Supporting Positional Predicates in Efficient XPath Axis Evaluation for DOM Data Structures

Torsten Grust   Jan Hidders   Maurice Van Keulen

Philippe Michiels

Roel Vercammen[1]

July 7, 2004

Abstract

In this technical report we propose algorithms for implementing the axes for element nodes in XPath given a DOM-like representation of the document.

First, we construct algorithms for evaluating simple step expressions, withoout any (positional) predicates. The time complexity of these algorithms is at most $\mathcal{O}(l + m)$ where l is the size of the input list and m the size of the output list. This improves upon results in [6] 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.

In a second phase we give a description of how the support for positional predicates can be added to the algorithms with a focus on maintaining the efficiency of evaluation. 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.

Contents

List of Algorithms

Chapter 1

Introduction

Many correct XPath implementations, *e.g.*, Galax [5], employ an evaluation strategy that closely corresponds to the XQuery Formal Semantics [3]. This guarantees correctness, but in many cases, results in inefficient query evaluation. The most common problems with this approach are

1. unnecessary intermediate sorting and duplicate-elimination operations, which are implied by the formal semantics after each axis step of a path expression, and

2. the fact that some nodes are unnecessarily accessed multiple times upon the evaluation of one step.

The former also jeopardizes pipelining of XPath step evaluations. Many partial solutions to these problems have already been presented.

In the context of these implementations that literally implement the formal semantics, some partial techniques have been developed to tackle the problem of order and duplicates. For instance, in [11, 4] it is statically decided which sorting and duplicate-elimination operations are necessary to guarantee correctness. This approach, however, does not solve the problem of unnecessarily accessing nodes multiple times, and in many cases some sorting and duplicate-elimination operations still occur in the evaluation of a path expression.

In a second, more complicated approach, efficient algorithms are presented for which the result for all input nodes of the step is computed at once. These algorithms [9, 8] guarantee that the result is free from duplicates and in document order and also solves the problem of redundant accesses of nodes. However, these algorithms cannot be used for path expressions containing positional predicates, which is one of the problems we will tackle in this work.

Note that none of the two former approaches is better than the other one. The first approach has the drawback that not all combinations of axes can be handled efficiently. On the other hand, when in the first approach sorting and duplicate elimination is found to be unnecessary, naive evalution of the corresponding step is more efficient than it is under the second, more complex approach. Hence there is a trade-off to be made between these two approaches. As a result, one could propose a hybrid approach, where some steps of a path expression are evaluated using the first approach, and other steps by employing the second approach. The hybrid approach can benefit from the results in this paper in the sense that we now also have a choice for steps that contain a positional predicate.

The *main contributions* of this work are

1. 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. the development of axis algorithms with support for evaluation of multiple predicates which remain efficient for the majority of XPath expressions.

By efficient we mean within the polynomial complexity bounds identified by Gottlob et al. [7, 6]. They have shown that XPath evaluation is P-complete, and presented an algorithm to evaluate XPath expressions in $\mathcal{O}(|Q|^2 \times |D|^4)$, where $|D|$ denotes the document size and $|Q|$ the size of the path expression. Nevertheless, most XPath implementations show exponential behavior in the size of the query. The algorithms we present are polynomial for XPath expressions that do not contain arbitrarily deeply nested predicates, with a worst case time complexity of $\mathcal{O}(|Q| \times |D|^2)$.

Chapter 2

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. For such a tree, the *document order* is defined as the strict total order over its nodes, which is defined by the preorder tree-walk over the tree. A preorder tree-walk is a simple recursive descent from the root, i.e., it begins with the root node, then goes to its first child, does a preorder tree-walk rooted at that child, proceeds with the second child, again doing a preorder tree-walk rooted at that second child, etc. In this way, you visit each node in document order.
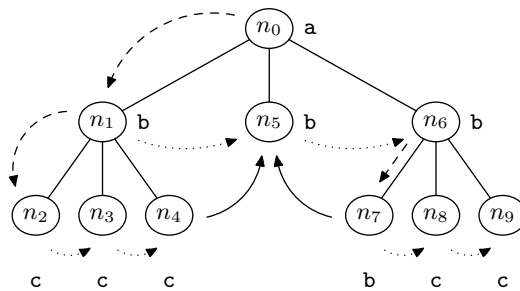
Figure 2.1: The data model for an example XML document.

The *physical data model* describes in an abstract way how we assume that documents are stored. Our first assumption is that the partial functions in Figure 2.2 are available for document nodes (if undefined, the result as assumed to be **null**) and can be evaluated in $\mathcal{O}(1)$ time:

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

Figure 2.2: Node accessor functions. These functions return **null** if the requested property is undefined for node n.

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

Furthermore, we define the following function (also assumed $\mathcal{O}(1)$) to retrieve the next node in document order based on $\mathsf{fc}(n)$ and $\mathsf{ff}(n)$.

Function *nextNode(n)*

---
if $\mathsf{fc}(n) \neq$ **null then**
$\quad | \quad \mathsf{fc}(n);$
else
$\quad \lfloor \quad \mathsf{ff}(n);$

Our second assumption is that there are functions such that we can retrieve the preorder and postorder ranks in $\mathcal{O}(1)$ time:

- $\mathsf{pre}(n)$ returns the preorder rank of n

- $\mathsf{post}(n)$ returns the postorder rank 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 [14] for the original DOM data model.

We will frequently use the data type *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

Lists are assumed to be represented as a reference to a tuple that consists of a reference to the beginning and the end of a doubly linked list. Therefore, we can assume that all the operations above and assignments and parameter passing can be done in $\mathcal{O}(1)$ time. Furthermore, it means that if an argument of a function or procedure is a *List*, then it is passed by reference, hence all operations applied to the formal argument are in fact applied to the original list.

We can use an *Iterator* on a *List*. The following operations are available:

- *newIterator(L)* returns a new iterator for the list L

- *toFirst(I)* sets the iterator to the first element of the list

- *toLast(I)* sets the iterator to the last element of the list

- *toNext(I)* sets the iterator to the next element of the list

- *toPrev(I)* sets the iterator to the previous element of the list

- *current(I)* returns the reference of the tuple to which the iterator points

We also use the data type *Tuple* frequently. A tuple with k fields $f_1, \ldots, f_k$ can be assigned a value in following way: $t := \langle f_1 = a_1, \ldots, f_k = a_k \rangle$. A field with name f of a tuple t can be accessed by $t.f$. If $t.f$ is a pointer, it is implicitly dereferenced.

Chapter 3

Supporting (Positional) Predicates

In [12] only path expressions without positional predicates are supported. In this paper we want to extend these algorithms with support for evaluating positional predicates while remaining efficient. By this we mean that we still do not want to visit any nodes that are not part of the input and/or output, and where possible we would like to preserve the pipelining capabilities of the step algorithms.

For a large part, path expressions with multiple predicates can simply be evaluated by evaluating `axis::nodetest` and subsequently filtering out those elements for which one of the `predicate`s does not hold. Step expressions containing *positional predicates*, however, need special attention, because for these this simple strategy may not produce correct answers as the following example demonstrates.

Example Consider the tree in Figure 3.1. Suppose, we are about to to evaluate the path expression[1]

$$/desc::a/desc::b[1]$$

The result is (n_1, n_3). Note that we may not alternatively evaluate `/desc::a/desc::b` (yielding (n_1, n_3, n_4)) and then evaluate the predicate for each of the resulting nodes, because the predicate refers to the *relative* position of a descendant b-node with respect to its context a-node. Take, for example, node n_3. With respect to n_0, it has position 2; with respect to n_2, however, it has position 1. Because of the latter, n_3 is part of the result. In other words, result nodes can have more than one relative position with respect to different context nodes and each of these may qualify the node for the predicate.
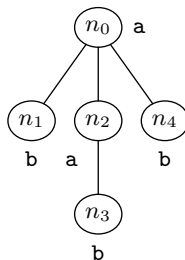

Figure 3.1: Example XML tree without the document node.

Because of this, we explicitly provide an additional algorithm for each axis, that efficiently tracks position information and evaluates the predicate for each (node,position)–combination.

---

[1]For reasons of space, we use a short notation for the axes, e.g. `descendant` becomes `desc`.

In order to support the correct evaluation of predicates, we need to somehow keep track of the context, as nodes are processed. It is clear that in case positional predicates are present, we cannot just prune irrelevant nodes from the input list, like it is done in the earlier algorithms presented in [8]. Pruning these nodes without further notice corresponds to cutting away the vital context information we need to correctly evaluate positional predicates. The previous example demonstrates this.

To bypass this problem, we keep track of all relevant input nodes that we already have processed on a context stack L_{st}. For each of these nodes, we also store the number of result nodes for which this node acted as a context. The following example shows the consecutive states of the context stack L_{st} during the evaluation of the last step of the path expression `/desc::a/desc::*[2]` on the example tree in Figure 3.1.

Example The input list for the last step is the result of the expression `/desc::a`, which is the sequence n_0, n_2. The algorithm starts with the first input node, and pushes it on the context stack, with position 0 (see Figure 3.2 (a)). Now, we start processing the descendants of this context node. As we process the first descendant node n_1, we increment the context position of context node n_0 (see Figure 3.2 (b)). The next node to process is n_2, which is also a context node. We first increment the context position of n_0 and then register n_2 as a new context node with position 0 (Figure 3.2 (c)). Note that, since the predicate matches the current context position associated with n_0, the current node n_2 is sent to the output. Next, node n_3 is processed by incrementing the context position for all contexts (Figure 3.2 (d)). Finally node n_4 is processed. Since all descendants of n_2 have now been handled, we remove the second context from the stack and increase the context position for n_0 (Figure 3.2 (e)).

cn	**pos**
n_0	0

(a)

cn	**pos**
n_0	1

(b)

cn	**pos**
n_0	2
n_2	0

(c)

cn	**pos**
n_0	3
n_2	1

(d)

cn	**pos**
n_0	4

(e)

Figure 3.2: States of the context stack L_{st} while evaluating the last step of the expression `/desc::a/desc::*[2]`.

Just keeping the position of the current node with respect to each context, however, is not enough. In order to support the evaluation of multiple predicates, we will need to alter the structure of the current context stack. Besides the position of the current node relative to the context nodes, we also extend the stack by a column for each predicate in the predicate list. A node test can be seen as a predicate so we treat it in the same way. The example below shows how such a context stack then operates to decide whether the current node gets filtered by the predicates or not.

Looking at the example tree in Figure 3.1, we will try to evaluate the expression

$$\texttt{/desc::b/anc::a[child::b][2][cnt(desc::a)>3]}$$

After the evaluation of the step `desc::b`, the input list for the next step, L_{in} contains the nodes n_1, n_3 and n_4. While evaluating the next step, the algorithm runs iteratively over the candidate output nodes. Since the step contains a reverse axis, this is done in reverse doc-order: n_2, n_0. With the current algorithm and upon handling of node n_0, the context stack looks like this:

cn	**pos**
n_4	1
n_3	2
n_1	1

The problem now is that upon the evaluation of multiple predicates, each predicate works on the result of the previous predicate. So we need to keep track of how many nodes have come through for each predicate, in case a positional predicate occurs in the list of predicates.

We can do this by adding a column for each predicate to the tuples on the context stack. Column **pos** is replaced by a column that is labelled with the node test for this step, indicating that it is only incremented if the node test succeeds. We now have a stack with tuples consisting of a context node and a list of positional indices:

cn	a	[child::b]	[2]	[cnt(desc::a)>3]
n_4	1	0	0	0

Upon evaluating the predicates in the algorithms the following is done for each of the context entries in the modified context stack:

1. Predicate columns are processed left-to-right.

2. For each of the predicate columns, the predicate is evaluated.

   - If the predicate evaluates to `true`, the column's value is incremented with 1 and this step is repeated for the next predicate;
   - If it evaluates to `false`, then the predicate evaluation stops for this context;

3. If the predicate is a positional predicate, it must take as position the value of the *previous* predicate column;

4. If the last predicate evaluates to `true`, the candidate output node is sent to the output. Note that we must complete the evaluation of the predicates for all contexts. Even if we already know that the candidate node can be sent to the output list, we have to evaluate the predicates in the remainder of the context stack, because the positions that are calculated can be necessary for correctly processing future candidate output nodes that use those contexts.

We can illustrate the operation of this technique by means of the above example. Suppose we have evaluated the first step, resulting in the sequence $\{n_1,n_3,n_4\}$. Before running through the candidate output nodes we put the first context node n_4 on the context stack. The first candidate output node to look at is n_2. It has n_3 as only context node. Note that we need to keep n_4 on the context stack, since there are still ancestors of n_4 to be processed. However, n_4 is not a context of the current node n_2. To solve this, we consider only a part of the stack, called the *active part* of the context stack. We assume that this part is always located at the top of the stack. The context stack now looks as follows (active part is highlighted):

cn	a	[child::b]	[2]	[cnt(desc::a)>3]
n_3	1	0	0	0
n_4	1	0	0	0

Since the first predicate evaluates to `true`, the corresponding value gets incremented. In the second column this new value is checked against the position index and, since they do not match, predicate evaluation is stopped for this context. There are no more active contexts, so nothing goes to the output, and we proceed with the next candidate output node n_0. The context stack now looks as follows:

cn	a	[child::b]	[2]	[cnt(desc::a)>3]
n_3	1	1	0	0
n_4	1	0	0	0

Two contexts, n_1 and n_4 become active and thus the stack now looks like this:

cn	a	[child::b]	[2]	[cnt(desc::a)>3]
n_1	1	0	0	0
n_3	2	1	0	0
n_4	1	0	0	0

9

Similar to the previous step, the evaluation of the predicates for contexts n_1 and n_4 stops after the first predicate. However, for the context n_3, the first predicate evaluates to `true` again. Since now the new value of the first predicate column does match the position index of the second column, also this predicate column's value gets incremented. Finally, also the last predicate evaluates to `true` and as such, the node n_0 is sent to the output. The context stack now looks as follows:

cn	a	[child::b]	[2]	[cnt(desc::a)>3]
n_1	1	0	0	0
n_3	2	2	1	1
n_4	1	0	0	0

We represent the list of predicates as a list L_P of functions, that each take for parameters:

- n – the candidate output node;

- *position* – the context position of the candidate output node in the context sequence.

Function *createPosList(L_P)*

---

begin
    $I :=$ newIterator(L_P);
    toFirst(I);
    $L_{pos} :=$ newList();
    while *current(I)* $\neq$ **null do**
        addAfter(L_{pos}, 0);
        toNext(I);
    return L_{pos};
end

The positions of the numbers in the lists associated to the contexts in L_{st} correspond to positions of the predicate functions in the predicate list L_P. Note that the axis step's node test is taken as the first predicate. We define two functions, a function *createPosList* that constructs the ordered list which associates every predicate with a position and a function *evaluatePredicates* that checks whether a node gets filtered by the predicates and if it is, it is sent to the output list.

Before continuing with the discussion of the several axis algorithms, we shortly discuss the correctness Algorithm 3. As a precondition, we assume that the context stack L_{st} contains all contexts of the current node n and that these are in the top section of L_{st} which is delimited by the iterator I_{until}. For all entries on the context stack it holds that the positions in the position lists denote how many nodes did pass all predicates up to the corresponding predicate *before* the processing of the current node n. This last condition also holds as postcondition. The main invariant of this algorithm is that after evaluating a certain predicate for the current node n, the position corresponding to this predicate and the current context node indicates how many nodes have passed through this predicate for this particular context. Achieving this for every predicate and every context enforces the postcondition.

Note that in this case, the items on the context stack are processed one by one. This results in multiple evaluations of predicates. For predicates that do not refer to the context position, this is unnecessary. A possible optimization would be to evaluate the predicates that do not refer to the context position only once for all contexts.

Also note that there is a small catch in the evaluation of predicates. Algorithm 3 describes an approach that does not jeopardize pipelining, in the sense that none of the output nodes need to be buffered. A problem occurs when the predicate involves aggregate function calls like the `last()` function. The size of the context sequence is only known if the previous predicate has been completely evaluated (*i.e.*, once the algorithm for that step expression ran to completion). As such, the algorithm presented here above does not support the correct evaluation of this kind of predicates. However there are easy ways around this problem, while preserving complexity results. It is obvious, though, that any such solution will break the pipeline.

Algorithm 3: evaluatePredicates

Input:

- L_{st} : the context stack,

- I_{until} : iterator delimiting the active part of the context stack,

- n : the current node,

- L_P : the list of predicates

Output: a boolean indicating whether the current node is retained by the predicates

1 begin

2 $I := \text{newIterator}(L_{st})$;

3   toFirst(I);

4   output := **false**;

     `// for all active contexts`

5   **while** *current(I) $\neq$ current(I_{until})* **do**

6 $I_P := \text{newIterator}(L_P)$;

7   toFirst(I_P);

       `// get position list for current context`

8   positions := current(I).pl;

9 $I_{pos} := \text{newIterator}(positions)$;

10   toFirst(I_{pos});

11   isOutputNode := **true**;

12   pos := 0;

       `// for all 'true' predicates`

13   **while** *(current(I_P) $\neq$ **null**) $\wedge$ isOutputNode* **do**

         `// eval.  current predicate`

14   **if** *current(I_P)(n, pos)* **then**

           `// increase position`

15   current(I_{pos})++;

16   **else**

17   isOutputNode := **false**;

         `// set context pos.`

18   pos := current(I_{pos});

         `// next predicate`

19   toNext(I_P);

         `// next position index`

20   toNext(I_{pos});

21   **if** *isOutputNode $\wedge$ $\neg$output* **then**

22   output := **true**;

23   toNext(I);

24   **return** *output*;

25 end

This also is the reason for which we cannot evaluate reverse steps in a pipelined manner. Consider two possible strategies for evaluating reverse axes:

1. We can process all nodes in reverse document order. In this way positional predicates evaluate correctly within the presented algorithm because the last output node gets context position 1, the next to last node gets context position 2, and so on. The problem is that we then have to reverse the result sequence in order for it to be in document order, which requires us to wait for the last output node.

2. We could also contruct an algorithm that processes the nodes in document order, but this requires us to know the size of the result sequence to correctly evaluate any positional predicates. Since this size is only known to us after processing all nodes, a second pass over the nodes is required. Obviously, this approach would jeopardize pipelining too.

Chapter 4

Axis Algorithms

Given a *document* and a *context node sequence*, an axis step results in a sequence of those document nodes — in document order and without duplicates — that can be reached by following the axis from one of the context nodes. In this chapter, we will give an algorithm for determining the result sequence for each axis. Note that, in general, one will not get a sequence of nodes in document order and without duplicates by simply computing the axis step for each context node and concatenating the resulting sequences. Hence, a subsequent sorting and duplicate removal step would be necessary to conform to XPath semantics. Unfortunately, this is rather expensive. Therefore, we aim to provide algorithms that have *beneficial time complexity properties*. We assume the input document and context node sequences to be in document order and duplicate free.

4.1 Descendant Axis

4.1.1 Informal Description

As said, one will, in general, not get a sequence of nodes in document order and without duplicates by simply computing an axis step for each context node and concatenating the resulting sequences. An example of such a case is a context node sequence containing nodes n_1 and n_2 where n_2 is a descendant of n_1. Observe that $descendant(n_2) \subseteq descendant(n_1)$. Consequently, concatenating $descendant(n_1)$ and $descendant(n_2)$ in general produces duplicates.

The algorithm given in this section makes use of the fact that document order corresponds with a preorder tree-walk through the XML-tree. Observe that for any node, when you visit that node in a preorder tree-walk, you subsequently visit *all* descendants of that node and then encounter *all* following nodes of that node. This property can be succesfully used in an algorithm that, in one preorder tree-walk, computes the descendants of *an entire sequence* of context nodes without generating duplicate nodes. Furthermore, since a preorder tree-walk corresponds with document order, result nodes are automatically generated in document order as well.

4.1.2 The Algorithm

We now present the function that determines the descendants of a *sequence* of context nodes. The resulting list of nodes is in document order and duplicate-free provided that the input list L_{in} is in document order and duplicate-free.

This algorithm is largely based on the staircase join algorithm of [9]. The function starts with the first context node n (line 4). From there, n' walks in preorder from the first child of n (line 5) to the first following of n (line 6) using the *NextNode* function (line 10). It adds nodes n' to L_{out} along the way (line 7). Since this is a preorder tree-walk, nodes are added in document order.

Since L_{in} is in document order, any context node c that is a descendant of n occurs immediately behind n in L_{in}. We already saw that for any such c, $descendant(c) \subseteq descendant(n)$. Hence, the

---

Algorithm 4: allDescOrd

Input: L_{in}, the input sequence
Output: a boolean indicating whether the current node is retained by the predicates

1 begin
2 L_{out}:= newList();
3   **while** $\neg$ *empty(L_{in})* **do**
4 n := delFirst(L_{in});
5 n' := fc(n);
6   **while** $n' \neq$ **null** $\wedge$ $n' \neq$ ff(n) **do**
7   addAfter(L_{out},n');
8   **if** $n' = $ *first(L_{in})* **then**
9   delFirst(L_{in});
10 n' := NextNode(n');
11 end

---

descendants of c are already in L_{out}. Therefore, if, during our walk through all descendants of n, we encounter that node in L_{in}, it is removed from L_{in} (line 9). This is called *pruning* in [9].

The algorithm keeps L_{out} in document order. Initially, L_{out} is empty and in the first iteration, nodes are added in document order. If it is guaranteed that nodes added in a subsequent iteration are all behind the ones already in L_{out}, we have proven that L_{out} always remains in document order. Observe that, because of pruning, any nodes left in L_{in} after the first iteration are following nodes of n. Therefore, n is assigned in a next iteration a node that is following the previous one. Since all descendants of a node n appear immediately after n, but before following nodes of n, it is guaranteed that descendants of a node n in one iteration appear after those of a previous iteration. This proofs that L_{out} remains in document order.

Finally, L_{out} is the result of the function, L_{out} remains in document order, and we never add nodes to L_{out} that are already there, so the result will be in document order and duplicate-free. Moreover, we visit each node in L_{in} once. Besides that, we never visit a node that is not in the result, nor do we visit a node twice. Therefore, the time complexity of *allDescOrd* is $\mathcal{O}(l + m)$, where l is the size of L_{in} and m the size of L_{out}.

4.1.3 Supporting Positional Predicates

Algorithm 5 describes how to evaluate a `descendant` axis step that is followed by a number of predicates. It is largely based on the staircase join algorithm of [9] as it walks once through the
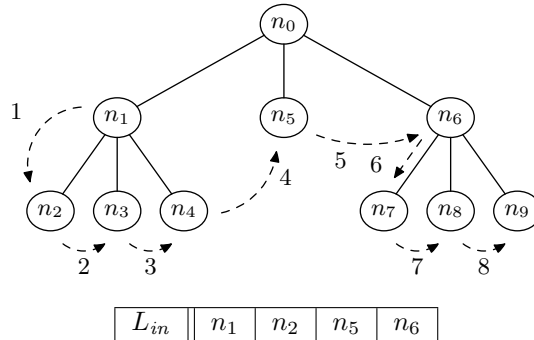
Figure 4.1: The walk described by the descendant algorithm for the input list above. When a node of L_{in} is encountered in the document, it registered as a context on L_{st}.

14

document (see Figure 4.1) in document order to determine all descendants of an entire context node sequence L_{in}. It is required that L_{in} is in document order.

Algorithm 5: descendantPredicates

Input:

- L_{in}: the list of input nodes,

- L_P: the list of predicates

Output: L_{out}: the list of output nodes

1 begin
2 $L_{out} :=$ newList();
3 $L_{st} :=$ newList();
4   **while** $\neg \; empty(L_{in})$ **do**
       `// fetch first node n from input`
5 $n :=$ delFirst(L_{in});
       `// push n on the context stack`
6   addBefore(L_{st},
7 $[cn := n, \; pl :=$createPosList(L_P)]);
       `// fetch first child n' of n`
8 $n' :=$ fc(n);
9   **while** $(n' \neq \textbf{null}) \wedge (\neg \; empty(L_{st}))$ **do**
10   **if** $evaluatePredicates(L_{st}, \textbf{null}, n', L_P)$ **then**
11   AddAfter(L_{out}, n');
12   **if** $n' = first(L_{in})$ **then**
            `// register context node`
13   addBefore(L_{st}, [$cn :=$delFirst(L_{in}),
14 $pl :=$createPosList(L_P)]);
         `// get follower of n'`
15 $n' :=$ nextNode(n');
16   **while** $(\neg \; empty(L_{st})) \wedge$
17   ($post(n') > post(first(L_{st}).cn)$ **do**
            `// remove obsolete context from` L_{st}
18   delFirst(L_{st});
19   **return** L_{out}
20 end

As is explained in Chapter 3, there is a problem with pruning nodes from the input list as it is done in Algorithm 4. Instead of just pruning the nodes from the input list, we will keep these context nodes on a context stack, along with the information required for the correct evaluation of multiple positional predicates.

The outer while loop of line 4 iterates over the context nodes in the input list using variable n. The while loop of line 9 scans the document using variable n' from the first child of n (line 8), taking each time the next node in document order (line 15) until the last descendant. The last descendant has been reached if we reached the end of the document ($n' = \textbf{null}$) or we reached a node that is not a descendant of any context node so far apparent from the fact that the context stack is empty (line 9).

During processing, the context stack L_{st} contains information on all context nodes n for which n' is a descendant, together with the relative position of n' w.r.t. n for each predicate. If, during the scan, we reach a document node that is also a context node (line 12), the next couple of nodes will also be descendants of this context node. Therefore, the context node is pushed on the stack and removed from L_{in}. A context node is removed from the stack when subsequent document

nodes n' are no longer descendants of it (line 16), which is the case when the next n' appears to be a following node of the context node. The fact that we can push and pop context nodes on the stack while scanning document nodes relies on two important properties: (1) in a pre-order walk from a certain node, one first encounters all descendants of a node and then all following nodes, and (2) the input list is in document order, so we only need to check the first node of L_{in}.

A document node n' is a result node if it passes all predicates in L_P (line 10). *evaluatePredicates* updates the positions in L_{st} according to which predicates where satisfied.

The algorithm keeps L_{out} in document order. Initially, L_{out} is empty and in the first iteration of n, nodes are only added at the end in line 11. Since n' scans the document in document order, L_{out} remains in document order. If it is guaranteed that nodes added in a subsequent iteration are all behind the ones already in L_{out}, we have proven that L_{out} always remains in document order. Since L_{in} is in document order, L_{in} will first contain all context nodes that are descendants of n, and then all following nodes of n. The first are removed from L_{in} in line 12. Therefore, upon the next iteration of the outer while loop of line 4, the next node will be a following node of n. Since all descendants of following nodes of n, are guaranteed to be following nodes of n themselves, subsequent result nodes are behind the ones already in L_{out}. This proves that L_{out} remains in document order.

Finally, L_{out} remains in document order, and we never add nodes to L_{out} that are already there, so the result will be in document order and duplicate-free. Moreover, we visit each node in L_{in} once. Besides that, we never visit a node that is not a candidate result node, nor do we visit a node twice. Therefore, the data access complexity of *descendantPredicates* is $\mathcal{O}(l+m)$, where l is the size of L_{in} and m the number of descendants of L_{in}. For each result node, we potentially have to evaluate each predicate for each context (node,position)–combination. Since a node can have at most h ancestors, where h denotes the height of the XML document tree, predicate evaluation complexity is $\mathcal{O}(h \times m \times |L_P|)$.

4.2 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. If support for positional predicates is required, the previously given algorithm can easily be adjusted binding n' to the input node instead of its first child.

4.3 Ancestor Axis

4.3.1 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.

4.3.2 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.

Function *addAnc(L, n)*

---

begin
> $n' := \mathsf{pa}(n)$;
> **if** $n' \neq$ **null then**
> > addAnc(L,n');
> > addAfter(L, n');

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.

Function *addAncUntilLeft(L, n, n')*

---

begin
> $n'' := \mathsf{pa}(n)$;
> **if** $n'' \neq$ **null** $\wedge$ $\mathsf{pre}(n'') \geq \mathsf{pre}(n')$ **then**
> > addAncUntilLeft(L, n'', n');
> > addAfter(L, n'');

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 $\mathcal{O}(m)$ 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.

Algorithm 8: allAncOrd(L_{in})

---

1 begin
2 | $L_{out} :=$ newList();
3 | **if** $\neg empty(L_{in})$ **then**
4 | | $n :=$ delFirst(L_{in});
5 | | addAnc(L_{out}, n);
6 | **while** $\neg empty(L_{in})$ **do**
7 | | $n' :=$ delFirst(L_{in}); addAncUntilLeft(L_{out}, n' , n);
8 | | $n := n'$
9 | **return** L_{out};
10 end

In the first part of the algorithm (line 3-5) all ancestors of the first node in L_{in} are retrieved. After this a while loop (line 6-8) 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 $\mathcal{O}(l + m)$ if l is the size of L_{in} and m is the size of the result.

Algorithm 9: ancestorPredicates

Input:

- L_{in}: the list of input nodes,

- L_P: the list of predicates

Output: L_{out}: the list of output nodes

1 begin

2 $L_{out} :=$ newList();

3 $L_{st} :=$ newList();

4 $I :=$ newIterator(L_{st});

       `// reverse axis, start at end`

5 $n :=$ last(L_{in});

6   **while** $n \neq$ **null do**

          `// if` L_{st} `is empty, nothing happens`

7   **if** *evaluatePredicates(L_{st}, I, n, L_P)* **then**

8   AddAfter(L_{out}, n);

          `// if current node is input node`

9   **if** $\neg$ *empty(L_{in})* $\wedge$ *n = last(L_{in})* **then**

          `// ... register as a context`

10   addBefore(L_{st}, [$cn :=$ delLast(L_{in}),

          $pl :=$ createPosList(L_P)]);

11   **if** pa*(n)* $\neq$ **null then**

12   **if** $\neg$ *empty(L_{in})* $\wedge$ pre*(*pa*(n))* $<$ pre*(last(L_{in}))* **then**

             `// process prec's before anc's`

13 $n :=$ delLast(L_{in});

14   **else**

             `// continue with anc's`

15 $n :=$ pa(n);

16   addBefore(L_{st}, [$cn :=$ delLast(L_{in}),

             $pl :=$ createPosList(L_P)]);

17 $n :=$ pa(n);

18   toFirst(I);

19   **while** *current(I)* $\neq$ **null** $\wedge$ post*((current(I)).cn)* $<$ post*(n)* **do**

             `// locate` *active* `contexts`

20   toNext(I);

21   **else**

22 $n :=$ **null**;

23   **return** *reverse(L_{out})*;

24 end
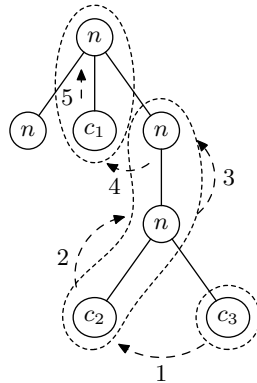
Figure 4.2: Nodes scanned by ancestor algorithm (context nodes processed in reverse document order c_3, c_2, c_1).

4.3.3 Supporting Positional Predicates

The ancestor algorithm (see Algorithm 9) uses the $\mathsf{pa}(n)$ pointers to walk the variable n from a context node up in the document to the root (line 15). It does this for each context node in L_{in} in reverse order (lines 5, 10, and 13). To avoid accessing and producing duplicate nodes, it stops whenever it encounters a node that is on a subsequent context node's path to the root (line 12). For example, Figure 4.2 illustrates the nodes accessed for context nodes c_1, c_2, and c_3.

The stack L_{st} maintains a list of encountered context nodes. At the end of each iteration, the while loop of line 19 moves the iterator I to the last context node on the stack for which n is an ancestor. The context nodes on the stack up to this point are called *active context nodes*. The *evaluatePredicates* (line 7) of a subsequent iteration will only check the predicates for the active context nodes. Note that context nodes are never removed from the stack once they end up on it. This is easily explained by noticing that the root node will be processed last. Since the root node is a candidate output node for all context nodes (except for itself), it requires all these nodes to be present on the stack.

Since we walk backwards through the tree, the result nodes are produced in reverse document order. Therefore, we have to reverse the output list at the end.

Let again l be the length of the input list en m the number of ancestors of L_{in}. Note that all context nodes are accessed once and moved from L_{in} to the stack. Furthermore, Figure 4.2 illustrates that our navigation strategy accesses each parent only once. Unfortunately, determining the active context nodes on the stack (line 19) accesses in the worst case all context nodes on the stack (max. l), and this happens for all m iterations. Finally, all output nodes (max. m) are accessed again in reversing L_{out}. Therefore, the data access complexity is $\mathcal{O}(l + 2m + l \times m)$. In all m iterations, the predicates are evaluated for all active context nodes (max. l), so the predicate evaluation complexity is $\mathcal{O}(l \times m \times \mid L_P \mid)$.

4.4 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. For supporting positional predicates, Algorithm 9 can be easily adapted. The only thing that needs to be done is postponing predicate evaluation (Line 7) until the current node is registered as a context (Lines 9 to 10).

4.5   Child Axis

4.5.1   Informal Description

We cannot use the approach of the previous axes here. Consider for example the fragment in Figure 4.3. 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 4.3: An XML fragment

4.5.2   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.

Function *allChildren(n)*

---

   begin
      $L := \text{newList}()$;
      $n' := \mathsf{fc}(n)$;
      while $n' \neq$ **null do**
         $\text{addAfter}(L, n')$;
         $n' := \mathsf{ns}(n')$;
      return L;
   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 $\mathcal{O}(m)$ 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.

The algorithm consists of two while loops. The first (line 4-18) 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 19-24) 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

---

Algorithm 11: allChildOrd

Input: L_{in}: the list of input nodes
Output: L_{out}: the list of output nodes

1 begin

2 L_{out}:= newList();

3 L_{st}:= newList();

4   **while** $\neg empty(L_{in})$ **do**

5 n := first(L_{in});

6   **if** $empty(L_{st})$ **then**

7 L' := allChildren(n);

8   addBefore(L_{st}, L');

9   delFirst(L_{in});

10   **else if** $empty(first(L_{st}))$ **then**

11   delFirst(L_{st});

12   **else if** $\mathsf{pre}(first(first(L_{st})))$ ¿ $\mathsf{pre}(n)$ **then**

13 L' := allChildren(n);

14   addBefore(L_{st}, L');

15   delFirst(L_{in});

16   **else**

17 n' := delFirst(first(L_{st}));

18   addAfter(L_{out}, n');

19   **while** $\neg empty(L_{st})$ **do**

20   **if** $empty(first(L_{st}))$ **then**

21   delFirst(L_{st});

22   **else**

23 n' := delFirst(first(L_{st}));

24   addAfter(L_{out}, n');

25   **return** L_{out};

26 end

---

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 $\mathcal{O}(l + m)$ where l is the size of L_{in} and m the size of L_{out}.

4.5.3 Supporting Positional Predicates

Adjusting the algorithm for supporting positional predicates in this case resulted in a completely different algorithm. Instead of keeping lists of unprocessed series of children on a stack, this approach only pushes the context node on the stack, along with the position of the first unprocessed child and a reference to the child itself. Since the descendants need to be processed before the the following nodes, we have to use the following strategy: If a candidate output node is a context node itself or has a descendant node that is a context node, then this context is pushed on the context stack before the next siblings of the candidate node are processed.

This time, the context stack consists of a three-tuple. It contains the context node (the parent), the current child node (meaning the first node of this context that still needs processing) and the position list of this current child node. Starting by the usual looping over input list, the algorithm pushes the context on the stack and initializes the current node n as being the first child node of the context. At rule 13, the predicates are evaluated and if they all evaluate to `true`, the current node is sent to the output. At the next rule, the current node is set to be the next sibling node. If there are no more siblings, the context is removed from the stack (rule 16). The conditional contsruction at rule 18 checks whether there are input nodes that are descendants of the current node, which need to be processed before continuing with the next sibling node. If this is the case, these input nodes are pushed on the context stack and the children are processed.

Correctness L_{out} contains only the child nodes of the input nodes. Only on Lines 8 and 19, n is bound to an input node that may not be an output node, but it is set to the first child before n is sent to L_{out}. The algorithm only iterate from the first child of each input node to the last child with the ns-operator. Since the input list L_{in} contains the nodes in document order, the contexts are also handled in document order. Thus, the contexts will appear on the context stack L_{st} in document order. Due to the fact that contexts that are completely processed are removed from the stack (Line 16), it also holds that every two contexts on the stack have an ancestor-descendant relationship. Since the algorithm also guarantees that the current node n is a child node of the top of the stack, n will always be a descendant of all context nodes in L_{st}. The fact that all descendant nodes of the current node are processed before the following nodes (see conditional on Line 18), ensures that the output nodes appear in document order on L_{out}. Further, no node is ever outputted more than once, since the document and the input list are scanned simultaniously.

Data Access Complexity The outer while loop accesses both the nodes in the input list (size l) and the nodes in the output list (size m) exactly once. Additionally, the semantics of child axis implies that every output node has at most one context node. As a result, for evaluating the predicates for the each of candidate output nodes, only $|L_P|$ accesses are required. Thus, the data access complexity is $\mathcal{O}(l + m \times |L_P|)$.

4.6 Parent Axis

4.6.1 Informal Description

The fundamental property that will be used for this axis is the following. If we retrieve the parent nodes of a duplicate-free sorted list of document nodes, we obtain a sublist of the list of nodes

---
Algorithm 12: childPredicates

Input:

- L_{in}: the list of input nodes

- L_P: predicate list

Output: L_{out}: the list of output nodes

1 begin

2 L_{out}:= newList();

3 L_{st}:= newList();

4 I := newIterator(L_{st});

5   toFirst(I);

6   toNext(I);

7   **while** $\neg\ empty(L_{in})$ **do**

      `// fetch first input node`

8 n := delFirst(L_{in});

      `// if child node present`

9   **if** fc$(n) \neq$ **null then**

        `// ...  add node as context`

10   addBefore(L_{st}, [$cn := n$, $fc := $ fc(n), $pl := $ createPosList(L_P)]);

      `// get first child`

11 $n := $ fc(n);

12   **while** $\neg\ empty(L_{st})$ **do**

        `// use only top of stack`

13   **if** $evaluatePredicates(L_{st},\ I,\ n,\ L_P)$ **then**

14   addAfter(L_{out}, n);

        `// register next child as current node`

15   first(L_{st}).fc := ns(first(L_{st}).fc);

        `// delete processed contexts`

16   **if** $first(L_{st}).fc = $ **null then**

17   delFirst(L_{st});

18   **if** $\neg\ empty(L_{st})\ \wedge$ pre$(first(L_{st}).fc) > $ pre$(first(L_{in}))$ **then**

        `// handle desc's of current node first`

19 $n := $ delFirst(L_{in});

20   **if** (fc$(n) \neq$ **null**) **then**

21   addBefore(L_{st}, [$cn := n$, $fc = $ fc(n), $pl := $ createPosList(L_P)]);

        `// get next sibling to be processed`

22 $n := $ first(L_{st}).fc;

23   toFirst(I);

24   toNext(I);

25   **return** L_{out};

26 end

23

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 4.3 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 $[[1], [2], [2, [4, [5]], [8, [9]]]]$.

The example demonstates that 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.

4.6.2   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.

Function *dupElimSort(L)*

---

begin

  L_{out} := newList();

  if ¬*empty(L)* **then**

  | n := delFirst(L);

  while ¬*empty(L)* **do**

  | n' := delFirst(L);

  | **if** $n' \neq n$ **then**

  | | addAfter(L_{out}, n');

  | | $n := n'$;

  return L_{out};

end

The time complexity of this function is clearly $\mathcal{O}(l)$ 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.

Function *addFlatList(L, L_{tr})*

---

begin

    while ¬*empty(L_{tr})* **do**

        n := delFirst(L_{tr});

        if *isList(n)* **then**

           addFlatList(L,L_{tr});

        else

           addAfter(L,n);

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 $\mathcal{O}(m)$ 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.

Algorithm 15: allParOrd

    Input: L_{in}: the list of input nodes

    Output: L_{out}: the list of output nodes

1 **begin**

2 L_{tr} := newList();

3   **while** ¬*empty(L_{in})* **do**

4 n := pa(delFirst(L_{in})); L := newList(); **while** ¬*empty(L_{tr})* ∧ pre*(first(last(L_{tr})))*

           ¿ pre*(n)* **do**

5 n' := delLast(L_{tr});

6   addBefore(L, n');

7   addBefore(L, n); addAfter(L_{tr}, L)

8 L_{dup} := newList();

9   addFlatList(L_{dup}, L_{tr});

10 L_{out}:= dupElimSort(L_{dup});

11   **return** L_{out};

12 **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 4-6 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 7 this tail is extended with n and finally on line 7 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 $\mathcal{O}(l)$ 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 4. 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.

4.6.3 Supporting Positional Predicates

This is the only axis for which we do not need to construct an alternative algorithm in order to support the correct evaluation of positional predicates. Note that every node has at most one parent. This implies that it is sufficient in the prvious algorithm to check whether the positional predicates are equal to one. Any other position is impossible and results in no output at all.

4.7 Following Axis

4.7.1 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.

4.7.2 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.

Function $addFoll(L, n)$

> **begin**
>> **if** $ff(n) \neq$ **null then**
>>> addAfter(L, $ff(n)$);
>>> addDesc(L, $ff(n)$);
>>> addFoll(L, $ff(n)$);
>
> **end**

The correctness of this procedure follows from the fact that $ff(n)$ 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 $\mathcal{O}(m)$ 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.

Algorithm 17: allFollOrd

> **Input**: L_{in}: the list of input nodes
> **Output**: L_{out}: the list of output nodes
> **begin**
>> L_{out}:= newList(); **if** $\neg empty(L_{in})$ **then**
>>> n := delFirst(L_{in});
>>> **while** $\neg empty(L_{in}) \wedge$ pre$(first(L_{in})) >$ pre$(n) \wedge$ post$(first(L_{in})) <$ post(n) **do**
>>>> n := delFirst(L_{in});
>>> addFoll(L_{out}, n);
>> **return** L_{out};
>
> **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 $\mathcal{O}(l + m)$ if l is the size of L_{in} and m the size of L_{out}.

4.7.3 Supporting Positional Predicates

Another complication arises when we want to efficiently support positional predicates for the following axis. Looking back to our small example in Figure 3.1, we can observe that if we process

our list of input nodes in document order, we encounter node n_2 before node n_3. Also, the set of following nodes of n_2 will always be a proper subset of the set of following nodes of n_3. Apparently, n_2 will become *active* as a context node *after* n_3 has been processed. This is why the so called *inactive* context nodes are buffered on a secondary stack until they become active.

Algorithm 18: followingPredicates

Input:

- L_{in}: the list of input nodes

- L_P: predicate list

Output: L_{out}: the list of output nodes
begin

    L_{out}:= newList(); `// context nodes for which we are handling descendants`
    L_{st_1} := newList();
    `// active context nodes`
    L_{st_2} := newList();
    $n :=$ **null**;

    while *(nextNode(n) $\neq$* **null***) $\vee$ $\neg$empty(L_{in})* **do**

        if $n =$ **null** **then**
            `// initialize n with node from input L`$_{in}$
            $n :=$ first(L_{in});

        `// current node = input node`
        if *$\neg$empty(L_{in}) $\wedge$ (n = first(L_{in}))* **then**
            `// add inactive context to L`$_{st_1}$
            addBefore(L_{st_1}, [$cn :=$ delFirst(L_{in})]);
            while $\neg$empty(L_{in}) $\wedge$ (post(*first(L_{in})*) < post(*first(L_{st_1}).cn*)) **do**
                `// buffer descendants in L`$_{st_1}$
                addBefore(L_{st_1}, delFirst(L_{in}));

        if *empty(L_{st_2})* **then**
            $n :=$ ff(first(L_{st_1}).cn);
        else
            $n :=$ nextNode(n);

        `// if n follows an inactive context`
        while *ff(first(L_{st_1}).cn) = n* **do**
            `// ... activate the context`
            addBefore(L_{st_2}, [$cn :=$ delFirst(L_{st_1}), $pl :=$ createPosList(L_P)]);

        if *evaluatePredicates(L_{st},* **null***, n, L_P)* **then**
            addAfter(L_{out}, n);

    return L_{out};
end

As stated before, we will need both a buffer L_{st_1} (Line 18) and a context stack L_{st_2} (Line 18). The `while`-loop on Line 18 runs until the input list L_{in} runs empty and the last following node of the last input node has been processed. On Line 18 n is initialized with the first input node if it is found to be `null`. This can be because the algorithm is just starting or if n is assinged a following node of itself, while no such node exists (this can happen on Lines 18 and 18). On Line 18, if the current node is the same as the first node on the input, this node is taken from the input list and registered as an inactive context node. Additionally, in the `while`-loop on Line 18, all descendants of this input node are also buffered (since we know they will only become active as a context after the last descendant input node).

On Line 18, if the context stack is empty (i.e., when the algorithm bootstraps), n is assigned the first following node of the last (deepest descendant) of the input nodes that were buffered. This will be the first candidate output node. In case there are one or more active contexts in L_{st_2}, all following nodes are candidate input nodes, and thus n is assigned the next node in document order (Line 18).

In the `while`-loop of Line 18, before evaluation of the predicates, all inactive contexts for which n is a follower node, are activated by moving them onto the context stack L_{st_2}. Since n can only be their first follower, their associated positions are set to 1. Next, on Line 18 the predicates are evaluated.

Correctness n is a follower node of the nodes on the context stack L_{st_2}. This property is enforced by copying context nodes from the buffer to context stack only if n indeed is a follower of these contexts (Line 18). The only exception occurs in the first iteration, where L_{st_1} is empty. In this case, n gets initialized properly on Line 18.

All contexts that are before n in document order, are either to L_{st_1} or L_{st_2}. Upon predicate evaluation, n is a descendant of all buffered contexts in L_{st_1}; i.e., there are no inactive contexts for which n is a follower. This is guaranteed by the fact that all inactive contexts on L_{st_1} are activated if n is a follower node. So, all valid contexts of n are on L_{st_2}, which is used for evaluating the predicates. Because every candidate output node is visited, L_{out} contains valid output nodes.

The variable n is used to scan the document in document order by using the *nextNode* function. After the first iteration, n is bound to the first candidate output node on Line 18. Afterwards, subsequent iterations move n to the next node in document order on Line 18. Since each iteration, n is considered only once as a candidate output node, the result is without duplicates and in document order.

Data Access Complexity   The outer while loop visits both the input nodes in L_{st} and the candidate output nodes exactly once. For each of the condidate output nodes the predicates are evaluated. In the worst case, the context stack contains all of the input nodes (size l) and thus the evaluation requires $m \times |L_P|$ time for each node. Thus, the resulting complexity is $\mathcal{O}(m \times l \times |L_P|)$. The copying of the contexts between the two stacks only add a constant factor, i.e. $\mathcal{O}(m \times (2l + l \times |L_P|))$, since the size of the copied stack in both while loops can be at most l.

4.8   Preceding Axis

4.8.1   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 lp 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.

4.8.2   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''.

Function $addAncBetween(L, n, n', n'')$

begin
> $n''' := \mathsf{pa}(n)$;
> **if** $n''' \neq \mathbf{null} \wedge (\mathsf{pre}(n''') \geq \mathsf{pre}(n')) \wedge (\mathsf{post}(n''') \leq \mathsf{post}(n''))$ **then**
>> addAncUntil(L, n''', n');
>> addAfter(L, n''');

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 $\mathcal{O}(m)$ 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.

Function $addAncUntilRight(L, n, n')$

begin
> $n'' := \mathsf{pa}(n)$;
> **if** $n'' \neq \mathbf{null} \wedge \mathsf{post}(n'') \leq \mathsf{post}(n')$ **then**
>> addAncUntilRight(L, n'', n');
>> addAfter(L, n'');

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 $\mathcal{O}(m)$ 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.

Function $addLeftUntil(L, n, n')$

begin
> **if** $\mathsf{lp}(n) \neq \mathbf{null}$ **then**
>> $n'' := \mathsf{lp}(n)$;
>> addLeftUntil(L, n'', n');
>> addAncBetween(L, n, n'', n');
>
> **else**
>> addAncUntilRight(L, n, n');
>
> addAfter(L, n);

end

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

Algorithm 22: allPrecOrd

  Input: L_{in}: the list of input nodes
  Output: L_{out}: the list of output nodes
  begin
    | L_{out}:= newList();
    | **if** $\neg empty(L_{in})$ **then**
    | | $n :=$ last(L_{in});
    | | **if** $\mathsf{lp}(n) \neq$ **null then**
    | | | addLeftUntil(L_{out}, $\mathsf{lp}(n)$, n);
    | **return** L_{out};
  end

4.8.3 Supporting Positional Predicates

This algorithm is basically a copy of the one for the following axis where the input list and candidate output nodes are processed backwards. This approach results in an output list that contains nodes in reversed document order, so as a final step we need to reverse this list.

4.9 Following-Sibling Axis

4.9.1 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 moved 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.

4.9.2 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.

Function *allFollSibl(n)*

  begin
    | $L :=$ newList();
    | $n':= \mathsf{ns}(n)$;
    | **while** $n' \neq$ **null do**
    | | addAfter(L, n');
    | | $n' := \mathsf{ns}(n')$;
    | **return** L;
  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 $\mathcal{O}(m)$ 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.

The correctness of this function is similar to that of the corresponding function for the child axis. The main difference is in line 25 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

---
Algorithm 23: precedingPredicates

Input:

- L_{in}: the list of input nodes

- L_P: predicate list

Output: L_{out}: the list of output nodes
begin

    L_{out} := newList();
    // context nodes for which we are handling descendants
    L_{st_1} := newList();
    // context stack
    L_{st_2} := newList();
    n := **null**;

    while *(prevNode(n)* ≠ **null***)* ∨ ¬*empty(L_{in})* **do**
        if n = **null** **then**
            // initialize n with last input node (reverse axis)
            n := last(L_{in});

        if ¬*empty(L_{in})* ∧ *(n = last(L_{in}))* **then**
            // buffer as inactive context
            addBefore(L_{st_1}, [cn := delLast(L_{in})]);

            while ¬*empty(L_{in})* ∧ *(*post*(last(L_{in}))* > post*(first(L_{st_1}).cn))* **do**
                // buffer ancestor input nodes
                addBefore(L_{st_1}, delLast(L_{in}));

        // if this is the first iteration
        if *empty(L_{st_2})* **then**
            // jump to the last preceding node
            n := lp(first(L_{st_1}).cn);
        else
            // jump to previous node in doc-order
            n := prevNode(n);
        while lp*(first(L_{st_1}).cn)* = n **do**
            // activate contexts
            addBefore(L_{st_2}, [cn := delFirst(L_{st_1}), pl := createPosList(L_P)]);

        if *evaluatePredicates(L_{st}, **null**, n, L_P)* **then**
            addAfter(L_{out}, n);

    return reverse(L_{out});
end

Algorithm 25: allFollSiblOrd

Input: L_{in}: the list of input nodes
Output: L_{out}: the list of output nodes
begin

    L_{out} := newList();

    L_{st} := newList();

    while $\neg empty(L_{in})$ **do**

        n := first(L_{in});

        if $empty(L_{st})$ **then**

            L' := allFollSibl(n);

            addBefore(L_{st}, L');

            delFirst(L_{in});

        else if $empty(first(L_{st}))$ **then**

            delFirst(L_{st});

        else if $\mathsf{pre}(first(first(L_{st}))) > \mathsf{pre}(n)$ **then**

            L' := allFollSibl(n);

            addBefore(L_{st}, L');

            delFirst(L_{in});

        else if $\mathsf{pre}(first(first(L_{st}))) < \mathsf{pre}(n)$ **then**

            n' := delFirst(first(L_{st}));

            addAfter(L_{out}, n');

        else

            delFirst(L_{in});

    while $\neg empty(L_{st})$ **do**

        if $empty(first(L_{st}))$ **then**

            delFirst(L_{st});

        else

            n' := delFirst(first(L_{st})); addAfter(L_{out}, n');
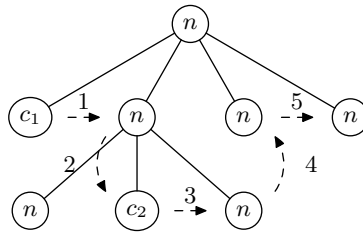
    return L_{out};

end

Figure 4.4: Document scan by followingSibling algorithm.

Algorithm 26: insertNextEntries

> **Input**:
>
> - L_{in}: the list of input nodes,
>
> - L_{st}: the context stack
>
> ```
> // adds contexts that are desc. of current node
> ```
> **begin**
> > **while** $\neg$ *empty(L_{in})* $\wedge$ pre*(first(L_{in}))* $<$ pre*(first(L_{st}).ns)* **do**
> > > n := delFirst(L_{in});
> > > **if** ns*(n)* $\neq$ **null then**
> > > > addBefore(L_{st}, [cn:=n,ns:=ns(n),
> > > > pl:=createPosList(L_P)]);
>
> **end**

---

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., $\mathcal{O}(l+m)$ where l is the size of L_{in} and m is the size of L_{out}.

The following-sibling algorithm navigates through the document starting from a context node and iteratively following the ns-pointer to the next sibling. To ensure that nodes are produced in document order, this navigation is interrupted if a context node on the input list happens to be a descendant of a previously processed sibling. In that case, the descendant's siblings are processed first as illustrated in Figure 4.4. Let us call a context node whose processing has been interrupted, an *inactive* context node. All active and inactive context nodes are kept on the stack together with the sibling node that is next in line. All predicates are evaluated for all active nodes on the stack. There are two cases when a new node is put on the stack:

1. If a descendant node is found in the input list, all nodes on the stack become inactive and the node from the input list is put on the stack (it becomes active).
2. If, while processing a sibling node, we also find it on the input list, the new context node is also put on the stack. Since both are active, a subsequent predicate evaluation will consider both.

The algorithm for following-sibling uses a helper function *insertNextEntries* (see Algorithm 26) that moves those context nodes from L_{in} onto the stack L_{st} that are descendants or preceding nodes of the topmost node on the stack. This condition can simply be checked by looking at the pre-order ranks.

The algorithm for following-sibling is presented in Algorithm 27. The outer while loop (line 27) puts a node from the input list on the stack whenever the stack becomes empty. The main loop is the while loop of line 27. Each iteration corresponds to navigating to the next following sibling (line 27) and processing it.

Lines 27 through 27 determine the active context nodes on the stack by moving iterator I to the first inactive context node. Only those context nodes are active that have a sibling currently in line that is equal to n. The subsequent predicate evaluation of line 27 will evaluate all predicates

Algorithm 27: followingSiblingPredicates

Input:

- L_{in}: the list of input nodes,

- L_P: the list of predicates

Output: L_{out}: the list of output nodes

begin

    L_{out}:= newList();

    L_{st}:= newList();

    // pointers into the stack

    I := newIterator(L_{st});

    I' := newIterator(L_{st});

    while $\neg\ empty(L_{in})$ **do**

        // fetch first input node

        n := delFirst(L_{in});

        // activation as context only if necessary

        if ns*(n)* $\neq$ **null** **then**

            addBefore(L_{st}, [cn:=n, ns:=ns(n),

            pl:=createPosList(L_P)]);

            // process any desc's before foll. sib's

            insertNextEntries(L_{in}, L_{st});

        while $\neg\ empty(L_{st})$ **do**

            // get next sibling of current context

            n := first(L_{st}).ns;

            toFirst(I);

            // move to last active context

            while $current(I) \neq$ **null** $\wedge$ $current(I).ns{=}n$ **do**

                toNext(I);

            // use contexts in active part

            if $evaluatePredicates(L_{st}, I, n, L_P)$ **then**

                addAfter(L_{out}, n);

            toFirst(I');

            while $I \neq I'$ **do**

                if ns*(n)* $=$ **null** **then**

                    // remove obsolete contexts

                    delFirst(L_{st});

                else

                    // get next sibling for current context

                    current(I').ns = ns(n);

                toNext(I');

            // process any desc's before foll. sib's

            insertNextEntries(L_{in}, L_{st});

    return L_{out};

end

for the active context nodes. If some predicate evaluates to true, the current node n is put on the output list.

Lines 27 through 27 update the stack by moving the sibling currently in line (the ns column on the stack) to the next one. Context nodes for which there is no next sibling are removed from the stack. The call to *insertNextEntries* in line 27 makes sure that context nodes of the input list are moved to stack when they need to become active (*i.e.*, they are a descendant of preceding node of the next node to be processed).

Concerning data access complexity, let l be the length of L_{in}, m the number of following siblings of L_{in}, s the maximum number of children a node can have, and h the height of the tree. Note that the stack will contain active and inactive context nodes. All active context nodes are in sibling relation to each other, so a maximum of s active context nodes. Inactive context nodes were those that were interrupted. For each level of the tree, we have again a maximum of s inactive context nodes. Hence, the stack contains a maximum of $h \times s$ nodes.

The two main while loops (lines 27 and 27) interleave each other and correspond to navigating through the document. Therefore, there are m iterations. Determining the active nodes on the stack (lines 27 through 27) touches maximally s nodes on the stack. Updating the stack in lines 27 through 27 also touches only active context nodes on the stack. Consequently, stack access complexity of $\mathcal{O}(2m \times s)$.

Finally, all context nodes are again moved once from L_{in} to the stack and all nodes are also removed from the stack. Therefore, data access complexity is $\mathcal{O}(2m \times s + 2l + m)$. Predicate evaluation complexity is dependent on the maximum number of active context nodes: $\mathcal{O}(s \times m \times |L_P|)$

4.10   Preceding-Sibling Axis

4.10.1   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 document order and we move nodes from the back of the lists on the stack to the front of the output list.

4.10.2   The Algorithm

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

Function *allPrecSibl(n)*

---
   begin
     L := newList();
     n':= $\mathsf{ps}(n)$;
     while $n' \neq$ **null do**
       addBefore(L, n');
       n' := $\mathsf{ps}(n')$;
     return L;
   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 $\mathcal{O}(m)$ 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.

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

Algorithm 29: allPrecSiblOrd(L_{in})

Input: L_{in}: the list of input nodes
Output: L_{out}: the list of output nodes
begin

    L_{out}:= newList();
    L_{st}:= newList();
    while $\neg empty(L_{in})$ **do**
        $n :=$ last(L_{in});
        if $empty(L_{st})$ **then**
            $L' :=$ allPrecSibl(n);
            addBefore(L_{st}, L');
            delLast(L_{in});
        else if $empty(first(L_{st}))$ **then**
            delFirst(L_{st});
        else if $\mathsf{pre}(last(first(L_{st}))) > \mathsf{pre}(n)$ **then**
            $L' :=$ allFollSibl(n);
            addBefore(L_{st}, L');
            delLast(L_{in});
        else if $\mathsf{pre}(last(first(L_{st}))) < \mathsf{pre}(n)$ **then**
            $n' :=$ delLast(first(L_{st}));
            addBefore(L_{out}, n');
        else
            delFirst(L_{in});

    while $\neg empty(L_{st})$ **do**
        if $empty(first(L_{st}))$ **then**
            delFirst(L_{st});
        else
            $n' :=$ delLast(first(L_{st}));
            addBefore(L_{out}, n');

    return L_{out};
end

4.10.3 Supporting Positional Predicates

Function *insertPrevEntries(L_{in}, L_{st})*

---
begin
    while $\neg empty(L_{in})$ $\wedge$ *(pre(last(L_{in}))* > pre*(first(L_{st}).ps)* **do**
        $n :=$ delLast(L_{in});
        if ps*(n)* $\neq$ **null then**
            addBefore(L_{st}, [$cn := n$, $ps :=$ ps(n), $pl :=$ createPosList(L_P)]);
end

Algorithm 31: PrcedingSiblingPredicates

Input: L_{in}: the list of input nodes
Output: L_{out}: the list of output nodes
begin
    $L_{out} :=$ newList();
    $L_{st} :=$ newList();
    $I :=$ newIterator(L_{st});
    $I' :=$ newIterator(L_{st});

    while $\neg empty(L_{in})$ **do**
        // reverse axis, start at the end
        n := delLast(L_{in});
        // if prec. sibling of n exists
        if ps*(n)* $\neq$ **null then**
            // ... register n as context
            addBefore(L_{st}, [$cn := n$, $ps :=$ ps(n), $pl :=$ createPosList(L_P)]);
            // handle desc's before prec. sib's
            insertPrevEntries(L_{in}, L_{st});

        while $\neg empty(L_{st})$ **do**
            // get the prec. sib. of current context
            $n :=$ first(L_{st}).ns;
            toFirst(I);
            while *(current(I)* $\neq$ **null***)* $\wedge$ *(current(I).ps = n)* **do**
                // locate active contexts
                toNext(I);

            // use only active contexts
            if *evaluatePredicates(L_{st}, I, n, L_P)* **then**
                addAfter(L_{out}, n);
            toFirst(I');
            while $I \neq I'$ **do**
                if ps*(n)* $=$ **null then**
                    // remove processed contexts
                    delFirst(L_{st});
                else
                    // get prec. sib. of active contexts
                    current(I').ps $=$ ps(n);
                toNext(I');

            // process desc's before sib's
            insertPrevEntries(L_{in}, L_{st});

    return reverse(L_{out});
end

Chapter 5

Conclusion

5.1 Complexity Bounds

The previous chapter contained complexity results for the specific axis algorithms. We will now discuss more generally the complexity bounds of our XPath evaluation strategy. In what follows, we express the size of the document as $|D|$, and the size of the query as $|Q|$. Furthermore, we assume that predicates can be evaluated in constant time.

Memory Complexity For the axes `child`, `parent`, `descendant`, `ancestor`, `desc-or-self` and `anc-or-self`, the algorithm's memory requirements are bound by the height of the document tree, which is known to be fairly small in many cases. In general, the memory requirements for *all* algorithms are bound by the size of the intermediate result produced by the previous step, because the amount of context nodes of the context stack is limited by the result of the previous step. Note that intermediate results of path expressions are usually small compared to the document. However, in the worst case, all nodes of the document are context nodes. Hence the size of the context stack is $\mathcal{O}(p \times |D|)$, where p is the number of predicates for this step. Since the separate step expressions can be evaluated sequentially, this is also the space complexity for evaluating an entire path expression.

Time Complexity Let us first look at the time complexity for a single step. For each candidate output node, the algorithms evaluate each predicate only once for every context on the context stack. Therefore, the total time required for the evaluation of a step is $\mathcal{O}(p \times |D^2|)$, using the assumption that the evaluation of a predicate can be done in constant time.

Suppose now that we have n steps in a path expression. Then the total time complexity is $\mathcal{O}(\Sigma_{i=1}^{n} p_i \times |D^2|)$, where p_i is the number of predicates of the ith step. Since we know that $(p_1 + \ldots + p_n)$ is bounded by $|Q|$, we obtain that the time complexity of evaluating a path expression is $\mathcal{O}(|Q| \times |D^2|)$.

In the previous paragraphs we made the assumption of constant evaluation time for predicates. If we drop this assumption, time complexity becomes exponential in the maximum nesting depth of the predicates in the path expression. This nesting depth, however, is in most real life queries rather small, and many problems with nested predicates can be solved by applying rewriting techniques presented in [13] and [10].

5.2 Conclusion

After presenting very efficient algorithms, i.e., linear in the size of the intermediate results, we have extendedthis approach with support for evaluating positional predicates. The algorithms mostly operate well within the bounds described by Gottlob et al. [7, 6]. Only when applied to path expressions that contain an arbitrarily deep nesting of predicates, evaluation becomes inefficient.

The research presented in this paper is complementary to the work described in [11]. There, the focus lies on removing the pipeline blocking `distinct-doc-order` operations from XPath evaluation plans. However, for most path expressions it is impossible to remove all of these operations. Steps for which these sorting operations cannot be removed, can be efficiently evaluated with the algorithms presented in this work and in [12], depending on whether positional predicates occur in the step.

Future work will concentrate on two issues:

(1) Path expressions containing nested predicates cause the algorithms to behave exponentially in the predicate nesting depth. This is because we do not handle predicates in bulk, like normal steps and have to evaluate each predicate once for every node and for each context. In our future research we will attempt to eliminate this source of inefficiency.

(2) Another problem left to be solved is the lack of support for aggregate functions inside predicates. In our current approach we apply all predicates to every node separately. However, if an aggregate function is called, it is impossible to evaluate before all nodes are processed. We need to device an elegant way to solve this issue.

Bibliography

[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] M. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Automata for avoiding unnecessary ordering operations in XPath implementations. Technical Report TR2004-02, University of Antwerp, 2004. `http://www.adrem.ua.ac.be/pub/TR2004-02.pdf`.

[5] 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`.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in XPath. In *9th International Conference on Data Base Programming Languages*, volume 2921 of *Lecture Notes in Computer Science*, 2003.

[12] J. Hidders and P. Michiels. Efficient xpath axis evaluation for dom data structures. In *Plan-X 2004, Informal Proceedings*, pages 54–63, 2004. `http://www.brics.dk/NS/03/4/BRICS-NS-03-4.pdf`.

[13] 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.

[14] 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.