Description

The problems for this axis are very similar to those of the child axis and can be solved in the same way, i.e., by introducing a stack of lists of nodes that contains the nodes that still need to be move to the output list. An extra complication is here that the following siblings of two different nodes may have nodes in common. The solution for this is simple: if we encounter simultaneously the same node in the input list and at the beginning of the list on top of the stack the we ignore the node in the input list.

12.2 The Algorithm

We first present a helper function that given a document node n returns a duplicate-free sorted list of all the following siblings of n.

```
1  funct allFollSibl(n)
2    L := newList();
3    n' := ns(n);
4    while n' ≠ null do
5         addAfter(L, n');
6         n' := ns(n')
7    od;
8    L
9  end
```

Correctness of this function follows from the fact that the function ns returns the first sibling of n that follows n in document order. The time complexity is $O(m)$ where m is the size of the result.

Next we present the actual algorithm which given a list L_{in} of document nodes returns a duplicate-free sorted list of all following siblings of the nodes in this list.

```
1  funct allFollSiblOrd(L_in)
2    L_out := newList();
3    L_st := newList();
4    while ¬empty(L_in) do
5         n := first(L_in);
6         if empty(L_st)
7           then L' := allFollSibl(n);
8                addBefore(L_st, L');
9                delFirst(L_in);
10        elsif empty(first(L_st))
11              then delFirst(L_st)
12        elsif pre(first(first(L_st))) > pre(n)
13              then L' := allFollSibl(n);
14                   addBefore(L_st, L');
15                   delFirst(L_in)
16        elsif pre(first(first(L_st))) < pre(n)
17              then n' := delFirst(first(L_st));
18                   addAfter(L_out, n')
19               else delFirst(L_in)
20        fi
21   od;
22   while ¬empty(L_st) do
23         if empty(first(L_st))
24           then delFirst(L_st)
25            else n' := delFirst(first(L_st));
26                 addAfter(L_out, n')
27         fi
28   od;
29   L_out
30 end
```

The correctness of this function is similar to that of the corresponding function for the child axis. The main difference is in line 19 where the case is considered that the current node in the input list, n, is equal to the first node of the list on top of the stack L_{st}. In this case the node n is removed from the input list without copying its siblings to the stack or the output list. The time complexity is also similar, i.e., $O(l + m)$

where l is the size of L_{in} and m is the size of L_{out}.

13 Preceding-Sibling Axis

13.1 Informal Description

This axis is symmetric to the following-sibling axis in the sense that we can use the same algorithm except that we have to do everything in reverse, i.e., we iterate over the input list in reverse and we move nodes from the back of the lists on the stack to the front of the output list.

13.2 The Algorithm

We first present a helper function that give a document node n returns a duplicate-free sorted list of all preceding siblings of n.

```
1  funct allPrecSibl(n)
2    L := newList();
3    n' := ps(n);
4    while n' ≠ null do
5          addBefore(L, n');
6          n' := ps(n')
7    od;
8    L
9  end
```

Similar to the previous axis correctness of this function follows from the fact that the function ps returns the last sibling of n that precedes n in document order. Also here the time complexity is $O(m)$ where m is the size of the result.

Finally we present the algorithm that given a duplicate-free and sorted list of document nodes L_{in} returns a duplicate-free and sorted list of all the preceding siblings of these document nodes.

```
1  funct allPrecSiblOrd(L_in)
2    L_out := newList();
3    L_st := newList();
4    while ¬empty(L_in) do
5          n := last(L_in);
6          if empty(L_st)
7            then L' := allPrecSibl(n);
8                 addBefore(L_st, L');
9                 delLast(L_in);
10         elsif empty(first(L_st))
11               then delFirst(L_st)
12         elsif pre(last(first(L_st))) > pre(n)
13               then L' := allFollSibl(n);
14                    addBefore(L_st, L');
15                    delLast(L_in)
16         elsif pre(last(first(L_st))) < pre(n)
17               then n' := delLast(first(L_st));
18                    addBefore(L_out, n')
19               else delFirst(L_in)
20         fi
21   od;
22   while ¬empty(L_st) do
23         if empty(first(L_st))
24           then delFirst(L_st)
25           else n' := delLast(first(L_st));
26                addBefore(L_out, n')
27         fi
28   od;
29   L_out
30 end
```

The correctness follows from the symmetry with the previous axis, and for the same reason the time complexity is also $O(l + m)$ with l the size of L_{in} and m the size of L_{out}.

14 Conclusion

The presented algorithms allow us to efficiently evaluate XPath axes, preserving both order and duplicate freeness, assuming that the document is stored as a DOM structure and pre- and post-numbers are available. The algorithms only evaluate the axis and do not evaluate node tests or predicates. As long as the predicates do not refer to the context set of the resulting nodes, i.e., use directly or indirectly the functions `position()` and `last()`, these can be easily applied to the result of the algorithms afterwards. If there is a reference to the context set then there is a problem because the resulting list of nodes does not contain any information about the context set. In this case we can either attempt to reconstruct the context set (e.g., if the step was `child::*` then the context set of n is simply all its siblings) or fall back to the optimization techniques proposed in [8]. Because both techniques guarantee that the result of each step is duplicate-free and sorted it is possible to mix the two techniques within the same path expression.

References

[1] Document Object Model (DOM). Available at: `http://www.w3c.org/dom/`.

[2] Simple API for XML (SAX). Available at: `http://www.saxproject.org/`.

[3] 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`.

[4] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, 2002.

[5] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. of the 22nd ACM SIGACT-SIGMOD-GIGART Symposium on Priciples of Database Systems (PODS)*, San Diego (CA), 2003.

[6] T. Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, Madison, 2002.

[7] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *VLDB 2003*, 2003.

[8] J. Hidders and P. Michiels. Avoiding Unnecessary Ordering Operations in XPath. In *DBPL 2003*, 2003.

[9] L. Segoufin. Typing and querying XML documents: Some complexity bounds. In *Proc. of the 22nd ACM SIGACT-SIGMOD-GIGART Symposium on Priciples of Database Systems (PODS)*, San Diego (CA), 2003.