The Complexity of XPath Query Evaluation and XML Typing

GEORG GOTTLOB, CHRISTOPH KOCH, and REINHARD PICHLER
Technische Universität Wien, Austria
LUC SEGOUFIN
INRIA, France

We study the complexity of two central XML processing problems. The first is XPath 1.0 query processing, which has been shown to be in PTIME in previous work. We prove that both the data complexity and the query complexity of XPath 1.0 fall into lower (highly parallelizable) complexity classes, while the combined complexity is PTIME-hard. Subsequently, we study the sources of this hardness and identify a large and practically important fragment of XPath 1.0 for which the combined complexity is LOGCFL-complete and, therefore, in the highly parallelizable complexity class NC^2. The second problem is the complexity of validating XML documents against various typing schemes like Document Type Definitions (DTDs), XML Schema Definitions (XSDs), and tree automata, both with respect to data and to combined complexity. For data complexity, we prove that validation is in LOGSPACE and depends crucially on how XML data is represented. For the combined complexity, we show that the complexity ranges from LOGSPACE to LOGCFL, depending on the typing scheme.

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Automata; F.4.1 [**Mathematical Logic and Formal Languages**]: Computational Logic; F.4.3 [**Mathematical Logic and Formal Languages**]: Classes defined by grammars or automata; H.2.3 [**Database Management**]: Query languages; I.7.2 [**Document Preparation**]: Markup languages

General Terms: Theory, Languages, Algorithms

Additional Key Words and Phrases: Complexity, LOGCFL, XML, XPath, DTD

1. INTRODUCTION

The *Extensible Markup Language* (XML) is emerging as the global standard for data exchange. Two of the most central problems related to processing XML data are the problem of *evaluating an XPath query* and the *document validation problem* (i.e., the problem of checking whether an XML document matches a predefined type). Both occur frequently in practice; the former because XPath forms an essential part of query and data transformation languages such as XQuery and XSLT. As an XQuery or an XSLT stylesheet usually contains several XPath queries, XPath queries are also probably the most common form of queries on XML. The latter

This research was supported by the Austrian Science Fund (FWF) under projects No. Z29-N04 and J2169 and by the EU Research Training Network GAMES.
This work is based on the extended abstracts [Gottlob et al. 2003a; Segoufin 2003] which both appeared in *Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2003)*, San Diego, California.
Contact details: Georg Gottlob and Christoph Koch, Database and Artificial Intelligence Group (E184/2), Technische Universität Wien, A-1040 Vienna, Austria. Email: {gottlob, koch}@dbai.tuwien.ac.at. Reinhard Pichler, Institut für Computersprachen (E185/2), Technische Universität Wien, A-1040 Vienna, Austria; Email: reini@logic.tuwien.ac.at. Luc Segoufin, INRIA, Parc Club Orsay Université, ZAC des vignes, 4 rue Jacques Monod, 91893 Orsay Cedex, France; WWW: http://www-rocq.inria.fr/∼segoufin

problem, validation, plays an important role because most web services check that their input and their output conform to a pre-specified type (SOAP requirement).

While it is folklore that both XQuery and XSLT are Turing-complete[1], the basic XPath processing and XML validation problems can be made subject to a complexity-theoretical study. Indeed, both problems are polynomial, but so far no detailed study of their precise complexity has been undertaken.

Many applications require the processing of large amounts of XML data using limited (CPU and memory) resources. Efficient practical algorithms for document validation and XPath processing are therefore crucial. In order to evaluate the efficiency of such algorithms it is often useful to know the best possible theoretical complexity bounds. The goal of this article is to provide such bounds.

The XPath Query Processing Problem

XPath 1.0 is the node-selecting query language central to most core XML-related technologies that are under the auspices of the W3C, including XQuery, XSLT, and XML Schema. Evaluating XPath queries efficiently is essential to the effectiveness and real-world impact of these technologies. The most natural question related to XPath query processing, its complexity, however, has received surprisingly little attention. The first polynomial-time algorithms for XPath processing (w.r.t. both the size of the data and the query, i.e., combined complexity, cf. [Vardi 1982]) were proposed only recently [Gottlob et al. 2002; 2003b]. The polynomial-time result of [Gottlob et al. 2002] was shown using a form of dynamic programming. Based on this, algorithms were presented that run in time $O(|t|^4 \cdot |Q|^2)$ and space $O(|t|^2 \cdot |Q|^2)$, where $|t|$ denotes the size of the XML document tree and $|Q|$ is the size of the query [Gottlob et al. 2003b]. In [Gottlob et al. 2002], also the logical core fragment of XPath was introduced, which was called *Core XPath* and which includes the logical and navigational (path processing) features of XPath but excludes the manipulation of data values (and thus arithmetics and string manipulations). Core XPath queries can be evaluated in time $O(|t| \cdot |Q|)$, i.e. time linear in the size of the query and of the data tree.

Now that the combined complexity of XPath is known to be polynomial, a natural question emerges, namely whether XPath is also PTIME-hard, or alternatively, whether it is in the complexity class NC and thus effectively parallelizable. In case the problem is PTIME-hard, it is interesting to understand the sources of this hardness, and to find large, effectively parallelizable fragments. Apart from theoretical interest, a precise characterization of these problems in terms of parallel complexity classes provides us with a detailed understanding of precisely what computational resources are required to solve them. For example, it is strongly conjectured that all algorithms for solving PTIME-hard problems actually require a polynomial amount of working memory. However, performing XPath query evaluation and document validation with limited memory resources is important in practice, e.g. in the context of data stream processing (which obtains its relevance from the fact that the primary purpose of XML is as a data exchange format).

[1] XQuery is a descendant of the Quilt language, which was deliberately chosen to be Turing-complete [Chamberlin et al. 2000], while there is even a Turing machine simulator implemented in XSLT available on the Web [Lyons 2001].

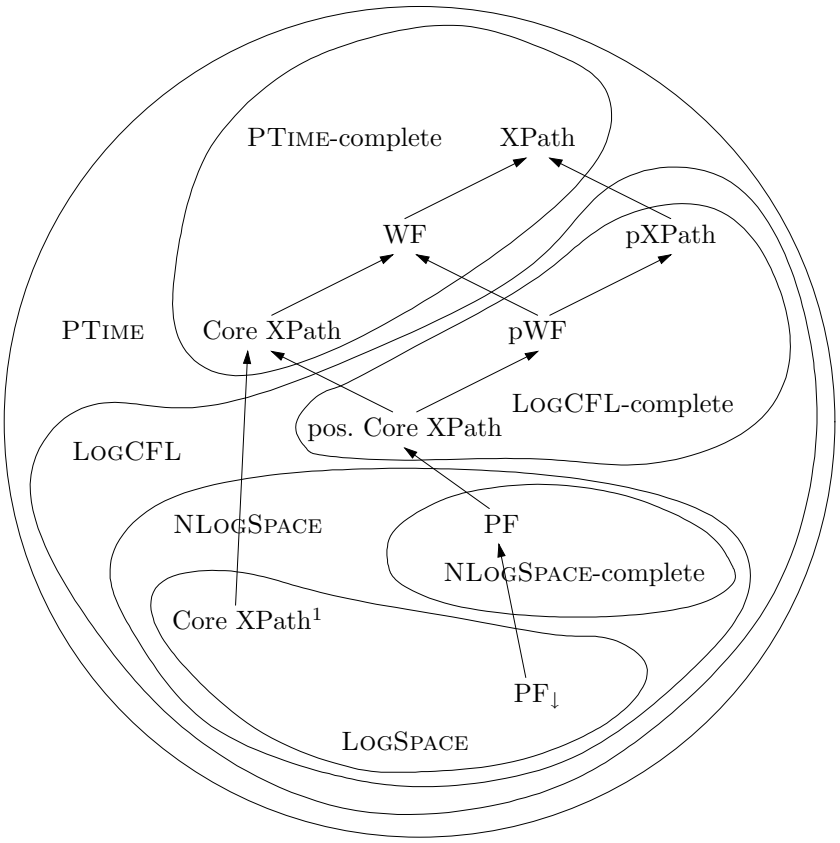

Fig. 1. Combined complexity of XPath. An arrow $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that language $\mathcal{L}_1$ is a fragment of language $\mathcal{L}_2$.

This article thus studies the precise complexity of XPath 1.0 query processing. The contributions are as follows.

—We establish the combined complexity of XPath to be PTIME-hard. This remains true even for the Core XPath fragment.

—We show that positive Core XPath, i.e. Core XPath without negation, is LOGCFL-complete, and thus highly parallelizable. Moreover, if the language is further restricted to the path expressions fragment (PF) without conditions, the complexity of evaluating queries is complete for NLOGSPACE.

—We study a fragment of PF (called $\text{PF}_{\downarrow}$) and a fragment of Core XPath with negation, Core XPath^1, which are obtained by restricting the navigational capabilities (in terms of supported *axes*) rather than the language features. We show that all three fragments are highly parallelizable by proving LOGSPACE upper bounds.

—We extend Core XPath by the arithmetics features of XPath, to the so-called *Wadler Fragment* (WF), and show that a large fragment of it, which we call pWF ("positive"/"parallel" WF), is still in LOGCFL. Thus, the evaluation of

pWF queries can be massively parallelized in theory because LogCFL is in the complexity class NC^2 of problems solvable in time $O(\log^2 n)$ with polynomially much hardware. The main features excluded from WF to obtain pWF are negation and sequences of condition predicates.

—This leads us to an even larger fragment of XPath, called pXPath, which we believe contains most practical XPath queries and for which query evaluation can be massively parallelized (the combined complexity is still LogCFL-complete).

—Finally, we complement our results on the combined complexity of XPath with a study of data complexity and query complexity. Both problems fall into low (highly parallelizable) complexity classes (LogSpace) even in the presence of negation in queries.
Since the data complexity of XPath is very low, details of the representation become important, as for the validation problem. We show that the data complexity of evaluating Core XPath is in TC^0 on string representations of the data trees, while it is LogSpace-complete on pointer representations.

The inclusion relationships[2] between fragments discussed and their (combined) complexities are shown in Figure 1.

The XML Validation Problem

In the second part of the paper we study the problem of checking whether an XML document conforms to some typing scheme. For this we model XML documents by labeled unranked ordered trees. This is classical in the literature (see for instance the surveys [Neven 2002; Suciu 2001; Abiteboul 2001]) because it permits to formalize XML typing specifications and XML query languages using unranked tree automata and their well-established theory (cf. [Comon et al. 1999; Brüggemann-Klein et al. 2001]).

We consider various kinds of typing systems under which we study the type checking problem for labeled trees. The first two, called DTDs and *extended* DTDs (which extend DTDs by a specialization mechanism [Papakonstantinou and Vianu 2000]), are motivated by XML standards as they roughly correspond to XML DTDs and XSDs. Both typing systems are subsumed by tree automata. We therefore generalize our study to the main variants of tree automata, namely top-down, bottom-up, deterministic, nondeterministic, and tree walking automata.

We consider two variants of the type checking problem. In the first, the type is fixed and the input consists only of the data tree. In the second, the type is also part of the input. The first case is called the *data complexity* of the validation problem and gives a good approximation of the behavior of the problem when the size of the type is assumed to be unimportant compared to the size of the data. The second is the *combined complexity* and gives a more accurate measure of the difficulty of the problem.

Our study shows that the data complexity is independent of the typing system while the typing system used is crucial for the combined complexity bounds.

The data complexity of the validation problem depends crucially on the coding of the input tree because the complexity classes involved are low (below LogSpace).

[2]In the drawing, we assume that LogSpace $\subset$ NLogSpace $\subset$ LogCFL $\subset$ PTime.

While the membership of a tree in a language defined by a tree automaton over a *ranked alphabet* was already studied in [Lohrey 2001], the extension to unranked trees is not immediate for coding issues. We consider two codings that reflect the two most widely used models for representing XML. The first is a pointer structure as in the DOM model [World Wide Web Consortium 2004]. The second is a string corresponding to the succession of opening and closing tags encountered during the depth-first, left-to-right traversal of the tree, as used in the SAX data model [SAX Project Collaboration 2004].

For the DOM-like coding we show that, for all typing systems considered here, the data complexity of the validation problem is in LogSpace and that this upper bound is tight. For the SAX-like coding we show that, for all typing systems considered here, the data complexity problem is even in NC^1. This therefore extends the results of [Lohrey 2001] to unranked trees with the SAX-like coding. Again our upper bound is tight.

The combined complexity of the validation problem depends on the syntax of the typing system and we obtain complexities ranging from LogSpace-complete to LogCFL-complete. Again the results for tree automata over ranked alphabets can be found in [Lohrey 2001]. For nondeterministic tree automata over ranked alphabets, [Lohrey 2001] showed that the validation problem is LogCFL-complete using a nontrivial reduction from membership in a context-free language. We give a new, short and elementary, proof of this result using a circuit characterization of LogCFL, and then extend it to unranked tree automata. Note again that the extension is not straightforward as the coding of an automaton over unranked trees is more complex than for trees over a ranked alphabet. Indeed, each transition now consists of a regular expression instead of a finite set of words. As a matter of fact, for deterministic unranked tree automata, we obtain complexities that are slightly greater than for the ranked case.

Some of the algorithms given in this article reduce the unranked tree case to the ranked one. This is done by encoding an unranked tree into a ranked one. It is a well known fact that there exists a mapping from unranked trees to ranked trees which preserves the recognition by regular tree automata [Suciu 2001; Papakonstantinou and Vianu 2003; Neven 2002]. As a side result of independent interest we show that for this transformation, the coding of the output (ranked) tree can be computed in TC^0 (resp. LogSpace) from the coding of the input (unranked) tree if the coding is SAX (resp. DOM), and that the output automaton can be computed in LogSpace from the input automaton.

The article is organized as follows. Section 2 introduces the necessary background from complexity theory and introduces (XML) trees and XPath. In Section 3 we provide a thorough complexity analysis of XPath evaluation. We first introduce XPath in Section 2.3. In Sections 3.1 through 3.5, we present several results on the combined complexity of XPath, while Section 3.6 studies the data and query complexity of XPath. Section 4 addresses the XML validation problem. We start by introducing the necessary foundations on logic, encodings, and typing formalisms. The membership problem for unranked tree automata is studied in Section 4.4. Section 4.5 is devoted to the combined complexity problem for validation. Finally, in Section 5, we give a brief discussion of our results.

2. PRELIMINARIES

2.1 Complexity-theoretic Background

We briefly discuss the complexity classes and some of their characterizations used throughout the article. For more thorough surveys of the related theory see [Johnson 1990; Papadimitriou 1994; Greenlaw et al. 1995].

By PTime, LogSpace, and NLogSpace we denote the well-known complexity classes of problems solvable on Turing machines in deterministic polynomial time, deterministic logarithmic space, and nondeterministic logarithmic space, respectively. A language (analogously, a problem) is in ULogSpace (unambiguous LogSpace) iff there is a LogSpace-bounded nondeterministic Turing Machine such that there is at most one accepting computation for each input.

It is a widely-held conjecture that problems complete for PTime are inherently sequential and cannot profit from parallel computation (cf. e.g. [Greenlaw et al. 1995]). Instead, a problem is called *highly parallelizable* if it can be solved within the complexity class NC of all problems solvable in polylogarithmic time on a polynomial number of processors working in parallel [Greenlaw et al. 1995].

A simple model of parallel computation is that of Boolean circuits. By a monotone circuit, we denote a circuit in which only the input gates may possibly be negated. All other gates are either $\wedge$-gates or $\vee$-gates (but no $\neg$-gates). A family of circuits is a sequence $\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, \ldots$, where the n-th circuit $\mathcal{G}_n$ has n inputs. Such a family is called LogSpace-*uniform* if there exists a LogSpace-bounded deterministic Turing machine which, on the input of n bits 1 (the string 1^n), outputs the circuit $\mathcal{G}_n$. DLogTime-uniformity is defined by means of a Turing machine that checks in deterministic logarithmic time whether a gate is connected to another one (cf. [Barrington et al. 1990] for details). Moreover, a family of circuits has *bounded fan-in* if all of the gates in these circuits have fan-in bounded by some constant. On the other hand, a family of monotone circuits is called *semi-unbounded* if all $\wedge$-gates are of bounded fan-in (without loss of generality, we may restrict the fan-in to two) but the $\vee$-gates may have unbounded fan-in.

By NC^i, we denote the class of languages recognizable using LogSpace-uniform Boolean circuit families of polynomial size and depth $O(\log^i n)$ (in terms of the size n of the input). By AC^0, we denote the class of languages that can be recognized by a DLogTime-uniform family of Boolean circuits of constant depth and polynomial size in the input consisting of $\wedge$ and $\vee$-gates of unbounded fan-in (and $\neg$-gates). TC^0 additionally has MAJORITY-gates of unbounded fan-in. NC^1 can be alternatively characterized using circuits as the class of languages recognizable by DLogTime-uniform Boolean circuits (there are no MAJORITY gates) of logarithmic depth and gates of bounded fan-in. (Thus, the uniformity measure used in this article for all three classes AC^0, TC^0 and NC^1 is DLogTime-uniformity as in [Barrington et al. 1990].) SAC^1 is the class of languages recognizable by LogSpace-uniform families of semi-unbounded circuits of depth $O(\log n)$ (SAC^1 circuits).

The following problem is known to be LogSpace-complete under AC^0-reductions.

[ORD]:	INPUT:	a directed graph $G = (V, E)$ which is a line and two nodes v_i, v_j
	OUTPUT:	true iff $v_i < v_j$ in the order induced by the graph.

PROPOSITION 2.1 [ETESSAMI 1997]. *ORD is* LOGSPACE*-complete under* AC^0 *reductions.*

LOGCFL is usually defined as the complexity class consisting of all problems LOGSPACE-reducible to a context-free language. LOGDCFL (resp. LOGUCFL) is the analogous class of all problems LOGSPACE-reducible to a *deterministic* (resp. *unambiguous*) context-free language. There are two important alternative characterizations of LOGCFL that we are going to use. They are recalled in Proposition 2.2 and 2.3, respectively.

PROPOSITION 2.2 [VENKATESWARAN 1991]. LOGCFL = SAC^1. SAC^1 *Circuit Value is* LOGCFL*-complete.*

An auxiliary pushdown automaton is a Turing machine with a distinguished input tape, a worktape, and a stack (of which strictly only the topmost element can be accessed at any time). We use the acronyms AuxPDA and NAuxPDA for its deterministic and nondeterministic versions, respectively.

PROPOSITION 2.3 [SUDBOROUGH 1977]. LOGCFL *(*LOGDCFL*) is the class of all decision problems solvable by a NAuxPDA (an AuxPDA) with a logarithmic space-bounded worktape in polynomial time.*

PROPOSITION 2.4 [BORODIN ET AL. 1989]. LOGCFL *is closed under complement.*

We have $AC^0 \subsetneq TC^0 \subseteq NC^1 \subseteq$ LOGSPACE $\subseteq$ NLOGSPACE $\subseteq$ LOGCFL $\subseteq NC^2 \subseteq NC \subseteq$ PTIME and LOGSPACE $\subseteq$ LOGDCFL $\subseteq$ LOGCFL. All inclusions $\subseteq$ are suspected to be strict. PTIME, LOGCFL, LOGDCFL, and NLOGSPACE are closed under LOGSPACE-reductions. The forms of reductions used for complexity classes are explicitly stated in the results. Roughly, we use LOGSPACE-reductions for complexity classes above LOGSPACE and AC^0 and DLOGTIME-reductions for complexity classes below LOGSPACE. In all classes completeness is relative to many-one reductions. We do not consider lower bounds within TC^0 because no complete problem for TC^0 under many-one reductions is known [Allender 2001].

2.2 Trees

We view XML documents as unranked ordered trees in which nodes are labeled using an alphabet Σ. Given a tree t, its number of nodes is denoted by $|t|$. The *rank* of a node n is the number of children of n. The *label* of a node n is denoted by label(n). The *rank* of a tree t is the maximum rank of any node n of t. A class of trees is *unranked* if there is no bound on their rank. Two representations of such trees will be considered:

Trees as strings. The second encoding is close to the model used in the SAX interface [SAX Project Collaboration 2004]. It is the actual text representation of an XML document, with no data values, which is a succession of opening and closing tags. It corresponds to a depth-first, left-to-right traversal of the tree. We reuse the notations of [Segoufin and Vianu 2002]. For each $a \in \Sigma$, let a itself represent the opening tag and $\bar{a}$ represent the closing tag for a. Let $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$.

We associate to each labeled tree t a string representation denoted $[t]$ and defined inductively as follows. If t consists of just a single (root) node labeled a, then $[t] =$

$a\bar{a}$. If t consists of a root labeled a and subtrees $t_1 \dots t_k$ then $[t] = a[t_1] \dots [t_k]\bar{a}$. If T is a set of trees, we denote by $\mathcal{L}(T)$ the language consisting of the string representations of the trees in T.

EXAMPLE 2.5. With this notation, the string associated to the labeled tree of Figure 2 is $rabc\bar{c}\bar{b}c\bar{c}c\bar{c}\bar{a}ab\bar{b}c\bar{c}c\bar{c}\bar{a}a\bar{a}\bar{r}$. □

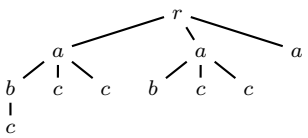

Fig. 2. A labeled tree.

If t is a tree, the size of $[t]$ is twice the number of nodes in t. Thus we also use the notation $|t|$ to denote the size of the string encoding of t.

In the string encoding, a forest is thus a "well-balanced" string over $\Sigma \cup \bar{\Sigma}$ (i.e., the parenthesis sequence in this string must be correct, where each pair $(a, \bar{a})$ with $a \in \Sigma$ and $\bar{a} \in \bar{\Sigma}$ is considered as left and right parenthesis, respectively). This can be checked efficiently as the following shows.

PROPOSITION 2.6 [BARRINGTON AND CORBETT 1989]. *Given a word w over the alphabet $\Sigma \cup \bar{\Sigma}$, it is decidable in* TC^0 *whether w is well-balanced.*

Trees as pointer structures. We represent trees as pointer structures roughly using the DOM encoding [World Wide Web Consortium 2004]: Each node is an object containing its label and pointers to (i) its parent, (ii) its first child, (iii) its previous sibling (i.e., left neighbour), and (iv) its next sibling (i.e., right neighbor).

Unless stated otherwise, we will represent trees as pointer structures.

2.3 A Brief Introduction to XPath

XPath 1.0 is a language with a large number of features and therefore somewhat unwieldy for theoretical treatment. In this article, we restrict ourselves to introducing only some of these features, and to giving an informal explanation of their semantics. For a detailed definition of the full XPath language, we refer to [World Wide Web Consortium 1999], and for a concise yet complete formal definition of the XPath semantics see [Gottlob et al. 2002].

In this section, we define two basic fragments of XPath. Core XPath, first defined in [Gottlob et al. 2002], supports the most commonly used features of XPath, path navigation and conditions with logical connectives, but excludes arithmetics, string manipulations, and some of the more esoteric aspects of the language. The second fragment, which was first discussed in [Wadler 2000], contains XPath's logical and arithmetic features, but excludes string manipulations. We refer to it as the *Wadler Fragment*, short WF.

We start by discussing Core XPath. We sketch the fragment in terms of its syntax and then informally discuss the semantics.

DEFINITION 2.7. The syntax of Core XPath is defined by the grammar

locpath ::= '/' locpath | locpath '/' locpath | locpath '|' locpath | locstep.
locstep ::= axis '::' ntst '[' bexpr ']' ... '[' bexpr ']'.
bexpr ::= bexpr '`and`' bexpr | bexpr '`or`' bexpr | '`not`(' bexpr ')' | locpath.
axis ::= '`self`' | '`child`' | '`parent`' |
'`descendant`' | '`descendant-or-self`' |
'`ancestor`' | '`ancestor-or-self`'
'`following`' | '`following-sibling`'
'`preceding`' | '`preceding-sibling`'.

where "locpath" is the start production, "axis" denotes axis relations (see below), and "ntst" denotes tags labeling document nodes or the star '*' that matches all tags ("node tests"). □

The main syntactical feature of Core XPath are *location paths*. Expressions enclosed in square brackets are called *conditions* or *predicates*.

The main application of XPath is the navigation in XML document trees. This is done using the *axis relations*, natural binary relations such as "child" and "descendant" between nodes, which we do not define here (but see [World Wide Web Consortium 1999; Gottlob et al. 2002]; they also have the intuitive meanings conveyed by their names). The probably most common use of XPath is to compose axis applications with selections of document nodes by their tags ("node tests"). For instance, the XPath expression /descendant::a/child::b selects all those nodes labeled "b" that are children of nodes labeled "a" that are in turn descendants of the root node (denoted by the initial slash).

Conditions enclosed in square brackets allow to impose additional constraints on node selections. For example,

/descendant::a/child::b[descendant::c and not(following-sibling::d)]

selects exactly those nodes v from the nodes in the result of /descendant::a/child::b that have *at least one*[3] descendant labeled "c" and do not have a right sibling in the tree that is labeled "d" (i.e., there is no child v' of the parent of v which follows v in the flow of the document and is labeled "d").

DEFINITION 2.8. The syntax of the "Wadler Fragment" WF is defined by the Core XPath grammar with the following extensions. "bexpr" is now

bexpr ::= bexpr '`and`' bexpr | bexpr '`or`' bexpr | '`not`(' bexpr ')' | locpath |
nexpr relop nexpr.

Moreover,

expr ::= locpath | bexpr | nexpr.
nexpr ::= '`position`()' | '`last`()' | number | nexpr arithop nexpr.
arithop ::= '+' | '-' | '*' | '`div`' | '`mod`'.
relop ::= '=' | '!=' | '<' | '<=' | '>' | '>='.

[3]Location paths occurring in conditions – i.e., within square brackets – have an "exists"-semantics, meaning that at least one node must match the location path starting from the current node.

"expr" (rather than "locpath") is now the start production and "number" denotes constant real-valued numbers. □

XPath is mainly understood as a language for selecting a subset of the nodes of an XML document tree. Hence, we shall refer to XPath *expressions* also synonymously as XPath *queries*. Note, however, that XPath query results can also be of different types, namely – for the WF – numbers and Booleans (as well as character strings for full XPath). XPath expressions are evaluated relative to a context, which by definition is a triple of a *context-node* and two integers, the so-called *context-position* and the *context-size*. For details, we refer to [World Wide Web Consortium 1999; Gottlob et al. 2002], but consider the example query

$$\text{child::a[position() + 1 = last()]}.$$

Relative to a context-triple (v, i, j), i and j are ignored when the location step child::a selects those children of v that are labeled "a". Let $\{w_1, \ldots, w_m\}$ be this set of nodes, where the indices correspond to the relative order of the nodes in the document, simply speaking[4]. The application of an axis causes a change of context to which the condition [position() + 1 = last()] is applied. The condition is tried on each of the triples $(w_1, 1, m), \ldots, (w_m, m, m)$. It will select all those nodes w_k for which $k + 1 = m$, i.e. it will select the singleton $\{w_{m-1}\}$.

In this work, we are going to study the complexity of XPath evaluation. Formally, we are thus considering the following *decision problem*: Given an XPath query Q, a data tree t, a context $\vec{c}$, and a result value r, one has to decide whether evaluating the query Q over the data tree t for the context $\vec{c}$ yields the result value r.

Sometimes, the result of an XPath query is independent of a context (e.g., absolute location paths). In this case, we shall use the short-hand notation $Q(t)$ to denote the result of evaluating Q over the data tree t.

Let $\mathcal{C}$ denote some complexity class and let X be some fragment of XPath. If the above decision problem of evaluating XPath queries from the fragment X is in the complexity class $\mathcal{C}$ (resp. $\mathcal{C}$-hard or $\mathcal{C}$-complete), then we shall say, as a short-hand, that "*the fragment X is in $\mathcal{C}$*" (resp. "*$\mathcal{C}$-hard*" or "*$\mathcal{C}$-complete*").

PROPOSITION 2.9 [GOTTLOB ET AL. 2002]. *The XPath query evaluation problem is in* PTIME *with respect to combined complexity.*

PROPOSITION 2.10 [GOTTLOB ET AL. 2002]. *Core XPath queries can be evaluated in time $O(|Q| \cdot |t|)$, where $|Q|$ denotes the size of the query and $|t|$ denotes the size of the data tree.*

3. COMPLEXITY OF XPATH QUERY EVALUATION

3.1 Combined Complexity of Core XPath

Core XPath expressions always evaluate to node sets. It is therefore convenient to consider the following modified complexity problem in our complexity analysis of Core XPath and some sub-fragments thereof, namely: Given a Core XPath query Q, a data tree t, a context node c, and a result node v, one has to decide whether the node v lies in the node set that we get by evaluating the query Q on the data

[4]To be precise, for some axes this order is reversed, see [World Wide Web Consortium 1999].

tree t for the context node c. It is easy to check that the complexity results obtained in the Sections 3.1 through 3.3 carry over to the definition of the XPath evaluation problem from Section 2.

Actually, it is easy to verify that we would get essentially the same complexity results (in particular, the same upper bounds) if we considered the functional problem of XPath evaluation rather than the decision problem, i.e.: Given a triple $(Q, t, \vec{c})$, one has to compute the result value r of evaluating the query Q over the data tree t for the context $\vec{c}$. For instance, for some fragment of (Core) XPath, let the decision problem of XPath evaluation be in the complexity class $\mathcal{C}$. Then we could solve the functional problem for this (Core) XPath fragment by checking in a loop over all nodes v of t whether v is in the resulting node set. Hence, this functional problem is in $LogSpace^{\mathcal{C}}$, i.e., in LogSpace with oracle in $\mathcal{C}$.

In this section, we show that XPath and even Core XPath are PTime-hard with respect to combined complexity, i.e. the problem of deciding, given a query Q, a data tree t, and a node v in t, whether $v \in Q(t)$.

Remark 3.1. All results presented in this article hold for data trees in which each tree node has a single label. However, to shorten and simplify some proofs about XPath fragments that support the child axis and conditions, it will sometimes be convenient to assume that data tree nodes may have multiple labels.[5] We can simulate a tree t_0 with multiple node labels by a tree t obtained from t_0 by relabeling each node with special label M (that is not used by t_0) and adding for each node v of t_0 and each label l carried by v in t_0 a new child of v labeled l to t. Now we can check whether an M-labeled node has label l using the condition expression child::l. (We will use the *shortcut* $T(l)$ for child::l.) A query such as descendant::*[T(a) and T(b)] on tree t_0 evaluates as descendant::M[child::a and child::b].

Let C be a circuit with gates $G_1, \ldots, G_n$. In the following, C is called layered iff there is an integer k (the number of layers) and a surjective function $f : \{G_1, \ldots, G_n\} \to \{1, \ldots, k\}$ which maps each gate of C to its layer such that if the output of G_i is connected to the input of G_j, then $f(G_i) + 1 = f(G_j)$.

Theorem 3.2. *Core XPath is* PTime*-complete w.r.t. combined complexity.*

Proof. The combined complexity even of full XPath was shown to be in PTime in [Gottlob et al. 2002], thus all we need to show is PTime-hardness. This is done by reduction from the *monotone Boolean circuit value* problem, which is PTime-complete. Note that the classical reduction from PTime-bounded Turing machines to (monotone) Boolean circuits proving this (see e.g. the proof of Theorem 8.1 in [Papadimitriou 1994]) only produces *layered* circuits.

Given an instance of this problem (a monotone Boolean circuit), let M denote the number of input gates and let N denote the number of all other gates in the circuit. Let the gates be named $G_1 \ldots G_{M+N}$. Without loss of generality[6], we may

[5]Of course, each node of an XML document tree can have only one tag, but multiple labels can be simulated using attributes. In our lower-bound proofs we will only use a lightweight unranked labeled tree model.

[6]The gates can be "sorted" to adhere to such an ordering in logarithmic space. This is trivial if the circuit is layered, which we may assume by the observation made above.

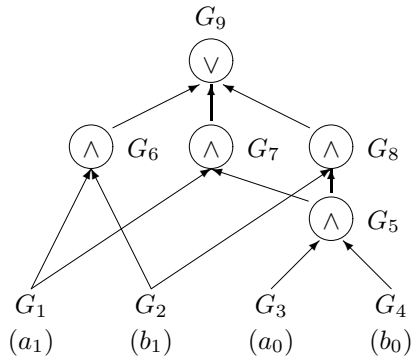

Fig. 3. A 2-bit full adder carry-bit circuit.

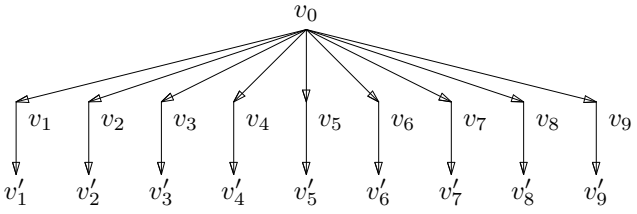

Fig. 4. Document tree corresponding to the carry-bit circuit.

assume that the gates $G_1 \dots G_{M+N}$ are numbered in some order such that no gate G_i depends on the output of another gate G_j with $j > i$. In particular, the input gates are named $G_1 \dots G_M$ and the output gate is G_{M+N}.

An example of a circuit with appropriately numbered gates is shown in Figure 3. This circuit computes the carry-bit of a two-bit full-adder, i.e. it tells whether adding the two-bit numbers a_1a_0 and b_1b_0 leads to an overflow. The carry-bit c_1 is computed as $(a_1 \wedge b_1) \vee (a_1 \wedge c_0) \vee (b_1 \wedge c_0)$ where $c_0 = a_0 \wedge b_0$ is the carry-bit of the lower digit (a_0 and b_0).

For a given instance of the monotone Boolean circuit value problem, we compute a pair of a document tree and a Core XPath query as follows.

The **document tree** representing the circuit consists of a root node v_0 with $M + N$ children $v_1 \dots v_{M+N}$, of which each v_i again has exactly one child v'_i (thus, the tree has depth two).

Node labels are taken from the alphabet $\{0, 1, G, R, I_1, \dots, I_N, O_1, \dots, O_N\}$ and each tree node is assigned a set of such labels. This is done as follows. The root node v_0 has no labels. The nodes $v_1 \dots v_{M+N}$ are assigned the label G each. (In a way described later, node v_i *represents* the value of gate G_i). Node v_{M+N} is also assigned label R (for "result"). Each node out of $v_1 \dots v_M$ is assigned the truth value at the input gate of the same index (i.e., out of $G_1 \dots G_M$), respectively. This is either the label 0 or 1. Each node v_{M+k} with $1 \leq k \leq N$ gets the additional label O_k. Moreover, if the output of gate G_i is an input of gate G_{M+k} (thus, by our gate ordering requirement, $i < M + k$), we add I_k to the labels of v_i. The

nodes $v'_1 \dots v'_M$ are labeled $\{I_1, \dots, I_N, O_1, \dots, O_N\}$ each and the nodes v'_{M+i}, for $1 \le i \le N$, are labeled $\{I_k, O_k \mid i < k \le N\}$.

For our carry-bit example of Figure 3 with $M = 4$ and $N = 5$, the tree is as shown in Figure 4. The node labels are as follows:

v_0: $\emptyset$	v_1: $\{G, v(a_1), I_2, I_3\}$	v_2: $\{G, v(b_1), I_2, I_4\}$
v_3: $\{G, v(a_0), I_1\}$	v_4: $\{G, v(b_0), I_1\}$	v_5: $\{G, O_1, I_3, I_4\}$
v_6: $\{G, O_2, I_5\}$	v_7: $\{G, O_3, I_5\}$	v_8: $\{G, O_4, I_5\}$
v_9: $\{G, R, O_5\}$	v'_1, v'_2, v'_3, v'_4: $\{I_1, \dots, I_5, O_1, \dots, O_5\}$	
v'_5: $\{I_2, \dots, I_5, O_2, \dots, O_5\}$	v'_6: $\{I_3, I_4, I_5, O_3, O_4, O_5\}$	v'_7: $\{I_4, I_5, O_4, O_5\}$
v'_8: $\{I_5, O_5\}$	v'_9: $\emptyset$	

where $v(a_1), v(b_1), v(a_0), v(b_0) \in \{0, 1\}$ are the truth values a_1, b_1, a_0, and b_0, respectively, at the input gates.

The **query** evaluating the circuit is

$$/\text{descendant-or-self::*}[T(R) \text{ and } \varphi_N]$$

with the condition expressions

$$\begin{aligned}
\varphi_k &:= \text{descendant-or-self::*}[T(O_k) \text{ and parent::*}[\psi_k]] \\
\psi_k &:= \begin{cases} \text{child::*}[T(I_k) \text{ and } \pi_k] & \dots\ G_{M+k} \text{ is an } \vee\text{-gate} \\ \text{not(child::*}[T(I_k) \text{ and not}(\pi_k)]) & \dots\ G_{M+k} \text{ is an } \wedge\text{-gate} \end{cases} \\
\pi_k &:= \text{ancestor-or-self::*}[T(G) \text{ and } \varphi_{k-1}].
\end{aligned}$$

for $1 \le k \le N$ and $\varphi_0 := T(1)$. It uses the intuition of processing one gate out of $G_{M+1} \dots G_{M+N}$ at a time, in the order of ascending index.

The **input node** for which we will check whether it is returned by our query on our document is v_{M+N}. Indeed, by our construction, the query will select node v_{M+N} iff the circuit evaluates to true, and no other node will be selected.

It is easy to see that the reduction can be effected in LogSpace. We next argue that it is also correct.

Discussion. We use the ordering of the circuit in that we, intuitively, will evaluate the circuit in Core XPath *one gate at a time*. We treat the circuit as if layered, with all gates of a layer of the same type ("$\wedge$" or "$\vee$") and only exactly one per layer with fan-in greater than one. (Our encoding permits *unbounded fan-in*, including one.) Figure 5 shows this alternative view of the example circuit of Figure 3. The $N = 5$ non-input gates have been aligned using five layers $L_1 \dots L_5$. The smaller empty circles denote "dummy" gates of fan-in one, which are needed to propagate the values of gates that are already available to the layers above. In our encoding, intuitively, all gates of layer L_k have the same type. The type of the dummy gates[7] in layer L_k is thus determined by the type of the one gate of fan-in greater than one (namely G_{M+k}). In the example, all gates of layers $L_1 \dots L_4$ are of type $\wedge$ and the gates of layer L_5 are all of type $\vee$.

The φ_k, ψ_k, and π_k are condition expressions, and there is a natural meaning to "φ_k matches node w" or equivalently "node w satisfies φ_k", which we will denote as

[7] In fact, the types of gates of fan-in one do not matter in the circuit: the conjunction as well as the disjunction of a *single* truth value is the identity. For this reason, and to save space, we do not show the types of the dummy gates in Figure 5.

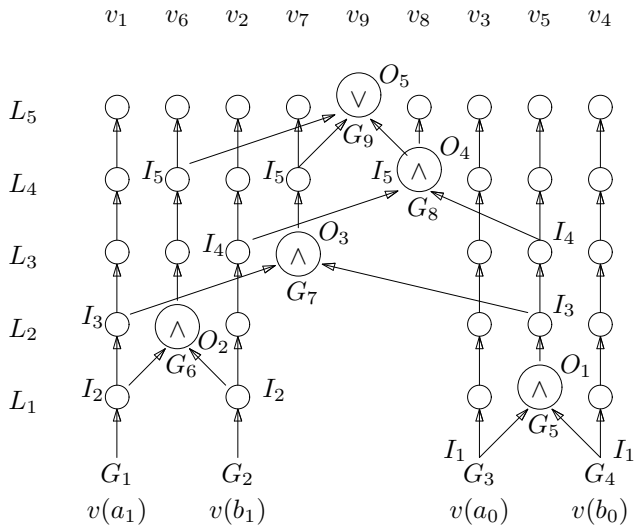

Fig. 5. Circuit of Figure 3 with gates serialized.

$w \in [\![\varphi_k]\!]$ below. Formally, $w \in [\![\varphi_k]\!]$ if and only if query /descendant-or-self::*[φ_k] selects node w. We define $w \in [\![\psi_k]\!]$ and $w \in [\![\pi_k]\!]$ analogously. This notation helps to imagine the query (tree) being processed bottom-up.

Claim. *Let* $0 \le k \le N$. *Then, for all* $1 \le i \le M + k$,

$$v_i \in [\![\varphi_k]\!] \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

This can be shown by an easy induction.

Induction start ($k = 0$). For a given $1 \le i \le M$, the gate G_i is an input gate, whose initial value (either 0 or 1) has been assigned to the corresponding node v_i as a label. By definition, φ_0 is the expression $T(1)$, which selects the nodes labeled 1 and therefore v_i if G_i is true. For the converse, the label 1 is not used elsewhere in the tree besides on nodes $v_1, \ldots, v_M$. Thus our claim holds for $k = 0$.

Induction step. Now assume that our claim holds for $k-1 \ge 0$ (i.e., $v_i \in [\![\varphi_{k-1}]\!]$ iff gate G_i has been established to be true by step $k - 1$). We show that it also holds for k. We proceed by computing first $[\![\pi_k]\!]$, then $[\![\psi_k]\!]$, and finally $[\![\varphi_k]\!]$.

Since π_k = ancestor-or-self::*[$T(G)$ and φ_{k-1}] matches the nodes both labeled G and in $[\![\varphi_{k-1}]\!]$, as well as their descendants, and precisely the nodes $v_1 \ldots v_{M+N}$ are labeled G, for $1 \le i \le M + k - 1$,

$$v_i \in [\![\varphi_{k-1}]\!] \Rightarrow v_i, v_i' \in [\![\pi_k]\!] \quad \text{and} \quad v_i \notin [\![\varphi_{k-1}]\!] \Rightarrow v_i, v_i' \notin [\![\pi_k]\!].$$

Hence, by induction, for $1 \le i \le M + k - 1$,

$$v_i \in [\![\pi_k]\!] \Leftrightarrow v_i' \in [\![\pi_k]\!] \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

In other words, the purpose of the auxiliary path π_k is to turn the property of v_i (relatively to φ_{k-1}) into a property of both v_i and v_i' (relatively to π_k). This step of "copying" the information on the truth value of G_i from v_i also to v_i' will turn

out to be crucial for treating the gate G_{M+k} and the dummy gates on the layer k simultaneously by a single formula ψ_k.

The formula ψ_k is at the heart of our construction and performs the actual computation of the $M+k$ gates in layer k. These are the "dummy" gates of fan-in one (which just propagate the truth values of gates $G_1, \ldots, G_{M+k-1}$ from layer $k-1$ to layer k), plus the one gate G_{M+k} of fan-in greater than one from the input circuit that resides in layer k. Using $[\![\psi_k]\!]$, we compute the truth value of gate G_{M+k} as a property of node v_0 and preserve the truth values of the gates $G_1, \ldots, G_{M+k-1}$ as properties of the nodes $v_1, \ldots, v_{M+k-1}$. We discuss the two forms of gates separately:

Gate G_{M+k}. In our document tree, a node v_i is labeled I_k iff gate G_{M+k} takes input from gate G_i. By the assumed ordering of gates of the input circuit, if v_i is labeled I_k (that is, G_i is a fan-in gate of G_{M+k}), then $i < M+k$. For this case we already know that

$$v_i \in [\![\pi_k]\!] \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

We distinguish two cases, depending on the type of gate G_{M+k}:

—G_{M+k} is an $\wedge$-gate and $\psi_k = \text{not(child::*}[T(I_k) \text{ and not}(\pi_k)])$: Here, $v_0 \in [\![\psi_k]\!]$ iff all children of v_0 that are labeled I_k (that is, the fan-in gates of G_{M+k}) satisfy π_k. By induction, that is the case if and only if G_{M+k} is true.

—G_{M+k} is an $\vee$-gate and $\psi_k = \text{child::*}[T(I_k) \text{ and } \pi_k]$: Here, $v_0 \in [\![\psi_k]\!]$ iff there is a child of v_0 labeled I_k (that is, a fan-in gate of G_{M+k}) which satisfies π_k. By induction, that is the case if and only if G_{M+k} is true.

Thus,

$$v_0 \in [\![\psi_k]\!] \Leftrightarrow \text{gate } G_{M+k} \text{ evaluates to true.}$$

Dummy gates. By construction v_i' has the label I_k for each $1 \leq i < M+k$. Moreover v_i has only one child (namely, v_i'), so when checking whether $v_i \in [\![\psi_k]\!]$, it does not matter whether ψ_k is of the $\wedge$- or the $\vee$-type. Therefore by induction we have for $1 \leq i < M+k$,

$$v_i \in [\![\psi_k]\!] \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

Formula $\varphi_k = \text{descendant-or-self::*}[T(O_k) \text{ and parent::*}[\psi_k]]$ "stores" the truth values of gates G_i in nodes v_i for all $1 \leq i \leq M+k$:

$$v_i \in [\![\varphi_k]\!] \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

Indeed, for all $1 \leq i \leq M+k$, $v_i \in [\![\varphi_k]\!]$ if and only if

—v_i is labeled O_k (thus, by the construction of the document tree, $i = M+k$) and its parent (v_0) satisfies $[\![\psi_k]\!]$ or

—v_i has a descendant (v_i') that is labeled O_k (this is the case for $1 \leq i < M+k$) and whose parent (v_i) satisfies $[\![\psi_k]\!]$.

This proves our claim.

The overall query /descendant-or-self::*$[T(R)$ and $\varphi_N]$ has a nonempty result (consisting of precisely the node v_{M+N}) exactly if the output gate G_{M+N} of the

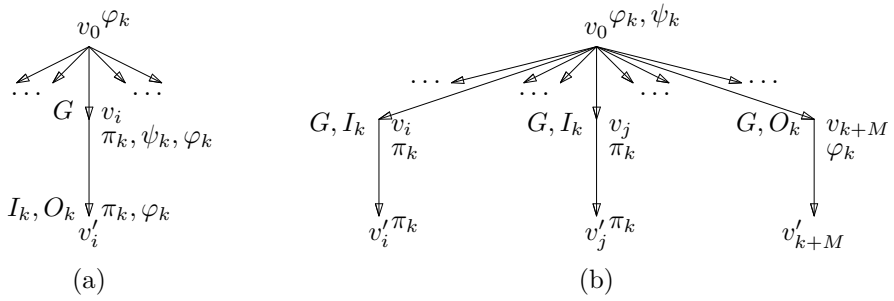

Fig. 6. Schematic design of relevant tree region and $\varphi_k/\psi_k/\pi_k$-matchings made for (a) dummy gates and (b) gates of fan-in greater than one (here, two).

circuit evaluates to true, because v_{M+N} is the only node labeled R and $v_{M+N} \in [\![\varphi_N]\!]$ if and only if G_{M+N} evaluates to true.

In summary, we have provided a LogSpace reduction that maps any monotone Boolean circuit to a Core XPath query and a document tree such that the query evaluated on the tree returns node v_{M+N} precisely if the circuit evaluates to true. As the monotone Boolean circuit value problem is P-complete, our theorem is proven. □

The crucial steps of the above proof are illustrated in the schematic designs of Figure 6, which show the regions of the document tree that we are interested in, the relevant labels, and the matchings of condition expressions φ_k, ψ_k, and π_k at step k, for both cases of gates (dummy gates in Figure 6 (a) and gate G_{M+k} in Figure 6 (b)). Figure 6 (a) shows the matchings implied by $v_i \in [\![\varphi_{k-1}]\!]$ (that is, gate G_i is true in layer $k-1$), namely, $v_i, v_i' \in [\![\pi_k]\!]$, $v_i \in [\![\psi_k]\!]$, and $v_0, v_i, v_i' \in [\![\varphi_k]\!]$. Figure 6 (b) shows matchings that hold for gate G_{M+k} with two fan-in gates G_i and G_j such that $v_i, v_j \in [\![\varphi_{k-1}]\!]$, namely, $v_i, v_i', v_j, v_j' \in [\![\pi_k]\!]$, $v_0 \in [\![\psi_k]\!]$, and $v_0, v_{M+k} \in [\![\varphi_k]\!]$. (Formula φ_k also matches other nodes above and below $v_1 \dots v_{M+k}$, but this does not matter because these nodes are labeled neither G nor R.)

Corollary 3.3. *Core XPath remains* PTime*-hard even if*

(*1*) *the document tree is limited to depth three and*

(*2*) *only the axes child, parent, and descendant-or-self are allowed.*

Proof. The previous proof has the stated properties, except that it uses the ancestor-or-self axis in the definition of π_k. All we need to do is to replace ancestor-or-self::* in π_k by descendant-or-self::*/parent::*. π_k then additionally matches the root node v_0, but this does not matter to the remainder of the construction because v_0 never carries an I_k label and thus never has an impact on ψ_k. □

We overstated the required tree depth in Corollary 3.3 to allow for multiple node labels to be encoded as additional children, as discussed in Remark 3.1. The document trees of the encoding of the proof of Theorem 3.2 are only of depth two.

Note also that the queries used in the encoding essentially do not branch out in terms of axis applications. That is, in each conjunction ("or" is not used) of expressions, there is at most one subexpression that contains an axis application.

3.2 Inside Core XPath

The result of the previous section is essentially negative: As Core XPath is PTIME-hard, it is considered unlikely that a parallel algorithm exists for evaluating all queries of this language. It is thus natural to search for fragments of Core XPath that we can show to be in NC and therefore highly parallelizable. In fact, such a fragment is obtained by removing negation ("not") from Core XPath. This fragment will be called *positive Core XPath*.

For proving the desired upper bound on the complexity of positive Core XPath and some related results in the next section, the XPath query is viewed as a tree or, equivalently, as a term.

DEFINITION 3.4. Let e be a Core XPath expression. W.l.o.g, we may assume that e contains no step-expressions of the form $e = \chi :: a[e_1] \dots [e_k]$ with $k > 1$ since they can be replaced by $\chi :: a[e_1 \text{ and } \dots \text{ and } e_k]$. This is due to the fact that Core XPath does not contain the functions position() and last().

We define the *query-tree* $\mathcal{T}_e$ (which we will denote in term notation) as follows: Each node in $\mathcal{T}_e$ is labeled either by a Boolean operator $(\vee, \wedge, \neg)$ or by a step-expression of the form $\chi :: a$ where $a \in \Sigma \cup \{*\}$ or by the operator / indicating an absolute location path.

$$\mathcal{T}_e = \begin{cases} /(\mathcal{T}_{e_1}) & \text{if } e = /e_1 \\ \vee(\mathcal{T}_{e_1}, \mathcal{T}_{e_2}) & \text{if } e = e_1 \mid e_2 \\ \chi :: a(\mathcal{T}_{e_1}) & \text{if } e = \chi :: a/e_1 \\ \chi :: a & \text{if } e = \chi :: a \\ \chi :: a(\wedge(\mathcal{T}_{e_1}, \mathcal{T}_{e_2})) & \text{if } e = \chi :: a[e_1]/e_2 \\ \chi :: a(\wedge(\mathcal{T}_{e_1}, \text{self} :: a)) & \text{if } e = \chi :: a[e_1] \\ \wedge(\mathcal{T}_{e_1}, \mathcal{T}_{e_2}) & \text{if } e = e_1 \text{ and } e_2 \\ \vee(\mathcal{T}_{e_1}, \mathcal{T}_{e_2}) & \text{if } e = e_1 \text{ or } e_2 \\ \text{not}(\mathcal{T}_{e_1}) & \text{if } e = \text{not}(e_1) \end{cases}$$

Finally, a special tag † is added for the leaf of the tree corresponding to the last step-expression of the path-expression which does not occur inside some predicate. □

For instance, the tree associated to the XPath expression

$$\text{following} :: a[\text{descendant} :: b \text{ or descendant} :: c]/\text{following} :: d$$

is shown in Figure 7.

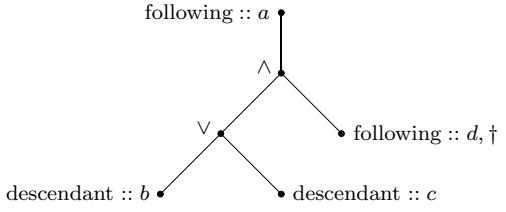

Fig. 7. Query-tree for expression following :: a[descendant :: b or descendant :: c]/following :: d.

This query tree will never be materialized but all queries will be parsed by doing a partial depth-first, left-to-right traversal of the corresponding tree. A pointer to a Core XPath expression will always point to the first character of a step-expression or a Boolean operator and will thus correspond to a node of the query-tree.

Note that a Core XPath expression e is essentially composed of location paths π which may occur in three different forms: π may be an operand of a Boolean expression (e.g., "not(π)", "π and e_1", "π or e_1"), π may be a predicate condition (e.g., $e = \chi :: a[\pi]$), or π may be the main path (i.e., $e = \pi$). Now let n be a node in the query-tree $\mathcal{T}_e$ with label $\chi :: a$. Then $\chi :: a$ is clearly a location step within some location path π contained in e, i.e., π is either of the form $\pi = \pi_1/\chi :: a/\pi_2$ or $\pi_1/\chi :: a[e']/\pi_2$ where π_1 and π_2 are path-expressions consisting of $k \geq 0$ location steps. We shall refer to $\pi_1/\chi :: a$ or $\pi_1/\chi :: a[e']$, respectively, as the "*location path corresponding to* n". Suppose that we want to evaluate the location path π relatively to some context-node u. Then we shall say that a node w from the data tree t "*satisfies*" the location path corresponding to n if w is indeed in the node set resulting from the evaluation of this location path w.r.t. the context-node u.

THEOREM 3.5. *The combined complexity of positive Core XPath is in* LOGCFL.

PROOF. We use Sudborough's characterization of LOGCFL ([Sudborough 1978; 1977]) and exhibit a NAuxPDA $\mathcal{A}$ which, on input (Q, t, c, v) – where Q is a query from positive Core XPath, t is a data tree, c is a context-node, and v is a node in t – decides in logarithmic space and polynomial time whether the evaluation of the query Q over the data tree t for the context-node c yields a node-set containing v. $\mathcal{A}$ will maintain a pointer p on the input XPath expression that will perform a partial depth-first, left-to-right traversal of the query-tree $\mathcal{T}_Q$. This traversal is *partial* in the sense that, for every node labeled with $\vee$, we will non-deterministically choose one child and completely exclude the subtree rooted at the other child from our traversal.

Moreover, $\mathcal{A}$ will use another pointer q on the input data tree t, such that the next step-expression $\chi :: a$ to be evaluated along our partial traversal of $\mathcal{T}_Q$ will be evaluated relatively to the node pointed to by q.

$\mathcal{A}$ starts by setting p to the root of the input query and q to the input context node c. In the main loop of the automaton $\mathcal{A}$, the "current node" n in the query-tree (i.e., the node pointed to by p) is processed as follows: If the label of n is / (indicating an absolute location path), then q is set to the root node of t and p is moved to the child node of n. If the label of n is $\vee$, then $\mathcal{A}$ guesses a number $i \in \{1, 2\}$ and p is moved on to the i-th child of n. In case of label $\wedge$, the automaton pushes the pointer q onto the stack and sets p to the first child of n. In all of these cases, the automaton $\mathcal{A}$ continues the main loop for the new value of p.

Finally, suppose that the label of n is a step-expression $\chi :: a$. Then $\mathcal{A}$ guesses a node u of t such that the path from the node pointed to by q to u satisfies the step-expression. If no such node exists then $\mathcal{A}$ rejects. If n contains the special tag † then $\mathcal{A}$ checks that $u = v$ holds, otherwise it rejects.

The continuation of $\mathcal{A}$ after processing a step-expression is as follows: If n has no successor (i.e., n contains the special tag † and the above check $u = v$ was positive), then $\mathcal{A}$ accepts. Otherwise, q is set to the node u in t that has just been guessed. If n has a child, then this child is the next node to be processed by $\mathcal{A}$. Finally, if n

has no child, then the successor of n in our *partial* depth-first traversal (recall that of all nodes labeled with $\vee$, only one child is selected whereas the subtree rooted at the other child is excluded from the traversal) must be the second child of some ancestor of n, such that this ancestor has the label $\wedge$. Then $\mathcal{A}$ pops q from the stack and continues with this second child of the $\wedge$-node.

It can be verified that this algorithm is correct. Suppose that $\mathcal{A}$ is processing a node n labeled with a step-expression $\chi :: a$. Of course, the node u guessed by $\mathcal{A}$ satisfies the step-expression $\chi :: a$. Moreover, the result of this step (i.e., the node u pointed to by q) is used as the context-node for a predicate (if present) and for the continuation of the location path after the step $\chi :: a$. Hence, by an easy induction argument, it can be shown that whenever $\mathcal{A}$ guesses a node u, then u in fact satisfies the whole location path corresponding to the node n in the query-tree. For the node n carrying the additional label †, the automaton checks that the corresponding location path (which is the input XPath expression e) is not only satisfied by "some" node u but indeed by the input node v. Note that absolute location paths $/\pi$ are handled correctly by $\mathcal{A}$, i.e., π is indeed evaluated relatively to the root of the data tree t. Likewise, Boolean operators are handled correctly: First of all, the context-node relatively to which $\vee$ and $\wedge$ are processed is exactly the node u obtained by the last location step. For $\vee$ it is clearly correct to check for only one of its operands that it is satisfied. For $\wedge$, the handling of the stack guarantees that also for the evaluation of the second operand the correct context-node is used. Finally, note that the effect of the Boolean function "and" is indeed the same as the effect of first restricting a location step by some predicate and then continuing the location path, i.e., $\chi :: a[\pi_1 \text{ and } \pi_2]$ is satisfied by some node of t iff $\chi :: a[\pi_1]/\pi_2$ is satisfied.

As for the complexity of $\mathcal{A}$, note that, given an XPath expression as a string, the pointer to the next node in this partial depth-first, left-to-right traversal can be computed in LogSpace. Moreover, $\mathcal{A}$ uses only a constant number of pointers to its input and thus indeed works in LogSpace. In the main loop, $\mathcal{A}$ processes each node of the query-tree at most once. Moreover, the work carried out when processing a node can be clearly done in polynomial time. In particular, the check performed by $\mathcal{A}$ when processing a step-expression can be done in time $O(|t|)$. Thus the NAuxPDA works in time $O(|e| \cdot |t|)$. This proves the theorem. □

Actually, rather than completely forbidding negation, it suffices to bound the depth of negation in order to guarantee that the evaluation of Core XPath expressions is in LogCFL.

Definition 3.6. Let e be an XPath expression. We say that "the depth of negation of e is bounded by 0" iff the not-function does not occur at all in e. For any $k > 0$, we say that "the depth of negation of e is bounded by k" iff for any subexpression not(e') occurring in e, the depth of negation of e' is bounded by some l with $0 \leq l < k$.

Theorem 3.7. *The combined complexity of positive Core XPath queries augmented by negation with bounded depth is in* LogCFL.

Proof. We proceed by induction on the bound k of the depth of negation:

Induction start ($k = 0$), i.e., the input query contains no negation at all. This case is covered by Theorem 3.5.

Induction step. Now assume that our claim holds for $k - 1 \geq 0$, i.e., on input (Q', t, c', v') (where Q' is a Core XPath query with depth of negation bounded by $k - 1$) one has to decide whether the node v' lies in the node set that we get by evaluating the query Q' on the data tree t for the context node c'. For short, we write $v' \in Q'(t)$. Recall from Proposition 2.4 that LogCFL is closed under complement. Hence, there exists a log-space and polynomial-time NAuxPDA $\bar{\mathcal{A}}_{k-1}$ that decides the complement problem, i.e., $v' \notin Q'(t)$.

Then we construct a NAuxPDA $\mathcal{A}_k$ for Core XPath expressions with depth of negation bounded by k as follows: Let (Q, t, c, v) denote the input to $\mathcal{A}_k$. Note that "not(e)" is equivalent to "not(self::*[e])" for any Core XPath expression e. (In fact, this equivalence holds for any XPath expression e without position() and last() function, since the self-step only alters context-position and context-size, whereas it leaves the context-node for the evaluation of e unchanged.) Hence, w.l.o.g., we may assume that the not-function only occurs in the form not(π) for a location path π = self::*[e].

Analogously to the NAuxPDA $\mathcal{A}$ in the proof of Theorem 3.5, the automaton $\mathcal{A}_k$ carries out a partial depth-first, left-to-right traversal of the query-tree. For nodes labeled with /, $\vee$, $\wedge$, or $\chi :: a$, we proceed as before. However, when we reach a sub-expression not(π) of Q with π = self::*[e] (i.e, the pointer p of the NAuxPDA $\mathcal{A}$ points to a node n in the query-tree labeled with "not"), then we proceed as follows: Note that we have to check that the current context-node c' is not selected when evaluating π relatively to c' itself. By definition, the depth of negation of π is bounded by $k-1$. Hence, the desired check can be done by running the NAuxPDA $\bar{\mathcal{A}}_{k-1}$ on the input (π, t, c', c').

The correctness of this algorithm follows immediately from the correctness proof of Theorem 3.5. Since the depth of negation is bounded by some constant k, there are at most k nested oracles $\bar{\mathcal{A}}_0, \ldots \bar{\mathcal{A}}_{k-1}$ working in parallel. Each of these oracles works in LogSpace and PTime. Hence, also the complexity of the automaton $\mathcal{A}_k$ is within these desired space and time bounds. For the space complexity, it is essential that the oracle $\bar{\mathcal{A}}_{k-1}$ of course operates on the same input tape as $\mathcal{A}_k$ (and we only pass on a pointer to the sub-expression π rather than π itself). □

As we will see next, the LogCFL upper bound established previously is also tight. In the following, we will abbreviate the n-times repeated application of an axis χ, $(\chi::*/)^{n-1}\chi::*$, as $\chi^n::*$. By χ^n::c, we denote $(\chi::*/)^{n-1}\chi$::c.

Theorem 3.8. *Positive Core XPath is* LogCFL*-hard (combined complexity).*

Proof. By reduction from SAC^1 circuit value, which is LogCFL-complete (see Proposition 2.2). As shown in [Gottlob et al. 2001], we may assume without loss of generality that all $\wedge$-gates have fan-in 2 and that the SAC^1 circuits are layered such that, for each k, the gates at height k are all of the same type (either $\wedge$ or $\vee$) and the internal gates of layer k only take input from gates of layer $k - 1$. (The input gates are at height 0.) Given a SAC^1 circuit, we employ a reduction which is a modified version of the one of the proof of Theorem 3.2.

Let M be again the number of the input gates of the circuit and let N be the

number of internal gates. Let K be the number of layers in the circuit, that is, the height of the circuit.

The **document tree** consists of nodes u_j, v_i, and $w_{i,j}$ for all $1 \leq i \leq M+N$, $M+1 \leq j \leq M+N$. The root node is u_{M+1}, and there are edges from u_j to u_{j+1} for $M+1 \leq j < M+N$, from u_{M+N} to v_i and from v_i to $w_{i,M+1}$ for all $1 \leq i \leq M+N$, and from $w_{i,j}$ to $w_{i,j+1}$ for all $1 \leq i \leq M+N$, $M+1 \leq j < M+N$.

A node v_i has a label from $\{G, R, 0, 1\}$ precisely if node v_i in the construction of the proof of Theorem 3.2 has that label. In addition, we assign label I_k to node $w_{i,j}$ iff internal $\vee$-gate G_j is in layer $1 \leq k \leq K$ and takes input from gate G_i. We assign label I_k^1 (resp., I_k^2) to node $w_{i,j}$ iff internal $\wedge$-gate G_j is in layer k and gate G_i is its left (resp., right) input. We assign label O_k to node $w_{j,j}$ iff internal gate G_j is in layer k.

The **query** evaluating the circuit is

$$/\text{descendant-or-self::*}[T(R) \text{ and } \varphi_K]$$

with the condition expressions

$$\begin{aligned}
\varphi_k &:= \text{descendant-or-self::*}[T(O_k) \text{ and parent}^{N+1}\text{::*}[\psi_k]] \\
\psi_k &:= \begin{cases} \text{child}^{N+1}\text{::*}[T(I_k) \text{ and } \pi_k] & \dots \text{ layer } k \text{ consists of } \vee\text{-gates} \\ \text{child}^{N+1}\text{::*}[T(I_k^1) \text{ and } \pi_k] \text{ and} \\ \quad \text{child}^{N+1}\text{::*}[T(I_k^2) \text{ and } \pi_k] & \dots \text{ layer } k \text{ consists of } \wedge\text{-gates} \end{cases} \\
\pi_k &:= \text{ancestor-or-self::*}[T(G) \text{ and } \varphi_{k-1}].
\end{aligned}$$

for $1 \leq k \leq K$ and $\varphi_0 := T(1)$.

Discussion. The proof uses the same ideas as that of Theorem 3.2, but now we evaluate $\wedge$-gates using conjunction rather than universal quantification (which we had obtained via negation). This requires us to define ψ_k for $\wedge$-layers k using two copies of π_k. Even though the query grows exponentially in the height of the circuit K, it can be computed in LogSpace because K is only logarithmic in the size of the input. But since the number of internal gates in the circuit is much larger than K (linear), we cannot compute the values of internal gates one at a time as we did in the proof of Theorem 3.2. This causes the second modification of that proof. We now evaluate all the gates of a layer of the circuit in a single step. We achieve this by computing the value of gate G_j at node u_j, which is always precisely $N+1$ levels above the unique descendant ($w_{j,j}$) of node v_j in the data tree that has an O_k label. The reduction exploits the fact that in a layered circuit, each internal gate at layer k takes input only from gates of layer $k-1$, thus we do not need "dummy gates" in this proof. The correctness of the reduction can be shown by a simple induction analogously to that of the proof of Theorem 3.2. □

Let PF be the fragment of Core XPath containing only the location paths, without conditions (i.e., no expressions enclosed in brackets are permitted).

Theorem 3.9. *PF is* NLogSpace*-complete under* LogSpace*-reductions with respect to combined complexity.*

Proof. Membership in NLogSpace is obvious: we can just guess one node after the other along the path while we verify each location step in LogSpace.

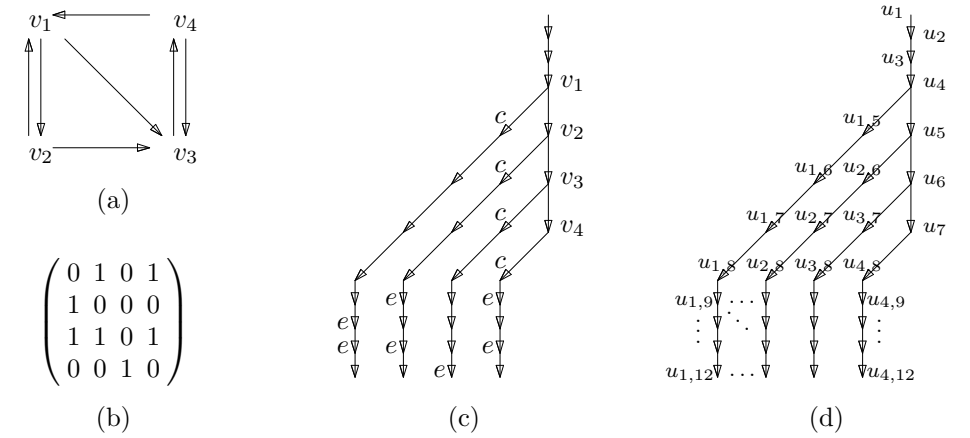

Fig. 8. Graph (a), its transposed adjacency matrix (b), and data tree with node labels (c) and node names (d) shown.

NLogSpace-hardness follows from a LogSpace-reduction from the *directed graph reachability* problem, which is NLogSpace-complete (cf. [Papadimitriou 1994]). Given a directed graph $G = (V, E)$, we compute data tree $T = (V_T, E_T)$ as

$$V_T = \{u_1, \ldots, u_{2\cdot|V|-1}\} \cup \{u_{k,l} \mid 1 \le k \le |V|,\ k + |V| \le l \le 3 \cdot |V|\},$$

$$E_T = \{\langle u_i, u_{i+1}\rangle \mid u_i, u_{i+1} \in V\} \cup \{\langle u_{i+|V|-1}, u_{i,i+|V|}\rangle \mid 1 \le i \le |V|\} \cup \{\langle u_{k,l}, u_{k,l+1}\rangle \mid u_{k,l}, u_{k,l+1} \in V\}.$$

Nodes $u_{i,i+|V|}$ are labeled c, nodes $u_{i+|V|-1}$ are labeled v_i, and node $u_{i,j+2\cdot|V|}$ is labeled e iff $\mathbf{a}_{ij} = 1$, where $\mathbf{a}$ is the adjacency matrix of G. An example of the construction can be found in Figure 8.

Let

$$\pi := \text{child::}c/\text{descendant::}e/\text{parent}^{2\cdot|V|}\text{::*}/\text{child}^{|V|}\text{::}c/\text{parent::*}.$$

Claim. *There is an edge from v_i to v_j in G if and only if node $u_{j+|V|-1}$ is reachable from node $u_{i+|V|-1}$ through path π.*

Indeed,

(1) child::c: node $u_{i+|V|-1}$ has precisely one child labeled c, $u_{i,i+4}$.
(2) descendant::e: By the definition of T, a descendant node of $u_{i,i+4}$, say $u_{i,2*|V|+j}$, is labeled e if and only if there is an edge from v_i to v_j in G.
(3) parent$^{2\cdot|V|}$::*: The $(2 \cdot |V|)$-th ancestor of $u_{i,2*|V|+j}$ is u_j.
(4) child$^{|V|}$ :: c: Node u_j has only one $|V|$-th descendant that is labeled c, $u_{j,j+|V|}$.
(5) parent::*: The parent of $u_{j,j+|V|}$ is $u_{j+|V|-1}$.

Note that $u_{i+|V|-1}$ is the unique node in T labeled v_i and $u_{j+|V|-1}$ is the unique node in T labeled v_j. It is easy to verify that, given a positive integer m, the query

$$/\text{descendant::}v_i/\underbrace{\pi/\ldots/\pi}_{m \text{ times}}/\text{self::}v_j$$

computes the node ($u_{j+|V|-1}$) labeled v_j if and only if node v_j is reachable from node v_i in graph G in m steps. To extend reachability in m steps to reachability in at most m steps, we add a self-loop for each node of G (or equivalently, set the main diagonal of the adjacency matrix to ones only). For $m = |E|$, this is graph reachability. □

3.3 Restricted-Axis Fragments of Core XPath

In this section, we take an approach different from the one of the previous section to obtain fragments of Core XPath which are highly parallelizable and can thus be processed with limited resources. Rather than pruning the language features, we restrict the navigational capabilities, i.e. the supported axes. We obtain better upper bounds for a number of sublanguages of Core XPath with restricted navigational capabilities.

In the following, by $\mathrm{PF}_{\downarrow}$ we denote PF with the permitted axes restricted to the downward axes "child" and "descendant".

THEOREM 3.10. *$PF_{\downarrow}$ is in* LOGSPACE *w.r.t. combined complexity.*

PROOF. We consider the problem of deciding, given a node w, whether w matches a $\mathrm{PF}_{\downarrow}$ query

$$/s_{1,1}/\ldots/s_{1,k_1}//\ldots//s_{n,1}/\ldots/s_{n,k_n}$$

(where the $s_{i,j}$ are node tests, i.e. labels or '*') on a given data tree.

This can be achieve in LOGSPACE because (i) it is enough to consider the path from w to the root and (ii) more importantly for each section $s_{i,1}/\ldots/s_{i,k_i}$ it is enough to consider the lowest possible place on that path with the corresponding pattern as it occurs after a descendant axis.

By the parent of a node and the depth of a node in the data tree we understand the usual graph-theoretic notions (where the depth of the root node is zero). We assume that k_1 may be zero to allow for queries of the form

$$//s_{2,1}/\ldots/s_{2,k_2}//\ldots//s_{n,1}/\ldots/s_{n,k_n}.$$

We apply the following test to match the query (backward) on the path from w to the root.

```
v := w;
for each i in the list [n,...,1] do
begin
   if depth(v) + 1 < k_i then fail;
   if i = 1 then for (depth(v) + 1 - k_1) times do v := parent(v);
   if procedure p fails then
   begin
      if i = n or i = 1 then fail
      else do v := parent(v) while procedure P fails;
   end;
end;
success
```

where the auxiliary procedure p is as follows:

```
/* v is a global variable */
u := v;
for each s_{i,j} in the list [s_{i,k_i}, ..., s_{i,1}] do
begin
   if s_{i,j} matches the label of u then u := parent(u);
   else fail;
end;
v := u;
success
```

We do not need to explicitly compute the lists $[n, \ldots, 1]$ or $[s_{i,k_i}, \ldots, s_{i,1}]$; instead one "cursor" for processing the query backward is sufficient for our purposes (the "//" serve as separators between node tests s_{i,k_i} and $s_{i+1,1}$). Note that the processing is not strictly backwards; the cursor sometimes has to backtrack to the node test just before the next occurrence of "//" to the right to retest the auxiliary procedure p or to compute some k_i. In addition, we only need three registers for storing tree nodes (u, v, and one further to compute depth(v)) and two counters, for depth(v) and k_i. Each of these is of size logarithmic in the size of the input.

It is easy to check (by induction on n) that the algorithm above is correct. □

Let Core XPath^1 be the restriction of Core XPath that supports only the axes *self, child*, and *parent*. Core XPath^1 allows only step-expressions which can go to a node at distance at most one from the current node.

THEOREM 3.11. *Core XPath^1 is in* LOGSPACE *w.r.t. combined complexity. This problem remains in* LOGSPACE *even if we augment Core XPath^1 by the "axes"* previous-sibling *and* next-sibling.[8]

PROOF. The first step of the algorithm is to remove the step-expressions that look *backward*: *parent* and *previous-sibling*. This can be done in LOGSPACE using rewriting rules. The idea is the same as for computing an automaton for the reverse language of L (the set of mirrors of words from L) given an automaton from L. This is achieved using simple rewriting rules replacing a *parent* relation using *child* relations. This fact has already been established and generalized in [Olteanu et al. 2002] which gives a complete list of rewriting rules removing the backward axes from an XPath expression. In our much simpler setting it is immediate to check that this can be done in LOGSPACE.

The input of the algorithm is a query Q and a data tree t. The algorithm will perform a depth-first, left-to-right traversal of the query-tree of Q and, for each (query) node, it will look for a matching (data) node by doing a depth-first, left-to-right traversal of the data tree. Those two traversals require only LOGSPACE as it suffices to maintain only two pointers on the current query and data nodes. The trick is that when no matching is found, backtracking can be performed in LOGSPACE as it suffices to execute each axis-step in reverse order.

[8]Actually, *previous-sibling* and *next-sibling* are not contained in XPath. However, these steps to siblings at distance 1 can be easily expressed in XPath using position arithmetics, namely as *preceding-sibling*[1] and *following-sibling*[1], respectively.

We now give more details. Recall that Boolean expressions can be validated in LogSpace by skipping the disjunctions as soon as one of its subtrees evaluates to 1 and successively evaluating all the subtrees of a conjunction. Negation is handled by reversing the truth value. In order to simplify the presentation in the following, we ignore the nodes of the query-tree labeled with a Boolean predicate and always evaluate all subtrees. Using the procedure for Boolean expressions described above, it is straightforward to extend it in order to include Boolean gates while maintaining the LogSpace constraint.

Let `next-sibling`, `child`, `parent` and `leaf` be LogSpace subroutines which, given a pointer to a node, compute the pointer of its right sibling, parent, first child or check whether the node is a leaf.

Let `match`(i, β, j) be a subroutine which takes as input two pointers i and j to the data tree and one pointer β to the query tree, and returns a pointer k to a data node such that (i) k is strictly bigger than j in the depth-first, left-to-right traversal of the data tree, and (ii) the path from i to k satisfies the location step defined by the label of β. `match` looks for the next (after j) data node which matches the current step as described by the query node β. If no such node exists, it returns 0. This subroutine can easily be implemented so as to run in LogSpace.

Let `backtrack`(α, i) be the subroutine which takes as input a pointer α to the query-tree and a pointer i to a node n of the data tree and returns a pointer k to a data node. It simply returns the pointer to the node m of the data tree such that the transition from m to n corresponds to the axis of the label of the query node pointed to by α.

With the above subroutines, Core XPath[1] queries can be evaluated using the following algorithm (where `done` is a shortcut for α `= root` $\wedge\ v \neq \bot$:

```
/* initialization */
α := root(query-tree); v:= ⊥; i := match(0,α,0);
if i = 0 then return 0;

/* main loop */
while ¬done do begin
  /* case analysis depending on the value of the current query node */

  /* case 1 :  we don't know yet the current status of this node */
  if v = ⊥ then begin
    /* if the current query node is a leaf, then the evaluation of
    /* this branch of the query is successful */
    if leaf(α) then v := 1
    /* if not we proceed downwards in the query-tree*/
    else begin β := child(α); j:=match(i,α,i);
      /* there is no matching data node, the current evaluation fails */
      if j = 0 then v := 0
      /* there is one, we proceed with it*/
      else begin α := β; i:=j; v := ⊥ end
    end
  end
```

```
  /* case 2 :  so far it is a success, we now have to investigate */
  /* the next sibling in the query-tree if one exists or else we */
  /* propagate the success upward */
  else if v = 1 then begin
    β := parent(α); j:= backtrack(α,i);
    if α has no sibling then begin α := β; i := j; v:=1 end
    else begin β' := next-sibling(α); k:=match(j, β',j);
      /* there is no matching data node, the current evaluation fails */
      if k = 0 then begin v := 0; α:= β; i:=j end
      /* there is one, we proceed with it*/
      else begin α := β'; i:=k; v := ⊥ end;
    end
  end

  /* case 3 :  so far it is a failure, we backtrack */
  else if v = 0 then begin
    β := parent(α); j:= backtrack(α,i);
    /* we now look for another possible candidate
    /* (a data node bigger than i) */
    k:=match(α,j,i);
    /* if there is none, proceed upward */
    if k=0 then begin α := β; i:=j; v := 0 end
    /* if there is one we continue with it */
    else begin i:=k; v:=⊥ end
  end
end; /*while*/
return v;
```

It is immediate to see that the algorithm is correct and in LogSpace. □

3.4 Parallelizing WF

We are now going to search for restrictions on WF (Wadler fragment) that push the complexity of the query evaluation problem down to LogCFL. To achieve this, we require that scalar values (i.e., values different from node sets) can be stored in logarithmic space. Moreover, we also have to exclude two important constructs from WF, namely negation and *iterated predicates*, i.e., expressions of the form $\chi :: a[e_1] \dots [e_k]$ with $k \geq 2$. The resulting XPath fragment will be referred to as the "positive" (or "parallel") WF (short pWF). It is formally defined as follows:

Definition 3.12. pWF is obtained by restricting WF in the following way:

(1) Expressions of the form $\chi :: a[e_1] \dots [e_k]$ with $k \geq 2$ are not allowed, where χ denotes an axis, a is a node test and the e_i's are XPath expressions.
(2) Negation may not be used.
(3) The nesting depth of arithmetic operators is bounded by some constant k. □

The first two restrictions above mean that the grammar from Definitions 2.7 and 2.8 has to be modified as follows:

locstep ::= axis '::' ntst '[' bexpr ']'
bexpr ::= bexpr '`and`' bexpr | bexpr '`or`' bexpr | locpath | nexpr relop nexpr.

REMARK 3.13. Note that positive Core XPath is strictly a fragment of pWF. This is due to the fact that the first restriction above plays no role in Core XPath. More generally, an XPath expression of the form $\chi :: a[e_1] \dots [e_k]$ is equivalent to $\chi :: a[e_1 \text{ and } \dots \text{ and } e_k]$ as long as position() and last() are not used.

THEOREM 3.14. *pWF is in* LOGCFL *with respect to combined complexity.*

PROOF. It is sufficient to consider the following decision problem: The input is given through a tuple $(t, Q, \vec{c}, v)$, where Q is a pWF query, t is a document tree, $\vec{c} = \langle cn, cp, cs \rangle$ is a context-triple and v is a value (of type Boolean, number or a single node in t). In case of result type Boolean, we require that $v = \text{true}$. Then we have to decide whether, relatively to the context $\vec{c}$, the query Q over the document t evaluates to v (in case of result type Boolean or number) or to some node set X with $v \in X$ (in case of result type node set).

The LOGCFL-membership of this problem is shown by appropriately modifying the NAuxPDA in the proof of Theorem 3.5. First of all, we have to modify the definition of the query-tree $\mathcal{T}_e$ of an XPath expression e. W.l.o.g., we may assume that all type conversions in Q are made implicit. In particular, $\chi :: a[\pi]$ is replaced by $\chi :: a[\text{boolean}(\pi)]$. Likewise, location paths π occurring as operands of Boolean expressions are replaced by $\text{boolean}(\pi)$. Moreover, the abbreviated syntax $\chi :: a[e]$ for some number expression e is replaced by $\chi :: a[\text{position}() = e]$. Then we have to consider expressions e of the form $Op(e_1, \dots, e_l)$ (written in prefix-notation), where Op is one of the operators "|", "boolean()", "and", "or", an arithmetic operator $(+, -, *, \dots)$, or a relational operator $(=, \neq, \leq, \dots)$. In this case, we simply set $\mathcal{T}_e = Op(\mathcal{T}_{e_1}, \dots, \mathcal{T}_{e_l})$ – where we are again using term notation. Moreover, step-expressions are no longer treated like the "and"-operator. Instead we define

$$\mathcal{T}_e = \begin{cases} \chi :: a & \text{if } e = \chi :: a \\ \chi :: a(\mathcal{T}_{e_1}) & \text{if } e = \chi :: a/e_1 \\ \chi :: a(\mathcal{T}_{e_1}) & \text{if } e = \chi :: a[e_1] \\ \chi :: a(\mathcal{T}_{e_1}, \mathcal{T}_{e_2}) & \text{if } e = \chi :: a[e_1]/e_2 \end{cases}$$

Finally, the special symbol † is no longer needed.

Our modified NAuxPDA $\mathcal{A}'$ will maintain a pointer p to the input XPath expression whose movements are again interpreted as a partial depth-first, left-to-right traversal of the query-tree. Suppose that p points to some node n in the query-tree and let $\mathcal{T}_e$ be the subtree rooted at n. Then we will write $expr(p)$ to refer to the corresponding sub-expression e of Q.

The modified NAuxPDA $\mathcal{A}'$ has to store more information on the worktape and also on the stack. In total, we need variables cn, cp, cs, r (for context-node, context-position, context-size, and result at some node n in the query-tree) as well as cn_i, cp_i, cs_i, r_i with $i \in \{1, 2\}$ (for each child node of n). Actually, if the result r or r_i is of type node set, then it suffices to store only one representative from the resulting node set r or r_i, respectively. Note that in Theorem 3.5, cp and cs are not needed, since Core XPath does not contain the functions position() and size(). Moreover, the necessity to store the result values r and r_i is mainly due to the possibility of arithmetic expressions in pWF.

Finally, also the partial traversal of the query-tree has to be slightly modified. Most importantly, each node n in the query-tree has to be processed twice: When n is visited for the *first time*, we guess values for the context cn, cp, cs and for the result r. In case of the root of $\mathcal{T}_Q$, rather than guessing $\langle cn, cp, cs\rangle$ and r, we assign the values of the input context $\vec{c}$ and the input result value v, respectively. Before $\mathcal{A}'$ moves on to the next node in $\mathcal{T}_Q$, the values cn, cp, cs, r at n are pushed onto the stack. When $\mathcal{A}'$ has finished processing the entire subtree below n, then n is visited for the *second time*. At this stage, the values of cn, cp, cs, r as well as cn_i, cp_i, cs_i, r_i for all required child nodes of n can be popped from the stack. In case of the operators "|" and "or", the values cn_i, cp_i, cs_i, r_i of exactly one child of n are required. In all other cases, the values cn_i, cp_i, cs_i, r_i of each child of n are required. On this second visit of n, the NAuxPDA $\mathcal{A}'$ has to check the consistency of the guesses cn, cp, cs, r and cn_i, cp_i, cs_i, r_i (with $i \in \{1, 2\}$) with the XPath expression $expr(p)$. For a leaf node n, the two visits of n collapse to a single one, i.e., first $\mathcal{A}'$ guesses values for cn, cp, cs, and r. Then $\mathcal{A}'$ checks the consistency of these guesses with the XPath expression $expr(p)$.

When a consistency check fails, then $\mathcal{A}'$ rejects. Otherwise $\mathcal{A}'$ pushes cn, cp, cs, r back onto the stack and p is moved on to the next node in $\mathcal{T}_Q$. All possible kinds of checks (depending on the form of $expr(p)$) are given in Table I, where we use the following notation: We write e for any XPath expression and π for a location path. By RelOp and ArithOp we denote relational operators ($=, \neq, \leq, \dots$) and arithmetic operators ($+, -, *, \dots$), respectively. The set of all nodes in t is denoted by dom and root denotes the conceptual root node in the XPath data model (cf. [World Wide Web Consortium 1999]).

Our partial traversal of $\mathcal{T}_Q$ starts at the root of $\mathcal{T}_Q$. Eventually, we shall have visited all nodes of $\mathcal{T}_Q$ that are required to determine the overall result and we shall come back to the root of $\mathcal{T}_Q$. If all the consistency checks thus carried out were successful, then $\mathcal{A}'$ accepts. As soon as one such check fails, then $\mathcal{A}'$ rejects.

The correctness of $\mathcal{A}'$ can be shown similarly to the correctness of $\mathcal{A}$ in the proof of Theorem 3.5. Moreover, the modified NAuxPDA $\mathcal{A}'$ still works simultaneously in LogSpace and PTime. In particular, for the bound on the *space complexity*, two observations are crucial: First, for pWF, every node in the query-tree has at most 2 child nodes. Second, we never have to compute node sets of intermediate results explicitly, e.g. checking $r \in Y$ and determining the position of r in Y and the size of Y can be done without explicitly computing the node set Y itself (cf. the consistency check for $\chi :: a[e]$ in Table I). □

Remark 3.15. It is well known that the complexity class LogCFL is inside the class NC^2 of problems solvable in time $O(\log^2 n)$ with polynomially much hardware working in parallel. In fact, given this intuition and the insights obtained from the reduction to NAuxPDA, it is not hard to find a highly parallel algorithm for evaluating pWF queries. The intuition for matching straight-line path queries (cf. our PF fragment from Section 3.2) is similar to parallel algorithms for graph reachability (cf. [Papadimitriou 1994]); however, rather than connecting nodes in a graph, the goal is to connect contexts with nodes in the query result. Additional synchronization is required for branches in the query-tree (e.g. "and"), which is not surprising as graph reachability is in NLogSpace and thus presumably simpler

expressions $expr(p)$ **at leaf nodes of the query-tree** $\mathcal{T}_Q$	
$expr(p)$	local consistency condition
$\chi :: a$	r can be reached from cn via $\chi :: a$
position()	$r = cp$
last()	$r = cs$
$const$	$r = const$

expressions $expr(p)$ **at internal nodes of the query-tree** $\mathcal{T}_Q$	
$expr(p)$	local consistency condition
$/\pi$	$cn_1 = \text{root} \wedge r = r_1$
$\pi_1 \mid \pi_2$	$(cn = cn_1 \wedge r = r_1) \vee (cn = cn_2 \wedge r = r_2)$
π_1/π_2	$(cn = cn_1 \wedge cn_2 = r_1 \wedge r = r_2)$
$\chi :: a[e]$	let $Y = \{y \in \text{dom} \mid\ y$ can be reached from cn via $\chi :: a\}$ $r \in Y \wedge$ (let $cp_{new} =$ position of r in Y, let $cs_{new} = \lvert Y \rvert$ $cn_1 = r \wedge cp_1 = cp_{new} \wedge cs_1 = cs_{new} \wedge r_1 = \text{true}$)
boolean(π)	$r = \text{true} \wedge (cn_1 = cn \wedge cp_1 = cp \wedge cs_1 = cs \wedge r_1 \in \text{dom})$
e_1 and e_2	$r = \text{true} \wedge [(cn_1 = cn \wedge cp_1 = cp \wedge cs_1 = cs \wedge r_1 = \text{true}) \wedge$ $(cn_2 = cn \wedge cp_2 = cp \wedge cs_2 = cs \wedge r_2 = \text{true})]$
e_1 or e_2	$r = \text{true} \wedge [(cn_1 = cn \wedge cp_1 = cp \wedge cs_1 = cs \wedge r_1 = \text{true}) \vee$ $(cn_2 = cn \wedge cp_2 = cp \wedge cs_2 = cs \wedge r_2 = \text{true})]$
e_1 RelOp e_2	$r = \text{true} \wedge r_1$ RelOp $r_2 \wedge [(cn_1 = cn \wedge cp_1 = cp \wedge cs_1 = cs) \wedge$ $(cn_2 = cn \wedge cp_2 = cp \wedge cs_2 = cs)]$
e_1 ArithOp e_2	$r = r_1$ ArithOp $r_2 \wedge [(cn_1 = cn \wedge cp_1 = cp \wedge cs_1 = cs) \wedge$ $(cn_2 = cn \wedge cp_2 = cp \wedge cs_2 = cs)]$

Table I. Local consistency checks for pWF.

than a LogCFL-complete problem. However, at the branches, the subexpressions below can be evaluated in parallel before finalizing the branch (i.e., proceeding bottom-up). □

The next result shows that pWF is in a sense a maximal LogCFL fragment of WF. Of course, we are not really interested in dealing with arbitrarily big number expressions. As far as the other two restrictions in Definition 3.12 are concerned, none of them can be simply omitted. This is clear for negation, as Core XPath is strictly a fragment of pWF extended by negation. As shown next, the final restriction is essential as well. By iterated predicates, we again refer to location steps of the form $\chi :: a[e_1] \dots [e_k]$ with $k \geq 2$.

Theorem 3.16. *The combined complexity of pWF queries extended by iterated predicates is* PTime*-complete.*

Proof. The proof is by an appropriate modification of the data tree t and the location paths φ, ψ, and π from the proof of Theorem 3.2.

XML document tree. We extend t to t' by adding one additional child w_i to every node v_i with $i \in \{0, \dots, M+N\}$ (as the right-most child, say). Each node w_i is labeled W. Hence, the condition $T(W)$ is fulfilled exactly by these new nodes. Moreover, for the node v_0, we introduce an additional label A ("auxiliary"). Thus,

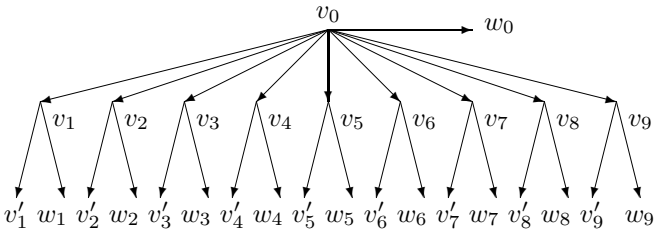

Fig. 9. Document tree t' of the proof of Theorem 3.16.

the condition $T(A)$ is only fulfilled by the root node v_0. For the use of these labels $T(W)$ and $T(A)$, recall Remark 3.1. The new document tree t' corresponding to the example in the proof of Theorem 3.2 is shown in Figure 9.

XPath query. The desired query (encoding the value of the gate G_{M+N}) is now

$$/\text{descendant-or-self::*}[T(R) \text{ and } \varphi'_N]$$

where the auxiliary location paths φ'_k, ψ'_k, and π'_k with $1 \le k \le N$ are defined as

$$\begin{aligned}
\varphi'_k &:= \text{descendant-or-self::*}[T(O_k) \text{ and parent::*}[\psi'_k]] \\
\psi'_k &:= \begin{cases} \text{child::*}[T(I_k) \text{ and } \pi'_k[\text{last}() > 1]] & \dots\ G_{M+k} \text{ is an } \vee\text{-gate} \\ \text{child::*}[(T(I_k) \text{ and } \pi'_k[\text{last}()=1]) \text{ or } T(W)][\text{last}()=1] & \dots\ G_{M+k} \text{ is an } \wedge\text{-gate} \end{cases} \\
\pi_k &:= \text{ancestor-or-self::*}[(T(G) \text{ and } \varphi'_{k-1}) \text{ or } T(A)].
\end{aligned}$$

Moreover, we set $\varphi'_0 := T(1)$ as in Theorem 3.2.

Of course, this problem reduction can be done in LogSpace. In order to prove the correctness of this reduction, we introduce the following notion: Let ρ and σ be XPath queries that are evaluated on the data tree t or on t', respectively. Then we call ρ and σ *equivalent on a node* x (that occurs both in t and t'), iff the evaluation of boolean(ρ) on t and the evaluation of boolean(σ) on t' for the context-node x yield the same result. Then the following equivalences hold:

Claim.
(1) For all $k \ge 1$, the expressions π_k and $\pi'_k[\text{last}() > 1]$ as well as not(π_k) and $\pi'_k[\text{last}() = 1]$ are equivalent on $v_1, \dots, v_{M+N}, v'_1, \dots, v'_{M+N}$.
(2) For all $k \ge 1$, the expressions ψ_k and ψ'_k are equivalent on $v_0, \dots, v_{M+N}$.
(3) For all $k \ge 0$, the expressions φ_k and φ'_k are equivalent on $v_1, \dots, v_{M+N}$.

We prove properties (1) through (3) simultaneously by induction on k.

Induction start ($k = 0$). Note that only φ_0 and φ'_0 are defined (but not ψ_0, ψ'_0, π_0, and π'_0). Moreover, the equivalence of φ_0 and φ'_0 is obvious by the definition $\varphi_0 = \varphi'_0 = T(1)$.

Induction step. We start with (1), where the central idea of "encoding" negation by iterated predicates becomes clear: By the induction hypothesis, φ_{k-1} and φ'_{k-1} are equivalent on $v_1, \dots, v_{M+N}$, that is, on the nodes for which the condition $T(G)$ is true. For the context-nodes $v_1, \dots, v_{M+N}$, the location path π'_k evaluates

to the same set as π_k plus the node v_0, due to the new disjunct $T(A)$ in the predicate of π'_k. Moreover, by the condition $T(G)$, the node set resulting from π_k never contains the node v_0. The equivalence of π_k with $\pi'_k[\text{last}() > 1]$ and the equivalence of $\text{not}(\pi_k)$ with $\pi'_k[\text{last}() = 1]$ (on $v_1, \ldots, v_{M+N}, v'_1, \ldots, v'_{M+N}$) are thus obvious.

Now consider (2). The case of an $\vee$-gate G_{M+k} is obvious since the equivalence of π_k and $\pi'_k[\text{last}() > 1]$ has just been shown for the nodes v_i and v'_i with $i \geq 1$, which are the only nodes that may possibly fulfill the condition $T(I_k)$. It remains to consider the case of an $\wedge$-gate G_{M+k}. Again, by property (1), $\text{not}(\pi_k)$ and $\pi'_k[\text{last}() = 1]$ are equivalent on all nodes v_i and v'_i with $i \geq 1$. Note that the predicate in the definition of ψ'_k contains the new disjunct $T(W)$. Hence, for any context-node v_i with $i \in \{0, \ldots, N\}$, the node set resulting from the query $\rho'_k :=$ child::*[($T(I_k)$ and π'_k[last()=1]) or $T(W)$] is exactly the node set resulting from $\rho_k :=$ child::*[$T(I_k)$ and not(π_k))] plus the new child w_i (labeled W) of v_i. Thus, the equivalence of $\psi_k = \text{not}(\rho_k)$ with $\psi'_k = \rho'_k[\text{last}() = 1]$ follows immediately.

We finally show (3). In the definition of φ_k and φ'_k, the predicates $[\psi_k]$ and $[\psi'_k]$ are evaluated after a parent-step, that is, for the nodes $v_0, \ldots, v_{M+N}$ only. By property (2), ψ_k and ψ'_k are equivalent on $v_0, \ldots, v_{M+N}$. Hence, ψ_k may indeed be replaced by ψ'_k in the definition of φ_k without altering the meaning of φ_k.

This proves the claim. The correctness of the whole problem reduction follows immediately from the equivalence (3) for $k = N$. □

Note that in the above proof of Theorem 3.16, we only made use of predicate sequences $[e_1] \ldots [e_k]$ whose length k was bounded by 2. We thus have:

COROLLARY 3.17. *The combined complexity of pWF extended by iterated predicates of the form $\chi :: a[e_1][e_2]$ is* PTIME*-complete.*

Despite the negative result of Theorem 3.16, there is of course a direction in which the pWF fragment can be extended without leaving the complexity class LOGCFL. Analogously to Theorem 3.7, we may allow bounded negation:

THEOREM 3.18. *The combined complexity of pWF queries augmented by negation with bounded depth is in* LOGCFL.

PROOF. The proof is by induction on the bound k on the depth of negation. If $k = 0$, then negation does not occur at all and we are back to Theorem 3.14. For the induction step, recall that LOGCFL is closed under complement (cf. Proposition 2.4). Hence, by the induction hypothesis and by Proposition 2.4, there exists a LOGSPACE and polynomial-time NAuxPDA $\bar{\mathcal{A}}_{k-1}$ that decides the following problem: Let Q' be an XPath query from pWF augmented by negation with depth bounded by $k-1$ and let the result type of Q' be boolean. Moreover, let t be a data tree and $\vec{c}'$ be a context. Then $\bar{\mathcal{A}}_{k-1}$ decides that Q' evaluates to *false* over the data tree t for the context $\vec{c}'$. Analogously to the proof of Theorem 3.7, we construct a NAuxPDA $\mathcal{A}_k$ with oracle $\bar{\mathcal{A}}_{k-1}$ s.t. $\mathcal{A}_k$ decides the XPath evaluation problem for pWF augmented by negation with depth bounded by k:

In principle, $\mathcal{A}_k$ carries out the same partial traversal of the query-tree $\mathcal{T}_Q$ as the NAuxPDA $\mathcal{A}$ from the proof of Theorem 3.14. However, when $\mathcal{A}_k$ encounters a subexpression of the form $\text{not}(Q')$, then $\mathcal{A}_k$ proceeds differently from $\mathcal{A}$: W.l.o.g., we may assume that the result of Q' is of type boolean, since otherwise we would

simply replace not(Q') by not(boolean((Q')). Then the node n in the query tree corresponding to the subexpression not(Q') is processed by $\mathcal{A}_k$ as follows: Analogously to the automaton $\mathcal{A}$ in the proof of Theorem 3.14, we first guess cn, cp, cs, and r with $r =$ true. But then, rather than visiting the subtree below n in the query-tree, we immediately carry out the consistency check for the values guessed at n by calling the oracle $\bar{\mathcal{A}}_{k-1}$ with the input $(Q', t, \vec{c}'$ false), i.e. $\bar{\mathcal{A}}_{k-1}$ checks that Q' evaluates to *false* over the data tree t for the context $\vec{c}'$. In other words, $\bar{\mathcal{A}}_{k-1}$ checks that not(Q') evaluates to *true* over the data tree t for the context $\vec{c}'$.

The correctness of the resulting automaton $\mathcal{A}_k$ can be easily verified. Moreover, analogously to the proof of Theorem 3.7, one can show that $\mathcal{A}_k$ works in LogSpace and PTime. □

3.5 Parallelizing XPath

In Section 3.4, we proved the LogCFL-membership for a fragment of XPath that was derived from WF by imposing some restrictions. In fact, we can get a much larger LogCFL fragment of XPath by starting from full XPath and defining the analogous restrictions. The important fact is again that the evaluation can be done without the need to ever compute node sets explicitly and without having to deal with scalars (i.e., values different from node sets) that do not fit into LogSpace. Analogously to pWF, we thus define:

Definition 3.19. *Positive (or parallel) XPath* (pXPath) is obtained by imposing the following restrictions on XPath:

(1) Expressions of the form $\chi :: a[e_1] \dots [e_k]$ with $k \geq 2$ are not allowed.
(2) The following functions may not be used: not, count, sum, string, and number as well as the string functions local-name, namespace-uri, name, string-length, and normalize-space.
(3) Constructs of the form e_1 RelOp e_2 where at least one of the expressions e_i is of type boolean, are forbidden.
(4) The depth of nesting of arithmetic operators and of the concat-function is bounded by some given constant L. Moreover, without loss of generality, the arity of the concat-function is bounded by some constant K. □

The above restrictions extend the ones in Definition 3.12 in the following way: The evaluation of expressions of the form count(e) and sum(e) requires the explicit computation of the node set value of e unless we again introduce loops over dom into the NAuxPDA as in Theorem 3.18. With the functions string and number as well as the string functions listed above, we would have to manipulate items of information in the data tree t whose size is not necessarily logarithmically bounded. Moreover, the functions string and number could also be used to "encode" negation, e.g., number(e) = 0 for a Boolean expression e evaluates to true, iff e evaluates to false. Similarly, constructs of the form e_1 RelOp e_2 where at least one of the expressions e_i is of type boolean are forbidden since they can also be used to "encode" negation, e.g., by an expression of the form $e \neq$ true().

Analogously to Theorem 3.14, we have

Theorem 3.20. *The combined complexity of pXPath is in* LogCFL.

PROOF. The LOGCFL-membership of pXPath can be established by the same kind of NAuxPDA as in the proof of Theorem 3.14. The only adaptation required is an extension of the consistency checks of the guesses cn, cp, cs, r at some node n (pointed to by p) in $\mathcal{T}_Q$ and cn_i, cp_i, cs_i, r_i (at the required child nodes of n) with the XPath expression $expr(p)$. Thus, for each of the additionally allowed XPath constructs, a new line with the corresponding local consistency check has to be added to Table I. Of course, now the result values may also be of type string (in addition to number, boolean, and node-set).

The polynomial time and logarithmic space upper bound on the complexity also holds for the thus extended NAuxPDA. In particular, in pXPath, the arity of the concat-function is bounded by some constant K. Moreover, for all other operators Op in XPath, the maximum arity is bounded by 3 anyway. Hence, the number of child nodes of the nodes in $\mathcal{T}_Q$ (and thus the number of blocks of variables cn_i, cp_i, cs_i, r_i maintained by $\mathcal{A}'$) is no longer bounded by 2 but it is still bounded by some constant. Moreover, it can be easily verified (by a "big" case distinction over all additionally allowed XPath constructs) that the new consistency checks required for pXPath also fit into logarithmic space and polynomial time. □

Finally, analogously to Theorems 3.7 and 3.18, also pXPath can be extended by negation with bounded depth without destroying the LOGCFL-membership.

THEOREM 3.21. *The combined complexity of pXPath augmented by negation with bounded depth is in* LOGCFL.

PROOF. This theorem can be shown by induction on the bound k on the depth of negation – exactly like Theorem 3.18. □

3.6 Query and Data Complexity

So far, we have addressed the combined complexity of various fragments of XPath. While the general problem is PTIME-hard, we have engineered large fragments that can be massively parallelized. We conclude this treatment with a study of the two main other complexity measures, the complexity of queries when either the size of the query or of the data is fixed.

THEOREM 3.22. *PF is* LOGSPACE*-hard under* AC^0*-reductions (with respect to data complexity) if data trees are represented as pointer structures.*

PROOF. Recall the problem ORD, which is LOGSPACE-complete under AC^0 reductions (see Proposition 2.1 [Etessami 1997]). It can be expressed by the PF query /descendant-or-self::v_i/descendant::v_j. The result follows. □

THEOREM 3.23. *XPath is in* LOGSPACE *w.r.t. data complexity.*

PROOF. In [Gottlob et al. 2002], a PTIME bottom-up dynamic programming algorithm for full XPath is given. It is based on the notion of so-called *context-value tables*, relations consisting of tuples $\langle \vec{c}, v \rangle$ containing a context $\vec{c}$ and a corresponding value v for (a subexpression of) the given query, one tuple for each meaningful context. (By the usual XPath semantics definitions, there are no more than n^3 meaningful contexts, where n is the number of nodes in the input tree.)

The algorithm of [Gottlob et al. 2002] simply computes one such context-value table for each node of the query-tree, bottom-up. As shown in [Gottlob et al. 2002],

context-value tables of XPath expressions are guaranteed to be of polynomial size. Given the context-value tables for the direct subexpressions $e_1, \ldots, e_n$, computing the context-value table of expression $Op(e_1, \ldots, e_n)$, where Op is an atomic XPath operation (a node in the query-tree), only requires a very simple computational task. It is beyond the scope of this article to study the operations of XPath in detail, but it is easy to verify by inspection of the complete definition of all XPath operations in [Gottlob et al. 2002] that each of the operations can be carried out in LogSpace, that is, the context-value table of an arbitrary XPath expression can be computed in LogSpace given the context-value tables of its direct subexpressions.

Since we consider data complexity, the query and the number of operations in its query-tree is fixed. The evaluation of a fixed XPath query requires the subsequent evaluation of a fixed number of steps that individually run in LogSpace. In order to be able to compose these LogSpace computations (which me may process according to any topological ordering of the nodes/operations in the parse tree of the query), we assume that each computation receives the input tree and the context-value tables computed earlier on the input tape in a suitable representation and writes its input followed by the newly computed context-value table to the output tape. Our theorem follows from the fact that LogSpace is closed under composition (cf. e.g. [Papadimitriou 1994]). □

In the case of data complexity, which is very low (within LogSpace for all of XPath), the precise model by which data trees are represented matters. The data complexity of Core XPath is lower if data trees are represented as strings rather than as pointer structures:

Theorem 3.24. *If the data trees are given as strings, Core XPath is in* TC^0 *w.r.t. data complexity.*

The proof of this theorem is delayed until the end of Section 4.3, where we will have introduced the necessary background.

Finally, we consider the query complexity of XPath.

Theorem 3.25. *XPath without multiplication or the "concat" operation is in* LogSpace *w.r.t. query complexity.*

Proof. Let Q be the input query and t the (fixed) data tree. All operations in Q have a fixed arity not greater than $K = 3$.

Let us first assume that Q does not contain operations such as + that make strings or numbers grow (logarithmically) with the size of the query. Then, it is known from [Gottlob et al. 2002] that the size of each context-value table is bounded by the *constant* $|t|^4$. To compute the context-value table holding the result of Q on t, we simply have to make a bottom-up traversal of the query-tree of Q, which can be performed in LogSpace. Regarding storage requirements, only a stack bounded by $K \cdot \log |Q|$ context-value tables is needed, which holds context-value tables computed bottom-up but not used yet and waiting to be employed for the computation of context-value tables higher up in the query tree. (Note that this is *not* the depth of the query-tree, which is not necessarily bounded by $O(\log |Q|)$.)

Computing context-value tables bottom-up step by step is important for handling path expressions and negation well. For string- or number-typed expressions e, these

relations do not have to be materialized, but results can be generated and checked top-down when computing a node set-typed context-value table for an expression that contains e as a direct subexpression with only an additional $|t| \cdot \log |Q|$-sized memory window. □

We did not provide a lower bound for the query complexity of XPath, but conjecture a considerable fragment of XPath to be ALogTime-complete with respect to query complexity.

4. XML VALIDATION

In this section we study the complexity of validating an XML document against a typing system. We consider various kinds of typing systems. The first two, called DTDs and *extended* DTDs (which extend DTDs by a specialization mechanism [Papakonstantinou and Vianu 2000]), are motivated by XML standards as they roughly correspond to XML DTDs and XSDs. Both typing systems are subsumed by tree automata. We therefore generalize our study to the main variants of tree automata, namely top-down, bottom-up, deterministic, nondeterministic, and tree walking automata.

We consider two variants of the type checking problem. In the first, the type is fixed and the input consists only of the data tree. In the second, the type is also part of the input. The first case is called the *data complexity* of the validation problem and gives a good approximation of the behavior of the problem when the size of the type is assumed to be unimportant compared to the size of the data. The second is the *combined complexity* and gives a more accurate measure of the difficulty of the problem.

We start by defining in Section 4.1 all the typing systems considered in this article. The complexities involved for validation are low (below LogSpace) therefore the coding of the input tree is crucial. Moreover many proofs use a logical characterization of the corresponding complexity classes. Preliminary results about logics and complexities are presented in Sections 4.2 and Section 4.3. Section 4.4 deals with data complexity while Section 4.5 considers combined complexity.

4.1 DTDs and Tree Automata

We assume familiarity with regular languages, regular expressions, and finite state automata on strings, in their nondeterministic (NSA) and deterministic (DSA) flavors. In this article, NSA and DSA are ϵ-free automata. If A is an NSA, e a regular expression, $\mathcal{L}(A)$ and $\mathcal{L}(e)$ denote the associated regular languages. Context-free grammars (CFG) and context-free languages (CFL) are defined in the normal way (cf. [Rozenberg and Salomaa 1997]). If G is a CFG, $\mathcal{L}(G)$ denotes the associated CFL.

Tree types and DTDs. DTDs and their variants provide a typing mechanism for XML documents. We will use two notions of types for trees. The first one corresponds closely to the DTD formalism proposed for XML documents, and we therefore (by slight abuse) continue to use the same term. A DTD consists of an

extended context-free grammar[9] over alphabet Σ (we make no distinction between terminal and non-terminal symbols at this point). Since regular expressions are closed under union, we can assume without loss of generality, that each DTD has a unique rule $a \rightarrow e$ for each symbol $a \in \Sigma$. We then denote the regular expression e for a by R_a. An XML document satisfies a DTD τ (or is valid w.r.t. τ) if, for each node labeled a, the ordered list of its children forms a word contained in R_a. For example, the labeled tree of Figure 2 is valid w.r.t. the DTD

$$r \rightarrow a^* \quad a \rightarrow bc^* \mid \epsilon \quad b \rightarrow c \mid \epsilon \quad c \rightarrow \epsilon$$

The second type system corresponds closely to XML Schema. It is an extension of DTDs with *specialization* (also called decoupled types) which, intuitively, allows one to define the type of a tag by several "cases" depending on the context. Extended DTDs (EDTD) have been studied in [Papakonstantinou and Vianu 2000] and are equivalent to formalisms proposed in [Beeri and Milo 1999; Cluet et al. 1998]. They are present in a restricted form in XML Schema.

By $SAT()$, we denote the set of trees that satisfy a given DTD or a given automaton, respectively. Then we can formally define EDTDs as follows:

DEFINITION 4.1. An EDTD over Σ is a tuple $\tau = (\Sigma, \Sigma', \tau', \mu)$ where (i) Σ and Σ' are finite alphabets, (ii) τ' is a DTD over Σ', and (iii) μ is a mapping from Σ' to Σ which is canonically extended to trees. A tree document t over Σ satisfies an EDTD τ iff $t \in \mu(SAT(\tau'))$. □

Intuitively, Σ' provides for some a's in Σ a set of specializations of a, namely those $a' \in \Sigma'$ for which $\mu(a') = a$.

When considering only string representations it may be easier to see DTDs and EDTDs as extended context-free grammars with terminal alphabet $\Sigma \cup \bar{\Sigma}$, non-terminal alphabet $V = \Sigma'$, containing rules of the form $S \rightarrow aR_S\bar{a}$, where $S \in V, a \in \Sigma$, R_S is a regular relation over V. DTDs have the restriction that for each $a \in \Sigma$ there is a unique rule involving a. EDTDs do not have this restriction and have one rule per element in Σ'.

If τ is a (E)DTD, $\mathcal{L}(\tau)$ is the language over $\Sigma \cup \bar{\Sigma}$ consisting of the string representations of all tree documents in $SAT(\tau)$, that is, $\mathcal{L}(SAT(\tau))$.

Tree automata. We also consider tree automata (TA). We recall quickly their definitions for unranked trees. The reader is referred to [Brüggemann-Klein et al. 2001] for a more detailed presentation. As usual they are either bottom-up (BU) or top-down (TD) and either deterministic (D) or nondeterministic (N).

A nondeterministic top-down tree automaton (NTDTA) is a tuple (Σ, Q, q_0, δ) where Q is a finite set of states, q_0 is the initial state, and $\delta \subseteq \Sigma \times Q \times Q^*$ is such that $\delta(a, q)$ is a regular language over Q for each $a \in \Sigma$ and each $q \in Q$ (we assume that for $a \in \Sigma$ and each $q \in Q$, $\delta(a, q)$ is given as an NSA). The computation starts at the root of the tree in the initial state and proceeds towards the leaves of the tree by assigning states to the children of a node according to its label, a, the current state, q, and $\delta(a, q)$ (see [Brüggemann-Klein et al. 2001] for more details). The tree

[9]In an extended CFG, the right-hand sides of productions are regular expressions over terminals and non-terminals.

is accepted if, for each leaf n of the tree, the automaton reaches n in a state q such that ϵ is in $\delta(\text{label}(n), q)$. A deterministic TDTA (DTDTA) is an NTDTA such that for each $a \in \Sigma$, each $q \in Q$, and each $n \in \mathbb{N}$ there exists a unique $w \in \delta(a, q)$ of length n.

A nondeterministic bottom-up tree automaton (NBUTA) is a tuple (Σ, Q, F, δ) where Q is a finite set of states, F is a set of accepting states and $\delta \subseteq \Sigma \times Q^* \times Q$ is such that $\{w \in \Sigma^* \mid \delta(a, w, q)\}$ is a regular language over Q for each $a \in \Sigma$ and each $q \in Q$. The computation starts at the leaves by assigning to each leaf of label a a state q such that $\delta(a, \epsilon, q)$ and proceeds towards the root of the tree by assigning states q to a node according to its label, a, the sequence of states of its children, w, and such that $\delta(a, w, q)$ (see [Brüggemann-Klein et al. 2001] for more details). The tree is accepted if the automaton reaches the root in an accepting state. A deterministic BUTA (DBUTA) is an NBUTA such that for each $a \in \Sigma$ and each $w \in Q^*$ there is a unique q such that $\delta(a, w) = q$.

If A is a tree automaton (bottom-up or top-down), then we denote by $\text{SAT}(A)$ the set of trees accepted by A, and by $\mathcal{L}(A)$ the set of string encodings of trees accepted by A.

Fact 4.2 (Folklore, see also [Brüggemann-Klein et al. 2001]). *Let T be a set of labeled unranked trees. T is defined by an EDTD iff T is recognized by an NBUTA iff T is recognized by an NTDTA iff T is recognized by a DBUTA. In this case, T is said to be a regular tree language. On the other hand, DTDs and DTDTA fail to capture all regular languages.*

Finally we consider tree walking automata (TWA). TWA were initially introduced in [Aho and Ullman 1971] (see also [Engelfriet and Hoogeboom 1999; Engelfriet et al. 1999]). A nondeterministic TWA (NTWA) is a tuple $(\Sigma, Q, q_0, F, \delta)$ where Q is a finite set of states, q_0 is the initial state, F is a set of accepting states and $\delta \subseteq \Sigma \times Q \times \{\text{root, leaf, leftchild, rightchild, other}\} \times Q \times \{\text{up, down, left, right, stay}\}$. The computation starts at the root in the initial state and proceeds by walking in the tree as follows: when reaching node n of label a in state q, the automaton checks whether the status of the current node is a root or a leaf or a leftmost (rightmost) child or just a regular inner node, and, changes its state to q' and moves in direction D sucj that $\delta(a, q, \text{status}, q', D)$. A TWA accepts whenever it reaches an accepting states.

A deterministic TWA (DTWA) is an NTWA such that there is a unique q' and D such that $\delta(a, q, \text{status}, q', D)$ for each $a \in \Sigma$ and each $q \in Q$.

It can be shown that NTWA and DTWA recognize only regular tree languages. It was shown recently that NTWA cannot be determinized [Bojanczyk and Colcombet 2004b] and that they do not recognize all regular languages [Bojanczyk and Colcombet 2004a].

4.2 Logic and Complexity

We will use characterizations of some complexity classes in terms of logic. For this, we consider strings as first-order structures whose universe consists of an initial segment of the integers $\{1, \ldots, n\}$, where n is the length of the string and each integer corresponds to a position in the string. First-order logic (FO) over these structures may use a unary predicate $Q_a(i)$ for each letter a in Σ saying that the

position i of the string is an a, a binary predicate $\leq$ denoting the usual order over the integers and a binary predicate $\mathrm{BIT}(i,j)$ which is true iff the i-th bit of j in the binary representation of j is 1. Without loss of generality we can assume that FO also use two constants *min* and *max* denoting the first and last positions in the input string. We also consider FOM, an extension of FO with a majority quantifier M such that $Mx\ \varphi(x)$ is true iff at least half of the positions in the input string validate φ. The following results are well known:

PROPOSITION 4.3 [BARRINGTON ET AL. 1990]. FO = AC^0 *and* FOM = TC^0.

Let $\sharp y\ \varphi(y)$ denote the number of positions y in the input string such that $\varphi(y)$ holds.

We will often make use of the following Lemma which was proved in [Barrington et al. 1990] (Lemma 11.1). It says that addition and counting are definable in FOM. Note that addition can be used to check equality of numbers bigger than the size of the model. For instance one can express in FOM the fact that $x + y = 2 * z$ where x, y, z are variables ranging over the domain of the model. This allows to speak about numbers bigger than the size of the model as expected for a logic characterizing TC^0 over strings.

PROPOSITION 4.4 [BARRINGTON ET AL. 1990]. *The expressions* $x + y = z$ *and* $x = \sharp y\ \varphi(y)$ *are expressible in* FOM.

A function $f : \Sigma^* \to \Gamma^*$ for which there exists a constant k such that $|f(w)| \leq k|w|$ is a FOM reduction iff there exist FOM formulas $\varphi(x)$ and $\varphi_a(x)$ for each symbol $a \in \Gamma$ such that, for each first-order structure of a word w and each $i \in \{1, \ldots, k \cdot |w|\}$, $w \models \varphi(i)$ iff $i = |f(w)|$ and $w \models \varphi_a(i)$ iff the i-th position of $f(w)$ is an a.

The following is straightforward.

PROPOSITION 4.5 [LOHREY 2001]. TC^0 *and* NC^1 *are closed under* FOM*-reductions.*

4.3 Encodings and Complexity

When dealing with trees we often need to consider subtrees. When they are coded as strings this means that we need to consider well balanced substrings. If w is a word coding a tree t and i, j are positions in this word then we denote by $\vartheta(i,j)$ (resp. $\vartheta^*(i,j)$) the fact that the substring of w starting at position i and ending at position j is the coding of a (non-empty) tree (resp. a (possibly empty) sequence of trees).

Checking whether ϑ or ϑ^* holds amounts to check whether the corresponding substring is well-balanced or not. Therefore from Proposition 2.6 and Proposition 4.3 we immediately have:

COROLLARY 4.6. ϑ *and* ϑ^* *are expressible in* FOM.

The following lemma gives the complexities of switching between the two representations for trees.

LEMMA 4.7. *(1) Given a tree* t *as a pointer structure,* $[t]$ *can be computed in* LOGSPACE *from* t. *(2) Given* t *as a string, the pointer encoding can be computed in* TC^0 *from* $[t]$.

PROOF. (1) Given a tree t as a pointer structure, $[t]$ correspond to a depth-first left-first traversal of t. This traversal is easily doable in LOGSPACE from t.

(2) In order to switch from the string representation to the pointer representation we need to compute for each node (i) its first child, (ii) its left and right siblings, and (iii) its parent.

Using Proposition 4.3, we show that this can be done in TC^0 from the string representation of t by exhibiting the corresponding FOM formulas. In the following we identify a node of t with the position of the corresponding opening parenthesis in its word encoding. Recall the definitions of ϑ and ϑ^* as FOM formulas in Corollary 4.6. Let $\mathrm{Open}(i)$ be the FOM formula that checks that the position i is in Σ (as opposed to $\bar{\Sigma}$).

$$\text{Next-Sibling}(i,j) \;\equiv\; \mathrm{Open}(i) \wedge \mathrm{Open}(j) \wedge i \leq j \;\wedge\; \vartheta(i, j-1)$$
$$\text{Previous-Sibling}(i,j) \;\equiv\; \mathrm{Open}(i) \;\wedge\; \mathrm{Open}(j) \;\wedge\; j \leq i \;\wedge\; \vartheta(j, i-1)$$
$$\text{First-Child}(i,j) \;\equiv\; \mathrm{Open}(i) \wedge \mathrm{Open}(j) \wedge j = i+1$$
$$\text{Parent}(i,j) \;\equiv\; \mathrm{Open}(i) \wedge \mathrm{Open}(j) \wedge j \leq i \wedge (i = j+1 \vee \vartheta^*(j+1, i-1))$$ □

REMARK 4.8. The complexity of (1) of Lemma 4.7 is optimal as it was shown in [Cook and McKenzie 1987] that computing the depth-first, left-to-right traversal of a tree is FLOGSPACE-hard under NC^1 reductions, where FLOGSPACE is the functional variant of LOGSPACE. □

We are now ready to prove Theorem 3.24, in which we claimed that if the data trees are given as strings, Core XPath is in TC^0 w.r.t. data complexity.

PROOF OF THEOREM 3.24. Let e be the Core XPath expression and let the data tree t be coded as a string. We show that the query evaluation problem (with the query fixed) can be expressed by a FOM formula $\varphi_e(\mathrm{root}, y)$ such that $t \models \varphi_e(i,j)$ iff the node pointed to by j in t satisfies $e(t)$ when evaluated starting at node i. Then Theorem 3.24 follows from Proposition 4.3 and Lemma 4.7.

The proof is by induction on e. We only give a few examples. If e is $e'/\chi :: a$, then $\varphi_e(x,y)$ is $\exists z\ \varphi_{e'}(x,z) \wedge \mathit{step}_\chi(z,y) \wedge Q_a(y)$, where $\mathit{step}_\chi(x,y)$ is the FOM formula checking that the node y can be reached from y via the χ-axis. We gave examples of such formulas in Lemma 4.7. The others steps can be obtained similarly. If e is $e_1[e_2]$, then $\varphi_e(x,y)$ is $\varphi_{e_1}(x,y) \wedge \exists z \varphi_{e_2}(y,z)$. The remaining cases are left to the reader. □

REMARK 4.9. Full XPath also has the ability of counting the number of nodes that satisfy a given expression and of performing tests on the result. Because FOM also has this capability, extending the above core fragment with such tests, assuming that the arithmetic stays within TC^0 (recall that TC^0 can perform comparisons, addition and multiplication), does not affect the result of Theorem 3.24. □

4.4 Data Complexity

We consider first the most general case and consider membership of a tree in a regular tree language. This will give upper-bounds (depending on the coding chosen for the input tree) for all of the considered typing systems. In the last part of this section we will see that these upper-bounds are actually matched by all of them. Therefore the data complexity of the validation problem does not depend on the typing system used.

Let T be a regular tree language. Consider the following decision problem:

[Val$_T$]	INPUT:	a labeled tree t
	OUTPUT:	true iff $t \in T$.

The complexity of VAL$_T$ depends on the coding used for t. We denote by VAL$_T^{\text{dom}}$ and VAL$_T^{\text{sax}}$, the decision problems associated respectively to the DOM-like and the SAX-like encoding. We start with the case where t is coded as a pointer (tree) structure as it better illustrates the difference between ranked and unranked trees.

4.4.1 *Trees as pointer structures.* In this section, t is given as a pointer structure. We want to check whether $t \in \text{SAT}(A)$ for some unranked tree automaton A. We proceed as follows: first we consider the case where all trees have a fixed rank and then we reduce the unranked case to the ranked case.

Assume first that all trees have a rank bounded by k. Let T be an arbitrary regular (rank k) tree language and let A be a DBUTA recognizing T.

A naive approach would be to perform a depth-first left-first traversal of the input tree storing partial results (the state reached by A on the corresponding subtree) in an auxiliary memory. More precisely assume that the traversal of the tree has just completely processed the subtree rooted at n. In other words we have computed the state q that A reaches at n. Let m be the parent of n. We distinguish three cases. If n is the first and not only child of m then we create a new array H_m of size $O(k)$ in auxiliary memory that will be used to store partial computations of A on the siblings of n. We initiate H_m by storing q in it and proceed to the next sibling of n. If n has still a sibling which has not been processed yet, we update H_m by adding q and proceed with the next sibling. Finally if n is the last child of m then we compute the state of m using q and H_m, free the memory used by H_m and proceed upward with m.

Let's consider the memory used by this algorithm. Assume the traversal is currently at node n in the input tree. Consider the path from the root to n. The memory used at this moment is exactly an array H_s of size $O(k)$ for all nodes s in this path having a sibling which subtree is already completely processed. Thus the total size used in the worst case is $O(d \cdot k)$ where d is the depth of the input tree. As the depth of the input tree can be linear in the size of t. Thus the overall space used is $O(k \cdot |t|)$.

In order to lower the complexity we order the siblings of each node n according to the size (number of nodes) of their corresponding subtrees, taking the largest first and the left-most sibling first in case of equal size (this strategy of considering largest subtrees first is common for trees, see [Lindell 1992] for instance). We denote by $<$ this order. Let NEXT-SIBLING$_<(i)$, FIRST-CHILD$_<(i)$ and LAST-CHILD$_<(i)$ be the functions returning the pointers to the nodes of t corresponding to the next-sibling (first-child, last-child) of the node pointed by i where *first last* and *next* are relative to $<$. Notice that the size of a subtree can be computed in LogSpace and therefore the functions NEXT-SIBLING$_<$, FIRST-CHILD$_<$, LAST-CHILD$_<$ can be computed in LogSpace. We now revisit the algorithm above with a depth-first $<$-first (instead of depth-first left-first) traversal. That is we use the functions NEXT-SIBLING$_<$, FIRST-CHILD$_<$, LAST-CHILD$_<$ in place of the predicates next sibling, first

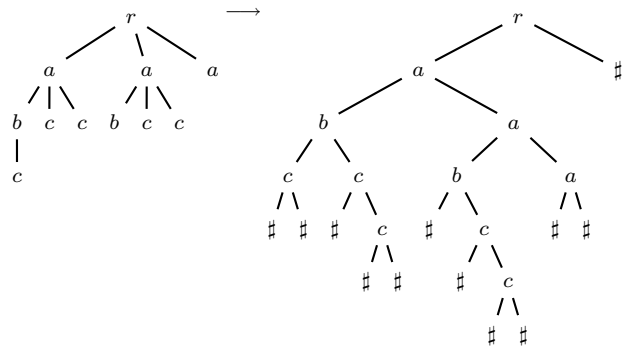

Fig. 10. From unranked to ranked trees.

child and last child. Notice that this traversal remains computable in LogSpace. We can thus show the following lemma:

Lemma 4.10. *If T is a regular ranked tree language then* $\mathrm{VAL}_T^{\mathrm{dom}}$ *is in* LogSpace.

Proof. Let's consider the memory used by the new algorithm described above. Assume the traversal is currently at node n it the input tree. Consider the path from the root to n. As before the memory used at this moment is exactly an array H_s of size $O(k)$ for all nodes s in this path having a sibling which subtree is already completely processed. Let $s_1 \cdots s_i$ be the sequence (from root to n) of such nodes s. The memory space used at node n is $O(k \cdot i)$. For simplicity set s_0 to be the root of t. For $0 \leq j \leq i$ let α_j be the size of the subtree rooted at s_j. Fix j, $1 \leq j \leq i$. We know that s_j has a sibling which has been processed earlier than s_j, therefore, by the choice of $<$ the size of the subtree rooted at s_j is at most half the size of the subtree rooted at the parent of s_j. That is we have $\alpha_j \leq \alpha_{j-1}/2$. By induction we infer that $\alpha_i \leq \alpha_0/2^i$. As $\alpha_i \geq 1$ we must have $i \leq \log |t|$ and the space used at node n is $O(k \cdot \log |t|)$. The overall space complexity is thus $O(k \cdot \log |t|)$ which is LogSpace when k is fixed. □

This algorithm does not generalize to unranked trees because it is linear in the rank of the tree which, in general, could be linear in $|t|$. For that reason, when the trees are unranked, any depth-first evaluation is doomed to fail. However the LogSpace complexity can be obtained by preprocessing the unranked tree in order to encode it as a ranked tree. This reduction is fairly standard in the literature [Suciu 2001; Papakonstantinou and Vianu 2003; Neven 2002]. Let $\sharp$ be a new symbol and $\Sigma^\sharp$ be $\Sigma \cup \{\sharp\}$. The transformation takes an unranked labeled tree t over Σ and returns a binary labeled tree t' over $\Sigma^\sharp$. The transformation is denoted by rank and is defined inductively as follows (see Figure 10): If t is the empty tree then $\mathrm{rank}(t)$ is a single node labeled $\sharp$. Let $\bar{T}$ and $\bar{T}'$ be possibly empty forests and t be a tree whose root is labeled by $a \in \Sigma$ and whose children form the forest $\bar{T}$, then $\mathrm{rank}(t\bar{T}')$ is the binary tree rooted at a node labeled by a whose left child is $\mathrm{rank}(\bar{T})$ and whose right child is $\mathrm{rank}(\bar{T}')$.

Notice that RANK is 1-1 and, importantly, that it preserves regularity.

LEMMA 4.11 (FOLKLORE, SEE ALSO LEMMA 4.24). *Given a regular tree language T, the set $T' = \{t' \mid t' = \text{RANK}(t) \text{ and } t \in T\}$ is a regular tree language.*

Moreover it can be checked that this encoding can be done in LOGSPACE. The algorithm just copies the input nodes, sets the links appropriately (this can be done in LOGSPACE because this concerns only finitely many local nodes), and finally adds as many $\sharp$ nodes as required.

LEMMA 4.12. *Let t be a tree coded by its pointer structure. Then the pointer structure coding* RANK(t) *can be computed in* LOGSPACE *from t.*

Lemma 4.10, Lemma 4.11, Lemma 4.12 and the closure of LOGSPACE under composition immediately entail the main result of this section:

THEOREM 4.13. *For any regular tree language T,* $\text{VAL}_T^{\text{dom}}$ *is in* LOGSPACE.

The complexity of the above theorem is optimal as shown below.

THEOREM 4.14. *There exists T such that* $\text{VAL}_T^{\text{dom}}$ *is complete for* LOGSPACE *under* AC^0 *reductions.*

PROOF. We prove the hardness of $\text{VAL}_T^{\text{dom}}$ by reducing ORD to it, which is LOGSPACE-complete under AC^0 reductions (cf. Proposition 2.1, [Etessami 1997]).

Let T be the regular family of trees consisting of a single branch forming a string in the regular language $a^*ba^*ca^*$.

Let G be a directed graph that is a line and v_i, v_j two nodes of G. G can be seen as a tree t and a pointer structure for t can be constructed from G in the obvious way. The labels of the nodes of t are set such that v_i is labeled b, v_j is labeled c and all other nodes are labeled a. This reduction is First-Order definable and thus computable in AC^0 [Barrington et al. 1990].

It can be checked that ORD is true for G, v_i, v_j iff $t \in T$. This concludes the proof. □

4.4.2 *Trees as strings.* In this section we assume that trees are encoded as a string and study the problem $\text{VAL}_T^{\text{sax}}$. Recall that if t is a tree, $[t]$ denotes its string encoding. The main result of this section is:

THEOREM 4.15. *For all regular T,* $\text{VAL}_T^{\text{sax}}$ *is in* NC^1.

The proof of the theorem follows the strategy of the previous section: it first reduces the unranked case to the ranked one. The reduction is based on the following Lemma:

LEMMA 4.16. *Let t be a tree. Then* $[\text{RANK}(t)]$ *is computable in* TC^0 *from* $[t]$.

PROOF. By Proposition 4.3 it suffices to give a logical FOM formula corresponding to the reduction. We assume the notation used in Corollary 4.6 where $\vartheta(i,j)$, $\vartheta^*(i,j)$ and open(i) are FOM-formulas denoting respectively that the substring from i to j is the coding of a tree, a forest, and that the label at position i is in Σ. We will also use a formula close(i) denoting that the label at position i is in $\bar{\Sigma}$.

In the following we identify a node of the tree by the position of the corresponding opening parenthesis in its word encoding. The following FOM formula thus checks that a node i is a leaf of t (its matching closing parenthesis is the next symbol):

$$\text{Leaf}(i) \;\equiv\; \text{open}(i) \wedge \vartheta(i, i+1)$$

The following FOM formula checks that a node i has no right sibling (its matching closing parenthesis is either the last one of the string or is followed by a closing symbol, ending the subtree of its parent):

$$\text{NoRightSibling}(i) \;\equiv\; \exists j \;(\vartheta(i,j) \wedge (\text{close}(j+1) \vee j = \max))$$

The following FOM formula checks that two nodes i and j are siblings (there is a well-defined forest between i and j):

$$\begin{aligned}\text{Sibling}(i,j) \;\equiv\; & \text{open}(i) \wedge \text{open}(j) \\ & \wedge [(i=j) \vee ((i<j \rightarrow \vartheta^*(i,j-1)) \wedge (j<i \rightarrow \vartheta^*(j,i-1)))]\end{aligned}$$

Finally the following FOM formula checks that j is an ancestor of i (the closing parenthesis of j occurs after i):

$$\text{Ancestor}(i,j) \;\equiv\; \exists u \;(j \leq i \leq u \wedge \vartheta(j,u))$$

Let $t' = \text{RANK}(t)$. By definition, t' contains at most two extra nodes per node of t, thus $|t'| \leq 3 \cdot |t|$. More precisely t' contains the same nodes as t plus nodes labeled $\sharp$, one for each leaf of t and one for each node of t having no right sibling. Because each node of the tree corresponds to 2 characters in its string representation the size z of $[t']$ can be expressed by the following FOM formula (recall Proposition 4.4):

$$\exists x, y \;\; ([z = \max + 2x + 2y] \wedge [x = \sharp i \text{ Leaf}(i)] \wedge [y = \sharp i \text{ NoRightSibling}(i)])$$

Let n be a node labeled by $a \in \Sigma$ in t. Let i be the position in $w' = [t']$ of the opening tag of n and j be its position in $w = [t]$. We are looking for a FOM formula defining i from j and w. i is computed by counting the number of tags of each type occurring before i in w'. The depth-first left-to-right traversal of t and t' will go through the same nodes in the same order (assuming we skip the nodes labeled by $\sharp$ in t'). Thus there are exactly as many opening tags before i in w' as before j in w and the latter can easily be computed from j and w and is denoted by x in the formula below. In order to derive the number of symbols $\sharp$ occurring before i, it suffices to count the number of nodes that are leaves and the number of nodes that do not have a right sibling and occur before j in w. This number is denoted by y in the formula below. Note that each such node accounts for two symbols: one $\sharp$ and one $\bar{\sharp}$. Therefore in order to get i from j and w it remains to compute the number (denoted by z below) of closing tags that must occur before i. From the definition of RANK this number is exactly the number (x) of nodes visited before n minus the number of left siblings of all the ancestors of n. Indeed the subtrees below those nodes are not yet completely computed by RANK. In conclusion, the

following FOM formula $\varphi_a(i)$ gives the positions in t' labeled by a:

$$\begin{aligned}\exists j\ \ & (Q_a(j) \wedge \exists x, y, z \\ & x = \sharp u\ u \leq j \wedge \mathrm{open}(u) \quad \wedge \\ & y = \sharp u\ \exists v\ (u \leq v \leq j \wedge \vartheta(u, v)) \wedge \mathrm{Leaf}(u) + \\ & \qquad \sharp u\ \exists v\ (u \leq v \leq j \wedge \vartheta(u, v)) \wedge \mathrm{NoRightSibling}(u) \quad \wedge \\ & z = x - \sharp u\ [u \leq j \wedge \exists v\ (\mathrm{Sibling}(u, v) \wedge \mathrm{Ancestor}(v, j))] \quad \wedge \\ & i = x + 2y + z)\end{aligned}$$

Next we let i be the position in w' of the closing tag of n and j be its position in w. Notice from the definition of RANK that in w' the closing tag of n occurs when the entire forest of the siblings of n is processed. Let m be the rightmost sibling of n and let k be the position of its closing tag in w. We can proceed as above but considering all nodes before m and its subtree, instead of all the nodes occurring before n, while computing x, y and z: We thus consider all positions prior to k. In the formula below, j' is the position of the opening tag of n.

$$\begin{aligned}\exists j, j', k, x, y, z\ \ & (Q_{\bar{a}}(j) \wedge \vartheta(j', j) \wedge \vartheta^*(j', k) \wedge (\forall k'\ k < k' \rightarrow \neg\vartheta^*(j', k')) \wedge \\ & x = \sharp u\ u \leq k \wedge \mathrm{open}(u) \wedge \\ & y = \sharp u\ \exists v\ (u \leq v \leq j \wedge \vartheta(u, v)) \wedge \mathrm{Leaf}(u) + \\ & \qquad \sharp u\ \exists v\ (u \leq v \leq j \wedge \vartheta(u, v)) \wedge \mathrm{NoRightSibling}(u) \quad \wedge \\ & z = x - \sharp u\ [u \leq k \wedge \exists v\ (\mathrm{Sibling}(u, v) \wedge \mathrm{Ancestor}(v, j'))] \quad \wedge \\ & i = x + 2y + z)\end{aligned}$$

The formulas for $\sharp$ and $\bar{\sharp}$ are obtained similarly.

This concludes the proof of the lemma. □

The ranked case is proved by adapting techniques of [Lohrey 2001] for the coding used here for trees. Intuitively it reduces, in TC^0, the problem to checking membership in a parenthesis CFL which is known to be in NC^1 [Buss 1987].

Proof of Theorem 4.15. Let T be a regular tree language and let T' be its image under RANK (see Lemma 4.11). Let $A = (\Sigma^\sharp, Q, \delta, F)$ be a BUTA recognizing T'. Let $G = (Q, \Sigma^\sharp, R, F)$ be the CFG where R contains $q \rightarrow aq'q''\bar{a}$ iff $\delta(a, q', q'') = q$. It is easy to see that $t \in T$ iff $[\mathrm{RANK}(t)] \in L(G)$ and that G is computed in linear time from A. Therefore by Lemma 4.16 and Proposition 4.5, it suffices to show that $[\mathrm{RANK}(t)] \in L(G)$ can be checked in NC^1. This can be achieved by adapting the proof of Theorem 1 [Lohrey 2001] to our coding of a tree. Let $t' = \mathrm{RANK}(t)$. Let β be the string operation defined inductively on string representations of trees as follows (note that the trees are now binary and complete (each node either have 0 or 2 children)):

- $\beta(\epsilon) = \epsilon$
- $\beta(auv\bar{a}) = \langle a\beta(u)\beta(v)\rangle$ where u and v are well-balanced strings.

Consider now the CFG $G' = (Q, \Sigma^\sharp \cup \{\langle, \rangle\}, R', q_0)$ where R' contains $q \rightarrow \langle aq'q''\rangle$ iff $q \rightarrow aq'q''\bar{a}$ is in R. It is easy to check that $[t] \in L(G)$ iff $\beta([t]) \in L(G')$. We show that β is a FOM reduction.

Let $w = [t]$ for a complete binary tree t. The FOM formula giving the size i of $w' = \beta(w)$ is given by $i = 3.\sharp j\ \mathrm{open}(j)$. The FOM formula giving the positions i of w' containing the symbol $a \in \Sigma^\sharp$ is given by the formula below. In this formula, j

is the position in w of the node whose image by β is i, x computes the number of nodes which have already been completely processed by β before reaching j (notice that each such node is encoded with 3 characters in w' and all of them are before i) and y computes the nodes that are not yet completely processed by β before reaching j, namely the ancestors of j (notice that each such node has exactly 2 characters that occurs before i and one after the closing bracket).

$$\begin{aligned}\varphi_a(i) \equiv \quad & \exists j \;\; Q_a(j) \wedge \\ & x = \sharp u \; \exists v \; \mathrm{open}(u) \wedge u \leq v \leq j \wedge \vartheta(u,v) \wedge \\ & y = \sharp u \; \mathrm{Ancestor}(u,j) \wedge \\ & i = 3x + 2y\end{aligned}$$

The formulas for $\langle$ and $\rangle$ are obtained similarly.

Therefore, by Proposition 4.5, it suffices to show that $w \in \mathcal{L}(G')$ can be checked in NC^1. In general, checking that a word belongs to a CFL is in LogCFL which is believed to be much more powerful than NC^1. But G' is a special CFG, it is a parenthesis CFG as defined in [McNaughton 1967]. It turns out that membership in a parenthesis CFL can be done in NC^1 [Buss 1987]. This concludes the proof of the theorem. □

Theorem 4.17. *There exists T such that $\mathrm{VAL}_T^{\mathrm{sax}}$ is complete for NC^1 under* DLogTime *reductions.*

Proof. The upper-bound of Theorem 4.15 is optimal because there exist regular string languages L such that the problem of checking whether a word w is in L is complete for NC^1 under DLogTime reductions. Such a language can be found in [Barrington et al. 1990]. This regular language can easily be turned into a regular tree language. □

4.4.3 *Other typing systems.* All the typing systems mentioned in the introduction, TA, TWA, DTD, and EDTD, define regular tree languages. Moreover, all of them can encode any string regular language and there exist regular languages for which testing membership is NC^1-complete (see [Vollmer 1999]) when the input is given as a string, and LogSpace-complete (see Theorem 4.14) when the input is given as a pointer structure. From this, Theorem 4.13, and Theorem 4.15, we obtain the following corollary:

Corollary 4.18. *Let τ be a DTD, EDTD, TWA, or a TA. VAL_τ^{dom} is in* LogSpace *and $\mathrm{VAL}_T^{\mathrm{sax}}$ is in NC^1. Both upper bounds are optimal for each of the typing systems.*

4.5 Combined Complexity

In the previous section, the input was simply the encoding of a tree. In this section, we consider the problem of validation when the type is also part of the input. Let C be a class of tree types. For instance, C could be the class of all DTDs, or all EDTDs or all tree automata etc. More formally we will study the following problems:

[TreeM(C)]	INPUT:	a tree t, a type $\tau \in C$
	OUTPUT:	true iff $t \in \mathrm{SAT}(\tau)$

As all types considered in this article are a generalization of string regular languages, we recall the complexities for strings.

[StringM(C)]	INPUT:	a string w, a type $\tau \in C$
	OUTPUT:	true iff $w \in \mathcal{L}(\tau)$

LEMMA 4.19. *[Jiang and Ravikumar 1991; Jones 1975]*

(*1*) *StringM(DSA) is* LOGSPACE*-complete under* AC^0 *reductions.*

(*2*) *StringM(NSA) is* NLOGSPACE*-complete under* AC^0 *reductions*[10]*.*

(*3*) *StringM(Regular-expression) is* NLOGSPACE*-complete under* AC^0 *reductions*[11]*.*

All families of automata that we study are extensions of finite automata over strings thus Lemma 4.19 implies that all the complexities are going to be above LOGSPACE. This Lemma together with Lemma 4.7 implies that the encoding of the input tree is now irrelevant and we will arbitrarily use the string encoding or the pointer encoding depending on what we need. On the other hand the complexity of TreeM(C) will now depend on C.

4.5.1 *DTDs.* Recall that a DTD associates a (unique) regular expression to each symbol of Σ. We have:

THEOREM 4.20. *TreeM(DTD) is* NLOGSPACE*-complete under* AC^0 *reductions.*

PROOF. The upper bound is shown as follows. We will use a pointer p_1 in order to do a depth-first, left-to-right traversal of the input tree. Another pointer p_2 will be used to parse the DTD. An extra pointer p_3 on the tree will be used for local computations. For each node pointed by p_1 the word formed by the labels of the list of its children can be parsed in LOGSPACE using p_3. The pointer p_2 is set to the regular expression of the DTD corresponding to the label pointed by p_1. It is now possible to check that the word pointed by p_3 is indeed in the regular expression pointed by p_2 in NLOGSPACE (this is Lemma 4.19).

The lower bound comes from the fact that the problem is obviously as hard as StringM(Regular-expression) which is NLOGSPACE-complete by Lemma 4.19. □

4.5.2 *TWA.* For TWA, the complexity depends on whether the automaton is deterministic or not. The following extends the result for two-way string automata:

THEOREM 4.21. *TreeM(DTWA) is* LOGSPACE*-complete under* AC^0 *reductions. TreeM(NTWA) is* NLOGSPACE*-complete under* AC^0 *reductions.*

PROOF. We assume the tree is given by its string encoding. The lower bound follows directly from Lemma 4.19. For the upper bound, a Turing machine needs only two pointers, one to record the current position of the head on the input string w, and one to point to the current state. This can be done in LOGSPACE. To get to the next step, it browses deterministically (resp. nondeterministically) the list

[10] [Jones 1975] actually proved completeness under so called Rudimentary-Log reductions which are equivalent to AC^0 as mentioned in [Vollmer 1999]

[11] This can be derived from (2) because there is a LOGSPACE reduction from regular expressions to ϵ-free NSA [Jiang and Ravikumar 1991]

of transitions of the automaton until it gets one corresponding to the current label and state, and then moves the head pointer and the state pointer accordingly.

The remaining difficulty is that a DTWA may loop when it rejects a tree. Recall that such an automata accepts when it reaches an accepting state. It way very well loop when it rejects its inputs. To cope with this situation, together with the simulation of the automata as described above, we also simulates the automata two steps by two steps. That is we also record its configuration every other step. This can also be done in LogSpace as above.

Now if the DTWA reaches an accepting configuration this is detected by the above algorithm. Otherwise it loops. In this case, there is a time t such that the configuration reached by the DTWA is exactly the same as the one reached at time $2t$ and this is detected by the algorithm above.

Altogether this gives the required complexities. □

4.5.3 *Tree Automata.* For tree automata over a ranked alphabet, TreeM was considered in [Lohrey 2001]: TreeM(Ranked-NBUTA) and TreeM(Ranked-NTDTA) are LogCFL-complete, TreeM(Ranked-DBUTA) is in LogDCFL and TreeM(Ranked-DTDTA) is LogSpace-complete. For TreeM(Ranked-NBUTA), the hardness proof of [Lohrey 2001] uses a reduction from membership in a context-free language. The data tree then roughly corresponds to the skeleton of a derivation tree of the input word and, from the CFG, a tree automaton is constructed that checks that a derivation tree of the input word can be superimposed on the data tree. The difficulty is to achieve this within a bounded rank because there is *a priori* no reason for the derivation tree of the input word to have a bounded rank. A combinatorial result of Ruzzo [Ruzzo 1980] is used to show that this is indeed always possible.

We give a very elementary short new proof of this result based on the SAC^1 characterization of LogCFL due to Venkateswaran [Venkateswaran 1991]. Recall that SAC^1 is the class of languages recognizable by LogSpace-uniform families of semi-unbounded ($\wedge$ gates with bounded fan-in but $\vee$ gates with unbounded fan-in) circuits of depth $O(\log n)$ (SAC^1 circuits).

A SAC^1 family of Boolean circuits is in *normal form* (NF), if the following conditions are satisfied: (i) the fan-in of all $\wedge$ gates is 2, (ii) a level can be assigned to the nodes of each circuit such that all inputs are at level 0 and each gate of level i receives all its inputs from nodes of level $i-1$, (iii) the circuits are strictly alternating between $\vee$ gates and $\wedge$ gates (odd-level gates are $\vee$ gates and even-level gates are $\wedge$ gates), and, (iv) each circuit has an odd number of levels, thus the output gates are $\wedge$ gates. Lemma 4.6 of [Gottlob et al. 2001] shows that we can assume without loss of generality, that SAC^1 circuits are in NF. Indeed we have:

Lemma 4.22. *[Gottlob et al. 2001] It is* LogCFL*-complete to test membership in a* SAC^1 *family of circuits in NF.*

Let w be a word of length n. A *proof tree* T_w for the circuit C_n of a SAC^1 family in NF on input word w is a rooted tree t_w together with a labeling function λ_w from nodes N_w of t_w to gates of C_n such that: (i) the root of t_w is labeled with the output gate of C_n, (ii) if $\lambda_w(u) = g$ and g is an $\wedge$ gate then u has exactly 2 children u_1 and u_2 such that $\lambda_w(u_1)$ and $\lambda_w(u_2)$ are the input gates of g, (iii) if

$\lambda_w(u) = g$ and g is an $\vee$ gate then u has a unique child u_1 such that $\lambda_w(u_1)$ is one of the input gates of g, and, (iv) each leaf u of t_w is labeled with an input gate of C_n such that the corresponding bit in w is 1. Intuitively a *proof tree* can be seen as a certificate that the circuit evaluates to 1 on input w.

Notice that for each NF family of SAC^1 circuits and each w of length n the tree t_w of the *proof tree* T_w only depends on n (the labeling function λ_w depends on w and C_n). We note by t_n this tree, denoted as the *skeleton* of the proof trees for inputs of size n in [Gottlob et al. 2001].

Examples of circuits in NF, together with their corresponding proof trees and skeletons can be found in [Gottlob et al. 2001]

The reduction now works as follows: given an input word w of size n and a SAC^1 family C in NF, the data tree will be the skeleton of all *proof trees* for inputs of size n, and the tree automaton will be isomorphic to the circuit C_n. Thus, any accepting run of this automaton corresponds to a function λ_w which is, together with t_n, a *proof tree* of w for C.

LEMMA 4.23 [LOHREY 2001]. *Both TreeM(Ranked-NBUTA) and TreeM(Ranked-NTDTA) are* LOGCFL*-complete.*

PROOF. The upper-bound is immediate as a NTDTA over ranked trees can be seen as a CFL whith states as non-terminals. We consider now the lower bound. We show that membership in a SAC^1 language in NF can be reduced in LOGSPACE to TreeM(Ranked-NTDTA) (TreeM(Ranked-NBUTA) is treated similarly). We then conclude using Lemma 4.22.

Let $(C_n)_{n\in\mathbb{N}}$ be a family of NF circuits in SAC^1 and $w \in \{0,1\}^*$. Let n be the length of w. We have to compute in LOGSPACE a ranked tree t and a ranked NTDTA A such that $t \in \mathrm{SAT}(A)$ iff w is accepted by G_n.

Let t be t_n, the skeleton of the *proof trees* for inputs of size n as defined above, where each node is assigned the same label a. Let $A = (\{a\}, Q, q_0, \delta)$ where $Q = V$, the set of gates of C_n, q_0 is the output gate of C_n. δ is defined as follows. $\delta(a, q) = \{q_1 q_2\}$ if q is an $\wedge$ gate with input gates q_1 and q_2. If q is an $\vee$ gate, then $\delta(a, q) = \{q' \mid q' \text{ is an input gate of } q\}$. If q is an input gate of C_n, then $\delta(a, q) = \{\epsilon\}$ if the corresponding bit of w is 1, and $\delta(a, q) = \emptyset$ otherwise.

It is now straightforward to verify that $t \in \mathrm{SAT}(A)$ iff t, together with the labeling that assigns to each node of t the state of A corresponding to an accepting run, forms a *proof tree* of C_n on input w. It remains to show that the reduction is in LOGSPACE: the fact that t_n can be computed in LOGSPACE from n is shown in [Gottlob et al. 2001], and the LOGSPACE computation of A is immediate from the LOGSPACE uniformity of SAC^1 circuits. □

The general result for unranked trees is obtained by reduction to the ranked case. Recall Lemma 4.11. Given a regular tree language T, the tree language RANK(T) is regular. In what follows, we overload the notation of RANK, and denote by RANK(A) a tree automaton recognizing the regular tree language RANK(SAT(A)). The following shows that the reduction can be computed in LOGSPACE.

LEMMA 4.24. *Given an NTDTA or NBUTA A,* RANK*(A) can be computed in* LOGSPACE*.*

PROOF. We first prove the NTDTA case by giving the LOGSPACE construction of $A' = \text{RANK}(A)$. Let $A = (\Sigma, Q, q_0, \delta)$ be an NTDTA. For each letter $a \in \Sigma$ and each state $q \in Q$ let $A^{a,q} = (Q, \Delta^{a,q}, p_0^{a,q}, F^{a,q}, \delta^{a,q})$ be a nondeterministic automaton recognizing the regular string language $\delta(a, q)$. The alphabet is Q and the set of states $\Delta^{a,q}$. Let Δ be $\bigcup_{a,q} \Delta^{a,q}$.

To check whether a tree is accepted by A, A' has to verify that both *vertical* and *horizontal* transitions of A are satisfied. The vertical ones correspond to transition of A, the horizontal ones to the regular expressions occurring in the transition table of A. This is manageable by an automata because in the coding, a vertical transition is a move to the left child while an horizontal transition is a move to the right child. The automaton A' just have to propagate to its left and right child the corresponding transition.

In order to do so, a state of A' will be a quadruple of $\Sigma \times Q \times \Delta \times Q$. The meaning of a state $\langle a, q, p, q' \rangle$ is that the automaton is currently simulating $A^{a,q}$, has reached p in this simulation and is guessing that the next letter $A^{a,q}$ will read is q'. There is also a state $q_\sharp$ which is expected to read a node labeled $\sharp$ and an initial state q'_0.
Let $A' = (\Sigma^\sharp, Q', q'_0, \delta')$ be the NTDTA defined by :

- $Q' = \{\langle a, q, p, q' \rangle \mid a \in \Sigma, q \in Q, q' \in Q, p \in \Delta\} \cup \{q_\sharp, q'_0\}$.
- $\delta'(\sharp, q_\sharp) = \epsilon$.
  $q_\sharp$ accepts iff it is on a leaf labeled by $\sharp$.
- $\delta'(a, q'_0) = \{\langle a, q_0, p_0^{a,q_0}, q \rangle q_\sharp \mid q \in Q\}$.
  If the root is labeled a, A' starts a simulation of A^{a,q_0} on the left child of the root and checks that the right child of the root is labeled by $\sharp$.
- $\langle a, q', p_0^{a,q'}, q'' \rangle \langle b, q, p', q''' \rangle \in \delta'(a, \langle b, q, p, q' \rangle)$ if $p' \in \delta^{b,q}(q', p)$.
  On a node labeled a, A' can start a simulation of $A^{a,q'}$ on its left son and continue the simulation of $A^{b,q}$ on its right son.
- $\langle a, q', p_0^{a,q'}, q'' \rangle q_\sharp \in \delta'(a, \langle b, q, p, q' \rangle)$ if $p \in F^{b,q}$.
  On a node labeled a, if the simulation of $A^{b,q}$ reached an accepting state, A' can also start a simulation of $A^{a,q'}$ on its left son and check that its right son is labeled by $\sharp$.
- $q_\sharp \langle b, q, p', q'' \rangle \in \delta'(a, \langle b, q, p, q' \rangle)$ if $p' \in \delta^{b,q}(q', p)$ and $p_0^{a,q'} \in F^{a,q'}$.
  On a node labeled a, if a can be a leaf (that is ϵ is in $A^{a,q'}$ or equivalently $p_0^{a,q'} \in F^{a,q'}$), then A' can check that its left son is labeled by $\sharp$ and continue the simulation of $A^{b,q}$ on its right son.
- $q_\sharp q_\sharp \in \delta'(a, \langle b, q, p, q' \rangle)$ if $p \in F^{b,q}$ and $p_0^{a,q'} \in F^{a,q'}$.
  On a node labeled a, if a can be a leaf and the simulation of $A^{b,q}$ reached an accepting state, then A' checks that both sons are labeled by $\sharp$.

It is straightforward to check that the above computation is correct (A' is indeed $\text{RANK}(A)$) and that it can be done within LOGSPACE: 4 pointers are needed in order to construct the states of A', and 12 pointers suffice to compute the transition function δ'.

The proof for NBUTA is immediate as it is the dual of an NTDTA. □

THEOREM 4.25. *TreeM(NBUTA) and TreeM(NTDTA) are* LOGCFL*-complete under* LOGSPACE *reductions.*

PROOF. The theorem follows immediately from Lemma 4.23 and Lemma 4.24. □

4.5.4 *Other typing systems.* Given an EDTD, the NTDTA equivalent to it is immediately obtained in LOGSPACE, and vice-versa. Thus Theorem 4.25 implies:

COROLLARY 4.26. *TreeM(EDTD) is* LOGCFL*-complete under* LOGSPACE *reductions.*

For deterministic tree automata, the situation is not as simple. It is easy to see that the RANK transformation does not preserve determinism. For TreeM(DBUTA) in the ranked case [Lohrey 2001] gives an algorithm which works in LOGDCFL. By [Sudborough 1978], LOGDCFL coincides with the class of languages recognizable by deterministic auxiliary pushdown automata in logarithmic space and polynomial time. The algorithm used in [Lohrey 2001] is similar to the one given in Section 4.4.1 where the order $<$ is the depth-first, left-to-right traversal of the tree. The difference is that the set Q of states of the automaton is now part of the input and therefore Γ is an alphabet whose letters are of size $\log|Q|$. This is why an auxiliary stack of logarithmic size is required. In order to compute the state of a node from the states of its k children it uses an intermediate storage (called H in the algorithm presented in Section 4.4.1) of size $k \cdot \log|Q|$ and this cannot be extended to unranked trees as k can be arbitrarily large. When the tree is unranked we can proceed as follows. The auxiliary structure H containing k states, one for each son, is incrementally stored in the stack as a sequence of k elements. The k children of n are then evaluated successively and the state of the automata reached by the automata on each child is pushed on the stack. When all k children of a node n labeled a have been visited, the stack contains the sequence of states H. The state of n is now computed by guessing the (unique) state q for n and then deterministically checking that the guess is correct by simulating the automaton $\delta(a, q)$ on the word of size k located in the top of the stack. By definition of DBUTA there is a unique accepting run therefore the complexity is LOGUCFL. The precise complexity remains open.

For TreeM(DTDTA) the situation is also slightly more complex for unranked trees than for ranked ones. Indeed it is LOGSPACE-complete under AC^0 reductions for the ranked case. The algorithm given in [Lohrey 2001] works as follows. It scans through all the leaves of the tree and for each of them checks that the path going from the root to that leaf does verify the transitions given by the automaton. Because the automaton is deterministic, this is sufficient for validation. The navigation in the tree is indeed in LOGSPACE, it remains to compute, given a node n of label a and a state q, the state of the i-th child of n. This can easily be done in LOGSPACE in the ranked case by scanning $\delta(a, q)$. When the input tree is unranked, one has to extract from the automaton of $\delta(a, q)$, the i-th character of the (unique) word of length k accepted by this automaton, where k is the number of children of n. This can be done in ULOGSPACE by guessing, letter by letter, the unique word of length k of $\delta(a, q)$ while simulating $\delta(a, q)$. Again the precise complexity remains open.

THEOREM 4.27. *(1) TreeM(DBUTA) is in* LOGUCFL.
(2) TreeM(DTDTA) is in ULOGSPACE.

5. CONCLUSION

In this article, we have carried out a thorough complexity analysis of the XPath evaluation problem as well as the XML validation problem.

We considered the current standard, XPath 1.0 [World Wide Web Consortium 1999]. The now proposed XPath 2.0 language includes most of XQuery and is Turing-complete; however, most real-world path queries will remain expressible in XPath 1.0, which is a strict fragment of XPath 2.0.

Our analysis of the combined complexity of *XPath evaluation* revealed that negation in XPath is the main obstacle to constructing parallel algorithms. In contrast, for positive Core XPath (i.e., Core XPath without negation), we were able to establish LOGCFL-completeness and, thus, parallelizability. Moreover, we defined various extensions of positive Core XPath (like pWF, pXPath, pWF plus negation with bounded depth) for which the combined complexity remains unchanged. However, as soon as we add XPath constructs that allow one to "encode" negation (like iterated predicates in pWF), then we are back to PTIME-completeness.

Besides, we also investigated further restrictions of Core XPath, which led to even lower complexity classes (i.e., NLOGSPACE in case of PF and LOGSPACE in case of $\mathrm{PF}_{\downarrow}$ and Core XPath[1].

Finally, we studied the query and data complexity of XPath. In both cases, we came up with results showing that these complexities are in the low complexity class LOGSPACE (in case of query complexity, we had to exclude the XPath constructs "concat" and multiplication, which cause the result value to grow linearly with respect to the query).

As far as the *XML validation problem* is concerned, we have seen that checking the membership of an unranked labeled tree in a regular tree language is not more difficult than checking the membership of a string in a regular string language. Thus the data complexity of checking whether an XML document conforms to a type is independent of the typing system as soon as this typing system is an extension of regular string languages and a restriction of regular tree languages. On the other hand, we have seen that the coding used for the data tree is crucial and that two of the most heavily used models of XML data give two different complexity bounds: NC^1 vs. LOGSPACE.

The combined complexity of validation does not depend on the coding of the input data tree but on the type language. It was shown that, depending on the typing language, complexities vary from LOGSPACE-complete to LOGCFL-complete.

Acknowledgments

We thank Wim Martens for suggestions that helped to improve this article.

REFERENCES

ABITEBOUL, S. 2001. Semistructured data: from practice to theory. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01).*

AHO, A. V. AND ULLMAN, J. D. 1971. Translations on a context-free grammar. *Information and Control 19*, 439–475.

Allender, E. 2001. The division breakthroughs. *The Computational Complexity Column, EATCS Bulletin 74*, 61–77.

Barrington, D. A. M. and Corbett, J. 1989. On the relative complexity of some languages in NC. *Information Processing Letters 32*, 251–256.

Barrington, D. A. M., Immerman, N., and Straubing, H. 1990. On Uniformity within NC. *Journal of Computer and System Sciences 41,* 3, 274–306.

Beeri, C. and Milo, T. 1999. Schemas for integration and translation of structured and semi-structured data. In *Proc. 7th International Conference on Database Theory (ICDT'99)*. Lecture Notes in Computer Science, vol. 1540. Springer, 296–313.

Bojanczyk, M. and Colcombet, T. 2004a. personnal comunication.

Bojanczyk, M. and Colcombet, T. 2004b. TWA cannot be determinized. In *31st International Colloquium on Automata, Languages and Programming (ICALP)*.

Borodin, A., Cook, S. A., Dymond, P. W., Ruzzo, W. L., and Tompa, M. 1989. "Two Applications of Inductive Counting for Complementation Problems". *SIAM Journal of Computing 18*, 559–578.

Brüggemann-Klein, A., Murata, M., and Wood, D. 2001. "Regular Tree and Regular Hedge Languages over Non-ranked Alphabets: Version 1, April 3, 2001". Tech. Rep. HKUST-TCSC-2001-05, Hong Kong University of Science and Technology, Hong Kong SAR, China. Available at ftp://ftp11.informatik.tu-muenchen.de/pub/misc/caterpillars/.

Buss, S. R. 1987. The Boolean formula value problem. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC'87)*.

Chamberlin, D., Robie, J., and Florescu, D. 2000. "Quilt: An XML Query Language for Heterogeneous Data Sources". In *Third International Workshop on the Web and Databases (WebDB)*. LNCS 1997. Springer, 1–25.

Cluet, S., Delobel, C., Simeon, J., and Smaga, K. 1998. "Your Mediators need Data Conversion!". In *Proc. 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*. 177–188.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. 1999. Tree Automata Techniques and Applications. Available at http://www.grappa.univ-lille3.fr/tata.

Cook, S. A. and McKenzie, P. 1987. "Problems Complete for Deterministic Logarithmic Space". *Journal of Algorithms 8*, 385–394.

Engelfriet, J., Hoogeboom, H., and van Best, J.-P. 1999. Trips on trees. *Acta Cybernetica 14*, 51–64.

Engelfriet, J. and Hoogeboom, H. J. 1999. Tree-walking pebble automata. In *Jewels are forever, contributions to Theoretical Computer Science in honor of Arto Salomaa*, J. Karhumki, H. Maurer, G. Paun, and G.Rozenberg, Eds. Springer-Verlag, 72–83.

Etessami, K. 1997. "Counting Quantifiers, Successor Relations, and Logarithmic Space". *Journal of Computer and System Sciences 54,* 3, 400–411.

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

Gottlob, G., Koch, C., and Pichler, R. 2003a. "The Complexity of XPath Query Evaluation". In *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'03)*. 179–190.

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

Gottlob, G., Leone, N., and Scarcello, F. 2001. The Complexity of Acyclic Conjunctive Queries. *Journal of the ACM 43,* 3, 431–498.

Greenlaw, R., Hoover, H. J., and Ruzzo, W. L. 1995. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press.

Jiang, T. and Ravikumar, B. 1991. "A Note on the Space Complexity of some Decision Problems for Finite Automata". *Information Processing Letters 40*, 25–31.

Johnson, D. S. 1990. "A Catalog of Complexity Classes". In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. 1. Elsevier Science Publishers B.V., Chapter 2, 67–161.

Jones, N. D. 1975. Space-Bounded Reducibility among Combinatorial Problems. *Journal of Computer and System Sciences 11*, 68–85.

Lindell, S. 1992. A Logspace Algorithm for Tree Canonization. In *Proc. 24th Annual ACM Symposium on Theory of Computing (STOC'92)*.

Lohrey, M. 2001. On the Parallel Complexity of Tree Automata. In *Proc. Rewriting Techniques and Applications (RTA'01)*. Lecture Notes in Computer Science, vol. 2051. Springer, 201–215.

Lyons, R. 2001. "Turing Machine Markup Language". http://www.unidex.com/turing/.

McNaughton, R. 1967. Parenthesis Grammars. *Journal of Computer and System Sciences 14,* 3, 490–500.

Neven, F. 2002. Automata, Logic, and XML. In *Proc. Computer Science Logic, 16th International Workshop (CSL'02)*. Lecture Notes in Computer Science, vol. 2471. Springer, 2–26.

Olteanu, D., Meuss, H., Furche, T., and Bry, F. 2002. XPath: Looking Forward. In *Proc. Workshop on XML-Based Data Management (XMLDM'02)*. A full version, Technical Report PMS-FB-2001-17, is available at `http://www.pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2002-4.pdf`.

Papadimitriou, C. H. 1994. *Computational Complexity*. Addison-Wesley.

Papakonstantinou, Y. and Vianu, V. 2000. DTD inference for views of XML data. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'00)*. 35–46.

Papakonstantinou, Y. and Vianu, V. 2003. Incremental Validation of XML Documents. In *Proc. 9th International Conference on Database Theory (ICDT'03)*. Lecture Notes in Computer Science, vol. 2572. Springer, 47–63.

Rozenberg, G. and Salomaa, A., Eds. 1997. *Handbook of Formal Languages*. Springer.

Ruzzo, W. 1980. "Tree-size Bounded Alternation". *Journal of Computer and System Sciences 21*, 218–235.

SAX Project Collaboration. 2004. Simple API for XML. Available at http://www.saxproject.org/.

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

Segoufin, L. and Vianu, V. 2002. Validating streaming XML documents. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'02)*. 53–64.

Suciu, D. 2001. Typechecking for Semistructured Data. In *Proc. Database Programming Languages, 8th International Workshop (DBPL'01)*. Lecture Notes in Computer Science, vol. 2397. Springer, 1–20.

Sudborough, I. 1977. "Time and Tape Bounded Auxiliary Pushdown Automata". In *Proc. Mathematical Foundations of Computer Science, 6th Symposium (MFCS'77)*. Springer Verlag, LNCS 53, 493–503.

Sudborough, I. 1978. "On the Tape Complexity of Deterministic Context-free Languages". *Journal of the ACM 25,* 3, 405–414.

Vardi, M. Y. 1982. "The Complexity of Relational Query Languages". In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC'82)*. 137–146.

Venkateswaran, H. 1991. "Properties that Characterize LOGCFL". *Journal of Computer and System Sciences 43*, 380–404.

Vollmer, H. 1999. *Introduction to circuit complexity*. Springer.

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

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

World Wide Web Consortium. 2004. Document Object Model. Available at http://www.w3c.org/dom, level 1 specification available at http://www.w3.org/TR/REC-DOM-Level-1/.