Efficient Algorithms for Processing XPath Queries

GEORG GOTTLOB, CHRISTOPH KOCH, and REINHARD PICHLER
Technische Universität Wien, Vienna, Austria

Our experimental analysis of several popular XPath processors reveals a striking fact: Query evaluation in each of the systems requires time exponential in the size of queries in the worst case. We show that XPath can be processed much more efficiently, and propose main-memory algorithms for this problem with polynomial-time combined query evaluation complexity. Moreover, we show how the main ideas of our algorithm can be profitably integrated into existing XPath processors. Finally, we present two fragments of XPath for which linear-time query processing algorithms exist and another fragment with linear-space/quadratic-time query processing.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages – *query languages*; I.7.2 [**Document and Text Processing**]: Document Preparation – *markup languages*

General Terms: Languages, Algorithms

Additional Key Words and Phrases: XML, XPath, Efficient Algorithms

1. INTRODUCTION

XPath has been proposed by the W3C [World Wide Web Consortium 1999] as a practical language for selecting nodes from XML document trees. The importance of XPath is due to its potential application as an XML query language per se and its role at the core of several other XML-related technologies, such as XSLT, XPointer, and XQuery. Since XPath and related technologies will be tested in ever-growing deployment scenarios, its implementations need to scale well *both* with respect to the size of the XML data and the growing size and intricacy of the queries (usually referred to as *combined complexity*).

Recently, there has been some work on the query evaluation problem for very restrictive fragments of XPath (usually in the context of data stream processing) [Altinel and Franklin 2000; Chan et al. 2002; Green et al. 2003; Peng and Chawathe 2003; Gupta and Suciu 2003; Bar-Yossef et al. 2004] and on related problems such as *structural joins* and XML query pattern matching [Bruno et al. 2002; Chan et al. 2002; Al-Khalifa et al. 2002]. However, to the best of our knowledge, no research results on processing full XPath or even moderately large fragments of this language have been published which may serve as yardsticks for new algorithms.

Contributions. In this article, we show that it is possible to noticeably improve the efficiency of existing and future XPath engines. We claim that current implementations of XPath processors do not live up to their potential. The way XPath

Authors' address: Database and Artificial Intelligence Group, Technische Universität Wien, A-1040 Vienna, Austria, {gottlob, koch}@dbai.tuwien.ac.at, reini@logic.at
This work was supported by the Austrian Science Fund (FWF) under project No. Z29-NO4. It is based on conference papers [Gottlob et al. 2002] and [Gottlob et al. 2003b]. All methods and algorithms presented here are covered by a pending patent. Further resources, updates, and possible corrections will be made available at http://www.xmltaskforce.com.

is defined in [World Wide Web Consortium 1999] motivates an implementation approach that leads to highly inefficient (exponential-time) XPath processing, and many implementations seem to have naively followed this intuition. Likewise, the semantics of a fragment of XPath defined in [Wadler 2000], which uses a fully functional formalism, motivates an exponential-time algorithm.

To get a better understanding of the state-of-the-art of XPath implementations, we experiment with four existing XPath processors, namely XALAN, XT, Saxon, and Microsoft Internet Explorer 6 (IE6). XALAN [Apache Foundation 2004] is a framework for processing XPath and XSLT which is freely available from the Apache Foundation. XT [Clark 1999] is a freely available XSLT[1] processor written by James Clark. Saxon [Kay 2003] is a freely available XSLT processor which was written by Michael Kay. Finally, IE6 [Microsoft Corporation 2001] is a commercial Web browser which supports the formatting of XML documents using XSL. Our experiments show that the time consumption of all four systems in general grows exponentially in the size of XPath queries. This exponentiality is a very practical problem. Of course, queries tend to be short, but we will argue that meaningful practical queries are *not short enough* to allow the existing systems to handle them.

The main contributions of this article, apart from our experiments, are the following:

—We define a formal bottom-up semantics of XPath (i.e., for the full language as proposed in [World Wide Web Consortium 1999]), which leads to a bottom-up main-memory XPath processing algorithm that runs in low-degree polynomial time in terms of the data and of the query size in the worst case. By a *bottom-up algorithm* we mean a method of processing XPath while traversing the *parse tree* of the query from its leaves up to its root.

—We discuss a general mechanism for translating our bottom-up algorithm into a top-down one. ("Top-down" again relates to the parse tree of the query.) Both have the same worst-case bound on running times but the latter may compute fewer useless intermediate results than the bottom-up algorithm.

—The top-down algorithm is enhanced to a new algorithm MinContext, which employs several heuristics. This new algorithm also slightly improves the worst-case complexity bounds.

—We show how the main ideas of our algorithms can be integrated into existing XPath processors. Practical experiments with Xalan confirm that these modifications indeed suffice to eliminate the source of exponential time complexity from these systems.

—We present a linear-time algorithm (in both data and query size) for a practically useful fragment of XPath, which we will call *Core XPath* in the sequel.

In the experiments presented in this article, we show that evaluating such queries in XALAN and XT already takes exponential time in the size of the queries in the worst case. The processing time of IE6 for this fragment grows polynomially in the size of queries, but requires quadratic time in the size of the XML *data* (when the query is fixed).

[1]Of course, XSLT allows to embed and execute arbitrary XPath queries.

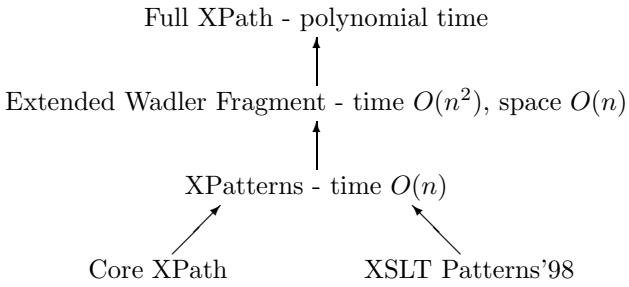


Fig. 1. XPath fragments considered in this article.

—We discuss the now superseded language of *XSLT Patterns* of the XSLT draft of December 16th, 1998 [World Wide Web Consortium 1998]. Since then, full XPath has been adopted as the XSLT Pattern language. XSLT Patterns remains interesting, as it shares many features with XPath and is a useful practical query language. We extend this language with all of the XPath axes and call it *XPatterns* to keep it short. Surprisingly, XPatterns queries can be evaluated very efficiently, in linear time in the size of the data and the query.

—We define the *Extended Wadler Fragment*, a very large fragment of XPath for which we provide an evaluation algorithm that works in quadratic time and linear space with respect to the size of the XML document. This fragment is of great practical value, since the vast majority of useful queries fall into it. Moreover, it pinpoints those features of XPath that are the most "expensive", even though their practical value is questionable.

—Finally, we present the algorithm OptMinContext, which combines the above results into one query processor with the following properties. (a) It supports all of XPath, with the runtime bounds obtained for the MinContext algorithm. Moreover, (b) for (subexpressions of) queries that fall either into the linear-time *Core XPath Fragment* or the quadratic-time, linear-space Extended Wadler Fragment, the OptMinContext algorithm adheres to these best known bounds.

An overview of the various query language fragments considered in this article and data complexity bounds of the associated algorithms is given in Figure 1. By $\mathcal{L}_1 \leftarrow \mathcal{L}_2$, we denote that language $\mathcal{L}_1$ subsumes language $\mathcal{L}_2$: XPatterns fully subsumes the Core XPath language, and subsumes XSLT Patterns'98 (except for a minor detail). XPatterns is a fragment of XPath. Likewise, the Extended Wadler Fragment fully subsumes Core XPath. Moreover, the integration of XSLT Patterns'98 into the Extended Wadler Fragment does not lead to a deterioration of the complexity bounds.

Structure. The structure of this article is as follows. In Section 2, we provide experimental results for existing XPath processors. Section 3 introduces axes for navigation in trees. Section 4 presents the data model of XPath and auxiliary functions used throughout the article. Section 5 defines the semantics of XPath in a concise way. Section 6 houses the bottom-up semantics definition and algorithm for full XPath, and Section 7 comes up with the modifications to obtain a top-down algorithm. In Section 8, we present several heuristics which also improve

the worst-case complexity of XPath evaluation. The improvement of other XPath processors by integrating our main ideas into them is dealt with in Section 9. Section 10 presents linear-time fragments of XPath (Core XPath and XPatterns). In Section 11, we show how to evaluate the Extended Wadler Fragment in linear space and quadratic time. We conclude with Section 12.

2. STATE-OF-THE-ART OF XPATH SYSTEMS

In this section, we evaluate the efficiency of four XPath engines, namely Apache XALAN (the Lotus/IBM XPath implementation which has been donated to the Apache Foundation), James Clark's XT, Michael Kay's Saxon, and Microsoft Internet Explorer 6 (IE6). The latter is a commercial product while the others are, as we believe, the three most popular freely available XPath engines.

We show by experiments that all four implementations require time exponential in the size of the queries in the worst case. Furthermore, we show that even the simplest queries, with which IE6 can deal efficiently in the size of the queries, take quadratic time in the size of the data. Note that the goal of these experiments is not to compare the systems against each other, but to test the scalabilities of their XPath processing algorithms.

XT, Saxon, and IE6 are not literally XPath engines, but are able to process XPath embedded in XSLT transformations. We used the xsl:for-each performative to obtain the set of all nodes an XPath query would evaluate to.

The version of XALAN used for the experiments was Xalan-j_2_2_D11 with the Xerces XML parser. We used the current version of XT with release tag 19991105, as available on James Clark's home page, in combination with his XP parser through the SAX driver. Finally, we used Saxon version 6.5.2 for our experiments. All of these three systems are Java implementations.

We ran XALAN and XT on a 360 MHz (dual processor) Ultra Sparc 60 with 512 MB of RAM running Solaris. Saxon was run on a Windows 2000 machine with a 700 MHz Pentium III processor and 256 MB of RAM. Finally, IE6 was evaluated on a Windows 2000 machine with a 1.2 GHz AMD K7 processor and 1.5 GB of RAM. The timings reported on here for Saxon and IE6 have the precision of ± 1 second, since Windows 2000 does not allow for the same accurate timing as Solaris.

For our experiments, we generated simple, flat XML documents. Each document $DOC(i)$ was of the form

$$\langle \mathrm{a} \rangle \underbrace{\langle \mathrm{b}/\rangle \ldots \langle \mathrm{b}/\rangle}_{i \text{ times}} \langle /\mathrm{a} \rangle$$

and its tree thus contained $i + 1$ element nodes.

In this section, the reader is assumed familiar with XPath and standard notions such as *axes* and *location steps* (cf. [World Wide Web Consortium 1999]). A formal definition of XPath follows in subsequent sections of this article.

Experiment 1: Exponential-time Query Complexity of XALAN and XT. In this experiment, we used the fixed document $DOC(2)$ (i.e., $\langle \mathrm{a} \rangle \langle \mathrm{b}/\rangle \langle \mathrm{b}/\rangle \langle /\mathrm{a} \rangle$). Queries were constructed using a simple pattern. The first query was '//a/b'. The $i+1$-th query was obtained by taking the i-th query and appending '/parent::a/b'. For instance, the third query was '//a/b/parent::a/b/parent::a/b'.

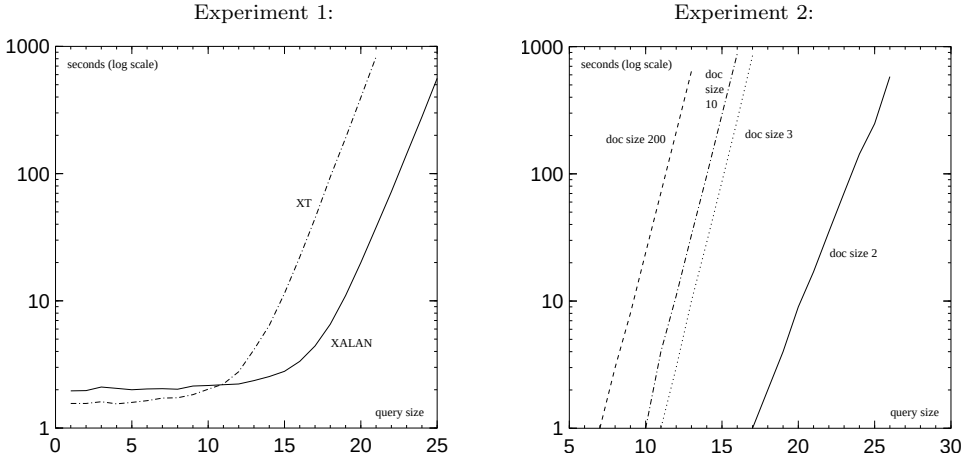


Fig. 2. Query complexity of XT and XALAN (Experiment 1) and of Saxon (Experiment 2).

It is easy to see that the time measurements reported in Figure 2 (Experiment 1), which uses a log scale Y axis, grow exponentially with the size of the query. The sharp bend in the curves is due to the near-constant runtime overhead of the Java VM and of parsing the XML document.

Discussion. The runtime behavior observed can be explained with the following pseudocode fragment, which seems to appropriately describe the basic query evaluation strategy of XT and XALAN.

procedure process-location-step(n_0, Q)
/* n_0 is the context node; query Q is a list of location steps */
begin
  node set S := apply Q.**head** to node n_0;
  if (Q.**tail** is not empty) **then**
    for each node $n \in S$ **do** process-location-step(n, Q.**tail**);
end

It is clear that each application of a location step to a context node may result in a set of nodes of size linear in the size of the document (e.g., each node may have a linear number of descendants or nodes appearing after it in the document). If we now proceed by recursively applying the location steps of an XPath query to individual nodes as shown in the pseudocode procedure above, we end up consuming time exponential in the size of the query in the worst case, even for very simple path queries. As a (simplified) recurrence, we have

$$\text{Time}(|Q|) := \begin{cases} |D| * \text{Time}(|Q|-1) & \ldots \quad |Q| > 0 \\ 1 & \ldots \quad |Q| = 0 \end{cases}$$

where $|Q|$ is the length of the query and $|D|$ is the document size, or equivalently

$$\text{Time}(|Q|) = |D|^{|Q|}.$$

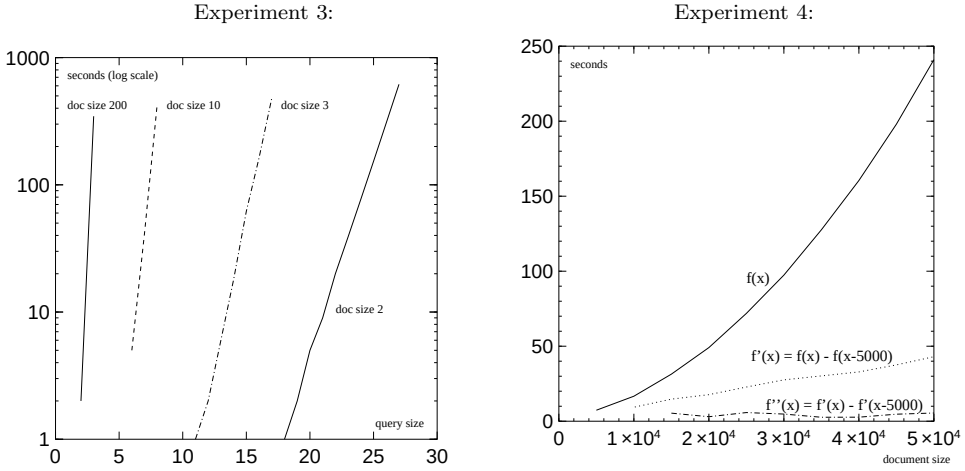


Fig. 3. Exponential-time query complexity of IE6 (Experiment 3) and Quadratic-time data complexity of IE6 (Experiment 4).

The class of queries used puts an emphasis on simplicity and reproducibility (using the *very* simple document ⟨a⟩⟨b/⟩⟨b/⟩⟨/a⟩). Interestingly, each 'parent::a/b' sequence quite exactly doubles the times both systems take to evaluate a query, as we first jump (back) to the tree root labeled "a" and then experience the "branching factor" of two due to the two child nodes labeled "b".

Experiment 2: Exponential-time Query Complexity of Saxon. In our second experiment, we executed queries that nest two important features of XPath, namely paths and relational operators, using Saxon. To this end, we slightly modified our XML-documents $DOC(i)$ to $DOC'(i)$ in that the b-elements are no longer empty. Instead they now all contain a simple text node with contents "c". Hence, $DOC'(i)$ is of the form

$$\langle \text{a} \rangle \underbrace{\langle \text{b} \rangle \text{c} \langle / \text{b} \rangle \ldots \langle \text{b} \rangle \text{c} \langle / \text{b} \rangle}_{i \text{ times}} \langle / \text{a} \rangle$$

The first three queries that we ran on the XML-documents $DOC'(i)$ for $i \in \{2, 3, 10, 200\}$ were

//*[parent::a/child::* = 'c']
//*[parent::a/child::*[parent::a/child::* = 'c'] = 'c']
//*[parent::a/child::*[parent::a/child::*[parent::a/child::* = 'c'] = 'c'] = 'c']

and it is clear how to continue this sequence.

The timings summarized in Figure 2 (Experiment 2) clearly show that Saxon requires time exponential in the size of the query.

Experiment 3: Exponential-time Query Complexity of Internet Explorer 6. In our third experiment, we executed queries that again nest two important features of XPath, namely paths and arithmetics, using IE6. The first three queries were

//a/b[count(parent::a/b) > 1]

//a/b[count(parent::a/b[count(parent::a/b) > 1]) > 1]

//a/b[count(parent::a/b[count(parent::a/b[count(parent::a/b) > 1]) > 1]) > 1]

Again it is clear how to continue this sequence.

Also this experiment was carried out for four document sizes (2, 3, 10, and 200). Figure 3 (Experiment 3) shows that also IE6 requires time exponential in the size of the query.

Experiment 4: Quadratic-time Data Complexity for Simple Path Queries (IE6). For our fourth experiment, we took a fixed query and benchmarked the time taken by IE6 for various document sizes. The query was '//a' + $q(20)$ + '//b' with

$$q(i) := \begin{cases} \text{'//b[ancestor::a'} + q(i-1) + \text{'//b]/ancestor::a'} & \dots \ i > 0 \\ \text{''} & \dots \ i = 0 \end{cases}$$

(Note: The size of queries $q(i)$ is of course $O(i)$.)

For instance, the query of size two, i.e. '//a' + $q(2)$ + '//b', according to this scheme is //a//b[ancestor::a//b[ancestor::a//b]/ancestor::a//b]/ancestor::a//b .

The granularity of measurements (in terms of document size) was 5000 nodes. Figure 3 (Experiment 4) shows that IE6 takes quadratic time w.r.t. the size of the data already for this simple class of path queries. Note that f' and f'' in Figure 3 are the first and second derivatives, respectively, of our graph of timings f.

The query complexity of IE6 for such queries is polynomial as well. Due to space limitations, we do not provide a graph for this experiment.

Queries that cause exponential runtime. It is usually argued that real-world queries are small, so query complexity is of minor relevance to practice. However, realistic document sizes allow only for *very* short queries to be dealt with by current XPath engines. We demonstrate this in Experiment 3 for IE6, and verified it for the other systems as well. XPath query engines need to be able to deal with increasingly sophisticated queries, along the current trend to delegate larger and larger parts of data management problems to query engines, where they can profit from their efficiency and can be made subject to optimization. The intuition that XPath can be used to match a large class of *tree patterns* [Shasha et al. 2002; Kilpeläinen 1992; Bruno et al. 2002] in XML documents also implies to a certain degree that queries may be of some size.

The queries used in the previous experiments employ antagonist axes (such as "child" and "parent") to jump back and forth within the input documents. Our queries may seem contrived, but our goal was to exhibit queries that use only few XPath language constructs but still cause current XPath engines to take exponential time. Queries using antagonist axes such as "following" and "preceding" instead of "child" and "parent" do have practical applications, such as when we want to put restrictions on the relative positions of nodes in a document.

Moreover, if we make the realistic assumption that the documents are always much larger than the queries ($|Q| << |D|$), it is not even necessary to jump back and forth with antagonist axes. We can use queries such as

$$\text{//following::*/following::*/}\dots\text{/following::*}$$

to observe exponential behavior. We illustrate this by two further experiments. In

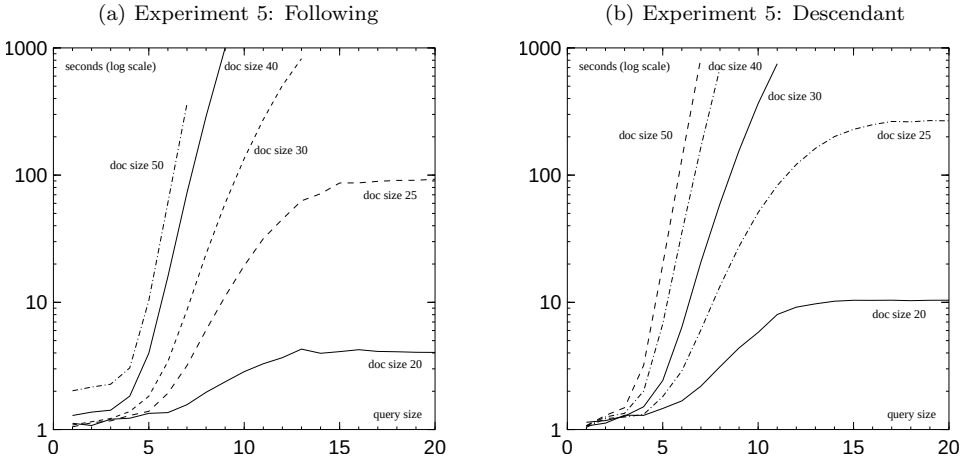


Fig. 4. Complexity of Xalan for queries that employ exclusively forward axes.

both we used Xalan-Java2 (release 2.6.0) with Xerces 2.6.2 on a PC with 128MB of main memory running FreeBSD 4.10.

Experiment 5: Exponential-time Complexity with Forward Axes only. In the first experiment, each query of size k was of the form

$$\text{count}(//b\underbrace{/\text{following::b/following::b/}\ldots\text{/following::b}}_{k-1 \text{ times}})$$

The graph of Figure 4 (a) reports on the query complexity of this family of queries for documents $DOC(i)$ with $i \in \{20, 25, 30, 40, 50\}$. What shows is that the complexity is exponential in the size of the query up to a size that depends on the document, at which the cost of query evaluation in terms of query size stabilizes.

The most frequently used XPath axes are "child" and "descendant", so we also present an analogous experiment for the latter axis. Figure 4 (b) shows the evaluation times of queries of the form

$$\text{count}(\underbrace{//\text{b//b}\ \ldots\ //\text{b}}_{k \text{ times}})$$

(for size k.) The documents here were constructed in a different fashion; a document of size i was

$$\underbrace{\langle b\rangle \ldots \langle b\rangle}_{i \text{ times}}\underbrace{\langle /b\rangle \ldots \langle /b\rangle}_{i \text{ times}}$$

that is, a (non-branching) path of i b-nodes. The experiments were run for paths of 20, 25, 30, 40, and 50 b-nodes.

Consider document size 50. What happens here (assuming again the naive algorithm sketched in the discussion of Experiment 1) is basically that the first "descendant" location step evaluates to 50 nodes, for each of which the second location step evaluates to between zero and 49 nodes (all nodes below the current node),

and so forth for a recursion k steps deep. XML documents of depth 50 are not commonplace, but it is easy to see that the same naive algorithm is also very costly on massive (wide) XML trees of moderate depth.

Of course, simple path queries using only the downward axes ("child" "descendant", and "descendant-or-self") can be evaluated more efficiently using special purpose techniques such as structural joins (cf. [Al-Khalifa et al. 2002] and follow-up work) and methods developed for evaluating simple XPath queries on data streams (see e.g. [Altinel and Franklin 2000]). These techniques work only for very small fragments of XPath. We observed that if we remove the enclosing "count" from the queries of Experiment 5, Xalan evaluates them much more efficiently. However, queries that use advanced language features such as "count" are evaluated naively.

Question. The following question naturally arises from our experiments: Is there an algorithm for processing XPath with guaranteed polynomial-time behavior (combined complexity), or even one that requires only linear time for simple queries? In Section 6, we are able to provide a positive answer to this.

3. XPATH AXES

In this section, we formally define *XPath axes*, that is, the interpreted binary relations that XPath provides for navigating in XML document trees. We also present techniques for evaluating XPath axes; these take linear time and are thus worst-case optimal. Special algorithms for evaluating axes that work more efficiently in practice have been proposed in the context of structural joins (see e.g. [Al-Khalifa et al. 2002; Bruno et al. 2002]) and XML-frontends for relational database management systems [Grust et al. 2004], so the main role of this section is to provide a foundation for the formal semantics of XPath that we will give later on. The actual techniques for evaluating axes in our efficient XPath processing algorithms will be interchangeable.

In XPath, an XML document is viewed as an unranked (i.e., nodes may have a variable number of children), ordered, and labeled tree. Before we make the data model used by XPath precise (which distinguishes between several types of tree nodes) in Section 4, we introduce the main mode of navigation in document trees employed by XPath – axes – in the abstract, ignoring node types. We will point out how to deal with different node types in Section 4.

All of the artifacts of this and the next section are defined in the context of a given XML document. Given a document tree, let dom be the set of its nodes, and let us use the two functions

$$\text{firstchild, nextsibling} : \text{dom} \to \text{dom},$$

to represent its structure[2]. "firstchild" returns the first child of a node (if there are any children, i.e., the node is not a leaf), and otherwise "null". Let $n_1, \ldots, n_k$ be the children of some node in document order. Then, $\text{nextsibling}(n_i) = n_{i+1}$, i.e., "nextsibling" returns the neighboring node to the right, if it exists, and "null" otherwise (if $i = k$). We define the functions firstchild^{-1} and nextsibling^{-1} as the inverses of the former two functions, where "null" is returned if no inverse exists

[2]Actually, "firstchild" and "nextsibling" are part of the XML Document Object Model (DOM).

child := firstchild.nextsibling*
parent := (nextsibling^{-1})*.firstchild^{-1}
descendant := firstchild.(firstchild $\cup$ nextsibling)*
ancestor := (firstchild^{-1} $\cup$ nextsibling^{-1})*.firstchild^{-1}
descendant-or-self := descendant $\cup$ self
ancestor-or-self := ancestor $\cup$ self
following := ancestor-or-self.nextsibling.nextsibling*.descendant-or-self
preceding := ancestor-or-self.nextsibling^{-1}.(nextsibling^{-1})*.descendant-or-self
following-sibling := nextsibling.nextsibling*
preceding-sibling := (nextsibling^{-1})*.nextsibling^{-1}

Table I. Axis definitions in terms of "primitive" tree relations "firstchild", "nextsibling", and their inverses.

for a given node. Where appropriate, we will use binary relations of the same name instead of the functions. ($\{\langle x, f(x)\rangle \mid x \in \text{dom},\ f(x) \neq \text{null}\}$ is the binary relation for function f.)

The axes *self*, *child*, *parent*, *descendant*, *ancestor*, *descendant-or-self*, *ancestor-or-self*, *following*, *preceding*, *following-sibling*, and *preceding-sibling* are binary relations $\chi \subseteq \text{dom} \times \text{dom}$. Let self := $\{\langle x, x\rangle \mid x \in \text{dom}\}$. The other axes are defined in terms of our "primitive" relations "firstchild" and "nextsibling" as shown in Table I (cf. [World Wide Web Consortium 1999]). $R_1.R_2$, $R_1 \cup R_2$, and R_1^* denote the concatenation, union, and reflexive and transitive closure, respectively, of binary relations R_1 and R_2. Let $E(\chi)$ denote the regular expression defining χ in Table I. It is important to observe that some axes are defined in terms of other axes, but that these definitions are acyclic.

Definition 3.1. (Axis Function) Let χ denote an XPath axis relation. We define the function $\chi : 2^{\text{dom}} \to 2^{\text{dom}}$ as $\chi(X_0) = \{x \mid \exists x_0 \in X_0 : x_0 \chi x\}$ (and thus overload the relation name χ), where $X_0 \subseteq \text{dom}$ is a set of nodes. □

Algorithm 3.2. (Axis Evaluation)
Input: A set of nodes S and an axis χ
Output: $\chi(S)$
Method: $\text{eval}_\chi(S)$

function $\text{eval}_{(R_1 \cup \cdots \cup R_n)^*}(S)$ **begin**
  $S' := S$; /* S' is represented as a list */
  while there is a next element x in S' **do**
    append $\{R_i(x) \mid 1 \leq i \leq n,\ R_i(x) \neq \text{null},\ R_i(x) \notin S'\}$ to S';
  return S';
end;
function $\text{eval}_\chi(S) := \text{eval}_{E(\chi)}(S)$.
function $\text{eval}_{\text{self}}(S) := S$.
function $\text{eval}_{e_1.e_2}(S) := \text{eval}_{e_2}(\text{eval}_{e_1}(S))$.
function $\text{eval}_R(S) := \{R(x) \mid x \in S\}$.
function $\text{eval}_{\chi_1 \cup \chi_2}(S) := \text{eval}_{\chi_1}(S) \cup \text{eval}_{\chi_2}(S)$.

where $S \subseteq \text{dom}$ is a set of nodes of an XML document, e_1 and e_2 are regular expressions, R, R_1, ..., R_n are primitive relations or their inverses, χ_1 and χ_2 are

axes, and χ is an axis other than "self". □

Clearly, some of the axes could have been defined in a simpler way in Table I (e.g., ancestor equals parent.parent*). However, the definitions, which use a limited form of regular expressions only, allow to compute $\chi(S)$ in a very simple way, as evidenced by Algorithm 3.2.

Consider the directed graph $G = (V, E)$ with $V = \text{dom}$ and $E = R_1 \cup \cdots \cup R_n$. The function $\text{eval}_{(R_1 \cup \cdots \cup R_n)^*}(S)$ computes the set of nodes reachable from any of the nodes in S in zero or more steps. (This is easier than computing the reflexive and transitive closure of G as a binary relation). It can be implemented to run in linear time in terms of the size of the data (corresponding to the edge relation E of the graph[3]) in a straightforward manner; (non)membership in S' is checked in constant time using a direct-access version of S' maintained in parallel to its list representation. Naively, this could be an array of bits, one for each member of dom, telling which nodes are in S'.[4]

LEMMA 3.3. *Let $S \subseteq dom$ be a set of nodes of an XML document and χ be an axis. Then,*

(1) $\chi(S) = eval_\chi(S)$ *and*
(2) Algorithm 3.2 runs in time $O(|dom|)$.

PROOF ($O(|\text{DOM}|)$ RUNNING TIME). The time bound is due to the fact that each of the eval functions can be implemented so as to visit each node at most once and the number of calls to eval functions and relations joined by union is constant (see Table I). □

4. DATA MODEL

Let dom be the set of nodes in the document tree as introduced in the previous section. Each node is of one of seven *types*, namely root, element, text, comment, attribute, namespace, and processing instruction. As in DOM [World Wide Web Consortium], the root node of the document is the only one of type "root", and is the *parent* of the document element node of the XML document. The main type of non-terminal node is "element", the other node types are self-explaining (cf. [World Wide Web Consortium 1999]). Nodes of all types besides "text" and "comment" have a name associated with them.

A *node test* is an expression of the form $\tau()$ (where τ is a node type or the wildcard "node", matching any type) or $\tau(n)$ (where n is a node name and τ is a type whose nodes have a name). $\tau(*)$ is equivalent to $\tau()$. We define a function T which maps each node test to the subset of dom that satisfies it. For instance, $T(\text{node}()) = \text{dom}$ and $T(\text{attribute}(\text{href}))$ returns all attribute nodes labeled "href".

EXAMPLE 4.1. Consider $DOC(4)$ of Section 2. It consists of *six* nodes – the document element node a labeled "a", its four children $b_1, \ldots, b_4$ (labeled "b"), and a root node r which is the parent of a. We have $T(\text{root}()) = \{r\}$, $T(\text{element}()) = \{a, b_1, \ldots, b_4\}$, $T(\text{element}(a)) = \{a\}$ and, finally, $T(\text{element}(b)) = \{b_1, \ldots, b_4\}$. □

[3]Note that $|E| \approx 2 \cdot |T|$, where $|T|$ is the size of the edge relation of the document tree.
[4]A remotely similar idea is used in the TPQSimulation algorithm of [Ramanan 2002].

Now, XPath axes differ from the abstract, untyped axes of Section 3 (which we refer to using a subscript $_0$ below) in that there are special child axes "attribute" and "namespace" which filter out all resulting nodes that are not of type attribute or namespace, respectively. In turn, all other XPath axis functions remove nodes of these two types from their results. We can express this formally, simulating XPath axes using abstract axes, as

$$\begin{aligned} \text{attribute}(S) &:= \text{child}_0(S) \cap T(\text{attribute}()) \\ \text{namespace}(S) &:= \text{child}_0(S) \cap T(\text{namespace}()) \end{aligned}$$

and for all other XPath axes χ,

$$\chi(S) := \chi_0(S) - (T(\text{attribute}()) \cup T(\text{namespace}())).$$

Node tests that occur explicitly in XPath queries must not use the types "root", "attribute", or "namespace"[5]. In XPath, axis applications χ and node tests t always come in *location step* expressions of the form $\chi::t$. The node test n (where n is a node name or the wildcard *) is a shortcut for $\tau(n)$, where τ is the *principal node type* of χ. For the axis attribute, the principal node type is attribute, for namespace it is namespace, and for all other axes, it is element. For example, child::a is short for child::element(a) and child::* is short for child::element(*).

Note that for a set of nodes S and a typed axis χ, $\chi(S)$ can be computed in linear time – just as for the untyped axes of Section 3.

Let $<_{doc}$ be the binary document order relation, such that $x <_{doc} y$ (for two nodes $x, y \in \text{dom}$) iff the opening tag of x precedes the opening tag of y in the (well-formed) document. The function $\text{first}_{<_{doc}}$ returns the first node in a set w.r.t. document order. We define the relation $<_{doc,\chi}$ relative to the axis χ as follows. For $\chi \in \{\text{self, child, descendant, descendant-or-self, following-sibling, following}\}$, $<_{doc,\chi}$ is the standard document order relation $<_{doc}$. For the remaining axes, it is the reverse document order $>_{doc}$. Moreover, given a node x and a set of nodes S with $x \in S$, let $\text{idx}_\chi(x, S)$ be the index of x in S w.r.t. $<_{doc,\chi}$ (where 1 is the smallest index).

Given an XML Document Type Definition (DTD) [World Wide Web Consortium 2000] that uses the ID/IDREF feature, some element nodes of the document may be identified by a unique id. The function $\text{deref\_ids} : \text{string} \to 2^{\text{dom}}$ interprets its input string as a whitespace-separated list of keys and returns the set of nodes whose ids are contained in that list.

The function $\text{strval} : \text{dom} \to \text{string}$ returns the *string value* of a node, for the precise definition of which we refer to [World Wide Web Consortium 1999]. Notably, the string value of an element or root node x is the concatenation of the string values of the nodes in $\text{descendant}(\{x\}) \cap T(\text{text}())$ visited in document order. The functions to_string and to_number convert a number to a string resp. a string to a number according to the rules specified in [World Wide Web Consortium 1999].

This concludes our discussion of the XPath data model, which is complete except for some details related to namespaces. This topic is mostly orthogonal to our discussion, and extending our framework to also handle namespaces (without a

[5]These node tests are also redundant with '/' and the "attribute" and "namespace" axes.

penalty with respect to efficiency bounds) is an easy exercise[6].

5. SEMANTICS OF XPATH

In this section, we present a concise definition of the semantics of XPath [World Wide Web Consortium 1999]. We assume the syntax of this language known, and cohere with its so-called *unabbreviated* form. This means that

—in all occurrences of the child or descendant axis in the XPath expression, the axis names have to be stated explicitly; for example, we write /descendant::a/child::b rather than //a/b.

—Bracketed condition expressions $[e]$, where e is an expression that produces a number (see below), correspond to [position() = e] in unabbreviated syntax. For example, the abbreviated XPath expression //a[5], which refers to the fifth node (with respect to document order) occurring in the document which is labeled "a", is written as /descendant::a[position() = 5] in unabbreviated syntax.

—All type conversions have to be made explicit (using the conversion functions string, number, and boolean, which we will define below). For example, we write /descendant::a[boolean(child::b)] rather than /descendant::a[child::b].

Moreover, as XPath expressions may use variables for which a given binding has to be supplied with the expression, each variable is replaced by the (constant) value of the input variable binding.

These assumptions do not cause any loss of generality, but reduce the number of cases we have to distinguish in the semantics definition below.

The main syntactic constructs of XPath are *expressions*, which are of one of four types, namely *node set*, *number*, *string*, or *boolean*. Each expression evaluates relative to a context $\vec{c} = \langle x, k, n \rangle$ consisting of a *context node* x, a *context position* k, and a *context size* n [World Wide Web Consortium 1999]. By the *domain of contexts*, we mean the set

$$\mathbf{C} = \mathrm{dom} \times \{\langle k, n \rangle \mid 1 \leq k \leq n \leq |\mathrm{dom}|\}.$$

Let

$$\begin{aligned} \mathit{ArithOp} &\in \{+, -, *, \mathrm{div}, \mathrm{mod}\}, & \mathit{EqOp} &\in \{=, \neq\}, \\ \mathit{RelOp} &\in \{=, \neq, \leq, <, \geq, >\}, & \mathit{GtOp} &\in \{\leq, <, \geq, >\}. \end{aligned}$$

By slight abuse of notation, we identify these arithmetic and relational operations with their symbols in the remainder of this article. However, it should be clear whether we refer to the operation or its symbol at any point. By $\pi, \pi_1, \pi_2, \ldots$ we denote location paths.

DEFINITION 5.1. (Semantics of XPath) Each XPath expression returns a value of one of the following four types: number, node set, string, and boolean (abbreviated num, nset, str, and bool, respectively). Let $\mathcal{T}$ be an expression type and the

[6]To be consistent, we also will not discuss the "local-name", "namespace-uri", and "name" core library functions [World Wide Web Consortium 1999].

Note that names used in node tests may be of the form NCName:*, which matches all names from a given namespace named NCNAME.

(* absolute location paths *)
$P[\![/\pi]\!](x) := P[\![\pi]\!](\text{root})$

(* composition of location paths *)
$P[\![\pi_1/\pi_2]\!](x) := \bigcup_{y \in P[\![\pi_1]\!](x)} P[\![\pi_2]\!](y)$

(* "disjunction" of location paths *)
$P[\![\pi_1|\pi_2]\!](x) := P[\![\pi_1]\!](x) \cup P[\![\pi_2]\!](x)$

(* location steps *)
$P[\![\chi{::}t[e_1]\cdots[e_m]]\!](x) :=$
begin
  $S := \{y \mid x\chi y,\ y \in T(t)\}$;
  for $1 \le i \le m$ (in ascending order) **do**
    $S := \{y \in S \mid [\![e_i]\!](y, \text{idx}_\chi(y,S), |S|) = \text{true}\}$;
  return S;
end;

Fig. 5. Standard semantics of location paths.

semantics $[\![e]\!] : \mathbf{C} \to \mathcal{T}$ of XPath expression e be defined as follows.

$$[\![\pi]\!](\langle x,k,n\rangle) := P[\![\pi]\!](x) \qquad [\![\text{position}()]\!](\langle x,k,n\rangle) := k$$
$$[\![\text{string}()]\!](\langle x,k,n\rangle) := \text{strval}(x) \qquad [\![\text{last}()]\!](\langle x,k,n\rangle) := n$$
$$[\![\text{number}()]\!](\langle x,k,n\rangle) := \text{to\_number}(\text{strval})(x)$$

For all other kinds of expressions $e = Op(e_1,\ldots,e_m)$ mapping a context $\vec{c}$ to a value of type $\mathcal{T}$,

$$[\![Op(e_1,\ldots,e_m)]\!](\vec{c}) := \mathcal{F}[\![Op]\!]([\![e_1]\!](\vec{c}),\ldots,[\![e_m]\!](\vec{c})),$$

where $\mathcal{F}[\![Op]\!] : \mathcal{T}_1 \times \cdots \times \mathcal{T}_m \to \mathcal{T}$ is called the *effective semantics function* of Op. The function P is defined in Figure 5 and the effective semantics function $\mathcal{F}$ is defined in Table II. □

To save space, we at times re-use function definitions in Table II to define others. However, our definitions are not circular and the indirections can be eliminated by a constant number of unfolding steps. Moreover, we define neither the number operations floor, ceiling, and round nor the string operations concat, starts-with, contains, substring (two versions), substring-before, substring-after, string-length, normalize-space, translate, and lang in Table II, but it is very easy to obtain these definitions from the XPath Recommendation [World Wide Web Consortium 1999].

The compatibility of our semantics definition (modulo the assumptions made in this article to simplify the data model) with [World Wide Web Consortium 1999] can easily be verified by inspection of the latter document.

It is instructive to compare the definition of $P[\![\pi_1/\pi_2]\!]$ in Figure 5 with the procedure process-location-step of Section 2 and the claim regarding exponential-time query evaluation made there. In fact, if the semantics definition of [World Wide Web Consortium 1999] (or of this section, for that matter) is followed rigorously to obtain an analogous functional implementation, query evaluation using this implementation requires time exponential in the size of the queries.

6. BOTTOM-UP EVALUATION OF XPATH

In this section, we present a semantics and an algorithm for evaluating XPath queries in polynomial time which both use a "bottom-up" intuition. We discuss the intuitions which lead to polynomial time evaluation (which we call the "context-value table principle"), and establish the correctness and complexity results.

Definition 6.1. (Semantics) We represent the four XPath expression types nset, num, str, and bool using relations as shown in Table III. The bottom-up

Expr. E : **Operator Signature** $\Rightarrow$ **Semantics** $\mathcal{F}[\![E]\!]$
$\mathcal{F}[\![\text{constant number } v : \rightarrow \text{num}]\!]() := v$
$\mathcal{F}[\![\mathit{ArithOp} : \text{num} \times \text{num} \rightarrow \text{num}]\!](v_1, v_2) := v_1 \ \mathit{ArithOp}\ v_2$
$\mathcal{F}[\![\text{count} : \text{nset} \rightarrow \text{num}]\!](S) := \lvert S \rvert$
$\mathcal{F}[\![\text{sum} : \text{nset} \rightarrow \text{num}]\!](S) := \Sigma_{n \in S}\ \text{to\_number}(\text{strval}(n))$
$\mathcal{F}[\![\text{id} : \text{nset} \rightarrow \text{nset}]\!](S) := \bigcup_{n \in S} \mathcal{F}[\![\text{id}]\!](\text{strval}(n))$
$\mathcal{F}[\![\text{id} : \text{str} \rightarrow \text{nset}]\!](s) := \text{deref\_ids}(s)$
$\mathcal{F}[\![\text{constant string } s : \rightarrow \text{str}]\!]() := s$
$\mathcal{F}[\![\text{and} : \text{bool} \times \text{bool} \rightarrow \text{bool}]\!](b_1, b_2) := b_1 \wedge b_2$
$\mathcal{F}[\![\text{or} : \text{bool} \times \text{bool} \rightarrow \text{bool}]\!](b_1, b_2) := b_1 \vee b_2$
$\mathcal{F}[\![\text{not} : \text{bool} \rightarrow \text{bool}]\!](b) := \neg b$
$\mathcal{F}[\![\text{true}() : \rightarrow \text{bool}]\!]() := \text{true}$
$\mathcal{F}[\![\text{false}() : \rightarrow \text{bool}]\!]() := \text{false}$
$\mathcal{F}[\![\mathit{RelOp} : \text{nset} \times \text{nset} \rightarrow \text{bool}]\!](S_1, S_2) :=$ $\exists n_1 \in S_1, n_2 \in S_2 : \text{strval}(n_1)\ \mathit{RelOp}\ \text{strval}(n_2)$
$\mathcal{F}[\![\mathit{RelOp} : \text{nset} \times \text{num} \rightarrow \text{bool}]\!](S, v) := \exists n \in S : \text{to\_number}(\text{strval}(n))\ \mathit{RelOp}\ v$
$\mathcal{F}[\![\mathit{RelOp} : \text{nset} \times \text{str} \rightarrow \text{bool}]\!](S, s) := \exists n \in S : \text{strval}(n)\ \mathit{RelOp}\ s$
$\mathcal{F}[\![\mathit{RelOp} : \text{nset} \times \text{bool} \rightarrow \text{bool}]\!](S, b) := \mathcal{F}[\![\text{boolean}]\!](S)\ \mathit{RelOp}\ b$
$\mathcal{F}[\![\mathit{EqOp} : \text{bool} \times (\text{str} \cup \text{num} \cup \text{bool}) \rightarrow \text{bool}]\!](b, x) := b\ \mathit{EqOp}\ \mathcal{F}[\![\text{boolean}]\!](x)$
$\mathcal{F}[\![\mathit{EqOp} : \text{num} \times (\text{str} \cup \text{num}) \rightarrow \text{bool}]\!](v, x) := v\ \mathit{EqOp}\ \mathcal{F}[\![\text{number}]\!](x)$
$\mathcal{F}[\![\mathit{EqOp} : \text{str} \times \text{str} \rightarrow \text{bool}]\!](s_1, s_2) := s_1\ \mathit{EqOp}\ s_2$
$\mathcal{F}[\![\mathit{GtOp} : (\text{str} \cup \text{num} \cup \text{bool}) \times (\text{str} \cup \text{num} \cup \text{bool}) \rightarrow \text{bool}]\!](x_1, x_2) :=$ $\mathcal{F}[\![\text{number}]\!](x_1)\ \mathit{GtOp}\ \mathcal{F}[\![\text{number}]\!](x_2)$
$\mathcal{F}[\![\text{string} : \text{num} \rightarrow \text{str}]\!](v) := \text{to\_string}(v)$
$\mathcal{F}[\![\text{string} : \text{nset} \rightarrow \text{str}]\!](S) :=$ **if** $S = \emptyset$ **then** "" **else** $\text{strval}(\text{first}_{<_{doc}}(S))$
$\mathcal{F}[\![\text{string} : \text{bool} \rightarrow \text{str}]\!](b) :=$ **if** b=true **then** "true" **else** "false"
$\mathcal{F}[\![\text{boolean} : \text{str} \rightarrow \text{bool}]\!](s) :=$ **if** $s \neq$ "" **then** true **else** false
$\mathcal{F}[\![\text{boolean} : \text{num} \rightarrow \text{bool}]\!](v) :=$ **if** $v \neq \pm 0$ **and** $v \neq \mathit{NaN}$ **then** true **else** false
$\mathcal{F}[\![\text{boolean} : \text{nset} \rightarrow \text{bool}]\!](S) :=$ **if** $S \neq \emptyset$ **then** true **else** false
$\mathcal{F}[\![\text{number} : \text{str} \rightarrow \text{num}]\!](s) := \text{to\_number}(s)$
$\mathcal{F}[\![\text{number} : \text{bool} \rightarrow \text{num}]\!](b) :=$ **if** b=true **then** 1 **else** 0
$\mathcal{F}[\![\text{number} : \text{nset} \rightarrow \text{num}]\!](S) := \mathcal{F}[\![\text{number}]\!](\mathcal{F}[\![\text{string}]\!](S))$

Table II. XPath effective semantics functions.

Expression Type	**Associated Relation** R
num	$R \subseteq \mathbf{C} \times \mathbb{R}$
bool	$R \subseteq \mathbf{C} \times \{\text{true}, \text{false}\}$
nset	$R \subseteq \mathbf{C} \times 2^{\text{dom}}$
str	$R \subseteq \mathbf{C} \times \text{char}^*$

Table III. Expression types and associated relations.

Expr. E : **Signature**	**Semantics** $\mathcal{E}_\uparrow[\![E]\!]$
location step χ::t : $\rightarrow$ nset	$\{\langle x_0, k_0, n_0, \{x \mid x_0 \chi x,\ x \in T(t)\}\rangle \mid$ $\langle x_0, k_0, n_0\rangle \in \mathbf{C}\}$
location step $E[e]$ over axis χ: nset $\times$ bool $\rightarrow$ nset	$\{\langle x_0, k_0, n_0, \{x \in S \mid \langle x, \mathrm{idx}_\chi(x, S), \lvert S\rvert, \mathrm{true}\rangle$ $\in \mathcal{E}_\uparrow[\![e]\!]\}\rangle \mid \langle x_0, k_0, n_0, S\rangle \in \mathcal{E}_\uparrow[\![E]\!]\}$
location path $/\pi$: nset $\rightarrow$ nset	$\mathbf{C} \times \{S \mid \exists k, n : \langle \mathrm{root}, k, n, S\rangle \in \mathcal{E}_\uparrow[\![\pi]\!]\}$
location path π_1/π_2 : nset $\times$ nset $\rightarrow$ nset	$\{\langle x, k, n, \bigcup\{Z \mid \langle y, k_2, n_2, Z\rangle \in \mathcal{E}_\uparrow[\![\pi_2]\!],\ y \in Y\}\rangle$ $\mid 1 \leq k \leq n \leq \lvert\mathrm{dom}\rvert,\ \langle x, k_1, n_1, Y\rangle \in \mathcal{E}_\uparrow[\![\pi_1]\!]\}$
location path $\pi_1 \mid \pi_2$: nset $\times$ nset $\rightarrow$ nset	$\{\langle x, k, n, S_1 \cup S_2\rangle \mid \langle x, k, n, S_1\rangle \in \mathcal{E}_\uparrow[\![\pi_1]\!],$ $\langle x, k, n, S_2\rangle \in \mathcal{E}_\uparrow[\![\pi_2]\!]\}$
position() : $\rightarrow$ num	$\{\langle x, k, n, k\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$
last() : $\rightarrow$ num	$\{\langle x, k, n, n\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$
string() : $\rightarrow$ str	$\{\langle x, k, n, \mathrm{strval}(x)\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$
number() : $\rightarrow$ num	$\{\langle x, k, n, \mathrm{to\_number}(\mathrm{strval}(x))\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$

Table IV. Expression relations for location paths, position(), last(), string(), and number().

semantics of expressions is defined via a semantics function

$$\mathcal{E}_\uparrow : \text{Expression} \rightarrow 2^{\mathbf{C} \times (\text{nset} \cup \text{num} \cup \text{str} \cup \text{bool})},$$

given in Table IV and as

$$\mathcal{E}_\uparrow[\![Op(e_1, \ldots, e_m)]\!] := \{\langle \vec{c}, \mathcal{F}[\![Op]\!](v_1, \ldots, v_m)\rangle \mid \vec{c} \in \mathbf{C},\ \langle \vec{c}, v_1\rangle \in \mathcal{E}_\uparrow[\![e_1]\!],\ \ldots,\ \langle \vec{c}, v_m\rangle \in \mathcal{E}_\uparrow[\![e_m]\!]\}$$

for the remaining kinds of XPath expressions. □

Now, for each expression e and each $\langle x, k, n\rangle \in \mathbf{C}$, there is exactly one v s.t. $\langle x, k, n, v\rangle \in \mathcal{E}_\uparrow[\![e]\!]$, and which happens to be the value $[\![e]\!](\langle x, k, n\rangle)$ of e on $\langle x, k, n\rangle$ (see Definition 5.1).

THEOREM 6.2. *Let e be an arbitrary XPath expression, $\langle x, k, n\rangle \in \mathbf{C}$ a context, and $v = [\![e]\!](\langle x, k, n\rangle)$ the value of e. Then, v is the unique value such that $\langle x, k, n, v\rangle \in \mathcal{E}_\uparrow[\![e]\!]$.*

The main principle that we propose at this point to obtain an XPath evaluation algorithm with polynomial-time complexity is the notion of a *context-value table* (i.e., a relation for each expression, as discussed above).

Context-value Table Principle. Given an expression e that occurs in the input query, the context-value table of e specifies all valid combinations of contexts $\vec{c}$ and values v, such that e evaluates to v in context $\vec{c}$. Such a table for expression e is obtained by first computing the context-value tables of the direct subexpressions of e and subsequently combining them into the context-value table for e. Given that the size of each of the context-value tables has a polynomial bound and each of the combination steps can be effected in polynomial time (all of which we can assure in the following), query evaluation in total under our principle also has a polynomial time bound[7]. □

[7]The number of expressions to be considered is fixed with the parse tree of a given query.

Query Evaluation. The idea of Algorithm 6.3 below is so closely based on our semantics definition that its correctness follows directly from the correctness result of Theorem 6.2.

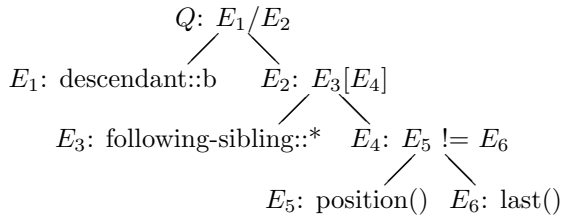
ALGORITHM 6.3. (Bottom-up algorithm for XPath)
Input: An XPath query Q;
Output: $\mathcal{E}_\uparrow[\![Q]\!]$.
Method:

let Tree(Q) be the parse tree of query Q;
$\mathbf{R} := \emptyset$; (* a *set* of context-value tables *)
for each atomic expression $l \in$ leaves(Tree(Q)) **do**
  compute table $\mathcal{E}_\uparrow[\![l]\!]$ and add it to $\mathbf{R}$;
while $\mathcal{E}_\uparrow[\![\text{root(Tree}(Q))]\!] \notin \mathbf{R}$ **do**
begin
  take an $\text{Op}(l_1, \ldots, l_n) \in$ nodes(Tree(Q)) s.t. $\mathcal{E}_\uparrow[\![l_1]\!], \ldots, \mathcal{E}_\uparrow[\![l_n]\!] \in \mathbf{R}$;
  compute $\mathcal{E}_\uparrow[\![\text{Op}(l_1, \ldots, l_n)]\!]$ using $\mathcal{E}_\uparrow[\![l_1]\!], \ldots, \mathcal{E}_\uparrow[\![l_n]\!]$;
  add $\mathcal{E}_\uparrow[\![\text{Op}(l_1, \ldots, l_n)]\!]$ to $\mathbf{R}$;
end;
return $\mathcal{E}_\uparrow[\![\text{root(Tree}(Q))]\!]$. □

EXAMPLE 6.4. Consider document $DOC(4)$ of Section 2. Let dom $= \{r, a, b_1, \ldots, b_4\}$, where r denotes the root node, a the document element node (the child of r, labeled a) and $b_1, \ldots, b_4$ denote the children of a in document order (labeled b). We want to evaluate the XPath query Q, which reads as

descendant::b/following-sibling::*[position() != last()]

over the input context $\langle a, 1, 1\rangle$. We illustrate how this evaluation can be done using Algorithm 6.3: First of all, we have to set up the parse tree



of Q with its 6 proper subexpressions E_1, ..., E_6. Then we compute the context-value tables of the leaf nodes E_1, E_3, E_5 and E_6 in the parse tree, and from the latter two the table for E_4. By combining E_3 and E_4, we obtain E_2, which is in turn needed for computing Q. The tables[8] for E_1, E_2, E_3 and Q are shown in Figure 6. Moreover,

$$\begin{aligned}
\mathcal{E}_\uparrow[\![E_5]\!] &= \{\langle x, k, n, k\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\} \\
\mathcal{E}_\uparrow[\![E_6]\!] &= \{\langle x, k, n, n\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\} \\
\mathcal{E}_\uparrow[\![E_4]\!] &= \{\langle x, k, n, k \neq n\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}
\end{aligned}$$

[8]The k and n columns have been omitted. Full tables are obtained by computing the Cartesian product of each table with $\{\langle k, n\rangle \mid 1 \leq k \leq n \leq |\text{dom}|\}$. This kind of restriction to the "relevant context" will be put on a formal basis in Section 8.

$\mathcal{E}_\uparrow[\![Q]\!]$	
x	val
r	$\{b_2, b_3\}$
a	$\{b_2, b_3\}$
b_1	$\{\,\}$
b_2	$\{\,\}$
b_3	$\{\,\}$
b_4	$\{\,\}$

$\mathcal{E}_\uparrow[\![E_1]\!]$	
x	val
r	$\{b_1, b_2, b_3, b_4\}$
a	$\{b_1, b_2, b_3, b_4\}$
b_1	$\{\,\}$
b_2	$\{\,\}$
b_3	$\{\,\}$
b_4	$\{\,\}$

$\mathcal{E}_\uparrow[\![E_2]\!]$	
x	val
r	$\{\,\}$
a	$\{\,\}$
b_1	$\{b_2, b_3\}$
b_2	$\{b_3\}$
b_3	$\{\,\}$
b_4	$\{\,\}$

$\mathcal{E}_\uparrow[\![E_3]\!]$	
x	val
r	$\{\,\}$
a	$\{\,\}$
b_1	$\{b_2, b_3, b_4\}$
b_2	$\{b_3, b_4\}$
b_3	$\{b_4\}$
b_4	$\{\,\}$

Fig. 6. Context-value tables of Example 6.4.

The most interesting step is the computation of $\mathcal{E}_\uparrow[\![E_2]\!]$ from the tables for E_3 and E_4. For instance, consider $\langle b_1, k, n, \{b_2, b_3, b_4\}\rangle \in \mathcal{E}_\uparrow[\![E_3]\!]$. b_2 is the first, b_3 the second, and b_4 the third of the three siblings following b_1. Thus, only for b_2 and b_3 is the condition E_2 (requiring that the position in set $\{b_2, b_3, b_4\}$ is different from the size of the set, three) satisfied. Thus, we obtain the tuple $\langle b_1, k, n, \{b_2, b_3\}\rangle$ which we add to $\mathcal{E}_\uparrow[\![E_2]\!]$.

We can read out the final result $\{b_2, b_3\}$ from the context-value table of Q. □

Remark 6.5. An intuition for the Context-value Table Principle and Algorithm 6.3 can also be gained from the nice fact that every *acyclic conjunctive query* can be evaluated in polynomial time [Yannakakis 1981]. Now, if we assume that we have each of the operations readily pre-computed as a relation, each XPath query can be viewed as an acyclic conjunctive query over these relations, and Algorithm 6.3 is a reformulation of Yannakakis' Algorithm on such queries (where context-value tables are intermediate join results). However, this intuition fails in general because computed XPath values (even numbers) take space polynomial in the size of the input, and the relations of arithmetical operators or certain string functions would be of exponential size. Thus, this intuition only works for certain fragments of XPath. □

Theorem 6.6. *XPath can be evaluated bottom-up in polynomial time (combined complexity). More precisely, for an XML document D and an XPath query Q, the bottom-up algorithm 6.3 works in time $O(|D|^5 \cdot |Q|^2)$ and space $O(|D|^4 \cdot |Q|^2)$.*

Proof. Let $|Q|$ be the size of the query and $|D|$ be the size of the data. During the bottom-up computation of a query Q using Algorithm 6.3, $O(|Q|)$ relations ("context-value tables") are created. All relations have a functional dependency from the context (columns one to three) to the value (column four). The size of each relation is $O(|D|^3)$ times the maximum size of such values. The size of bool relations is bounded by $O(|D|^3)$ and the size of nset relations by $O(|D|^4)$.

Numbers and strings computable in XPath are of size $O(|D| \cdot |Q|)$: "concat" on strings and arithmetic multiplication on numbers are the most costly operations (w.r.t. size increase of values) on strings and numbers[9]. Here, the lengths of the

[9]For the conversion from a node set to a string or number, only the first node in the set is chosen. Of the string functions, only "concat" may produce a string longer than the input strings.

argument values add up such that we get to sizes $O(|D| \cdot |Q|)$ at worst, even in the relation representing the "top" expression Q itself.

The overall space bound of $O(|D|^4 \cdot |Q|^2)$ follows. Note that no significant additional amount of space is required for intermediate computations.

Let each context-value table be stored as a three-dimensional array, such that we can find the value for a given context $\langle x, k, n\rangle$ in constant time. Given m context-value tables representing expressions $e_1, \ldots, e_m$ and a context $\langle x, k, n\rangle$, any m-ary XPath operation $Op(e_1, \ldots, e_m)$ on context $\langle x, k, n\rangle$ can be evaluated in time $O(|D| \cdot I)$; again, I is the size of the input values and thus $O(|D| \cdot |Q|)$. This is not difficult to verify; it only takes very standard techniques to implement the XPath operations according to the definitions of Table II (sometimes using auxiliary data structures created in a preprocessing step). The most costly operator is $RelOp : \text{nset} \times \text{nset} \to \text{bool}$, and this one also takes the most ingenuity. We assume a pre-computed table

$$\{\langle n_1, n_2\rangle \mid n_1, n_2 \in \text{dom},\ \text{strval}(n_1)\ RelOp\ \text{strval}(n_2)\}$$

that can be used to carry out the operation in time $O(|D^2|)$ given two node sets.

Each of the expression relations can be computed in time $O(|D|^3 \cdot |D|^2 \cdot |Q|)$ at worst when the expression semantics tables of the direct subexpressions are given. (The $|Q|$ factor is due to the size bound on strings and numbers generated during the computation.) Moreover, $O(|Q|)$ such computations are needed in total to evaluate Q. The $O(|D|^5 \cdot |Q|^2)$ time bound follows. □

REMARK 6.7. Note that contexts can also be represented in terms of pairs of a current and a "previous" context node (rather than triples of a node, a position, and a size), which are defined relative to an axis and a node test (which, however, are fixed with the query). For instance, the corresponding ternary context for $\vec{c} = \langle x_0, x\rangle$ w.r.t. axis χ and node test t is $\langle x, \text{idx}_\chi(x, Y), |Y|\rangle$, where $Y = \{y \mid x_0 \chi y,\ y \in T(t)\}$. Thus, position and size values can be recovered on demand.

Through this change, it is possible to push down the maximum number of rows in each context-value table from $O(|D|^3)$ to $O(|D|^2)$. We thus obtain an improved worst-case space bound of $O(|D|^3 \cdot |Q|^2)$ and time bound of $O(|D|^4 \cdot |Q|^2)$ for XPath query evaluation by a bottom-up algorithm. Actually, such a restriction to those context triples that can possibly be generated by the pairs of previous/current context node (w.r.t. some axis χ and node test t) is implicit in the top-down algorithm to be presented in Section 7. In particular, the improved complexity bounds that we have just mentioned are exactly the ones that we will get in Theorem 7.5. □

7. TOP-DOWN EVALUATION OF XPATH

In the previous section, we obtained a bottom-up semantics definition which led to a polynomial-time query evaluation algorithm for XPath. Despite this favorable complexity bound, this algorithm is still not practical, as usually many irrelevant intermediate results are computed to fill the context-value tables which are not used later on. Next, building on the context-value table principle of Section 6, we

The "translate" function of [World Wide Web Consortium 1999], for instance, does not allow for arbitrary but just single-character replacement, e.g. for case-conversion purposes.

(* absolute location paths *)
$\mathcal{S}_\downarrow[\![/\pi]\!](X_1,\ldots,X_k)$:=
  $\mathcal{S}_\downarrow[\![\pi]\!](\underbrace{\{\text{root}\},\ldots,\{\text{root}\}}_{k \text{ times}})$

(* composition of location paths *)
$\mathcal{S}_\downarrow[\![\pi_1/\pi_2]\!](X_1,\ldots,X_k)$:=
  $\mathcal{S}_\downarrow[\![\pi_2]\!](\mathcal{S}_\downarrow[\![\pi_1]\!](X_1,\ldots,X_k))$

(* "disjunction" of location paths *)
$\mathcal{S}_\downarrow[\![\pi_1 \mid \pi_2]\!](X_1,\ldots,X_k)$:=
  $\mathcal{S}_\downarrow[\![\pi_1]\!](X_1,\ldots,X_k) \cup^{\langle\rangle}$
  $\mathcal{S}_\downarrow[\![\pi_2]\!](X_1,\ldots,X_k)$

(* location steps *)
$\mathcal{S}_\downarrow[\![\chi{::}t[e_1]\cdots[e_m]]\!](X_1,\ldots,X_k)$:=
begin
  $S := \{\langle x,y\rangle \mid x \in \bigcup_{i=1}^{k} X_i,\ x\,\chi\,y, \text{ and } y \in T(t)\}$;
  for each $1 \le i \le m$ (in ascending order) **do**
  begin
    for each x, let $S_x = \{z \mid \langle x,z\rangle \in S\}$;
    for $\langle x,y\rangle \in S$,
      let $Ct_S(x,y) = \langle y,\ \text{idx}_\chi(y,S_x),\ |S_x|\rangle$;
    $T := \{Ct_S(x,y) \mid \langle x,y\rangle \in S\}$;
    let $T = \{t_1,\ldots,t_l\}$; (* we fix some order on T *)
    $\langle r_1,\ldots,r_l\rangle := \mathcal{E}_\downarrow[\![e_i]\!](t_1,\ldots,t_l)$;
    $S := \{\langle x,y\rangle \in S \mid \exists i : t_i = Ct_S(x,y) \wedge r_i \text{ is true}\}$;
  end;
  for each $1 \le i \le k$ **do**
    $R_i := \{y \mid \langle x,y\rangle \in S, x \in X_i\}$;
  return $\langle R_1,\ldots,R_k\rangle$;
end;

Fig. 7. Top-down evaluation of location paths.

develop a top-down algorithm based on vector computation for which the favorable (worst-case) complexity bound carries over but in which the computation of a large number of irrelevant results is avoided.

Given an m-ary operation $Op : D^m \to D$, its vectorized version $Op^{\langle\rangle} : (D^k)^m \to D^k$ is defined as

$$Op^{\langle\rangle}(\langle x_{1,1},\ldots,x_{1,k}\rangle,\ldots,\langle x_{m,1},\ldots,x_{m,k}\rangle) :=$$
$$\langle Op(x_{1,1},\ldots,x_{m,1}),\ldots,Op(x_{1,k},\ldots,x_{m,k})\rangle$$

For instance, $\langle X_1,\ldots,X_k\rangle \cup^{\langle\rangle} \langle Y_1,\ldots,Y_k\rangle := \langle X_1 \cup Y_1,\ \ldots,X_k \cup Y_k\rangle$. Let

$$\mathcal{S}_\downarrow :\ \text{LocationPath} \ \to\ \text{List}(2^{\text{dom}}) \ \to\ \text{List}(2^{\text{dom}})$$

be the auxiliary semantics function for location paths defined in Figure 7. We basically distinguish the same cases (related to location paths) as for the bottom-up semantics $\mathcal{E}_\uparrow[\![\pi]\!]$. Given a location path π and a list $\langle X_1,\ldots,X_k\rangle$ of node sets, $\mathcal{S}_\downarrow$ determines a list $\langle Y_1,\ldots,Y_k\rangle$ of node sets, s.t. for every $i \in \{1,\ldots,k\}$, the nodes reachable from the context nodes in X_i via the location path π are precisely the nodes in Y_i. $\mathcal{S}_\downarrow[\![\pi]\!]$ can be obtained from the relations $\mathcal{E}_\uparrow[\![\pi]\!]$ as follows. A node y is in Y_i iff there is an $x \in X_i$ and some p,s such that $\langle x,p,s,y\rangle \in \mathcal{E}_\uparrow[\![\pi]\!]$.

Definition 7.1. The semantics function $\mathcal{E}_\downarrow$ for arbitrary XPath expressions is of the following type:

$$\mathcal{E}_\downarrow :\ \text{XPathExpression} \ \to\ \text{List}(\mathbf{C}) \ \to\ \text{List}(\text{XPathType})$$

Given an XPath expression e and a list $(\vec{c}_1,\ldots,\vec{c}_l)$ of contexts, $\mathcal{E}_\downarrow$ determines a list $\langle r_1,\ldots,r_l\rangle$ of results of one of the XPath types number, string, boolean, or node set. $\mathcal{E}_\downarrow$ is defined as

$$\mathcal{E}_\downarrow[\![\pi]\!](\langle x_1,k_1,n_1\rangle,\ \ldots,\ \langle x_l,k_l,n_l\rangle) := \mathcal{S}_\downarrow[\![\pi]\!](\{x_1\},\ldots,\{x_l\})$$
$$\mathcal{E}_\downarrow[\![\text{position}()]\!](\langle x_1,k_1,n_1\rangle,\ \ldots,\ \langle x_l,k_l,n_l\rangle) := \langle k_1,\ldots,k_l\rangle$$

$\mathcal{E}_\downarrow[\![\text{last}()]\!](\langle x_1, k_1, n_1\rangle, \ldots, \langle x_l, k_l, n_l\rangle) := \langle n_1, \ldots, n_l\rangle$

$\mathcal{E}_\downarrow[\![\text{string}()]\!](\langle x_1, k_1, n_1\rangle, \ldots, \langle x_l, k_l, n_l\rangle) := \langle \text{strval}(x_1), \ldots, \text{strval}(x_l)\rangle$

$\mathcal{E}_\downarrow[\![\text{number}()]\!](\langle x_1, k_1, n_1\rangle, \ldots, \langle x_l, k_l, n_l\rangle) :=$
$\langle \text{to\_number}(\text{strval}(x_1)), \ldots, \text{to\_number}(\text{strval}(x_l))\rangle$

and

$$\mathcal{E}_\downarrow[\![Op(e_1, \ldots, e_m)]\!](\vec{c}_1, \ldots, \vec{c}_l) := \mathcal{F}[\![Op]\!]^{\langle\rangle}(\mathcal{E}_\downarrow[\![e_1]\!](\vec{c}_1, \ldots, \vec{c}_l), \ldots, \mathcal{E}_\downarrow[\![e_m]\!](\vec{c}_1, \ldots, \vec{c}_l))$$

for the remaining kinds of expressions. □

EXAMPLE 7.2. Consider the XPath query

/descendant::a[count(descendant::b/child::c) + position() < last()]/child::d

Let $L = \langle\langle y_1, 1, l\rangle, \ldots, \langle y_l, l, l\rangle\rangle$, where the y_i are those nodes reachable from the root node through the descendant axis and which are labeled "a". The query is evaluated top-down as

$$\mathcal{S}_\downarrow[\![\text{child::d}]\!](\mathcal{S}_\downarrow[\![\text{descendant::a}[e]]\!](\{root\}))$$

where e is defined as $e :=$ count(descendant::b/child::c) + position() < last().

Moreover, $\mathcal{E}_\downarrow[\![e]\!](L)$ is computed as

$$\mathcal{F}[\![\text{count}]\!]^{\langle\rangle}(X) +^{\langle\rangle} \mathcal{E}_\downarrow[\![\text{position}()]\!](L) <^{\langle\rangle} \mathcal{E}_\downarrow[\![\text{last}()]\!](L)$$

and

$$X = \mathcal{S}_\downarrow[\![\text{child::c}]\!](\mathcal{S}_\downarrow[\![\text{descendant::b}]\!](\{y_1\}, \ldots, \{y_l\})).$$

Note that the arity of the tuples used to compute the outermost location path is one, while it is l for e. □

EXAMPLE 7.3. Given the query Q, document DOC(4), and context $\langle a, 1, 1\rangle$ of Example 6.4, we evaluate Q as $\mathcal{E}_\downarrow[\![Q]\!](\langle a, 1, 1\rangle) = \mathcal{S}_\downarrow[\![E_2]\!](\mathcal{S}_\downarrow[\![\text{descendant::b}]\!](\{a\}))$. Again, E_2 is the subexpression

following-sibling::*[position() != last()].

First, we obtain $\mathcal{S}_\downarrow[\![\text{descendant::b}]\!](\{a\}) = \langle\{b_1, b_2, b_3, b_4\}\rangle$. For the computation of the location step $\mathcal{S}_\downarrow[\![E_2]\!](\langle\{b_1, b_2, b_3, b_4\}\rangle)$, we proceed as described in the algorithm of Figure 7. We initially obtain the set

$$S = \{\langle b_1, b_2\rangle, \langle b_1, b_3\rangle, \langle b_1, b_4\rangle, \langle b_2, b_3\rangle, \langle b_2, b_4\rangle, \langle b_3, b_4\rangle\}$$

and the contexts $\vec{t} = \langle\langle b_2, 1, 3\rangle, \langle b_3, 2, 3\rangle, \langle b_4, 3, 3\rangle, \langle b_3, 1, 2\rangle, \langle b_4, 2, 2\rangle, \langle b_4, 1, 1\rangle\rangle$.

The check of condition E_4 returns the filter

$$\vec{r} = \langle \text{true}, \text{true}, \text{false}, \text{true}, \text{false}, \text{false}\rangle.$$

which is applied to S to obtain $S = \{\langle b_1, b_2\rangle, \langle b_1, b_3\rangle, \langle b_2, b_3\rangle\}$. Thus, the query returns $\langle\{b_2, b_3\}\rangle$. □

The correctness of the top-down semantics follows immediately from the corresponding result in the bottom-up case and from the definition of $\mathcal{S}_\downarrow$ and $\mathcal{E}_\downarrow$.

```
<?xml version="1.0"?>
<a id = "10">
      <b id = "11">
            <c id = "12">21 22</c>
            <c id = "13">23 24</c>
            <d id = "14">100</d>
      </b>
      <b id = "21">
            <c id = "22">11 12</c>
            <d id = "23">13 14</d>
            <d id = "24">100</d>
      </b>
</a>
```

Fig. 8. Sample XML document.

THEOREM 7.4. *(Correctness of $\mathcal{E}_\downarrow$) Let e be an arbitrary XPath expression. Then $\langle v_1, \ldots, v_l\rangle = \mathcal{E}_\downarrow[\![e]\!](\vec{c}_1, \ldots, \vec{c}_l)$ iff $\langle \vec{c}_1, v_1\rangle, \ldots, \langle \vec{c}_l, v_l\rangle \in \mathcal{E}_\uparrow[\![e]\!]$.*

$\mathcal{S}_\downarrow$ and $\mathcal{E}_\downarrow$ can be immediately transformed into function definitions in a top-down algorithm. We thus have to define one evaluation function for each case of the definition of $\mathcal{S}_\downarrow$ and $\mathcal{E}_\downarrow$, respectively. The functions corresponding to the various cases of $\mathcal{S}_\downarrow$ have a location path and a list of node sets of variable length $(X_1, \ldots, X_k)$ as input parameter and return a list $(R_1, \ldots, R_k)$ of node sets of the same length as result. Likewise, the functions corresponding to $\mathcal{E}_\downarrow$ take an arbitrary XPath expression and a list of contexts as input and return a list of XPath values (which can be of type num, str, bool or nset). Moreover, the recursions in the definition of $\mathcal{S}_\downarrow$ and $\mathcal{E}_\downarrow$ correspond to recursive function calls of the respective evaluation functions. Analogously to Theorem 6.6, we get

THEOREM 7.5. *The immediate functional implementation of $\mathcal{E}_\downarrow$ evaluates XPath queries in polynomial time (combined complexity). More precisely, for an XML document D and an XPath query Q, the top-down algorithm based on $\mathcal{E}_\downarrow$ as described above works in time $O(|D|^4 \cdot |Q|^2)$ and space $O(|D|^3 \cdot |Q|^2)$.*

Finally, note that using arguments relating the top-down method of this section with (join) optimization techniques in relational databases, one may argue that the context-value table principle is also the basis of the polynomial-time bound of Theorem 7.5.

8. THE ALGORITHM MINCONTEXT

8.1 Preliminaries

We shall illustrate our new algorithm MINCONTEXT by means of the following running example:

EXAMPLE 8.1. Let D be the XML document in Figure 8. Note that every element of this document is uniquely determined by the attribute "id". Hence, in the context of this example, we use the notation x_i to refer to the element whose attribute "id" has the value i. We thus have dom $= \{r, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$, where r is the root node (i.e., the parent of x_{10}).

Now suppose that we want to evaluate the XPath query $Q \equiv$ /descendant::*/descendant::*[position() > last()*0.5 or self::* = 100] over the XML document D for the context $\langle x_{10}, 1, 1\rangle$.

The parse tree $\mathcal{T}$ of Q is depicted in Figure 10. Note that we have replaced "self::* = 100" at E_7 by "string(self::*) = 100" in order to make the type conversion (from node set to string) explicit. The context-value table of each subexpression of Q is depicted in Figure 9. By "x", "p", and "s", we denote context-node, context-position, and context-size. In the last column of each table, we have the result "*val*". In the context-value tables of the subexpressions Q, E_1, E_2, E_3, and E_4, we have omitted the columns for the context-position and the context-size. Analogously to Example 6.4, this is justified by the fact that "p" and "s" are irrelevant for these path expressions. We shall come back to this point in Section 8.2.

In all of the tables in Figure 9, we have omitted some rows, which have no influence on the overall result. Recall that we are evaluating Q for the context $\langle x_{10}, 1, 1\rangle$. Hence, in the table of Q, we only consider the context-node x_{10}. In the tables corresponding to E_1 and E_2, we only consider the root node r, since this is the only interesting context-node when we move up from E_1 to $/\,E_1$. On the other hand, in the tables corresponding to E_3 and E_4, the root node r is omitted, since it cannot be reached by the preceding location step "/descendant::*". Likewise, for the expressions $E_5, \ldots, E_{14}$, we set up the context-value tables only for those context triples $\langle x, p, s\rangle$ which can be reached by the preceding location steps "/descendant::*/descendant::*". All of these restrictions of the context-value tables are due to the considerations on the *top-down* evaluation according to the semantics function $\mathcal{E}_\downarrow$ from Definition 7.1. In particular, this top-down evaluation guarantees that no context-value table contains more than $|\text{dom}|^2$ entries, corresponding to all possible pairs of a previous and a current context node w.r.t. the axis in the last location step. Without this improvement, we would have to consider up to $|\text{dom}|^3$ possible triples $\langle x, p, s\rangle$ in each context-value table.

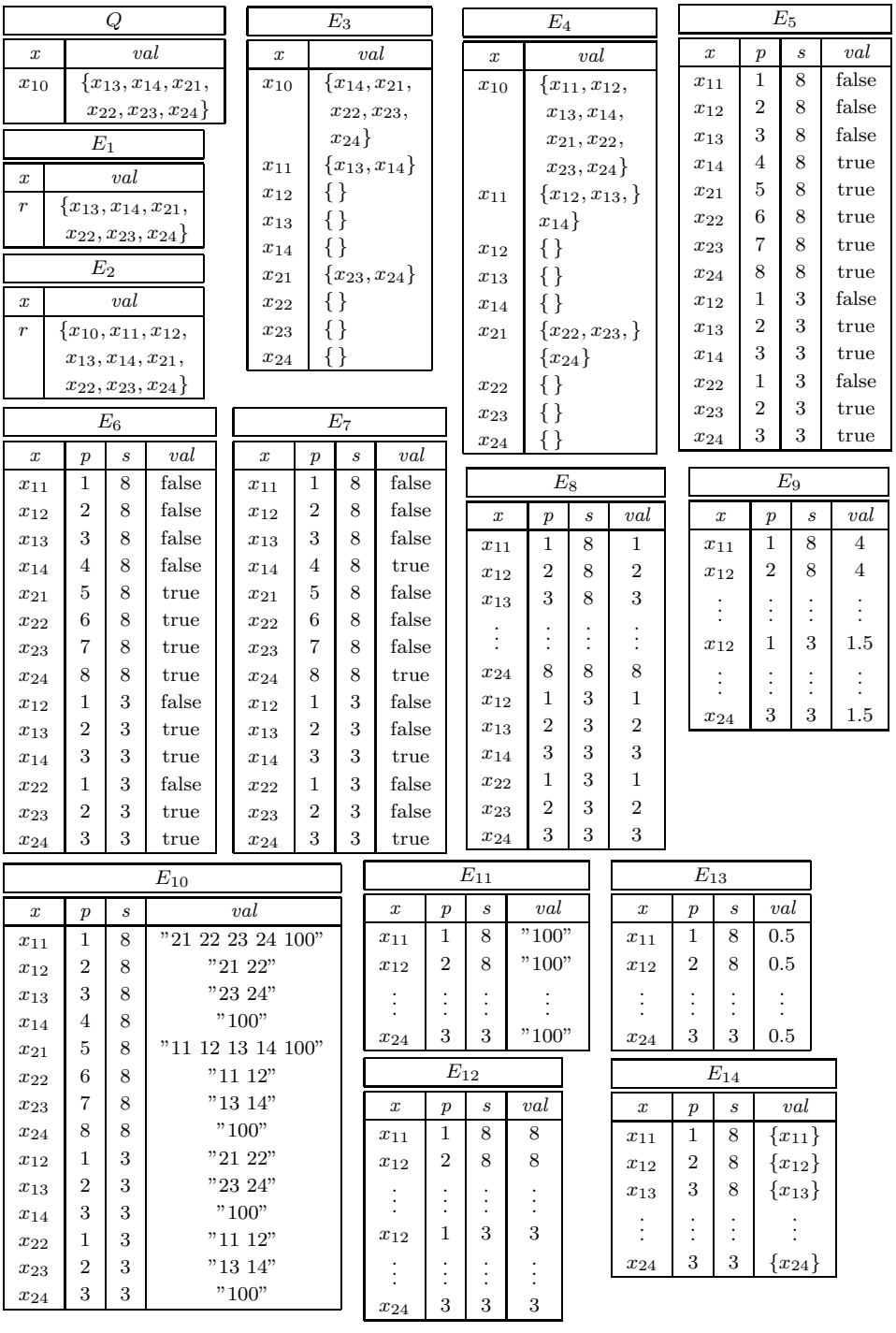
The final result of evaluating Q over D for the context $\langle x_{10}, 1, 1\rangle$ is $\{x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$. It can be read out from the context-value table corresponding to the input XPath expression Q. □

In the context of our running example, by slight abuse of notation, we are writing E_i both to denote subexpressions of the input XPath query Q and nodes in the parse tree $\mathcal{T}$. However, if the distinction between a node N in the parse tree and a subexpression E of the input XPath query does matter, then we shall write $expr(N)$ to denote the XPath expression corresponding to the node N. Conversely, for an expression E, we write $node(E)$ to denote the node in the parse tree corresponding to E. Given a node N in the parse tree, we shall write $table(N)$ to denote the context-value table at the node N. Finally, it is convenient to write "≡" for syntactic equality.

8.2 The Main Ideas

The primary goal of our new algorithm MinContext is to keep the context information that has to be considered at each stage as small as possible. This is achieved by combining several ideas:

Restriction to the relevant context. Suppose that we want to evaluate an



Q

x	val
x_{10}	$\{x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$

E_1

x	val
r	$\{x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$

E_2

x	val
r	$\{x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$

E_3

x	val
x_{10}	$\{x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$
x_{11}	$\{x_{13}, x_{14}\}$
x_{12}	$\{\,\}$
x_{13}	$\{\,\}$
x_{14}	$\{\,\}$
x_{21}	$\{x_{23}, x_{24}\}$
x_{22}	$\{\,\}$
x_{23}	$\{\,\}$
x_{24}	$\{\,\}$

E_4

x	val
x_{10}	$\{x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$
x_{11}	$\{x_{12}, x_{13}, \}$ $x_{14}\}$
x_{12}	$\{\,\}$
x_{13}	$\{\,\}$
x_{14}	$\{\,\}$
x_{21}	$\{x_{22}, x_{23}, \}$ $\{x_{24}\}$
x_{22}	$\{\,\}$
x_{23}	$\{\,\}$
x_{24}	$\{\,\}$

E_5

x	p	s	val
x_{11}	1	8	false
x_{12}	2	8	false
x_{13}	3	8	false
x_{14}	4	8	true
x_{21}	5	8	true
x_{22}	6	8	true
x_{23}	7	8	true
x_{24}	8	8	true
x_{12}	1	3	false
x_{13}	2	3	true
x_{14}	3	3	true
x_{22}	1	3	false
x_{23}	2	3	true
x_{24}	3	3	true

E_6

x	p	s	val
x_{11}	1	8	false
x_{12}	2	8	false
x_{13}	3	8	false
x_{14}	4	8	false
x_{21}	5	8	true
x_{22}	6	8	true
x_{23}	7	8	true
x_{24}	8	8	true
x_{12}	1	3	false
x_{13}	2	3	true
x_{14}	3	3	true
x_{22}	1	3	false
x_{23}	2	3	true
x_{24}	3	3	true

E_7

x	p	s	val
x_{11}	1	8	false
x_{12}	2	8	false
x_{13}	3	8	false
x_{14}	4	8	true
x_{21}	5	8	false
x_{22}	6	8	false
x_{23}	7	8	false
x_{24}	8	8	true
x_{12}	1	3	false
x_{13}	2	3	false
x_{14}	3	3	true
x_{22}	1	3	false
x_{23}	2	3	false
x_{24}	3	3	true

E_8

x	p	s	val
x_{11}	1	8	1
x_{12}	2	8	2
x_{13}	3	8	3
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{24}	8	8	8
x_{12}	1	3	1
x_{13}	2	3	2
x_{14}	3	3	3
x_{22}	1	3	1
x_{23}	2	3	2
x_{24}	3	3	3

E_9

x	p	s	val
x_{11}	1	8	4
x_{12}	2	8	4
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{12}	1	3	1.5
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{24}	3	3	1.5

E_{10}

x	p	s	val
x_{11}	1	8	"21 22 23 24 100"
x_{12}	2	8	"21 22"
x_{13}	3	8	"23 24"
x_{14}	4	8	"100"
x_{21}	5	8	"11 12 13 14 100"
x_{22}	6	8	"11 12"
x_{23}	7	8	"13 14"
x_{24}	8	8	"100"
x_{12}	1	3	"21 22"
x_{13}	2	3	"23 24"
x_{14}	3	3	"100"
x_{22}	1	3	"11 12"
x_{23}	2	3	"13 14"
x_{24}	3	3	"100"

E_{11}

x	p	s	val
x_{11}	1	8	"100"
x_{12}	2	8	"100"
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{24}	3	3	"100"

E_{12}

x	p	s	val
x_{11}	1	8	8
x_{12}	2	8	8
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{12}	1	3	3
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{24}	3	3	3

E_{13}

x	p	s	val
x_{11}	1	8	0.5
x_{12}	2	8	0.5
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{24}	3	3	0.5

E_{14}

x	p	s	val
x_{11}	1	8	$\{x_{11}\}$
x_{12}	2	8	$\{x_{12}\}$
x_{13}	3	8	$\{x_{13}\}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{24}	3	3	$\{x_{24}\}$

Fig. 9. Context-value tables of Example 8.1.

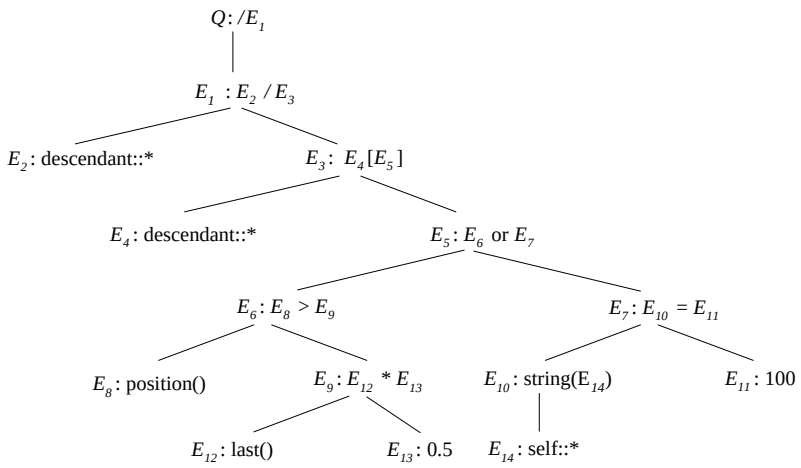


Fig. 10. Parse tree $\mathcal{T}$ of Q in Example 8.1.

XPath expression Q via the context-value table principle. Then we have to compute a table of up to $|\text{dom}|^2$ entries for each node in the parse tree of Q. Recall that this is already an improved bound due to the top-down evaluation via the semantics function $\mathcal{E}_{\downarrow}$. However, in many cases, the result of a subexpression depends solely on parts of the context information. Hence, we can restrict the context-value table at every node N in the parse tree to the "*relevant context*" $Relev(N) \subseteq \{\text{'cn'}, \text{'cp'}, \text{'cs'}\}$, which can be computed by a single bottom-up traversal of the parse tree as follows:

—*Base cases.* If N is a leaf node of the parse tree, then we have to distinguish all possible cases concerning the form of the subexpression $expr(N)$ corresponding to N, namely: If $expr(N)$ is a constant or an expression of the form "true()" or "false()", then we set $Relev(N) := \emptyset$. In case of $expr(N) \equiv$ position() or $expr(N) \equiv$ last(), we set $Relev(N) := \{\text{'cp'}\}$ or $Relev(N) := \{\text{'cs'}\}$, respectively. Finally, if $expr(N)$ is a location step or a parameterless XPath core library function that refers to the context-node (like string(), number(), etc.), then we set $Relev(N) := \{\text{'cn'}\}$.

—*Compound expressions.* If an inner node N of the parse tree corresponds to a location step within a location path, then we set $Relev(N) := \{\text{'cn'}\}$. In all other cases, let $\{N_1, \ldots, N_k\}$ denote the set of child nodes of N. Then we set $Relev(N) := \bigcup_{i=1}^{k} Relev(N_i)$.

$Relev(N)$ depends on the XPath query Q only (but not on the XML-document). Obviously, the computation of all these sets $Relev(N)$ can be done in time $O(|Q|)$.

Example 8.2. By a bottom-up traversal of the parse tree $\mathcal{T}$ in Figure 10, we get the following sets of relevant contexts:

$Relev(E_2) = \{\text{'cn'}\}$ $\quad Relev(E_4) = \{\text{'cn'}\}$ $\quad Relev(E_8) = \{\text{'cp'}\}$
$Relev(E_{12}) = \{\text{'cs'}\}$ $\quad Relev(E_{13}) = \{\,\}$ $\quad Relev(E_{14}) = \{\text{'cn'}\}$
$Relev(E_{11}) = \{\,\}$ $\quad Relev(E_9) = Relev(E_{12}) \cup Relev(E_{13}) = \{\text{'cs'}\}$

$Relev(E_6) = Relev(E_8) \cup Relev(E_9) = \{\text{'cp'}, \text{'cs'}\}$
$Relev(E_{10}) = Relev(E_{14}) = \{\text{'cn'}\}$
$Relev(E_7) = Relev(E_{10}) \cup Relev(E_{11}) = \{\text{'cn'}\}$
$Relev(E_5) = Relev(E_6) \cup Relev(E_7) = \{\text{'cn'}, \text{'cp'}, \text{'cs'}\}$
$Relev(E_3) = \{\text{'cn'}\}$ $\quad Relev(E_1) = \{\text{'cn'}\}$ $\quad Relev(Q) = \{\text{'cn'}\}$

Note that Q, E_1, and E_3 correspond to location steps. That is why the child nodes of these nodes in the parse tree play no role in computing $Relev(Q)$, $Relev(E_1)$, and $Relev(E_3)$.

For the nodes Q, $E_1 \ldots E_4$, the context-value tables in Figure 9 have already been reduced to the relevant context. For E_5, no reduction is possible, since we have $Relev(E_5) = \{\text{'cn'}, \text{'cp'}, \text{'cs'}\}$. For the remaining nodes, the reduced context-value tables are displayed in Figure 11. □

E_6

p	s	val
1	8	false
2	8	false
3	8	false
4	8	false
5	8	true
6	8	true
7	8	true
8	8	true
1	3	false
2	3	true
3	3	true

E_7

x	val
x_{11}	false
x_{12}	false
x_{13}	false
x_{14}	true
x_{21}	false
x_{22}	false
x_{23}	false
x_{24}	true

E_{11}

val
"100"

E_8

p	val
1	1
2	2
3	3
⋮	⋮
8	8

E_9

s	val
8	4
3	1.5

E_{10}

x	val
x_{11}	"21 22 23 24 100"
x_{12}	"21 22"
x_{13}	"23 24"
x_{14}	"100"
x_{21}	"11 12 13 14 100"
x_{22}	"11 12"
x_{23}	"13 14"
x_{24}	"100"

E_{12}

s	val
8	8
3	3

E_{13}

val
0.5

E_{14}

x	val
x_{11}	$\{x_{11}\}$
x_{12}	$\{x_{12}\}$
x_{13}	$\{x_{13}\}$
⋮	⋮
x_{24}	$\{x_{24}\}$

Fig. 11. Restriction to the relevant context.

Special treatment of location paths on the outermost level. (i.e., location paths that do not occur inside another XPath expression). Note that the context-value table algorithm computes a table of size $O(|\text{dom}|^2)$ for all location steps of an input location path (according to the semantics function $\mathcal{S}_\downarrow$ in Figure 7). This is due to the fact that we compute for every possible context-node x the resulting node set. However, at no stage in the computation, we are really interested in the whole information as to which next node $x_j \in \text{dom}$ can be reached from which previous node $x_i \in \text{dom}$. Instead, it suffices to know the *set* of all nodes $x_j \in \text{dom}$ that can be reached from *any* of the previous nodes $x_i \in \text{dom}$. Hence, the results of location steps on the outermost level should be treated as a subset $\subseteq \text{dom}$ rather than as a relation $\subseteq \text{dom} \times 2^{\text{dom}}$. Of course, the final result now has to be read out from the context-value table corresponding to the last location step (rather than from the context-value table of the root node of the parse tree).

Example 8.3. The XPath query Q from Example 8.1 has in fact a location path on the outermost level. Hence, the 2-dimensional context-value tables of

the location paths "/descendant::*/descendant::*[...]" (at $node(Q)$ in the parse tree), "descendant::*/descendant::*[...]" (at $node(E_1)$), and "descendant::*[...] (at $node(E_3)$) can be replaced by the following node sets (or, equivalently, the 1-dimensional tables): $X = \{r\}$ at $node(Q)$ (i.e., the only node selected by "/" is the root node r), moreover, $Y = \{x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$ at $node(E_1)$ (i.e., the nodes selected by "descendant::*" when starting from r), and, finally, $Z = \{x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$ at $node(E_3)$ (i.e., the nodes selected by "descendant::[E_5]" when starting from any node in Y). Then the final result of Q is the node set Z corresponding to the node E_3 in the parse tree. □

Treating position and size in a loop. The central idea of the context-value table principle is the *simultaneous* evaluation of each subexpression *for all possible contexts* in a single table. However, a close inspection of the various kinds of expressions that have to be evaluated (cf. Tables II and IV) reveals that such a simultaneous evaluation for all possible contexts is only necessary (in order to avoid exponential time complexity) for the context-node x. In contrast, for the context-position and/or context-size, a *loop over all possible values* $\langle p, s\rangle$ leads to a significant improvement of the space complexity without any deterioration of the time complexity. Hence, the evaluation of any predicate p should be done as follows: First the subtree in the parse tree corresponding to the predicate p is traversed so as to evaluate all subexpressions of p that do not depend on the (current) context-position and/or context-size. Then the evaluation of the predicate p for the complete context (possibly involving position and/or size) is done in a loop over all possible values $\langle x, p, s\rangle$.

Example 8.4. Recall the query Q from Section 8.1. After the location steps "/descendant::*/descendant::*", we are left with the set $X = \{x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$ of candidates that may possibly be selected by $Q \equiv$ /descendant::*/descendant::*[E_5]. Now X has to be restricted in the following way to the set X' of those nodes for which E_5 evaluates to "true".

(1) First, we traverse the subtree of the parse tree rooted at E_5 top-down and evaluate those parts which are independent of the value of p and s at $node(E_5)$. We thus set up the context-value tables of E_7, E_{10}, E_{11}, E_{13}, and E_{14} as in Figure 11.

(2) Then, in a loop over all $O(|\text{dom}|^2)$ pairs of previous/current context-nodes (w.r.t. the "descendant"-axis), we compute the set of those nodes $X' \subseteq X$, for which the predicate E_5 is true, i.e.: $X' := \{x \in X \mid (\exists p)(\exists s)$ s.t. E_5 evaluates to "true" for the context $\langle x, p, s\rangle\}$. Of course, this comes down to checking all the rows of the context-table of E_5 (and also of E_6, E_8, E_9, and E_{12}) However, in contrast to Figure 9, we do not set up the entire tables at once. Instead, we treat these contexts $\langle x, p, s\rangle$ in a loop, e.g.: for $\langle x, p, s\rangle = \langle x_{23}, 7, 8\rangle$ we compute the rows of E_5, E_6, E_8, E_9, and E_{12} for $p = 7$ and $s = 8$ only. Moreover, we look up the row of E_7 for the context-node x_{23}. We thus get the overall value "true" of E_5 for this single context $\langle x_{23}, 7, 8\rangle$. Hence, x_{23} is added to X'. □

8.3 Procedures of the Algorithm MinContext

The MinContext algorithm consists of three principal procedures, namely *eval_outermost_locpath*, *eval_by_cnode_only*, and *eval_single_context*. They are briefly explained below. In the Appendix A, pseudocode presentations will be provided.

—The procedure *eval_outermost_locpath* evaluates an input expression e in case that e is a location path. It takes a node N in the parse tree and a node set $X \subseteq \text{dom}$ as input and returns the set Y of nodes that can be reached via the path e from any context-node $x \in X$.

—The procedure *eval_by_cnode_only* takes a node N in the parse tree and a set X of possible context-nodes as input. It does not return a result value as such. However, for every node M in the subtree rooted at N, it computes $table(M)$, provided that $expr(M)$ does not depend on the (current) context-position/size.

—The procedure *eval_single_context* evaluates arbitrary XPath expressions for a single context $\langle x, p, s\rangle$. It takes a node N in the parse tree and a context $\langle x, p, s\rangle$ as input and returns the result value of $expr(N)$ for this context. The procedure *eval_single_context* may only be called after *eval_by_cnode_only* has been called for the node N.

In the Appendix A, we shall also give the pseudocode of the auxiliary procedure *eval_inner_locpath*, which is called inside *eval_by_cnode_only* in case of a location path inside a predicate. Note that in all of these procedures, the parse tree of an input query and the context value tables (i.e., $table(N)$ for nodes N in the parse tree) are treated as global variables in order to increase the readability.

On the top-level, our XPath evaluation method works as follows:

Algorithm 8.5. (MinContext)
Input: XPath query Q, XML document D, context $\langle x, p, s\rangle$;
Output: Result value of Q over D for the context $\langle x, p, s\rangle$;
Method:

Let $\mathcal{T}$ be the parse tree of Q;
Let R be the root node of $\mathcal{T}$;
if Q is a location path **then return** *eval_outermost_locpath*$(R, \{x\})$;
else
begin
  eval_by_cnode_only$(R, \{x\})$;
  return *eval_single_context*$(R, \langle x, p, s\rangle)$;
end; □

The MinContext algorithm will be put to work in a detailed example in Section 11. Below, we show that the heuristics introduced here help to reduce the worst-case complexity:

Theorem 8.6. *The* MinContext *algorithm evaluates arbitrary XPath queries in time* $O(|D|^4 * |Q|^2)$ *and space* $O(|D|^2 * |Q|^2)$*, where* $|D|$ *is the size of the XML-document and* $|Q|$ *is the size of the XPath query.*

Proof. As far as the *space complexity* is concerned, note that we only set up context-value tables where the number of possible contexts is bounded by $|\text{dom}| <$

$|D|$ (namely for nodes N in the parse tree with $Relev(N) \subseteq \{\text{'cn'}\}$). Of course, there are at most $|Q|$ context-value tables required. Moreover, as was shown in the proof of Theorem 6.6, the size of the result value of any subexpression e of Q for any context $\vec{c}$ is restricted by $O(|D| * |Q|)$. We thus get the desired bound on the space complexity.

As for the *time complexity*, we evaluate each subexpression e of the input query Q for at most $|\text{dom}|^2$ different contexts (be it in a single context-value table or in a loop over all possible values $\langle x, p, s\rangle$ corresponding to previous/current context-node). In other words, we consider at most $|D|^2 * |Q|$ pairs $(e, \vec{c})$ consisting of a subexpression e of Q and a context $\vec{c}$. Moreover, it was shown in the proof of Theorem 6.6, that the time required for computing each result value is bounded by $O(|D|^2 * |Q|)$. Hence, we indeed end up with the upper bound $O(|D|^4 * |Q|^2)$ on the time complexity. □

9. IMPROVEMENT OF EXISTING XPATH PROCESSORS

9.1 Integrating the CVT-principle

During the evaluation process of some input XPath query Q, the existing XPath processors (in particular, Saxon, Xalan, XT, see [Clark 1999; Kay 2003; Apache Foundation 2004]) repeatedly evaluate subexpressions e of Q for contexts $\vec{c} \in \mathbf{C}$, where $\vec{c}$ is of the form $\vec{c} = \langle x, p, s\rangle$ for some context-node x, context-position p, and context-size s. As was pointed out in Section 2, these systems, in general, do the evaluation of the same subexpression e of Q for the same context $\vec{c} \in \mathbf{C}$ more than once. This is the very reason why their time complexity is, in general, exponential. By incorporating data structures analogous to the context-value tables described in Section 6, multiple evaluations of the same subexpression e of Q for the same context $\vec{c} \in \mathbf{C}$ can be avoided, thus reducing the time complexity to polynomial time. The data structures used for this purpose will be referred to as "data pool". It contains triples of the form $\langle e, \vec{c}, v\rangle$, where e is a subexpression of the input XPath query Q, $\vec{c} \in \mathbf{C}$ is a context, and v is the result value obtained when evaluating e for the context $\vec{c}$. In other terms, $(\vec{c}, v)$ can be considered as a row in the context-value table corresponding to e. Initially, the data pool is empty, i.e., it contains no such triples.

In order to guarantee that no evaluation of the same subexpression e for the same context $\vec{c}$ is done more than once, we have to add two further components to the existing systems, namely a "storage procedure" and a "retrieval procedure". Prior to the evaluation of any subexpression e for any context $\vec{c}$, the retrieval procedure is called, which checks whether a triple $\langle e', \vec{c}', v\rangle$ with $e = e'$ and $\vec{c} = \vec{c}'$ already exists in the data pool. If this is the case, then the result value v of e for the context $\vec{c}$ is returned without further computation. On the other hand, after the evaluation of any subexpression e for any context $\vec{c}$ yielding the result value v, the storage procedure stores the triple $\langle e, \vec{c}, v\rangle$ in the data pool.

Let the basic evaluation step of an existing system be referred to as "atomic-evaluation", which takes an XPath expression e and a context $\vec{c}$ as an input and returns the corresponding result value v. Then this "atomic-evaluation" simply has to be replaced by the following procedure based on the context-value table principle:

ALGORITHM 9.1. (Improved Basic Evaluation Step)
Input: XPath expression e, context $\langle x, p, s\rangle$;
Output: Result value of e for the context $\langle x, p, s\rangle$;
Method:

```
function atomic-evaluation-CVT (e, c⃗)
begin
  if there exists a v, s.t. ⟨e, c⃗, v⟩ is in the data pool then
    return v;                              /* retrieval procedure */
  else
  begin
    v := atomic-evaluation (e, c⃗);         /* basic evaluation step */
    store ⟨e, c⃗, v⟩ in the data pool;      /* storage procedure */
    return v;
  end;
end;
```

9.2 A Polynomial-time Recursive XPath Processor

Recall from Definition 5.1 the formal semantics $[\![e]\!]$ and $P[\![\pi]\!]$ of arbitrary XPath expressions e and of location paths π, respectively. It is straightforward to transform this semantics definition into a recursive XPath evaluation processor. In fact, we just have to define one evaluation function for each case of the definition of $[\![\cdot]\!]$ and $P[\![\cdot]\!]$, respectively. The functions corresponding to the various cases of $[\![\cdot]\!]$ take an arbitrary XPath expression e and a context $\vec{c}$ as input and return an XPath value (which can be of type num, str, bool or nset). Analogously, the functions corresponding to $P[\![\cdot]\!]$ have a location path π and a node $x \in$ dom as input parameter and return a node set. Moreover, the recursions in the definition of $[\![\cdot]\!]$ and $P[\![\cdot]\!]$ correspond precisely to recursive function calls of the respective evaluation functions.

As was pointed out in Section 2, such a recursive algorithm provides a realistic model of the evaluation methods of existing XPath processors like Saxon, Xalan, and XT. Moreover, it also follows from our analysis in Section 2, that the resulting XPath evaluation method requires exponential time in the worst case. However, the situation changes completely, if we modify all these function definitions in the sense of Algorithm 9.1, i.e.: Before any evaluation function corresponding to $[\![\cdot]\!]$ is called recursively with some input $(e, \vec{c})$, we first check whether a triple $\langle e', \vec{c}', v\rangle$ with $e = e'$ and $\vec{c} = \vec{c}'$ already exists in the data pool. If this is the case, then the result value v of e for the context $\vec{c}$ is returned without further computation. Likewise, before an evaluation function corresponding to $P[\![\cdot]\!]$ is called with some input (π, x), we first check whether some triple $\langle e', \vec{c}', v\rangle$ with $\pi = e'$ and $\vec{c}' = \langle x, c_p, c_s\rangle$ for arbitrary context-position c_p and context-size c_s already exists in the data pool.

On the other hand, after the evaluation of an expression e for a context $\vec{c}$ yielding the result value v, we store the triple $\langle e, \vec{c}, v\rangle$ in the data pool. Likewise, for a location path π and context-node x yielding the result value v, we may store all possible triples $\langle \pi, \vec{c}, v\rangle$ with $\vec{c} = \langle x, c_p, c_s\rangle$ in the data pool.

This modification of the evaluation procedure leads to the following favorable complexity result:

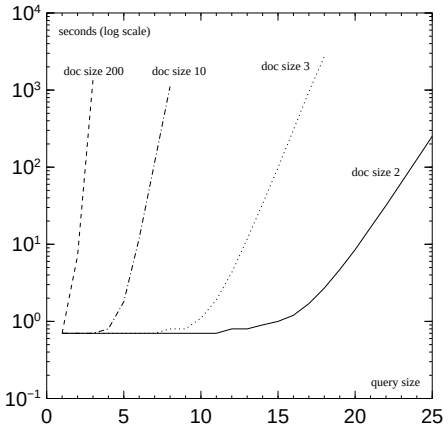


Fig. 12. Repetition of Experiment 3 from Section 2 with Xalan.

THEOREM 9.2. *The functional implementation of* $[\![\cdot]\!]$ *and* $P[\![\cdot]\!]$ *using a data pool, a storage function, and a retrieval function in the way described above evaluates XPath queries in polynomial time (combined complexity).*

PROOF. Let Q denote the input XPath expression and D the XML document over which Q has to be evaluated. Of course, our evaluation procedure only evaluates subexpressions e of Q and there exist at most $O(|Q|)$ of them. Moreover, the number of contexts to be considered for every subexpression is bounded by $O(|D|^3)$. By construction, our XPath evaluation algorithm never calls an evaluation procedure twice for evaluating a given subexpression e of Q for a given context $\vec{c}$. Hence, the total number of (recursive) function calls is bounded by the maximum number of distinct tuples $\langle e, \vec{c}, v\rangle$, i.e., $O(|D|^3 \cdot |Q|)$. It thus only remains to show that the work to be carried out inside every such function (not considering recursive function calls) is polynomially bounded. But this can be easily checked by inspecting all the different cases of the definition of $[\![\cdot]\!]$ and $P[\![\cdot]\!]$, respectively. □

9.3 Experimental Results

We evaluated the approach proposed in this section by partially integrating a data pool into Xalan. It turned out that even minor changes can lead to a dramatical improvement of the runtime behavior. Specifically, the data pool was only integrated for the evaluation of XPath functions (but not for location paths, whose handling by Xalan is basically untraceable). Then the XPath queries from Experiment 3 in Section 2 (with nested paths and calls of the count-function) were evaluated with the "original" Xalan and with the slightly modified one, respectively. Similarly as in Figure 3 (Experiment 3), the exponential time behavior of Xalan is immediately clear in Figure 12.

In contrast, in Table V, we see that the time required to process the documents of size 10 and 200 when a data pool is used. Actually, in case of document size 10, the effort for the XPath evaluation itself is so small compared with the overhead (e.g. for the Java virtual machine) that we get an (almost) constant time behavior. On the other hand, for document size 200, the last column of the table shows that

$\|Q\|$	Xalan classic		Xalan+data pool	
	10	200	10	200
1	0.7	0.7	0.7	2.5
2	0.7	7.1	0.7	4.8
3	0.7	1343.0	0.7	7.1
4	0.8	-	0.7	9.3
5	1.8	-	0.7	11.6
6	12.0	-	0.7	14.0
7	116.0	-	0.7	16.2
8	1115.0	-	0.7	18.5

Table V. Exponential speed-up of Xalan via a data pool for XPath functions (document sizes 10 and 200; times in seconds).

the computation time increases (almost) *linearly* with the size of the query.

10. LINEAR-TIME FRAGMENTS OF XPATH

10.1 Core XPath

In this section, we define a fragment of XPath (called Core XPath) which constitutes a clean logical core of XPath (cf. [Gottlob and Koch 2002]). The only objects that are manipulated in this language are sets of nodes (i.e., there are no arithmetical or string operations). Besides from these restrictions, the full power of location paths is supported, and so is the matching of such paths in condition predicates (with an "exists" semantics), and the closure of such condition expressions with respect to boolean operations "and", "or", and "not".

We define a mapping of each query in this language to a simple algebra over the set operations $\cap$, $\cup$, '$-$', χ (the axis functions from Definition 3.1), and an operation $\frac{\text{dom}}{\text{root}}(S) := \{x \in \text{dom} \mid \text{root} \in S\}$, i.e. $\frac{\text{dom}}{\text{root}}(S)$ is dom if root $\in S$ and $\emptyset$ otherwise.

Note that each XPath axis has a natural *inverse*: self^{-1} = self, child^{-1} = parent, descendant^{-1} = ancestor, descendant-or-self^{-1} = ancestor-or-self, following^{-1} = preceding, and following-sibling^{-1} = preceding-sibling.

LEMMA 10.1. *For each pair of nodes $x, y \in$ dom and axis χ, $x\chi y$ iff $y\chi^{-1}x$.*

(Proof by a very easy induction.)

DEFINITION 10.2. Let the (abstract) syntax of the Core XPath language be defined by the EBNF grammar

```
cxp:          locationpath | '/' locationpath
locationpath: locationstep ('/' locationstep)*
locationstep: χ '::' t | χ '::' t '[' pred ']'
pred:         pred 'and' pred | pred 'or' pred
              | 'not' '(' pred ')' | cxp | '(' pred ')'
```

"cxp" is the start production, χ stands for an axis (see above), and t for a "node test" (either an XML tag or "*", meaning "any label"). The semantics of Core

XPath queries is defined by a function $\mathcal{S}_{\rightarrow}$

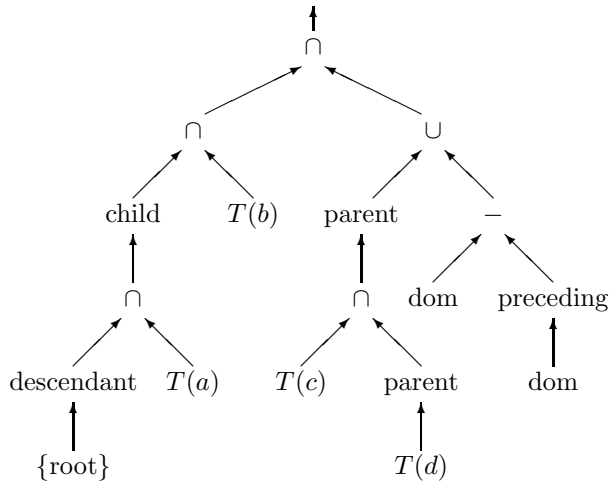
$$
\begin{aligned}
\mathcal{S}_{\rightarrow}[\![\chi{::}t]\!](N_0) &:= \chi(N_0) \cap T(t) \\
\mathcal{S}_{\rightarrow}[\![/\chi{::}t]\!](N_0) &:= \chi(\{\text{root}\}) \cap T(t) \\
\mathcal{S}_{\rightarrow}[\![\pi/\chi{::}t]\!](N_0) &:= \chi(\mathcal{S}_{\rightarrow}[\![\pi]\!](N_0)) \cap T(t) \\
\mathcal{S}_{\rightarrow}[\![\pi[e]]\!](N_0) &:= \mathcal{S}_{\rightarrow}[\![\pi]\!](N_0) \cap \mathcal{E}_1[\![e]\!] \\
\mathcal{S}_{\leftarrow}[\![\chi{::}t]\!] &:= \chi^{-1}(T(t)) \\
\mathcal{S}_{\leftarrow}[\![\chi{::}t[e]]\!] &:= \chi^{-1}(T(t) \cap \mathcal{E}_1[\![e]\!]) \\
\mathcal{S}_{\leftarrow}[\![\chi{::}t/\pi]\!] &:= \chi^{-1}(\mathcal{S}_{\leftarrow}[\![\pi]\!] \cap T(t)) \\
\mathcal{S}_{\leftarrow}[\![\chi{::}t[e]/\pi]\!] &:= \chi^{-1}(\mathcal{S}_{\leftarrow}[\![\pi]\!] \cap T(t) \cap \mathcal{E}_1[\![e]\!]) \\
\mathcal{S}_{\leftarrow}[\![/\pi]\!] &:= \frac{\text{dom}}{\text{root}}(\mathcal{S}_{\leftarrow}[\![\pi]\!]) \\
\mathcal{E}_1[\![e_1 \text{ and } e_2]\!] &:= \mathcal{E}_1[\![e_1]\!] \cap \mathcal{E}_1[\![e_2]\!] \\
\mathcal{E}_1[\![e_1 \text{ or } e_2]\!] &:= \mathcal{E}_1[\![e_1]\!] \cup \mathcal{E}_1[\![e_2]\!] \\
\mathcal{E}_1[\![\text{not}(e)]\!] &:= \text{dom} - \mathcal{E}_1[\![e]\!] \\
\mathcal{E}_1[\![\pi]\!] &:= \mathcal{S}_{\leftarrow}[\![\pi]\!]
\end{aligned}
$$

where N_0 is a set of context nodes and a query π evaluates as $\mathcal{S}_{\rightarrow}[\![\pi]\!](N_0)$. □

Example 10.3. The Core XPath query

/descendant::a/child::b[child::c/child::d or not(following::*)]

is evaluated as specified by the query tree



(There are alternative but equivalent query trees due to the associativity and commutativity of some of our operators.) □

The semantics of XPath and Core XPath (defined using $\mathcal{S}_{\leftarrow}$, $\mathcal{S}_{\rightarrow}$, and $\mathcal{E}_1$) coincide in the following way:

THEOREM 10.4. *Let π be a Core XPath query and $N_0 \subseteq dom$ be a set of context nodes. Then,*

$$\begin{aligned} \mathcal{S}_{\leftarrow}[\![\pi]\!] &= \{x \mid \mathcal{S}_{\downarrow}[\![\pi]\!](\{x\}) \neq \emptyset\} \\ \mathcal{E}_1[\![e]\!] &= \{x \mid \mathcal{E}_{\downarrow}[\![e]\!](\{\langle x, 1, 1\rangle\})\} \\ \langle \mathcal{S}_{\rightarrow}[\![\pi]\!](N_0)\rangle &= \mathcal{S}_{\downarrow}[\![\pi]\!](\langle N_0\rangle). \end{aligned}$$

This can be shown by easy induction proofs. Thus, Core XPath (evaluated using $\mathcal{S}_{\rightarrow}$) is a fragment of XPath, both syntactically and semantically.

THEOREM 10.5. *Core XPath queries can be evaluated in time $O(|D|*|Q|)$, where $|D|$ is the size of the data and $|Q|$ is the size of the query.*

PROOF. Given a query Q, it can be rewritten into an algebraic expression E over the operations χ, $\cup$, $\cap$, '$-$', and $\frac{\text{dom}}{\text{root}}$ using $\mathcal{S}_{\rightarrow}$, $\mathcal{S}_{\leftarrow}$, and $\mathcal{E}_1$ in time $O(|Q|)$. Each of the operations in our algebra can be carried out in time $O(|D|)$. Since at most $O(|Q|)$ such operations need to be carried out to process E, the complexity bound follows. □

10.2 XPatterns

We extend our linear-time fragment Core XPath by the operation id: nset → nset of Table II by defining "id" as an axis relation

$$id := \{\langle x_0, x\rangle \mid x_0 \in \text{dom},\ x \in \text{deref\_ids}(\text{strval}(x_0))\}$$

Queries of the form $\pi_1/\text{id}(\pi_2)/\pi_3$ are now treated as $\pi_1/\pi_2/\text{id}/\pi_3$.

LEMMA 10.6. *Let $\pi_1/id(\pi_2)/\pi_3$ be an XPath query s.t. $\pi_1/\pi_2/id/\pi_3$ is a query in Core XPath with the "id" axis. Then, the semantics of the two queries relative to a set of context nodes $N_0 \in dom$ coincide:*

$$\mathcal{S}_{\downarrow}[\![\pi_1/id(\pi_2)/\pi_3]\!](\langle N_0\rangle) = \mathcal{S}_{\rightarrow}[\![\pi_1/\pi_2/id/\pi_3]\!](N_0).$$

THEOREM 10.7. *Queries in Core XPath with the "id" axis can be evaluated in time $O(|D| * |Q|)$.*

PROOF. The interesting part of this proof is to define a function id: $2^{\text{dom}} \rightarrow 2^{\text{dom}}$ and its inverse consistent with the functions of Definition 3.1 which is computable in linear time. We make use of a binary auxiliary relation "ref" which contains a tuple of nodes $\langle x, y\rangle$ iff the text belonging to x in the XML document, but which is directly inside it and not further down in any of its descendants, contains a whitespace-separated string referencing the identifier of node y.

For example, let $\text{id}(i) = n_i$. Then, for the XML document ⟨t id=1⟩ 3 ⟨t id=2⟩ 1 ⟨/t⟩ ⟨t id=3⟩ 1 2 ⟨/t⟩ ⟨/t⟩, we have ref := $\{\langle n_1, n_3\rangle, \langle n_2, n_1\rangle, \langle n_3, n_1\rangle, \langle n_3, n_2\rangle\}$.

Relation "ref" can be efficiently computed in a preprocessing step. It does not satisfy any functional dependencies, but it is guaranteed to be of linear size w.r.t. the input data (however, not in the tree nodes). Now we can encode $\text{id}(S)$ as those nodes reachable from S *and its descendants* using "ref".

$\text{id}(S) := \{y \mid x \in \text{descendant-or-self}(S),\ \langle x, y\rangle \in \text{ref}\}$
$\text{id}^{-1}(S) := \text{ancestor-or-self}(\{x \mid \langle x, y\rangle \in \text{ref},\ y \in S\})$

This computation can be performed in linear time. □

"@n", "@*", "text()", "comment()", "pi(n)", and "pi()" (where n is a label) are simply sets provided with the document (similar to those obtained through the node test function T).
"=s" (s is a string) can be encoded as a unary predicate whose extension can be computed using string search in the document before the evaluation of our query starts. Clearly, this can be done in linear time.
first-of-any $:= \{y \in \text{dom} \mid \nexists x : \text{nextsibling}(x, y)\}$ last-of-any $:= \{x \in \text{dom} \mid \nexists y : \text{nextsibling}(x, y)\}$
"id(s)" is a unary predicate and can easily be computed (in linear time) before the query evaluation.

Table VI. Some unary predicates of XLST Patterns.

We may define XPatterns as the smallest language that subsumes Core XPath and the XSLT Pattern language of [World Wide Web Consortium 1998] (see also [Wadler 1999] for a good and formal overview of this language) and is (syntactically) contained in XPath. Stated differently, it is obtained by extending the language of [World Wide Web Consortium 1998] without the first-of-type and last-of-type predicates (which do not exist in XPath) to support all of the XPath axes. As pointed out in the introduction, XPatterns is an interesting and practically useful query language. Surprisingly, XPatterns queries can be evaluated in linear time.

THEOREM 10.8. *Let D be an XML document and Q be an XPatterns query. Then, Q can be evaluated on D in time $O(|D| * |Q|)$.*

PROOF. XPatterns extends Core XPath by the "id" axis and a number of features which are definable as unary predicates, of which we give an overview in Table VI. It becomes clear by considering the semantics definition of [Wadler 1999] that after parsing the query, one knows of a fixed number of predicates to populate, and this action takes time $O(|D|)$ for each. Thus, since this computation precedes the query evaluation – which has a time bound of $O(|D| * |Q|)$ – this does not pose a problem. "id(c)" (for some fixed string c) may only occur at the beginning of a path, thus in a query of the form id(c)/π, π is evaluated relative to the set id(c) just as, say, {root} is for query /π. □

Let Σ be a finite set of all possible node names that a document may use (e.g., given through a DTD). The unary first-of-type and last-of-type predicates can be computed in time $O(|D| * |\Sigma|)$ when parsing the document, but are of size $O(|D|)$:

$$\text{first-of-type}() := \bigcup_{l \in \Sigma} \left(T(l) - \text{nextsibling}^+(T(l))\right)$$

$$\text{last-of-type}() := \bigcup_{l \in \Sigma} \left(T(l) - (\text{nextsibling}^{-1})^+(T(l))\right)$$

where $R^+ = R.R^*$.

11. A LINEAR-SPACE FRAGMENT OF XPATH

11.1 The Extended Wadler Fragment

In [Wadler 2000], Wadler considers a useful fragment of XPath with predicates made up of location paths on the one hand and arithmetic expressions with the

functions position() and last() on the other hand. This fragment is the key to a big fragment of XPath, which can be processed in *linear space* and *quadratic time* w.r.t. to the size of the XML data. We shall identify some restrictions on XPath that guarantee the linear space complexity. It will turn out that these restrictions also suffice to guarantee the quadratic time complexity. In fact, it is easy to check that the fragment in [Wadler 2000] fulfills these restrictions. Hence, we shall refer to our XPath fragment as the "Extended Wadler Fragment".

Suppose that we want to evaluate an XPath expression e. Actually, if the result type of e is scalar (i.e., num, bool or str), then we can simply evaluate e as in Section 8. We just have to make sure that the size of scalar values is independent of the XML data. Hence, we require

Restriction 1. The XPath functions which select data from an XML document, are not allowed, i.e., local-name, namespace-uri, name, string, number, string-length, and normalize-space. □

On the other hand, if the result of e is a (linearly big) node set, then e cannot simply be evaluated simultaneously for all (linearly many) possible context-nodes, since this would require quadratic space. Of course, we must not treat the context-nodes in a loop since this has been identified in Section 2 as the very reason why previous XPath evaluation algorithms require exponential time. Instead, we need a different strategy. Recall from Section 5 that we assume that all type conversions in an XPath expression are made explicit. Hence, (by Restriction 1) expressions that evaluate to a node set can only occur in one of the following five forms:

(1) boolean(*nset*) (2) *nset* RelOp *scalar* (3) *nset* RelOp *nset*

(4) count(*nset*) (5) sum(*nset*)

where RelOp $\in \{=, \neq, \leq, <, \geq, >\}$, *nset* denotes an expression whose result is a node set, and *scalar* denotes any other expression. Below, we shall present an optimization for the first two cases. Unfortunately, this method does not work in case of the latter three ones. We thus require

Restriction 2. Expressions of the form *nset* RelOp *nset* as well as calls of the functions count and sum are not allowed. Moreover, for expressions of the form *nset* RelOp *scalar* we require that *scalar* does not depend on any context. □

As for the form that an *nset*-expression can have, we distinguish two principal cases, namely location paths or expressions of the form id(e). Of course, the calls of id can be arbitrarily nested. However, ultimately, we either have $e \equiv \mathrm{id}(\mathrm{id}(\ldots(c)\ldots))$ or $e \equiv \mathrm{id}(\mathrm{id}(\ldots(\pi)\ldots))$, where c is a string-expression and π is a location path. For the latter case, we rewrite $\mathrm{id}(\mathrm{id}(\ldots(\pi)\ldots))$ to the form $\pi/\mathrm{id}/\mathrm{id}/\ldots/\mathrm{id}$, i.e., analogously to Section 10.2, we consider "id" as a new axis. Hence, in this case, expressions of the form $\mathrm{id}(\mathrm{id}(\ldots(\pi)\ldots))$ are treated as location paths. For the former case, we impose

Restriction 3. In expressions of the form $\mathrm{id}(\mathrm{id}(\ldots(\ c)\ldots))$, where c is a string-expression, we require that c must not depend on any context. □

Actually, *nset*-expressions of the form $\mathrm{id}(\mathrm{id}(\ldots(c)\ldots))$, where c does not depend on any context, can be simply evaluated by the algorithm from Section 8 in linear

space. For any other *nset*-expressions (i.e. location paths, possibly involving the id-"axis"), we observe that (because of Restriction 2) *nset*-expressions are only allowed to occur as operands of expressions that yield a boolean result value. In particular, the context-value table for the whole expression (of the form "*nset* RelOp *scalar*" or "boolean (*nset*)"), clearly requires linear space only. We just have to avoid the explicit computation of the context-value table for the location path *nset*. This can be achieved as follows.

Bottom-up evaluation of certain location paths. A location path π inside an expression of the form boolean(π) or π RelOp c has an $\exists$-semantics, e.g., boolean(π) evaluates to "true" for a context-node x, iff *there exists* at least one node in the node set resulting from the evaluation of π. Thus, the set of nodes x, for which boolean(π) or π RelOp c evaluates to "true" can be computed as follows:

—First compute the "initial node set" Y. For an expression boolean(π), we set $Y :=$ dom. An expression π RelOp c with c of type bool is treated like the expression boolean(π) RelOp c. For any other type of c, we set $Y := \{x \mid$ self::* RelOp c evaluates to "true" for the context-node $x\}$.

—Compute X by propagating Y backwards via the *inverse location steps* of π.

As for the backward propagation of a node set via the *inverse location steps*, we proceed as follows: If we have $\pi = \chi_1 :: */\chi_2 :: */ \ldots /\chi_n :: *$, then we set $X_n := Y$ and $X_{i-1} := \chi_i^{-1}(X_i)$ (where χ_i^{-1} denotes the inverse axis from Section 10.1) for every $i \in \{1, \ldots, n\}$. Hence, $X := X_0$ is the desired node set. Note that if χ_i is the id-"axis", then we have $\chi_i^{-1}(X_i) = \mathrm{id}^{-1}(X_i)$. Recall from Lemma 3.3 (for "ordinary" axes) and Theorem 10.7 (for the id-axis), that $\chi_i^{-1}(X_i)$ can be computed in time $O(|D|)$ for any node set $X_i \subseteq$ dom.

Now let $\pi \equiv \chi_1 :: t_1[e_{11}] \ldots [e_{1k_1}]/ \ldots /\chi_n :: t_n[e_{n1}] \ldots [e_{nk_n}]$. Then we have to restrict each node set X_i to the set X_i' of those nodes for which the node test t_i holds and apply the inverse axis function χ_i^{-1} to X_i'. For the predicates we proceed analogously to the MinContext algorithm, by calling the procedures *eval_by_cnode_only* and *eval_single_context*. In the Appendix A, we give the pseudocode of a procedure *eval_bottomup_path* (plus the auxiliary procedure *propagate_path_backwards*) for expressions π RelOp c and boolean(π), respectively. Note that in the procedure *propagate_path_backwards* we assume (w.l.o.g.) that all occurrences of "|" have been removed. This can be easily achieved by replacing "boolean($\pi_1|\pi_2|\ldots|\pi_k$)" and "$\pi_1|\pi_2|\ldots|\pi_k$ RelOp c" by "boolean(π_1) or ... or boolean(π_k)" and "(π_1 RelOp c) or ... or (π_k RelOp c)".

11.2 The Algorithm OptMinContext

In order to incorporate the above ideas of a bottom-up evaluation of certain location paths, our MinContext algorithm has to be modified to a new algorithm OptMinContext as follows:

Algorithm 11.1. (OptMinContext)
Input: XPath query Q, XML document D, context $\langle x, p, s\rangle$;
Output: Result value of Q over D for the context $\langle x, p, s\rangle$;
Method:

evaluate all "bottom-up location paths" inside Q
(starting with the innermost ones in case of nesting);
call MinContext
(Of course, subexpressions that have already been
evaluated bottom-up are not evaluated again); □

We illustrate the algorithms OptMinContext and MinContext by the following example:

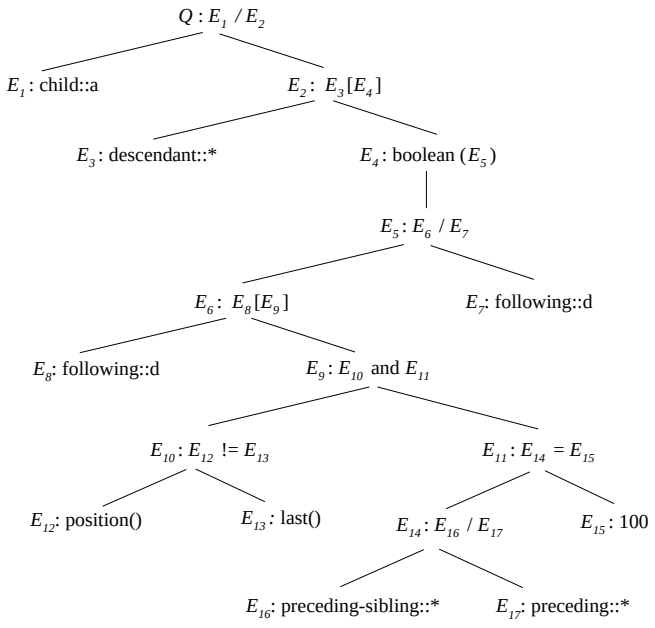


Fig. 13. Parse tree $\mathcal{T}$ of Q in Example 11.2.

Example 11.2. Let the XPath query Q be defined as $Q \equiv$ /child::a/descendant::*[boolean(following::d[(position() != last()) and (preceding-sibling::*/preceding::* = 100)]/following::d)]. We want to evaluate Q over the XML document D from Figure 8. We do not need an input context, since Q is an absolute location path.

The parse tree $\mathcal{T}$ of Q is depicted in Figure 13. Q has two inner location paths E_5 and E_{14} which have to be evaluated bottom-up. Note that we have omitted the explicit type conversion of E_{14} to string(E_{14}), since this is not needed in case of our bottom-up evaluation here. Analogously to the running example in Section 8, we write E_i both to refer to subexpressions of Q and to nodes in the parse tree $\mathcal{T}$.

We start the bottom-up evaluation with the innermost location path, namely E_{14}: The initial node set is $Y := \{x_{14}, x_{24}\}$, which corresponds to all context-nodes for which "self::* = 100" evaluates to "true". To this node set, we first apply following (= preceding^{-1}), which yields the node set $\{x_{21}, x_{22}, x_{23}, x_{24}\}$. By applying following-sibling (= preceding-sibling^{-1}) to this, we get $\{x_{23}, x_{24}\}$. Hence,

the context-value table of the node E_{11} is the 2-dimensional table $\subseteq$ dom $\times$ {true, false}, s.t. exactly the nodes in $\{x_{23}, x_{24}\}$ have the value "true" in the second column.

For the bottom-up evaluation of the path E_5, we have to take the node tests and the predicate E_9 into account. We start the evaluation with $Y :=$ dom. Now we have to apply the inverse step of following::d. Hence, we first restrict Y to the set Y' of those nodes for which the node test "d" yields "true", i.e. $Y' = \{x_{14}, x_{23}, x_{24}\}$. By applying preceding (= following^{-1}) to Y', we get $Y'' = \{x_{11}, x_{12}, x_{13}, x_{14}, x_{22}, x_{23}\}$. Now we have to apply the location step following::d[E_9] backwards. To this end, we first restrict Y'' to the elements with name d. We thus get $Y''' = \{x_{14}, x_{23}\}$. Now we have to check for which nodes in Y''' (together with appropriate values for p and s) the predicate E_9 evaluates to "true". To this end, we first call the procedure *eval_by_cnode_only* to evaluate those nodes in the parse tree rooted at E_9 which do not depend on (the current values of) p and s. Actually, in this case, only the subtree rooted at E_{11} has this property. However, *table*(E_{11}) has already been determined by the bottom-up evaluation described above. Hence, the call of procedure *eval_by_cnode_only* has no effect here. Note that $X =$ following$^{-1}(Y''') = \{x_{11}, x_{12}, x_{13}, x_{14}, x_{22}\}$. In order to evaluate the predicate E_9 also for p and s via the procedure *eval_single_context*, we have to consider all combinations of previous/current context-node (in $X \times Y'''$) w.r.t. the "following"-axis. Actually, both nodes in Y''' can be extended by appropriate values of p and s to a context triple, s.t. E_9 evaluates to "true" for this context, e.g.: $\langle x_{14}, 2, 6\rangle$ and $\langle x_{23}, 5, 6\rangle$ (which are both obtained via the previous context-node x_{12}). Hence, the predicate E_9 does not lead to a restriction of Y'''. Therefore, the desired context-value table $\subseteq$ dom $\times$ {true, false} of the node E_4 has the value "true" in the second column exactly for the nodes in X.

Finally, we evaluate the location path at the outermost level of Q by calling the procedure *eval_outermost_locpath*. The location step child::a yields the set $\{x_{10}\}$ independent of any input context. Moreover, by the step descendant::*, we get dom $- \{r, x_{10}\}$. However, these nodes have to be intersected with the set X computed above. Hence, the final result of evaluating the query Q is $\{x_{11}, x_{12}, x_{13}, x_{14}, x_{22}\}$. □

Below, we show that Restrictions 1 through 3 lead to the desired improvement of the efficiency.

THEOREM 11.3. *The* OPTMINCONTEXT *algorithm evaluates XPath queries from the* Extended Wadler Fragment *(i.e., the set of all XPath expressions fulfilling the Restrictions 1 through 3 from Section 11.1) in space $O(|D| * |Q|^2)$ and time $O(|D|^2 * |Q|^2)$, where $|D|$ is the size of the XML-document and $|Q|$ is the size of the XPath query.*

PROOF. We first consider the *space complexity*: Of course, an input XPath expression Q has at most $|Q|$ subexpressions. Hence, it suffices to show that the information that we have to store for each subexpression e of Q is bounded by $O(|D| * |Q|)$:

For subexpressions e along an outermost location path, we have to propagate node sets in dom, whose size is clearly bounded by $|D|$. For an inner location

path e, we may assume by Restriction 2 from Section 11.1 that e occurs in the form boolean(e) or e RelOp c, where the relevant context of c is empty. Then the backward evaluation of such a location path e again comes down to propagating node sets in dom.

Now consider the case of the id-function, i.e., by Restriction 3 from Section 11.1, e is either of the form $e \equiv \mathrm{id}(\mathrm{id}(\ldots(\ \pi)\ldots))$ for some location path π or $e \equiv \mathrm{id}(\mathrm{id}(\ldots(\ c)\ldots))$, where c has an empty relevant context. In the former case, e is treated like any other inner location path. In the latter case, the context-value table corresponding to e consists of a single row whose size is of course bounded by $|\mathrm{dom}| \leq |D|$.

Finally, suppose that e is any other expression. Then, in particular, e evaluates to a number, a string, or a boolean value. Recall from Section 8 that we explicitly set up the context-value table for e only if the relevant context of e is a subset of {'cn'}. Hence, the context-value table has at most $|D|$ rows. Moreover, by Restriction 1 from Section 11.1, it is guaranteed that scalar values do not depend on the input document. They are thus bounded by $O(|Q|)$. Hence, in all of the above cases of the subexpression e, the information that we have to store is bounded by $O(|D| * |Q|)$.

For the *time complexity*, we have to show that the time required to deal with each subexpression e of Q is bounded by $O(|D|^2 * |Q|)$. To this end, we distinguish the same cases as above:

The propagation of node sets can be done in time $O(|D|)$ – be it in forward direction for location paths on the outermost level or in backward direction for inner location paths (see Lemma 3.3 for "ordinary" axes and Theorem 10.7 for the id-function).

For any other expression e, the computation of the result value of e for a single context $\vec{c}$ takes at most $O(|Q|)$ time, given that the subexpressions of e have already been evaluated. This is due to Restriction 1 from Section 11.1. Moreover, each subexpression e of Q has to be evaluated for at most $|D|^2$ contexts. Hence, the total time required by each subexpression e of Q is indeed bounded by $O(|D|^2 * |Q|)$. □

In fact, even a slightly stronger property holds for our algorithm, namely:

COROLLARY 11.4. *Let Q be an arbitrary XPath query to which our* OPTMINCONTEXT *algorithm is applied. Moreover, let e be a subexpression in Q, s.t. e is in the Extended Wadler Fragment. If e is a location path, then we also require that either $Q \equiv e$ or $Q \equiv \pi'/e$ for an arbitrary relative or absolute location path π' or e occurs in the form boolean(e) or $e\,\mathrm{RelOp}\,c$ (where c is independent of any context) in Q. Then e is evaluated in space $O(|D| * |e|^2)$ and time $O(|D|^2 * |e|^2)$. These upper bounds apply to the total space and time required by the* OPTMINCONTEXT *algorithm to evaluate e for all relevant contexts.*

PROOF. As far as the *space complexity* is concerned, we have to show that the information to be stored for each subexpression e' of e is bounded by $O(|D| * |e|)$.

First suppose that e' evaluates to a node set. If e' is a location path then, by the Restrictions 1 and 2 plus the additional restrictions imposed by Corollary 11.4, the evaluation of e' comes down to propagating node sets in dom – either in forward direction (if Q itself is a location path and e' is a subexpression along this outermost location path) or in backward direction (if e' occurs in the form boolean(e') or

$e'\,\mathrm{RelOp}\,c$). This is also true for the case that e' is of the form $e' \equiv \mathrm{id}(\mathrm{id}(\ldots(\pi)\ldots))$ for some location path π. On the other hand, if e' is of the form $e' \equiv \mathrm{id}(\mathrm{id}(\ldots(\ c)\ldots))$, where c has an empty relevant context, then the context-value table corresponding to e' consists of a single row whose size is clearly bounded by $|\mathrm{dom}| \leq |D|$.

The case that e' evaluates to a number, a string, or a boolean value is treated exactly like in the proof of Theorem 11.3. Likewise, the upper bound on the *time complexity* can be established by the same considerations as in the proof of Theorem 11.3. □

An analogous result to Corollary 11.4 can also be shown for the Core XPath Fragment introduced in Section 10.1.

COROLLARY 11.5. *Let Q be an arbitrary XPath query to which our* OPTMINCONTEXT *algorithm is applied. Moreover, suppose that π is a location path from Core XPath, s.t. $Q \equiv \pi$ or $Q \equiv \pi'/\pi$ for an arbitrary relative or absolute location path π' or π occurs in Q in the form boolean(π) or $\pi\,\mathrm{RelOp}\,c$ (where c is independent of any context).*

Then π is evaluated in time $O(|D| * |\pi|)$. This upper bound applies to the total time required by the OPTMINCONTEXT *algorithm to evaluate π for all relevant contexts.*

PROOF. Core XPath expressions of the form $\chi :: t[\pi']$ are a short-hand for $\chi :: t[\mathrm{boolean}(\pi')]$. Likewise, any occurrence of a location path π as an operand of one of the logical operators "and", "or", and "not" may be replaced by $\mathrm{boolean}(\pi)$ without changing the meaning of the XPath query (i.e., we just make the type conversion explicit). Hence, Core XPath is clearly contained in the Extended Wadler Fragment.

Actually, the only reason why we have quadratic time complexity w.r.t. the size of the data $|D|$ in Corollary 11.4 is that we possibly have to evaluate predicates in a loop over all (quadratically many) pairs of previous/current context-node in order to take the context-position and context-size into account. However, in Core XPath, position() and last() are not allowed and, therefore, no such loop is required.

On the other hand, the quadratic time complexity w.r.t. the size $|e|$ of a subexpression e of $|Q|$ in Corollary 11.4 is due to the possible blow up of strings and numbers by iterated application of XPath core library functions like concat and arithmetic operations. However, these constructs are not contained in Core XPath. We thus end up with a linear time upper bound both w.r.t. to the data and the query for these subexpressions. □

12. CONCLUSIONS

In this article, we presented the first XPath query evaluation algorithm that runs in polynomial time with respect to the size of both the data and of the query. Our results will allow for XPath engines to deal efficiently with very sophisticated queries.

Moreover, we have presented several interesting fragments of XPath for which the query evaluation can be even further optimized. We have also shown how the ideas presented here can be profitably integrated into existing XPath processors thus reducing their complexity from exponential to polynomial time.

$\|Q\|$	IE6			XMLTaskforce XPath					
	10	20	200	10	20	200	500	1000	2000
1				0.00	0.00	0.00	0.00	0.00	0.01
2			2	0.00	0.00	0.02	0.12	0.57	2.57
3			346	0.00	0.00	0.02	0.23	1.14	5.07
4		1	-	0.00	0.00	0.05	0.33	1.70	7.58
5		21	-	0.00	0.00	0.07	0.44	2.32	10.09
6	5	406	-	0.00	0.00	0.07	0.54	2.88	12.58
7	42	-	-	0.00	0.00	0.09	0.67	3.45	15.42
8	437	-	-	0.00	0.00	0.12	0.78	4.03	17.58
9	-	-	-	0.00	0.00	0.14	0.90	4.59	20.46
10	-	-	-	0.00	0.00	0.15	0.99	5.13	22.61
20	-	-	-	0.00	0.01	0.30	2.20	10.80	47.90
30	-	-	-	0.01	0.01	0.48	3.19	16.72	73.13
40	-	-	-	0.01	0.02	0.65	4.39	22.37	98.50
50	-	-	-	0.01	0.02	0.80	5.53	28.13	123.79

Table VII. Benchmark results in seconds for IE6 vs. our implementation ("XMLTaskforce XPath"), on the queries of Experiment 2 and document sizes 10, 20, and 200.

We have made a main-memory implementation of the top-down algorithm of Section 7. Table VII compares it to IE6 along the assumptions made in Experiment 2 (i.e., the queries of which were strictly the most demanding of all three experiments).[10] It shows that our algorithm scales linearly in the size of the queries and quadratically (for this class of queries) in the size of the data. Our implementation is still an early prototype without sophisticated optimizations. It closely coheres to the specification given in this article. Our system and further resources related to this article can be downloaded from `http://www.xmltaskforce.com`.

In this paper, we have exclusively dealt with XPath 1. Note that XPath 2, which currently has the status of a W3C working draft (see [World Wide Web Consortium 2004]), significantly extends XPath 1 in many directions. Of course, XPath 1 will remain an important fragment of XPath 2. Moreover, the ideas presented in this paper can be easily extended to cope with some of the new XPath 2 features, e.g.: It is no problem to deal with the simple types of XML Schema in the last column of our context-value tables. Likewise, most of the new built-in functions and operators easily fit into the CVT-framework. On the other hand, adding node variables to XPath 1 immediately leads to NP-completeness of the XPath evaluation problem. The situation gets even worse if we consider full XPath 2 (including the use of arbitrary XQuery expressions), which is known to be Turing-complete. Hence, there is clearly no hope to extend our CVT-principle to these features of XPath 2.

Finally, note that in [Gottlob et al. 2003a], which is based on work subsequent to the research reported on in this article, the upper bounds on the complexity of

[10]Note that the performance of the systems should not be directly compared since they were run on different systems – IE6 is only available on MS Windows. IE6 was benchmarked using the hardware setup of Experiment 2, while XMLTaskforce XPath (release tag 20040812) was benchmarked using the hardware setup of Experiment 5. Note that the timings of the IE6 only have the precision of ± 1 second. Hence, we left those entries in Table VII empty where we got values below 1 second since these values are not meaningful.

XPath of the present article are complemented by tight lower bounds. It is shown that full XPath 1 is *complete* for polynomial time. Further complexity results related to XPath can be found in [Segoufin 2003]. In [Bar-Yossef et al. 2004], several memory lower bounds for evaluating XPath queries over XML streams are shown by communication complexity methods.

Acknowledgments

We thank G. Moerkotte and the anonymous reviewers of VLDB 2002, ICDE 2003, and ACM TODS, whose constructive comments have helped to considerably improve this article, and P. Fankhauser for a wealth of pointers to the literature.

We also thank J. Siméon for pointing out to us that the mapping from XPath to the XML Query Algebra [World Wide Web Consortium 2002], which will be the standard semantics definition for XPath 2, in a direct functional implementation also leads to exponential-time query processing on XPath 1 (which is a fragment of XPath 2).

Finally, we thank Thomas Lukasser for integrating the context-value table principle into Xalan, as reported on in Section 9, and for contributing considerably to the implementation of the XMLTaskforce XPath engine.

REFERENCES

Al-Khalifa, S., Srivastava, D., Jagadish, H. V., Koudas, N., Patel, J. M., and Wu, Y. 2002. "Structural Joins: A Primitive for Efficient XML Query Pattern Matching". In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE'02)*. San Jose, California.

Altinel, M. and Franklin, M. 2000. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'2000)*. Cairo, Egypt, 53–64.

Apache Foundation. 2004. Xalan-Java version 2.2.D11. http://xml.apache.org/xalan-j/.

Bar-Yossef, Z., Fontoura, M., and Josifovski, V. 2004. "On the Memory Requirements of XPath Evaluation over XML Streams". In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'04)*. 177–188.

Bruno, N., Koudas, N., and Srivastava, D. 2002. "Holistic Twig Joins: Optimal XML Pattern Matching". In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. Madison, Wisconsin.

Chan, C. Y., Fan, W., Felber, P., Garofalakis, M. N., and Rastogi, R. 2002. "Tree Pattern Aggregation for Scalable XML Data Dissemination". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*. Hong Kong, China, 826–837.

Chan, C. Y., Felber, P., Garofalakis, M. N., and Rastogi, R. 2002. "Efficient Filtering of XML Documents with XPath Expressions". In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE'02)*. San Jose, California, USA.

Clark, J. 1999. XT. A Java Implementation of XSLT. http://www.jclark.com/xml/xt.html/.

Gottlob, G. and Koch, C. 2002. "Monadic Queries over Tree-Structured Data". In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*. Copenhagen, Denmark, 189–202.

Gottlob, G., Koch, C., and Pichler, R. 2002. "Efficient Algorithms for Processing XPath Queries". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*. Hong Kong, China, 95–106.

Gottlob, G., Koch, C., and Pichler, R. 2003a. "The Complexity of XPath Query Processing". In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'03)*. San Diego, CA USA.

Gottlob, G., Koch, C., and Pichler, R. 2003b. "XPath Query Evaluation: Improving Time and Space Efficiency". In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE'03)*. Bangalore, India, 379–390.

Green, T. J., Miklau, G., Onizuka, M., and Suciu, D. 2003. "Processing XML Streams with Deterministic Automata". In *Proc. of the 9th International Conference on Database Theory (ICDT'03)*.

Grust, T., van Keulen, M., and Teubner, J. 2004. "Accelerating XPath Location Steps in Any RBMDS". *ACM Transactions on Database Systems* **29**, 1 (Mar.).

Gupta, A. K. and Suciu, D. 2003. "Stream Processing of XPath Queries with Predicates". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. 419–430.

Kay, M. 2003. Saxon version 6.5.2. http://saxon.sourceforge.net/.

Kilpeläinen, P. 1992. Tree matching problems with applications to structured text databases. Ph.D. thesis, Department of Computer Science, University of Helsinki. Report A-1992-6.

Microsoft Corporation. 2001. Internet Explorer IE6. http://www.microsoft.com/windows/ie/default.asp.

Peng, F. and Chawathe, S. 2003. "XPath Queries on Streaming Data". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. 431–442.

Ramanan, P. 2002. "Efficient Algorithms for Minimizing Tree Pattern Queries". In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. Madison, Wisconsin.

Segoufin, L. 2003. "Typing and Querying XML Documents: Some Complexity Bounds". In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'03)*. 167–178.

Shasha, D., Wang, J. T. L., and Giugno, R. 2002. "Algorithmics and Applications of Tree and Graph Searching". In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'02)*.

Wadler, P. 2000. "Two Semantics for XPath". Draft paper available at http://www.research.avayalabs.com/user/wadler/.

Wadler, P. December 1999. "A Formal Semantics of Patterns in XSLT". In *Markup Technologies*. Philadelphia. Revised version in Markup Languages, MIT Press, June 2001.

World Wide Web Consortium. DOM Specification http://www.w3c.org/DOM/.

World Wide Web Consortium. 1998. XSL Working Draft http://www.w3.org/TR/1998/WD-xsl-19981216.

World Wide Web Consortium. 1999. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/.

World Wide Web Consortium. 2000. "Extensible Markup Language (XML) 1.0 (Second Edition)". http://www.w3.org/TR/REC-xml.

World Wide Web Consortium. 2002. "XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Aug. 16th 2002). `http://www.w3.org/TR/query-algebra/`.

World Wide Web Consortium. 2004. XML Path Language (XPath) 2.0, Working Draft. http://www.w3.org/TR/2004/WD-xpath20-20040723/.

Yannakakis, M. 1981. "Algorithms for Acyclic Database Schemes". In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB'81)*.

Appendix

A. PSEUDOCODE PRESENTATIONS

Analogously to Section 8, we use the following notation: For a node N in the parse tree, we write $table(N)$ to denote the context-value table at N and $expr(N)$ to denote the subexpression of Q corresponding to N. Conversely, for a subexpression e of Q, we denote by $node(e)$ the corresponding node in the parse tree. For the sake of better readability, the parse tree $\mathcal{T}$ of the input query Q and the context value tables $table(N)$ for the nodes N in $\mathcal{T}$ are treated as global variables.

procedure eval_outermost_locpath:

input: node N in the parse tree
set X of possible context-nodes

output: set Y of nodes that can be reached from X via $expr(N)$.

```
begin
/* case distinction over expr(N) */
 if expr(N) = /π then return
  eval_outermost_locpath(node(π), {root});
 elseif expr(N) = π1|π2 then
 begin
  Y1 := eval_outermost_locpath(node(π1), X);
  Y2 := eval_outermost_locpath(node(π2), X);
  return Y1 ∪ Y2;
 end;
 elseif expr(N) = π1/π2 then
 begin
  Y := eval_outermost_locpath(node(π1), X);
  return
   eval_outermost_locpath(node(π2), Y);
 end;
 elseif expr(N) = χ :: t[e1]...[eq] then
 begin
  Y := nodes reachable from X via χ :: t;
  for i := 1 to q do
   eval_by_cnode_only(node(ei), Y);
  R := ∅;
  if (∀i ∈ {1,...,q}) ({'cp','cs'}∩
   Relev(node(ei))) = ∅ holds then
   for each y ∈ Y do
    if ∀i ∈ {1,...,q}
     eval_single_context(node(ei), ⟨y,*,*⟩) =
     true then R := R ∪ {y};
     /* "*" denotes the wildcard for
     irrelevant parts of the context */
  else /* some ei depends on 'cp' or 'cs' */
  begin
   for each x ∈ X do
   begin
    Z := {z ∈ Y | xχz};
    for i := 1 to q do
    begin
     let Z = {z1,...,zm} be ordered
      according to the axis χ; /* i.e., in
      document order or reverse order. */
     Z' := ∅;
     for j := 1 to m do
      if eval_single_context(node(ei),
      ⟨zj, j, m⟩) = true then Z' := Z' ∪ {zj};
     Z := Z'; /* i.e., Z = {z ∈ dom |
     xχz and e1,...,ei hold} */
    end; /* for i */
    R := R ∪ Z;
   end; /* for each x */
  end; /* some ei depends on 'cp' or 'cs' */
  return R;
 end; /* case distinction over expr(N) */
end;
```

procedure eval_by_cnode_only:

input: node N in the parse tree
set X of context-nodes (If 'cn' $\notin$ $Relev(N)$, then X may consist of the wildcard "$*$" only.)

output: modifies the global data $table(M)$ of nodes M below N in the parse tree.

```
begin
 if {'cp','cs'} ∩ Relev(N) ≠ ∅ then
 begin
  let N1,...,Nk be the child nodes of N
   in the parse tree;
  for i := 1 to k do
   eval_by_cnode_only(Ni, X);
 end;
 elseif expr(N) = π then
  table(N) := eval_inner_locpath(π, X);
 else
 begin
  let expr(N) = Op(e1,...,ek);
  for i := 1 to k do
   eval_by_cnode_only(node(ei), X);
  table(N) := {(c, F[[Op]](r1,...,rk) |
   ∃c ∈ X s.t. (∀i ∈ {1,...,k}) (ci, ri) ∈
   table(node(ei)) holds, where ci is the
   projection of c to the relevant context
   of node(ei)};
 end;
end;
```

procedure eval_single_context:

input: node N in the parse tree
single context triple $\langle x, p, s\rangle$, s.t. the wildcard "$*$" may be used for any irrelevant part of the context.

output: result value of $expr(N)$ for the context $\langle x, p, s\rangle$.

begin
if {'cp','cs'} $\cap$ $Relev(N) = \emptyset$ **then**
begin
let $(c, r) \in table(N)$ where c is the projection of $\langle x, p, s\rangle$ to the relevant context of N;
return r;
/* i.e. result value according to $table(N)$ */
end;
else
begin
let $expr(N) = Op(e_1, \ldots, e_k)$;
for $i := 1$ **to** k **do** $r_i :=$
$eval\_single\_context(node(e_i), \langle x, p, s\rangle)$;
return $\mathcal{F}[\![Op]\!](r_1, \ldots, r_k)$;
end;
end;

procedure eval_inner_locpath:

input: node N in the parse tree
set X of possible context-nodes

output: $table(N) \subseteq \text{dom} \times 2^{\text{dom}}$

begin
/* case distinction over $expr(N)$ */
if $expr(N) = /\pi$ **then**
begin
$R' := eval\_inner\_locpath(node(\pi), \{root\})$;
return $\{(x_0, x) \mid x_0 \in X \wedge (root, x) \in R'\}$;
end;
elseif $expr(N) = \pi_1|\pi_2$ **then**
begin
$R_1 := eval\_inner\_locpath(node(\pi_1), X)$;
$R_2 := eval\_inner\_locpath(node(\pi_2), X)$;
return $R_1 \cup R_2$;
end;
elseif $expr(N) = \pi_1/\pi_2$ **then**
begin
$R_1 := eval\_inner\_locpath(node(\pi_1), X)$;
let $Y := \{x \mid \exists x_0\text{: } (x_0, x) \in R_1\}$;
$R_2 := eval\_inner\_locpath(node(\pi_2), Y)$;
return $\{(x_0, x) \mid \exists x_1\text{: } (x_0, x_1) \in R_1 \wedge$
$(x_1, x) \in R_2\}$;
end;
elseif $expr(N) = \chi :: t[e_1] \ldots [e_q]$ **then**
begin
$Y :=$ nodes reachable from X via $\chi :: t$;
for $i := 1$ **to** q **do**
$eval\_by\_cnode\_only(node(e_i), Y)$;
if $(\forall i \in \{1, \ldots, q\})$ ({'cp','cs'}$\cap$
$Relev(node(e_i))) = \emptyset$ holds **then**
begin
$Y' := \emptyset$;
for each $y \in Y$ **do**
if $\forall i \in \{1, \ldots, q\}$
$eval\_single\_context(node(e_i), \langle y, *, *\rangle) =$
true **then** $Y' := Y' \cup \{y\}$;
$R := \{(x, y) \mid x \in X \wedge y \in Y' \wedge x\chi y\}$;
end;
else /* some e_i depends on 'cp' or 'cs' */
begin
$R := \emptyset$;
for each $x \in X$ **do**
begin
$Z := \{z \in Y \mid x\chi z\}$;
for $i := 1$ **to** q **do**
begin
let $Z = \{z_1, \ldots, z_m\}$ be ordered according to the axis χ; /* i.e., in document order or reverse order. */
$Z' := \emptyset$;
for $j := 1$ **to** m **do if**
$eval\_single\_context(node(e_i), \langle z_j, j, m\rangle)$
= true **then** $Z' := Z' \cup \{z_j\}$;
Z := Z'; /* $Z = \{z \in \text{dom} \mid$
$x\chi z$ and $e_1, \ldots, e_i$ hold$\}$ */
end; /* for i */
$R := R \cup (\{x\} \times Z)$;
end; /* for each x */
end; /* some e_i depends on 'cp' or 'cs' */
return R;
end; /* case distinction over $expr(N)$ */
end;

eval_bottomup_path:

input: node N in the parse tree with $expr(N) \equiv$ boolean(π) or $expr(N) \equiv$ π RelOp c, s.t. π is a "bottom-up location path", c is independent of the context, and c is of type nset, str, or num.

output: The global data structure $table(N)$ is filled in.

begin
/* Step 1: determine the initial node set Y */
if $expr(N) =$ boolean(π) **then** $Y :=$ dom;
elseif $expr(N) = \pi$ RelOp c **then**
begin
$eval\_by\_cnode\_only(node(s), \{*\})$; /* note that c is independent of the context */
if c is of type nset **then**
begin
$Y := \{y \mid \exists z \in table(node(c)) \mid$
strval(y) RelOp strval(z)$\}$;
end;

```
  elseif c is of type str then
  begin
   let val denote the only element in
     table(node(s));
   Y := {y | strval(y) RelOp val};
  end;
  elseif c is of type num then
  begin
   let val denote the only element in
     table(node(c));
   Y := {y | to_number(strval(y)) RelOp val};
  end;
 end;
/* Step 2: propagate Y backwards via π and
 fill in table(N) */
 let M1 := node(π);  /* i.e., M1 corresponds
  to the first location step of π */
 let M2 denote the node in the parse tree
  corresponding to the last location step of π;
 X := propagate_path_backwards(Y, M1, M2);
 table(N) := {(x, true) | x ∈ X}∪
  {(x, false) | x ∈ (dom − X)};
end;
```

propagate_path_backwards:

```
input: node set Y ⊆ dom
       nodes M1 and M2 in the parse tree
         (corresponding to first/last step of a
         "bottom-up path" π)
output: node set X ⊆ dom, where
          X := {x ∈ dom | ∃y ∈ Y, s.t.
          y is reachable from x via π}
begin
 if Y = ∅ then return ∅;
 /* case distinction over all possible
 location steps at M2: */
 if location step at M2 is '/' then R := dom;
 /* this is the top of an absolute location
 path and Y ≠ ∅ holds */
 elseif location step at M2 is id then
 begin
  R := F[[Op]]^-1(Y);
 end;
 elseif location step at M2 is χ :: t[e1]...[eq]
 then begin
  Y' := {y ∈ Y |  node test t is true for y};
  for i := 1 to q do
   eval_by_cnode_only(node(ei), Y');
  if (∀i ∈ {1,...,q}) ({'cp','cs'}∩
  Relev(node(ei))) = ∅ holds then
  begin
   Y'' := ∅;
   for each y ∈ Y' do
    if ∀i ∈ {1,...,q}
     eval_single_context(node(ei), ⟨y, *, *⟩) =
       true then Y'' := Y'' ∪ {y};
   R := χ^-1(Y'');
  end;
  else  /* some ei depends on 'cp' or 'cs' */
  begin
   X' := χ^-1(Y');
   R := ∅;
   for each x ∈ X' do
   begin
    Z := {z ∈ Y' | xχz};
    for i := 1 to q do
    begin
     let Z = {z1,...,zm} be ordered
      according to the axis χ;  /* i.e.,
      document order or reverse order. */
     Z' := ∅;
     for j := 1 to m do
      if eval_single_context(node(ei),
       ⟨zj, j, m⟩) = true then
       Z' := Z' ∪ {zj};
     Z := Z'; /* i.e., Z = {z ∈ dom |
      xχz and e1,...,ei hold} */
    end;  /* for i */
    if Z ≠ ∅ then R := R ∪ {x};
   end; /* for each x */
  end;  /* some ei depends on 'cp' or 'cs' */
 end;  /* case distinction over M2 */
 if M1 = M2 then return R; /* i.e., we have
  reached the top of the location path */
 else
 begin
  let M2' be the father node of M2;
   /* i.e., M2' corresponds to the location
   step above M2 in π */
  return
   propagate_path_backwards(R, M1, M2');
 end;
end;
```