XPath Query Evaluation: Improving Time and Space Efficiency*

Georg Gottlob
Inst. für Informationssysteme
Technische Universität Wien
A-1040 Vienna, Austria
gottlob@dbai.tuwien.ac.at

Christoph Koch
Inst. für Informationssysteme
Technische Universität Wien
A-1040 Vienna, Austria
koch@dbai.tuwien.ac.at

Reinhard Pichler
Inst. für Computersprachen
Technische Universität Wien
A-1040 Vienna, Austria
reini@logic.tuwien.ac.at

Abstract

Contemporary XPath query engines evaluate queries in time exponential in the sizes of input queries, a fact that has gone unnoticed for a long time. Recently, the first main-memory evaluation algorithm for XPath 1.0 with polynomial time combined complexity, i.e., which runs in polynomial time both with respect to the size of the data and the queries, has been published (cf. [11]).

In this paper, we present several important improvements and extensions of that work, including new XPath processing algorithms with improved time and space efficiency. Moreover, we define a very large and practically relevant fragment of XPath for which a further optimized form of query evaluation is possible. Apart from its immediate relevance for XPath query processing, our work also sheds new light at those features of XPath 1.0 which are most costly relative to their practical usefulness.

1. Introduction

XPath is a distinguished member of a whole family of XML-related technologies proposed by the W3C (such as XSLT, XPointer, and XQuery, cf. [1]), since most of these other technologies use XPath as their core mechanism for addressing nodes in XML documents. In order to use XPath successfully in practice, XPath processors must run efficiently both w.r.t. the size of the XML data and the growing size and intricacy of the queries (usually referred to as combined complexity).

In the past few years, there has been some work on related problems such as query containment for XPath [8, 13, 17], XPath query transformation and optimization [5, 12], and contributions towards a formal semantics definition of XPath [9, 16, 10]. Moreover, the expressiveness and complexity of various fragments of XSLT [3, 14, 2] and XML query pattern matching [15, 4, 6] have been investigated. However, only very recently, the first *polynomial-time* algorithm for evaluating *arbitrary* XPath 1.0 queries has been published (cf. [11]). In contrast, experiments with three existing XPath processors (namely XALAN [19], XT [7], and Microsoft Internet Explorer 6) revealed that they all consume time exponential in the size of the queries in the worst case (cf. [11]).

The main contributions of this paper are the following:

- In [11], two approaches for evaluating XPath 1.0 expressions were presented. The more efficient one (referred to as "top-down" evaluation) works in time $O(|D|^5 * |Q|^2)$ and space $O(|D|^4 * |Q|^2)$, where $|D|$ denotes the size of the data and $|Q|$ is the size of the query. Even though this is clearly better than previous exponential time algorithms, it is still not fully satisfactory. In this paper, we present a new algorithm MINCONTEXT which allows us to push down the time complexity to $O(|D|^4 * |Q|^2)$ and the space complexity even to $O(|D|^2 * |Q|^2)$. As the size of data usually dominates the size of queries, reducing the degree of $|D|$ in these bounds clearly substantially improves the practical scalability of XPath evaluation algorithms.

- We define the *Extended Wadler Fragment*, a very large fragment of XPath 1.0 for which we provide an evaluation algorithm that works in time $O(|D|^2 * |Q|^2)$ and space $O(|D| * |Q|^2)$. This fragment is of great practical value, since the vast majority of useful queries fall into it. Moreover, it pinpoints those features of XPath 1.0 that are the most "expensive", even though their practical value is questionable.

- We present the algorithm OPTMINCONTEXT, which combines our previous results into one query processor with the following properties. (a) It supports all of XPath 1.0, with the improved runtime bounds obtained in this paper. Moreover, (b) for (subexpressions of)

* This work was supported by the Austrian Science Fund (FWF) under project No. Z29-INF. All methods and algorithms presented in this paper are covered by a pending patent.

queries that fall either into the quadratic-time, linear-space Extended Wadler Fragment or the linear-time *Core XPath Fragment* introduced in [11], the OPTMINCONTEXT algorithm adheres to these best known bounds.

2. Preliminaries

2.1. Data Model

An XML document can be thought of as an unranked, ordered, and labeled tree. By dom we denote the set of all nodes in this tree. Let Σ be a labeling alphabet (i.e., of "tags"). We define a function $T\colon (\Sigma \cup \{*\}) \to 2^{\text{dom}}$ ("node test") which assigns to each label (XML tag) the set of nodes labeled with it; moreover, $T(*) :=$ dom.

The document tree is represented by a number of binary *axis* relations $\chi \subseteq \text{dom} \times \text{dom}$. We consider the axes *self*, *child*, *parent*, *descendant*, *ancestor*, *descendant-or-self*, *ancestor-or-self*, *following*, *preceding*, *following-sibling*, and *preceding-sibling* (which carry the intuitive semantics defined in [18, 11]). For the actual computation of node sets resulting from a location step via an axis relation χ, we define the corresponding axis-function and also an inverse axis-function.

Definition 1 *For an XPath axis relation χ, we define the function $\chi : 2^{\text{dom}} \to 2^{\text{dom}}$ (and thus overload the relation name χ) as $\chi(X) = \{y \in \text{dom} \mid \exists x \in X : x\,\chi\,y\}$. Moreover, the* inverse axis *function $\chi^{-1} : 2^{\text{dom}} \to 2^{\text{dom}}$ is defined as $\chi^{-1}(Y) := \{x \in \text{dom} \mid \chi(\{x\}) \cap Y \neq \emptyset\}$.*

In [11], it is shown that for any axis χ and any node set $X \subseteq$ dom, the sets $\chi(X)$ and $\chi^{-1}(X)$ can be computed in time linear w.r.t. the size $|D|$ of the data, viz $O(|D|)$.

In order to keep the notation simple, all nodes are assumed to be of the same type; thus, we do not distinguish between element, attribute, and processing instruction nodes, among others. Moreover, for lack of space, we do not discuss the "namespace" and "attribute" axes as well as the "local-name", "namespace-uri", and "name" core library functions [18].

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

Each node in an XML document may be identified by a unique id. The function deref_ids : string $\to 2^{\text{dom}}$ interprets its input string as a whitespace-separated list of keys and returns the set of nodes whose id's are contained in that list. The function strval : dom $\to$ string returns the string value of a node, which is the concatenation of non-tag strings and non-comment strings between the node's start and end tags in the document. The functions to_string and to_number convert a number to a string or a string to a number, respectively, according to the rules specified in [18].

2.2. Syntax and Semantics of XPath 1.0

Concerning the *syntax of XPath 1.0* we stick to its *unabbreviated* form (see [18]). W.l.o.g., we assume that all type conversions are made explicit (using the conversion functions string, number, and boolean). Moreover, each variable is replaced by the (constant) value of the input variable binding.

The main structural feature of XPath are *expressions*, which are of one of four types, namely node set, number, string, or boolean. Every expression evaluates relative to a *context* consisting of a *context-node*, a *context-position*, and a *context-size* [18]. We represent these four types (denoted nset, num, str, and bool) using relations as shown in the table below, where $\mathbf{C} = \{\langle cn, cp, cs\rangle \mid cn \in \text{dom}$ and $1 \leq cp \leq cs \leq |\text{dom}|\}$ is the domain of contexts.

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

For the semantics function $\mathcal{E}_\downarrow$ in Definition 2 below, we introduce the following notation: 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, $\langle X_1, \ldots, X_k\rangle \cup^{\langle\rangle} \langle Y_1, \ldots, Y_k\rangle :=$
$\langle X_1 \cup Y_1, \ldots, X_k \cup Y_k\rangle$ with $X_i, Y_j \subseteq$ dom.

Definition 2 *Given an XPath expression e and a list $(\vec{c}_1, \ldots, \vec{c}_l)$ of contexts, the semantics function*

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

determines a list $\langle r_1, \ldots, r_l\rangle$ of results of one of the XPath types number, string, boolean, or node set with

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

$\mathcal{F}[\![\text{constant number } v : \rightarrow \text{num}]\!]() := v$
$\mathcal{F}[\![ArithOp : \text{num} \times \text{num} \rightarrow \text{num}]\!](v_1, v_2) :=$
$v_1\ ArithOp\ v_2$
$\mathcal{F}[\![\text{count} : \text{nset} \rightarrow \text{num}]\!](S) := |S|$
$\mathcal{F}[\![\text{sum} : \text{nset} \rightarrow \text{num}]\!](S) := \Sigma_{n \in S}\ \text{to\_number}(\text{strval}(n))$
$\mathcal{F}[\![\text{id} : \text{nset} \rightarrow \text{nset}]\!](S) := \bigcup_{n \in S}\ \text{deref\_ids}(\text{strval}(n))$
$\mathcal{F}[\![\text{id} : \text{str} \rightarrow \text{nset}]\!](s) := \text{deref\_ids}(s)$
$\mathcal{F}[\![\text{constant string } s : \rightarrow \text{str}]\!]() := s$
$\mathcal{F}[\![\text{and} : \text{bool} \times \text{bool} \rightarrow \text{bool}]\!](b_1, b_2) := b_1 \wedge b_2$
$\mathcal{F}[\![\text{or} : \text{bool} \times \text{bool} \rightarrow \text{bool}]\!](b_1, b_2) := b_1 \vee b_2$
$\mathcal{F}[\![\text{not} : \text{bool} \rightarrow \text{bool}]\!](b) := \neg b$
$\mathcal{F}[\![\text{true}() : \rightarrow \text{bool}]\!]() := \text{true}$
$\mathcal{F}[\![\text{false}() : \rightarrow \text{bool}]\!]() := \text{false}$

$\mathcal{F}[\![RelOp : \text{nset} \times \text{nset} \rightarrow \text{bool}]\!](S_1, S_2) :=$
$\exists n_1 \in S_1, n_2 \in S_2 : \text{strval}(n_1)\ RelOp\ \text{strval}(n_2)$
$\mathcal{F}[\![RelOp : \text{nset} \times \text{num} \rightarrow \text{bool}]\!](S, v) :=$
$\exists n \in S : \text{to\_number}(\text{strval}(n))\ RelOp\ v$
$\mathcal{F}[\![RelOp : \text{nset} \times \text{str} \rightarrow \text{bool}]\!](S, s) :=$
$\exists n \in S : \text{strval}(n)\ RelOp\ s$
$\mathcal{F}[\![RelOp : \text{nset} \times \text{bool} \rightarrow \text{bool}]\!](S, b) :=$
$\mathcal{F}[\![\text{boolean}]\!](S)\ RelOp\ b$
$\mathcal{F}[\![EqOp : \text{bool} \times (\text{str} \cup \text{num} \cup \text{bool}) \rightarrow \text{bool}]\!](b, x) :=$
$b\ EqOp\ \mathcal{F}[\![\text{boolean}]\!](x)$
$\mathcal{F}[\![EqOp : \text{num} \times (\text{str} \cup \text{num}) \rightarrow \text{bool}]\!](v, x) :=$
$v\ EqOp\ \mathcal{F}[\![\text{number}]\!](x)$
$\mathcal{F}[\![EqOp : \text{str} \times \text{str} \rightarrow \text{bool}]\!](s_1, s_2) := s_1\ EqOp\ s_2$
$\mathcal{F}[\![GtOp : (\text{str} \cup \text{num} \cup \text{bool}) \times (\text{str} \cup \text{num} \cup \text{bool}) \rightarrow$
$\text{bool}]\!](x_1, x_2) := \mathcal{F}[\![\text{number}]\!](x_1)\ GtOp\ \mathcal{F}[\![\text{number}]\!](x_2)$

$\mathcal{F}[\![\text{string} : \text{num} \rightarrow \text{str}]\!](v) := \text{to\_string}(v)$
$\mathcal{F}[\![\text{string} : \text{nset} \rightarrow \text{str}]\!](S) :=$
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** $v \neq NaN$ **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
$\mathcal{F}[\![\text{number} : \text{nset} \rightarrow \text{num}]\!](S) := \mathcal{F}[\![\text{number}]\!](\mathcal{F}[\![\text{string}]\!](S))$

Figure 1. Effective semantics function $\mathcal{F}$.

$\langle k_1, \ldots, k_l \rangle$
$\mathcal{E}_\downarrow[\![last()]\!](\langle x_1, k_1, n_1 \rangle, \ \ldots, \ \langle x_l, k_l, n_l \rangle) := \langle n_1, \ldots, n_l \rangle$
$\mathcal{E}_\downarrow[\![Op(e_1, \ldots, e_m)]\!](\vec{c}_1, \ldots, \vec{c}_l) :=$
$\mathcal{F}[\![Op]\!]^{\langle\rangle}(\mathcal{E}_\downarrow[\![e_1]\!](\vec{c}_1, \ldots, \vec{c}_l), \ldots, \mathcal{E}_\downarrow[\![e_m]\!](\vec{c}_1, \ldots, \vec{c}_l))$

The effective semantics function $\mathcal{F}$ for other XPath 1.0 operations Op is defined in Figure 1 (for lack of space, several string and number operations were omitted, cf. [11]).

For location paths π, the auxiliary semantics function

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

is defined as follows:

$\mathcal{S}_\downarrow[\![\chi::t[e_1] \cdots [e_m]]\!](X_1, \ldots, X_k) :=$
begin
$S := \{\langle x, y \rangle \mid x \in \bigcup_{i=1}^{k} X_i,\ x\ \chi\ y$, *and* $y \in T(t)\}$;
for each $1 \leq i \leq m$ *(in ascending order)* **do**
begin
fix some order $\vec{S} = \langle \langle x_1, y_1 \rangle, \ldots, \langle x_l, y_l \rangle \rangle$ *for* S;
$\langle r_1, \ldots, r_l \rangle := \mathcal{E}_\downarrow[\![e_i]\!](t_1, \ldots, t_l)$ *where*
$t_j = \langle y_j,\ idx_\chi(y_j, S_j),\ |S_j| \rangle$ *and* $S_j := \{z \mid \langle x_j, z \rangle \in S\}$;
$S := \{\langle x_i, y_i \rangle \mid r_i$ *is true*$\}$;
end;
for each $1 \leq i \leq k$ **do** $R_i := \{y \mid \langle x, y \rangle \in S, x \in X_i\}$;
return $\langle R_1, \ldots, R_k \rangle$;
end;

$$\mathcal{S}_\downarrow[\![/\pi]\!](X_1, \ldots, X_k) \ := \ \mathcal{S}_\downarrow[\![\pi]\!](\overbrace{\{root\}, \ldots, \{root\}}^{k \text{ times}})$$

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

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

2.3. The Context-Value Table Principle

The main principle proposed in [11] to obtain an XPath 1.0 evaluation algorithm with polynomial-time complexity is the notion of a *context-value table* (i.e., a relation for each expression, as described in Section 2.2). It works as follows: 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 carried out in polynomial time (which was in fact shown in [11]), query evaluation in total under this principle has a polynomial time bound. The resulting semantics function was called $\mathcal{E}_\uparrow$ in [11], indicating that this is a strict *bottom-up* evaluation.

The second method of [11] for evaluating XPath – which corresponds to the semantics function $\mathcal{E}_\downarrow$ recalled in Definition 2 – follows a *top-down* intuition. Even if not immediately obvious, it is closely based on the former bottom-up method, however avoiding the computation of parts of intermediate context-value tables of subexpressions that are never used in subsequent computations.

2.4. The Running Example

Sample XML document. Our algorithms will be illustrated by applying various sample XPath queries to the XML document in Figure 2. 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

```
<?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>
```

Figure 2. Sample XML document.

to the element whose attribute "id" has the value i. We thus have dom $= \{x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$.

Sample XPath query. Let $e \equiv$ /descendant::*/descendant::*[position() > last()*0.5 or self::* = 100] be an XPath query that has to be evaluated for the context $\langle x_{10}, 1, 1\rangle$. The parse tree $\mathcal{T}$ of e is depicted in Figure 3. The correspondence between the nodes in $\mathcal{T}$ and the subexpressions of e is given in the table below. Actually, we have slightly simplified $\mathcal{T}$ in that we have omitted two separate child nodes of N_7 corresponding to the subexpressions last() and 0.5 of last() * 0.5.

Node in $\mathcal{T}$	Subexpression of e
N_1	/descendant::*/descendant::*[position() > last()*0.5 or self::* = 100]
N_2	descendant::*[position() > last()*0.5 or self::* = 100]
N_3	position() > last()*0.5 or self::* = 100
N_4	position() > last()*0.5
N_5	self::* = 100
N_6	position()
N_7	last()*0.5
N_8	self::*
N_9	100

The context-value table of each node in $\mathcal{T}$ is depicted in Figure 4. By "cn", "cp", and "cs", we denote context-node, context-position, and context-size. In the last column of each table, we have the result "res". Note that the context-value tables of the nodes N_1 and N_2 have been simplified in several ways: In both tables, we have omitted the columns for the context-position and the context-size. This is justified since "cp" and "cs" are irrelevant for the path expressions corresponding to the nodes N_1 and N_2. We shall come back to this point in Section 3.1. In the table corresponding to N_1, the result (of the *absolute* location path e) is the same for all possible contexts. We have only filled in the first row of this table in Figure 4. Moreover, in the table corresponding to N_2, the resulting node set is empty for all values of cn except for $\{x_{10}, x_{11}, x_{21}\}$. We have omitted the remaining rows of the table corresponding to N_2 in Figure 4 since they have no effect on the overall result of evaluating e anyway.

In the tables corresponding to the nodes $N_3, \ldots, N_9$, the improvement due to the *top-down* evaluation according to the semantics function $\mathcal{E}_\downarrow$ from Definition 2 becomes apparent: Rather than considering all $|\text{dom}|^3$ possible triples $\langle cn, cp, cs\rangle$, it suffices to evaluate "position() > last()*0.5 or self::* = 100" and its subexpressions for those values of $\langle cn, cp, cs\rangle$ which can be reached by the preceding location steps "/descendant::*/descendant::*". More generally, the 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. to the axis in the last location step.

The final result of evaluating e is $\{x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$. It can be read out from the context-value table of the root node N_1 of $\mathcal{T}$.

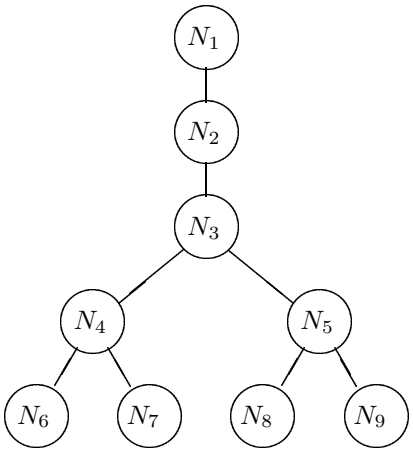

Figure 3. Parse tree $\mathcal{T}$ of e.

In the sequel, it is convenient to use the following notation for nodes N in the parse tree: By $expr(N)$ we 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. By $table(N)$, we denote the context-value table at the node N. Finally, it is convenient to write "$\equiv$" for syntactic equality.

3. The Algorithm MINCONTEXT

3.1. 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:

N_1

cn	res
x_{10}	$\{x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$

N_2

cn	res
x_{10}	$\{x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$
x_{11}	$\{x_{13}, x_{14}\}$
x_{21}	$\{x_{23}, x_{24}\}$

N_3

cn	cp	cs	res
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"

N_4

cn	cp	cs	res
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"

N_5

cn	cp	cs	res
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"

N_6

cn	cp	cs	res
x_{11}	1	8	1
x_{12}	2	8	2
x_{13}	3	8	3
⋮	⋮	⋮	⋮
x_{22}	1	3	1
x_{23}	2	3	2
x_{24}	3	3	3

N_7

cn	cp	cs	res
x_{11}	1	8	4
x_{12}	2	8	4
⋮	⋮	⋮	⋮
x_{12}	1	3	1.5
⋮	⋮	⋮	⋮
x_{24}	3	3	1.5

N_8

cn	cp	cs	res
x_{11}	1	8	$\{x_{11}\}$
x_{12}	2	8	$\{x_{12}\}$
x_{13}	3	8	$\{x_{13}\}$
⋮	⋮	⋮	⋮
x_{24}	3	3	$\{x_{24}\}$

N_9

cn	cp	cs	res
x_{11}	1	8	100
x_{12}	2	8	100
x_{13}	3	8	100
⋮	⋮	⋮	⋮
x_{24}	3	3	100

Figure 4. Context-value tables of e.

Restriction to the relevant context. Suppose that we want to evaluate an 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. (Actually, this is already an improved bound due to the top-down evaluation via the semantics function $\mathcal{E}_\downarrow$. With a strict bottom-up evaluation via the function $\mathcal{E}_\uparrow$ mentioned in Section 2.3, this bound even deteriorates to $|\text{dom}|^3$). 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_i in the parse tree to the "*relevant context*" $Relev(N_i) \subseteq \{\text{'cn'}, \text{'cp'}, \text{'cs'}\}$, 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 input 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 3 For the leaf nodes of $\mathcal{T}$ in Figure 3, we have $Relev(N_6) = \{\text{'cp'}\}$, $Relev(N_7) = \{\text{'cs'}\}$, $Relev(N_8) = \{\text{'cn'}\}$, and $Relev(N_9) = \emptyset$. The nodes N_1 and N_2 in $\mathcal{T}$ correspond to location paths. Hence, $Relev(N_1) = Relev(N_2) = \{\text{'cn'}\}$ holds. Finally, for the remaining inner nodes of $\mathcal{T}$, we get $Relev(N_3) = Relev(N_4) = \{\text{'cn'}, \text{'cp'}, \text{'cs'}\}$, and $Relev(N_5) = \{\text{'cn'}\}$.

Note that the tables corresponding to N_1 and N_2 have already been reduced to the relevant context in Figure 4. For the nodes $Relev(N_3)$ and $Relev(N_4)$ no simplification is possible. Finally, in Figure 5, the reduced tables for the nodes $N_5, \ldots, N_9$ are shown. □

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

N_5: self::* = 100	
cn	res
x_{11}	"false"
x_{12}	"false"
x_{13}	"false"
x_{14}	"true"
x_{21}	"false"
x_{22}	"false"
x_{23}	"false"
x_{24}	"false"

N_6: position()	
cp	res
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8

N_8: self::*	
cn	res
x_{11}	$\{x_{11}\}$
x_{12}	$\{x_{12}\}$
x_{13}	$\{x_{13}\}$
x_{14}	$\{x_{14}\}$
x_{21}	$\{x_{21}\}$
x_{22}	$\{x_{22}\}$
x_{23}	$\{x_{23}\}$
x_{24}	$\{x_{24}\}$

N_7: last()*0.5	
cs	res
8	4
3	1.5

N_9: 100
res
100

Figure 5. Restriction to the relevant context.

steps of an input location path (according to the semantics function $\mathcal{S}_\downarrow$ recalled in Definition 2). This is due to the fact that we compute for every possible context-node cn the resulting node set. However, at no stage in the computation, we are really interested in the whole information as to which next node $x_j \in$ dom can be reached from which previous node $x_i \in$ dom. Instead, it suffices to know the *set* of all nodes $x_j \in$ dom that can be reached from *any* of the previous nodes $x_i \in$ dom. Hence, the results of location steps on the outermost level should be treated as a subset $\subseteq$ dom rather than as a relation $\subseteq$ 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 4 The XPath query e from Section 2.4 has in fact a location path on the outermost level. Hence, the 2-dimensional context-value tables of the location paths "/descendant::*/descendant::*[...]" (at the node N_1) and "descendant::*[...] (at N_2) can be replaced by the node sets (or, equivalently, the 1-dimensional tables) $X = \{x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$ (at N_1) and $Y = \{x_{13}, x_{14}, x_{21}, x_{22}, x_{23}, x_{24}\}$ (at N_2), respectively. Then the final result of e is the node set Y corresponding to the node N_2 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. Section 2) reveals that such a simultaneous evaluation for all possible contexts is only necessary (in order to avoid exponential time complexity) for the context-node cn. In contrast, for the context-position and/or context-size, a *loop over all possible values* $\langle cp, cs\rangle$ 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 $\langle cn, cp, cs\rangle$.

Example 5 Recall the query e from Section 2.4. 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 $e \equiv$ /descendant::*/descendant::* [$expr(N_3)$]. Now X has to be restricted in the following way to the set X' of those nodes for which $expr(N_3)$ evaluates to "true".

1. First, we traverse the subtree of the parse tree rooted at N_3 top-down and evaluate those parts, which are independent of the value of cp and cs at N_3. We thus set up the context-value tables of N_5, N_8, and N_9 as in Figure 5.

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 $expr(N_3)$ is true, i.e.: $X' := \{cn \in X \mid (\exists cp)(\exists cs)$ s.t. $expr(N_3)$ evaluates to "true" for the context $\langle cn, cp, cs\rangle\}$. Of course, this comes down to checking all the rows of the context-table at N_3 (and thus also of N_4, N_6, and N_7). However, in contrast to Figure 4, we do not set up the entire tables at once. Instead, we treat these contexts $\langle cn, cp, cs\rangle$ in a loop, e.g.: for $\langle cn, cp, cs\rangle = \langle x_{23}, 7, 8\rangle$ we compute the rows of $table(N_4)$, $table(N_6)$, and $table(N_7)$ for $cp = 7$ and $cs = 8$ only. Moreover, we look up the row of N_5 for the value $cn = x_{23}$. We thus get the overall value "true" of the predicate $expr(N_3)$ for this single context $\langle x_{23}, 7, 8\rangle$. Hence, x_{23} is added to X'. □

3.2. Procedures of the Algorithm MINCONTEXT

The MINCONTEXT algorithm consists of three main procedures $eval\_outermost\_locpath$, $eval\_by\_cnode\_only$, and $eval\_single\_context$. They are briefly explained below. In Section 6, a pseudo-code presentation of these procedures 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 $X \subseteq$ 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 $\langle cn, cp, cs\rangle$. It takes a node N in the parse tree and a context $\langle cn, cp, cs\rangle$ as input and returns the result value of $expr(N)$ for this context. $eval\_single\_context$ may only be called after the procedure $eval\_by\_cnode\_only$ has been called for the node N.

In Section 6, we shall also give the pseudo-code 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.

Now let e be an input XPath expression and $\langle cn, cp, cs\rangle$ the input context. Moreover, let R denote the root node in the parse tree of e. Then we have:

Algorithm 6 (MINCONTEXT)

if e is a location path **then**
  return $eval\_outermost\_locpath(R, \{cn\})$;
else
  $eval\_by\_cnode\_only(R, \{cn\})$;
  return $eval\_single\_context(R, \langle cn, cp, cs\rangle)$;
fi;

The MINCONTEXT algorithm will be put to work in a detailed example at the end of Section 4. Below, we state the main result of this section:

Theorem 7 *XPath can be evaluated in time* $O(|D|^4 * |Q|^2)$ *and space* $O(|D|^2 * |Q|^2)$.

Proof Sketch. 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 $Relev(N) \subseteq \{\text{'cp'}\}$). Of course, there are at most $|Q|$ context-value tables required. It remains to show that the result value of any subexpression e of Q for any context-node $cn \in$ dom is restricted by $O(|D| * |Q|)$. In case of the result types bool and nset, this is clearly the case. As for the result types str and num, we observe that values selected from the XML document D are bounded by $|D|$. Moreover, of all the XPath core library functions, only concat may possibly produce a string that is longer than its arguments. But then the resulting string is clearly bounded by $|D| * |Q|$.

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 $\langle cn, cp, cs\rangle$ corresponding to previous/current context-node). It can be shown by induction on the structure of e that the time required for computing each result value is bounded by $O(|D|^2 * |Q|)$. □

4. The Extended Wadler Fragment

In [16], 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 [16] 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 3. 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 [11] as the very reason why previous XPath evaluation algorithms require exponential time. Instead, we need a different strategy. Recall 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 $e \equiv$ id(id(...(s)...)) or $e \equiv$ id(id(...(π)...)), where s is a string-expression and π is a location path. For the latter case, we rewrite id(id(...(π)...)) to the form π/id/id/.../id. In other words, we consider "id" as a new axis. Hence, in this case, expressions of the form id(id(...(π)...)) are treated as location paths. For the former case, we impose

Restriction 3. In expressions of the form id(id(...(s)...)), where s is a string-expression, we require that s must not depend on any context. □

Actually, $nset$-expressions of the form id(id(...(s)...)), where s does not depend on any context, can be simply evaluated by the algorithm from Section 3 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 s has an $\exists$-semantics, e.g., boolean(π) evaluates to "true" for a context-node cn, iff *there exists* at least one node in the node set resulting from the evaluation of π. Thus, the set of nodes cn, for which boolean(π) or π RelOp s 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 s with s of type bool is treated like boolean(π) RelOp s. For any other type of s, we set $Y := \{cn \mid$ self::* RelOp s evaluates to "true" for the context-node $cn\}$.

- 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 Definition 1) for every $i \in \{1, \ldots, n\}$. Hence, $X := X_0$ is the desired node set. Note that if χ is the id-"axis", then we have $\chi_i^{-1}(X_i) = \mathcal{F}[\![Op]\!]^{-1}(X_i)$. Actually, in [11], it was shown that also $\mathcal{F}[\![Op]\!]^{-1}(X_i)$ 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 Section 6, we give the pseudocode of a procedure $eval\_bottomup\_path$ (plus the auxiliary procedure $propagate\_path\_backwards$) for expressions π RelOp s 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 s" by "boolean(π_1) or ...or boolean(π_k)" and "(π_1 RelOp s) or ...or (π_k RelOp s)".

5. 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 8 (OPTMINCONTEXT)

evaluate all "bottom-up location paths"
(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:

Example 9 Let Q denote an XPath query that is applied to the XML-document from Figure 2, where Q is defined as $Q \equiv$ /child::a/descendant::*[boolean(following::d[(position() != last()) and (preceding-sibling::*/preceding::* = 100)]/following::d)]. In order to facilitate the discussion, we assign names to some subexpressions of Q, namely: $Q \equiv$ /child::a/descendant::* [boolean(π)] with $\pi \equiv$ following::d[e_1 and e_2]/following::d, $e_1 \equiv$ position() != last(), $e_2 \equiv \rho = 100$, and $\rho \equiv$ preceding-sibling::*/preceding::*. The parse tree $\mathcal{T}$ of Q is depicted in Figure 6. The correspondence between subexpressions of Q and nodes in $\mathcal{T}$ is shown in the following table:

node	subexpression of Q	
N_1	/child::a/descendant::*[boolean(π)]	
N_2	descendant::*[boolean(π)]	
N_3	boolean(π)	
N_4	following::d[e_1 and e_2]/following::d	($\equiv \pi$)
N_5	e_1 and e_2	
N_6	following::d	
N_7	position() != last()	($\equiv e_1$)
N_8	$\rho = 100$	($\equiv e_2$)
N_9	position()	
N_{10}	last()	
N_{11}	preceding-sibling::*/preceding::*	($\equiv \rho$)
N_{12}	100	
N_{13}	preceding::*	

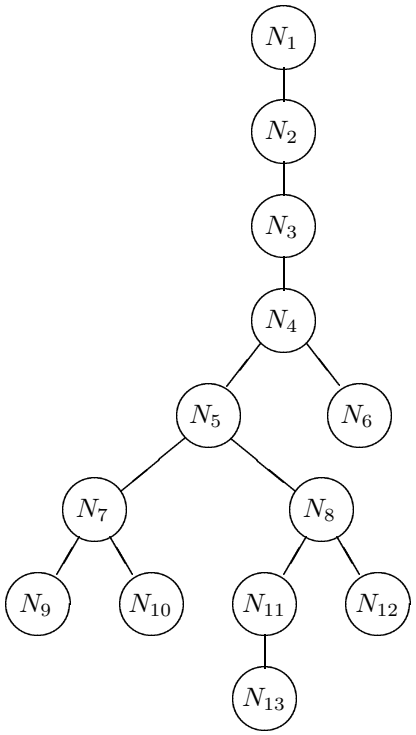

Figure 6. Parse tree of Q.

Q has two inner location paths π and ρ, which both have to be evaluated bottom-up. We start with the innermost one, namely ρ: 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 N_8 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 π, we have to take the node tests and the predicate "e_1 and e_2" 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_1 and e_2] 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 cp and cs) the predicate "e_1 and e_2" 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 N_5 which do not depend on (the current values of) cp and cs. Actually, in this case, only the subtree rooted at N_8 has this property. However, $table(N_8)$ has already been determined by a bottom-up evaluation. 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_1 and e_2" also for cp and cs 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 cp and cs to a context triple, s.t. "e_1 and e_2" 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_1 and e_2" does not lead to a restriction of Y'''. Therefore, the desired context-value table $\subseteq$ dom $\times$ {true, false} of the node N_3 has the value "true" in the second column exactly for the nodes in X.

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

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

Theorem 10 *The* Extended Wadler Fragment *(i.e., the set of all XPath expressions fulfilling the Restrictions 1 through 3 from Section 4) can be evaluated in space $O(|D| * |Q|^2)$ and time $O(|D|^2 * |Q|^2)$.*

It is easy to check, that even a slightly stronger property holds for our algorithm, namely:

Corollary 11 *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 $Q = e$ or e occurs in the form boolean(e) or e RelOp s (where s is independent of any context) in Q. Then e is evaluated in space $O(|D| * |e|^2)$ and time $O(|D|^2 * |e|^2)$. □

In [11], the so-called "*Core XPath*" was defined, which we recall below. It was shown in [11] that any XPath expression that is fully contained in this fragment, can be evaluated in linear time. An analogous result to Corollary 11 can also be shown for the Core XPath fragment.

Definition 12 *Let the Core XPath language be defined by its abstract EBNF syntax as follows:*

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

where "cxp" is the start production, χ stands for an axis, and t for a "node test".

Theorem 13 *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 that occurs as a subexpression in Q either on the outermost level or in the form boolean(π) or π RelOp s (where s is independent of any context). Then π is evaluated in time $O(|D| * |\pi|)$.*

Proof Sketch. Core XPath expressions of the form $\chi :: t[\pi']$ are a short-hand for $\chi :: t$[boolean(π')]. Hence, Core XPath is clearly contained in our linear space fragment. Actually, the only reason why we have quadratic time complexity in Theorem 10 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. We thus get a linear time upper bound for these subexpressions. □

6. Pseudo-Code Presentations

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
if $expr(N) = /\pi$ **then**
return $eval\_outermost\_locpath(node(\pi), \{root\})$;
elseif $expr(N) = \pi_1|\pi_2$ **then**
$Y_1 := eval\_outermost\_locpath(node(\pi_1), X)$;
$Y_2 := eval\_outermost\_locpath(node(\pi_2), X)$;
return $Y_1 \cup Y_2$;
elseif $expr(N) = \pi_1/\pi_2$ **then**
$Y := eval\_outermost\_locpath(node(\pi_1), X)$;
return $eval\_outermost\_locpath(node(\pi_2), Y)$;
elseif $expr(N) = \chi :: t[e_1] \ldots [e_q]$ **then**
$Y :=$ nodes reachable from X via $\chi :: t$;
for $i :=$ 1 **to** q **do** $eval\_by\_cnode\_only(node(e_i), Y)$; **od**;
$R := \emptyset$;
if $(\forall i \in \{1, \ldots, q\})$
$(\{$'cp', 'cs'$\} \cap Relev(node(e_i))) = \emptyset$ holds **then**
for each $y \in Y$ **do**
if $\forall i \in \{1, \ldots, q\}$ $eval\_single\_context(node(e_i),$
$\langle y, *, * \rangle) =$ true **then** $R := R \cup \{y\}$; **fi**;
od;
else /* i.e., at least one e_i depends on cp or cs */
for each $x \in X$ **do**
$Z := \{z \in Y \mid x \chi z\}$;
for $i :=$ 1 **to** q **do**
let $Z = \{z_1, \ldots, z_m\}$ ordered according to axis χ;
/* i.e., in document order or in 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** $Z' := Z' \cup \{z_j\}$; **fi**;
od;
Z := Z';
/* $Z = \{z \in$ dom $\mid x \chi z$ and $e_1, \ldots, e_i$ hold$\}$ */
od;
$R := R \cup Z$;
od;
fi; /* are all predicates e_i independent of the context? */
return R;
fi; /* case distinction over all possible forms of $expr(N)$ */
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 wild card "$*$" only.)

output: modifies the global data $table(M)$ of nodes M below N

begin
if $\{$'cp', 'cs'$\} \cap Relev(N) \neq \emptyset$ **then**
let $N_1, \ldots, N_k$ be the child nodes of N in the parse tree;
for $i :=$ 1 **to** k **do** $eval\_by\_cnode\_only(N_i, X)$;
elseif $expr(N) = \pi$ **then**
$table(N) := eval\_inner\_locpath(\pi, X)$;
else
let $expr(N) = Op(e_1, \ldots, e_k)$;
for $i :=$ 1 **to** k **do** $eval\_by\_cnode\_only(node(e_i), X)$; **od**;
$table(N) := \{(c, \mathcal{F}[\![Op]\!](r_1, \ldots, r_k) \mid \exists c \in X$ s.t.
$(\forall i \in \{1, \ldots, k\})$ $(c_i, r_i) \in table(node(e_i))$ holds,
where c_i is the projection of c to the relevant context
of $node(e_i)\}$;
fi;

end;

procedure eval_single_context:

input: node N in the parse tree
single context triple $\langle cn, cp, cs\rangle$, s.t. the wild card "$*$" may be used for each irrelevant part of the context.

output: result value of $expr(N)$ for the context $\langle cn, cp, cs\rangle$.

begin
if {'cp', 'cs'} $\cap$ $Relev(N) = \emptyset$ **then**
let $(c, r) \in table(N)$ with $c = proj_N(\langle cn, cp, cs\rangle)$;
return r; /* i.e., result value according to $table(N)$ */
else
let $expr(N) = Op(e_1, \ldots, e_k)$;
for $i := 1$ **to** k **do**
$eval\_single\_context(node(e_i), \langle cn, cp, cs\rangle)$; **od**;
return $\mathcal{F}[\![Op]\!](r_1, \ldots, r_k)$;
fi;
end;

procedure eval_inner_locpath:

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

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

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

eval_bottomup_path:

input: node N in the parse tree
$expr(N) \equiv$ boolean(π) or $expr(N) \equiv \pi$ RelOp s, s.t. π is a "bottom-up location path", s is independent of the context, and s 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 s **then**
$eval\_by\_cnode\_only(node(s), \{*\})$;
/* by assumption, s is independent of the context */
if s is of type nset **then**
$Y := \{y \mid \exists z \in table(node(s)) \mid$
strval(y) RelOp strval(z)$\}$;
elseif s is of type str **then**
let val denote the only element in $table(node(s))$;
$Y := \{y \mid$ strval(y) RelOp $val\}$;
elseif s is of type num **then**
let val denote the only element in $table(node(s))$;
$Y := \{y \mid$ to_number(strval(y)) RelOp $val\}$;
fi;
fi;

/* Step 2: propagate Y backwards via π and fill in $table(N)$ */
let $M_1 := node(\pi)$;
/* i.e., M_1 corresponds to the first location step of π */
let M_2 denote the node in the parse tree corresponding to the last location step of π;
$X := propagate\_path\_backwards(Y, M_1, M_2)$;
$table(N) := \{(x,$ true$) \mid x \in X\} \cup$
$\{(x,$ false$) \mid x \in ($dom $- X)\}$;
end;

propagate_path_backwards:

input: node set $Y \subseteq$ dom
nodes M_1 and M_2 in the parse tree
(corresponding to first/last step of a "bottom-up path" π)

output: node set $X \subseteq$ dom, where $X := \{x \in$ dom $\mid \exists y \in Y$, s.t. y is reachable from x via $\pi\}$

begin
if $Y = \emptyset$ **then return** $\emptyset$; **fi**;
if location step at M_2 is '/' **then**
/* i.e., this is the top of an absolute location path */
$R :=$ dom; /* the case $Y = \emptyset$ has already been treated */
elseif location step at M_2 is id **then**
$R := \mathcal{F}[\![Op]\!]^{-1}(Y)$;
elseif location step at M_2 is $\chi :: t[e_1] \ldots [e_q]$ **then**
$Y' := \{y \in Y \mid$ node test t is true for $y\}$;
for $i :=$ 1 **to** q **do** $eval\_by\_cnode\_only(node(e_i), Y')$; **od**;
if $(\forall i \in \{1, \ldots, q\})$
$(\{$'cp', 'cs'$\} \cap Relev(node(e_i))) = \emptyset$ holds **then**
$Y'' := \emptyset$;
for each $y \in Y'$ **do**
if $\forall i \in \{1, \ldots, q\}$ $eval\_single\_context(node(e_i),$
$\langle y, *, * \rangle) =$ true **then** $Y'' := Y'' \cup \{y\}$; **fi**;
od;
$R := \chi^{-1}(Y'')$;
else /* i.e., at least one e_i depends on context-position/size */
$X' := \chi^{-1}(Y')$;
$R := \emptyset$;
for each $x \in X'$ **do**
$Z := \{z \in Y' \mid x \chi z\}$;
for $i :=$ 1 **to** q **do**
let $Z = \{z_1, \ldots, z_m\}$ ordered according to axis χ;
$Z' := \emptyset$;
for $j :=$ 1 **to** m **do**
if $eval\_single\_context(node(e_i), \langle z_j, j, m \rangle) =$
true **then** $Z' := Z' \cup \{z_j\}$; **fi**;
od;
Z := Z';
/* $Z = \{z \in$ dom $\mid x \chi z$ and $e_1, \ldots, e_i$ hold$\}$ */
od;
if $Z \neq \emptyset$ **then** $R := R \cup \{x\}$;
od;
fi; /* are all predicates e_i independent of the context? */
fi; /* case distinction over all possible location steps at M_2*/
if $M_1 = M_2$ **then return** R;
/* i.e., we have reached the top of the location path */
else
let M_2' be the father node of M_2
/* i.e., M_2' corresponds to the location step above M_2 in π */
then return $propagate\_path\_backwards(R, M_1, M_2')$;
fi;
end;

7. Conclusion and Future Work

We have presented a new XPath 1.0 evaluation algorithm MINCONTEXT which leads to a significantly improved time and space complexity behavior w.r.t. a previous approach. Moreover, we have identified a very large fragment of XPath 1.0 that can be evaluated even more efficiently by another new algorithm which we called OPTMINCONTEXT.

The algorithms presented here aim at efficient main memory XPath processor implementations in the first place. However, improving the complexity bounds (w.r.t. the size of the XML data) is clearly of great relevance also to using our techniques for XPath processors that query XML documents stored in a database.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[2] M. Benedikt, W. Fan, and G. M. Kuper. Structural Properties of XPath Fragments. In *Proc. ICDT'03*, 2003. To appear.

[3] G. J. Bex, S. Maneth, and F. Neven. A Formal Model for an Expressive Fragment of XSLT. In *Proc. CL 2000*, LNCS 1861, pages 1137–1151. Springer, 2000.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holisitic Twig Joins: Optimal XML Pattern Matching. In *SIGMOD'02*, Madison, Wisconsin, 2002. ACM Press.

[5] F. Bry, D. Olteanu, H. Meuss, and T. Furche. Symmetry in XPath. Technical Report PMS-FB-2001-16, LMU München, 2001. Short version.

[6] C. Y. Chan, W. Fan, P. Felber, M. N. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proc. VLDB'02*, 2002.

[7] J. Clark. XT. A Java Implementation of XSLT, available at http://www.jclark.com/xml/xt.html/.

[8] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath. In *Proc. KRDB'01*, 2001.

[9] P. Fankhauser. A Mapping of XPath 1.0 to the XML Query Algebra (with J. Clark, M. Fernandez, and J. Siméon), Nov. 2001. Personal Communication.

[10] G. Gottlob and C. Koch. Monadic Queries over Tree-Structured Data. In *Proc. LICS'02*, 2002.

[11] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. VLDB'02*, 2002.

[12] J. McHugh and J. Widom. Query Optimization for XML. In *VLDB'99*, pages 315–326, 1999.

[13] G. Miklau and D. Suciu. Containment and Equivalence of XPath Expressions. In *Proc. PODS'02*, pages 65–76, 2002.

[14] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proc. PODS 2000*, pages 11–22, 2000.

[15] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE'02*, San Jose, California, 2002.

[16] P. Wadler. Two Semantics for XPath, 2000. availabe at http://www.research.avayalabs.com/user/wadler/.

[17] P. T. Wood. On the Equivalence of XML Patterns. In *CL 2000*, LNCS 1861, pages 1152–1166, 2000.

[18] World Wide Web Consortium. XPath Recommendation http://www.w3c.org/TR/xpath/.

[19] Xalan-Java version 2.2.D11. http://xml.apache.org/xalan-j/.