XPath Processing in a Nutshell*

Georg Gottlob, Christoph Koch, and Reinhard Pichler

Database and Artificial Intelligence Group
Technische Universität Wien, A-1040 Vienna, Austria
{gottlob, koch}@dbai.tuwien.ac.at, reini@logic.at

Abstract

We provide a concise yet complete formal definition of the semantics of XPath 1 and summarize efficient algorithms for processing queries in this language. Our presentation is intended both for the reader who is looking for a short but comprehensive formal account of XPath as well as the software developer in need of material that facilitates the rapid implementation of XPath engines.

1 Introduction

XPath [10] is a practical language for selecting nodes from XML document trees. Its importance stems from its suitability as an XML query language per se and it being at the core of several other XML-related technologies, such as XSLT, XPointer, and XQuery.

Previous semantics definitions of XPath have either been overly lengthy [10, 12], or were restricted to fragments of the language (e.g. [6, 7]). In this paper, we provide a formal semantics definition of full XPath, intended for the implementor or the reader who values a concise, formal, and functional description. As an extension to [3], we also provide a brief but complete definition of the XPath data model.

Given a query language, the prototypical algorithmic problem is *query evaluation*, and its hardness needs to be formally studied. The query evaluation problem was not addressed for full XPath in any depth until recently [3, 2, 4][1]. We summarize the main ideas of that work in this paper, with an emphasis on usefulness to a wide audience.

Since XPath and related technologies will be tested in ever-growing deployment scenarios, implementations of XPath engines need to scale well *both* with respect to the size of the XML data and the growing size and intricacy of the queries. (Database theoreticians refer to this as *combined complexity*.)

We present main-memory XPath processing algorithms that run in time guaranteed to be polynomial in the size of the complete input, i.e. query *and* data. As experimentally verified in [3], where these and other results where first presented, contemporary XPath engines do not have this property, and require time exponential in the size of queries at worst.

We present polynomial-time XPath evaluation algorithms in two flavors, *bottom-up* and *top-down*[2]. Both algorithms have the same worst-case polynomial-time bounds. The former approach comes with a clear and simple intuition why our algorithms are polynomial, while the latter requires to compute considerably fewer useless intermediate results and consequently performs much better in practice.

The structure of this paper is as follows. Section 2 introduces XPath axes. Section 3 presents the data model of XPath and auxiliary functions used throughout the paper. Section 4 defines the semantics of XPath in a concise way. Section 5 houses the bottom-up semantics definition and algorithm. Section 6 comes up with the modifications to obtain a top-down algorithm. We conclude with Section 7.

2 XPath Axes

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 3, 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 3.

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

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

* 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. Further resources, updates, and possible corrections are available at `http://www.xmltaskforce.com`.

[1] The novelty of our results is a bit surprising, as considerable effort has been made in the past to understand the XPath query containment problem (e.g. [1, 5, 8]), which – with its relevance to query optimization – is only a means to the end of efficient query evaluation.

[2] By this we refer to the mode of traversal of the expression tree of the query while computing the query result.

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

to represent its structure[3]. "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, 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 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 1 (cf. [10]). $R_1.R_2$, $R_1 \cup R_2$, 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 1. It is important to observe that some axes are defined in terms of other axes, but that these definitions are acyclic.

Definition 2.1 (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 $X_0 \subseteq$ dom is a set of nodes. □

Algorithm 2.2 (Axis Evaluation)
Input: A set of nodes S and an axis χ
Output: $\chi(S)$
Method: $\text{eval}_\chi(S)$

function $\text{eval}_{\text{self}}(S) := S$.
function $\text{eval}_\chi(S) := \text{eval}_{E(\chi)}(S)$.
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 $\text{eval}_{\chi_1 \cup \chi_2}(S) := \text{eval}_{\chi_1}(S) \cup \text{eval}_{\chi_2}(S)$.
function $\text{eval}_{(R_1 \cup \ldots \cup R_n)^*}(S)$ **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;

where $S \subseteq$ 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, χ_1 and χ_2 are axes, and χ is an axis other than "self". □

Clearly, some axes could have been defined in a simpler way in Table 1 (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 2.2.

The function $\text{eval}_{(R_1 \cup \ldots \cup R_n)^*}$ essentially computes graph reachability (not transitive closure). It can be implemented to run in linear time in terms of the data 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').

Lemma 2.3 *Let $S \subseteq$ dom be a set of nodes of an XML document and χ be an axis. Then, (1) $\chi(S) = \text{eval}_\chi(S)$ and (2) Algorithm 2.2 runs in time $O(|\text{dom}|)$.*

3 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 [9], 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. [10]). Nodes of all types besides "text" and "comment" have a name associated with it.

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, $T(\text{node}())$ = dom and $T(\text{attribute}(\text{href}))$ returns all attribute nodes labeled "href".

Example 3.1 Consider a document consisting of six nodes – the root node r, which is the parent of the document element node a (labeled "a"), and its four children $b_1, \ldots, b_4$ (labeled "b"). We have dom = $\{r, a, b_1, \ldots, b_4\}$, firstchild = $\{\langle r, a\rangle, \langle a, b_1\rangle\}$, nextsibling = $\{\langle b_1, b_2\rangle, \langle b_2, b_3\rangle, \langle b_3, b_4\rangle\}$, $T(\text{root}()) = \{r\}$, $T(\text{element}()) = \{a, b_1, \ldots, b_4\}$, $T(\text{element}(a)) = \{a\}$, and $T(\text{element}(b)) = \{b_1, \ldots, b_4\}$. □

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

Now, XPath axes differ from the abstract, untyped axes of Section 2 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 as

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

and for all other XPath axes χ (let χ_0 be the abstract axis of the same name),

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

Node tests that occur explicitly in XPath queries must not use the types "root", "attribute", or "namespace"[4]. 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 2.

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) [11] that uses the ID/IDREF feature, each element node 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 [10]. Notably, the string value of an element or root node x is the concatenation of the string values of descendant text nodes $\{y \mid \text{descendant}(\{x\}) \cap T(\text{text}())\}$ 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 [10].

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 penalty with respect to efficiency bounds) is an easy exercise. [5]

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

$P[\![\chi::t[e_1]\cdots[e_m]]\!](x) :=$
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;
$P[\![\pi_1|\pi_2]\!](x) := P[\![\pi_1]\!](x) \cup P[\![\pi_2]\!](x)$
$P[\![/\pi]\!](x) := P[\![\pi]\!](\text{root})$
$P[\![\pi_1/\pi_2]\!](x) := \bigcup_{y \in P[\![\pi_1]\!](x)} P[\![\pi_2]\!](y)$

Figure 1: Standard semantics of location paths.

4 Semantics of XPath

In this section, we present a concise definition of the semantics of XPath 1 [10]. We assume the syntax of this language known, and cohere with its *unabbreviated* form [10]. We use a normal form syntax of XPath, which is obtained by the following rewrite rules, applied initially:

1. Location steps $\chi::t[e]$, where e is an expression that produces a number (see below), are replaced by the equivalent expression $\chi::t[e = \text{position}()]$.
2. All type conversions are made explicit (using the conversion functions string, number, and boolean, which we will define below).
3. Each variable is replaced by the (constant) value of the input variable binding.

The main syntactic construct 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 [10]. $\mathbf{C} = \text{dom} \times \{\langle k, n\rangle \mid 1 \le k \le n \le |\text{dom}|\}$ is the domain of contexts. Let $\mathit{ArithOp} \in \{+, -, *, \text{div}, \text{mod}\}$, $\mathit{RelOp} \in \{=, \neq, \le, <, \ge, >\}$, $\mathit{EqOp} \in \{=, \neq\}$, and $\mathit{GtOp} \in \{\le, <, \ge, >\}$. By slight abuse of notation, we identify these arithmetic and relational operations with their symbols in the remainder of this paper. 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.

[5]To be consistent, we also will not discuss the "local-name", "namespace-uri", and "name" core library functions [10].

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

Definition 4.1 (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 semantics $[\![e]\!] : \mathbf{C} \to \mathcal{T}$ of XPath expression e be defined as follows.

$$\begin{aligned} [\![\pi]\!](\langle x,k,n\rangle) &:= P[\![\pi]\!](x) \\ [\![\text{position}()]\!](\langle x,k,n\rangle) &:= k \\ [\![\text{last}()]\!](\langle x,k,n\rangle) &:= n \end{aligned}$$

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 $\mathcal{F}[\![Op]\!] : \mathcal{T}_1 \times \ldots \times \mathcal{T}_m \to \mathcal{T}$ is called the *effective semantics function* of Op. The function P is defined in Figure 1 and the effective semantics function $\mathcal{F}$ is defined in Table 2. □

To save space, we at times re-use function definitions in Table 2 to define others. However, our definitions are not circular and the indirections can be eliminated by a constant number of unfolding steps. Moreover, for lack of space, we define neither the number operations floor, ceiling, and round nor the string operations concat, starts-with, contains, substring-before, substring-after, substring (two versions), string-length, normalize-space, translate, and lang in Table 2, but it is very easy to obtain these definitions from the XPath 1 Recommendation [10].

The compatibility of our semantics definition (modulo the assumptions made in this paper to simplify the data model) with [10] can easily be verified by inspection of the latter document.

It is instructive to look at the definition of $P[\![\pi_1/\pi_2]\!]$ in Figure 1 in more detail. It is easy to see that, if this semantics definition is followed rigorously to obtain an analogous functional implementation, query evaluation using this implementation requires time exponential in the size of the queries. All systems evaluated in [3] went into this (or a very similar) trap.

5 Bottom-up Evaluation of XPath

In this section, we present a bottom-up semantics and algorithm for evaluating XPath queries in polynomial time. 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 5.1 (Semantics) We represent the four XPath expression types nset, num, str, and bool using relations as shown in Table 3. The bottom-up semantics of expressions is defined via a semantics function

$$\mathcal{E}_\uparrow : \text{Expression} \to \text{nset} \cup \text{num} \cup \text{str} \cup \text{bool},$$

given in Table 4 and as

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

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

Table 2: 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 3: Expression types and associated relations.

Expr. E : **Operator 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 $\{\langle x_0, k_0, n_0, \{x \in S \mid \langle x, \text{idx}_\chi(x, S), \lvert S\rvert, \text{true}\rangle \in \mathcal{E}_\uparrow[\![e]\!]\}\rangle$ $\mid \langle x_0, k_0, n_0, S\rangle \in \mathcal{E}_\uparrow[\![E]\!]\}$
location path $/\pi$: nset $\rightarrow$ nset $\mathbf{C} \times \{S \mid \exists k, n : \langle \text{root}, k, n, S\rangle \in \mathcal{E}_\uparrow[\![\pi]\!]\}$
location path π_1/π_2 : nset $\times$ nset $\rightarrow$ nset $\{\langle x, k, n, z\rangle \mid\ 1 \le k \le n \le \lvert\text{dom}\rvert,$ $\langle x, k_1, n_1, Y\rangle \in \mathcal{E}_\uparrow[\![\pi_1]\!],$ $\bigcup_{y \in Y} \langle y, k_2, n_2, z\rangle \in \mathcal{E}_\uparrow[\![\pi_2]\!]\}$
location path $\pi_1 \mid \pi_2$: nset $\times$ nset $\rightarrow$ nset $\mathcal{E}_\uparrow[\![\pi_1]\!] \cup \mathcal{E}_\uparrow[\![\pi_2]\!]$
position() : $\rightarrow$ num $\{\langle x, k, n, k\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$
last() : $\rightarrow$ num $\{\langle x, k, n, n\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$

Table 4: Expression relations for location paths, position(), and last().

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. $\langle x, k, n, v\rangle \in \mathcal{E}_\uparrow[\![e]\!]$.

Theorem 5.2 *Let e be an arbitrary XPath expression. Then, for context node x, position k, and size n, the value of e is v, where 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[6]. □

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

Query Evaluation. The idea of Algorithm 5.3 below is so closely based on our semantics definition that its correctness follows directly from the correctness result of Theorem 5.2.

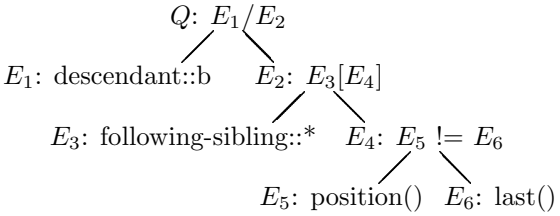
Algorithm 5.3 (Bottom-up algorithm for XPath)
Input: An XPath query Q;
Output: $\mathcal{E}_\uparrow[\![Q]\!]$.
Method:

let Tree(Q) be the parse tree of query Q;
$\mathbf{R} := \emptyset$;
for each atomic expression $l \in$ leaves(Tree(Q)) **do**
  compute table $\mathcal{E}_\uparrow[\![l]\!]$ and add it to $\mathbf{R}$;
while $\mathcal{E}_\uparrow[\![\text{root}(\text{Tree}(Q))]\!] \notin \mathbf{R}$ **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 $\mathcal{E}_\uparrow[\![\text{root}(\text{Tree}(Q))]\!]$. □

Example 5.4 Consider the document of Example 3.1. 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 5.3: 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[7] for E_1, E_2, E_3 and Q are shown in Figure 2. Moreover,

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

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

[7]The k and n columns have been omitted. Full tables are obtained by computing the cartesian product of each table with $\{\langle k, n\rangle \mid 1 \le k \le n \le \lvert\text{dom}\rvert\}$.

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

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

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

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

Figure 2: Context-value tables of Example 5.4.

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. □

Theorem 5.5 *XPath can be evaluated bottom-up in polynomial time (combined complexity).*

(For a proof sketch of this result see [3].)

6 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 5, we 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 3. 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 $\langle x, p, s, y\rangle \in \mathcal{E}_\uparrow[\![\pi]\!]$.

$\mathcal{S}_\downarrow[\![\chi{::}t[e_1]\cdots[e_m]]\!](X_1, \ldots, X_k) :=$
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 $1 \le i \le m$ (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 $t_j = \langle y_j,\ \text{idx}_\chi(y_j, S_j),\ |S_j|\rangle$
      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 $1 \le i \le k$ **do**
    $R_i := \{y \mid \langle x, y\rangle \in S, x \in X_i\}$;
  return $\langle R_1, \ldots, R_k\rangle$;
end;

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

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

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

Figure 3: Top-down evaluation of location paths.

Definition 6.1 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

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

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

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

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 6.2 Given the query Q, data, and context $\langle a, 1, 1\rangle$ of Example 5.4, we evaluate Q as $\mathcal{E}_\downarrow[\![Q]\!](\langle a, 1, 1\rangle) = \mathcal{S}_\downarrow[\![E_2]\!](\mathcal{S}_\downarrow[\![\text{descendant::b}]\!](\{a\}))$ where $\mathcal{S}_\downarrow[\![\text{descendant::b}]\!](\{a\}) = \langle\{b_1, b_2, b_3, b_4\}\rangle$.

To compute $\mathcal{S}_\downarrow[\![E_2]\!](\langle\{b_1, b_2, b_3, b_4\}\rangle)$, we proceed as described in the algorithm for location steps in Figure 3. 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 list of 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 $\mathcal{E}_\downarrow$.

Theorem 6.3 *(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 5.5, we get

Theorem 6.4 *The immediate functional implementation of $\mathcal{E}_\downarrow$ evaluates XPath queries in polynomial time (combined complexity).*

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 6.4.

7 Conclusions

The algorithms presented in this paper empower XPath engines to deal efficiently with very sophisticated queries while scaling up to large documents.

In [3, 4], in addition to an experimental verification of our claim that current XPath engines evaluate queries in exponential time only, we have also presented fragments of XPath which contain most of the XPath queries likely to be used in practice and can be evaluated particularly efficiently, using methods that are beyond the scope of this paper.

We have made a main-memory implementation of the top-down algorithm of Section 6. Resources such as this implementation and publications will be made accessible at `http://www.xmltaskforce.com`.

Acknowledgments

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

References

[1] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath. In *Proc. KRDB 2001*, CEUR Workshop Proceedings 45, 2001.

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

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

[4] G. Gottlob, C. Koch, and R. Pichler. "XPath Query Evaluation: Improving Time and Space Efficiency". In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE'03)*, Bangalore, India, Mar. 2003. to appear.

[5] G. Miklau and D. Suciu. "Containment and Equivalence for an XPath Fragment". In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'02)*, pages 65–76, Madison, Wisconsin, 2002.

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

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

[8] P. T. Wood. "On the Equivalence of XML Patterns". In *Proc. 1st International Conference on Computational Logic (CL 2000)*, LNCS 1861, pages 1152–1166, London, UK, July 2000. Springer-Verlag.

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

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

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

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